INFO-0064 Embedded systems

Exercise session 4
Software architectures

Kevin Hogan
kevin.hogan@ulg.ac.be
Information

Slides and code examples available on:

http://www.montefiore.ulg.ac.be/~khogan/
Planning for today

• Real-time constraints
• Software architectures
• Situations
Real-time constraints

• **Soft** real-time constraints
  ➔ Timings should be verified. If they are not verified, performances are lowered.
  example: LCD display

• **Hard** real-time constraints
  ➔ Timings must be verified! If they are not verified, fatal failure.
  example: airbags in car
Software architectures

• Round-Robin
• Round-Robin with interrupts
• Function-queue-scheduling
• Real-time operating system
Round-Robin

• Every tasks are executed one after another
• No interrupts
• Simple implementation
• Maximal latency = duration of the entire loop
• Problem if there is a task with a long time of execution
• Add new tasks can affect the correctness of the system

```c
void main(void){
while (TRUE){
    if (!! task 1 is ready){
        !! operations of task 1;
    }

    if (!! task n is ready){
        !! Operations of task n;
    }
}
}...
Round-Robin: example

- Electronic thermometer

Sensor: PT100 (100 Ohm at 0 °C & 0.3 Ohm/°C)

Display in Celsius [°C] or Fahrenheit [°F] depending on the value of a switch

```c
void main(void){
    while (TRUE){
        !! resistor measurement
        !! switch read
        !! convert measure to the chosen temperature
        !! display update
    }
}
```
Round-Robin with interrupts

• Non-urgent tasks are executed in a robin-round fashion
• Urgent tasks are executed by interrupt routines
• Urgent tasks can be prioritized
• Non-urgent tasks have the same priority
• **Shared-data** problems arise
• Time response for a non-urgent task:
  duration of the main loop + interrupts
Round-Robin with interrupts

```c
volatile BOOL ready1 = 0, ..., readyn = 0;

interrupt void urgent1(void){
    !! urgent operations of task 1;
    ready1 = 1;
}

interrupt void urgentn(void){
    !! urgent operations of task n;
    readyn = 1;
}

int main(void){
    while (TRUE){
        if (ready1){
            !! non-urgent operations of task1;
            ready1 = 0;
        }
        ...
        if (readyn){
            !! non-urgent operations of taskn;
            readyn = 0;
        }
    }
}
```
Round-Robin with interrupts: example

• Propeller clock

volatile BOOL next_image = 0;

interrupt void Timer0(void){
    !! Update current pixel line;
}

interrupt void CompleteRev(void){
    !! Update image;
next_image = 1;
}

void main(void){
    init();
    while (TRUE){
        If (next_image){
            !! Compute next image
next_image = 0;
        }
        !! Check switches
        !! Select image to display
    }
}
Waiting-queue-scheduling

• Tasks are separated in urgent and non-urgent tasks
• Interrupt routines execute urgent tasks and add the non-urgent ones to a waiting queue
• The main loop executed the waiting tasks in the queue one after an other with any priority scheme
• Waiting-queue implementation can be tricky
• Low priority tasks might never be executed
#include "queue.h"

static volatile queue waiting_queue;

interrupt void urgent1(void){
    !! urgent operations of task 1;
    !! add task1 to waiting_queue;
}

interrupt void urgentn(void){
    !! urgent operations of task n;
    !! add taskn to waiting_queue;
}

void main(void){
    !! initialize waiting_queue with an empty content;
    while (TRUE){
        while (!queue_is_empty(waiting_queue)){
            !! extract a function from waiting_queue;
            !! execute this function;
        }
    }
}

void task1(void){
    !! non-urgent operations of task 1;
}

void taskn(void){
    !! non-urgent operations of task n;
}
Real-time operating system (RTOS)

• Interrupt routines execute urgent tasks and signal that non-urgent tasks are ready to be executed
• The operating system invokes dynamically the non-urgent tasks
• The OS is able to suspend the execution of a task to allow another one to be executed
• The OS handles communication between tasks

```c
#include "signal.h"

interrupt void urgent1(void){
   !! urgent operations of task 1;
   !! send signal 1;
}

interrupt void urgentn(void){
   !! urgent operations of task n;
   !! send signal n;
}
```
Real-time operating system (RTOS)

• Data communication between tasks/interrupts must be coordinated
• Complex implementation (but you don’t have to do it yourself)
• Robustness against modifications
• The OS uses a certain portion of the processor resources (2% to 4%)

```c
void task1(void){
    !! wait for signal 1;
    !! non-urgent operations of task 1;
}

void taskn(void){
    !! wait for signal n;
    !! non-urgent operations of task n;
}

void main(void){
    !! initialize the operating system;
    !! create and enable tasks;
    !! start task sequencing;
}
```
## Software architectures comparison

<table>
<thead>
<tr>
<th></th>
<th>Priority</th>
<th>Max response time</th>
<th>Robustness against modifications</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Round-robin</strong></td>
<td>None</td>
<td>Sum of tasks execution time</td>
<td>Bad</td>
<td>Very simple</td>
</tr>
<tr>
<td><strong>Round-robin with interrupts</strong></td>
<td>Interrupts</td>
<td>Interrupts + sum of tasks execution time</td>
<td>Good for interrupts, bad for tasks</td>
<td>Concurrent access</td>
</tr>
<tr>
<td><strong>Function queue scheduling</strong></td>
<td>Interrupts &amp; tasks</td>
<td>Interrupts + longest tasks execution time</td>
<td>Quite good</td>
<td>Concurrent access; function queue</td>
</tr>
<tr>
<td><strong>Real-time operating system</strong></td>
<td>Interrupts &amp; tasks</td>
<td>Interrupts</td>
<td>Very good</td>
<td>Very complex but within OS</td>
</tr>
</tbody>
</table>
Situations

We want to obtain the greatest amount of control over the system response time.

➡️ Select the simplest architecture that will meet your response requirements
Situations

Simple video game (such as PONG)

What has to be considered?

• Display the image (PAL signal: 625 lines @ 50Hz)
• Game management (i.e. compute the position of the ball)
• Game control (buttons, controller)
Situations

Vending machine

*What has to be considered?*

- Display information
- Handle buttons & coin acceptor
- Check sensors
- Motors control
Situations

Vehicle embedded electronics

*What has to be considered?*

- Sensor measurement (pedal, speed, switches, ...)
- Engine control (ignition, turbo, injection, cooling system, ...)
- Cruise-control
- Display
- GPS