v 1.1
20221106

Embedded Systems and IoT

Ingegneria e Scienze Informatiche - UNIBO
a.a 2022/2023
Lecturer: Prof. Alessandro Ricci

[module-2.3]
TASK-BASED ARCHITECTURES

OUTLINE

- Task-based Architecture
  - integration with synchronous FSM
- Cooperative scheduling of tasks
- Execution analysis
  - CPU utilisation, Worst-Case Execution Time,
  - problems
    - overrun, jitter
- Tasks with deadlines and priority-based scheduling
  - static and dynamic priority
- Tasks based on event-triggered FSM

TASK-BASED ARCHITECTURES

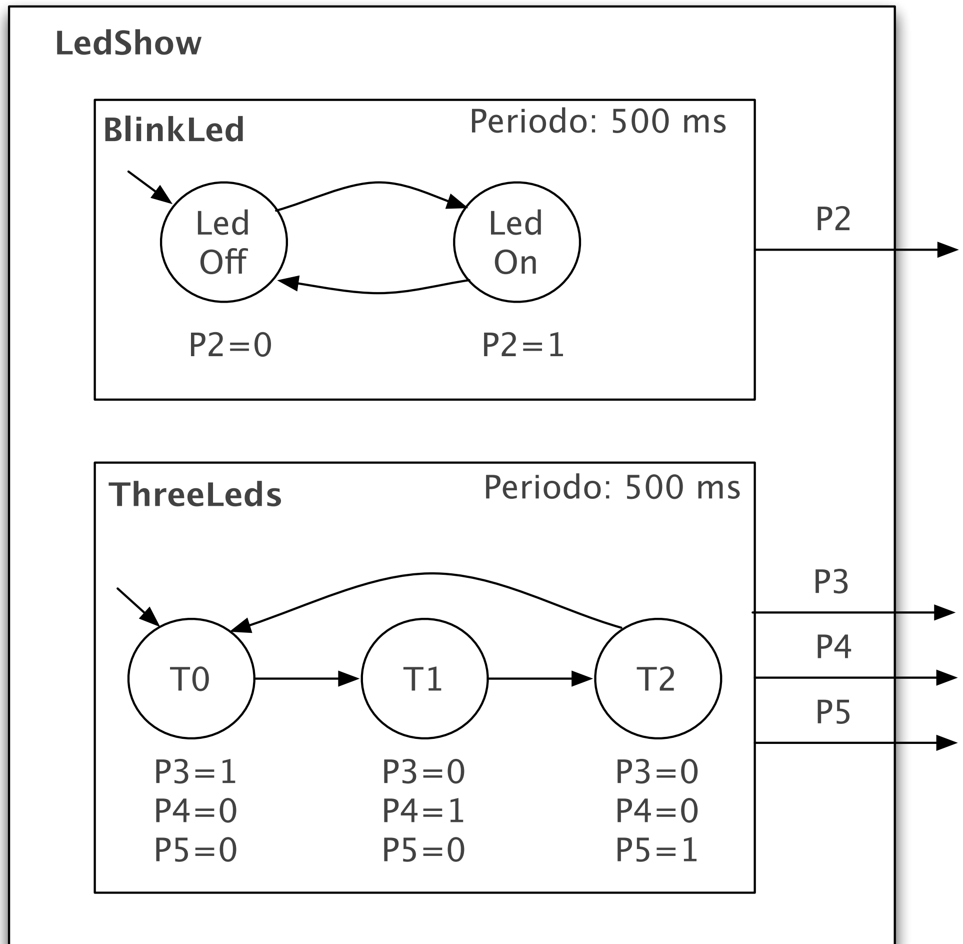
- Problem: complex embedded software modelling and design
  - need of proper approaches to decompose/modularise the behaviour and functionalities
- Approach: Task-based architectures
  - The behaviour of the embedded software is decomposed into a *set* of **concurrent tasks**
  - each task represents a specific well-defined confided unit of work/ job to be done
  - the behaviour of *each* task can be described by a FSM
  - the global behaviour is the result of the execution and interaction of the concurrent FSMs

EXAMPLE: LED-SHOW

- LedShow example: 3 + 1 leds (p. 68, [PES])
  - One led: blinking with period: 500 ms
  - The other three leds: turn on/off in sequence, with a time interval of 500ms
- This could be modelled as a single task / FSM, however the modelling is much easier and simpler if we model it as the explicit composition of 2 tasks/FSM
  - single task/FSM version: much higher number of states and transitions

BLOCK DIAGRAM

- Representing each task as a block (rectangle), showing the input and output of each task/block:



TASK DECOMPOSITION: ADVANTAGES

- **Modularity**
  - every task is an independent module
  - the “interface” of the module in this case is the set of input/output variables / objects that the task is using (as input/output)
    - these can be shared with other modules
- Advantages
  - reduced complexity of the single task vs. the global behaviour
  - easier debugging
  - reusability

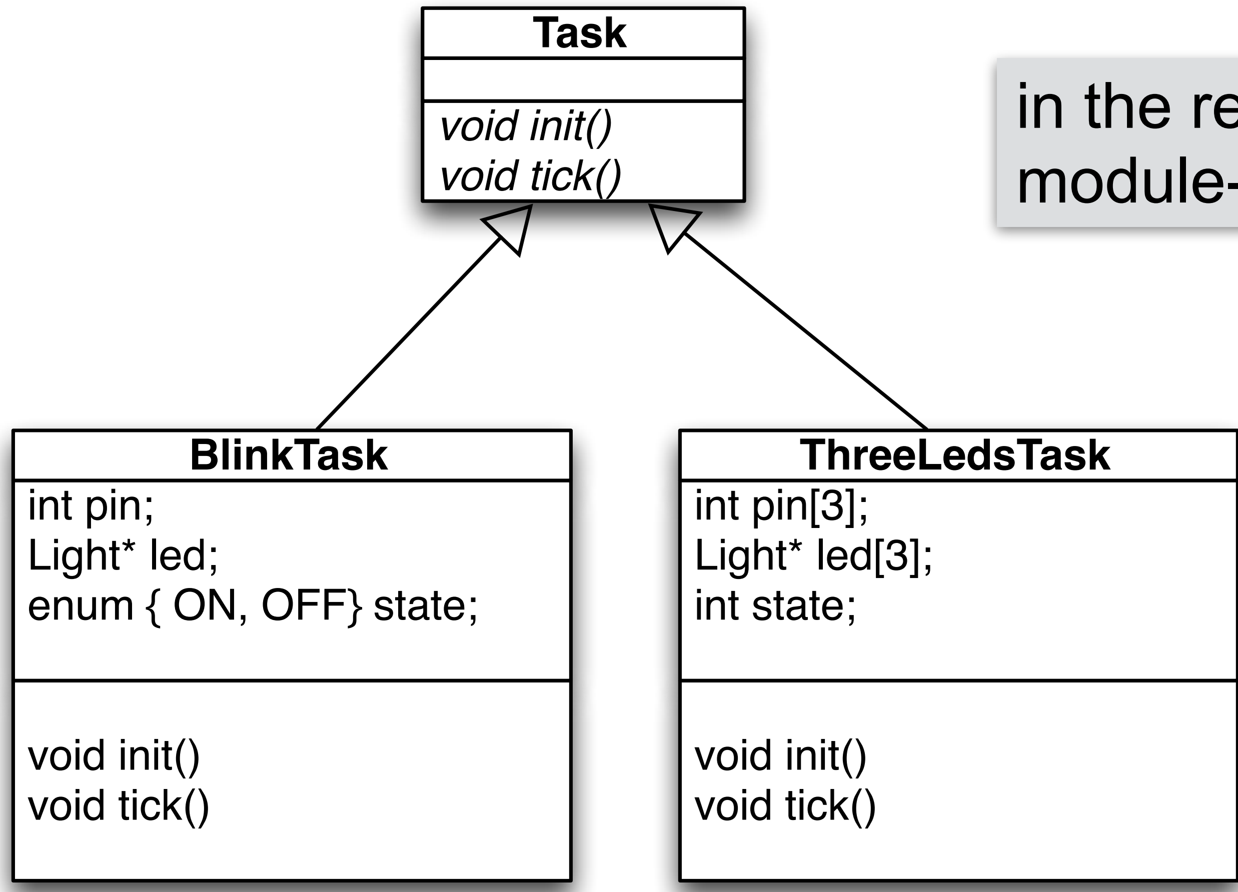
TASK DECOMPOSITION: CHALLENGES

- Tasks are **concurrent**
  - their execution overlaps in time
  - conceptually each task has its own logical control flow
- Tasks may have *dependencies* that create interactions among tasks and that need to be properly managed
  - interaction and coordination mechanisms
  - in our case, typically based on shared variables/objects

IMPLEMENTING TASK-BASED ARCHITECTURES - FIRST

- Introducing an abstract base class **Task**
  - **init** method
    - to initialise the task, called once
  - **tick** method (equivalent to the “step” method in FSM)
    - encapsulating the behaviour of the task
    - called periodically (period p)
- Each concrete tasks is implemented by extending this class
- Task execution carried on by periodically calling the tick method from the main loop

LED-SHOW EXAMPLE IN ARDUINO



in the repo:
module-lab-2.3/led-show

LED-SHOW EXAMPLE IN ARDUINO: TASK ABSTRACT CLASS

```
#ifndef __TASK__
#define __TASK__

class Task {

public:
  virtual void init() = 0;
  virtual void tick() = 0;

};

#endif
```

LED-SHOW EXAMPLE IN ARDUINO: BLINK TASK

```
#ifndef __BLINKTASK__
#define __BLINKTASK__

#include "Task.h"
#include "Led.h"

class BlinkTask: public Task {

  int pin;
  Light* led;
  enum { ON, OFF} state;

public:

  BlinkTask(int pin);
  void init();
  void tick();
};

#endif
```

BlinkTask.h

```
#include "BlinkTask.h"

BlinkTask::BlinkTask(int pin){
  this->pin = pin;
}

void BlinkTask::init(){
  led = new Led(pin);
  state = OFF;
}

void BlinkTask::tick(){
  switch (state){
    case OFF:
      led->switchOn();
      state = ON;
      break;
    case ON:
      led->switchOff();
      state = OFF;
      break;
  }
}
```

BlinkTask.cpp

LED-SHOW EXAMPLE IN ARDUINO: THREE LEDS TASK

```
#ifndef __THREELEDSTASK__
#define __THREELEDSTASK__

#include "Task.h"
#include "Led.h"

class ThreeLedsTask: public Task {

  int pin[3];
  Light* led[3];
  int state;

public:

  ThreeLedsTask(int pin0, int pin1,
                int pin2);
  void init();
  void tick();
};

#endif
```

ThreeLedsTask.h

```
#include "ThreeLedsTask.h"

ThreeLedsTask::ThreeLedsTask(int pin0, int pin1,
                             int pin2){
  this->pin[0] = pin0;
  this->pin[1] = pin1;
  this->pin[2] = pin2;
}

void ThreeLedsTask::init(){
  for (int i = 0; i < 3; i++){
    led[i] = new Led(pin[i]);
  }
  state = 0;
}

void ThreeLedsTask::tick(){
  led[state]->switchOff();
  state = (state + 1) % 3;
  led[state]->switchOn();
}
```

ThreeLedsTask.cpp

LED-SHOW EXAMPLE IN ARDUINO: MAIN LOOP

```
#include "Timer.h"
#include "BlinkTask.h"
#include "ThreeLedsTask.h"

Timer timer;

BlinkTask blinkTask(2);
ThreeLedsTask threeLedsTask(3,4,5);

void setup(){
  blinkTask.init();
  threeLedsTask.init();
  timer.setupPeriod(500);
}

void loop(){
  timer.waitForNextTick();
  blinkTask.tick();
  threeLedsTask.tick();
}
```

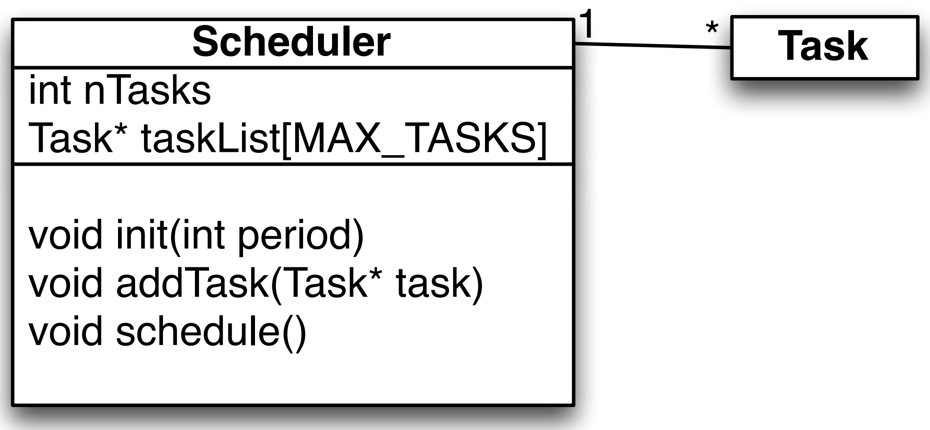
led-show.ino

MANAGING DIFFERENT PERIODS

- As soon as we want to manage tasks/FSMs with different periods, it is necessary:
  - to keep track of the specific period for each task
  - implement a task scheduling so that each task is called with its own period
- To this purpose, we implement a simple cooperative scheduler multi-tasking

SIMPLE COOPERATIVE SCHEDULER

- Scheduler
  - keeps track of a list of tasks to be executed



- Cooperative round-robin strategy
  - the scheduler has a period p equals to the *greatest common divisor* among the periods of the tasks
  - at each period p, it advances the execution of each tasks by calling its method tick
  - internally, it uses a timer to realise the periodic behaviour

A SIMPLE SCHEDULER

```cpp
#ifndef __SCHEDULER__
#define __SCHEDULER__

#include "Timer.h"
#include "Task.h"

#define MAX_TASKS 10

class Scheduler {

  int basePeriod;
  int nTasks;
  Task* taskList[MAX_TASKS];
  Timer timer;

public:

  void init(int basePeriod);
  virtual bool addTask(Task* task);
  virtual void schedule();

};

#endif
```

Scheduler.h


```cpp
#include "Scheduler.h"

void Scheduler::init(int basePeriod){
  this->basePeriod = basePeriod;
  timer.setupPeriod(basePeriod);
  nTasks = 0;
}

bool Scheduler::addTask(Task* task){
  if (nTasks < MAX_TASKS-1){
    taskList[nTasks] = task;
    nTasks++;
    return true;
  } else {
    return false;
  }
}

void Scheduler::schedule(){
  timer.waitForNextTick();
  for (int i = 0; i < nTasks; i++){
    if (taskList[i]->updateAndCheckTime(basePeriod)){
      taskList[i]->tick();
    }
  }
}
```

Scheduler.cpp

CLASS TASK REVISITED

```
#ifndef __TASK__
#define __TASK__

class Task {
  int myPeriod;
  int timeElapsed;

public:
  virtual void init(int period){
    myPeriod = period;
    timeElapsed = 0;
  }

  virtual void tick() = 0;

  bool updateAndCheckTime(int basePeriod){
    timeElapsed += basePeriod;
    if (timeElapsed >= myPeriod){
      timeElapsed = 0;
      return true;
    } else {
      return false;
    }
  }

};
#endif
```

LED-SHOW EXAMPLE

```
#include "Scheduler.h"
#include "BlinkTask.h"
#include "ThreeLedsTask.h"

Scheduler sched;

void setup(){

  sched.init(50);

  Task* t0 = new BlinkTask(2);
  t0->init(500);
  sched.addTask(t0);

  Task* t1 = new ThreeLedsTask(3,4,5);
  t1->init(150);
  sched.addTask(t1);

}

void loop(){
  sched.schedule();
}
```

LED-SHOW EXAMPLE

```
#ifndef __BLINKTASK__
#define __BLINKTASK__

#include "Task.h"
#include "Led.h"

class BlinkTask: public Task {

  int pin;
  Light* led;
  enum { ON, OFF} state;

public:

  BlinkTask(int pin);
  void init(int period);
  void tick();
};

#endif
```

BlinkTask.h

```
#include "BlinkTask.h"

BlinkTask::BlinkTask(int pin){
  this->pin = pin;
}

void BlinkTask::init(int period){
  Task::init(period);
  led = new Led(pin);
  state = OFF;
}

void BlinkTask::tick(){
  switch (state){
    case OFF:
      led->switchOn();
      state = ON;
      break;
    case ON:
      led->switchOff();
      state = OFF;
      break;
  }
}
```

BlinkTask.cpp

LED-SHOW EXAMPLE

```
#ifndef __THREELEDSTASK__
#define __THREELEDSTASK__

#include "Task.h"
#include "Led.h"

class ThreeLedsTask: public Task {

  int pin[3];
  Light* led[3];
  int state;

public:

  ThreeLedsTask(int pin0, int pin1,
                int pin2);
  void init(int period);
  void tick();
};

#endif
```

ThreeLedsTask.h

```
#include "ThreeLedsTask.h"

ThreeLedsTask::ThreeLedsTask(int pin0, int pin1,
                             int pin2){
  this->pin[0] = pin0;
  this->pin[1] = pin1;
  this->pin[2] = pin2;
}

void ThreeLedsTask::init(int period){
  Task::init(period);
  for (int i = 0; i < 3; i++){
    led[i] = new Led(pin[i]);
  }
  state = 0;
}

void ThreeLedsTask::tick(){
  led[state]->switchOff();
  state = (state + 1) % 3;
  led[state]->switchOn();
}
```

ThreeLedsTask.cpp

REMARKS

- Implicit strong discipline to be adopted in task design
  – the execution of the tick method should have a duration always less than the scheduler period
- A different way to use timers and interrupts in schedulers concerns implementing the schedule step (schedule, tick) directly inside the interrupt/timer handler
  – **interrupt-driven schedulers**
  – main drawback
    - task execution would be inside interrupt handlers => constraints and limitations

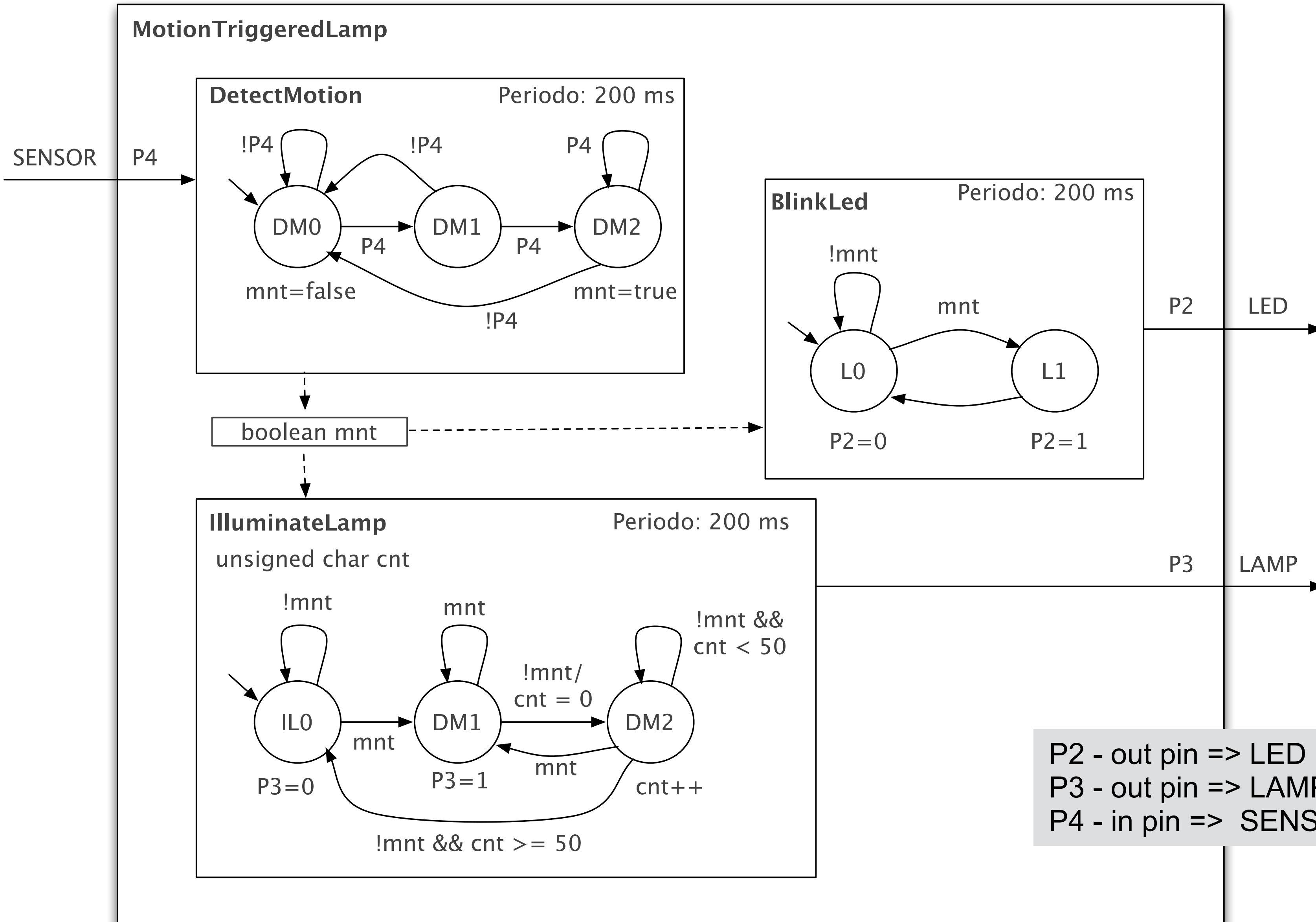
TASK DEPENDENCIES

- Task can have dependencies that requires various form of interactions
  - *temporal dependencies*
    - e.g.: a task T3 can be executed only after tasks T1 and T2
  - *producers/consumers*
    - e.g. a task T1 needs an info produced by a task T2
  - *data-oriented*
    - e.g. task T1 and T2 need to share some data
- In our case we represent and manage these dependencies through **shared variables**

EXAMPLE: MOTION-TRIGGERED LAMP SYSTEM

- Let’s consider an embedded system in which a led is switched on as soon as some movement is detected, in some environment
  - it is equipped with a movement sensor, connected on the input pin P4
  - a movement is detected when it happens that P4 is sampled to be HIGH for two times in sequence, considering a period of 200ms.
  - when a movement is detected, a led connected to the output pin P4 should be switched on and keep the light on for 10 seconds, after the last detection time
  - the system includes a led, connected to P2, that should blink with period 200ms when a movement is detected (and keeps being detected)
- Design with three tasks
  - DetectMotion
  - IlluminateLamp
  - BlinkLed

EXAMPLE: MOTION-TRIGGERED LAMP SYSTEM



P2 - out pin => LED
P3 - out pin => LAMP
P4 - in pin => SENSOR

SHARED VARIABLES, ATOMIC ACTIONS AND RACE CONDITIONS

- Generally speaking, shared variables among concurrent tasks may lead to race conditions, in the case of concurrent reads/ writes
- In our case (cooperative) there could not be races
  - each task tick is executed by the scheduler in sequence
  - the execution is atomic from the point of view of the whole system
- However the execution of multiple ticks of a task can be interleaved with the ones of other tasks
  - this may lead to high-level races, in some cases

EXPLOITING THE SLEEP MODE

- If the period of a scheduler is large enough, we can exploit the timer-driven behaviour and the sleep modality to save power
  - at each cycle, the scheduler after executing the task ticks, it goes to sleep until the next timer tick awakes it for the next cycle
- This is a very important extension for systems that are battery powered
  - nevertheless, in order to be applied, the scheduler period should be large enough to make irrelevant the latency of the microcontroller / MCU board to enter/exit the sleep modality

CPU USAGE AND SCHEDULING

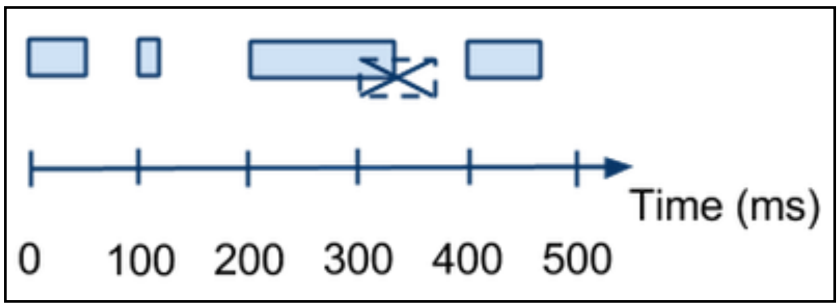
- In synchronous FSM, we assume that actions have zero duration (i.e. instantaneous)
  – or: we abstract from their real duration
- Actually in the real system actions have always a duration and we need to carefully check that the chosen period is compatible with that duration, to avoid problems

OVERRUN EXCEPTION

IMPORTANT

- **Overrun exception -** when the execution time of actions exceeds the period
  - it is called a *timer overrun* if the scheduler is timer-based using interrupts (interrupt-driven schedulers)
    - in that case a new interrupt is generated before the conclusion of the interrupt handler

- Overrun example



sync machine with period = 100ms

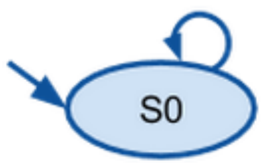
Depending on the state, different actions are executed. At t = 200ms, the duration of actions exceed the period…

CODE ANALYSIS

- Overrun exception can be spotted by doing the analysis of the code (assembly lines)
  - estimating the total duration of actions
  - check if in the worst case this duration exceeds the period
- Example ([PES], p. 101)
  - task CountThree with a single state

CountThree Period: 500 ms

unsigned char cnt;



	Estimated assembly instructions	
cnt = 0;	3	
cnt = cnt + A0;	+ 3	
cnt = cnt + A1;	+ 3	
cnt = cnt + A2;	+ 3	
cnt = cnt + A3;	+ 3	
B1 = (cnt >= 3);	+ 3 + 2	Total: 20

CPU UTILISATION PARAMETER AND WORST-CASE EXECUTION TIME

- The **CPU utilisation parameter** is the percentage of time in which the CPU (microcontroller) is used to execute a task:

  U = (time used for a task by the CPU / total time) * 100%

- The **Worst-Case-Execution-Time** (**WCET**) is the execution time in a period in the worst case
- In the case of multiple states/transitions, we consider the longest sequence of instructions

OVERRUN CASE

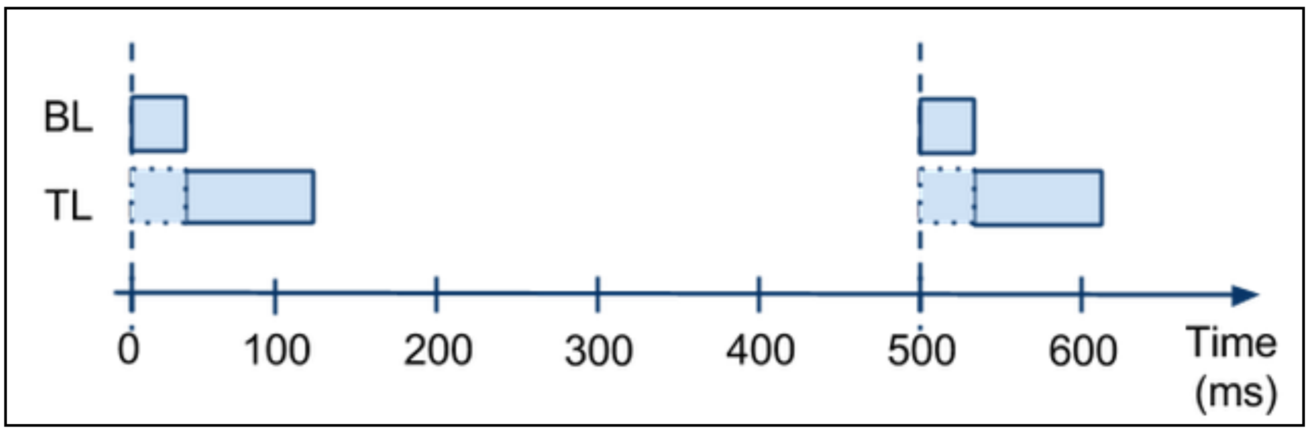
- We compute U. If U is > 100%, then an overrun exception may occur
- In that case, in order to solve/avoid the problem, we can:
  - increase the period of the FSM
  - optimise the sequence of instructions to reduce the WCET
  - break long sequences in smaller sequences of actions
  - use a faster MCU
  - remove functionalities/behaviours from the system

CPU UTILISATION & WCET IN THE CASE OF MULTIPLE TASKS

- The analysis of CPU utilisation factor is even more frequent in systems with multiple tasks
- If tasks have the same period, then the WCET is computed by summing the individual WCETs of the tasks

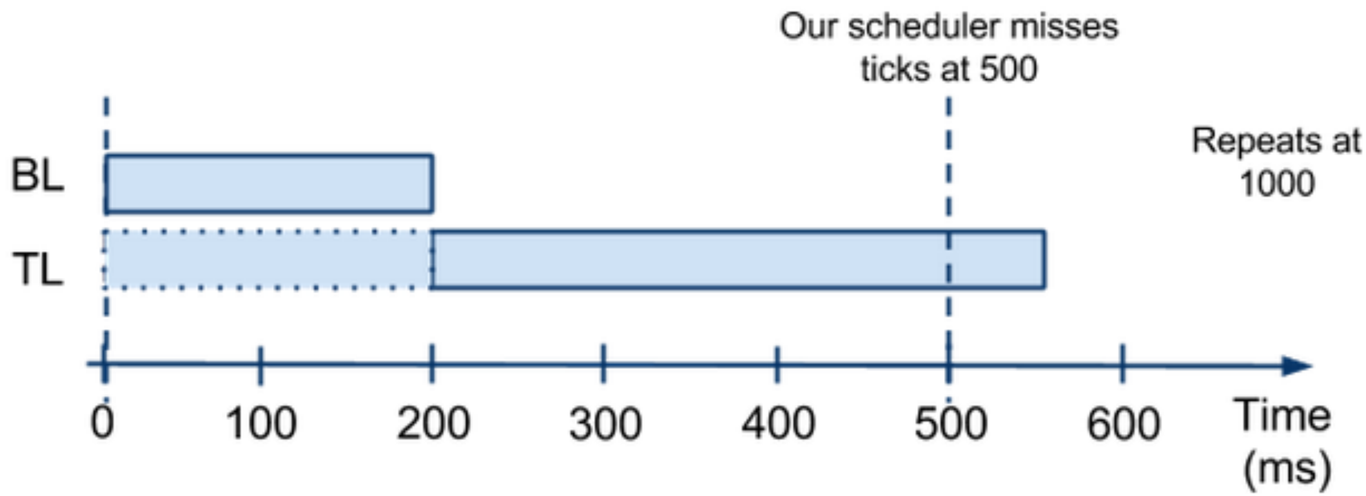
LED-SHOW EXAMPLE

- LedShow example
  - BlinkLed (BL) and ThreeLeds (TL), each with a 500 ms period
  - Analysis example
    - suppose that the MCU would execute 100 instructions/sec, i.e. 0.01 sec/instruction
    - WCET for BL = 3 instructions => 3*0.01 = 0.03 sec
    - WCET per TL = 9 instructions => 9*0.01 = 0.09 sec
    - U = (0.03+0.09)/0.5 (being 0.5 the period) = 24 %



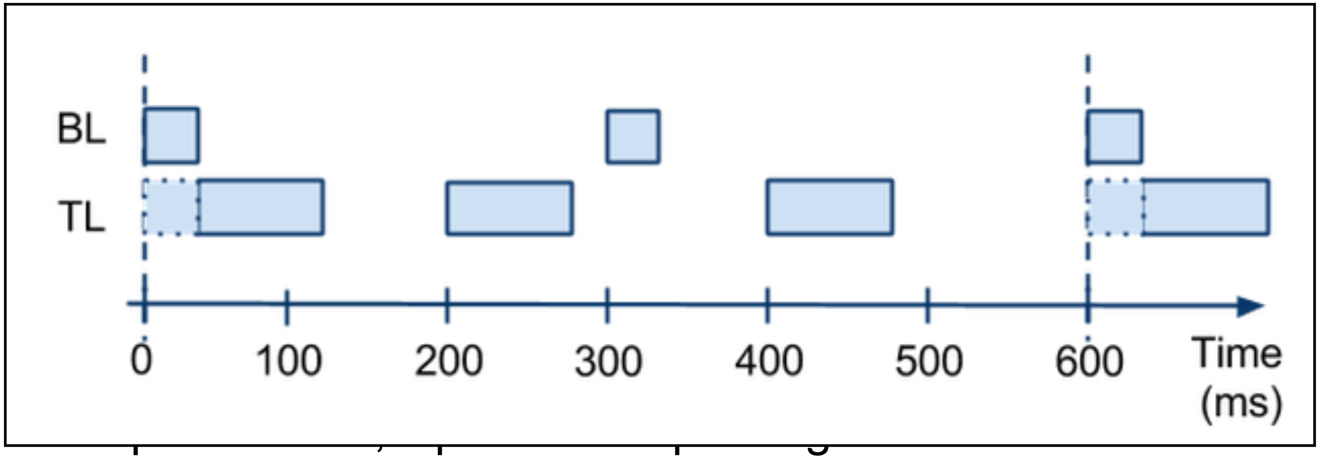
LED-SHOW WITH OVERRUN

- Suppose that the WCET for BL is 200 ms and that the WCET of TL is 350
- Then: U = (0.200+0.35)/0.5 = 1.1 => overrun



WHEN PERIODS ARE DIFFERENT

- In the general case, tasks can have *different periods*
- The WCET can be calculated by considering the **hyper-periods:**
  - periods that are the minimum common multiplier of the periods
- Example: LedShow - BL with period 300ms and TL period 200ms



  - in 600 ms, BL executes 600/300 = 2 times, mentre TL executes 600/200 = 3 times
  - the U parameter in the hyper-period is: (2*20ms+3*90ms)/600ms = 55%

SUMMING UP

- Steps for the U calculation for $T_1 ... T_n$ tasks over a microcontroller M
  - we determine the duration R of a single instruction of the MCU
  - we analyse the code of each T_i, computing its WCET
    - we first compute the max number of instructions in a tick and the we multiply it for R
  - we determine the hyperperiod H, as minimum common multiplier among the periods of all tasks (T1.H, T2.H,…)
  - U is then computed as:

    U = ((H/T1.period)*T1.WCET + (H/T2.period)*T2.WCET+...)/H*100

    - U > 100 => there will be overrun
    - U < 100 =>
      for a single task: overrun cannot occur
      multiple tasks: overrun can still occur

JITTER

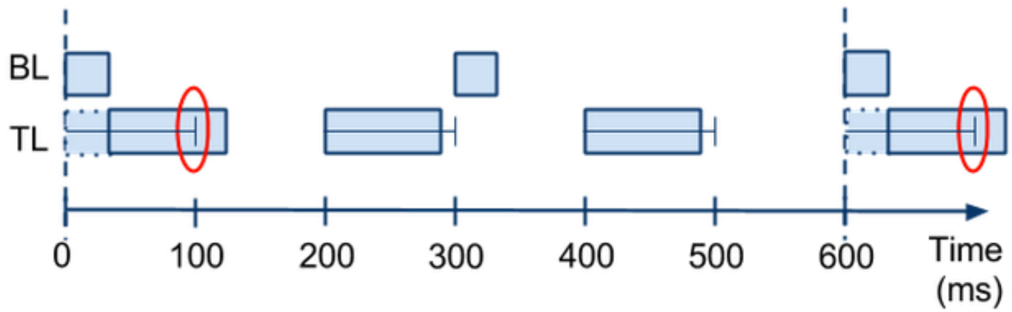
- The **jitter** is the delay that occurs between the time in which a task is ready to be executed and the time in which the task is effectively executed
- Different scheduling strategies may lead to different jitters
  - LedShow example
    - if BL has priority => the jitter of BL is 0, while the jitter of TL is 30ms
    - if TL has priority => the jitter of BL is 90ms, while the jitter of TL is 0
- In general, giving priority to shortest tasks leads to minimise the average jitter
  - in the example: priority to BL

DEADLINE

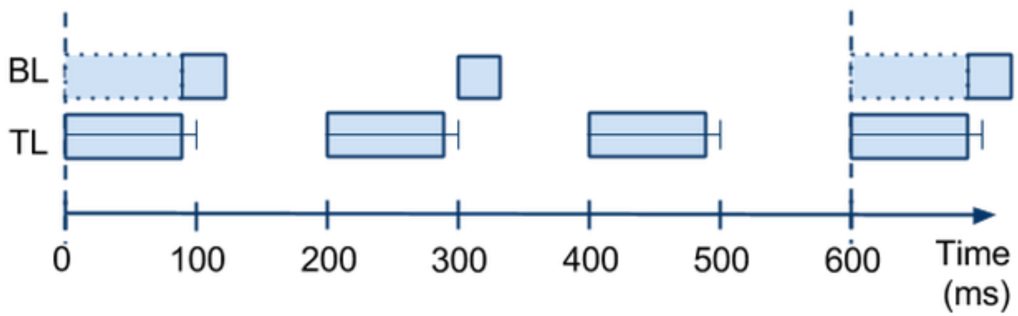
- The **deadline** *(scadenza)* is defined as the interval of time within which a task *must* be executed after being ready to execute (in each period)
- If a task is not executed within its deadline, we have a ***missed-deadline exception*** that can result in a system failure
  - if a deadline is not specified, then it is its period, by default
- The scheduling strategy impacts on jitters and then also on the possibility to have missed-deadline exceptions

EXAMPLE

- LedShow example
  - suppose that BL has period = 300ms, WCET 30ms, and deadline 300ms
  - TL has period = 200ms, WCET 90ms and deadline 100ms
- If we give priority to BL, we have a missed deadline for TL:



- If we give priority to TL:



- In this case, we have a bigger jitter, but we don’t have missed deadline

STATIC AND DYNAMIC PRIORITY SCHEDULING

- The **priority** concerns the order in which the tasks are executed
  - where there are multiple tasks that are ready, then the scheduler selects always the one with the higher priority
  - we can have both static and dynamic approaches in assigning the priority to tasks
    - static => priority assigned at design time and do not change at runtime
    - dynamic => priority assigned dynamically, according different strategies
- Key aspect in real-time OS (RTOS)
  - next modules

PERIODIC AND SPORADIC TASKS

- The tasks considered so far are **periodic**
  - they are executed periodically by the scheduler, given a period p, and this period corresponds to the period of the synchronous FSM
  - typically periodic tasks are static, that is: launched when the system is started and never removed
- Besides periodic tasks we can have **sporadic or aperiodic tasks**
  - i.e. tasks that are executed at arbitrary times

MANAGING APERIODIC TASKS

- Basic capabilities of a scheduler managing aperiodic tasks
  - dynamic insertion and removal of tasks
  - dynamic assignments of priorities
- Simple examples (in lab)
  - aperiodic task simulation with periodic tasks with “idle” state
    - simplest approach, not super efficient
  - tasks with active/not active meta-state
    - selected by a scheduler only if active
    - can be activated/de-activated by other tasks too

BIBLIOGRAPHY

- [PES] “Programming Embedded Systems - An introduction to Time-Oriented Programming”. Vahid et al.