RTAS 2015
Seattle
2015-04-15
A. Zuepke

AUTOBEST:

A United AUTOSAR-OS And ARINC 653 Kernel

Alexander Züpke, Marc Bommert, Daniel Lohmann

alexander.zuepke@hs-rm.de, marc.bommert@hs-rm.de, lohmann@cs.fau.de

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

easycore

Supported by:
Federal Ministry
for Economic Affairs
and Energy

on the basis of a decision
by the German Bundestag

Motivation

- Automotive and Avionic industry begin to face similar challenges:
  - Hyper integration: increasing HW & SW complexity
  - Energy consumption
  - Certification effort
  - Cost pressure
  - Security issues

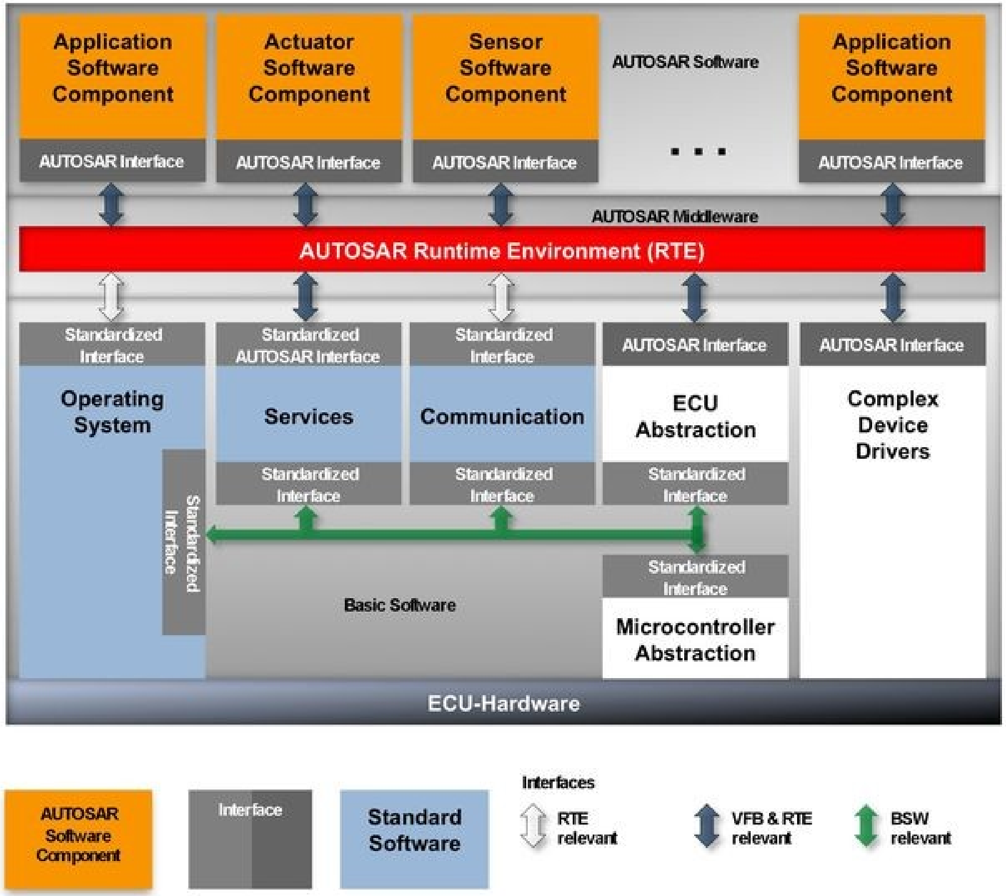
Motivation

- Automotive and Avionic industry begin to face similar challenges:
  - Hyper integration: increasing HW & SW complexity
  - Energy consumption
  - Certification effort
  - Cost pressure
  - Security issues
- But both industries use different OS standards!
  - Can't we challenge this with a single, unified operating system?
  - Combine avionics safety with the resource-efficiency of automotive systems?
  - And (probably) make it faster than existing systems?

Outline

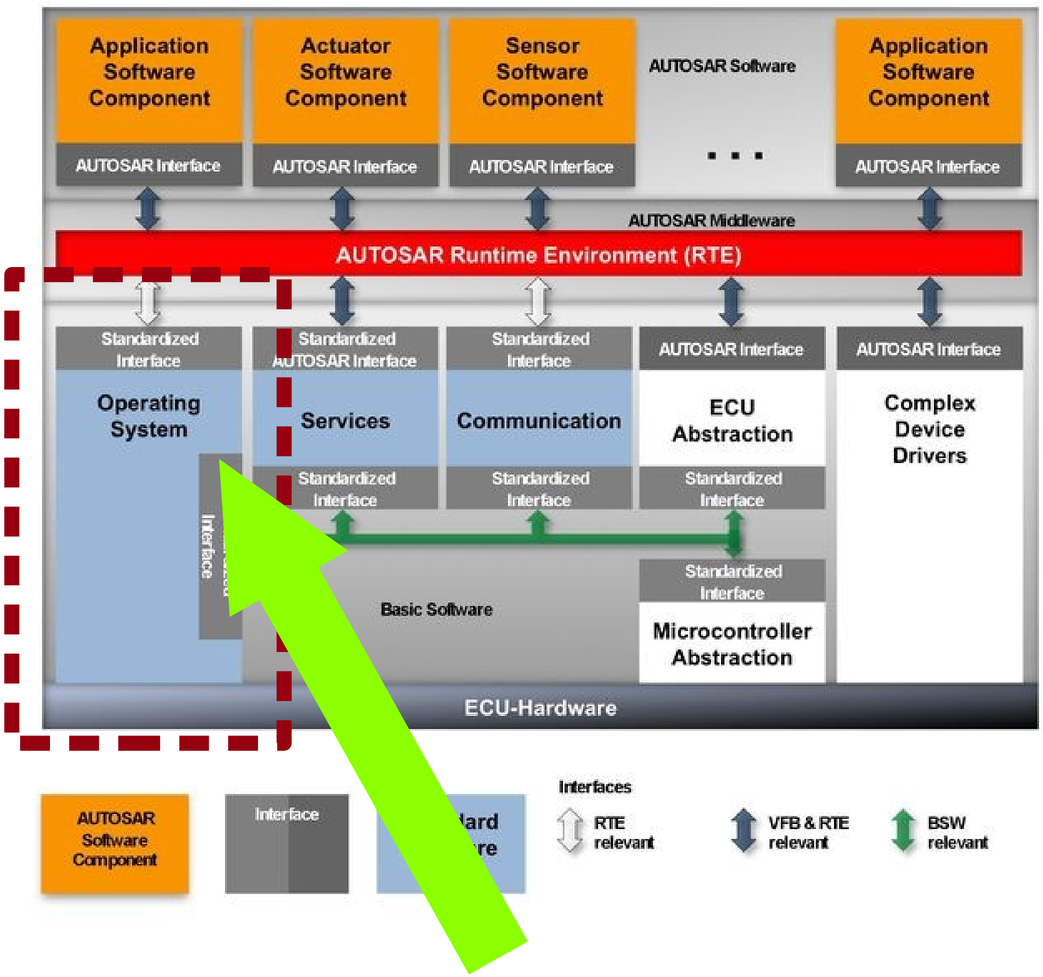
- AUTOSAR and ARINC 653
  - Short introduction
  - Task models
  - Partitioning concepts
  - Challenges
- AUTOBEST
  - Architecture
  - Lazy priority switching
  - Static Futexes
- Conclusions & Outlook

AUTOSAR



Source: http://www.autosar.org

AUTOSAR



OS Concepts:

- Tasks
- ISRs
- Event driven
- Fixed-priority scheduling
- Statically configured at compile time
- Isolation: group tasks into OS-Applications

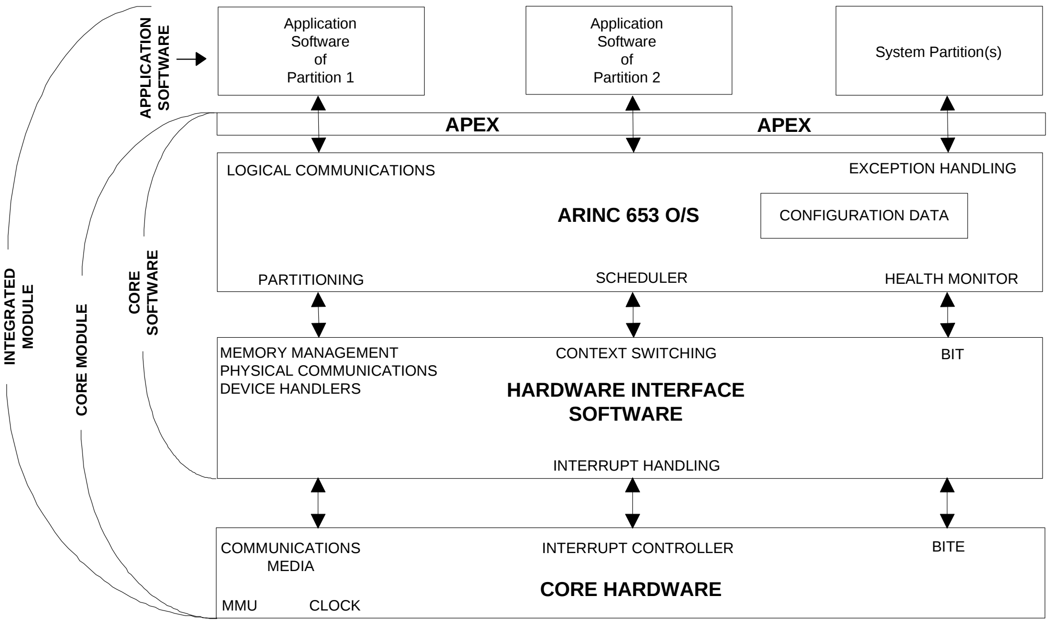
Here: focus on AUTOSAR-OS, the OS kernel

Source: http://www.autosar.org

ARINC 653

ARINC 653 Standard:

- Part 1 - Required Services
- Part 2 - Extended Services
- Part 3 - Conformity
- Part 4 - Subset Services

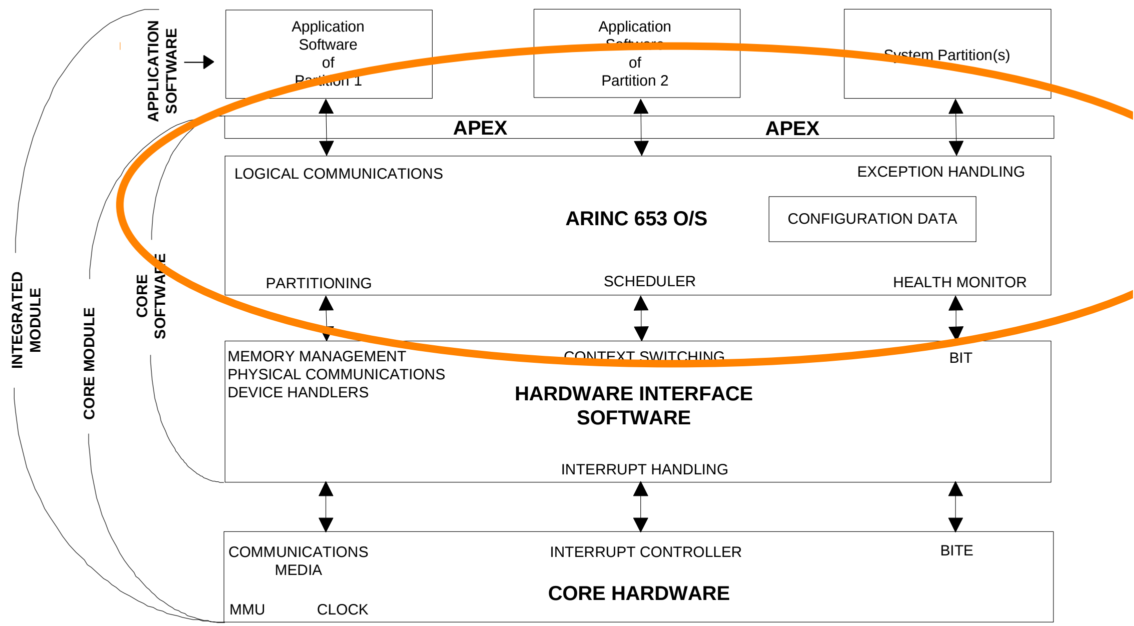


Source: ARINC 653 specification

ARINC 653

ARINC 653 Standard:

- Part 1 - Required Services
- Part 2 - Extended Services
- Part 3 - Conformity
- Part 4 - Subset Services

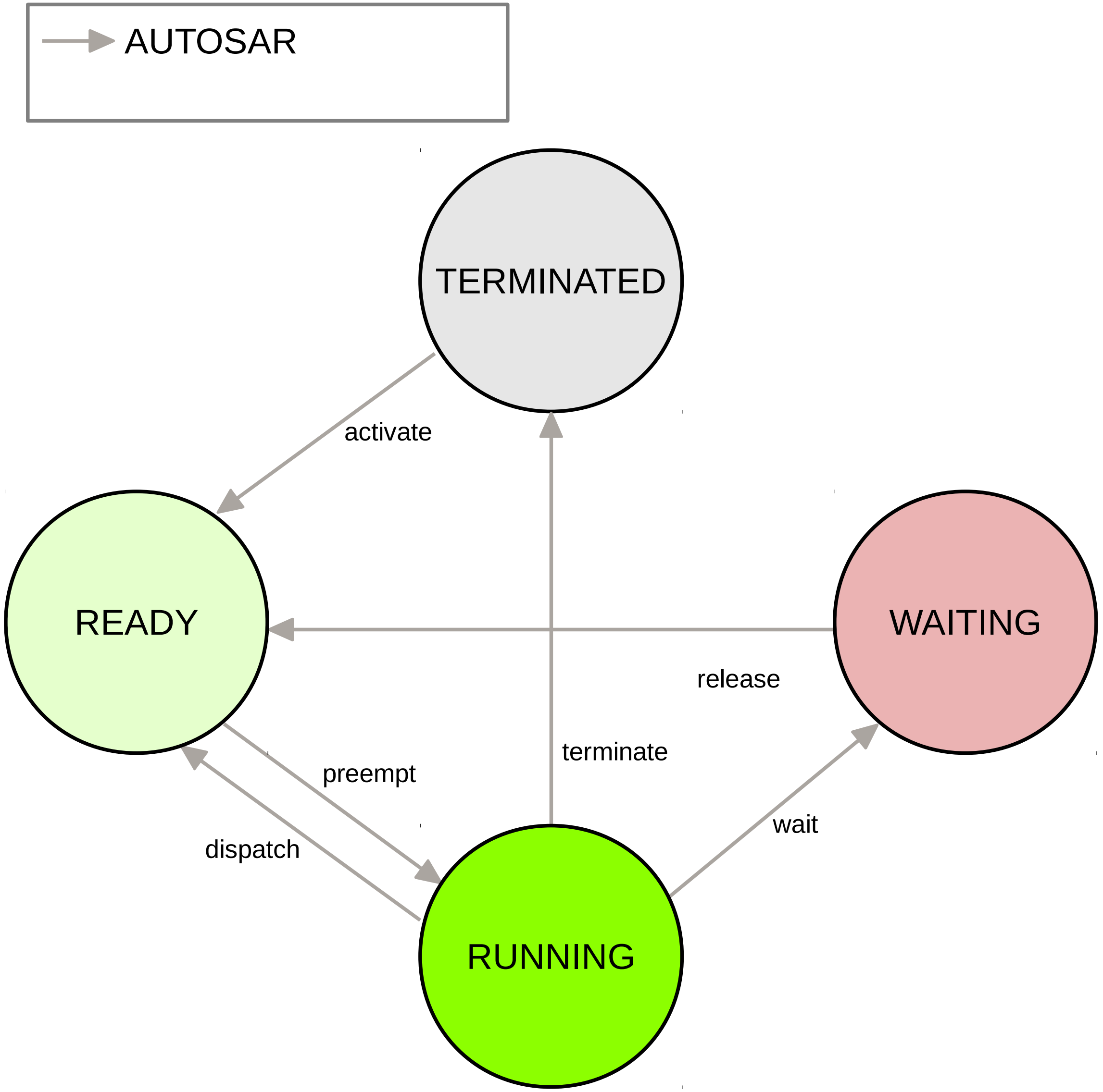


OS Concepts:

- Robust partitioning in space and time
- *Processes* (=Tasks) as executing entities
- Time driven and event driven
- Fixed-priority task scheduling, TDMA partition scheduling
- Task synchronization and partition communication means

Source: ARINC 653 specification

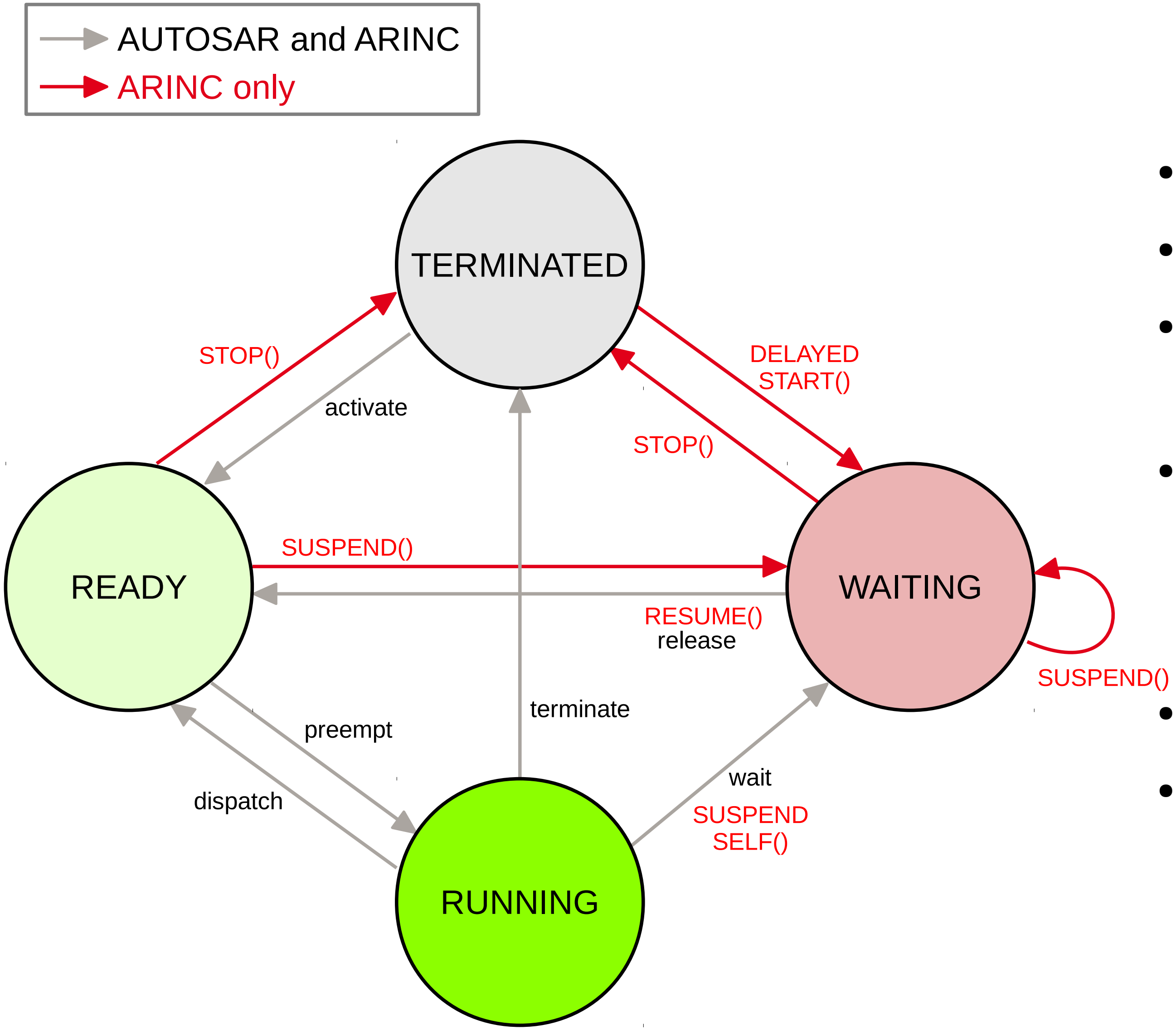
Task Model



AUTOSAR

- 4 task states
- Preemptive scheduling
- Waiting:
  - per-task event bitmask
- A task terminates when its job is complete

Task Model



AUTOSAR

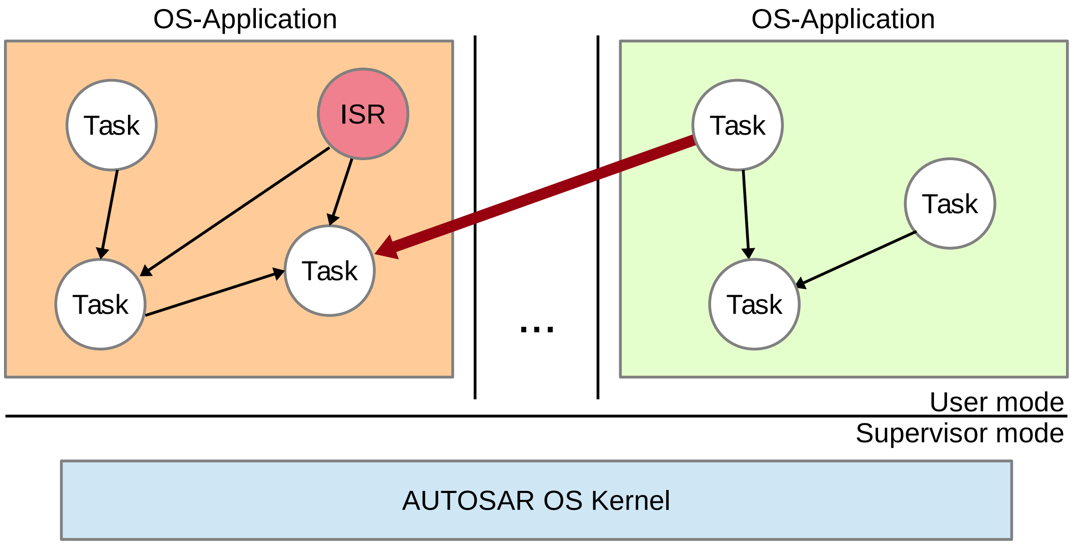
- 4 task states
- Preemptive scheduling
- Waiting:
  - per-task event bitmask
- A task terminates when its job is complete

ARINC 653

- Additional transitions
- Waiting:
  - Single-bit events
  - Semaphores
  - Buffers and blackboards
  - Queuing and sampling ports

Separation & Isolation

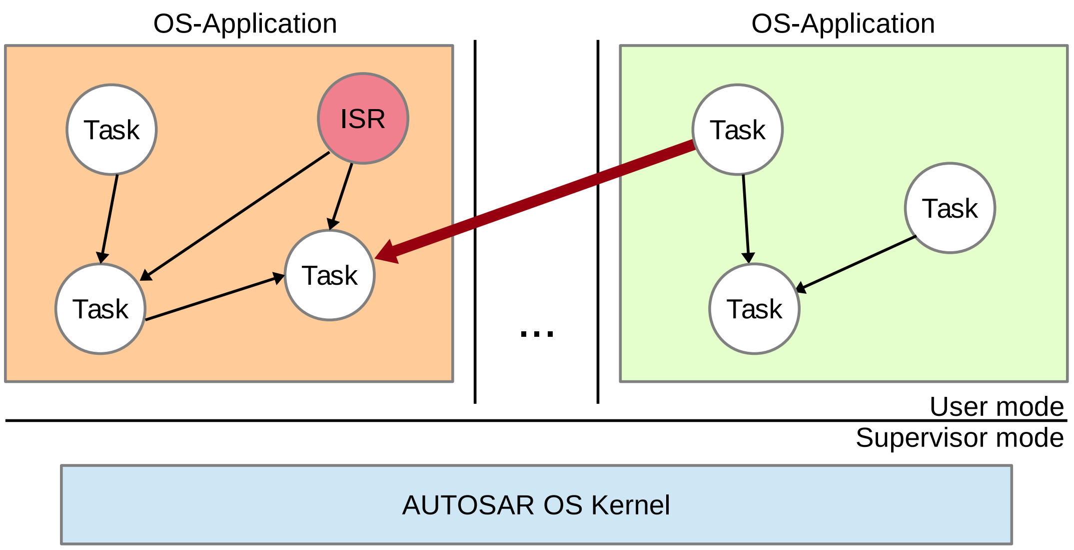
- AUTOSAR
  - *OS-Applications*
  - Optional concept
  - Memory protection
  - Configurable access to objects in other Applications



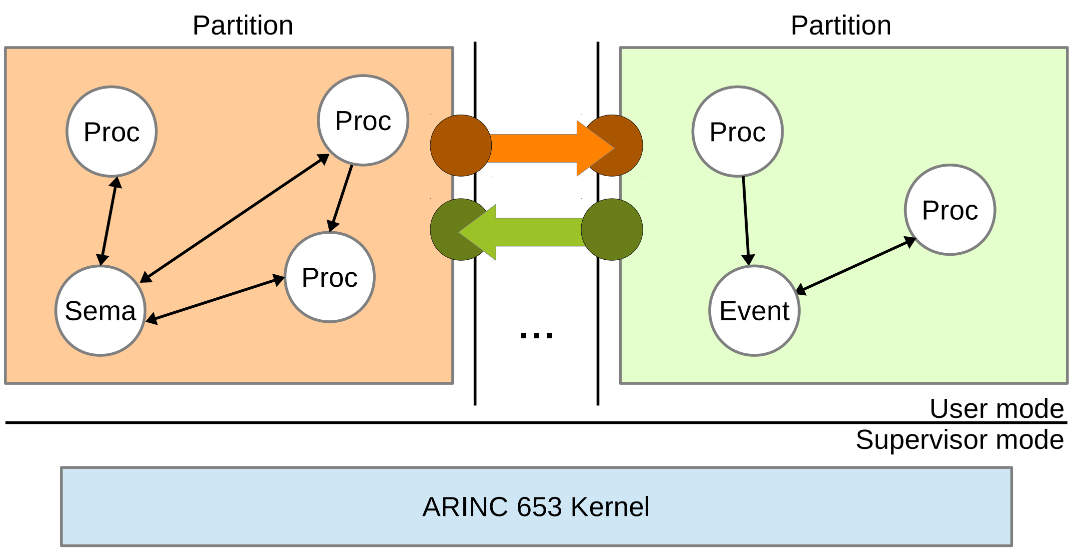
RTAS 2015
Seattle
2015-04-15
A. Zuepke

Separation & Isolation

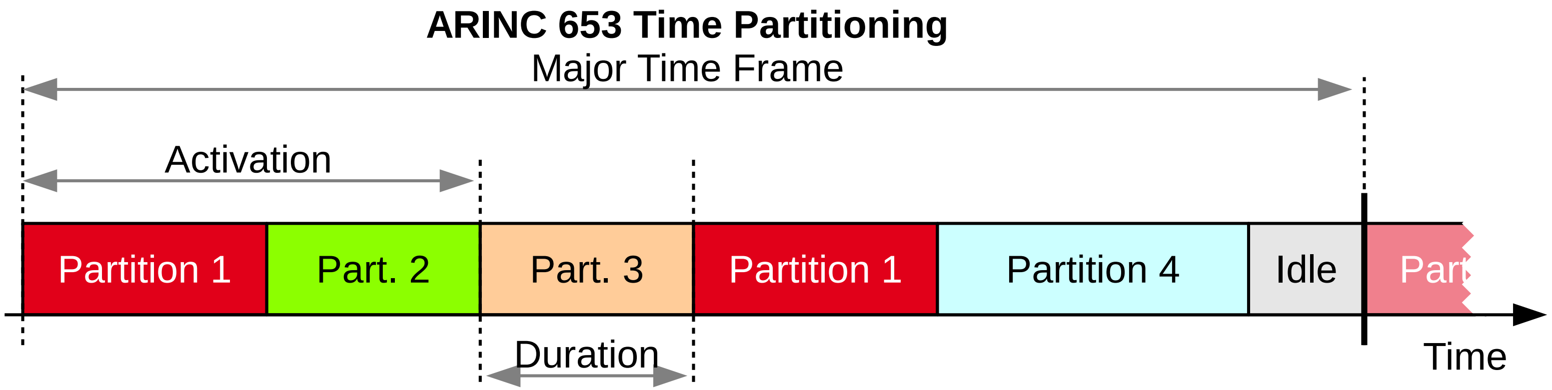
- AUTOSAR
  - *OS-Applications*
  - Optional concept
  - Memory protection
  - Configurable access to objects in other Applications



- ARINC 653
  - *Partitions*
  - Mandatory concept
  - Complete isolation
  - Explicit inter-partition communication means

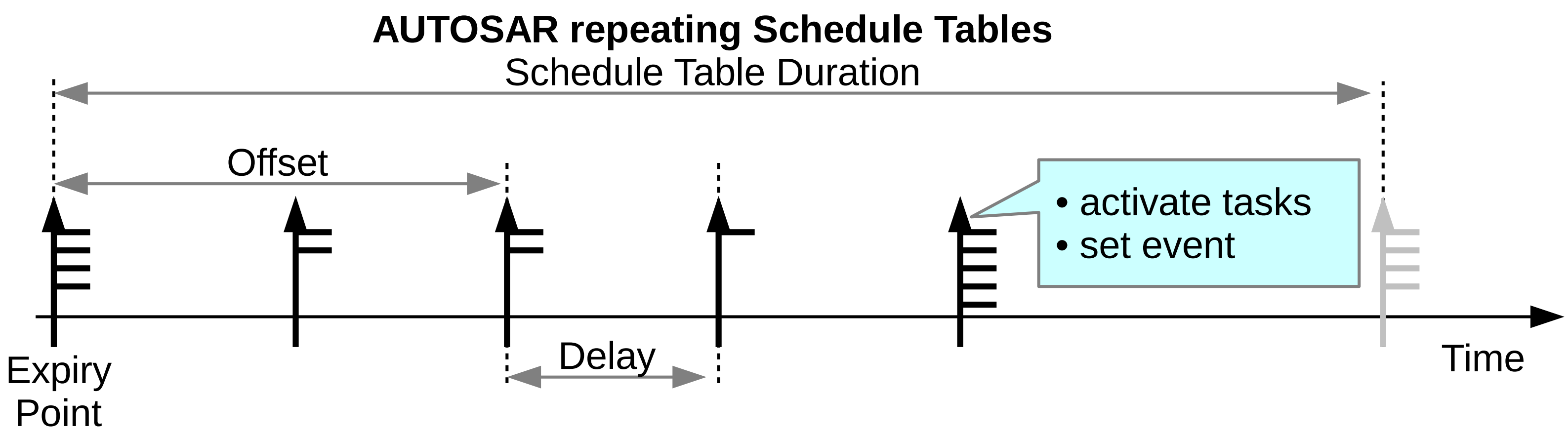


Time Partitioning



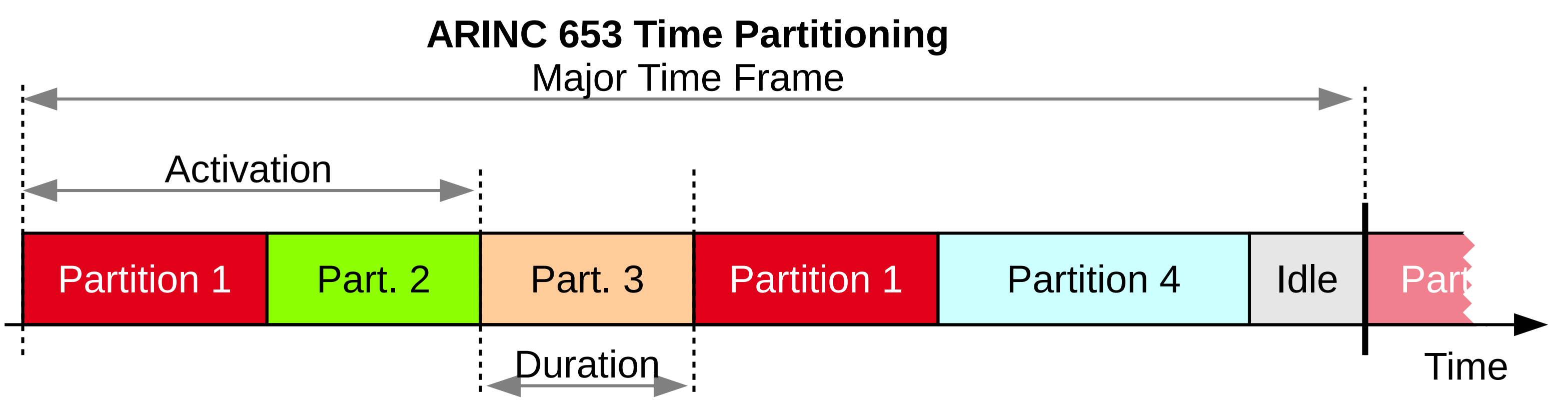
Time partitioning separates partitions and drives time-triggered tasks

Time-Triggered



AUTOSAR Schedule Tables allow similar time-triggered task activation

For temporal separation, optional timing protection facilities are available



Time partitioning separates partitions and drives time-triggered tasks

Differences

AUTOSAR

- Construction kit
- Task classes
- Scalability classes
- Isolation is an add-on
- Goals:
  - reduce resource usage
  - *keep it simple*

Differences

AUTOSAR

- Construction kit
- Task classes
- Scalability classes
- Isolation is an add-on
- Goals:
  - reduce resource usage
  - *keep it simple*

ARINC 653

- General purpose API
- No configuration options
- Certification
- Decoupling of partitions
- Goals:
  - fault mitigation
  - *safety first*

Challenges

- Support both AUTOSAR and ARINC 653 APIs
- Full ARINC 653 partitioning at minimal resource costs
- Performance comparable to other AUTOSAR implementations
- Keep system easy to (re-)certify

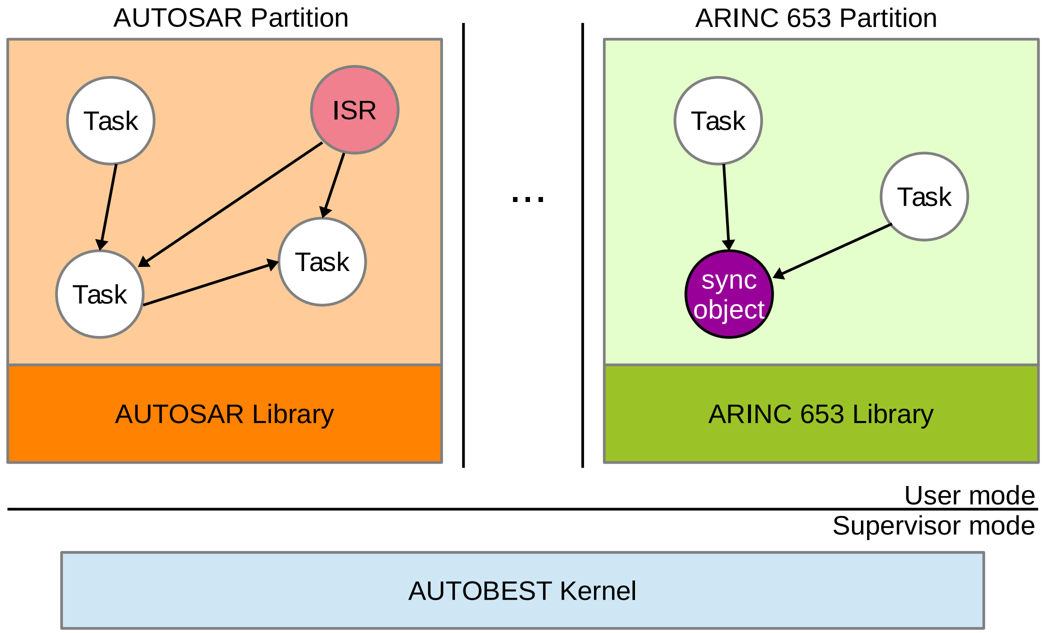
AUTOBEST

AUTOBEST Architecture

Statically configured microkernel

Partitions

- Space partitioning
- Time partitioning
- Driven by ARINC 653



Tasks

- Superset of AUTOSAR & ARINC 653 task API
- Keep OS specific differences out of the kernel
- User space libraries provide full AUTOSAR or ARINC 653 API

AUTOBEST Architecture

Scheduling

- Two-level scheduler
- Time-Partition Scheduling
  - One ready queue per time partition
  - Multiple space partitions can share one time partition
- Task Scheduling
  - Priority Scheduling with FIFO order on tie
- Fast critical sections using priority ceiling protocols
- Futex wait queues

AUTOBEST Architecture

Special Requirements of AUTOSAR

- Counters, Alarms, and Schedule Tables

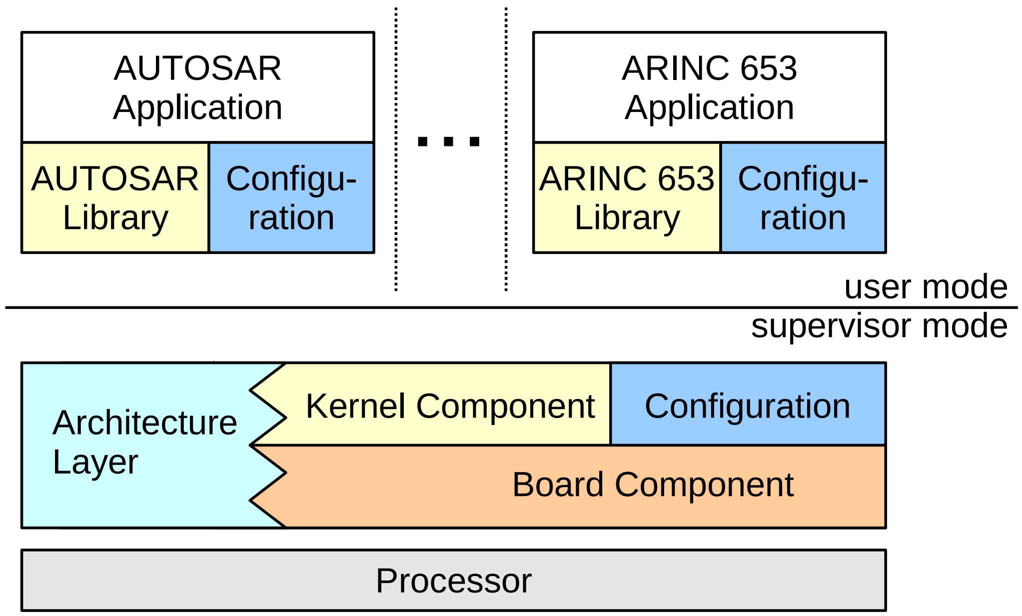
  → difficult to implement in user space
- Interrupt Handling
  - Allow ISRs in both kernel and user space
    - ISR cat 1 → kernel domain (no interaction with OS)
    - ISR cat 2a → kernel domain
    - ISR cat 2b → partition domain
  - Partitioned ISRs are mapped to high priority tasks
  - DisableInterrupts() → raise priority to partition maximum
- "*hooks*" (high priority pseudo tasks) for error handling

AUTOBEST Architecture

Special Requirements of ARINC 653

- Futex wait queues for ARINC sync objects
- 64-bit Nanosecond Timeout API
- Health Monitoring
  - Strict error handling using HM tables
  - Error process is mapped to a hook (like for AUTOSAR)
- Partitioning API
  - Start & Shutdown of other partitions
  - Privileged system calls
- Task deadline monitoring

AUTOBEST Architecture



Component Architecture

- Generic code
- Architecture code
- Board code
  - Boot, Interrupt Handling, ...
  - Kernel device drivers
- Configuration data
- OS specific libraries
- Kernel and partition code kept in dedicated binary images

System Configuration

- XML config → C language data structs (no C code, no #ifdefs)
- Binary reuse possible + reduces testing efforts

AUTOBEST Architecture

Resource Efficiency: RAM is precious!

- Split data model:
  - keep config in flash
  - keep data in RAM
- Most expensive:
  - task data (64 bytes)
  - register contexts
  - kernel stacks
  - ready queue per time partition (2 KiB / 256 priority levels)

Task Data

Queues use doubly-linked lists: → 24 bytes
- either ready queue or timeout queue
- sync object's wait queue
- deadline monitoring queue

Time tracking in 64-bit nanoseconds: → 24 bytes
- timeout expiry time
- phase shift of task activation
- deadline timeout

House keeping: → 16 bytes
- task state, priorities,
- pending events and task activations
- etc

AUTOBEST Architecture

Resource Efficiency: And flash as well!

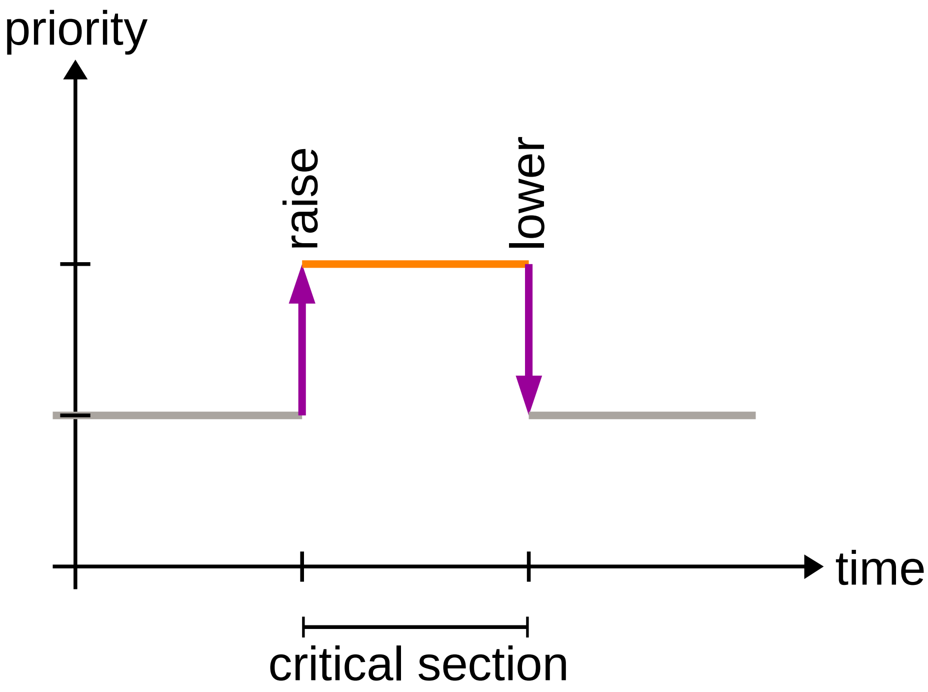
- Example Configuration:
  - 5 resource partitions, 1 time partition
  - 29 tasks, 10 hooks, 1 ISR
  - 4 schedule tables, 2 alarms
- Kernel memory usage on TMS570:
  - 13.5 KiB code (ARM thumb code, gcc -O2)
  - 7.5 KiB config
  - 17 KiB data (8 KiB stacks, 3.5 KiB registers, 2 KiB ready queues)

Lazy Priority Switching

Lazy Priority Switching

Typical critical section

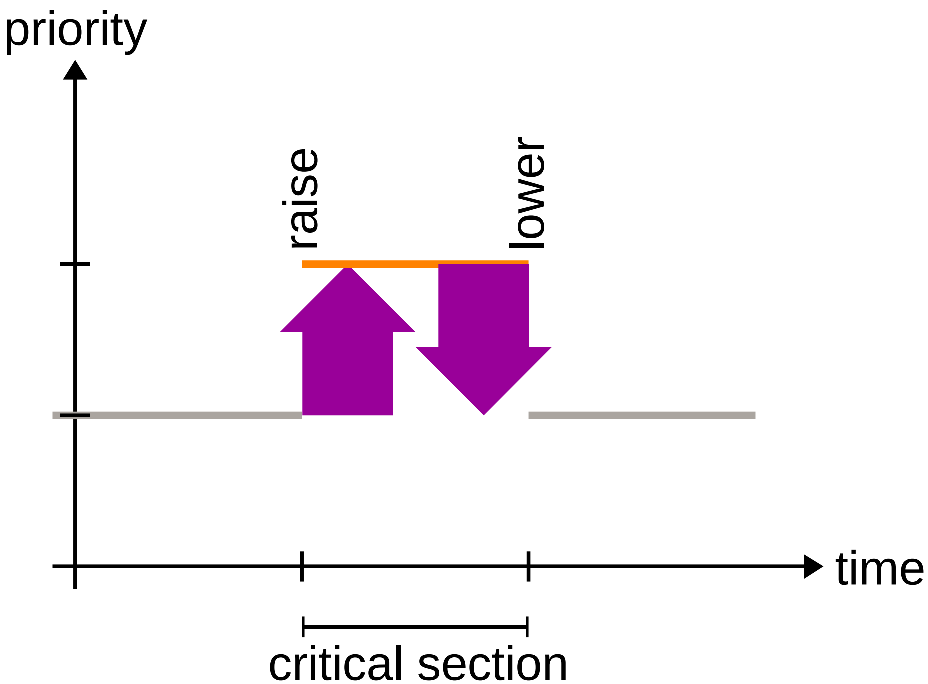
(using the AUTOSAR priority ceiling protocol)



Lazy Priority Switching

Typical critical section

(using the AUTOSAR priority ceiling protocol)



If critical sections are
short and frequent,
the
overhead
to change the caller's
scheduling priority
dominates
runtime costs!

Lazy Priority Switching

Change scheduling priority:

- System call overhead
- Raising priority: check & update some values
- Lowering priority: same + check for rescheduling

Lazy Priority Switching

Change scheduling priority:

- System call overhead
- Raising priority: check & update some values
- Lowering priority: same + check for rescheduling

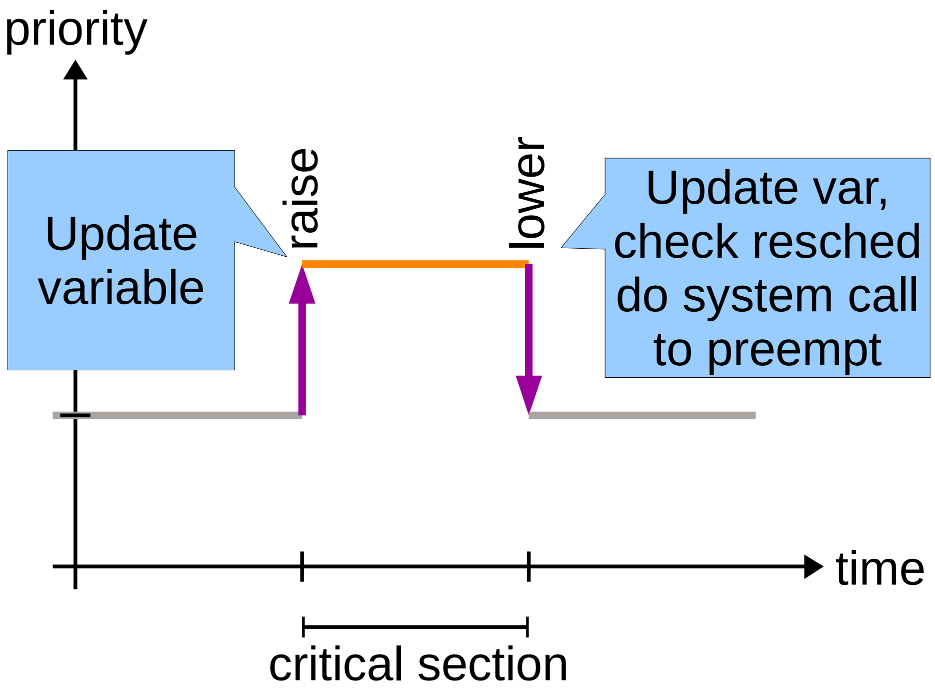
Idea for Optimization:

- Get rid of system call to set some values
- Kernel and user share variables to set priority
- Move rescheduling check into user space (checked when leaving a critical section)

Lazy Priority Switching

Optimized critical section:

One system call in the worst case (preemption)



Frequently used pattern, esp. in OS libraries!

Best case / no preemption:
MPC5646C: 688 → 31
TMS570: 843 → 94
(CPU cycles)

Futex Wait Queues

Futex Wait Queues

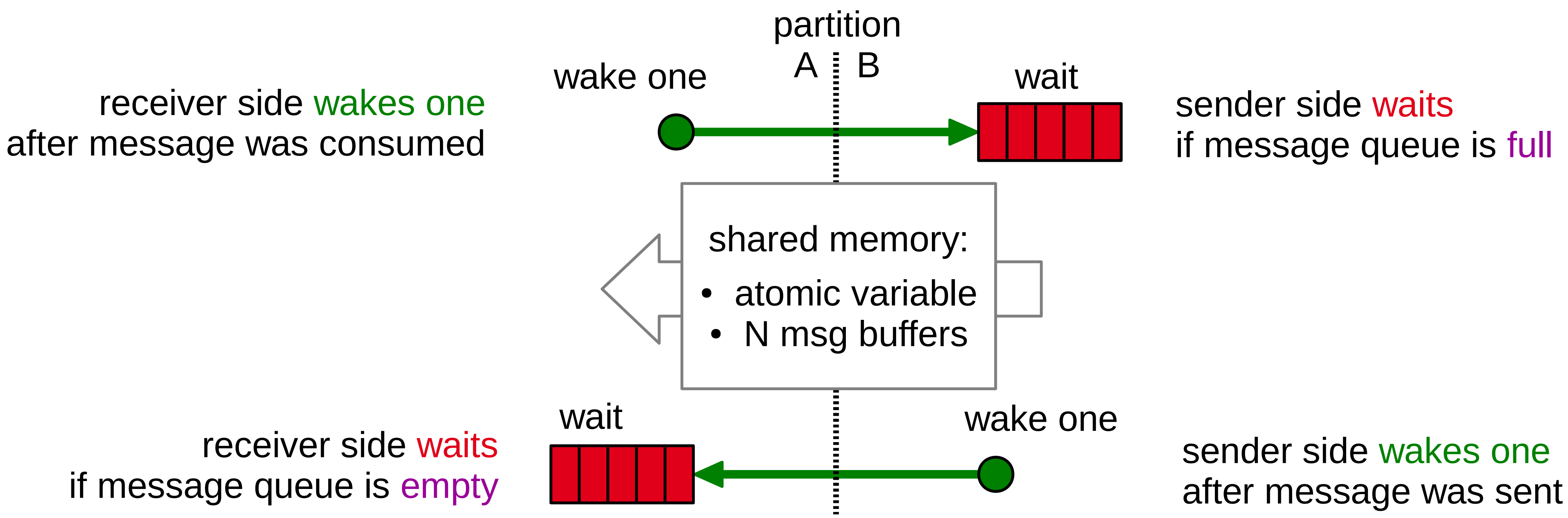
All ARINC 653 synchronization objects behave similar:

- Some tasks wait for something to happen
- Other tasks wake one or all waiters
- Each sync object has a wait queue
- Queue discipline: FIFO or priority-sorted (configured at partition start)
- Optional timeout
- Provide an API for wait queue operations

Futex Wait Queues

Example: Queuing port communication

- Shared memory for message buffers in queuing port channel
- One atomic Futex variable encodes read and write positions
- Two cross-connected wait queues



Futex Wait Queues

The kernel needs to manage wait queues only:

- Abstract wait and wake operations like in Linux Futexes
- Wait and wake sides can reside in different partitions

All ARINC 653 synchronization objects can be built upon Futexes and wait queues

Bonus: all copy operations done in user space (copy messages in user space at highest priority)

Conclusion

Conclusion

Conclusion

- Possible: unified kernel for AUTOSAR and ARINC
- Lazy priority switching improves performance
- Futex wait queues for ARINC 653 synchronization means keep complexity out of the kernel

Lessons learned

- Statically tailored kernels: good choice for safety-critical systems (much simpler than using runtime configuration)
- AUTOSAR nowadays provides similar functionality as ARINC 653

Outlook

Status

- Single core: PPC (e200) and ARM (Cortex R4)
- AUTOSAR OS 4.1 rev 3
- ARINC 653 part 1 suppl 3
- Pthread subset as additional OS environment

Future work

- Multi core: Infineon AURIX
- Real-world benchmarks
- Interrupts do not fit well to a time partitioned world
- Problem: Lock step not available for all cores

Thank you for your attention!

Backup Slides

POSIX

Subset of POSIX PSE5.1

- Pthread API
  - Pthreads
  - Mutexes
  - Condition Variables
- Minimal C-library
- Dynamic memory allocation
- Scheduling
  - SCHED_FIFO supported
  - SCHED_RR not supported
- No Signal Handling
- No Thread Cancellation

No additional efforts in the kernel!

Performance

Performance (Nov 2014)

- Freescale MPC5646C *Bolero* @120 MHz
  - Syscall 87 cycles / 0.73 µs
  - OSEK resource fast 29 cycles / 0.24 µs
  - OSEK resource slow 300 cycles / 2.50 µs
  - Task switch 390 cycles / 3.25 µs
- Texas Instruments TMS570LS3137 @180 MHz
  - Syscall 151 cycles / 0.83 µs
  - OSEK resource fast 70 cycles / 0.38 µs
  - OSEK resource slow 431 cycles / 2.39 µs
  - Partition switch overhead 659 cycles / 3.66 µs

AUTOBEST Architecture

Considered alternative: Hypervisor design

Microkernel	Hypervisor
Kernel knows all OS tasks	HV knows only partitions
Kernel schedules tasks and partitions	HV schedules partitions, delegates task scheduling to partition
API: OSEK + Partitioning API	API: Virtual CPU + virtual interrupt controller programming interface
ISRs: - Cat 2 ISRs → partitioned - Cat 1 ISRs → global	ISRs: similar
Cat 2 ISRs are scheduled like tasks	How to inject interrupts with priority?

The End