technische universität dortmund

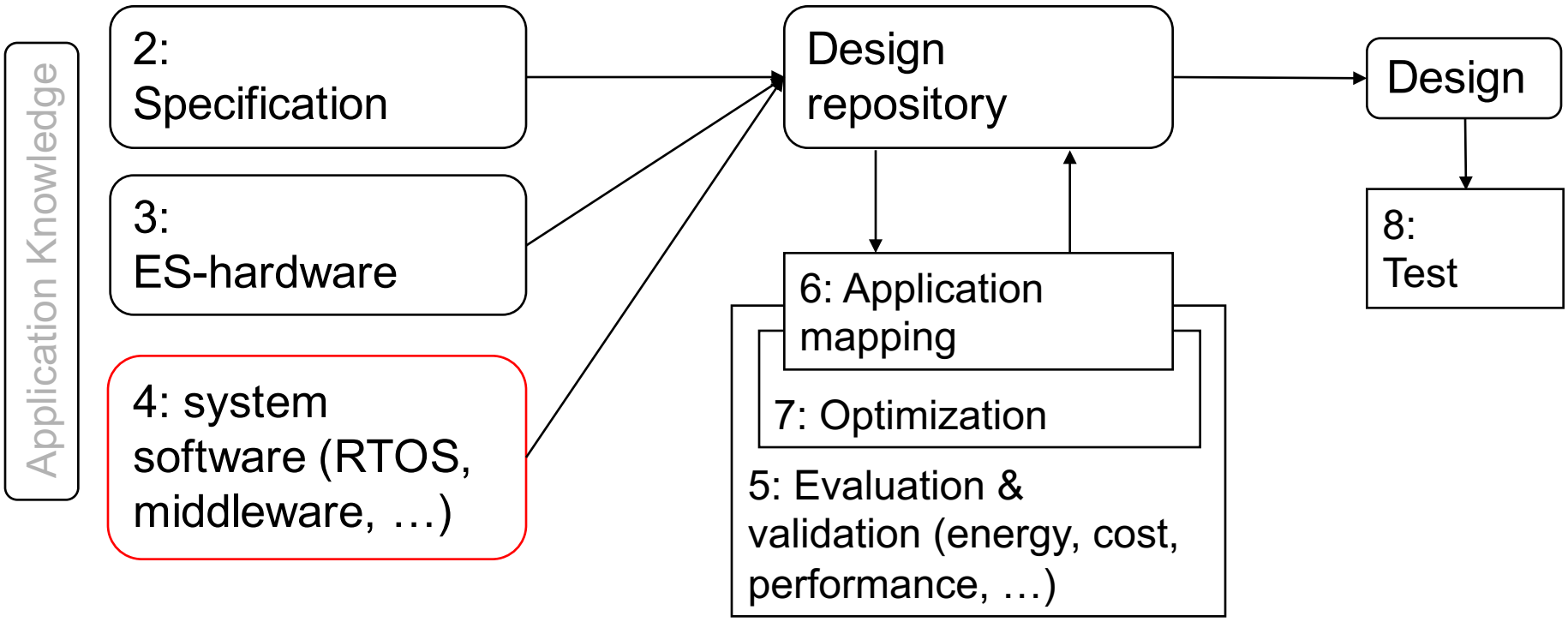
fakultät für informatik

Embedded & Real-time Operating Systems

Jian-Jia Chen
(Slides are based on
Peter Marwedel)
TU Dortmund, Informatik 12
Germany

2018年 10 月 30日

Structure of this course



Numbers denote sequence of chapters

Increasing design complexity + Stringent time-to-market requirements ☞Reuse of components

Reuse requires knowledge from previous designs to be made available in the form of **intellectual property** (IP, for **SW & HW**).

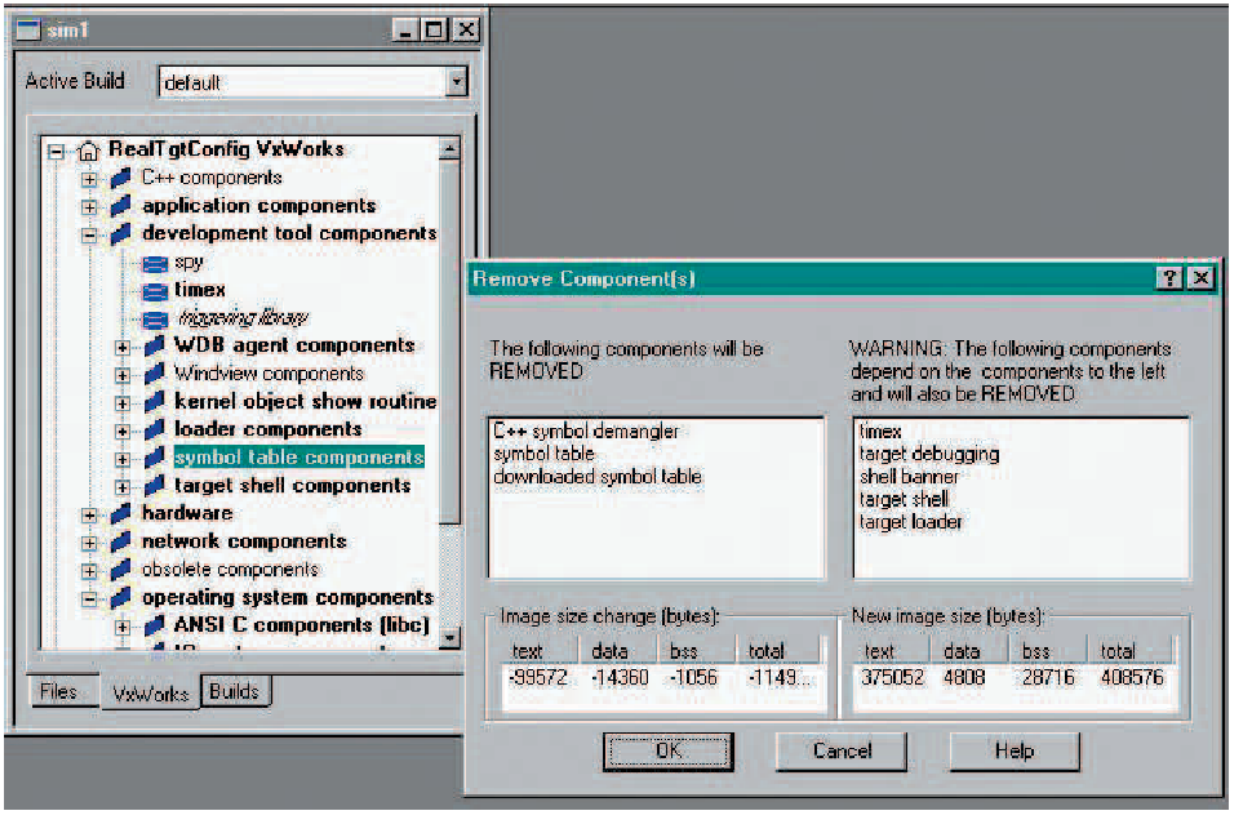
- HW
- Operating systems
- Middleware (Communication, data bases, …)
- ….

Embedded operating systems
- Characteristics: Configurability -

Configurability
No overhead for unused functions tolerated,
no single OS fits all needs, ☞ configurability needed.

- Conditional compilation (using #if and #ifdef commands).
- Object-orientation could lead to a of derivation subclasses.
- Aspect-oriented programming
- Advanced compile-time evaluation useful.
- Linker-time optimization (removal of unused functions)

Example: Configuration of VxWorks



http://www.windriver.com/products/development_tools/ide/tornado2/tornado_2_ds.pdf

Automatic dependency analysis and size calculations allow users to quickly custom-tailor the VxWORKS operating system.

Verification of derived OS?

Verification a potential problem of systems with a large number of derived OSs:

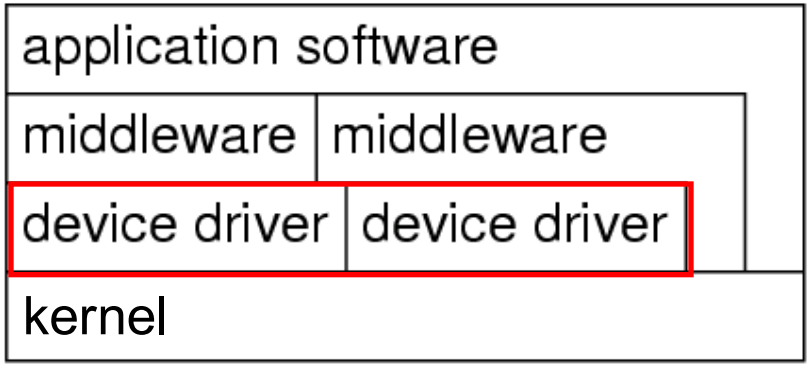
- Each derived OS must be tested thoroughly;
- Potential problem for eCos (open source RTOS from Red Hat), including 100 to 200 configuration points [Takada, 2001].

Embedded operating systems

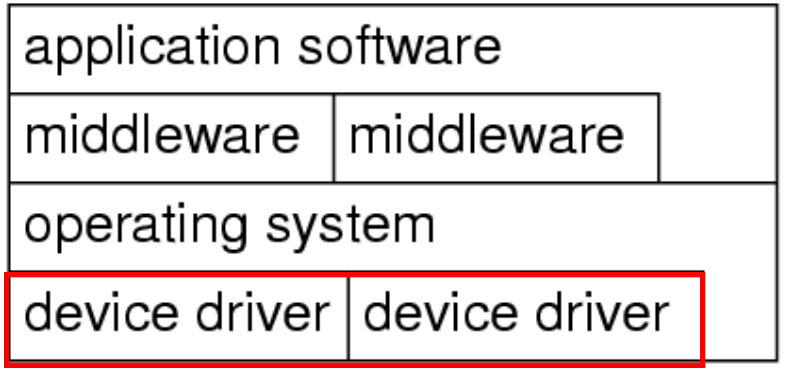
- Characteristics: Disk and network handled by tasks -

- **Effectively no device needs to be supported by all variants of the OS**, except maybe the system timer.
- Many ES without disk, a keyboard, a screen or a mouse.
- Disk & network handled by tasks instead of integrated drivers.

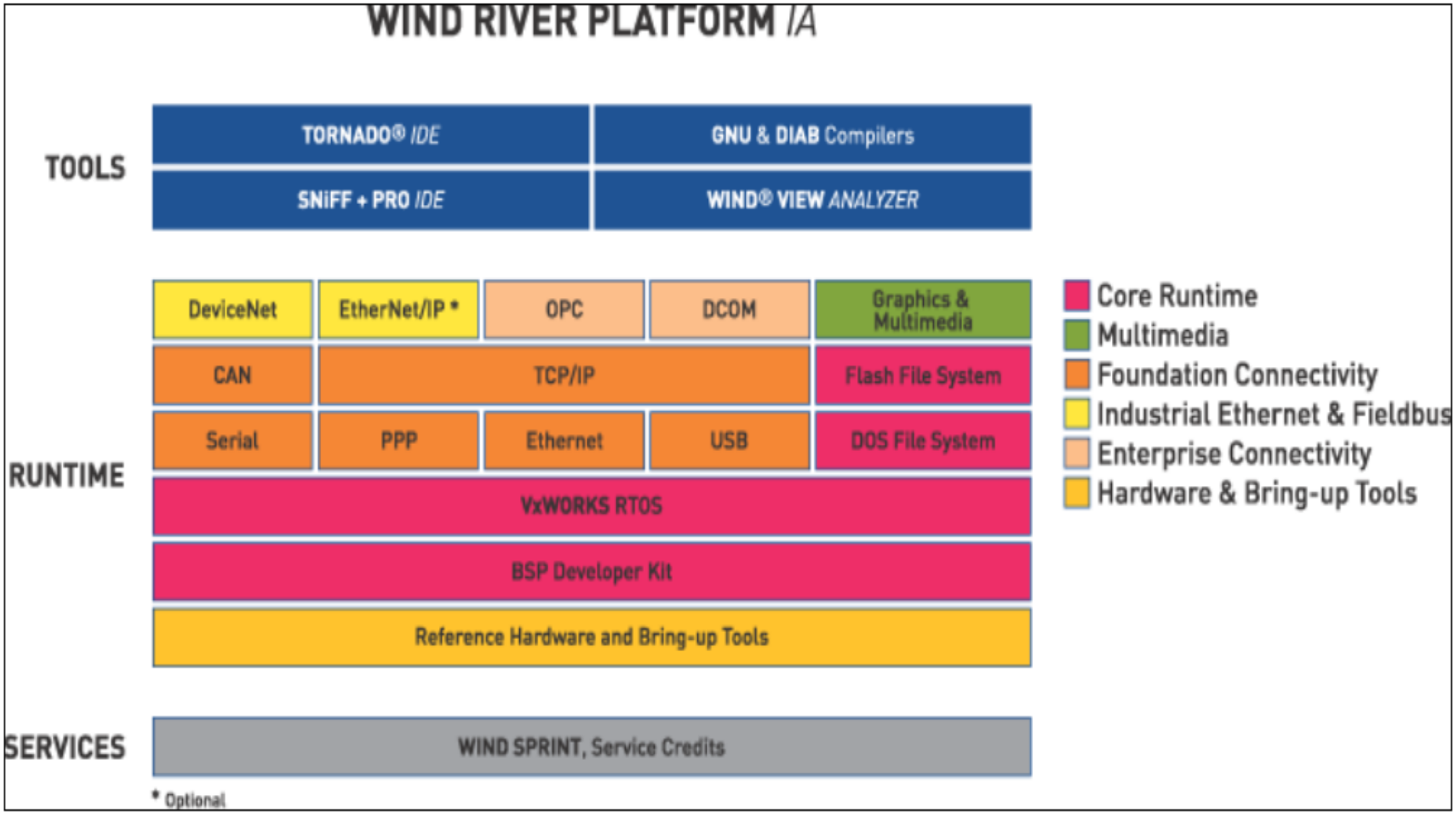
Embedded OS



Standard OS



Example: WindRiver Platform Industrial Automation



Embedded operating systems
- Characteristics: Protection is optional-

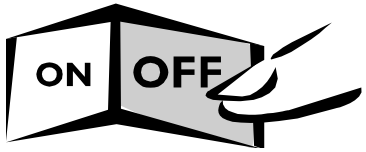
Protection mechanisms (user mode and privilege mode) not always necessary: especially for single-purpose ES untested programs rarely loaded, SW considered reliable.

Privileged I/O instructions not necessary and tasks can do their own I/O.

Example: Let **`switch`** be the address of some switch
Simply use

```
load register,switch
```

instead of OS call.



However, protection mechanisms may be needed for safety and security reasons.

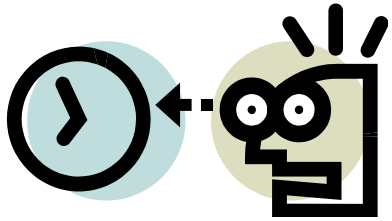
Embedded operating systems
- Characteristics: Interrupts not restricted to OS -

Interrupts can be employed by any process
For standard OS: serious source of unreliability.
Since

- embedded programs can be considered to be tested,
- since protection is not always necessary and
- since efficient control over a variety of devices is required,
- it is possible to let interrupts directly start or stop SW (by storing the start address in the interrupt table).
- More efficient than going through OS services.
- Reduced composability: if SW is connected to an interrupt, it may be difficult to add more SW which also needs to be started by an event.

Embedded operating systems
- Characteristics: Real-time capability-

Many embedded systems are real-time (RT) systems and, hence, the OSs used in these systems must be **real-time operating systems (RTOSs).**



RT operating systems - Definition and requirement 1: predictability -

Def.: *(A) real-time operating system is an operating system that supports the construction of real-time systems.*

The following are the three key requirements

1. **The timing behavior of the OS must be predictable.**
   ∀ services of the OS: Upper bound on the execution time!
   RTOSs must be timing-predictable:
   - short times during which interrupts are disabled,
   - (for hard disks:) contiguous files to avoid unpredictable head movements.

   [Takada, 2001]

Real-time operating systems requirement 2: Managing timing

2. OS should manage the timing and scheduling

- OS possibly has to be aware of task deadlines; (unless scheduling is done off-line).
- Frequently, the OS should provide precise time services with high resolution.

[Takada, 2001]

Time

Time plays a central role in “real-time” systems

Physical time: real numbers

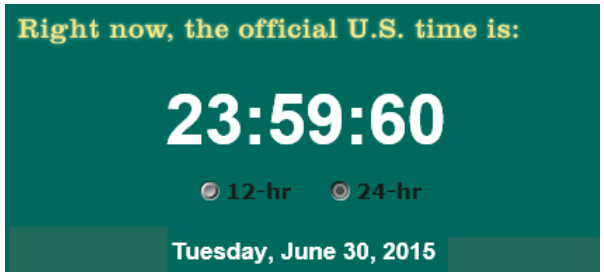
Computers: mostly discrete time

- Relative time: clock ticks in some resolution
- Absolute time: wall clock time
  - **International atomic time TAI**
    (french: *temps atomic internationale*)
    Free of any artifacts.
  - **Universal Time Coordinated (UTC)**
    UTC is defined by astronomical standards

TAI and UTC identical on Jan. 1st, 1958.

30 seconds had to be added since then.

UTC uses a leap second to adjust the time.



Internal synchronization

- Synchronization with one master clock
  - Typically used in startup-phases
- Distributed synchronization:

1. Collect information from neighbors
2. Compute correction value
3. Set correction value.

Precision of step 1 depends on how information is collected:

- Application level: ~500 µs to 5 ms
- Operation system kernel: 10 µs to 100 µs
- Communication hardware: < 10 µs

External synchronization

External synchronization guarantees consistency with actual physical time.

Trend is to use GPS for ext. synchronization

GPS offers TAI (International Atomic Time) and UTC time information.

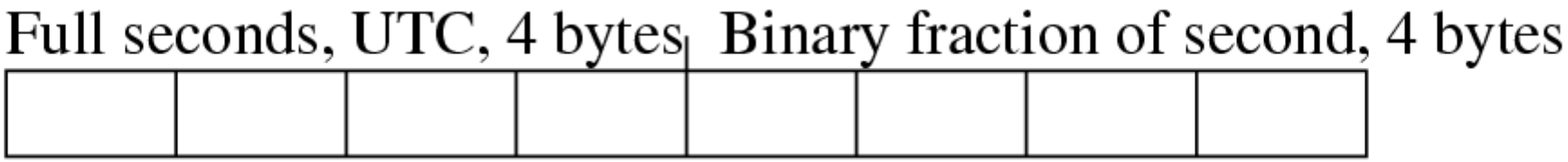
Resolution is about 100 ns.

GPS mouse

Problems with external synchronization

Problematic from the perspective of fault tolerance:
Erroneous values are copied to all stations.
Consequence: Accepting only small changes to local time.

Many time formats too restricted;
e.g.: NTP protocol includes only years up to 2036



Range up the years 2036; 136 year wrap around cycle

For time services and global synchronization of clocks see Kopetz, 1997.

Real-time operating systems requirement 3: Speed

3. **The OS must be fast**

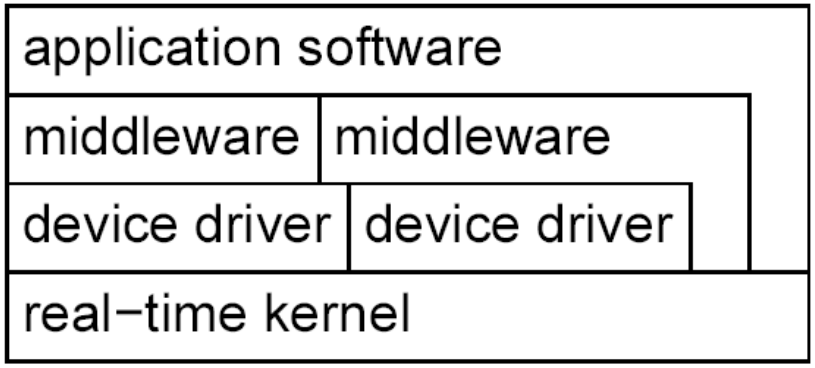
   Practically important.

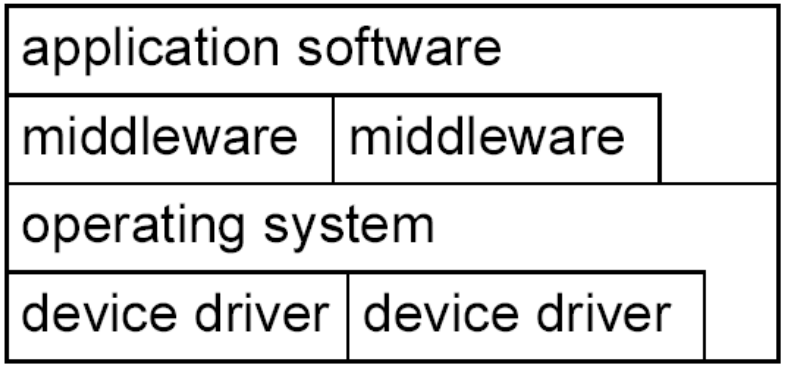
[Takada, 2001]

RTOS-Kernels

Distinction between

- real-time kernels and modified kernels of standard OSes.





Distinction between

- general RTOSs and RTOSs for specific domains,
- standard APIs (e.g. POSIX RT-Extension of Unix, OSEK) or proprietary APIs.

Functionality of RTOS-Kernels

Includes

- processor management,
- memory management,
- and timer management;

} resource management

- task management (resume, wait etc),
- inter-task communication and synchronization.

Classes of RTOSes:
1. Fast proprietary kernels

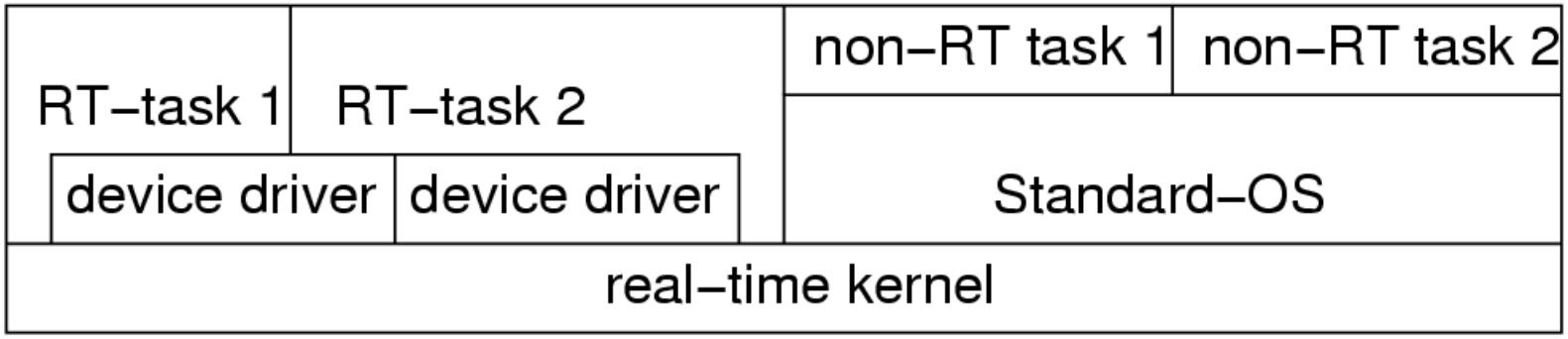
For complex systems, these kernels are inadequate, because they are designed to be fast, rather than to be predictable in every respect

[R. Gupta, UCI/UCSD]

Examples include
QNX, PDOS, VCOS, VTRX32, VxWORKS.

Classes of RTOSs:
2. RT extensions to standard OSs

Attempt to exploit comfortable main stream OS.
RT-kernel running all RT-tasks.
Standard-OS executed as one task.

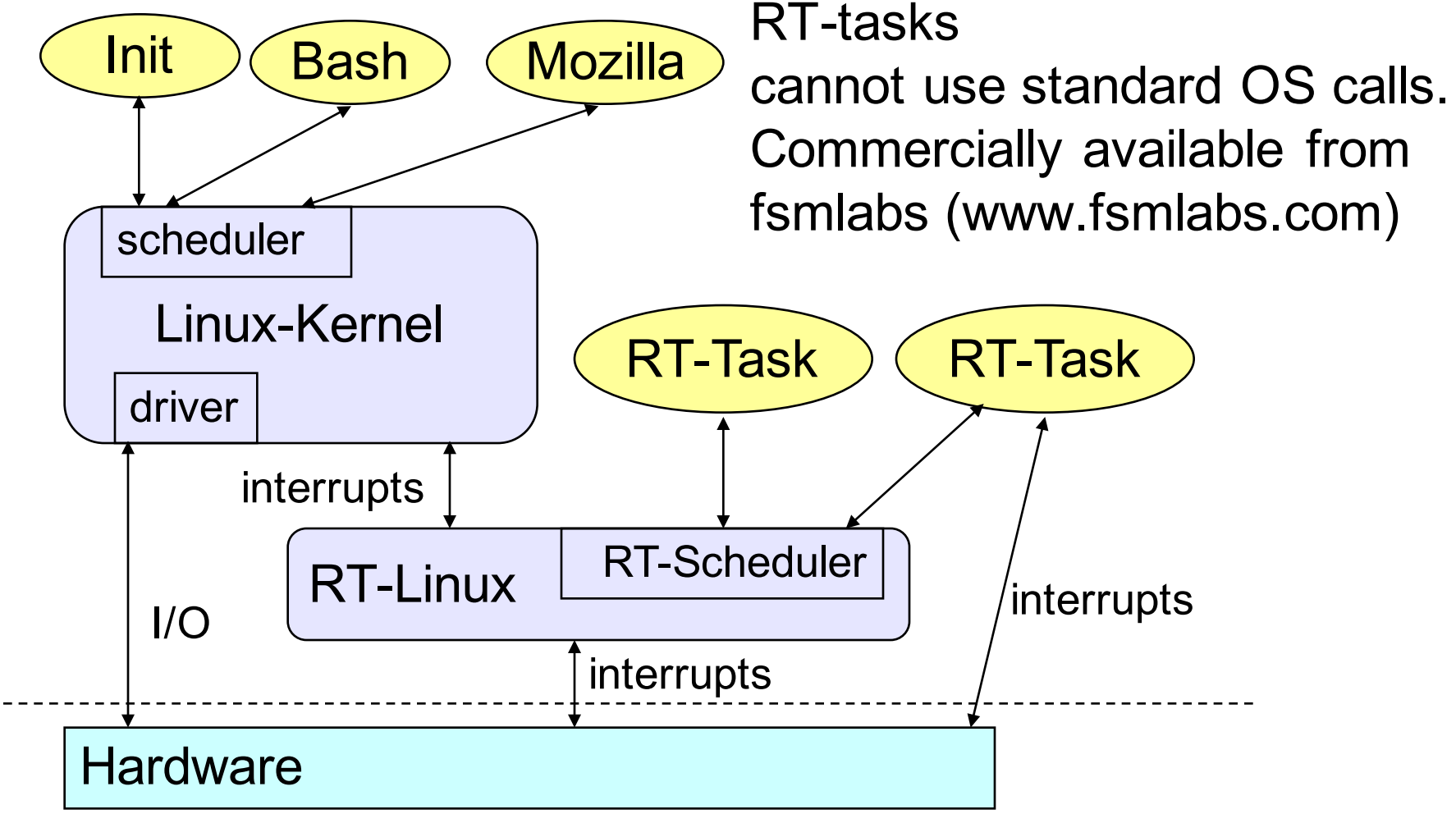


+ Crash of standard-OS does not affect RT-tasks;
- RT-tasks cannot use Standard-OS services;
  less comfortable than expected

RT extensions to standard OSs

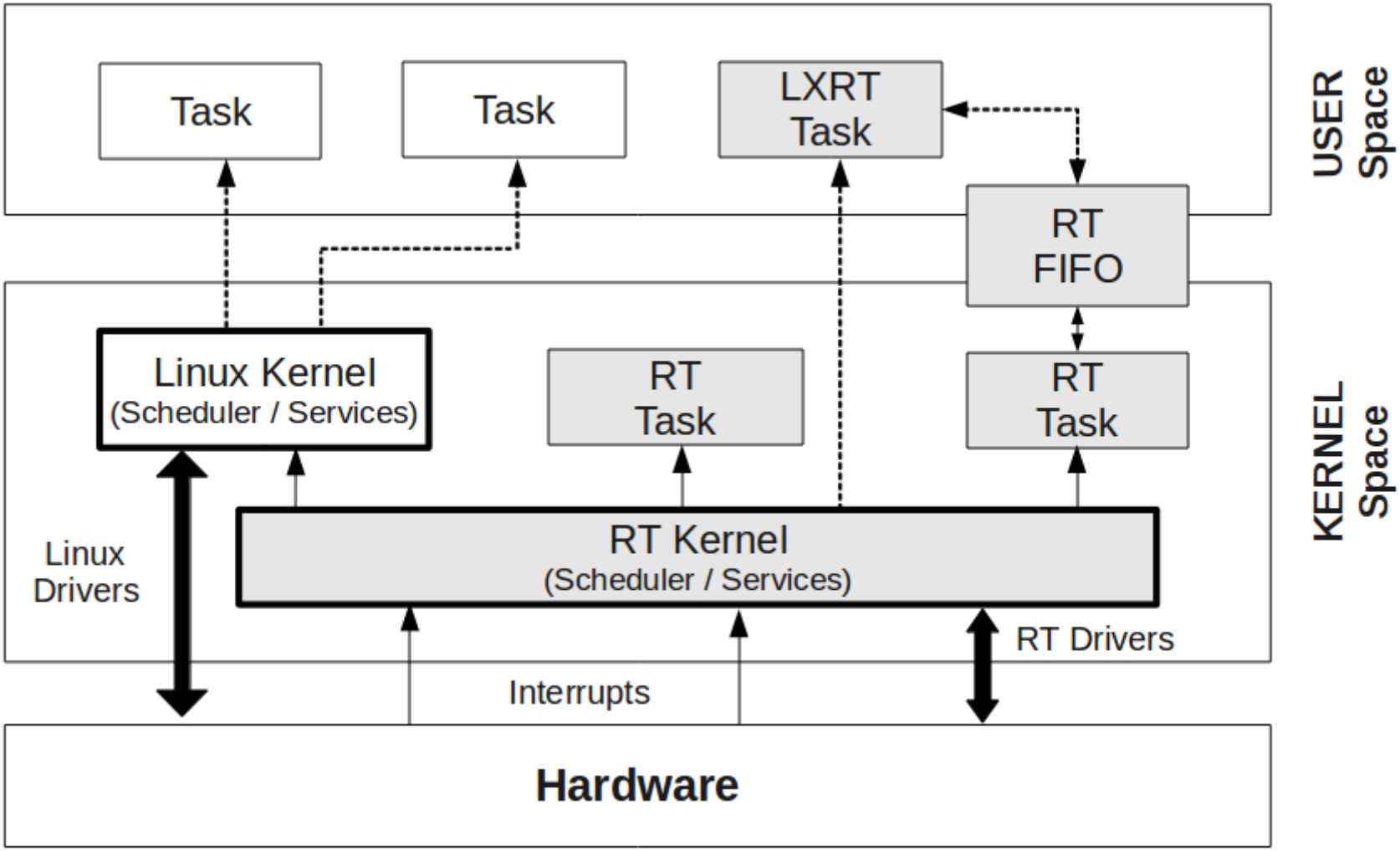
- A common approach is to extend Unix
  - Linux: RT-Linux, RTLinuxPro, RTAI, etc.
  - Posix: RT-POSIX
- Also done for Windows based on virtualization, e.g. RTOSWin, RT-Xen

Example: RT-Linux



RT-tasks
cannot use standard OS calls.
Commercially available from fsmlabs (www.fsmlabs.com)

Example (2):
RTAI – Real Time Application Interface



Evaluation

According to Gupta, trying to use a version of a standard OS:

not the correct approach because too many basic and inappropriate underlying assumptions still exist such as optimizing for the average case (rather than the worst case), ... ignoring most if not all semantic information, and independent CPU scheduling and resource allocation.

Dependences between tasks not frequent for most applications of std. OSs & therefore frequently ignored.

Situation different for ES since dependences between tasks are quite common.

Classes of RTOSs:
3. Research trying to avoid limitations

Research systems trying to avoid limitations.
Include MARS, Spring, MARUTI, Arts, Hartos, DARK, and Melody

Research issues [Takada, 2001]:

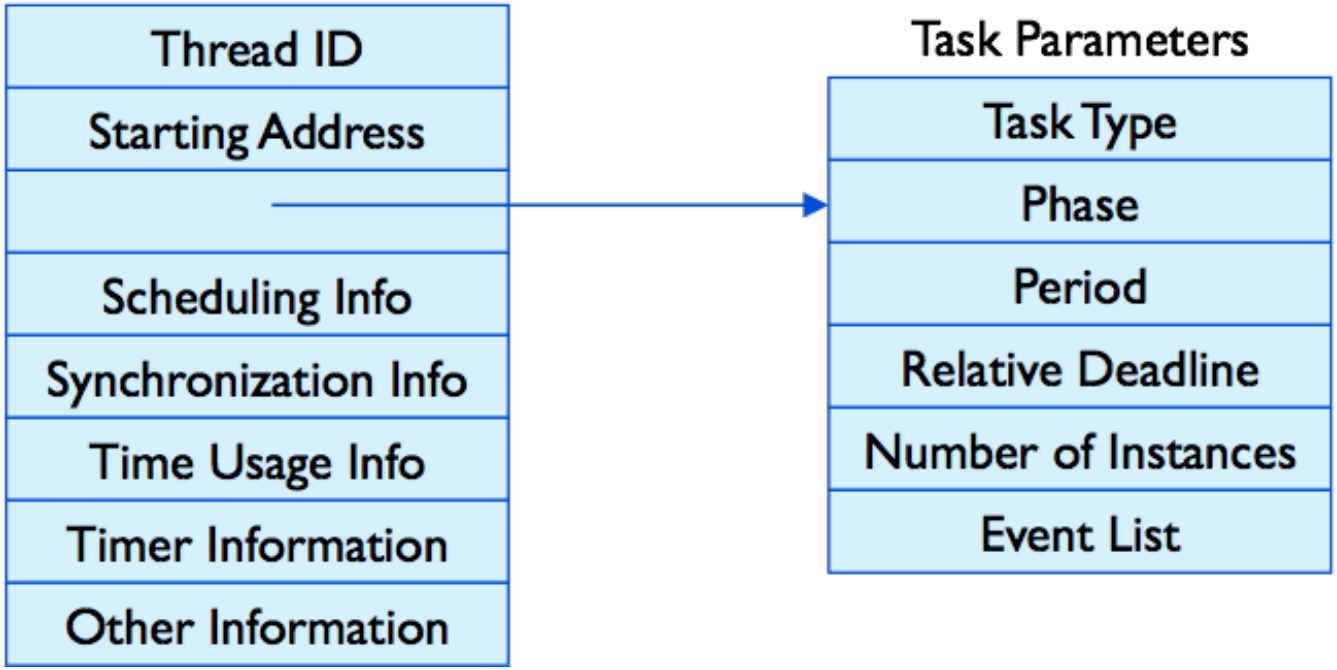
- low overhead memory protection,
- temporal protection of computing resources
- RTOSes for on-chip multiprocessors
- quality of service (QoS) control.

Task, Thread, Job

- Thread (Job): A basic unit of work handled by the scheduler
- Task: Threads implement the jobs of a task. Usually the same thread is re-used for each job of a task
- Thread Context: The values of registers and other volatile data that define the state and environment of the thread

Task, Thread, Job (Continued)

- TCB: The thread control block (TCB) is the data structure created when the kernel creates a thread
  - The TCB stores the context of the thread when it is not executing



Periodic Tasks and Threads

- **Periodic thread**: Reinitialized by the kernel and put to sleep (i.e., suspends) when the thread completes. Released by the kernel at the beginning of the next period (i.e., becomes ready)
- The task parameters (e.g., phase and period) are stored in a separate manner
- **Most commercial (RT or non-RT) OSs do not support periodic threads**
  - Instead, the thread itself sleeps (i.e., suspends itself via some system call) until the start of the next period after it finishes executing

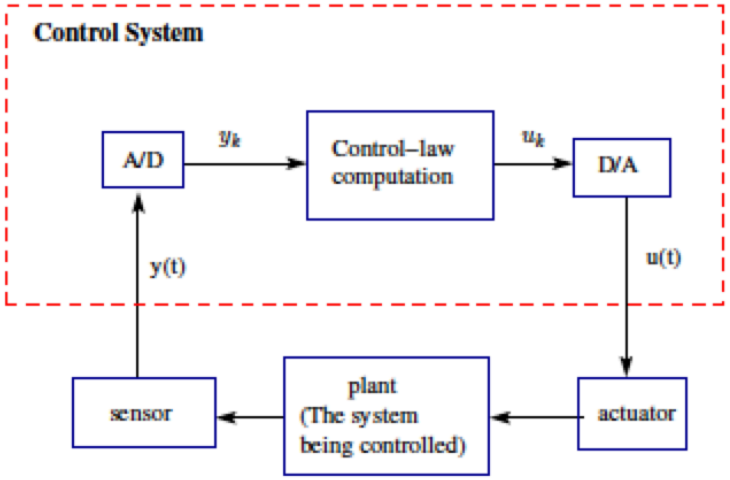
Example: Control System

Pseudo-code for this system

while (true)

- start := get the system tick;
- perform analog-to-digital conversion to get y;
- compute control output u;
- output u and do digital-to-analog conversion;
- end := get the system tick;
- $timeToSleep := T - (end - start)$;
- sleep $timeToSleep$;

end while



Example: Periodic Control System

Pseudo-code for this system

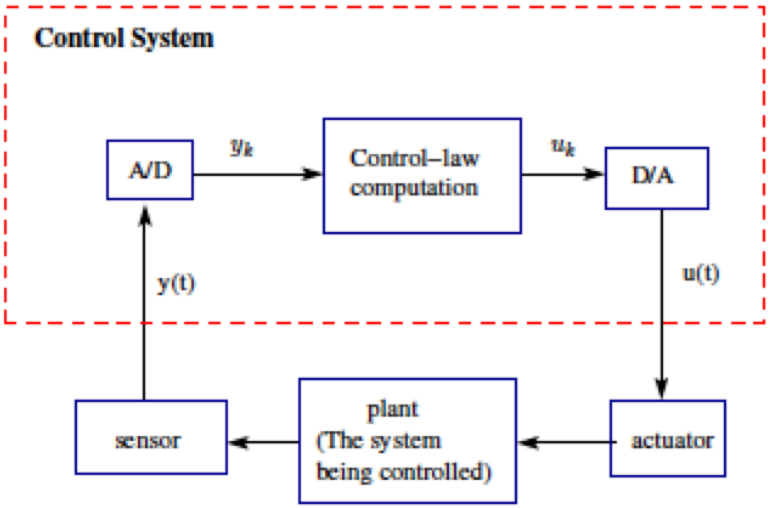
set timer to interrupt periodically with period T;

at each timer interrupt

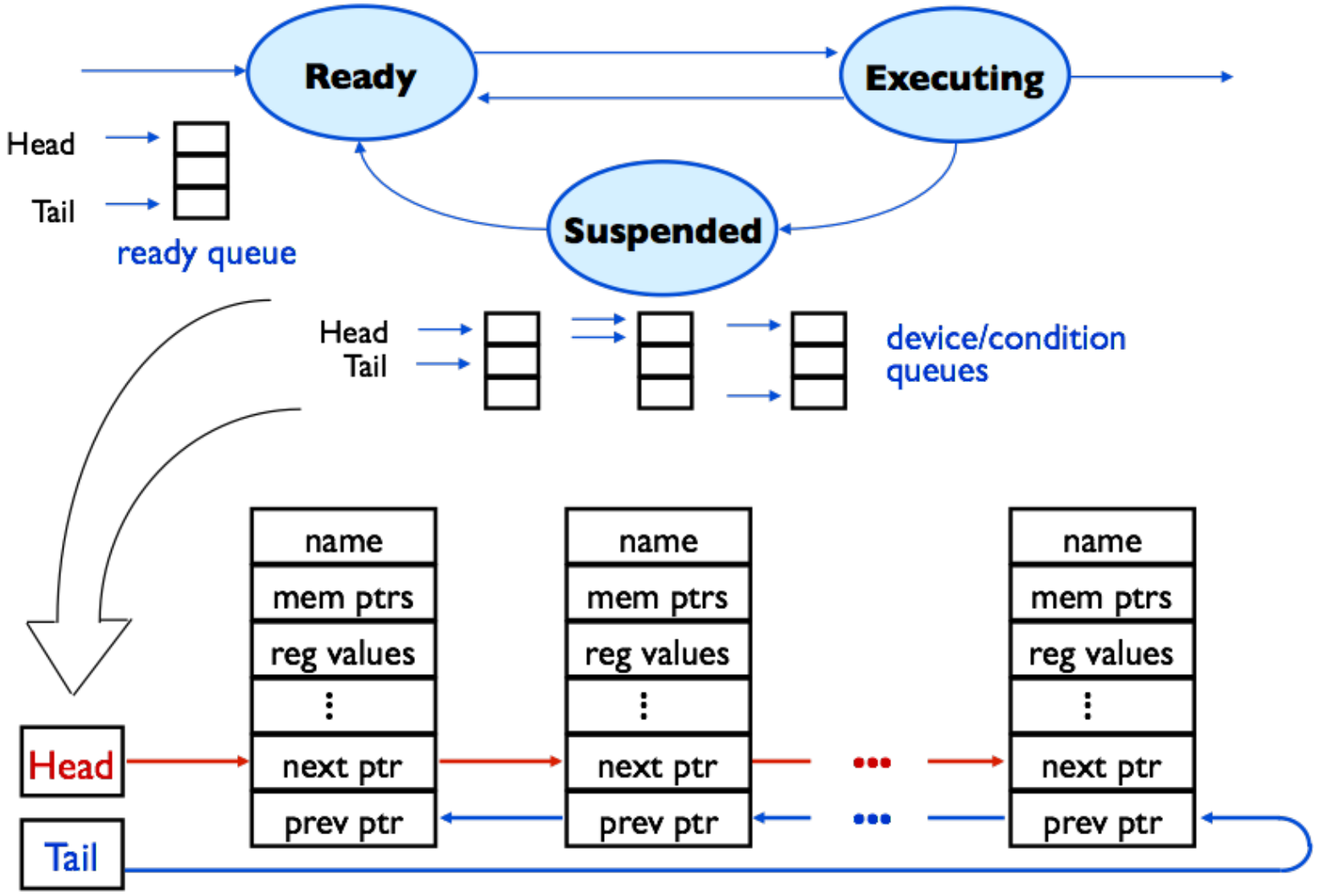
do

- perform analog-to-digital conversion to get y;
- compute control output u;
- output u and do digital-to-analog conversion;

od



Implementing and Managing State Transitions



Increasing design complexity + Stringent time-to-market requirements ☞Reuse of components

Reuse requires knowledge from previous designs to be made available in the form of **intellectual property** (IP, for **SW & HW**).

- HW
- Operating systems
- Middleware (Communication libraries, data bases, …)
- ….

Models of computation considered in this course

Communication/ local computations	Shared memory	Message passing Synchronous | Asynchronous	
Undefined components	Plain text, use cases | (Message) sequence charts		
Communicating finite state machines	StateCharts		SDL
Data flow			Kahn networks, SDF
Petri nets		C/E nets, P/T nets, …	
Discrete event (DE) model	VHDL, Verilog, SystemC, …	Only experimental systems, e.g. distributed DE in Ptolemy	
Imperative (Von Neumann) model	C, C++, Java [libraries]	C, C++, Java with libraries CSP, ADA |	

Summary

- General requirements for embedded operating systems
  - Configurability
  - I/O
  - Interrupts
- General properties of real-time operating systems
  - Predictability
  - Time services
  - Synchronization
  - Classes of RTOSs,
  - Device driver embedding
- Communication middleware
  - OSEK/VDX COM, CORBA, MPI

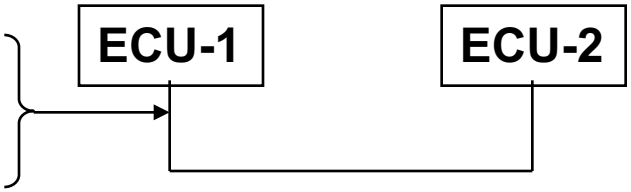
Spare Slides

OSEK/VDX COM

OSEK/VDX COM

- is a special communication standard for the OSEK automotive OS Standard
- provides an “Interaction Layer” as an API for internal and external communication via a “Network Layer” and a “Data Link” layer (some requirements for these are specified)
- specifies the functionality, it is not an implementation.

© P. Marwedel, 2011

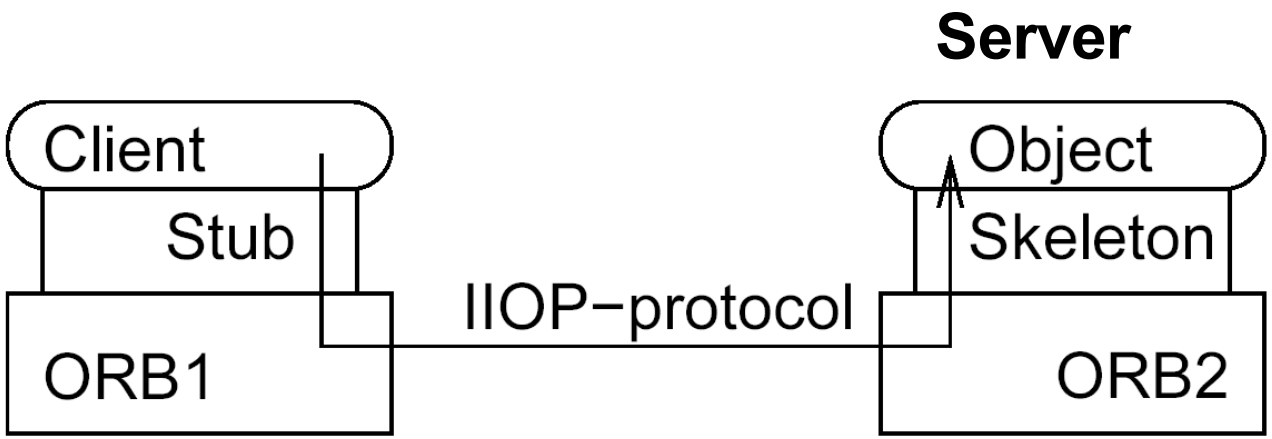


CORBA
(Common Object Request Broker Architecture)

Software package for access to remote objects;

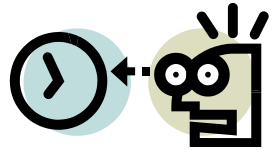
Information sent to Object Request Broker (ORB) via local stub.

ORB determines location to be accessed and sends information via the IIOP I/O protocol.



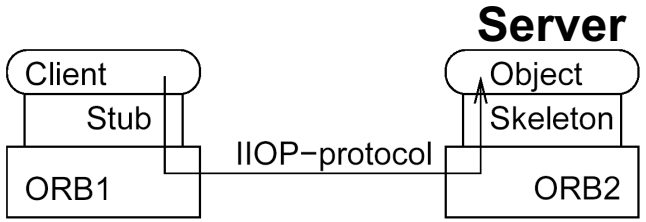
Access times unpredictable.

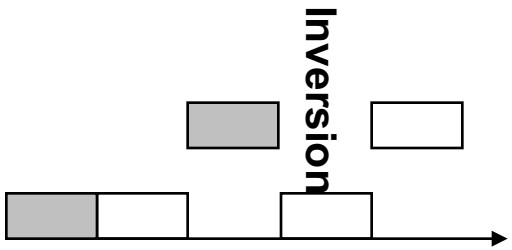
Real-time (RT-) CORBA



RT-CORBA

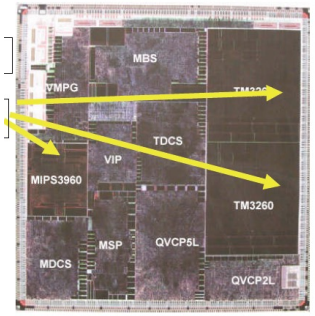
- provides *end-to-end predictability of timeliness in a fixed priority system*.
- *respects thread priorities between client and server for resolving resource contention*,
- provides thread priority management,
- provides priority inheritance,
- bounds latencies of operation invocations,
- provides pools of preexisting threads.





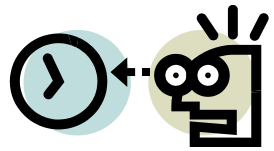
Message passing interface (MPI)

- Asynchronous/synchronous message passing
- Designed for high-performance computing
- Comprehensive, popular library
- Available on a variety of platforms
- Mostly for homogeneous multiprocessing
- Considered for MPSoC programs for ES;
- Includes many copy operations to memory (memory speed ~ communication speed for MPSoCs); Appropriate MPSoC programming tools missing.

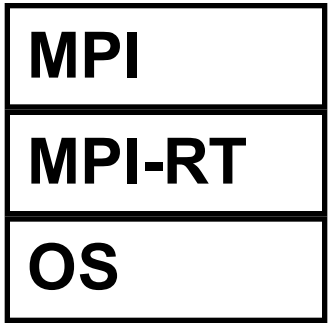


http://www.mhpcc.edu/training/workshop/mpi/MAIN.html#Getting_Started

RT-issues for MPI



- MPI/RT: a real-time version of MPI [MPI/RT forum, 2001].
- MPI-RT does not cover issues such as thread creation and termination.
- MPI/RT is conceived as a potential layer between the operating system and standard (non real-time) MPI.



Evaluation of MPI

Explicit

- Computation partitioning
- Communication
- Data distribution

Implicit

- Synchronization (implied by communic., explicit possible)
- Expression of parallelism (implied)
- Communication mapping

Properties

- Most things are explicit
- Lots of work for the user ("*assembly lang. for parallel prog.*")
- doesn't scale well when # of processors is changed heavily

Based on W. Verachtert (IMEC): *Introduction to Parallelism,* tutorial, DATE 2008