

## **Tutorial C2**

# **Real time and Digital Signal Processing Embedded Software**

Eric Verhulst  
Eonic Systems  
Belgium

Eric.Verhulst@eonic.com  
www.eonic.com

Peter Marwedel  
University of Dortmund  
Germany

marwedel@acm.org  
ls12-www.cs.uni-dortmund.de

# Welcome!

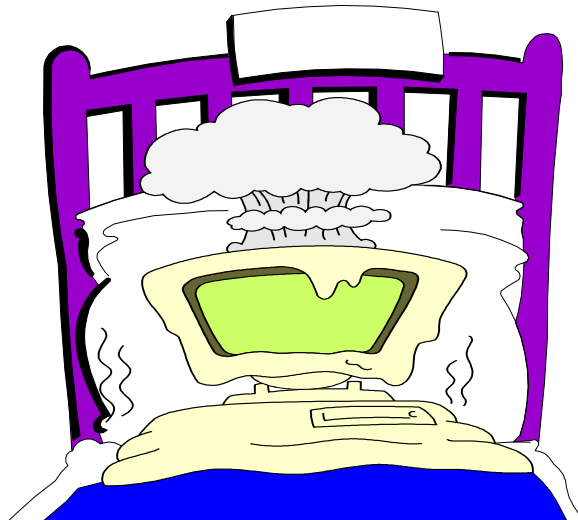
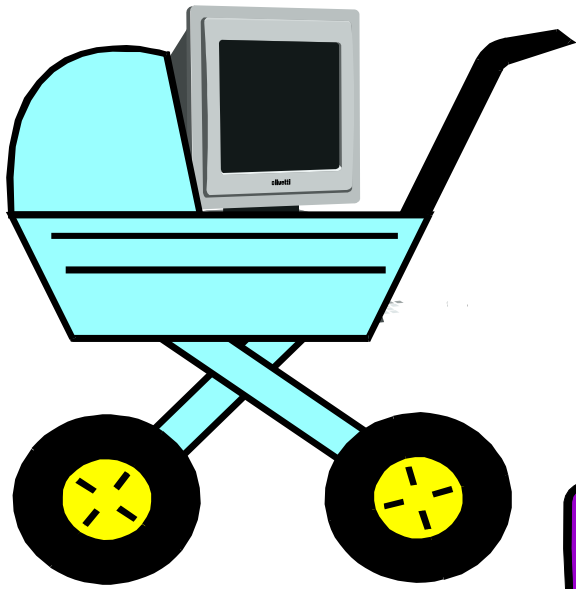
## Outline of the Tutorial:



- ➔ ● Introduction
- Real-time Operating Systems
  - Multi-tasking := Process Oriented Programming
  - Scheduling := meeting deadlines and minimizing CPU time
- Generation of Efficient Program Code
  - Code Compression
  - Compilation Techniques for DSP Processors
- Conclusion

# Context of the current talk: embedded systems

Are these the embedded systems we talk about?

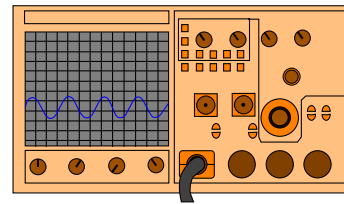
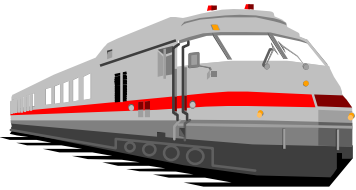


**No, certainly not today !  
Maybe tomorrow ?**

# Embedded Systems

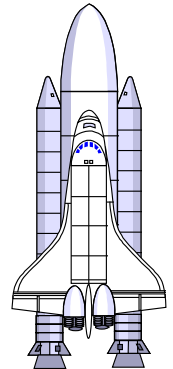
Embedded (real time) systems :=

- Systems that must behave correctly in time (it's not enough to produce a correct logical result)



Characteristics:

- fixed set of applications
- not recognised as information processing
- systems reading, processing and controlling physical quantities
- frequently safety-critical, must be reliable
- must be cost-efficient



# Focus on Embedded Processors & Software

---

## Why?

- **Flexibility** ←
- ☞ **Late design changes possible**
- ☞ ...
- **For (standard) processors: Reuse!**
- ☞ **Reduced time-to-market**
- ☞ **Power, size and cost constraints !**

**At the  
same  
time**



# What's the problem?

---

**If embedded systems are implemented mostly in software, then why don't we design them like software ?**

**In other words, why do we design them like hardware first ?**



# Some open problems

---

- **How do we capture the required behaviour of complex systems ?**
- **How do we validate specifications?**
- **Should programming language provide real-time features ?**
- **How do we translate specifications efficiently into implementation?**
- **How can we check that we meet real-time constraints?**
- **How do we validate embedded real-time software?  
(large volumes of data, testing may be safety-critical)**
- **Do software engineers ever consider electrical power?**
- **Which real-time operating system (RTOS) meets requirements?**

...

# Outline of the Tutorial:



- Introduction

- ➔ ● Real-time Operating Systems

- Multi-tasking := Process Oriented Programming
- Scheduling := meeting deadlines and minimizing CPU time

- Generation of Efficient Program Code

- Code Compression
- Compilation Techniques for DSP Processors

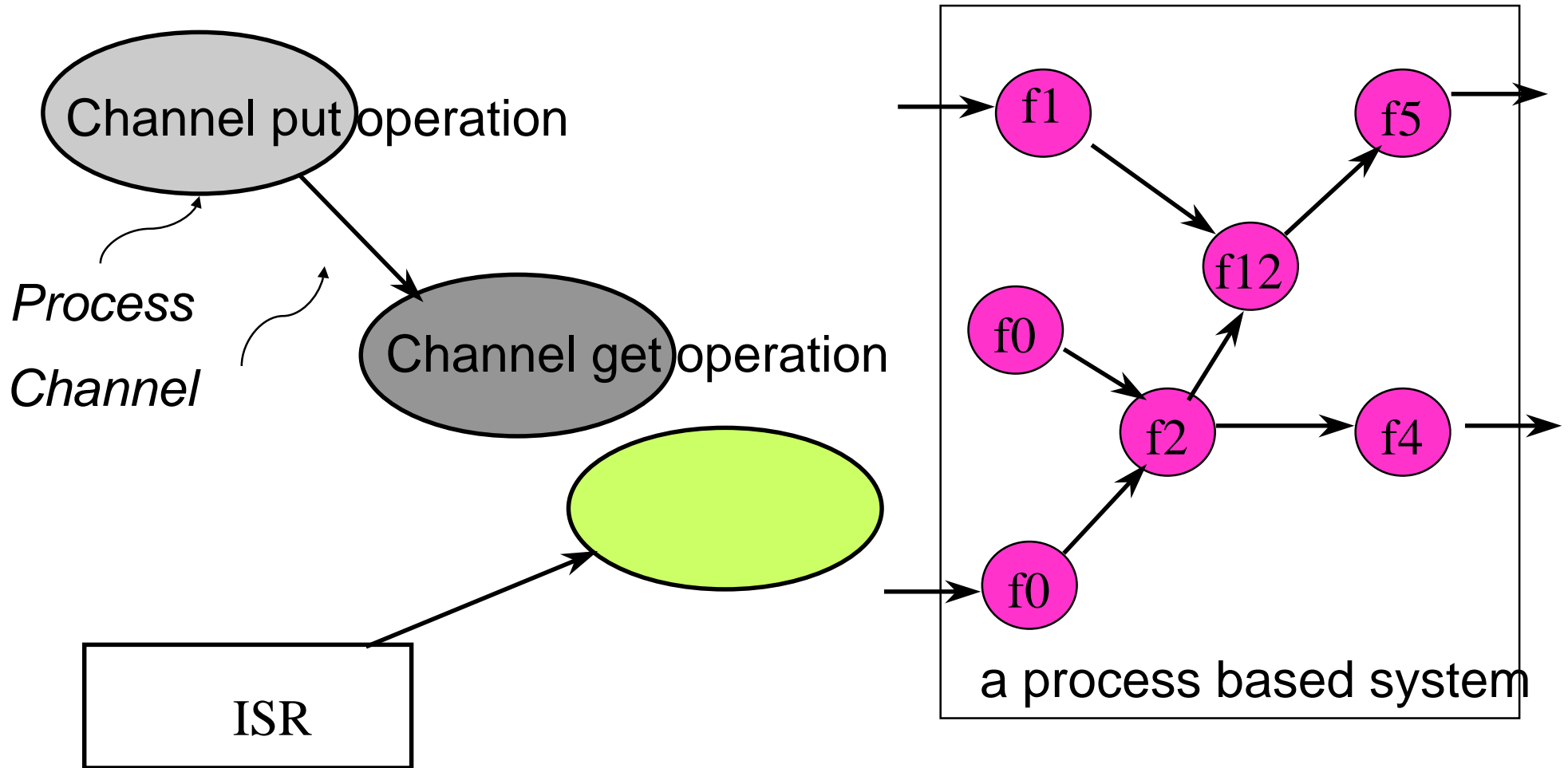
- Conclusion

# Process Oriented Programming

---

- Programming languages have two origins :
  - abstraction layer for HW (HEX -> ASM -> C-> C++)
  - specification methods with math/logical background
  - Both are mostly sequential by nature
  - Exceptions : e.g. CSP (Communicating Sequential Processes (C.A.R. Hoare))
- Object Oriented Programming :
  - information hiding for functions and datastructures
- RTOS : originated as an industry driven tool
  - models closely the real [parallel] embedded world

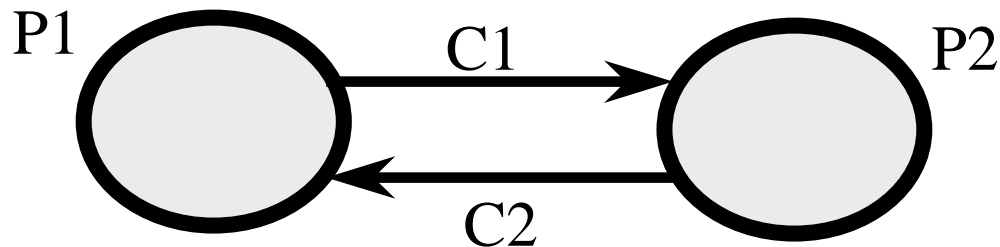
# System level design inspired by CSP : processes are abstract building blocks



Channel operation to interface with HW and other processes

# CSP explained in occam

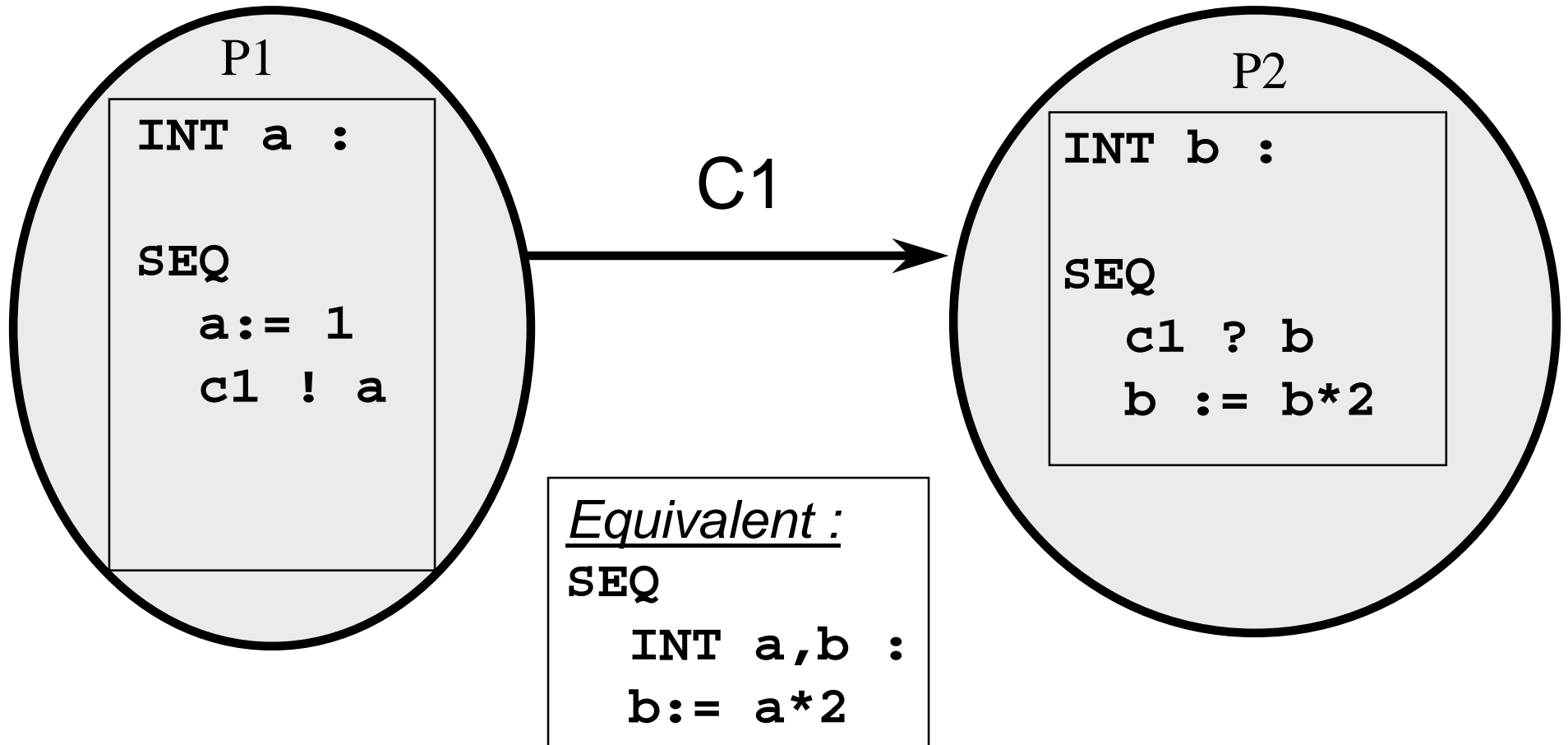
```
PROC P1, P2 :  
  CHAN OF INT c1, c2 :  
  
  PAR  
    P1  
    P2
```



Needed :  
Support for :

- context (= state)
- communication

# A small parallel program



# Virtuoso's Virtual Single Processor model

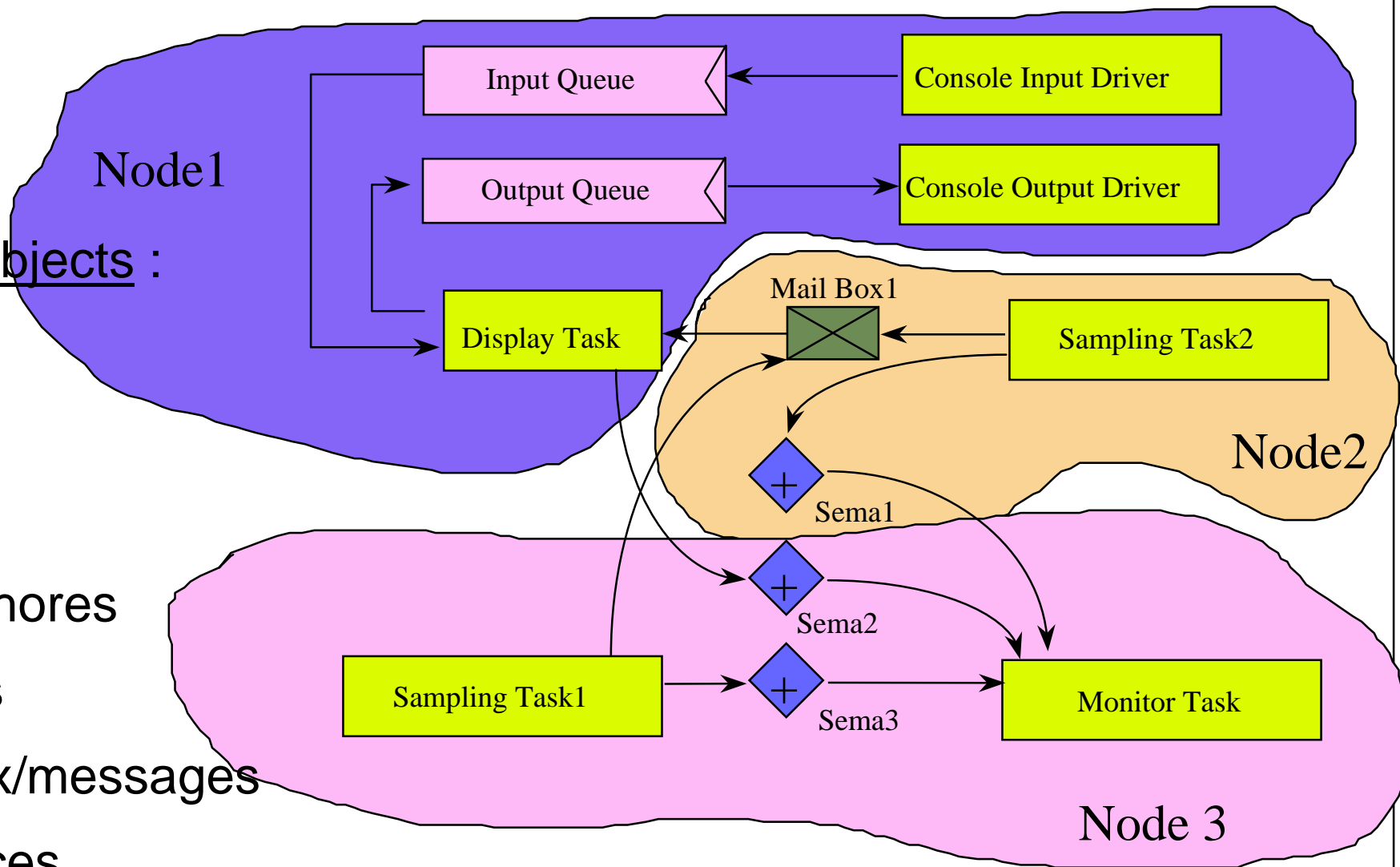
---

- CSP is nice but not really what engineers want in the real world
- Virtuoso's RTOS :
  - pre-emptive multi-tasking
  - general purpose, higher level comm services
  - distributed semantics and implementation
  - ANSI C, C++, [ADA], [JAVA]
  - layered architecture
  - prioritized, packet switching communication layer

# Virtuoso's VSP : a pragmatic CSP : distributed semantics

## RTOS Objects :

- tasks
- drivers
- events
- semaphores
- queues
- mailbox/messages
- resources



# What is an RTOS?

---

- Real-Time Operating System
  - an OS that operates in real-time
- But what's an "OS"?
- And what is "Real-Time"?

# What is an OS?

---

- Big range:
  - from DOS
  - to Unix/Windows NT
- RTOS: normally for embedded applications
  - limited memory & peripherals
- “OS” for today:
  - multi-tasking: multiple tasks run in “parallel”
  - allows efficient inter-task communication
  - allows efficient communication with devices

## What is Real-Time?

---

“The **correctness** of a real-time system does not only depend on the **logical** result of the computation but also on **the time** at which the results are produced.”

# What is Real-Time?

---

- **Distinction between “Hard” & Soft” real-time**
- **Key feature: “reaction time”**
  - system must have processed on an event before a deadline
    - copying one word of data & clear interrupt
    - perform complex calculations
- **Hard real-time :**
  - Real-time with strictly defined, absolute timing requirements.
    - Anti-collision system (often safety critical)
- **Soft real-time :**
  - Real-time with statistically defined requirements
    - E.g.: ticket reservation system (often subjective fairness policy)

# So, what is an RTOS?

---

- Programming system that allows the user to:
  - **communicate** with peripherals & other tasks
  - react in a **deterministic** way on external events
  - **share the CPU** time and resources timely between competing processing tasks
- With asynchronous events :
  - **dynamically scheduling** OS is needed
- With only synchronous events :
  - time or data driven **static schedule** can do the job
  - notion of modularity lost
  - potential side-effects when changes are made

## List of essential RTOS features

---

- Size matters (often) in embedded systems
- Prioritization of tasks
- Pre-emption of tasks
- Priority inheritance to avoid deadlock
- Reaction time to external & internal events
  - interrupt latency to various levels
  - task switch time
- Side-effect free behavior of synchronisation and communication services

# Size matters

---

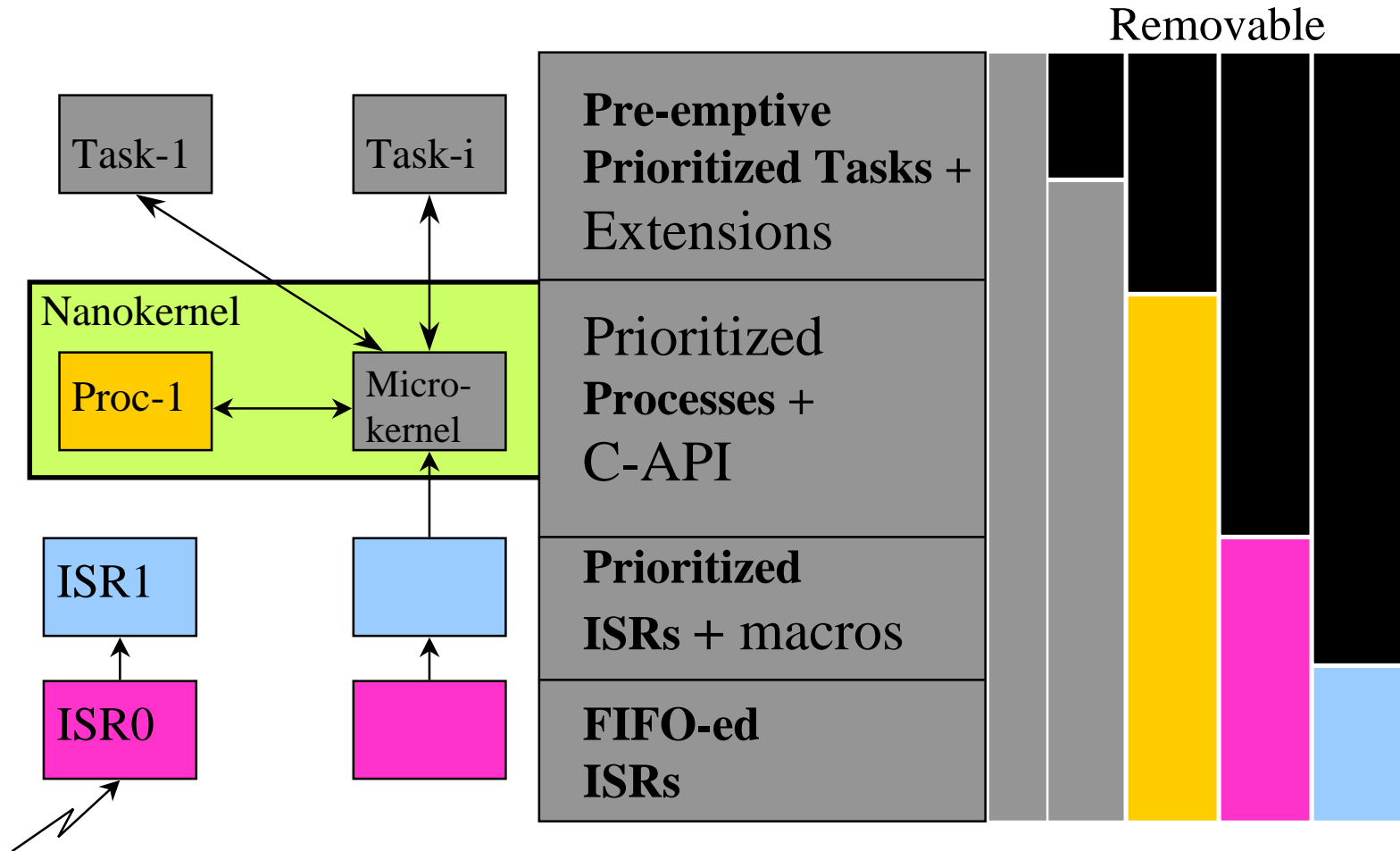
- DSP and ASIC processors have limited internal memory
- Internal memory is 0 wait state, multi-ported, wide bus
- External memory has wait states, narrow bus
- External memory sometimes on a shared bus
  - wait states are dependent on actions of other devices
- Running from external vs. internal
  - factor 3 to 15 performance difference!

# Size matters

---

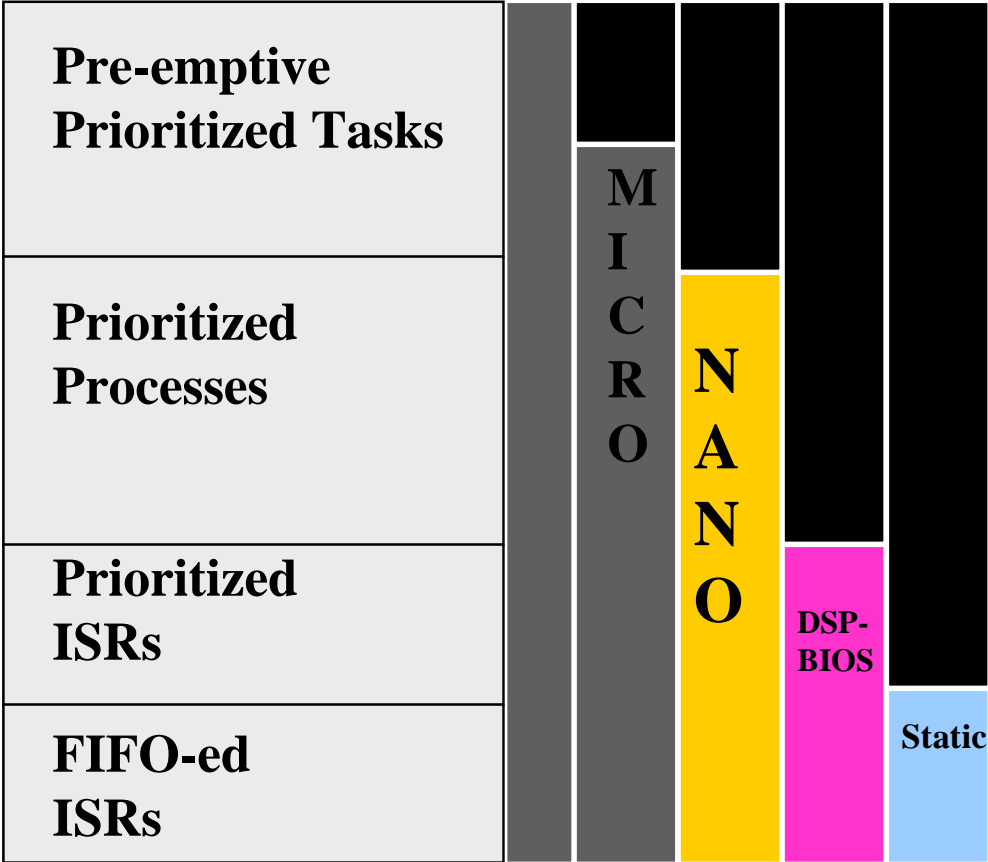
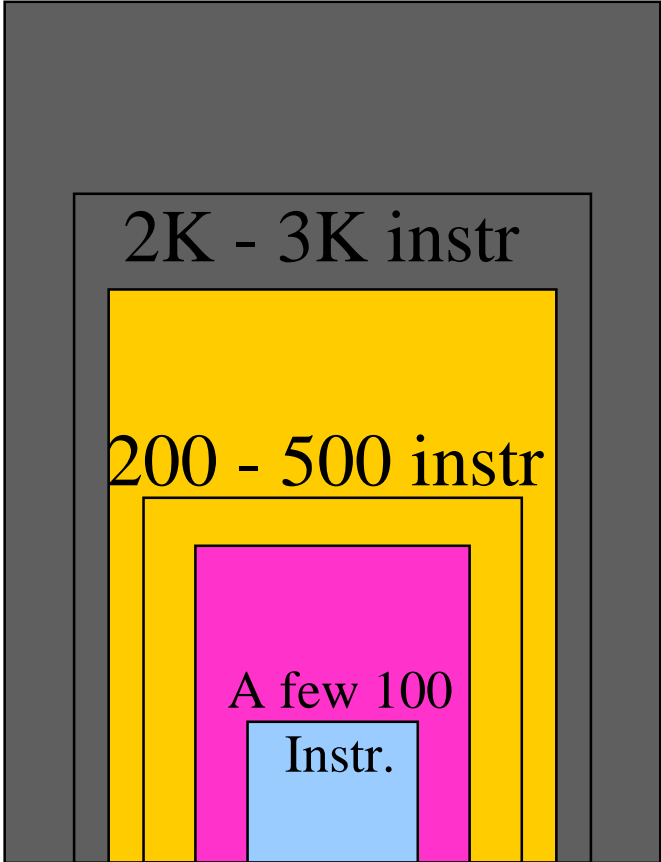
- Internal program memory:
  - from 6K to 40K instructions for typical high-end DSP
- RTOS + user code should fit in this!
  - OS for DSP can only take between 2K and 10K instructions
- Logical consequence:
  - Embedded RTOS has bounded capabilities
  - Modular architecture and layering helps
  - Optimal feature set for target domain

# Virtuoso v.4.x : SoftStealth™ Technology



# Virtuoso v.4.0 : SoftStealth™ Technology

+/- 10K instr. Max.



## Microkernel kernel class contributions

Code size	SHARC	%	C6201	%
<b>RMA (task scheduling)</b>	2742	29.98%	3347	34.80%
<b>System calls</b>	996	10.89%	574	5.97%
<b>FIFO Queue</b>	982	10.74%	840	8.73%
<b>Memory Maps</b>	395	4.32%	416	4.33%
<b>Mailbox</b>	1105	12.08%	1424	14.81%
<b>Memory Pools</b>	1006	11.00%	1104	11.48%
<b>Resources</b>	661	7.23%	648	6.74%
<b>Semaphores</b>	1258	13.76%	1264	13.14%
<b>Total Kernel</b>	9145	100%	9617	100%
<b>UNIT (instructions)</b>	48bit		32bit	
Virtuoso Implementation	VSP		SP	

## Prioritization & pre-emption

---

- Scheduling of higher priority task: immediately
  - Time-slicing is clearly not enough!
  - Thus: tasks must have associated priority
    - priority or deadline driven scheduler
  - Waiting for OS call to schedule higher priority task is not enough!
- Thus: pre-emption of currently running task
  - implies saving context
  - not the same as “interruption” (e.g. DSP BIOS)

# Avoiding deadlocks

---

- Deadlocks can occur:
  - low priority task “owns” a resource
  - higher priority task needs resource to run
  - middle priority task is ready to run
- Result:
  - middle priority task will run
  - higher priority task will miss its deadline
- Solution: priority inheritance
  - Task that holds a resource lock inherits priority of task requesting ownership

# Typical API of RTOS

---

- Message passing
  - message header + associated data buffer (variable size)
  - Major distinction: copy or non-copy behavior
    - non-copy: must also block on senders' side, except if memory comes from free store
  - One mailbox can be used for communication between multiple tasks
  - Not always present, can be emulated with pushing data pointers in queues

# Typical API of an RTOS : semantics of a service

---

- **Blocking :**
  - wait (eventually forever) till service completed
  - CSP behavior
  - blocking can be limited by time-out
- **Non Blocking :**
  - don't wait for service to complete
  - requires often polling
  - better is use of callback function but tricky
- **Group or set operations**
  - e.g. start a group of tasks

# Typical API of RTOS

---

## ● Timers

- Program a (software) timer to do something
  - at a given time: one-shot timer
  - periodically
- Call a function
- Signal a semaphore
- On a DSP Resolution is higher and more accurate than a “commercial” OS

# Typical API of RTOS

---

- Handle interrupts
  - Enable/disable interrupts
  - Install interrupt handlers
  - Interrupt handlers must be written as per requirements of OS!
- Major distinction between (RT)OSes
  - interrupt handling model
  - Key to (real-time) performance of system
  - Crucial for data-driven DSP systems

# Implementing an RTOS

---

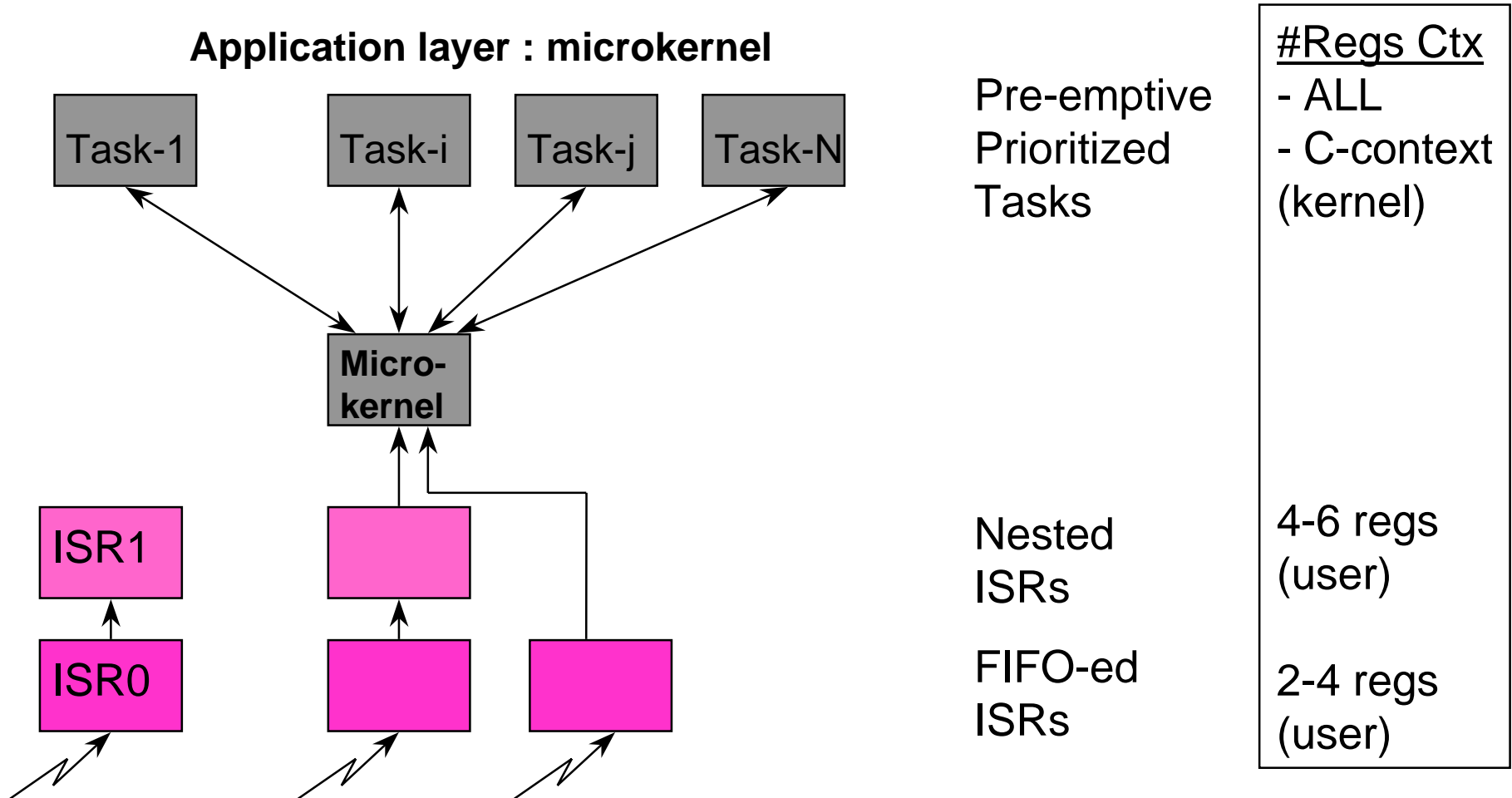
- Keep it small
  - select a small & orthogonal set of features
  - remove unused code
- Guarantee deterministic reaction to events
  - interrupt latency
  - task switch time

# Obtaining low task switch time

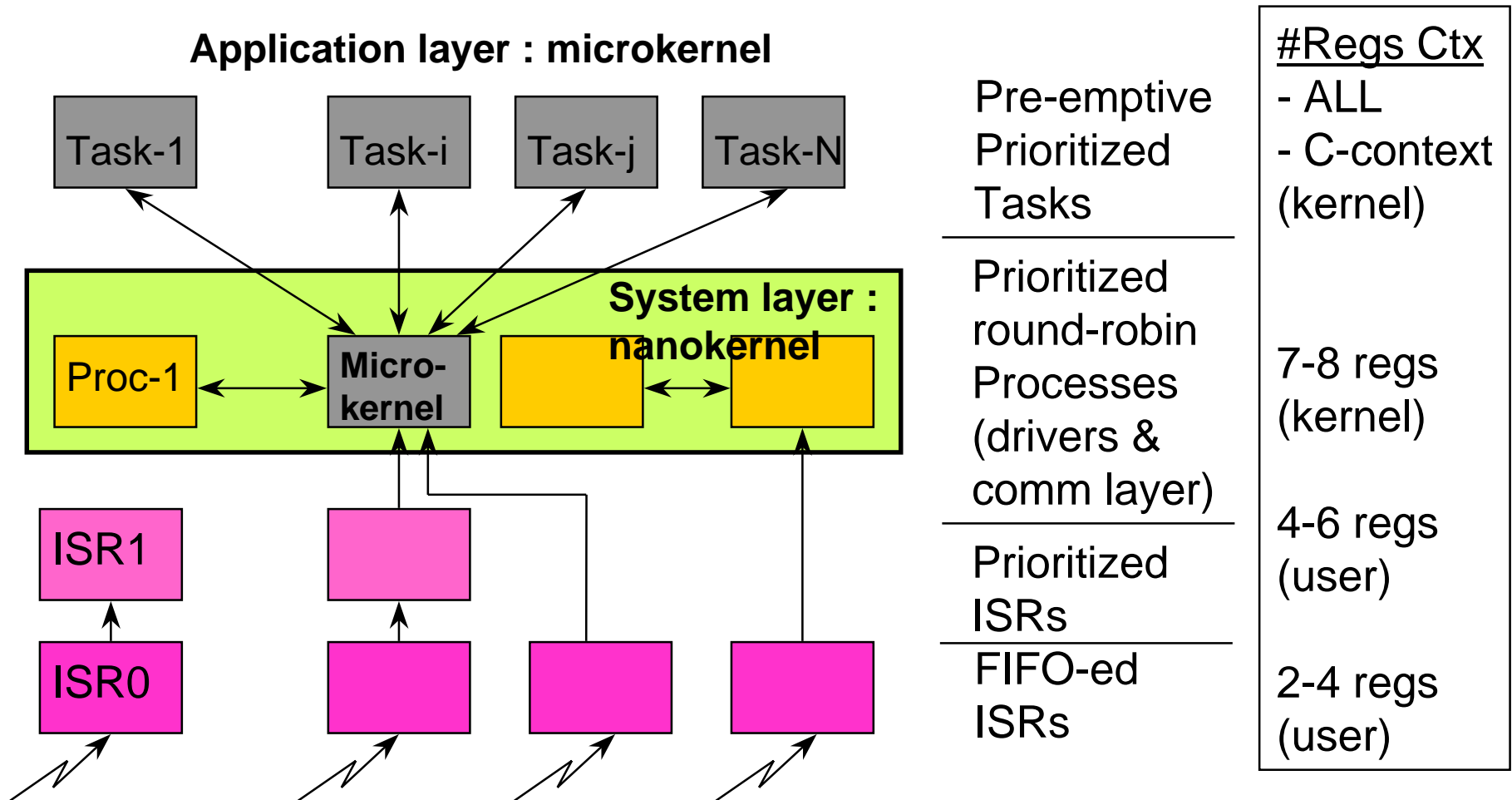
---

- Executing a kernel service *can* cause a task switch
  - put a message in a queue
  - other task is waiting for a message on that queue
- Implementation aspects:
  - make kernel entry as fast as possible:
    - switch minimum number of registers
    - if no task switch is necessary, only minimal context must be restored
  - optimize kernel services
    - optimal: assembly with minimal context
    - but: portability requires C...

# Typical RTOS architecture



# Virtuoso's four layered architecture for DSP



# Functional levels

---

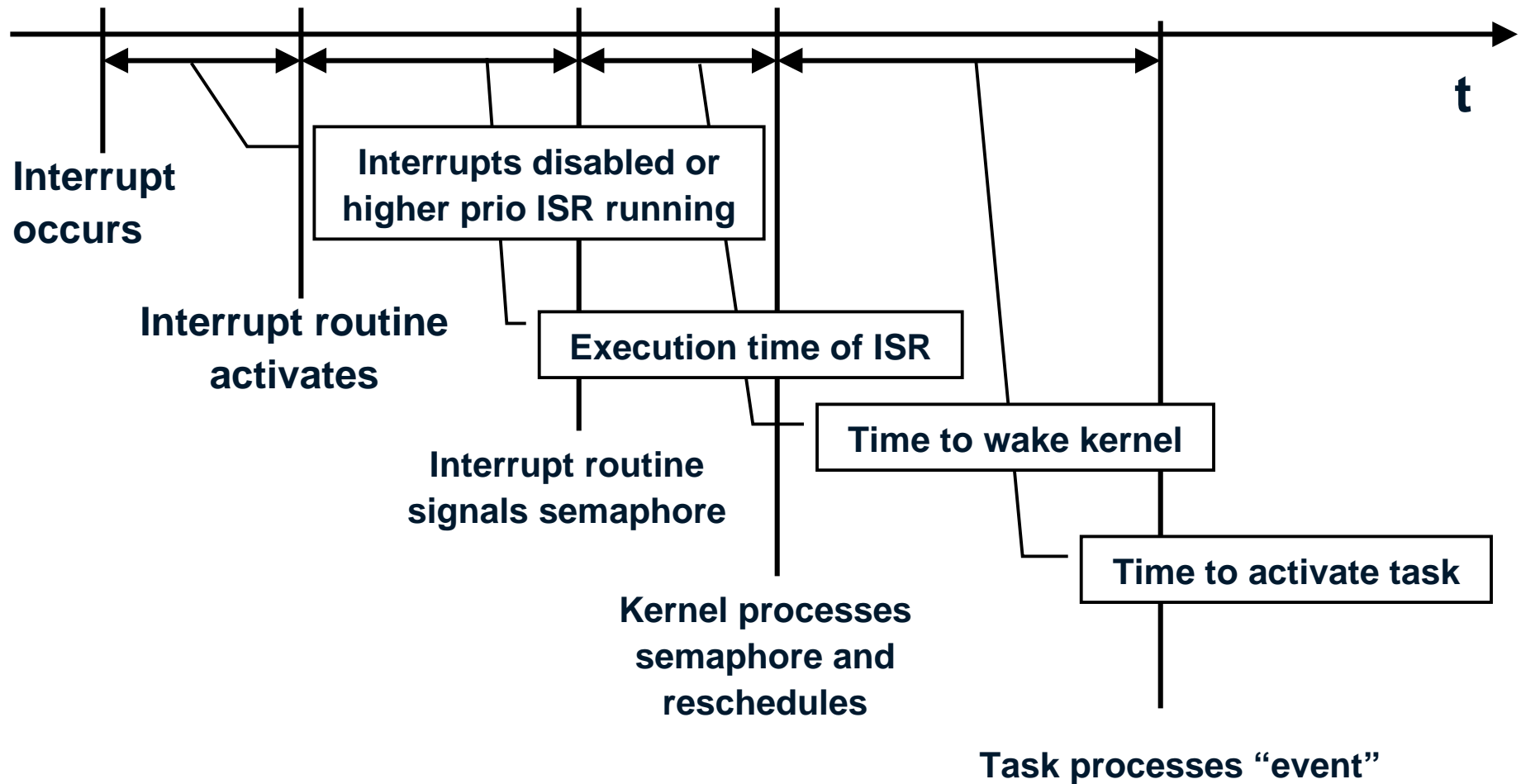
- **ISR0 :**
  - Interrupt Service Routines (ISR) with interrupts globally disabled ( = HW level support).
- **ISR1 :**
  - ISR with a status of globally enabled interrupts.
- **Nanokernel :**
  - Round robin scheduled processes + channels”
- **Microkernel :**
  - Pre-emptively scheduled tasks (full C contxt + ... )

## Obtaining low interrupt latency

---

- Interrupt latency: Eonic's definition
  - Time between moment the interrupt occurs and the first useful instruction of the code that handles the interrupt

# Interrupts: the road to handling



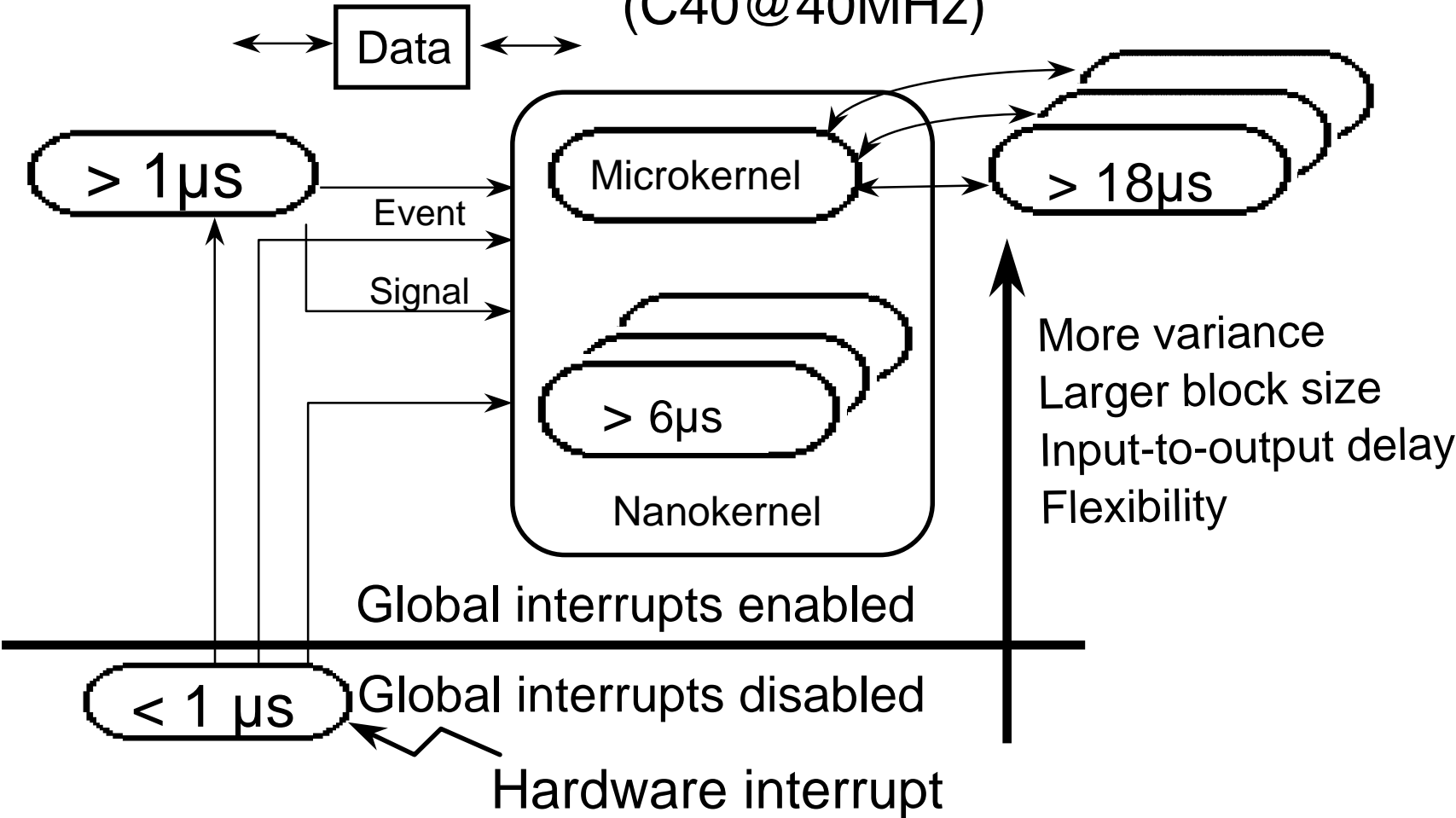
# ISR latency

---

- Depends to which level
  - handling completely from ISR or from task
- Time needed depends on:
  - other asynchronous interrupts triggered
  - maximum time kernel (or other code) disables interrupts
  - time to process sema signal
  - time to reschedule
  - time to activate a task
- Conclusion: adding a badly written user ISR can ruin the performance of the whole system!

# Choice of level

A function of the time-resolution and data block size required  
(C40@40MHz)



# Multiprocessing RTOS

---

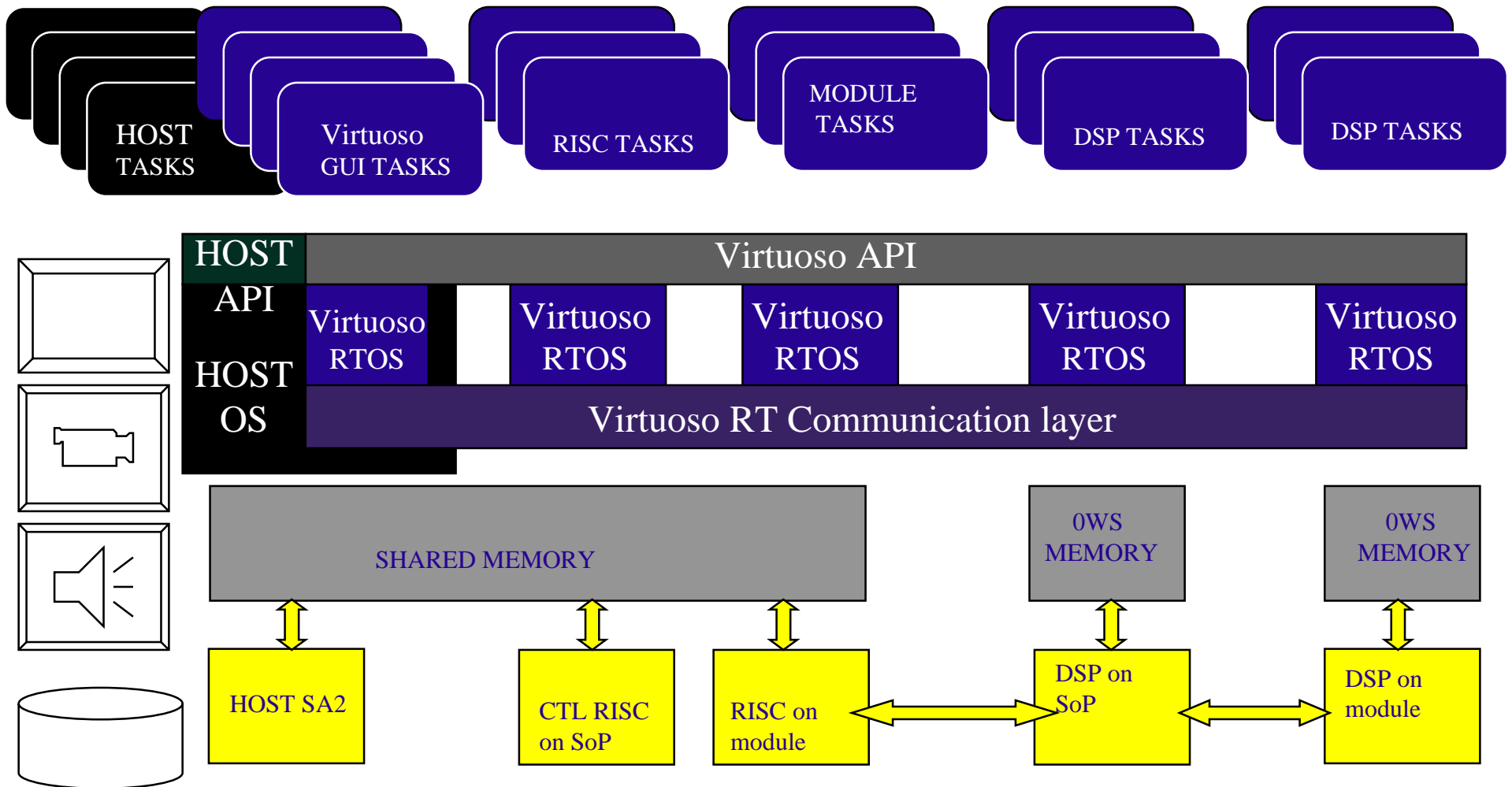
- High-end DSP needs more than one processor
- How do tasks on processors communicate?
  - OS should provide a communication layer
  - user code should be unaware of network topology
  - user code should be unaware where a task lives
    - compare with logical internet names vs IP numbers
- Heterogeneous architectures
  - RISC like controller + some DSP's at board level
  - DSP, RISC and FPGA cores on a SoP
- Sometimes multiple of these systems cooperate
- Challenge : make CSP methodology applicable to these big & complex systems...

# The VSP model

---

- VSP : Virtual Single Processor model
  - transparent parallel programming
    - cross development on any platform, incl. Development Host OS
    - portability
    - scalability, even on heterogeneous targets
  - distributed semantics :
    - program logic neutral to topology and object mapping
    - clean API provides for less programming errors
  - based on “CSP” (C.A.R. Hoare)
  - VSP = multitasking + message passing
  - VSP = process oriented programming
  - VSP = interfacing using communication protocols

# Virtuoso multicore : architecture



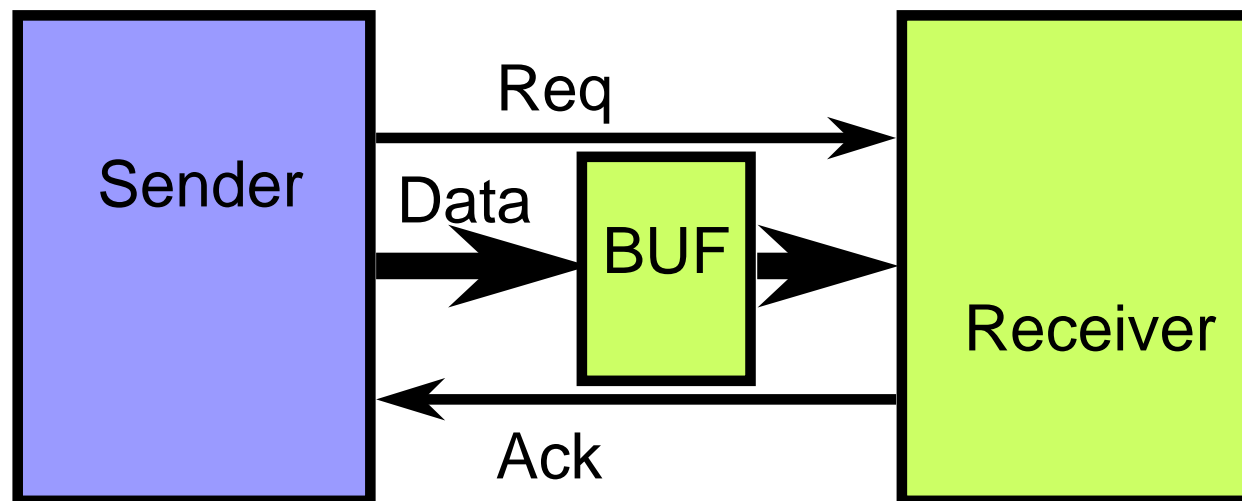
## Extension to FPGA

---

- Model can be extended to cover attached FPGA as well
- Needed :
  - C-level Design Tools (e.g. RT Designer by Frontier Design)
  - Pre-defined I/O blocks (typically FIFO's)

## CSP at the HW level

- Request/Ack protocol assures correct data transfer between async units, even at the register level
- Is like the mailbox mechanism



## RTOS objects : mapping onto HW

### Software

### Hardware



Task - Process

Logic State Machine



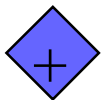
KS\_FifoPutW

FIFO memory



KS\_MsgPutW

shared memory + dma



KS\_SemaSignal

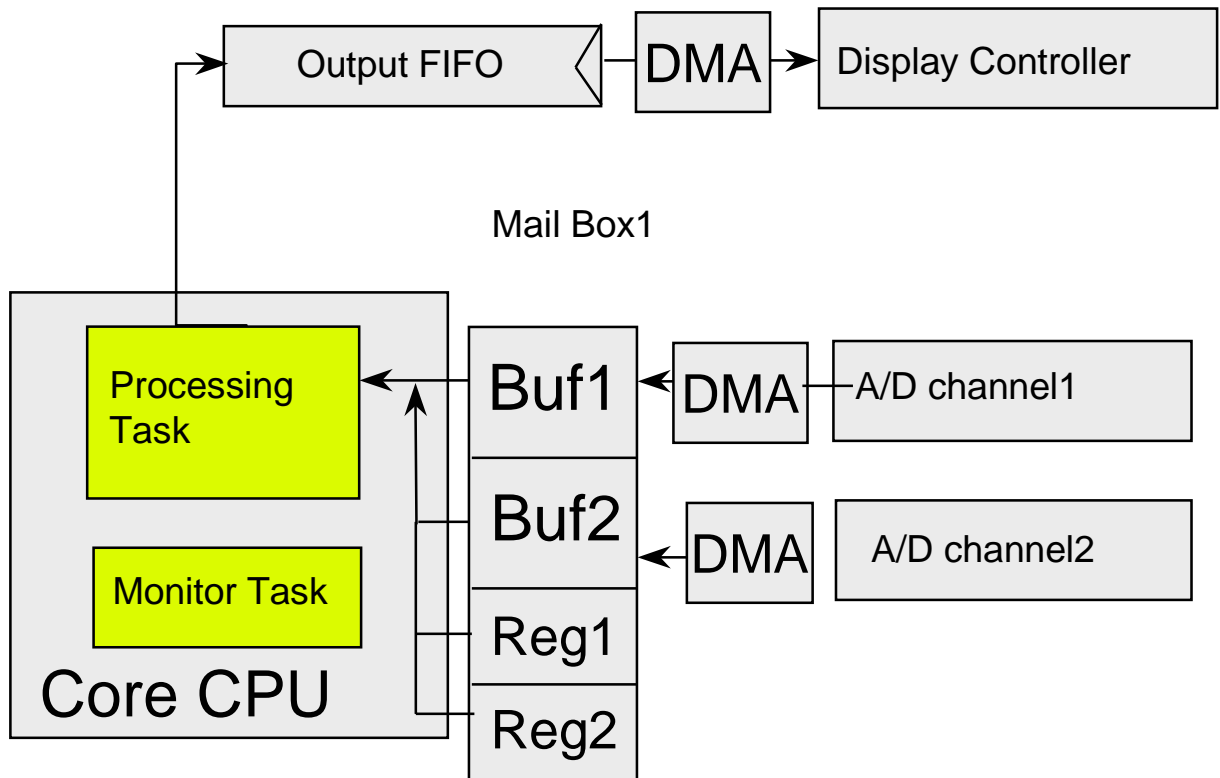
status register + counter

RTOS objects can be used for SW+HW system specification, simulation and implementation

# A SW-HW implementation

Steps :

1. Algorithm using MATLAB/SDT, Pegasus, ...
2. Simulate logic model with RTOS simulator on host OS like NT
3. Run RTOS program on target CPU
4. Map parts onto SW (C to ASM - binary)  
map parts onto HW (C to VHDL or RTL)



# Mapping the Virtuoso architecture into HW

---

- On today's processors :
  - Assembler required (a lot of it !)
    - No or few support for context switching
    - No or few support for prioritization :
      - no preemption support
      - often support for nesting of ISR
    - No or elementary support for communication
    - MMU and supervisor modes and other are for very coarse grain processes
  - Increasing latency due to pipelining
  - Increasing CPU to memory bandwidth gap

# The layers of an application : the forgotten needs

- The functional layers of an application

- I/O :

- Interrupt processing
    - Buffering data
    - Drivers (atomic datamovers)

<u>Level</u>	<u>#Reg Need</u>
--------------	------------------

ISR0	<b>(2-4 regs)</b>
------	-------------------

ISR1	<b>(4-6 regs)</b>
------	-------------------

Nanokernel process	<b>(8 regs)</b>
--------------------	-----------------

- Processing :

- Data driven : DSP
    - Control driven : decision logic

Task & coprocessors	<b>(all regs)</b>
---------------------	-------------------

Task	<b>(global)</b>
------	-----------------

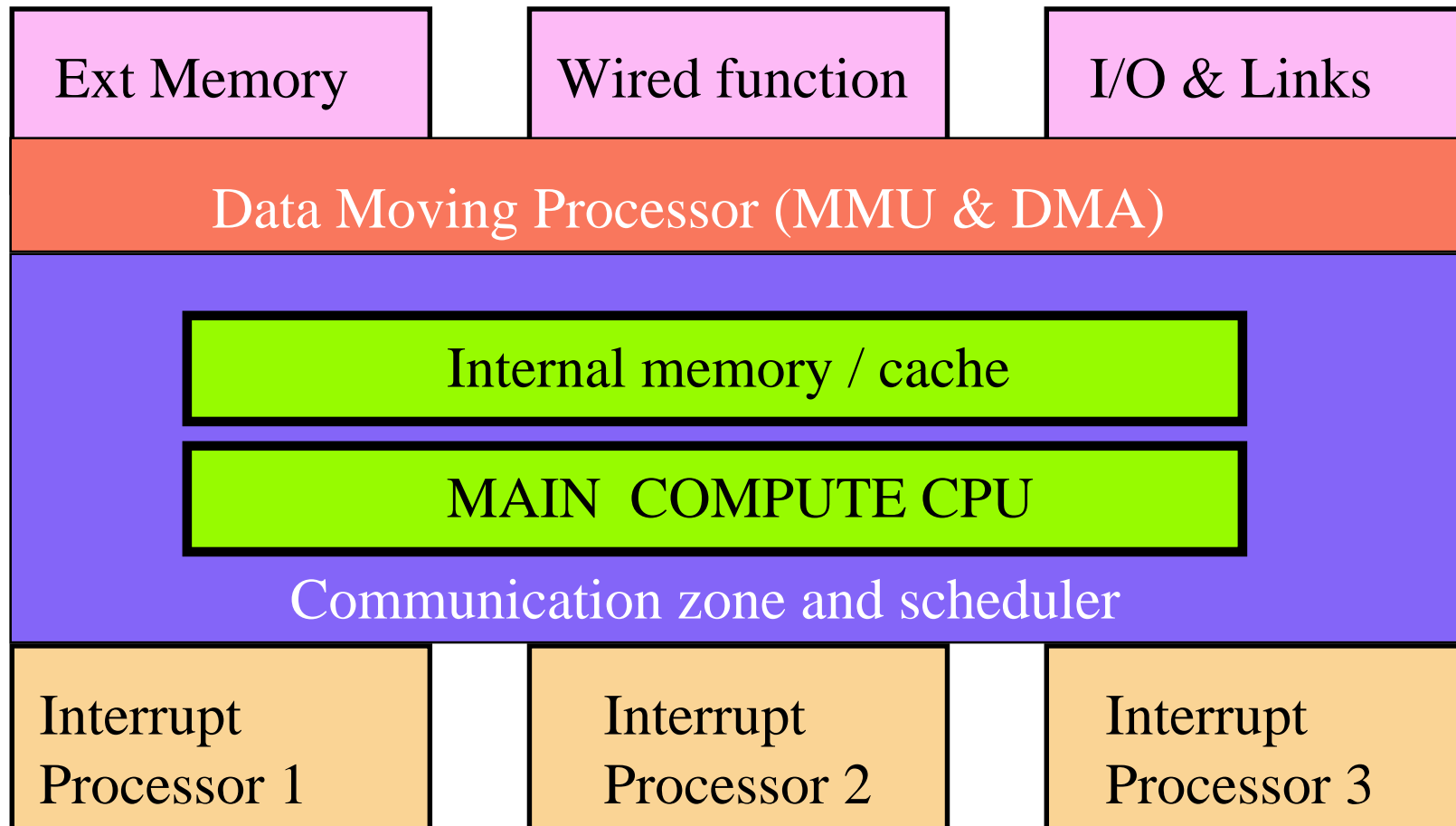
- Many implementation level issues are related to handling of context and preserving consistent state, while keeping the SW overhead down

# The state machine problem and its solution

---

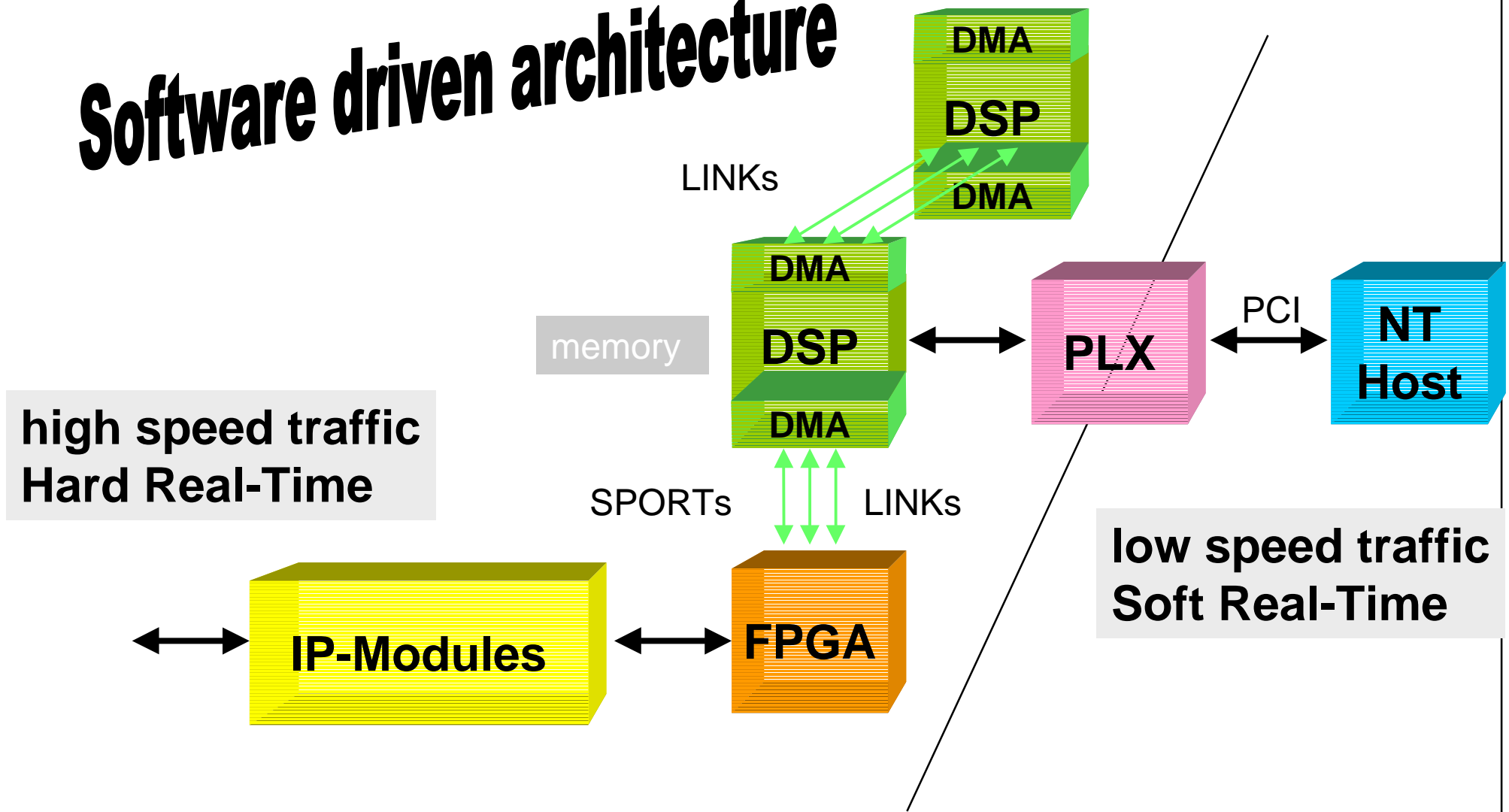
- Most processors are designed for throughput maximization
  - Single CPU handles processing and I/O
  - Large register context > < I/O & swapping
  - I/O “engines” (if any) are special purpose
- Increasing bandwidth gap CPU-memory
- Result : large, complex state machine
- Solution :
  - parallel CSP architecture at the CPU level
  - use asynchronous design techniques

# A CSP based processor that could be VSP friendly



# CompactPCI Atlas design as example

## Software driven architecture

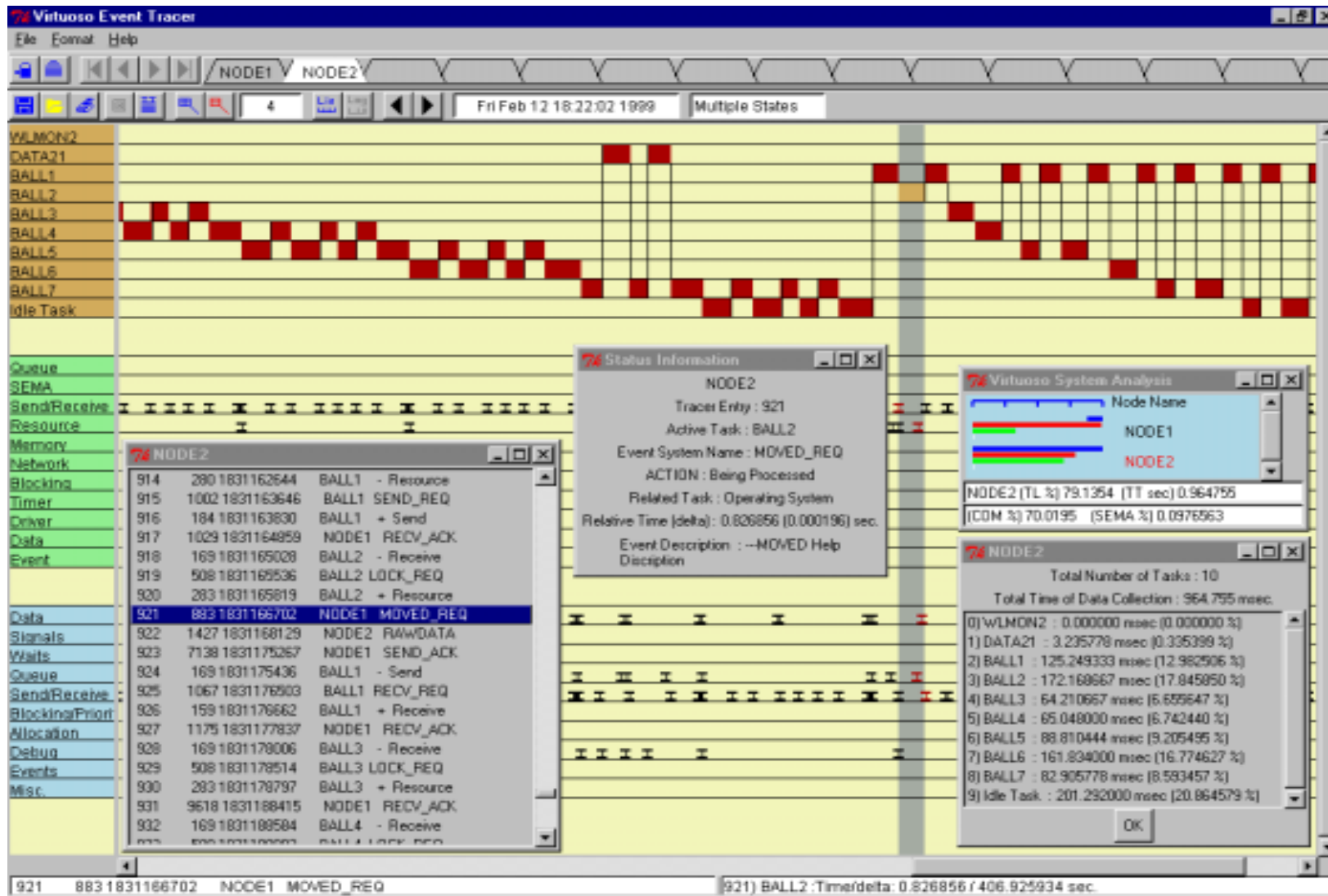


# Debugging and development tools

---

- Workload monitor using calibrated loop in idle task
- Emulator or J-TAG based low level debugger
  - break points
  - register and memory peek and poke
- Task level debugger using host connection or through J-TAG interface :
  - status of all kernel objects
  - statistics on use of kernel objects
  - timed execution trace using circular buffer or background dumping to host
  - adds overhead, difficult at lower levels

# Virtuoso Real-Time Tracing Monitor

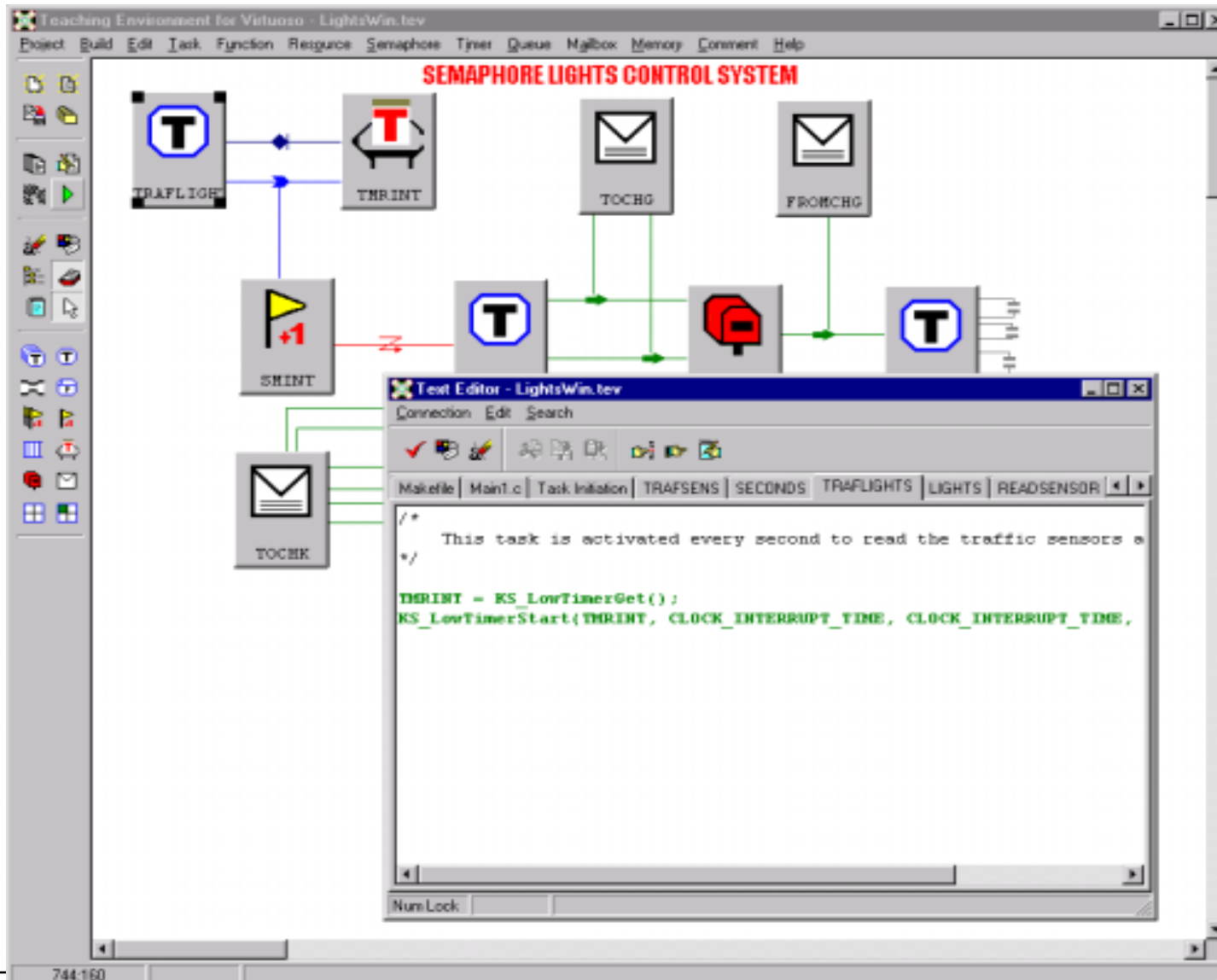


Logic Analyzer view of scheduling

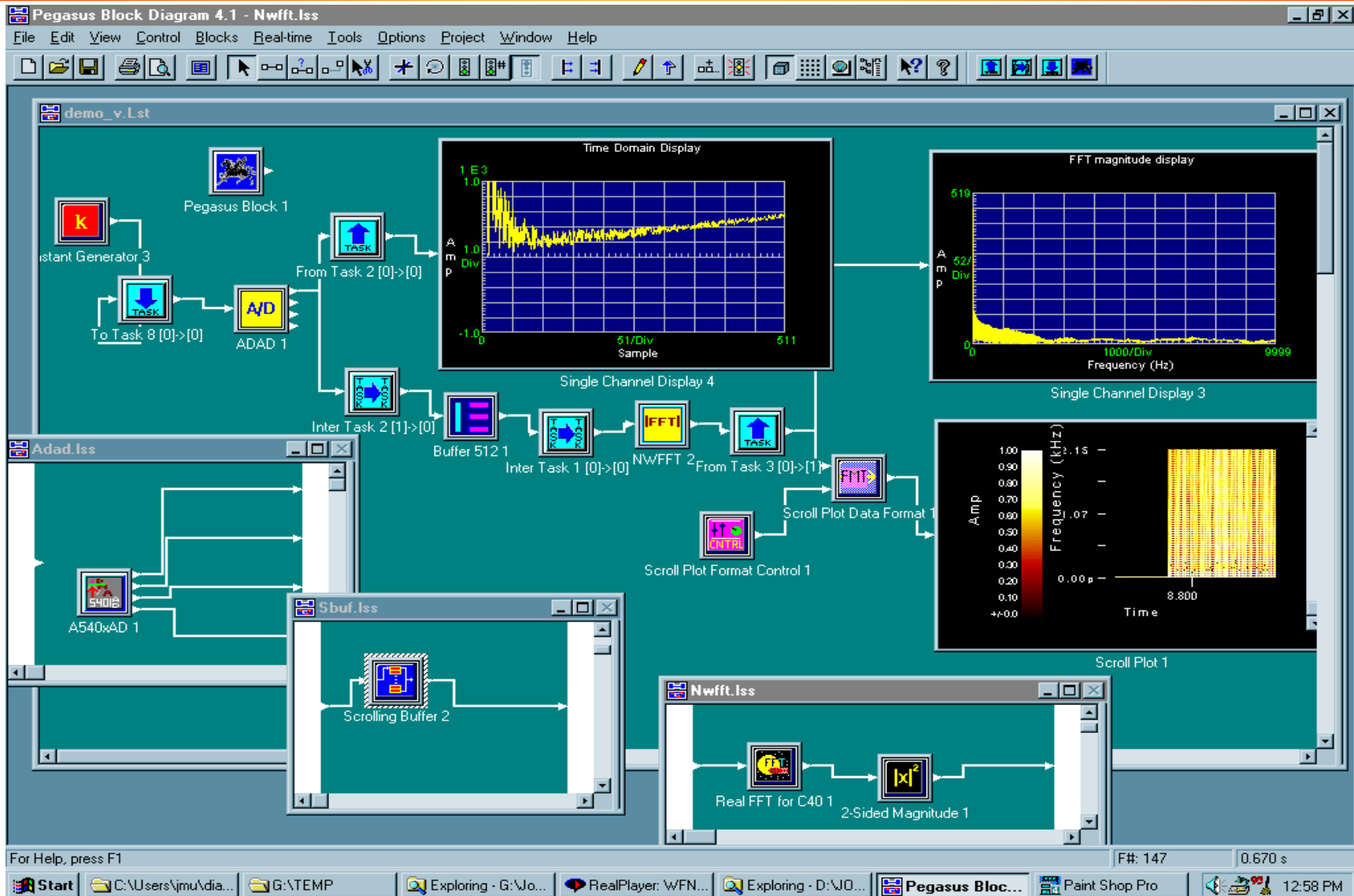
Details up to the cycle

Statistical Analysis

# Learning to think multi-tasking : Teaching Environment

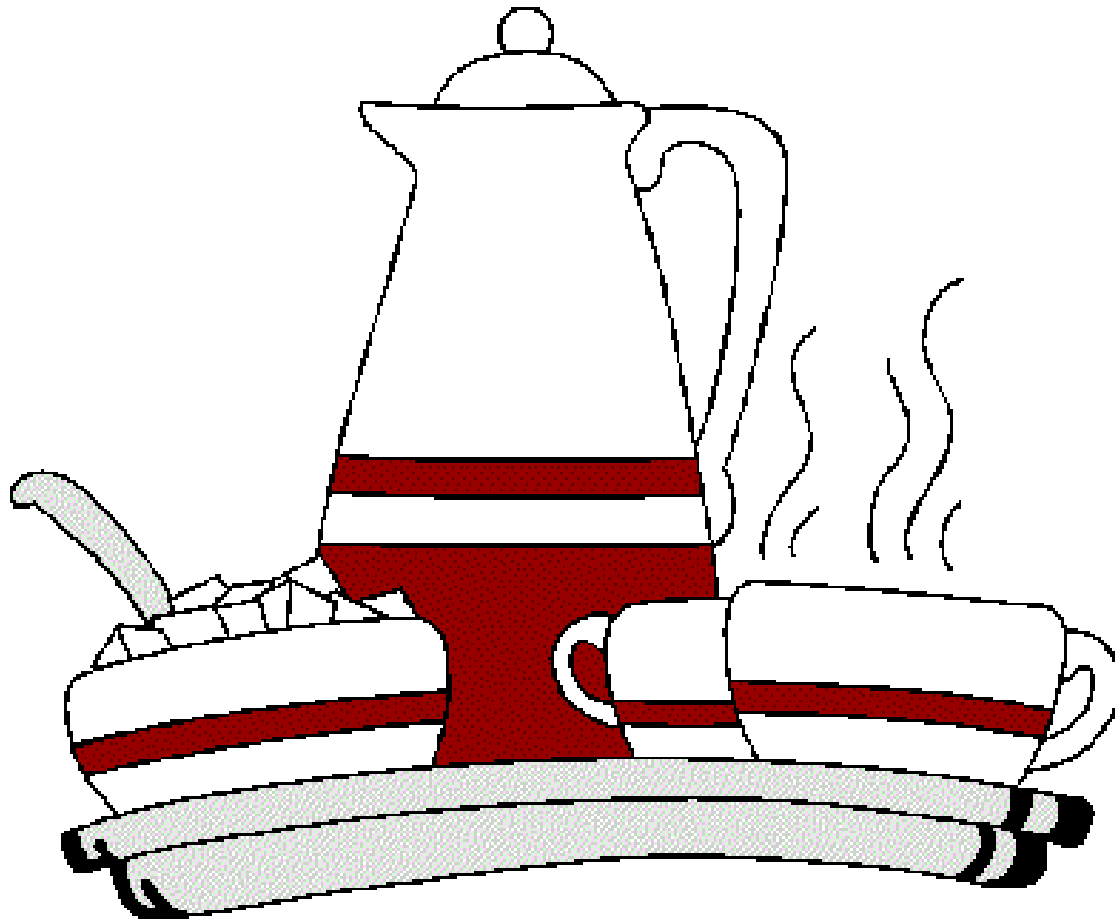


# Dataflow diagrammer : Pegasus for Virtuoso



**See you again after the break!**

---



(Second part by Peter Marwedel left out)