# Sistemi in tempo reale
# Anno accademico 2009 - 2010
## Cambi di modo

Giuseppe Lipari
`http://retis.sssup.it/~lipari`

Scuola Superiore Sant'Anna – Pisa

December 2, 2011

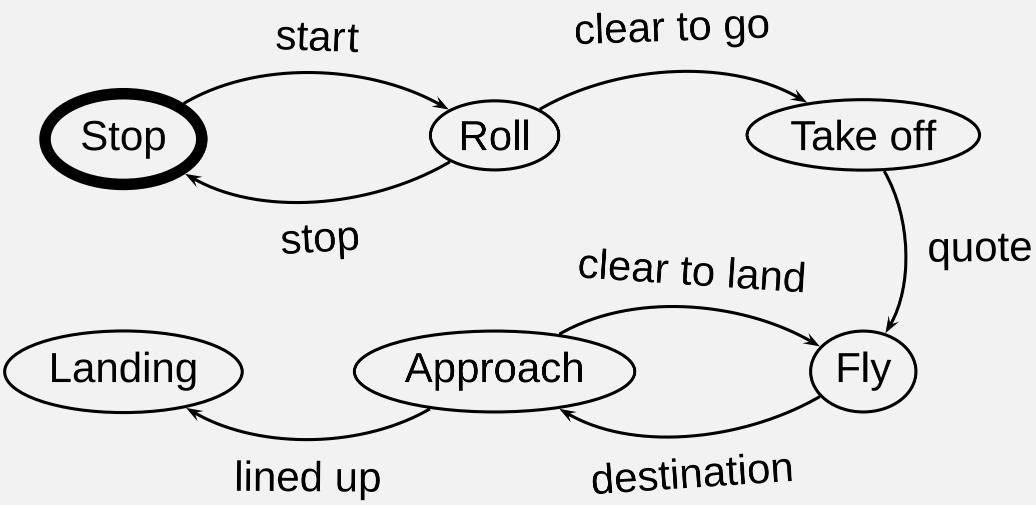
---

## Modes

- A real-time system can have different working *modes*
- Each mode defines the same system under different working conditions;
- Example: airplane
  - Typical modes are take-off, cruise, and landing;
  - During each mode, the system has different control goals; and it must run different control algorithms.
- Example: elevator
  - Clearly, an elevator goes across different states: idle, opening/closing doors, moving, etc.
  - Depending on the abstraction level, each mode can be sub-divided into internal modes. For example, when a elevator moves, we can distinguish between acceleration, stable state, deceleration. Also, we may need to distinguish between moving up and down

## Modes and transistions

- Modes can be represented by a state machine. For example, consider the previous example of airplane control:



---

## Modes and transistions

- A mode is a node in the diagram (a *state*)
  - Each mode is associated with a set of periodic or sporadic tasks
  - Different modes may have different task sets, or tasks with different characteristics
  - When the mode is active, the corresponding tasks are executed (steady state)
- A transition is an edge between two nodes:
  - A transition happens when certain conditions are verified;
  - For example, a user command, an external condition on the altitude or temperature, the landing of the airplane, etc.
- Upon the occurrence of a transition:
  - terminate all tasks that are in the current mode and will not be active in the new mode;
  - optionally, call a *transition function*;
  - activate the new set of tasks to be executed.

## Modes and tasks

- To implement modes:
  - One *manager task* that identifies when modes must be changed;
  - One global variable that identifies the current working mode (currmode);
- Modes can be implemented in two basic ways;
  1. **Type 1** A fixed set of tasks for all the modes; each task can execute different algorithms depending on the current mode;
  2. **Type 2** A different set of tasks for each mode.
- Of course, it is also possible to mix the two implementations.

---

## Implementation type 1

- **Type 1:** In this case, each task executes different code depending on the mode
- Suppose we synchronize at the beginning of the task instance. The code for each task is something like the following:

```
while(1) {
  switch (currmode) {
  M1 : // control algorithm
       // for mode 1
       break;
  M2 : // control algorithm
       // for mode 2
       break;
  default : break;
  }
  task_endcycle();
}
```

# Implementation type 2

- In this case, each task can be active only in a subset of the modes.
- Define $\mathcal{T}_1$ the tasks active in mode 1, and $\mathcal{T}_2$ the task active in mode 2.
  - Suppose that the list of modes for which a task is active are stored in 2-dimension array modes[task][mode].
  - If task i is active in mode currmode, then modes[i][currmode] is true, otherwise it is false.
- Typical code of the task;

```
while (1) {
  // control algorithm
  if (!mode[i][currmode]) task_disable();
  task_endcycle();
}
```

  - The primitive task_disable() suspends the periodic activations; they will be enabled again by an explicit task_activate() on the current task

---

# Type 1 vs. type 2

- In type 1, all tasks have the same parameters (period and priority) in every mode;
  - Implementation looks simpler, but does not scale well
  - The task code depends on the number and type of modes
  - From a software engineering point of view, the task code cannot be re-used easily
- In type 2, we have different tasks for different modes: therefore, from one mode to the other, we can change both the period, the priority and the computation time of a task
  - Taski mplementation is simple and scales well
  - The code of each task is self-contained and does not depend on the number and types of the modes in the system
  - Therefore, we can easily reuse this task.
  - However, the mode manager task is more complex, as it must take care of deactivating/activating tasks in the proper way

# Problems with mode changes

- There are several problems the designer must deal with when designing a multi-mode real-time system;
- The main problem is what happens during the transition between two modes. In particular, we must deal with
  1. Schedulability analysis
     - The system must remain schedulable across the transition
  2. Periodicity
     - Tasks that are present in both modes must continue to execute periodically, as nothing happened in the meanwhile
  3. Consistency of variables
     - Resource must remain consistent during mode change
     - We must take care of adjusting variables that are shared between old mode and new mode tasks (hybrid systems)
  4. Promptness
     - The transition should happen in the shortest possible interval of time
- Now we start dealing with problem 3.

# Consistency

- Clearly, we cannot change the control algorithm at any arbitrary point while the algorithm is executing;
  - A control algorithm updates its internal state variables while executing;
  - we must ensure that the state variable does not remain in an inconsistent state when we change mode;
  - the same happens if the task is accessing a shared resource with a critical section protected by a mutex; we cannot interrupt it and change algorithm, otherwise the mutex remains locked!
- This means that the change of control algorithm must be *synchronized* with appropriate *checkpoints*;
  - A checkpoint is a point in the code when it is *safe* to interrupt the algorithm, maintaining the consistency of the data;
  - The “easiest” checkpoints are at the beginning and at the end of the task instance.

# Implementation type 1

- Checkpoint at the job boundary
  - The task cannot change mode while is executing. It can only change mode at the beginning of one of its istance;
  - In this way we guarantee consistency of internal and external variables (state variables and output variables).
  - The only problem is that, if the task execution time is large, we must wait for the job to complete before we can complete the mode change
    - the mode change delay can be large
- To introduce other checkpoints, we could complicate the code:
  - divide each control algorithm in different blocks
  - check the change of mode at the end of every block.
- The code becomes much more complex!

---

# Implementation type 2

- In this case, the implementation of the mode change is outside the task
  - the mode manager activates and deactivates the tasks
  - We must guarantee that the mode manager does not *kill* a task while it is executing in the middle of a control task update! (*asynchronous cancellation*)
- Therefore, we have to implement a specific *protocol* to *synchronize* the mode manager task with the control task
  - The mode manager sends *a signal* to the control task and waits for it to respond
  - The control task will respond (and finish its execution) when reaching a propose *checkpoint*

## Mode manager

- The “mode manager” task manages *Mode Change Requests* (MCRs)
- The mode manager can be a periodic or aperiodic task;
  - In the first case (*periodic*), it periodically observes the state of the system and of the external variables and decides if a mode change must be performed;
  - In the second case (*aperiodic*), it is attached to an external interrupt (external condition) or it is explicitely activated by another task;
- The mode manager implements the state machine and controls transition between modes.
- From now on, we consider only **type 2** implementations.

## Implementation type 2: manager

- The task manager is structured as follows

```
while (1) {
  if (modeIsChanged()) {
    old_mode = curr_mode;
    curr_mode = getNewMode();
    transition(old_mode, new_mode);
    for (i=0; i < NTASK; i++) {
      if (mode[i][curr_mode] && !mode[i][old_mode])
      task_activate(tid[i]);
    }
  }
  task_endcycle();
}
```

- The manager is a periodic task that periodically checks for occurrence of mode changes.
- It waits for a change of mode (function modeIsChanged())
- When it happens, deactivates old mode tasks and performs transition functions, then activates all tasks belonging to the new mode and not active in the old mode.

# Transitions

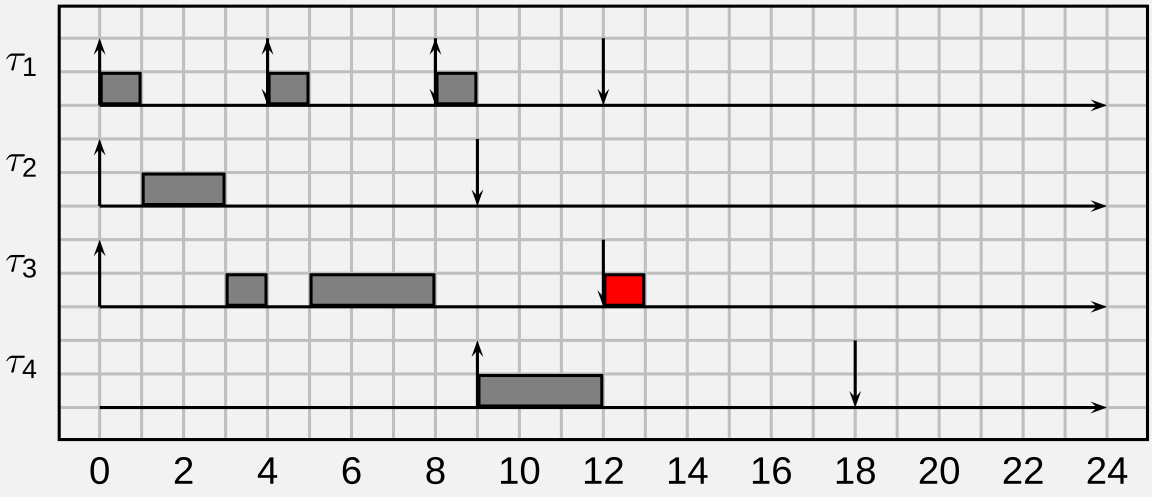
- Suppose the system must change from mode 1 to mode 2.
- To ensure a *smooth* transition between two modes, the states of control algorithms of mode 2 must be properly initialized;
- In other words, the initial conditions of mode 2 depend on the state conditions of mode 1.
    - Suppose, as an example, that we want to guarantee continuity of the signal and of the first derivative of the signal.
    - The, the internal conditions of the controller for mode 2 must be set so to ensure these two conditions;
- From a software point of view, for each transition we must call a set of functions to adjust the initial conditions of all control algorithms
- This can be done, for example, by specifying an appropriate *entry* behavior for the states

# Scheduling analysis

- Another important problem is schedulability:
- Suppose we are changing from mode 1 to mode 2, and that $\mathcal{T}_1$ is the set of tasks active in mode 1 and $\mathcal{T}_2$ is the set of tasks that are active during mode 2.
    - Set $\mathcal{T}_1 \setminus \mathcal{T}_2$ is the set of tasks that *leave the mode*;
    - Set $\mathcal{T}_2 \setminus \mathcal{T}_1$ is the set of tasks that *enter the mode*.
- It is important to guarantee that the system continues to be schedulable;
- Even if $\mathcal{T}_1$ and $\mathcal{T}_2$, each one considered in isolation, are schedulable, if the transistion is not done properly, some deadline could be missed during the transitory.

## Example of deadline miss during transition

- Consider $\mathcal{T}_1 = \{\tau_1, \tau_2, \tau_3\}$ and $\mathcal{T}_2 = \{\tau_1, \tau_2, \tau_4\}$ with:
  - $\tau_1 = (1, 4)$, $\tau_2 = (2, 9)$, $\tau_3 = (5, 12)$, and $\tau_4 = (3, 9)$
- Transition starts at time $t = 9$
  - Task $\tau_4$ must execute instead of task $\tau_3$ from time $t = 9$



## Mode Change protocols

- There are many ways to avoid this problem
  1. We can wait for the first idle time in the system (*idle time protocol*)
     - At that point, all tasks have completed their execution,
     - So we can safely deactivate the old-mode tasks and activate the new-mode ones
     - Old-mode tasks cannot influence new-mode tasks
     - Advantages: simplicity, does not require a specific schedulability analysis
     - Drawbacks: the transition delay can be large
  2. We can introduce new tasks as soon as it is possible, if the schedulability is guaranteed
     - old-mode task may complete their last instance after the MCR
     - new-mode tasks must be activated with a proper offset with respect to the MCR, so that no deadline is missed
     - Advantages: reduce the transition delay
     - Drawbacks: require schedulability analysis, may be difficult to implement

# Idle-Time protocol

- Implementation strategies:
  1. Low-priority mode manager
     - Mode manager running in low priority mode (thus executing only when all other tasks have finished
     - It activates deactivates old mode tasks, executes transition code, activates new mode tasks
     - Tasks must check their re-activation before starting
  2. dual priority mode manager
     - The mode runs at highest priority
     - At MCR, first deactivates old-mode tasks, then it goes to low-priority
     - When executing again, check completion of old-mode tasks, hence executes transition code and activates new-mode tasks
     - Tasks have to check their deactivation before sleeping