Course Outline

- Introduction
  - Characteristics of RTS
  - Real Time Operating Systems (RTOS)
  - OS support: scheduling, resource handling
  - Real Time Programming Languages
  - Language support, e.g. Ada tasking
  - Scheduling and Timing Analysis of RT Software
  - Design and Validation
  - Modeling, Verification and Testing
  - Reliability and Fault Tolerance
  - Fault tolerant, failure recovery, exception handling
  - Distributed real time systems
  - Real Time Communication: CAN Bus

Overall Structure of RT Systems

- Hardware (CPU, I/O device etc)
  - a clock!
- A real time OS (function as standard OS, with predictable behavior and well-defined functionality)
- A collection of RT tasks/processes (share resources, communicate/synchronize with each other and the environment)

Components of RT Systems

General-Purpose vs. Embedded RT Computer Systems

Characteristics of a RTS

- Large and complex — vary from a few hundred lines of assembler or C to 20 million lines of Ada estimated for the Space Station Freedom
- Concurrent control of separate system components — devices operate in parallel in the real-world; better to model this parallelism by concurrent entities in the program
- Facilities to interact with special purpose hardware — need to be able to program devices in a reliable and abstract way
- Mixture of Hardware/Software: some modules implemented in hardware, even whole systems, SoC

Characteristics of a RTS (ctn.)

- Extreme reliability and safety — embedded systems typically control the environment in which they operate; failure to control can result in loss of life, damage to environment or economic loss
- Guaranteed response times — we need to be able to predict with confidence the worst case response times for systems; efficiency is important but predictability is essential
Classification of RTS’s

- **Hard real-time** — systems where it is absolutely imperative that responses occur within the required deadline. E.g. Flight control systems, automotive systems, robotics etc.

- **Soft real-time** — systems where deadlines are important but which will still function correctly if deadlines are occasionally missed. E.g. Banking system, multimedia etc.

A single system may have both hard and soft real-time Subsystems. In reality many systems will have a cost function associated with missing each deadline.

### Example: a Car Controller

Activities of a car control system. Let

1. \( C \) = worst case execution time
2. \( T \) = (sampling) period
3. \( D \) = deadline

- Speed measurement: \( C=4\text{ms}, T=20\text{ms}, D=5\text{ms} \)
- ABS control: \( C=10\text{ms}, T=40\text{ms}, D=40\text{ms} \)
- Fuel injection: \( C=40\text{ms}, T=80\text{ms}, D=80\text{ms} \)
- Other software with soft deadlines e.g audio, air condition etc

Construct a controller meeting all the deadlines!

### Programming the car controller (1)

<table>
<thead>
<tr>
<th>Process Speed</th>
<th>Process ABS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop</td>
<td>Loop</td>
</tr>
<tr>
<td>read sensor, compute, display...</td>
<td>Read sensor, compute, react</td>
</tr>
<tr>
<td>sleep (0.02)/period/</td>
<td>sleep(0.04)</td>
</tr>
<tr>
<td>End loop</td>
<td>End loop</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process Fuel</th>
<th>Soft RT Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop</td>
<td>Loop</td>
</tr>
<tr>
<td>read data, compute, inject ...</td>
<td>read temperature</td>
</tr>
<tr>
<td>sleep(0.08)</td>
<td>el hils, stereo</td>
</tr>
<tr>
<td>End loop</td>
<td>....</td>
</tr>
</tbody>
</table>

### Programming the car controller (2)

<table>
<thead>
<tr>
<th>Process Speed</th>
<th>Process ABS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop</td>
<td>Loop</td>
</tr>
<tr>
<td>next := get-time + 0.02</td>
<td>next:=get-time + 0.04</td>
</tr>
<tr>
<td>read sensor, compute, display...</td>
<td>Read sensor, compute, react</td>
</tr>
<tr>
<td>sleep until next</td>
<td>sleep until next</td>
</tr>
<tr>
<td>End loop</td>
<td>End loop</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process Fuel</th>
<th>Soft RT Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop</td>
<td>Loop</td>
</tr>
<tr>
<td>next:=get-time + 0.08</td>
<td>read temperature</td>
</tr>
<tr>
<td>read data, compute, inject ...</td>
<td>el hils, stereo</td>
</tr>
<tr>
<td>sleep until next</td>
<td>....</td>
</tr>
<tr>
<td>End loop</td>
<td>End loop</td>
</tr>
</tbody>
</table>

Any problem?

- We forgot the execution times ...

  e.g. Process speed:

  \[ 20\text{ms} = \text{execution time} + \text{sleep}(X) \]
What is the problem now?

We don’t know if the deadlines are met!

- We need to know the execution times
- We need to do schedulability analysis
- We need to construct a schedule
- We need to implement/buy an RT OS kernel

Programming the car controller (3)

A feasible Schedule!

Design and Implementation of RT Systems

- Specification and Analysis
  - Requirement Specs
  - Design Specs
- Implementation
  - Hardware platform
  - OS kernel
  - Design/Programming Languages
  - Code generation vs. coding
- Validation
  - Verification
  - Testing

Real-time Programming Languages

- Assembly languages
- Sequential programming languages — e.g. Pascal, C.
  - Both normally require operating system support.
- High-level concurrent languages e.g. Concurrent Pascal, Ada, Modula-2, RT Java.
  - No/less operating system support!
- Other (design/prog) languages: Esterel, Lustre, SystemC, UML (modeling and Code generation)
- We will consider:
  - Ada 95 and C

Challenges in RT Systems Design

- Predictability: able to predict the future consequences of current actions
- Testability: easy to test if the system can meet all the deadlines
- Cost optimality: e.g. Energy consumption, memory blocks etc

Main desirable properties of RT Systems (2)

- Maintainability: modular structure to ease system modification
- Robustness: must not collapse when subject to peak load, exception, manage all possible scenarios
- Fault tolerance: hardware and software failures should not cause the system to crash - function down-grading
Predictability: the most important one

- The system behaviour is known before it is put into operation!
  e.g. Response times, deadlock freedom etc

Difficult (impossible?) to achieve!

Difficult to achieve predictability: Hardware & RTOS

- Cache sharing, processor pipelines, DMA ...
- Interrupt handling may introduce unbounded delays
- Priority inversion (low-priority tasks blocking high-priority tasks)
- Memory management (static allocation may not be enough, dynamic data structures e.g. Queue), no virtual memory
- Communication delays in a distributed environment

Difficult to achieve predictability: RT Tasks:

- Difficult to calculate the worst case execution time for tasks (theoretically impossible, halting problem)
  - Avoid dynamic data structures
  - Avoid recursion
  - Bounded loops e.g. For-loops only
- Complex synchronization patterns between tasks: potential deadlocks (formal verification)
- Multi-tasking, tasks that share resources

Problems to solve ...

- Missing deadlines (!)
- Deadlocks/livelocks
- Uncontrolled exception (ARIAN 5)
- Priority inversion (the Mars project)
- Uncontrolled code size, cost, ...
- Non-determinism and/or Race condition
- Overloaded

Problems to solve ...

- Missing deadlines (!)
- Deadlocks/livelocks
- Uncontrolled exception (ARIAN 5)
- Clock jitter (the golf war, Scud missile)
  - 57 microsec/min, 343ms/100 hours
  - 687 meters
- Priority inversion (the Mars project)
- Uncontrolled code size, cost, ...
- Non-determinism and/or Race condition
- Overloaded