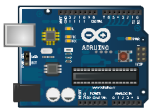
Real-Time Operating Systems

Ludovic Apvrille
ludovic.apvrille@telecom-paris.fr
Eurecom, office 470

perso.telecom-paris.fr/apvrille/OS/

What is an Embedded System?



Definitions

- Computer system designed for specific purpose
  - Tightly coupled hardware / software / analog integration

PC vs. embedded systems

- PC: general-purpose microprocessor and OS
  - High power consumption, heat production, large size
- Embedded system
  - Dedicated system
    - Dedicated hardware (e.g., DSPs) and software
  - Constraints of consumption, heat, size, performance, price, ...

Example of Embedded Systems

A Few Characteristics of Embedded Systems

Environment

- Temperature, vibration, impacts, variable power supply, interferences, corrosion, fire, water, radiations, small packaging, etc.

Safety critical systems

- Take into account possible failures and hazards if they may have major consequences
  - Hardware level: for example, a specification could be that the system should continue to work even if an electronic component fails (e.g. use of redundancy)
  - Safety at software level: e.g., handling of possible bugs and crash
    - Various execution modes

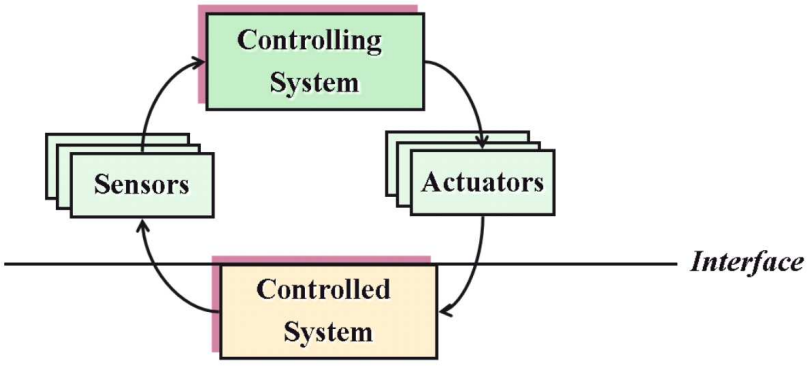
A Few Characteristics of Embedded Systems: Highly Available Systems

Number of 9s	Downtime per year	Typical application
3 Nines (99.9%)	≈ 9h	Desktop
4 Nines (99.99%)	≈ 1h	Enterprise server
5 Nines (99.999%)	≈ 5mn	Carrier-class server
6 Nines (99.9999%)	≈ 30s	Carrier network switch

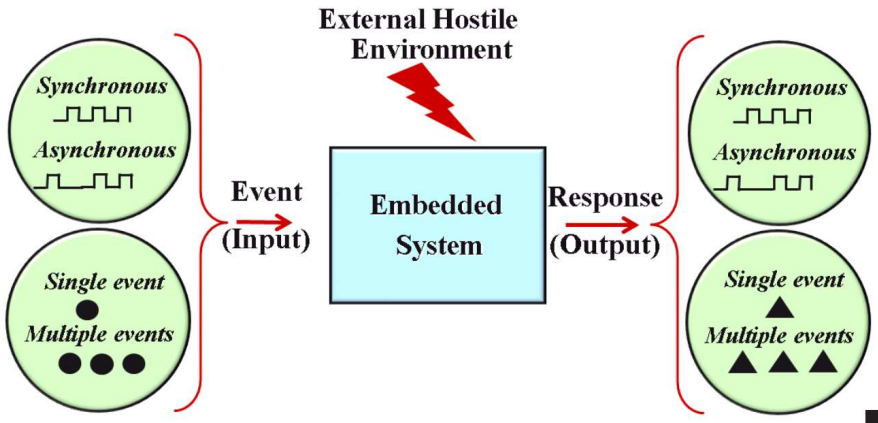
Source: "Providing Open Architecture High Availability solutions", Revision 1.0, Published by HA Forum, Feb. 2001

High-level Architecture

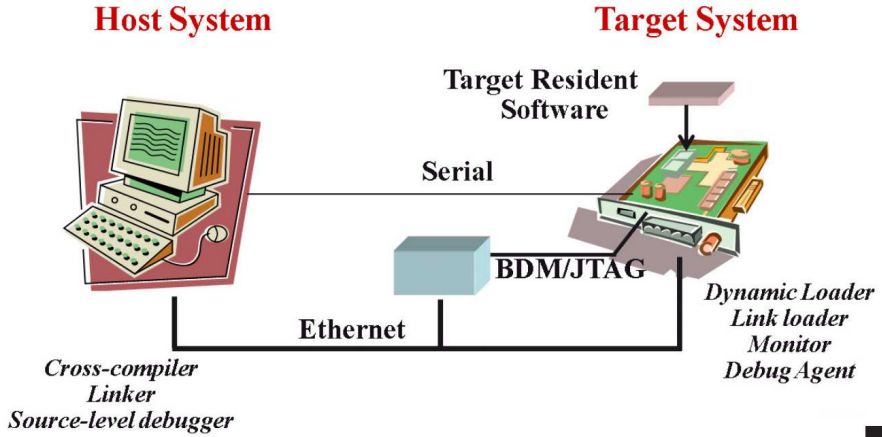
Embedded system = controlling system + controlled system + environment



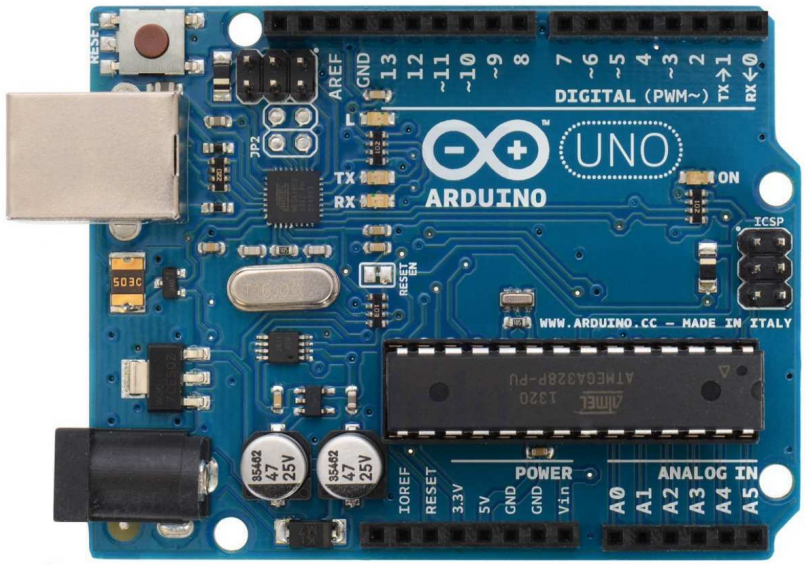
Response to External Events



Typical Development Environment



Example of a Target Embedded System



What is a Real-Time System?

Real-Time Systems have been defined as:

"Those systems in which the correctness of the system depends not only on the logical results of the computation, but also on the time at which the results are produced"

(J. Stankovic, "Misconceptions About Real-Time Computing", IEEE Computer, 21(10), October 1988)

What is a Real-Time System? (Cont.)

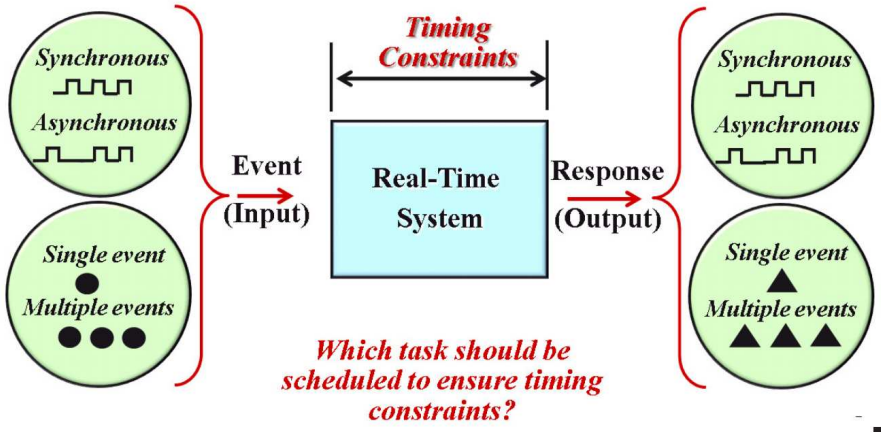
More precisely, a system is said to be a *Real-Time System* if:

- It reacts to external stimuli
  - System interacting with its environment
- It reacts to these stimuli within a limit of time, regardless of the load of the system
  - **WCET** = **W**orst **C**ase **E**xecution **T**ime
  - Timing correctness

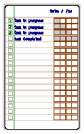
Vs. usual systems

- Only logical correctness
- Manage the average case
  - "Best effort"

Response to External Events



Time Constraints



Time constraint =
Constraint imposed on the timing behavior of each job of a task

Release time

- Instant of time when a job becomes available for execution

Deadline

- Instant of time at which a job is required to be completed
- *Absolute deadline* = release time + relative deadline

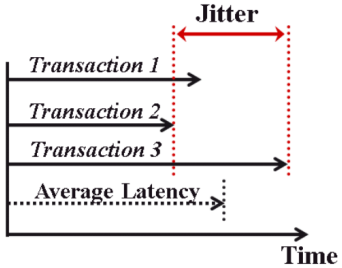
Response Time

- Difference between the completion time and the release time of a job

Latency and Jitter

- **System latency** = End-to-end delay between a change of state in the environment, and the corresponding output reaction produced by the system
  - Delay for scanning the environment, delay due to the OS, delay for executing calculations, delay for producing output results (communication delay)

- **Jitter** = Incertitude on delays
  - Load of the system, etc.
  - Should be low with regards to the latency



Incertitude / Determinism

- Computing time
  - Each activity can use various execution branches
- Communication time
  - Status of messages queues, delay of signals, etc.
- Interrupts
- Software or hardware exceptions

Hard and Soft Real-Time Systems

Comparison between hard and soft systems

1. The degree of tolerance of missed deadlines
2. The usefulness of computed results after missed deadlines
3. The severity of penalty incurred for failing to meet deadlines

	Hard system	Soft system
1	Zero tolerance	Non-zero
2	Useless	Not zero right after passing deadline
3	Catastrophic	Non-catastrophic

Complex real-time and embedded systems

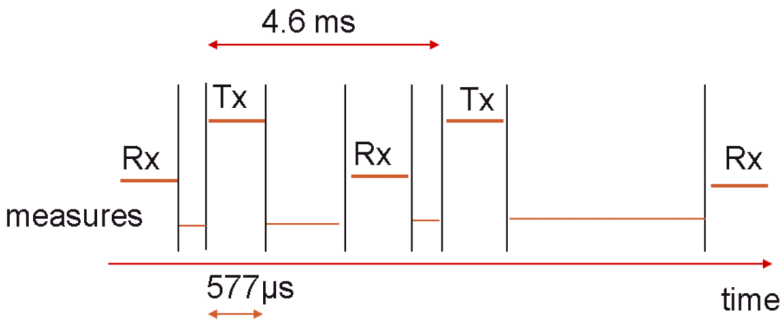
A complex real-time and embedded system is commonly a mix of hard and soft subsystems

Example ♯1: GSM Communication

Specification

- Software is in charge:
  - Of some operations performed at physical level (receiving, sending, measuring the level of the electromagnetic field, etc.)
  - Of logical operations such as scanning for incoming calls, maintaining connections when changing of beams, etc.
- Sending and receiving voice data is a critical task
  - 577μs of voice data are transmitted every 4.6ms
  - 577μs of voice data are received every 4.6ms
- Management of the mobility issue
  - If the distance to the relay is increasing, data should be emitted earlier to remain synchronized with the relay (Doppler effect)

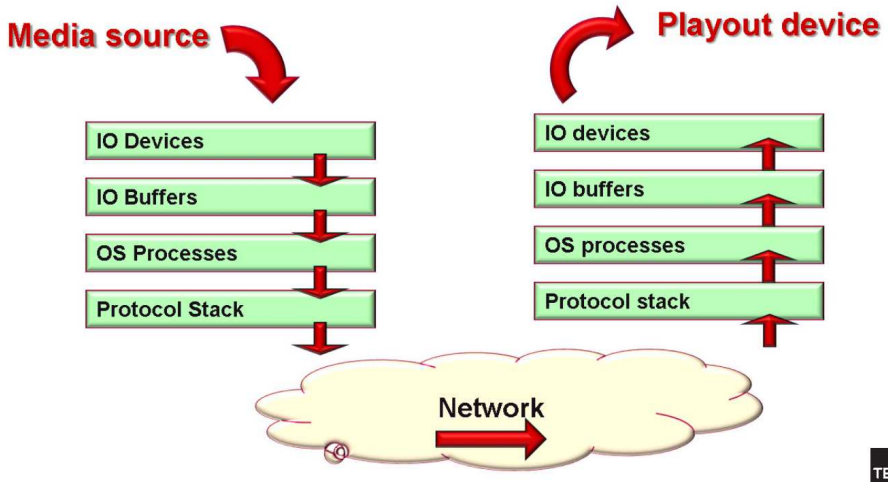
Example ♯1: A GSM Phone (Cont.)



Constraints

- Timing constraints on sending and receiving
- Global constraint: audio quality

Example ♯2: Video Streaming



True or False?

A real-time system runs faster than usual systems?

An elevator controlling system is a real-time system?

A real-time application can be executed on every operating systems?

Limitations of General-Purpose OS for Real-Time Systems

- **Scheduling policies** share CPUs in fair way → They don't take into account timing constraints
- **Memory management** (MMU, cache, dynamic allocation) introduces non-deterministic execution time
- **Input / output** management is non-deterministic
- **Communication mechanisms** introduce non-deterministic temporal behavior
- **Software timers** used for managing time are not fine-grained enough

→ Use of Real-Time Operating Systems

Key Characteristics

Reliability

- Need to operate for a long period of time
- System reliability depends on all system elements
  - Hardware, BSP, RTOS and application(s)

Predictability

- To a certain degree, and at least better than in GPOS
- Completion of system calls occurs within known frames
  - Benchmark on the variance of the response times for each system calls
  - WCET

Key Characteristics (Cont.)

Performance

- CPU performance: MIPS
- Throughput performance: bps

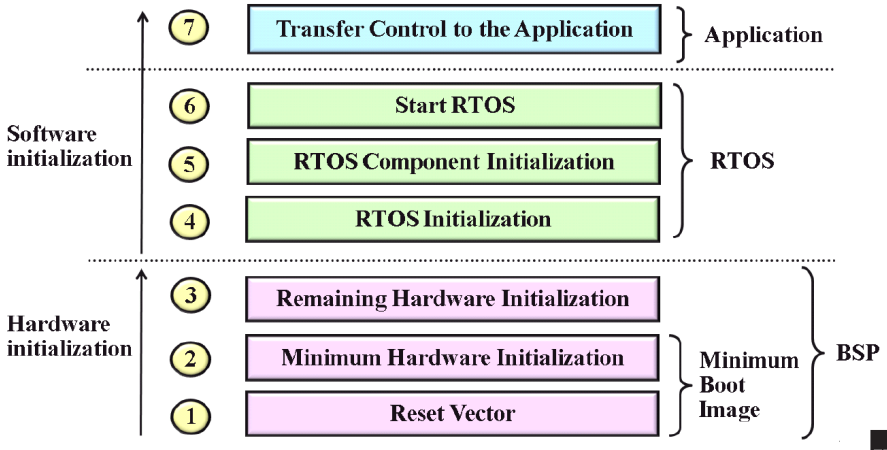
Compactness

- RTOS memory footprint should be small

Scalability, i.e handling application-specific requirements

- Modular components
- File systems, protocol stacks, etc.

Target Initialization Sequence



Examples of RTOS

RTOS

- FreeRTOS, iOS, VxWorks
- Linux for RT and embedded systems
  - RTAI, Xenomai, Wind River Linux, Android

Using FOSS for Embedded Systems

("FOSS" means "Free and Open Source Software")

- No royalties when distributing the embedded system
- Source code is available
  - Maintenance can be performed for as long as desired
  - Systems can be built from this source code

How to Make UNIX more "Real-Time"?

UNIX	Real-Time UNIX
Time-sharing based scheduling	Real-time scheduling classes
Services not reentrant	Fully reentrant libraries
IPC non deterministic	Reworking IPC with deterministic communication time
Large memory footprint	Modularity (micro-kernels)
Timing issues, timer accuracy	Preemptive kernel
...	...

Scheduling

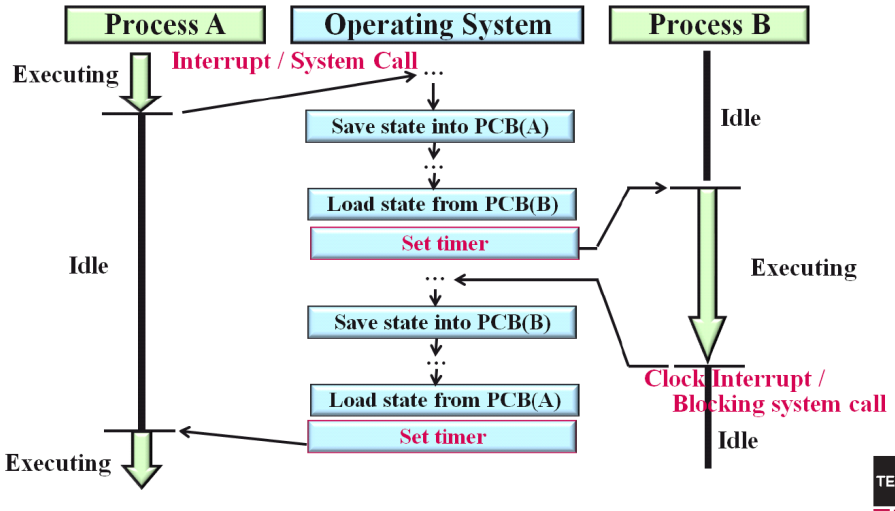
Limitations of schedulers in general-purpose OS

- Schedulers are defined with interactivity in mind
  - Dynamic priorities
  - Calculation tasks are penalized
- Timing constraints are not taken into account
- Schedulability analysis cannot be performed

Schedulers for Real-Time Systems

RMA, EDF, FCFS, Round-Robin, Fixed priority

Preemption Latency in a Priority-Based System



Preemption Points

Adding preemption points in the kernel ("Fully preemptive" kernel)

- Define a slice of time during which no preemption is allowed (e.g., 2,000 instructions)
- Rewrite all kernel primitives:
  - The duration during which no preemption is allowed must always be shorter than the defined slice of instructions
  - After each slice, call the scheduler (just in case)

Limitation: Defining the slice is not obvious

- Too long → average preemption latency is higher
- Too short → important overhead

Memory Management

Dynamic allocations

- Should be avoided at run-time
- Algorithms are based on fixed-size blocks
- Pools of objects
  - Avoid frequent allocation and fragmentation

No memory compaction

- Real-time compaction algorithms might be used

Most of the time, virtual addressing (i.e., MMU) is not used

- And sometimes even not implemented in RTOS!
- Page fault introduces non-determinism

IPC - Inter Process Communications

Limitations for RT systems

- No deterministic transmission delay for:
  - Signals, pipes, message passing, semaphores, (shared memory)

A Few Solutions

- Implementation with deterministic processing time
  - Example: RT-FIFO
    - RT-FIFO = FIFO with predefined maximum size and with pre-allocated messages

Input / Output Management

Limitations for RT systems

- Difficult or impossible to bound the time to perform an I/O operation

Solutions

- Drivers working closely with the kernel
- Deterministic delay in all inter-process communications
  - Communication structures with basic and deterministic algorithms
    - RT-FIFO, RT-signals, etc.

Reentrant Runtime Libraries

Issue

- Libraries offer services such as: Input / Output operations, dynamic memory allocations, etc.
- Services may have an internal state: buffer (I/O), pointer to the next chunk of free memory, etc.
- If preemption occurs when a service is being called, internal state of the service may be corrupted

Solutions

- Protection at application level: mutex, semaphore, etc.
  - Safe, but costly
- Use a reentrant version of the library (if available)
  - Safe, less costly (but more costly than non-reentrant librairies)

Exceptions and Interrupts

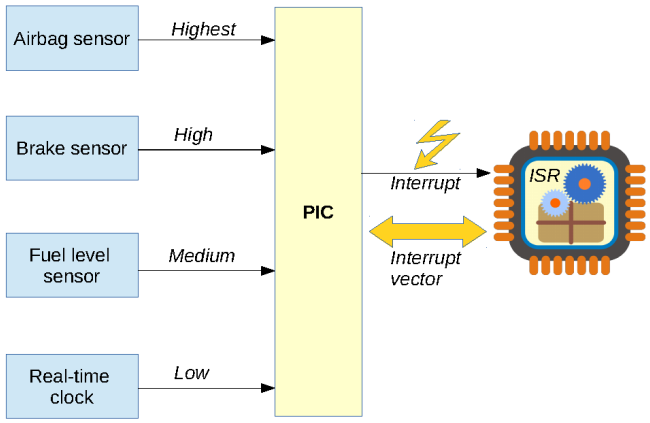
Exceptions

- *Events that disrupt the normal execution flow of the processor and force the processor to execute special instructions in a privileged state*
- Synchronous exceptions
  - Divide by zero
- Asynchronous exceptions
  - Pushing the reset button of the embedded system

Interrupts

- *= Asynchronous exceptions triggered by events external hardware devices generate*

Programmable Interrupt Controller



Programmable Interrupt Controller (Cont.)

- **Hardware-implemented**
- Prioritizing multiple interrupt sources so that at any time the highest priority interrupt is presented to the core CPU for processing
- PICs have a set of Interrupt Request Lines
  - Interrupt = physical signal on a given line
- Handlers specified at PIC level
  - **ESR** = **E**xception **S**ervice **R**outine
  - **ISR** = **I**nterrupt **S**ervice **R**outine

Programmable Interrupt Controller: Example

Source	Priority	Vector Address	IRQ	Max freq.	Description
Airbag sensor	Highest	14h	8	N/A	Detects the need to deploy airbags
Brake sensor	High	18h	7	N/A	Informs about the current braking power
Fuel Level Sensor	Med	1Bh	6	20Hz	Detects the level of fuel
RTC	Low	1Dh	5	100Hz	Clock runs at 10ms tick

Masking Interrupts

Why masking?

- Reduce the total number of interrupts raised by devices
  - Warning: Interrupts may be lost when masked
- Atomic operations

How is masking done?

- Disable the device so that it cannot assert additional interrupts
- Mask the interrupts of equal or lower priority levels
- Disable the line between the PIC and the core processor
  - Interrupts of any priority level do not reach the processor

Classification of Exceptions

Asynchronous

Maskable

Can be blocked or enabled by software

Non-maskable

Cannot be blocked or enabled by software

Synchronous

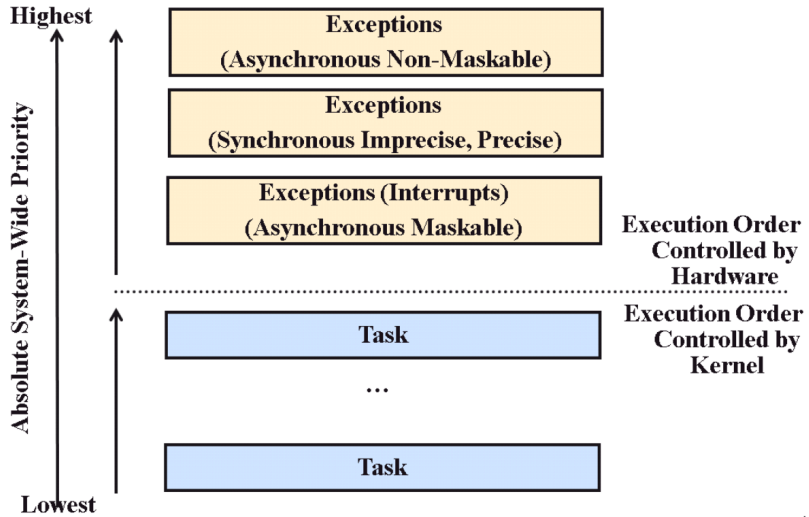
Precise

- Program counter points to the exact instruction which caused the exception (Offending instruction)
- Processor knows where to resume the execution

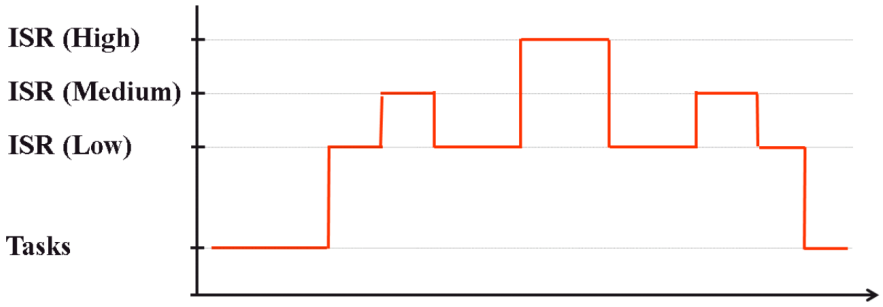
Non-precise

Because of *prefetching* and *pipelining* techniques, no knowledge about the exact instruction that caused the exception

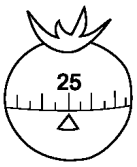
General Exceptions Priorities



Nested Interrupts

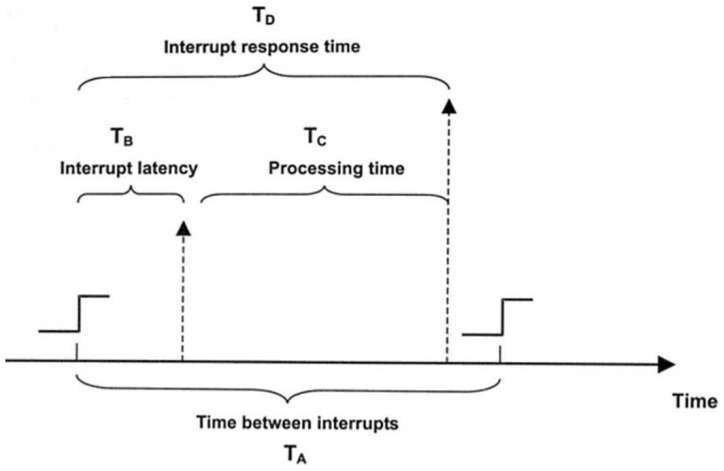


Exceptions Timing

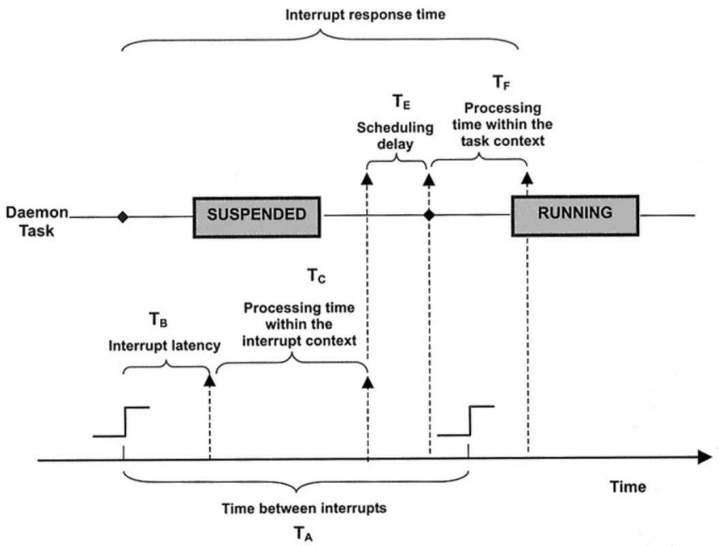


- ESR and ISR should be short
- Designers
  1. Should avoid situations where interrupts could be missed
  2. Should be aware of the interrupt frequency of each device
     - Maximum allowed duration can be deduced from this information
  3. Should take into account the fact that real-time tasks cannot execute when interrupts are being processed

Exceptions Timing (Cont.)



Cutting ISRs in Two Parts



Conclusion on ISRs

Benefits about cutting ISRs in two parts

- Lower priority interrupts can be handled
- Reduces the chance of missing interrupts
- Increase concurrency because devices can start working on next operations earlier
- Urgent tasks have a chance to be executed sooner

General guide for implementing ISRs

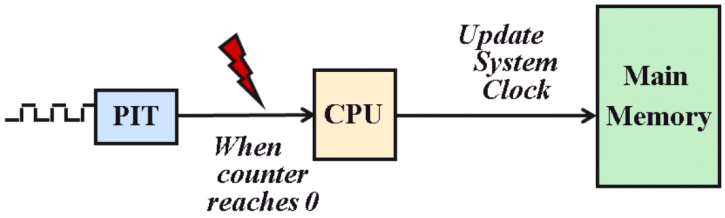
- An ISR should disable interrupts of the same level
- An ISR should mask all interrupts if it needs to execute a sequence of code as one atomic operation
- An ISR should avoid calling non-reentrant functions and blocking system calls

Timer and Timer Services

Use of Timers

Applications often have to perform jobs after a given delay has elapsed

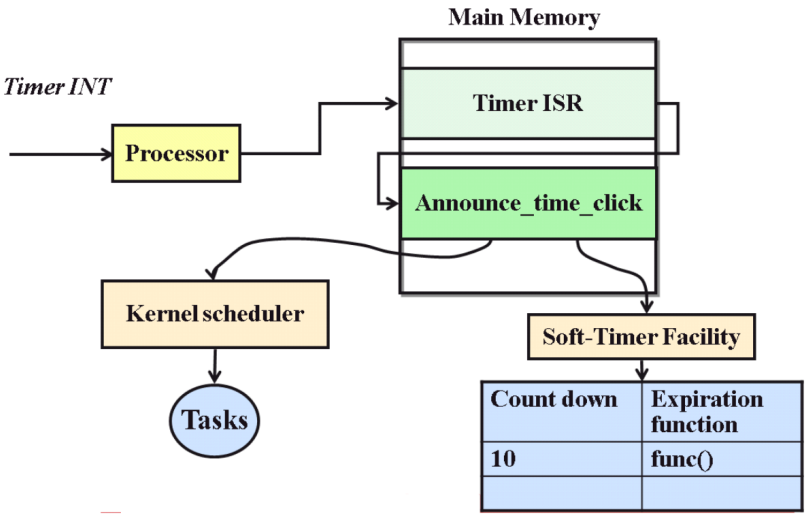
Programmable Interval Timers (PIT)



Timer Interrupt Service Routine

- Updating the system clock
  - Absolute time (calendar)
  - Elapsed time (since power up)
- Calling a registered kernel function to notify the occurrence of a pre-programmed period
  - *announce_time_tick*()
    - Calls the Scheduler and the *Soft-timer* facility
- Acknowledging the interrupt, reinitializing the necessary timer control register and returning from interrupt

Steps in Servicing Timer Interrupt



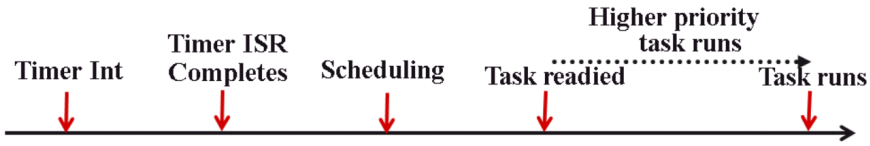
Soft-Timer Facility

Goal

- Allowing applications to start a timer
  - A callback function is called when the timer expires
- Allowing applications to stop or cancel a previously installed timer
- Internally maintaining the application timers

The timer ISR should be as fast as possible to avoid missing next timer interrupts

Timer Inaccuracy Due to Processing Delays



- ISR triggers asynchronous events to signal a task of the expiration of its timer
  - E.g., sending of a message, release of a semaphore
- Task scheduling delay, context switch
- Higher priority task may be running

Limitations of Linux for Real-Time Systems

- Time-sharing system
  - Interactivity-based scheduling: strict priority-based approach is needed in real-time systems!
- Long code sections where all external events are masked
  - Preemption is not possible
- Kernel cannot be preempted
  - When executing a system call (so, interrupts may be missed!)
  - When executing an ISR
- Device management

Towards a Linux for Real-Time Systems

- Introducing a real-time scheduler and preemption points
  - Patching the kernel
- Adding new kernel modules

Introducing a Real-Time Scheduler (Patch)

Patchs are said to be "preemptive"

- Kernel's latency is reduced
- Limited to soft real-time systems?

"RT-Preempt"

- Based on call to the scheduler whenever possible
- Kernel's data are protected with mutex

"Low Latency"

- Adds preemption points at carefully chosen places in the code
- Approach is less systematic
- Performance study demonstrates its efficiency with regards to the "RT-Preempt" patch

Modifying the Kernel (Modules)

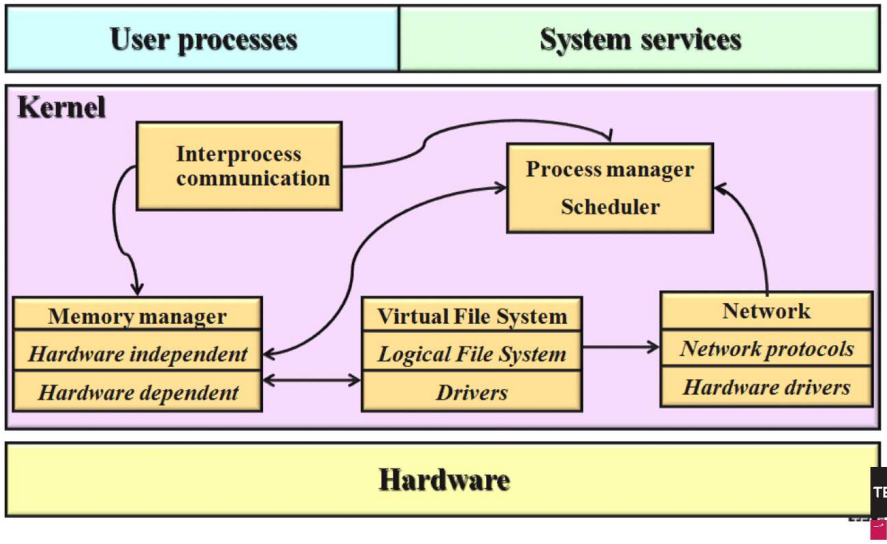
Basic idea: the Linux kernel will never be "real-time"

- Another scheduler is added to the kernel
  - Priority-based scheduling policy
  - Non real-time tasks are scheduled by the default Linux scheduler
  - Real-time tasks are programmed as linux kernel modules
- $\rightarrow$ Hard real-time system

Main implementations

- RTAI (http://www.rtai.org/)
- Xenomai (https://xenomai.org/)

Architecture of the Linux Kernel



Architecture of Xenomai