What is a real-time system?

• Lots of products contain embedded computers, e.g., cars, planes and medical equipment

• In such systems it's important to deliver correct functionality on time

• **Non-real-time systems**
  ▪ Correct function if produced result is correct

• **Real-time systems**
  ▪ Correct function if produced result is correct and **delivered on time**

Figure taken from Issovíc, D.: Real-time systems, basic course
Misconceptions about real-time systems

• Real-time ≠ fast: rather predictable than fast
  ▪ Fast calculation: minimize *average response time* for entire system
  ▪ Real-time: fulfill *individual time constraints* for each activity
  ▪ “A *man drowned in a river with an average depth of 20 centimeters*”
In real-time systems late data = bad data

- **Example:** Medical equipment must detect changes in the patient and respond on time...

Figure taken from Issovic, D.: Real-time systems, basic course
In real-time systems late data = bad data

- **Example:** ...otherwise...

Response to a critical event must be given on time

Figure taken from Issovic, D.: Real-time systems, basic course
• **Example:** An air bag must not be inflated too late, nor too early!

Real-time ≠ fast!
RT systems are usually safety-critical

- Example: Anti-lock Braking System (ABS)

1. Brake pedal pushed
2. Pressure passed to the brake fluid
3. Wheel disc brakes squeezed
4. If the brake pedal is pushed too hard, the wheel will lock → a sensor detects this and notifies the controller
5. Controller releases the pressure on the discs by releasing some brake fluid in a container
6. The fluid is pumped back to repeat the pressure on the discs

7. Entire process is repeated about 15 times/sek

Figure taken from Issovic, D.: Real-time systems, basic course
RT is not only safety-critical

Figure taken from Issovic, D.: Real-time systems, basic course
What is a real-time system?

“A real-time system is a system that reacts upon outside events and performs a function based on these and gives a response within a certain time. Correctness of the function does not only depend on correctness of the result, but also the timeliness of it.”

Real-time means that the system must be synchronized with the environment. The controlled process dictates the time scale (some processes have demand on response at second-level, others at milli- or even microsecond level).

Figure taken from Isovic, D.: Real-time systems, basic course
Interaction with the environment

Figure taken from Issovic, D.: Real-time systems, basic course
Example: an electrical engine

Figure taken from Isovic, D.: Real-time systems, basic course
What is real-time in this example?

• When computer system controls the speed, it has to:
  1. Observe the process, i.e., read the sensors (sampling)
  2. Decide what has to be done, i.e., execute the control algorithm
  3. Give a new control signal to the process via the actuator (actuation)

Figure taken from Issovic, D.: Real-time systems, basic course
• Temporal demands can be divided into two parts
  1. **How fast** do we have to process input and respond to the controlled process (delay)
  2. **How often** do we have to sample the environment to get a sufficiently good view of it (period)
Example: controlling the elevation with an autopilot

a) If we check the elevation often enough, we will discover changes in the terrain and have enough time to make corrections.
b) Sampling done too far apart ⇒ catastrophic consequences.

- Sampling too often means waste of resources (CPU). If too much time is spent in controlling the elevation, we might miss controlling something else.
- Thus it is important to distribute the time and resources in a good way.
Characteristics of real-time systems

- Close coupling to process – I/O
- **Predictably** fast handling of events
- Handling of **several system activities at the same time**
- Possibility to **prioritize** among system activities
- Design for **peak load** and fault tolerance
- Configuring of program execution as **cyclic** or **event triggered**
- Internally hold a view of the process being controlled, e.g., its different states
Classification of real-time systems

• The criticalness of the timing may be characterized by a cost function

Regular applications

Multimedia applications
User interaction

Safety-critical applications
Tasks and scheduling

- RT system is typically responsible for a number of concurrent activities
- Each activity has its deadline
  - Activity ~ Task
- The key problem in RT systems is how to execute the tasks such as each meets the deadline
What influences timeliness?

- Hardware
- Application
- Operating system
- Scheduler

Simple control loops

More complex applications with more tasks
Real-Time in Operating Systems

- “The ability of the operating system to provide a required level of service in a bounded response time.”
  - POSIX Standard 1003.1

- RTOS must:
  - Be deterministic
    - the worst-case execution time of each of its system calls is calculable
  - Have guaranteed worst-case interrupt latency
    - basically how long interrupts may be disabled
  - Have guaranteed worst-case context-switch times
    - how much is spent in the context-switch routine
Typical RTOS Constituents

- Process (task) management
  - Scheduler
  - Synchronization mechanism
    - Interprocess communication (IPC)
    - Semaphores
- Memory management
- Interrupt service mechanism
- I/O management, HAL
- Development Environments
- Communication subsystems (Option)
- Board Support Packages (BSP)
RTOS Examples

• FreeRTOS
• RTEMS
• SHaRK
• eCos, eCosPro
• RTLinux, RTAI
• LynxOS
• PikeOS
• QNX
• VxWorks
• Windows CE
• MicroC/OS-II
Working with RTOS

• Typically written in C or Ada

• Application defines a set of tasks with real-time properties

• In the case of embedded systems
  ▪ RTOS is minimal in size and heavily configurable
  ▪ Application is linked together with RTOS
  ▪ Resulting binary image is uploaded to the device
RTOS API Example

• Task manipulation in FreeRTOS
  ▪ `portBASE_TYPE xTaskCreate( pdTASK_CODE pvTaskCode, const portCHAR * const pcName, unsigned portSHORT usStackDepth, void *pvParameters, unsigned portBASE_TYPE uxPriority, xTaskHandle *pvCreatedTask );`
  ▪ `void vTaskDelete( xTaskHandle pxTask );`
  ▪ `void vTaskDelay( portTickType xTicksToDelay );`
  ▪ `void vTaskDelayUntil( portTickType *pxPreviousWakeTime, portTickType xTimeIncrement );`
  ▪ `unsigned portBASE_TYPE uxTaskPriorityGet( xTaskHandle pxTask );`
  ▪ `void vTaskPrioritySet( xTaskHandle pxTask, unsigned portBASE_TYPE uxNewPriority );`
  ▪ `void vTaskSuspend( xTaskHandle pxTaskToSuspend );`
  ▪ `void vTaskResume( xTaskHandle pxTaskToResume );`
  ▪ `void xTaskResumeFromISR( xTaskHandle pxTaskToResume );`
Task synchronization and communication

• Synchronization
  ▪ Semaphores
  ▪ Locks

• Communication
  ▪ Tasks typically share memory
  ▪ Message queues
struct AMessage
{
    portCHAR ucMessageID;
    portCHAR ucData[ 20 ];
} xMessage;

xQueueHandle xQueue;

// Task to create a queue and post a value.
void vATask( void *pvParameters )
{
    struct AMessage *pxMessage;

    // Create a queue capable of containing 10 pointers to AMessage structures.
    // These should be passed by pointer as they contain a lot of data.
    xQueue = xQueueCreate( 10, sizeof( struct AMessage * ) );
    if( xQueue == 0 )
    {
        // Failed to create the queue.
    }

    // ...
Example: Receiving a Message

// Send a pointer to a struct AMessage object. Don't block if the
// queue is already full.
pxMessage = & xMessage;
xQueueSend( xQueue, ( void * ) &pxMessage, ( portTickType ) 0 );

    // ... Rest of task code.

} // Task to receive from the queue.
void vADifferentTask( void *pvParameters )
{
    struct AMessage *pxRxedMessage;

    if( xQueue != 0 )
    {
        // Receive a message on the created queue. Block for 10 ticks if a
        // message is not immediately available.
        if( xQueueReceive( xQueue, &( pxRxedMessage ), ( portTickType ) 10 ) )
        {
            // pcRxedMessage now points to the struct AMessage variable posted by vATask.
        }
    }

    // ... Rest of task code.
}
Linux based RTOSes

- RTLinux & RTAI
  - Extensions to standard Linux
  - Provide a layer between HW and Linux
    - Interrupt handler
    - Schedules Linux as the lowest priority task
  - RTAI is GPL
  - RTLinux by WindRiver
    - Basically a set of modules
      - rtai, rtai_sched, rtai_fifos, ...
    - Application written as a module, which uses RTAI API
Separated OSes – Paravirtualisation

- Partitioned environment for multiple operating systems (e.g. PikeOS)
  - with different design goals, safety requirements, or security requirements
- Timing and memory isolation

Diagram:
- Guest Operating System
- Guest Runtime Environment
- PikeOS Native Task
- PikeOS System Software
- PikeOS Separation Microkernel
  - Architecture Support Package (ASP)
  - Platform Support Package (PSP)
Other heavily used RTOSes

• VxWorks
  ▪ commercial
  ▪ used for example in Boeing 787, Mars Reconnaissance Orbiter, Phoenix Mars Lander

• Windows CE
  ▪ component-based, embedded, real-time operating system
  ▪ different kernel compared to Windows NT or Windows 3.1
  ▪ deterministic latency, 256 levels of priority
  ▪ similar API to other Windows systems
Scheduling of tasks

• Periodic tasks – A type of task that consists of a sequence of identical instances, activated at regular intervals.

• Examples
  ▪ Speed regulation
  ▪ Monitoring sensors
  ▪ Audio/video sampling

• Task characterized
  ▪ Period – T
  ▪ Deadline – D
  ▪ Worst-case execution time - C
Periodic task execution

• Typically realized this way:
  ▪ RTOS kernels with support for periodic tasks
    
    \[
    \text{period}_T = 50; \\
    \text{void task}_T() \\
    \text{ /* do some work */ }
    \]

  ▪ RTOS kernels with no support for periodic tasks
    
    \[
    \text{void task}_T() \\
    \text{ while (1) { \\
    \text{ /* do some work */ }
    \text{ /* wait until next period */ }
    \}
    \]
Periodic tasks scheduling

Offline scheduling algorithms
- Timeline (cyclic) scheduling

Online scheduling algorithms
- Static priorities
  - Rate Monotonic
  - Deadline Monotonic
- Dynamic priorities
  - Earliest Deadline First
Timeline scheduling

<table>
<thead>
<tr>
<th>task</th>
<th>f</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>40 Hz</td>
<td>25 ms</td>
</tr>
<tr>
<td>B</td>
<td>20 Hz</td>
<td>50 ms</td>
</tr>
<tr>
<td>C</td>
<td>10 Hz</td>
<td>100 ms</td>
</tr>
</tbody>
</table>

\[ \Delta = \text{GCD (minor cycle)} \]
\[ T = \text{lcm (major cycle)} \]

• Guarantee:
  - \[ C_a + C_b \leq \Delta \]
  - \[ C_a + C_c \leq \Delta \]
Timeline scheduling

• Advantages
  ▪ Simple implementation
  ▪ Low run-time overhead
    • No context switches
  ▪ It allows jitter control
    • Ordering of tasks inside major cycle

• Disadvantages
  ▪ It is not robust during overloads
  ▪ It is difficult to expand the schedule
  ▪ It is not easy to handle aperiodic activities

But in fact, it suffices in many cases!
Rate Monotonic Scheduling (RM)

- Each task is assigned a fixed priority proportional to its rate (T)
  - Priorities are assigned before execution (T-based)
  - Preemptive
  - Always task with highest priority (that is ready) runs
Schedulability of RM

- We can verify schedulability of the tasks with a simple test.
- Each task uses the processor for a fraction of time, hence the total processor utilization is:

\[ U_p = \sum_{i=1}^{n} \frac{C_i}{T_i} \]

- Then we compute utilization bound: \( U_{lub} = n(2^{\frac{1}{n}} - 1) \)

- If \( U_p \leq U_{lub} \) the task set is certainly schedulable with the RM algorithm
- **Note:** If \( U_{lub} < U_p \leq 1 \) we cannot say anything about the feasibility of that task set.
Schedulability of RM

- If $U_p > 1$ the processor is overloaded hence the task set cannot be schedulable.
- However, there are cases in which $U_p < 1$ but the task set is not schedulable by RM.

- Example:

  $$U_p = \frac{3}{6} + \frac{4}{9} = 0.944$$

Figure taken from Buttazzo, G.: Task scheduling
What next?

• NSWE001 – Embedded and real-time systems
  ▪ 2/1 LS in graduate programme

• What more can be found there?
  ▪ More on RTOSes
  ▪ Much more about scheduling
  ▪ Use of shared resources, synchronization
  ▪ RT communication
  ▪ RT design
  ▪ RTSJ – Real-time Java
  ▪ Hands on experience
What next?

- NSWE003 – Model-driven development of real-time embedded systems in Matlab/Simulink
  - 0/2 ZS in graduate programme
- What more can be found there?
  - Model-based development of real-time embedded systems
  - Hands on experience with Lego NXT