ELEC-H410
REAL-TIME SYSTEMS

Chapter 4

Clock-driven Scheduling

Clock-driven Scheduling

CONTENTS

- **hypothesis**
- static scheduling
- dynamic scheduling
- conclusions

Clock-driven scheduling

hypothesis

- ***time-driven*** process
  - most and main tasks are periodic
    - number n fixed, known at design time
    - period $p_{i(min)}$ known for each i
    - no priority, independent tasks
  - limited number of aperiodic events
    - activated by IRQ
    - buffered in a queue
  - example : control of a positioning robot
- versus **event-driven** systems
  - random activation of events
  - concurrency and priority management
  - ex: video games, active suspension

Clock-driven Scheduling

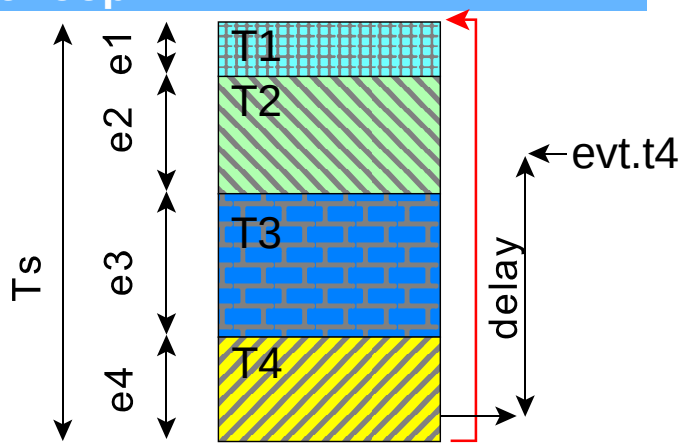
CONTENTS

- hypothesis
- **static scheduling**
  - **superloops**
  - **superloops synchronized by interrupts**
  - **multi-cyclic loops**
- dynamic scheduling
- overflow management
- conclusions

Clock-driven static scheduling

superloop

- NSC : $\Sigma \max(e_i) < T_{s\_min}$
- advantages
  - simplicity, robustness
  - µP well-exploited
  - *non-reentrant* code is allowed
- drawbacks
  - non deterministic (*jitter* on e_i)
    - T_s variable
    - risk of missing a periodic input
    - risk of jitter on the periodic outputs
    - unique frequency for all tasks (useless for slow tasks)
  - aperiodic events
    - detected by polling => min duration
    - detected by masked IRQ and polling of the flags
    - max response time = T_s
    - no priority



The simplest synchronous system for periodic tasks is the *superloop*. All the tasks are executed sequentially and the last one is terminated by a branch instruction to the beginning of the first one, in an infinite loop.
This figure shows us the cyclic execution of 4 tasks.
A necessary and sufficient condition (NSC) of schedulability is that the sum of execution times is shorter than the least sampling period T_{s_min} imposed by the process (i.e.typically 10 times less than the smallest time-constant, as we have seen in the previous)

$$\Sigma \max(e_i) < T_{s\_min}$$

It is an extremely **simple** system, very **robust** and **easy to debug** because, the tasks being successive:
- they do not share any resource
- they do not call each other mutually and hence can be written in non-reentrant code, which is more compact and more efficient

Superloops have also some disadvantages:
- the **jitter** on the execution times of the tasks creates a global jitter on the sampling period, with
  - a risk of missing strictly periodic entries
  - difficulties to produce strictly periodic outputs, if required
    - the sampling period is the same for all the tasks, which wastes the resources of the processor for the slow tasks, which are oversampled (it can be important if consumption is an issue)
- handling of the **aperiodic events** is not optimal, indeed:
  - an event can only be detected by the task which takes it into account; this detection is done by "polling" the corresponding input; if the event is a short pulse, it should be caught by masked interrupt, with polling of the interrupted flag; attention: some specifications prohibit interrupts
  - the **latency time** between the event and the moment when it is taken into account by the task is equal to the sampling period Ts in the worst case. It is deterministic, but not necessarily within the deadline
  - there is **no priority management** for the aperiodic events

Clock-driven static scheduling

deterministic supeloop (rare)

- principle
  - e_i always same duration for any input
  - padding instructions (NOP) to adjust T_s
- extremely tiresome

```
IF condition
THEN
   action 1 (143 cycles)
ELSE
   repeat 143
      NOP
ENDIF
```

```
CASE var
   var=case1
      action 1 (14 cycles)
      repeat 4
         NOP
   var=case2
      action 2 (18 cycles)
   otherwise
      repeat 18
         NOP
ENDCASE
```

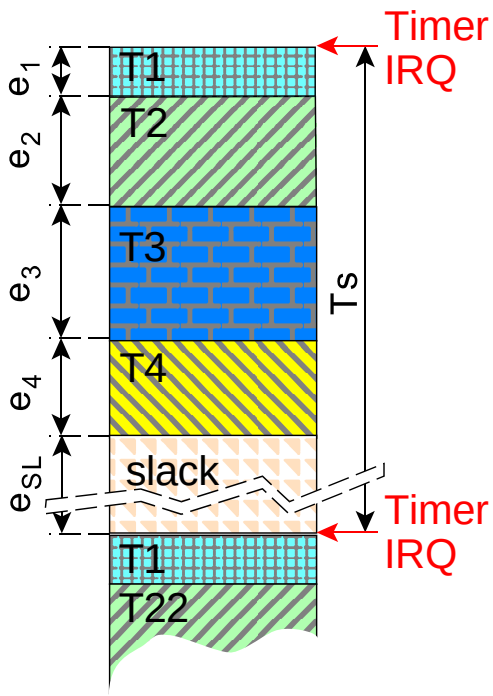

To avoid the jitter some specifications (space applications for example) can ask for an equalization of all the branches of all the conditional instructions (IF, CASE?..); this can be achieved by "padding" the shorter branches with NOP (No OPeration) instructions.
The sampling period is adjusted by adding a "dummy" task.

This is very rare because the determinism is obtained at the price of an extremely tiresome work for the programmer.

Clock-driven static scheduling

deterministic loop: Ts fixed by IRQ

- NSC : $\Sigma e_i < T_s$
- improvement
  - T_s constant
  - no more padding
  - $\exists$ *slack* e_{SL}
    - sleep = low-power mode
    - back = background task with no priority
    - aperiodic tasks
      - activated by other IRQ
      - queue



All current microcontrollers include timers which can generate periodic interrupts by counting n periods of the CPU clock.

It is the best way to fix the sampling period without having to resort to padding.

The ISR of the timer
- either performs all the tasks in sequence
- or sets a boolean flag that will trigger the execution of all tasks by the main program

The necessary and sufficient condition of schedulability remains obviously that the total duration of the periodic tasks is shorter than the sampling period.

The remaining CPU time, in which no useful action is performed is called the ***slack***; it can be used to:

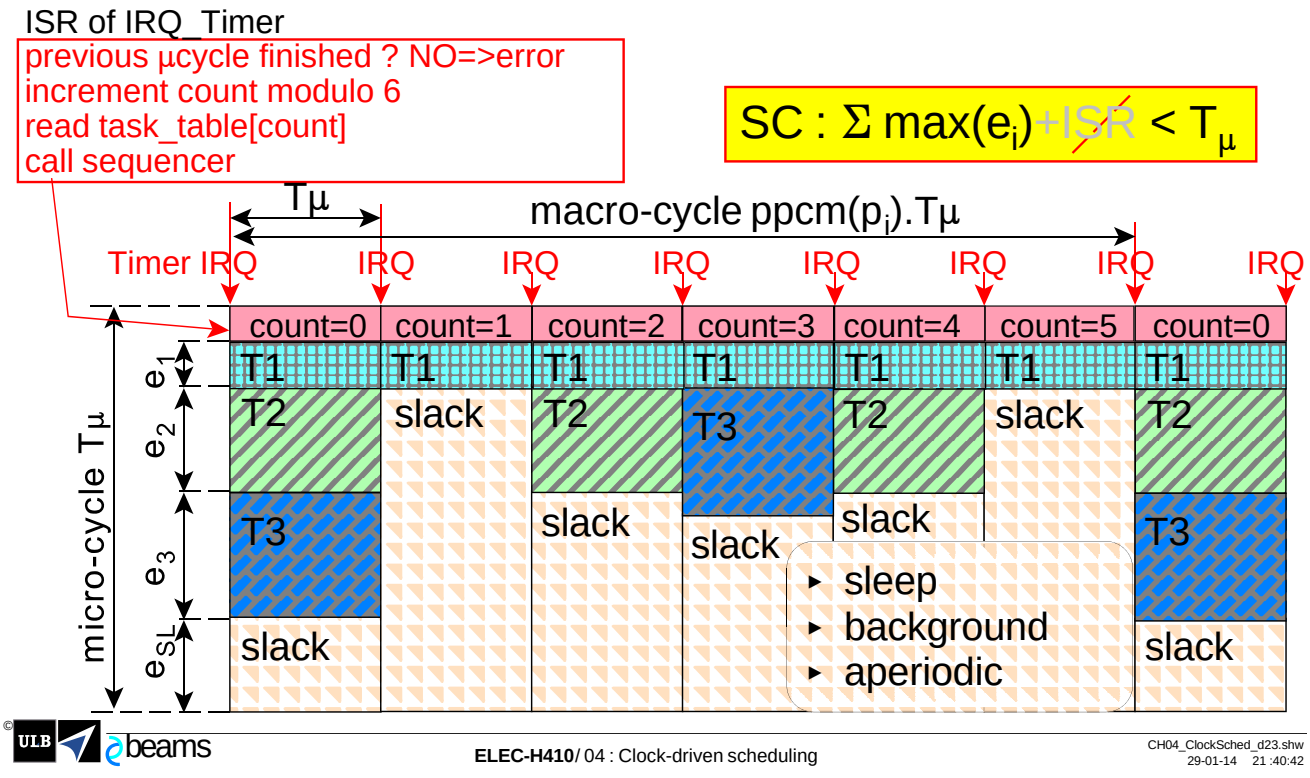
- put the processor in sleep mode to reduce consumption, which can be very interesting if the slack represents a significant part of the period (which is often the case is battery-powered embedded systems like fire alarms or water supply meters, for which the sampling frequency is very low.

- to execute one or more **background tasks** having neither priority, nor a deadline

- to handle **aperiodic** tasks released by other interrupt requests, which are generally placed in a queue

Clock-driven static scheduling

multicyclic scheduling



Scheduling must frequently manage tasks of very different periods, for example 1ms for the current control, 20ms for the speed control and 2s for the temperature control.

To limit the wasting of CPU time caused by the single sampling rate of the previous slide, we can use a mufti cyclic scheduling controlled by timer interrupts:
- the period of interruption TO is equal to the sampling period of the fastest task and is called **micro-cycle** (or minor cycle or frame),
- the periods of the other tasks are rounded to a multiple of T_μ : $p_i = n_i T_\mu$ where n_i is an integer

On this figure, we see a horizontal time axis over which the periodic IRQs of the timer are indicated.
At each of those instants, a column represents a new vertical time axis, which is the course of the micro-cycle.

There is a **macro-cycle** (or hyper-period or major cycles), at the end of which the same pattern of execution is reproduced

$H = LCM(n_i).T_\mu = M.T_\mu$ in this example $H = LCM(1,2,3).T_\mu = 6T_\mu$

Scheduling is carried out by the ISR of the timer which will:
- take emergency actions if all the jobs of the previous micro-cycle are not finished (see further §)
- increment a software counter modulo M
- consult a table, indexed on this counter, which contains the task list of the new micro-cycle
- call the tasks sequentially, on basis of this table

If we neglect the scheduling time, the schedulability condition remains the same $\Sigma e_i < T_\mu$, because there will always be a micro-cycle during which a job of each task will be executed.

In the majority of the micro-cycle, there will remain idle period (slack) to put the processor in sleep mode, to execute background tasks, or to handle aperiodic requests.

We thus have here a typical example of a table-based static off-line scheduling.

Clock-driven Scheduling

CONTENTS

- hypothesis
- simple static off-line solutions
- **dynamic scheduling**
  - aperiodic events (masked IRQ+polling)
  - **slack stealing**
  - **aperiodic events (IRQ)**
- overflow management
- conclusions

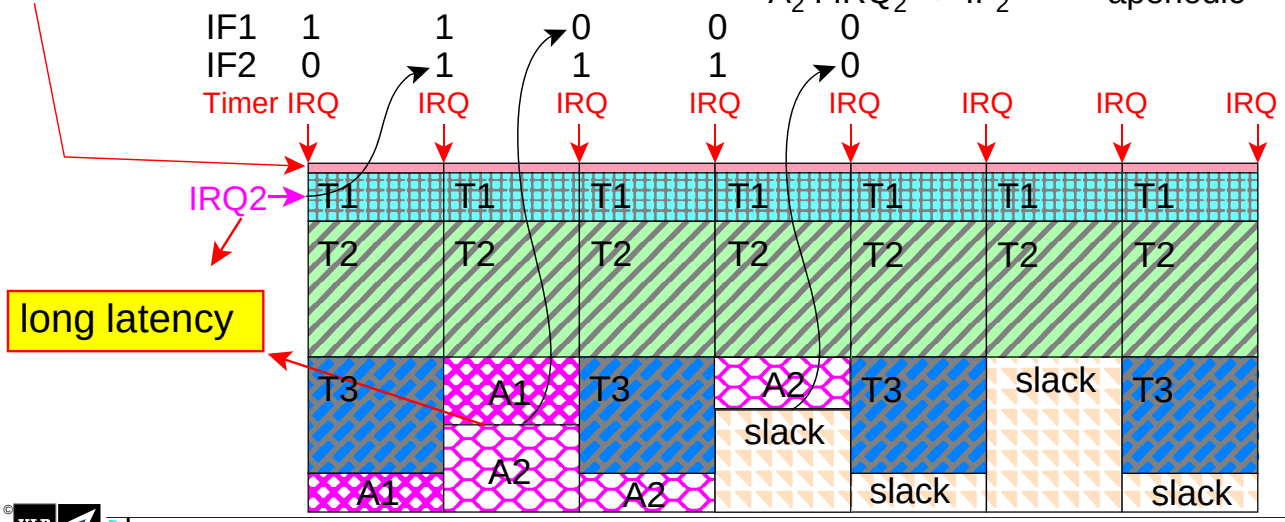
Clock-driven dynamic scheduling

periodic task + aperiodic tasks during the slack

ISR of Timer_IRQ

previous µcycle finished? NO=>error
increment *count* modulo H
read task_table[count]
read IF1 et IF2 and manage priorities
call sequencer

T_1 : p_1=1 periodic
T_2 : p_2=1 periodic
T_3 : p_3=2 periodic
A_1 : IRQ_1 => IF_1 aperiodic
A_2 : IRQ_2 => IF_2 aperiodic



Let us see now how a cyclic scheduler can become **dynamic**, to take into account the **aperiodic tasks** during the idle (slack) periods.
In this example, our system will include 3 periodic tasks T1, T2 and T3, and 2 aperiodic tasks A1 and A2, activated respectively by the interrupted requests IRQ1 and IRQ2.
To give the priority to cyclic operation, the interrupt coming from the timer is the only one which is not masked; the occurrences of IRQ1 and IRQ2 are detected at the beginning of each micro-cycle by polling the interrupt flags IF1 and IF2.

Suppose that, during the micro-cycle preceding the beginning of the figure, IRQ1 occurred, IF1 is thus active. At the beginning of the first micro-cycle visible on this figure, the scheduler consults his table, examines the state of IF1 and IF2, and deduces it has to launch the execution of the tasks T1, T2, T3 and A1.

During the execution of the first micro-cycle, interruption IRQ2 occurs and sets IF2.
As soon as the periodic tasks are finished, the slack is affected to the A1 task. A1 is unfortunately not finished at the end of the micro-cycle and is preempted by the interruption of the timer.
The scheduler takes back the control; IF_1 and IF_2 being set, a priority algorithm must decide in which order to execute A1 and A2 ; several policies are possible:

- FIFO *First-In-First-Out (*what happens in this figure; A_1 will thus be executed before A_2)
- any other priority policy, dynamics or not which could give the priority to A2

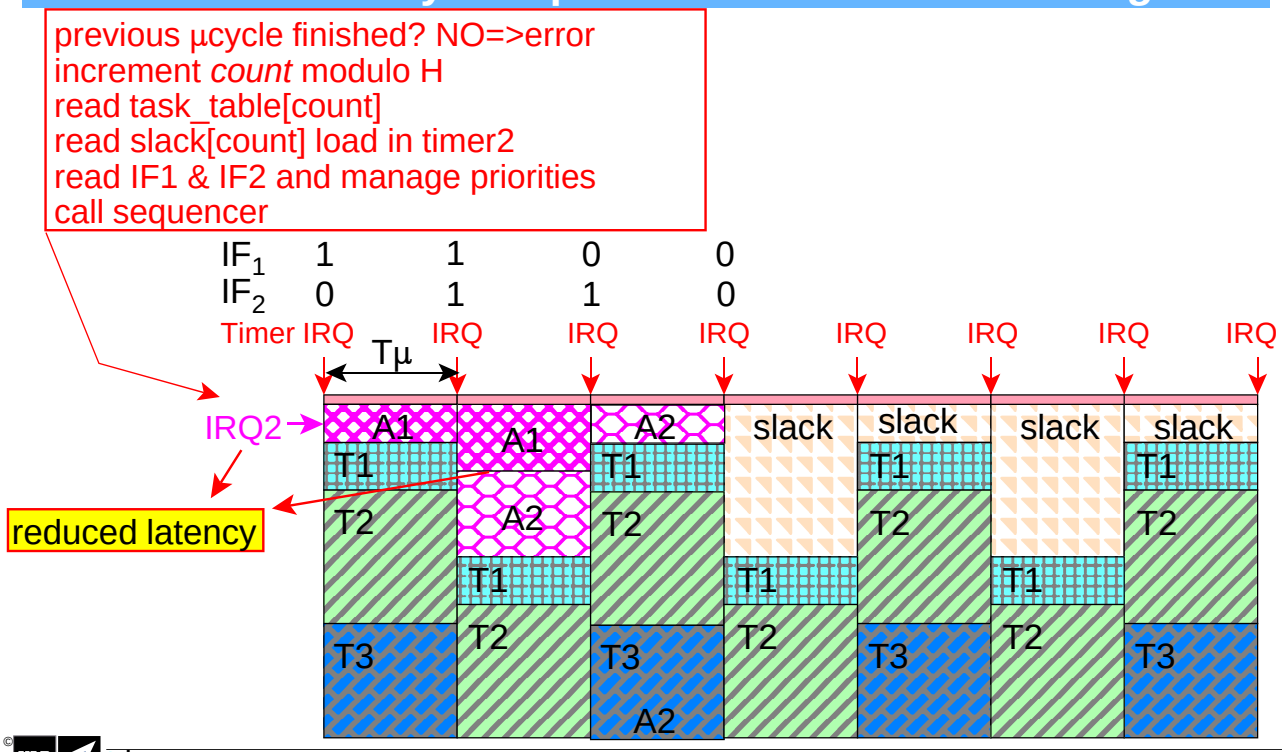
The end of the treatment of A1 must reset IF1 to zero to acknowledge the interruption. At the beginning of the third micro-cycle, only A2 is still in the queue of the aperiodic tasks.

The latency time for the aperiodic tasks is rather long, because it includes

- the time inherent to the mechanism of polling,
- the execution time due to the periodic tasks
- the treatment of aperiodic tasks which have got a higher priority.

Clock-driven dynamic scheduling

reduce latency for aperiodic tasks: *slack stealing*



To improve the response time to aperiodic events, the **slack stealing** can be used.
The principle is not basically different from that of the previous slide, except the slack is placed at the beginning of the micro-cycle.
The scheduling table must contain, for each micro-cycle, the list of tasks to be executed and the duration of the slack.

- at the beginning of the micro-cycle, the scheduler preloads a second timer with the duration of the slack, then releases the aperiodic tasks which are in the queue.

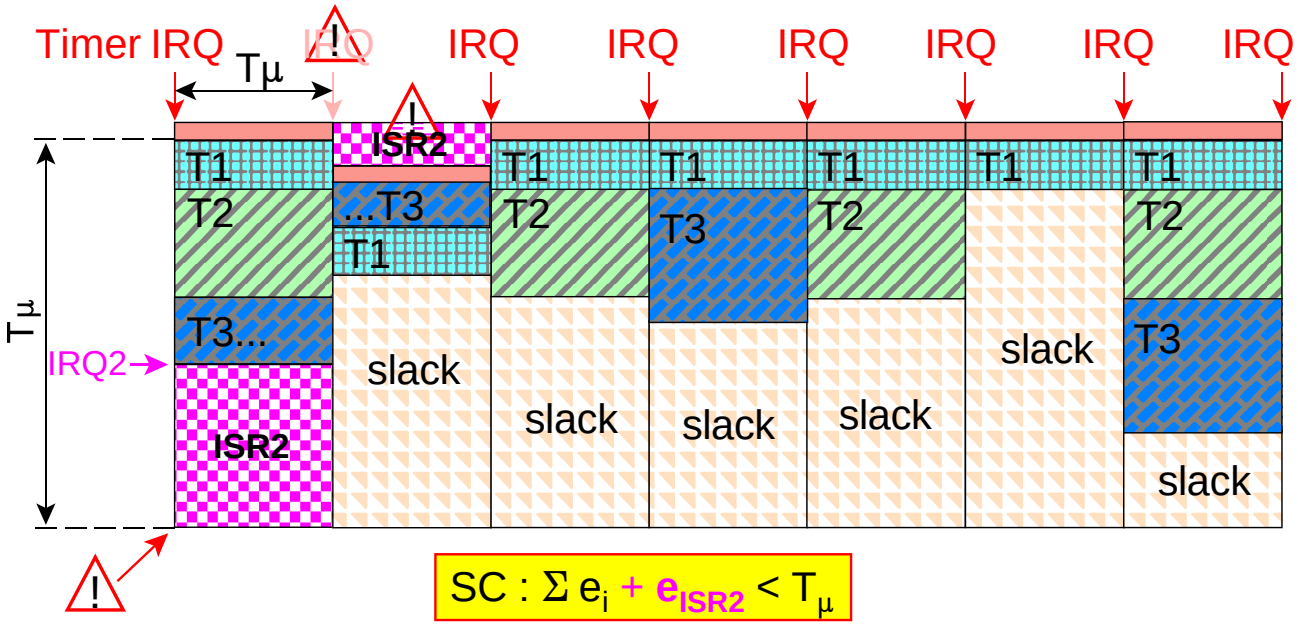
- when this second timer expires, it produces an interruption which will preempt the aperiodic task (if it is still in progress) and will start the periodic tasks.

The advantage of this technique is a **reduction of the latency time for the aperiodic tasks.**

The disadvantages are:
- a much more significant **jitter on the periodic tasks**, quite visible on task T1 (because periodic tasks have lost part of their priority in the system
- a second timer is required, which is not always available.

Clock-driven dynamic scheduling

aperiodic task with non-masked IRQ and bound ei



determinism => reserve time for ISR_2 in every μcycle => waste time

Let us come back to the simple multi-cyclic scheduling and try to **handle an aperiodic event** releasing an aperiodic task whose execution time is known at release time and **whose deadline is *hard***.
The most obvious means consists in connecting this event to an **IRQ whose priority is higher** than that of the main **timer**.

We have to consider the worst case: this interruption occurs in the most loaded micro-cycle (shortest slack). On this figure the high-priority interrupt request IRQ2 occurs within the first micro-cycle, during the execution of T3. The execution time of the interrupt service routine ISR2 prevents T3 from terminating before the end of the micro-cycle.
At this moment the timer interrupt occurs and sets the timer Interrupt Flag, but the execution if the timer ISR is delayed (its interrupt mask is set) until the RTI (Return from Interrupt) of ISR2.

When the timer ISR starts, its first action is to verify that all tasks from the previous micro-cycle are finished, which is not the case for T3. An exception procedure must decide how to manage this overflow (see further).
The solution illustrated here consists in letting T3 finish first, which introduces a jitter on task T1.

We must consider a more serious case: if the second micro-cycle is also extremely loaded, a "domino effect", will occur, which will propagate the shifts until the micro-cycle whose slack is long enough to reabsorb the delay.

The only way to preserve **determinism** is then to foresee the execution time of ISR2 in the slack of the most loaded micro-cycle, i.e. to **consider IRQ2 as a periodic task**, even if it is extremely rare. This can require an oversizing of the CPU and waste resources, and money.

Thus, the simple synchronous systems are well adapted to periodic tasks, a possible background task without priority and sporadic "soft" tasks. On the other hand, they do not handle easily "hard-deadline" aperiodic tasks which have to be transformed in periodic tasks.

Clock-driven dynamic scheduling

aperiodic tasks with more critical deadline

- more difficult to handle
  - parameters unknown at compile time
  - scheduler must be aware of e_{imax} and D_{imin} when a task is activated
- solution : acceptance test
  - ***slack stealing*** is adopted
  - $\exists$? Σ slack < D_i
    - YES : OK
    - NO take safe actions depending on the criticality of the expected delay
  - several aperiodic tasks
    - priority ruled by EDF: least D_i => highest priority
    - more complex acceptance tests

The most difficult case is of course an **aperiodic task** whose parameters (execution time and deadline) are only known until at the moment of its activation.

At the beginning of the first micro-cycle following this activation, the scheduler will carry out an acceptance test by calculating if the sum of the slacks available in the next micro-cycle is sufficient to meet the deadline of the aperiodic task.
- if it is the case, *slack stealing* can be used
- if the deadline cannot be respected, the scheduler must start an **emergency routine** whose actions will depend on the criticality of the tardiness (see futher: managemet of overflows)

If **several aperiodic tasks** are in the queue, the work the scheduler must use an **algorithm** to classify their **priorities** (e.g. EDF Earliest DeadLine First which will be detailed in the RTOS section later in this chapter)

Clock-driven Scheduling

CONTENTS

- hypothesis
- simple static off-line solutions
- dynamic scheduling
- **overflow management**
- conclusions

Clock-driven dynamic scheduling

managing overflows

- definition
  - end of μcycle and ∃ unfinished job(s)
- how is it detected?
  - end of job = reset of a flag
  - ISR_timer
    - check if all flags=0 at the beginning of μcycle
    - set flags=1 for jobs present in the table for the current μcycle
- causes
  - waiting for an event that is blocked
  - transient hardware failure (EMI, high energy particle)
  - *bug* in the software (unforeseen configuration)
  - high priority aperiodic IRQ (see previous slides)

It can happen that a micro-cycle overflows, i.e. at least one job is not finished when the next timer interrupt occurs.

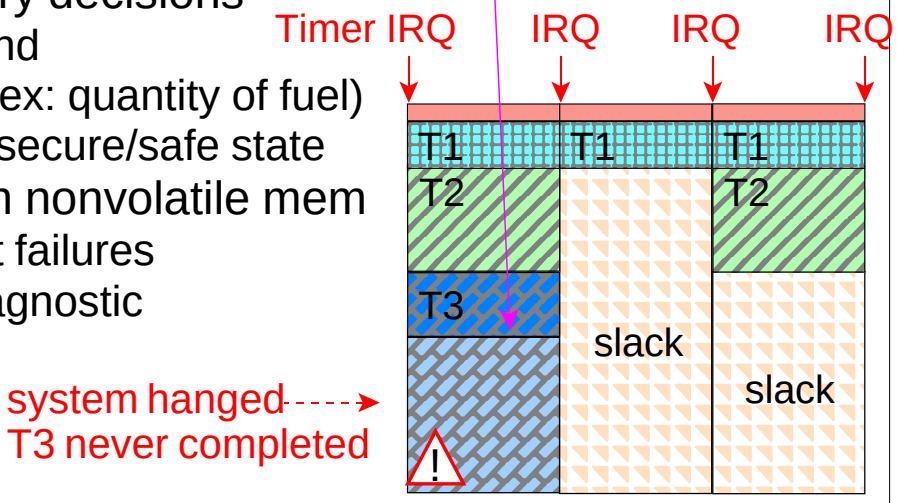
The overflow is not difficult to detect. The scheduler sets one flag for each task which has to be executed in the current micro-cycle, according to the scheduling table. When a job finishes, it resets its own flag. The timer ISR can check all these flags at the beginning of each micro-cycle. If all the flags are not reset an overflow has happened.

The causes of overflows are multiple:
- one of the jobs had to wait abnormally for an event (hardware or software)
- transient failure of the hardware due to a spurious electromagnetic signal or to a high energy particle
- the code of the task contains a bug which appears only for a particular and rare configuration
- a high-priority IRQ occurs in a very loaded micro-cycle, as we have just seen

Clock-driven dynamic scheduling

managing overflows: solutions

- if late result = useless result (ex: fuel injection)
  - abort the unfinished task
  - take conservatory decisions
    - default command
    - use last value (ex: quantity of fuel)
    - stop system in secure/safe state
  - note the event in nonvolatile mem
    - detect transient failures
    - postmortem diagnostic



In case of an overflow, a first criterion consists in **determining if the missing result is still usefull if it arrives too late.**

Let us take as example of an internal combustion engine with indirect fuel injection; if the quantity of fuel to be injected has not been computed before the intake valve is closed it is completely useless keep computing the result. An emergency action will be to keep the last known value.

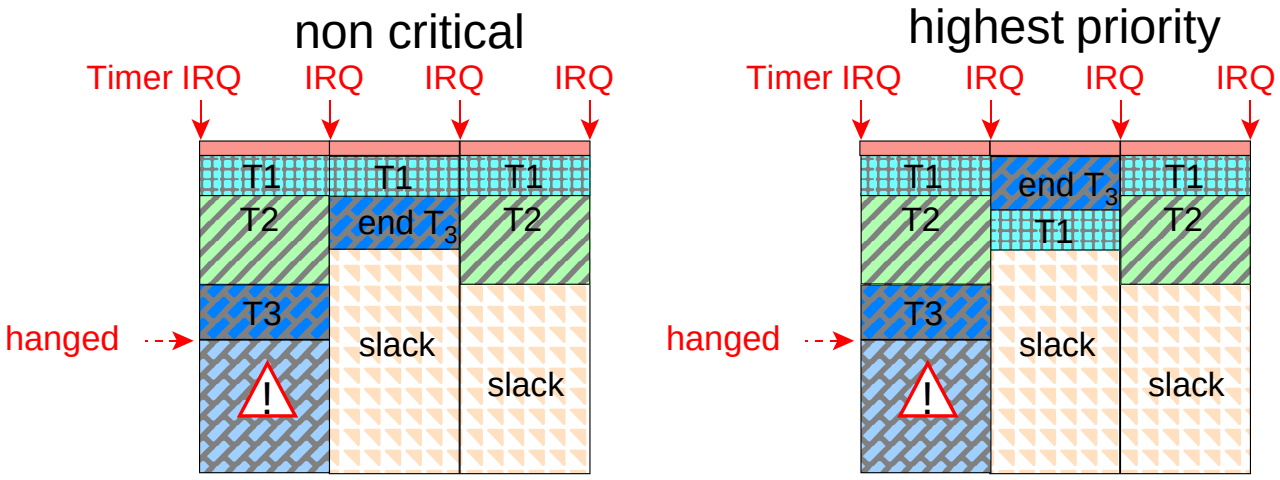
In some critical cases, it is necessary to put the system in a safe state and ir required, to stop it.

Registering the event in a nonvolatile memory "black box often provides an invaluable help for the debug or the maintenance; this information enables
- to detect the occurrence of transient failures which would otherwize have been missed
- to make a more relevant "post-mortem" diagnosis.

Clock-driven dynamic scheduling

managing overflows: solutions (2)

- if aborting causes instability of the system
  - preempt the incomplete job
    - normally happens due to Timer_IRQ at next µcycle
    - /!\ danger if T3 in hanged in a critical section /!\
  - consider the unfinished job as



It can happen that aborting the unfinished job would put the system in an unstable/unsafe state or that the emergency routine would be too heavy to implement.

In this case, it is to better try to continue the execution.

The hanged task will be normally preempted at the beginning of the next micro-cycle by the Timer IRQ.

It is the occasion to point out the **danger of the critical sections**, where all the interruptions are blocked. Critical sections have to be carefully written and debugged. Indeed, if a task is blocked in a critical section, the scheduler will never take the control back and the whole system is hanged; only a manual reset or a watchdog reset will make it possible to restart.
Any loop likely to become infinite and any unforeseen event must be completely prohibited in critical section.

To finish the task, there remain two options:
- keep the normal scheduling process and consider the job to be finished like belonging to an aperiodic task to be executed during the slack of the next micro-cycles
- give priority to the job to be finished, which will introduce jitter on the periodic tasks

Clock-driven scheduling

advantages

- simplicity of concept and design
  - table build off-line
  - on-line sequencer is very simple: read the table
- complex situations can be handled ***off-line***
  - dependancies
  - delays
  - sharing of resources
- security
  - no concurrent tasks
  - no deadlocks
  - validation, test and certification relatively easy
  - determinism
- aperiodic tasks executed during slacks
- good efficiency
  - the scheduler generates almost no overhead
  - if smart choice of periods => processor works ↘

Clock-driven scheduling

drawbacks

- not adapted to evolutive systems
  - maintenance and updates are difficult
  - modify or add a task => new scheduling + new validation
- not adapted to on-line reconfigurable systems
  - must be known a priori
    - number of tasks
    - activation instants/periods
    - execution time
- aperiodic tasks are not easy to manage
  - determinism => over dimension in computing power
  - arrive anyway to a scheduling with priorities

=> ∃ *priority driven* algorithmes