ELEC-H410
REAL-TIME SYSTEMS

Chapter 3

Real-Time Software: Introduction

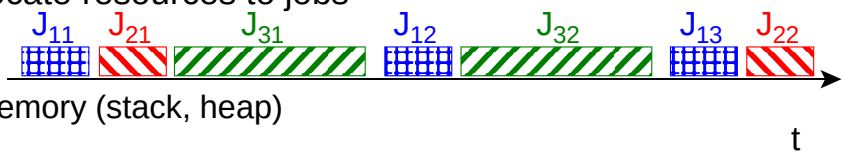
Real-Time Software: Introduction

CONTENTS

- **preliminaries**
  - **concept of task and job**
  - **temporal constraints**
  - **temporal parameters of a task**
- periodic and sporadic tasks
- aperiodic tasks
- polling and interruptions
- scheduling: definitions
- fault tolerance
- conclusions

Dividing a problem into tasks

- advantages to split in **tasks** T_i
  - re-use already existing tasks
  - write and debug simpler pieces of code individually
- example: control a of a motor
  - torque(current) control
  - speed control
  - position control
  - managements of events and alarms
    - end-switch, over temperature, over current,
  - man-machine interface (MMI)
- many systems involve (apparently) **parallel multitasking**
- a task releases jobs J_{ik} associated to time constraints
  - a **scheduler** will allocate resources to jobs
    - CPU time
    - I/O port
    - static or dynamic memory (stack, heap)



Origins of time constraints

ex: robot placing electronic components

- process
  - several time constants
- quality of service (QoS)
  - precision on the position (critical)
  - speed (↘ fabrication costs)
- available hard/soft platform => limitations
  - cost
  - volume
  - consumption
- man-machine interface (MMI)
  - display
  - emergency stop

The timing constraints have various origins.

Let us take the example of a robot which places components on a printed circuit board.

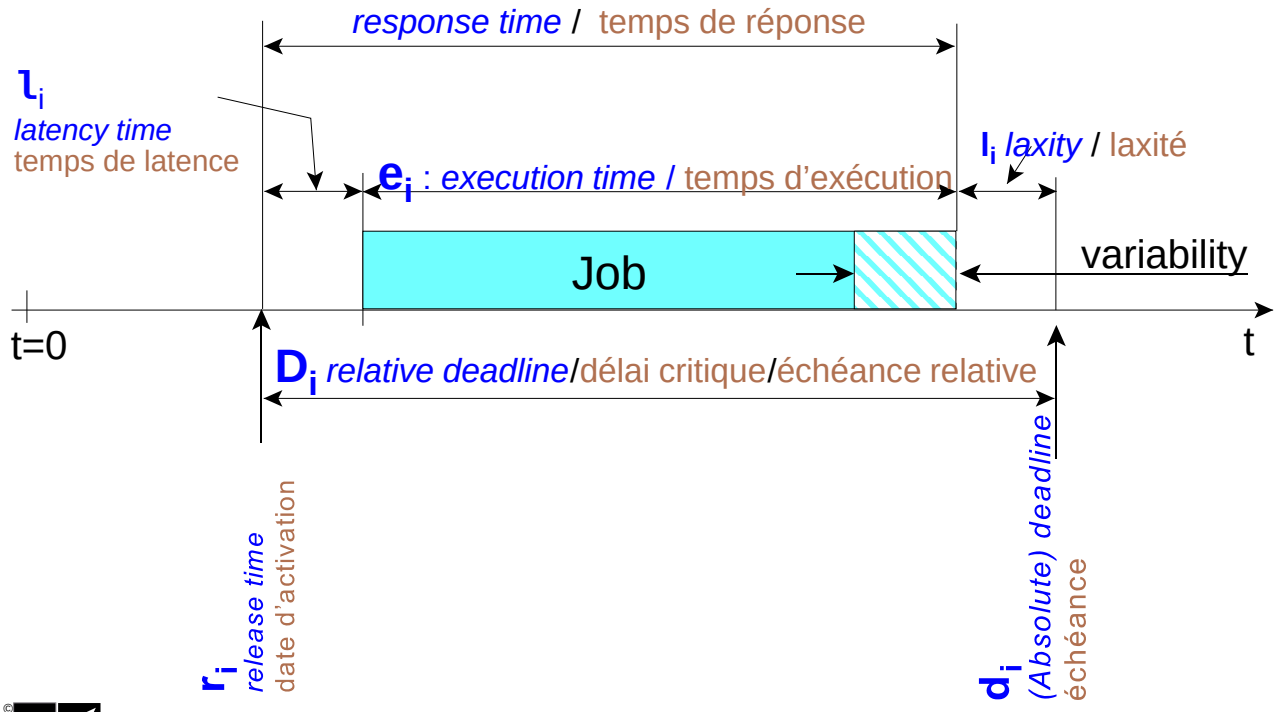
- the first constraints are the physical constraints of the process itself (inertia, motor windings,...) which determine the electrical and mechanical time-constants
- then comes the "performance" or "quality of service", which can be evaluated here by:
  - the precision of positioning (essential so that each pin of the component is welded onto the good pad on the PCB
  - the speed of positioning (components/per minute), which strongly conditions the manufacturing cost
  - the yield (final percentage of accepted boards after production tests)

- the control system also imposes its own constraints (cost, volume, consumption) which will limit the computing power and the electric power of the control device, and will introduce a compromise on the performances

- the man-machine interface, which includes the various commands (including an emergency stop) and the associated displays.

Timing parameters of a task

definitions



Several parameters are associated with each task.

REM: the word "time" is ambiguous since it is used to indicate either a precise moment or a lapse of time.

Referring to an arbitrary origin of time we shall define, for a job Ji:

- r_i : ***release time***: the moment when the job should start to be carried out, if all the necessary conditions were met
- l_i : ***latency time***, counted between the release and the beginning of the execution; li can be null if the processor is immediately ready to carry out this job ; the beginning of the execution is often slightly delayed for several reasons (a higher-priority task is still running, synchronization on another task, lack of a resource, lack of data)
- e_i : ***execution time***: it can sometimes vary significantly (because of conditional branches for example); it is important to know the amplitude of this *jitter* to determine the worst execution time; the lapse of time between the activation and the end of the execution is the ***response time***
- d_i : ***(absolute) deadline***, i.e. the moment on which the task must be finished to respect the constraints of the system
- D_i : ***(relative) deadline*****,** defined as the difference between the deadline and the activation; for the system to be feasible, the response time must be shorter than the relative deadline. REM : the adjectives "relative" and "absolute" are often omitted. In most case we are speaking about a relative deadline; generally the context helps to avoid confusion.
- L_i : ***laxity***: defined as the security margin between the end of the execution and the deadline; if the deadline is missed, the laxity becomes negative and is called ***tardiness***

Real-Time Software: Introduction

CONTENTS

- preliminaries
- **periodic and sporadic tasks**
- aperiodic tasks
- polling and interruptions
- scheduling: definitions
- fault tolerance
- conclusions

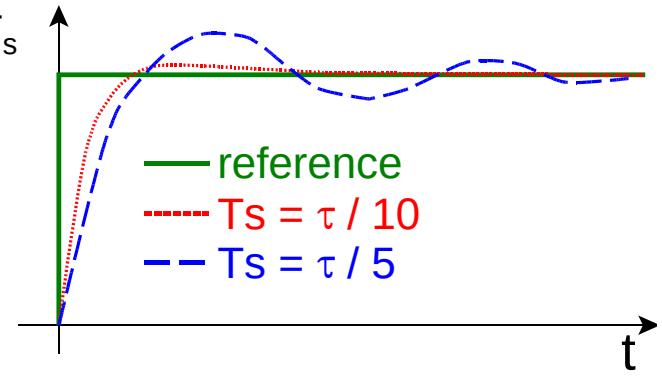
Periodic tasks

choice of the period: T = τ/10

- classical example: PI controller

  $u_k = u_{k-1} + K_P.(\epsilon_k - \epsilon_{k-1}) + K_I.\epsilon_k.T_s$

- contradictory requirements
  - stability and precision of the control
    - rule of the thumb: $T_s = \tau /10 = t_r / 20$
  - computing time
  - control effort
- trend : oversampling
  - minimize anti-aliasing filters
  - decimation + digital anti-aliasing filters (hard or soft)
- example motor current control
  - electrical time-constant: $\tau = 20ms$
  - period of the control task: $T_{reg} = 20ms/10 = 2ms$
  - oversampling at 1MHz of analog inputs: $T_s = 1\mu s$
  - digital decimation factor: 2 ms / 1 µs = 2000



One of the most frequent applications of the periodic tasks is the realization of **digital controllers** (regulators) working at a **given sampling period T_s**, which corresponds to a sampling frequency $f_s=1/T_s$.

The choice of this period is a compromise: if we increase the sampling frequency:

- the **quality** of the control is improved: better precision in the respect of the reference, less overshoots, shorter response time
- the **load of the processor** is increased, which can require a more expensive one
- the **control effort** is increased: we need more energy in the actuators (for example the frequency of commutation and thus losses in power semiconductors, or the peak current load in a servomotor). This energy will be limited either by the power supply, or by the thermal overload of the components

Please refer to the courses on the automatic control for more details. A "rule of the thumb" for the choice of T_s is

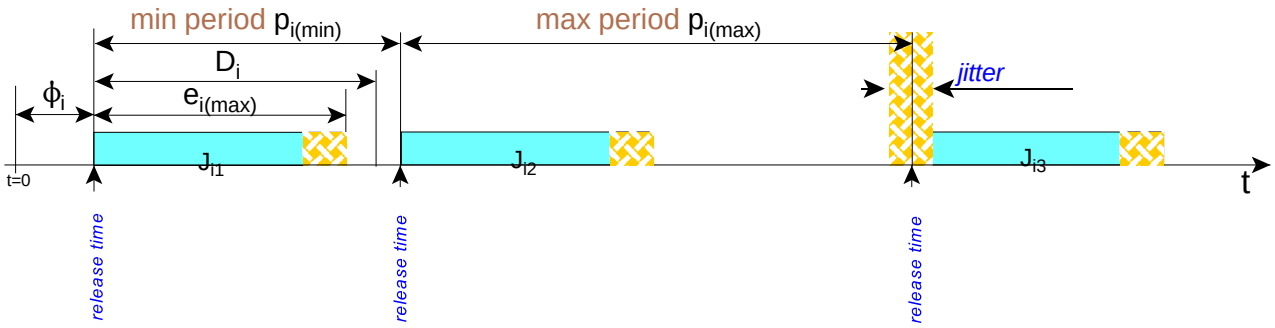
$T_s = \tau / 10$ where τ is the main time-constant to control

$T_s = t_r / 20$ where t_r is the rise-time (for a 1st order system $t_r=2,2\tau$)

Currently the tendency is to increase the sampling frequency of analog input signals, to simplify the design of anti-aliasing filters (weaker order and lower size, down to a simple R-C network). Afterwards numerical decimation is applied down to the appropriate sampling frequency for the control. This decimation can be carried out in software (processor DSP) or by hardware (FPGA).

Periodic tasks

a priori well-known period, but scheduler not perfect



$T_i = \{J_{i1}, J_{i2}, J_{i3}, ...\}$

$T_i(\phi_i, p_{i(min)}, e_{i(max)}, D_i)$ generally t=0 can be choose and $\phi_i = 0$

$T_i(p_i, e_i, D_i)$ if $D_i < p_i$ **constrained dead-line**

$T_i(p_i, e_i)$ if $D_i = p_i$ **implicit dead line**

$u_i = e_i / p_i$ **utilisation**

T_i is periodic if p_i is known at design time and jitter is small

In the most intuitive definition, a task T_i is periodical if its successive jobs $\{J_{i1}, J_{i2}, J_{i3}.....\}$ are released regularly. Let us take again the example of the digital controller, whose frequency is fixed by the division of the microprocessor clock by a *timer*.

The **four parameters** of a periodic task T_i are:

- $\mathbf{p_i}$: the **period** between two releases of the job
- $\boldsymbol{\phi_i}$: the **phase** of this period with reference to the origin of time
- $\mathbf{e_i}$: the **worst execution time** of the jobs
- $\mathbf{D_i}$: the **relative deadline**

and a derived parameter:

- $\mathbf{u_i = e_i / p_i}$: (maximal) task utilization, i.e. the proportion of period p_i devoted the execution (which is also the percentage of CPU time allotted to the task)

The number of parameters can be reduced to

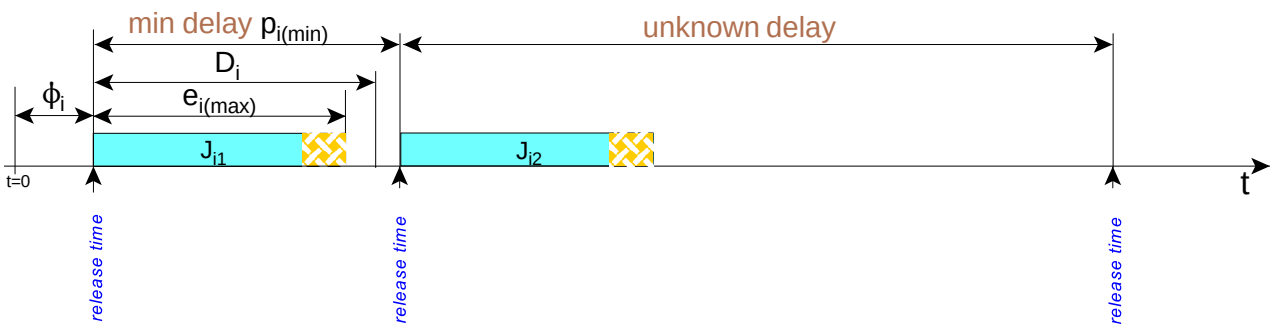
- 3: when the phase=0, by choosing the origin of times judiciously and assuming that all periodic tasks are started simultaneously, which is often the case (see labs)
- 2: when the **deadline D_i is equal to the period p_i**, which is called **an implicit deadline**. If the treatment of a job is finished before this job is reactivated, the task is considered as accomplished. Otherwise if **$D_i < p_i$,** it is called a **constrained deadline,** because it is more severe.

The *effective period* and the execution time are sometimes affected by *jitter*, i.e. a random variations around the average (depending on conditional branching and scheduling problems, see next §).

Consequently, we shall speak of a **periodic task when a reference period is known at design time** . At run-time, the scheduler will be considered as working properly if the average period corresponds to the reference and is only affected by an **acceptable jitter, due to scheduling imperfections** .

Sporadic task

parameters



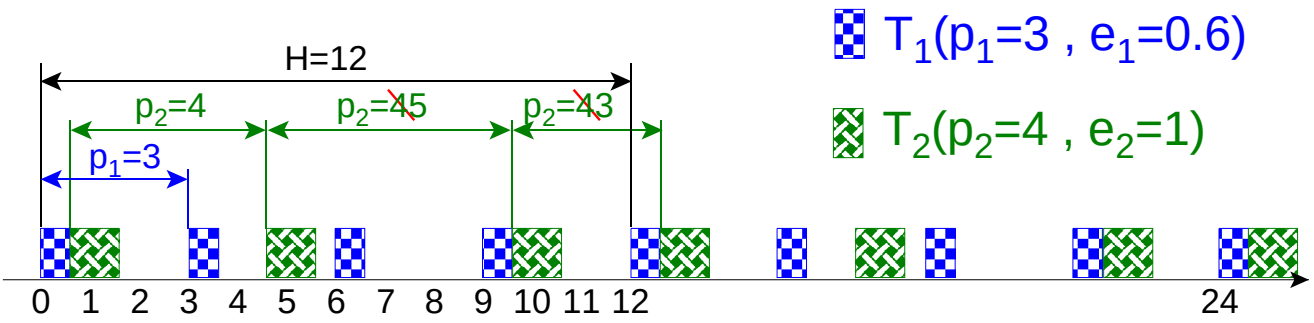
T_i is sporadic if
- only $p_{i(min)}$ is known at design time
- release time is only known at run-time

When only a **minimal lapse of time between two job releases** is known at design time, the task is called **sporadic**. The release time is only known at run-time.

REM: some authors [LIU2000] also call such a task periodic because they can be scheduled as if they were periodic, with the miminum period taken as reference period (which will of course consume more CPU resources)

Periodic tasks

scheduling of n periodic tasks T1..Tn



hyperperiod :	$H=lcm(p_1...p_n)$	$H=lcm(3,4)=12$
max number of jobs :	$\Sigma H/p_i$	12/4 + 12/3 = 7
task utilisation:	$u_i = e_i / p_i$	u_1=0.6/3=20% u_2=1/4=25%
total utilisation:	$U = \Sigma u_i$	U = 20%+25% = 45%

if U>1 => impossible : change µP or several µP

The majority of the systems are multitask. We consider here the case of a mixture of N periodic tasks $T_1..T_n$ of period $p_1..p_n$ and maximum execution time $e_1..e_n$.

The figure above illustrates the simple case of two strictly periodic tasks. The time scale is arbitrary.
A ***hyperperiod H*** (or **macro-cycle**) appears, at the end of which the same pattern comes again and again in the sequence of the jobs. This hyperperiod is the least common multiple of the periods p_i

$$H = lcm(p_i)$$

We can also define a **total rate of utilisation U** of the processor, which is the percentage of the time occupied in execution rated to the macro-cycle; U is the sum of the utilisations of each task:

$$U = \sum u_i / p_i$$

If the rate of utilisation U is higher than unity, the scheduling is impossible; we have to choose a faster processor to decrease the execution times, or to consider a multiprocessor system.

Let us finally notice that, even in such a simple system, it is **impossible to guarantee strict periodicity**.
The two jobs are in conflict in the neighbourhoods of time t=9. In this figure, we gave priority to task 1, so that task 2 must be delayed in t=9.6. The period p_2 is well equal to 4 on average, but takes values 3,4 and 5.

Real-Time Software: Introduction

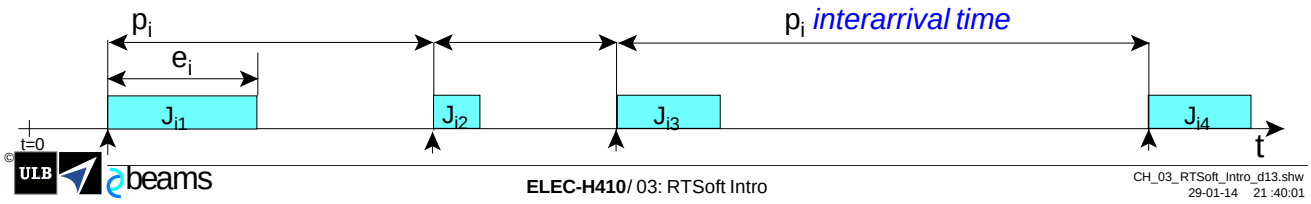
CONTENTS

- preliminaries
- periodic and sporadic tasks
- **aperiodic tasks**
- polling and interruptions
- scheduling: definitions
- fault tolerance
- conclusions

Aperiodic tasks

no prior information about release time

- examples
  - braking for a child crossing the road
  - failures, breaking, short circuit, alarm, lightning
- randomness
  - **inter activation delay can be arbitrarily small** (≠ sporadic)
    - distribution A
  - **execution time**
    - distribution B
- 2 classes
  - **soft deadline**:
    - ex: changing reference temperature of an oven with large thermal constant
  - **hard deadline**: security, safety or cost are critical
    - short circuit in the inverter of a locomotive



Events or disturbances occur in the majority of the systems. Their occurrence is random, and can be related to human actions like:

- pull the trigger of a game console, or click on a mouse
- start/stop of a cruise control
- push on a brake pedal
- change a reference of speed or temperature

It can happen to be more serious, like failures or accidents:

- stopping of a pipe
- mechanical break
- short-circuit
- lightning

Such events are sometimes called **asynchronous**, they can be characterized by

- a probability distribution of the inter-arrival time of the associated jobs
- a probability distribution of execution times
- a **lapse of time between successive activations** that can be **arbitrarily small** , which constitutes a fundamental difference compared to the periodic and sporadic systems; the notion of period is totally absent, hence the adjective **aperiodic**.

We shall also make a distinction based on the criticality of the deadline.

- soft deadline (for example regulating changing the temperature set point of the controller for a heating system)
- hard deadline (for example alarms or short-circuits)

Rem : in the literature [LIU2000] the definition is different

- "periodic" covers strictly periodic and sporadic systems, as already mentioned refers to strict periodic systems
- "aperiodic" is used for aperiodic events with soft deadline
- "sporadic" is used for aperiodic events with hard deadline

Aperiodic tasks

time constraints

- define an acceptable deadline
  - if not feasible in software => hardware (PLD/FPGA)
- define a degree of safety
  - **a single failure must not lead to a critical situation**
  - critical system => fault tolerant (redundancy)
- define priorities
  - sometimes easy: example of a printer
    - motor overcurrent (risk of damage, even fire?)
    - paper jam
  - often difficult
    - long studies and simulations
    - tests are difficult

We have just seen examples of aperiodic events that will require the release of aperiodic tasks.

According to the system and to the specific event, an acceptable deadline has to be defined for the task.

For example:
- 50ms between the action on a keyboard and the display of the character
- 5µs between the detection of an overcurrent in a transistor and the moment when the transistor is completely switched-off

Depending on the speed of the processor it can(not) be possible to meet the deadline. The 5µs deadline here above is unachievable by software (also called programmed logic), even with high-end processors. In this case, tasks must be carried out by hardware, such as PLD (Programmable Logic Devices) and FPGA (Field Programmable Gate Array).

Let us notice that, **if a task is critical for the safety** , it cannot be ensured by a processor alone, even if it is fast enough, because **a simple breakdown must not lead to a critical situatio** n. In this case, the **redundancy** is essential (see further: fault tolerance)

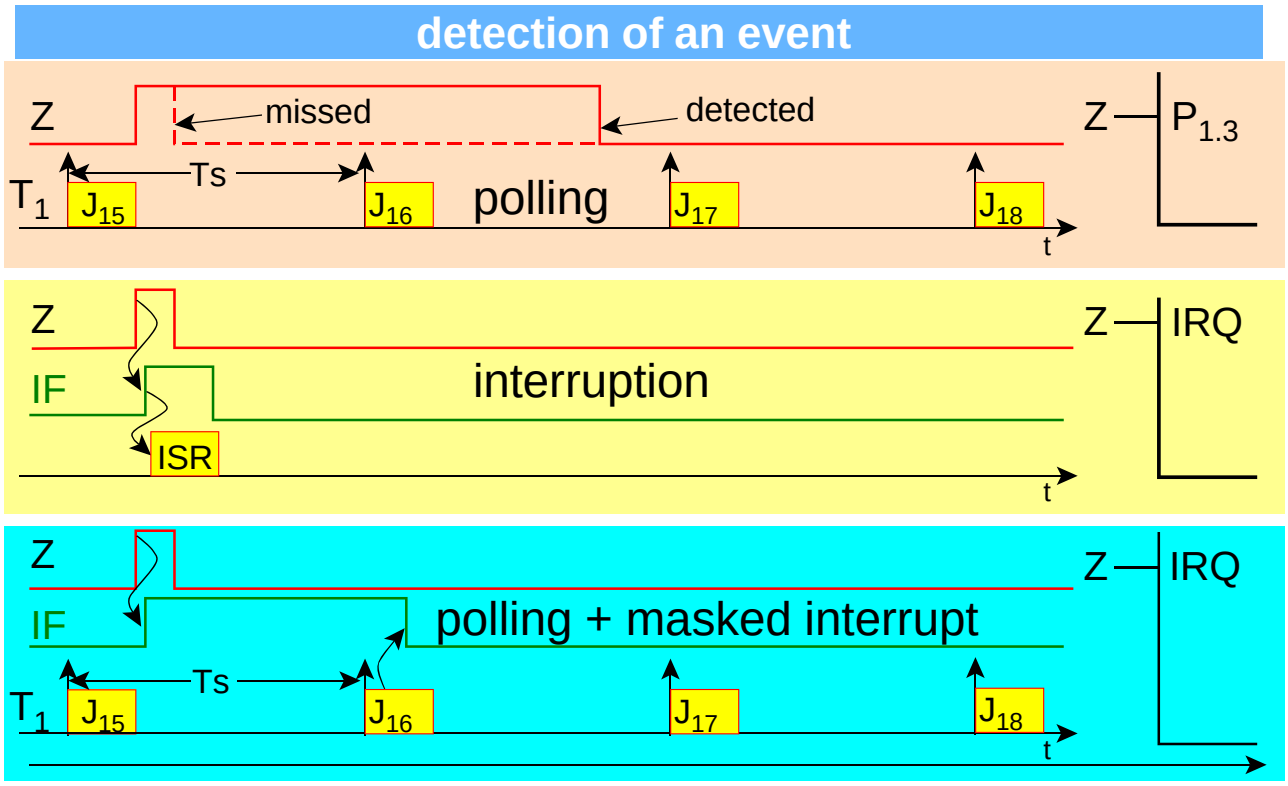
Lastly, if several events occur simultaneously, it will be necessary to define their priorities.
Sometimes, it is a simple problem (e.g. in a printer, a blocked motor likely to overheat a winding has obviously priority on a lack of paper.)
In many cases the definition of the priorities is complex and requires a lot of simulations and a long period of tests.

Real-Time Software: Introduction

CONTENTS

- preliminaries
- periodic and sporadic tasks
- aperiodic tasks
- **polling and interruptions**
  - **how to detect an event**
  - **typical delays**
  - **critical sections**
- scheduling: definitions
- fault tolerance
- conclusions

Polling or interruption ?



When you want to detect an external logical event, three techniques can be used:

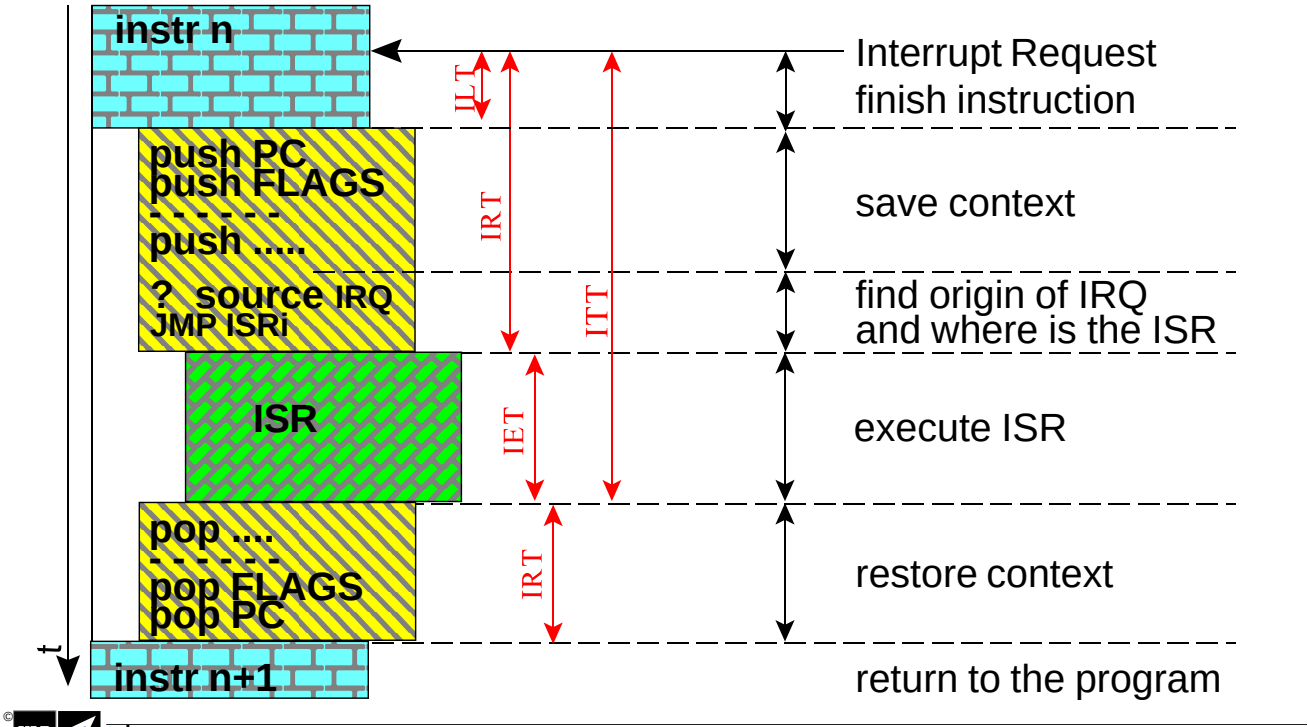
- the first one is the periodic examination of an input or **polling**: the signal Z to be detected in connected to an input port. The sampling of Z (and normally the associated processing) is carried out by a **periodic task** with a **period Ts**
  - Advantages:
    - **simplicity** and **robustness** (it is easier to debug a program without interruptions)
    - **determinism**: in the worst case Z is detected after a delay Ts
  - disadvantages :
    - **detection**, and thus also processing, is **delayed** by Ts in the worst case; this can be unacceptable if the deadline is less then Ts (constrained deadline)
    - detection can be missed if the event is shorter than Ts (e.g. optical barrier for counting)

- we can also use **interruptions**; in this case the signal Z is connected to a special interrupt request pin (or an input pin configurable as interrupt) and the active edge of the signal will set a flip-flop called Interrupt Flag (IF) within the µP. If the interruption is not masked, the processor will automatically (in hardware) call the ISR (Interrupt Service Routine) which will process the event.
  - advantages
    - taking into account the interruption can be very fast (see further § for the causes of possible delays)
    - the event Z is detected even if it is very short (the minimum duration is 1 or 2 machine cycles which is largely less than 1 µs for most processors)
  - disadvantages
    - it is more difficult to debug when interrupts occur
    - **absence of determinism** in the response time

- finally the third option is to work by **polling of the interrupt flag IF**; signal Z remains connected to the interrupt request input IRQ, but the interruption is masked. We get all the properties of polling, but we have no risk to miss the event. (/!\ IF has to be cleared during the treatment of the event Z)

Interrupts

timing of a non-masked interrupt



In this figure, time proceeds from top to bottom. The mask of interruption is not active.

- the interrupt request occurs during instruction N of the program. It cannot be taken into account immediately, because the current instruction must always finish. This delay is **ILT: Interrupt Latency Time**. Since the interruption is not masked, ILT is at most the duration of the current and hence the worst-case delay is the duration of the longest instruction of the processor.

- next, the processor saves the context of the program on the stack to be able to restart the interrupted program properly. Depending on the type of processor, the duration of this phase can vary considerably (i.e. the context is at least the Program Counter PC, but lots of internal registers can have to be saved). The majority of the processors also activate the Global Interrupt Mask, which prevents further interrupts (*nested interrupts* are not authorized in the default mode and must be authorized within the ISR).

- then begins the phase to investigate which is the cause of interruption, and the management of the priorities if several interruptions are concurrent

- once the interruption number is known the µP reads a vector table to extract the address of the first executable instruction of the ISR, then makes a call to this address, which launches the execution of the ISR. The delay between the interrupt request and the beginning of its service is **IRT: *Interrupt Response Time***.

- the ISR executes during **IET *(Interrupt Execution Time)***; the total delay between the interrupt request and the end of its service is **ITT *Interrupt Total Time***.

- following the instruction RETI (RETurn from Interrupt), the processor restores the context to be able to restart the normal execution of the program; this operation has got a duration **IRT *Interrupt Recovery Time***.

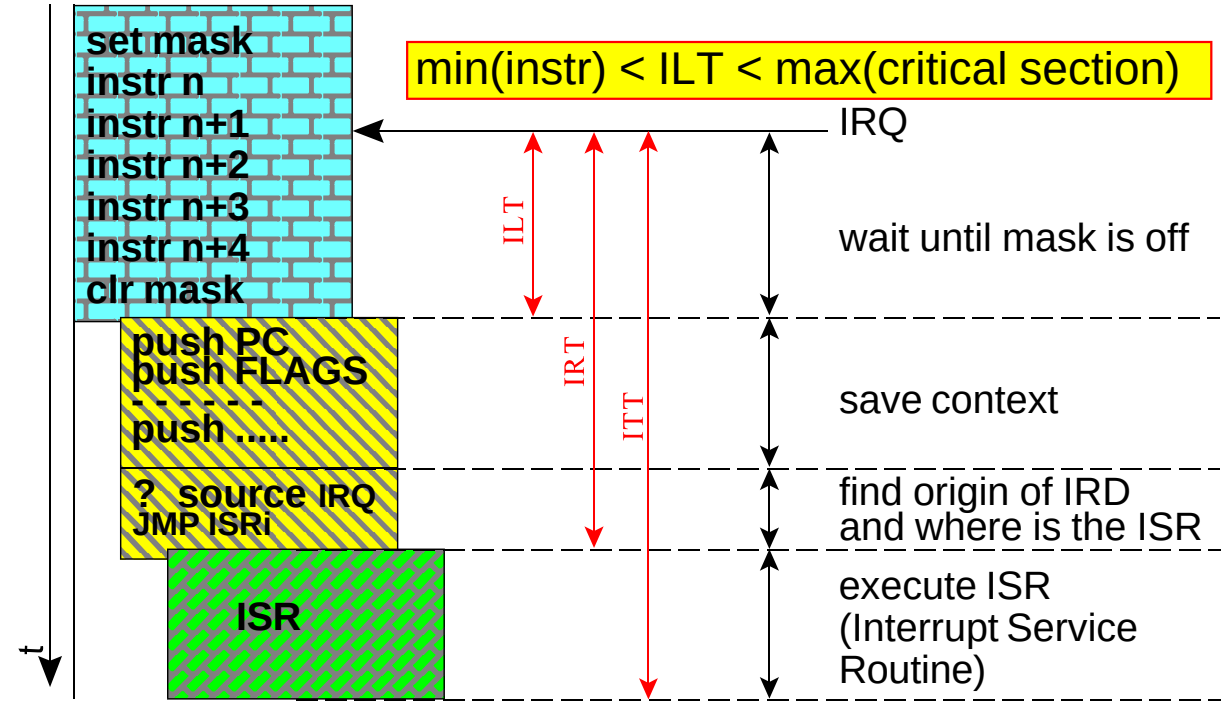
Interrupts

concept of critical section: IRQ have to be masked

- definition
  - zone of the code which must be executed without interruption
- examples
  - installing an interruption (ISR+vector)
  - configuration of a peripheral that can generate interrupts
  - saving the context
  - scheduling
  - output of a bitstream with accurate timings
  - access to a shared resource (peripheral, global variable)
- create a critical section
  - mask one, several or all IRQ

Interrupts

timing of a masked interrupt



If the interruption occurs during a critical section, the **latency time is increased**, since the request of interruption will be taken into account only at the end of the instruction which resets the Interrupt Mask.

The latency time is thus equal to:
- the duration of the shortest instruction, in the most favorable case
- the duration of the longest critical section, in the worst case

This explains the **lack of determinism** in the timing of the interruptions if critical sections are used.

The duration of ISR and critical sections should be reduced to a minimum.

Real-Time Software: Introduction

CONTENTS

- preliminaries
- periodic and sporadic tasks
- aperiodic tasks
- polling and interruptions
- **scheduling: definitions**
- fault tolerance
- conclusions

Definitions

scheduling problem

- Scheduling means managing the execution of jobs
  - from a set of tasks $\{T_i\}$ periodic, sporadic, aperiodic
  - while respecting the constraints
    - deadlines
    - possible dependencies between tasks
      - precedence
      - sharing of resources
      - synchronization
    - quality of service / performances
- 2 main steps
  - find a test to prove that $\{T_i\}$ can be scheduled ($\exists$ at least 1 solution)
  - find an algorithm to implement one of the solutions
- tools
  - development tools (hardware/software)
  - CAD to specify and validate the problem
  - schedulability analysis and efficient scheduler design are complex problems which have been a subject of research for 40 years and are always hot

Definitions

Families of algorithms

static

- tasks params known a priori
- all decisions at compile time
  - ♦ splitting in tasks
  - ♦ dependancies
- OK for frozen systems
- ⊕ ∃ much tools for analysis
- ⊕ predictability

dynamic

- tasks params known at run-time
- all decisions in real-time
  - ♦ based on ready tasks
  - ♦ dependancies
- OK for evolutionary systems
- ⊖ few tools for analysis
- ⊖ non predictable

off-line

- preliminary calc. off-line
- simple sequencer on-line
- ⊖ rigid
- ⊕ efficient
- ⊕ 1 solution proves feasability
- ⊕ reproducibility

on-line

- no preliminary calc.
- on-line algorithm
- ⊕ flexible
- ⊖ ↗ µP load
- ⊖ feasability difficult to prove
- ⊖ reproducibility difficult to guar.

Static scheduling means that:
- all the parameters of the tasks are known at design time
- all the decisions related to the structure of the tasks and the dependence between them are taken before running the system

Static scheduling is well appropriate for stable, non evolutionary systems (which is the case for lots of processes).
Many analysis tools allow to predict the behaviour of such systems.

Dynamic scheduling makes it possible to manage at run-time tasks whose parameters are only known when they are released. The scheduling algorithm then decides "on the fly" which task will run according to their priorities and possible dependencies.
Dynamic scheduling is appropriate for evolutionary systems and gives flexibility. On the other hand, less design/analysis tools are available and the behaviour is more difficult to predict, to guarantee and especially less reproducible, which is an important drawback for critical systems.

Definitions

preemptive / non-preemptive scheduling

preemptive

- scheduler can interrupt a task
- scheduling starts if
  - IRQ activates a task
  - current task ends/suspends
  - time-slice expired
- advantages
  - can minimize response time
  - a task cannot monopolize CPU
- disadvantages
  - scheduling takes µP resources + time
  - reproducibility more difficult to guarantee

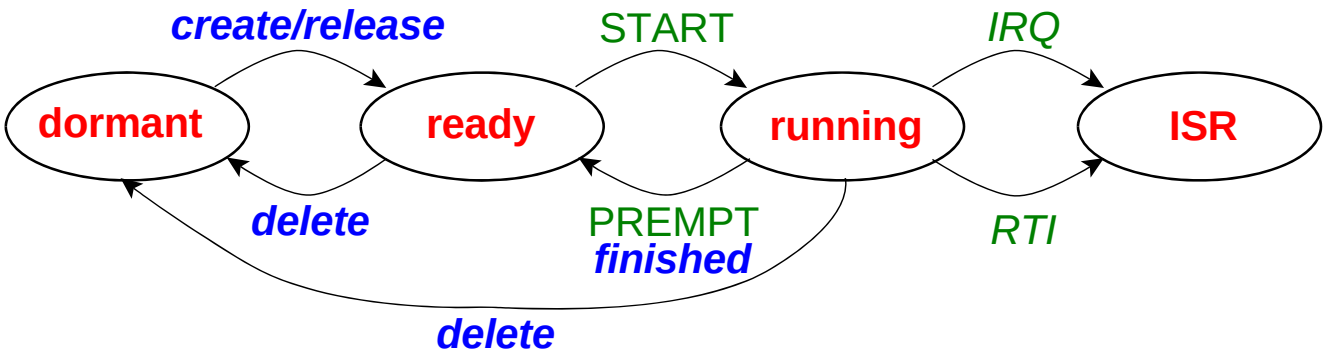
non-preemptive

- tasks give control back voluntarily
- scheduling starts if
  - end of current task
  - current task suspends itself
- advantages
  - tasks do not share resources
  - few µP resources + time
- disadvantages
  - longer response time
  - risk of hanging if a task keeps the µP indefinitely

Definitions

state of tasks and transitions

program action
SCHEDULER ACTION
task state



The figure illustrates the states of the tasks and the verbs associated with the various transitions.

- **"dormant"** corresponds to a task which resides in memory but which currently has not to be scheduled
- ***create/release*** the task will place it in the **"ready"** state; in most systems all tasks are created by the initialization code, just after the boot-up. The task receives all the resources required to run (static memory, stack) excepted the CPU
- **"ready"** corresponds to the state of all tasks that could run if the CPU was not busy executing the current **"running"** task (the only one to own the CPU)
- when the scheduler decides to ***start*** a task (i.e. change its state from "ready" to "running") it makes a ***context switch*** (the state of the previous task is saved and a call is made to the new task)
- a "running" task will go back to the "ready" state either because it is ***finished***, or because the scheduler decides to ***preempt*** it to start another task which has acquired a higher priority
- a running or ready task can be ***deleted*** to return to the dormant state. It is a rare event because **in most embedded systems all tasks remain ready during the whole life of the system** . Exceptions can be for instance an intermittent connection with a computer that creates a task to download new parameters; when the task is finished, the computer is disconnected, and the task can be deleted.
  REM: a task might be deleted because it has generated an exception (illegal memory access for example), this should not happen in a hard real-time system since the function of this task has to be fulfilled and this might be a safety issue. The task should be recreated immediately.
- a running task can be **interrupted** by a classical Interrupt Request (IRQ); the CPU will execute the ISR, terminated by a special instruction called Return from Interrupt (RTI). Normally, an interrupt is not a preemption, and the CPU is given back to the interrupted task, which is still in the running state.
  REM : the IRQ can be an event such that the ISR will release a high priority task, which leads the scheduler to preempt the running task (more details later in this chapter)

Definitions

capacity to schedule: clear-sighted and optimal

- clear-sighted: knows the future
  - can anticipate any release of tasks and their parameters

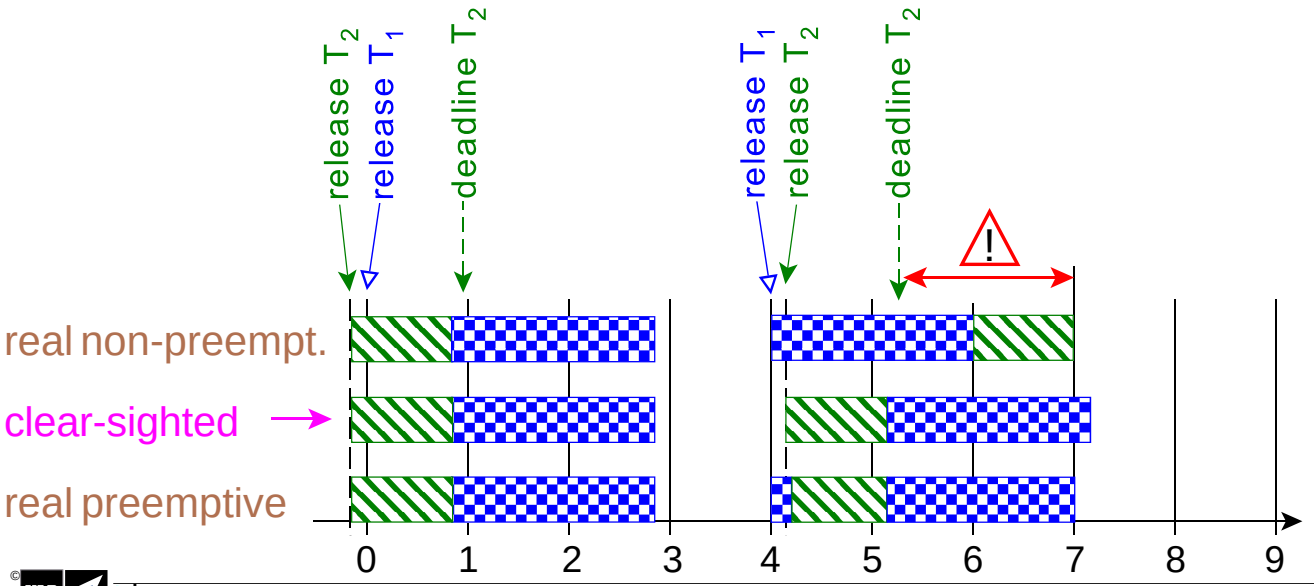
- optimal
  - can find a solution if clear-sighted can
  - no other algorithm can succeed if an optimal algorithm fails
  - generally ***dynamic => not optimal*** (at least without some restrictive hypothesis)

Definitions

example : clear-sighted vs real

T_1: p_1=4 e_1=2 D_1=4 strictly periodic

T_2: p_2=4±Δp e_2=1 D_2=1,1 sporadic + constrained deadline



This figure illustrates the difficulty of scheduling, even for a simple problem with two tasks of same priority. Task T_1 is strictly periodic, whereas the T_2 task is sporadic with the same average period, a small jitter and a strict *constrained deadline* D_2 (only 10% higher than the execution time e_2).

- the first release of T2 takes place right before time t=0; the dynamic scheduling algorithm sees that the only "ready" task is T_2 and starts it, T_2 is now "running"
- in t=0, task T_1 is released, but its execution is delayed until the end of T_2; the deadline D_1 is respected
- in t=4, task T_1 is released and, being the only ready task, starts running
- because of the jitter on the period of T_2, the activation of T_2 takes place just after t=4, and T_2 is delayed until the end of T_1; the **deadline of T2 is missed**.

A clear-sighted scheduler would have delayed the execution of T_1.

To solve this problem, we need a **preemptive scheduler** and a **priority algorithm based on the deadline**; in this case, the release of T_2 (at t=4+) gives the control back to the scheduler, which decides to preempt T_1 and starts T_2 instead. When T_2 is finished, T_1 remains the only ready task and the scheduler restarts its execution.

Real-Time Software: Introduction

CONTENTS

- preliminaries
- periodic and sporadic tasks
- aperiodic tasks
- polling and interruptions
- scheduling: definitions
- **fault tolerance**
- conclusions

Fault Tolerance

- fundamental rule: **a single failure must not cause a critical situation**
- solution : redundancy
  - hardware
    - 2 different mechanisms at least
    - as simple as possible (bi-metal, fuse,...)
    - voting circuits (majority 2 / 3)
  - hardware + software
  - software
    - code redundancy (hot research topic)
    - at least 2 independent versions
      - different programmers
      - different compilers

The rule is that a **single failure must never cause a safety problem** .

It is thus necessary at least to duplicate the mechanisms of safety, by the simplest possible devices, and relying on different mechanisms: this is called redundancy.

The redundancy can relate to the hardware, the software, or both.

We can for example:
- run the same task on 3 processors and use a reliable majority (2/3) voter
- add another unit dedicated to safety

Example: a processor runs a task dedicated to the control of the heating of a fluid that can explode.
- we can place several reliable temperature sensors (thermocouples) and use one measurement for the control.
- another task can monitor the temperature measured by another sensor and define an alarm threshold. This function can also be fulfilled by an analog amplifier coupled to a comparator connected to an interrupt request.
- finally the ultimate level of protection will be performed by a bimetal switch which will open the heating circuit if the temperature goes beyond the appropriate threshold

The study of the fault tolerance is outside the scope of this course.

Real-Time Software: Introduction

CONTENTS

- preliminaries
- periodic and sporadic tasks
- aperiodic tasks
- polling and interruptions
- fault tolerance
- **conclusions**

Conclusions

- a real-time system must achieve tasks
- tasks release jobs which are executed concurrently by interleaving them
- each job has got a deadline (more or less hard and critical)
- the type of a task depends on how it is released
  - periodic: min period known at design time
  - sporadic: min inter activation delay known at design time
  - asynchronous: inter activation delay arbitrarily small
- short events are best captured by interrupts
- schedulers are the heart of real-time multitasking
- critical => redundancy (hard and soft)