AUTOBEST:

A microkernel-based system (not only) for automotive applications

Marc Bommert, Alexander Züpke, Robert Kaiser

vorname.name@hs-rm.de

Supported by:
Federal Ministry for Economic Affairs and Energy
on the basis of a decision by the German Bundestag

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

easycore

About Us

Forschungsgruppe

„neue Betriebssystemkonzepte“

der Hochschule RheinMain

Marc Bommert, Alexander Züpke, Robert Kaiser

Marc & Alex: PhD students since October 2012

Thesis supervisors:

- Dieter Zöbel, Uni Koblenz-Landau
- Daniel Lohmann, Uni Erlangen-Nürnberg

Motivation

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

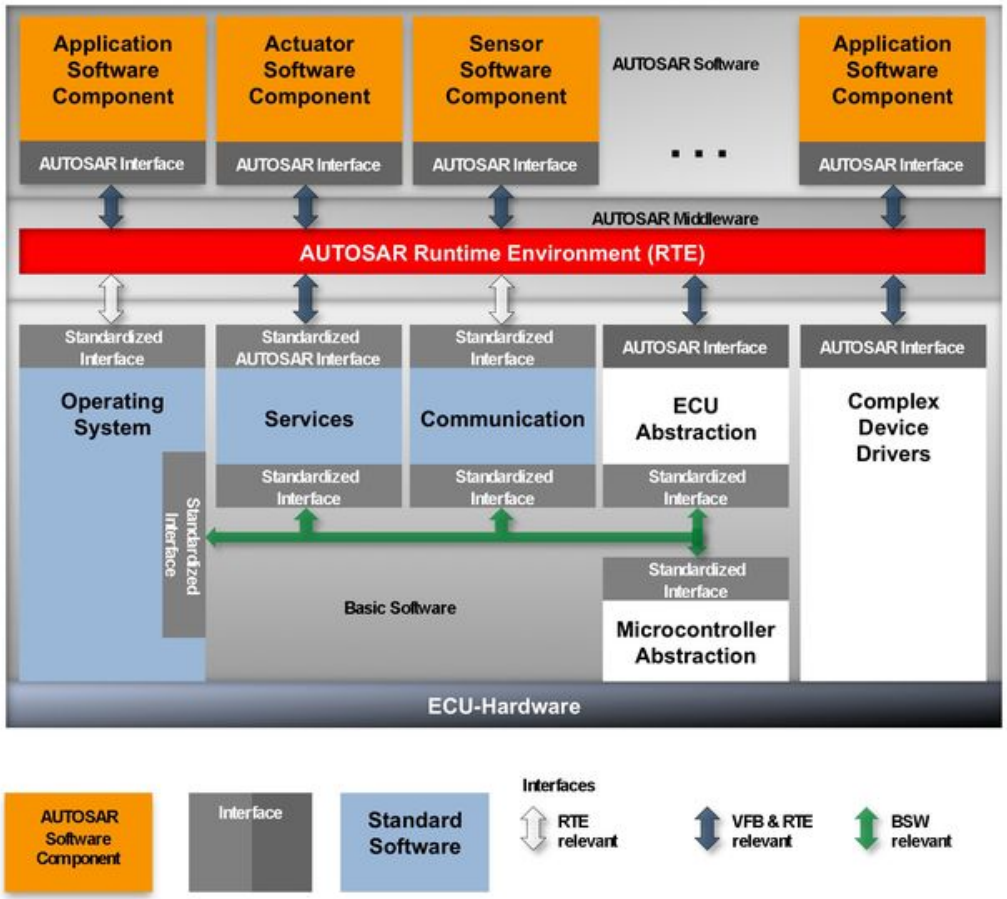
Motivation

- 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

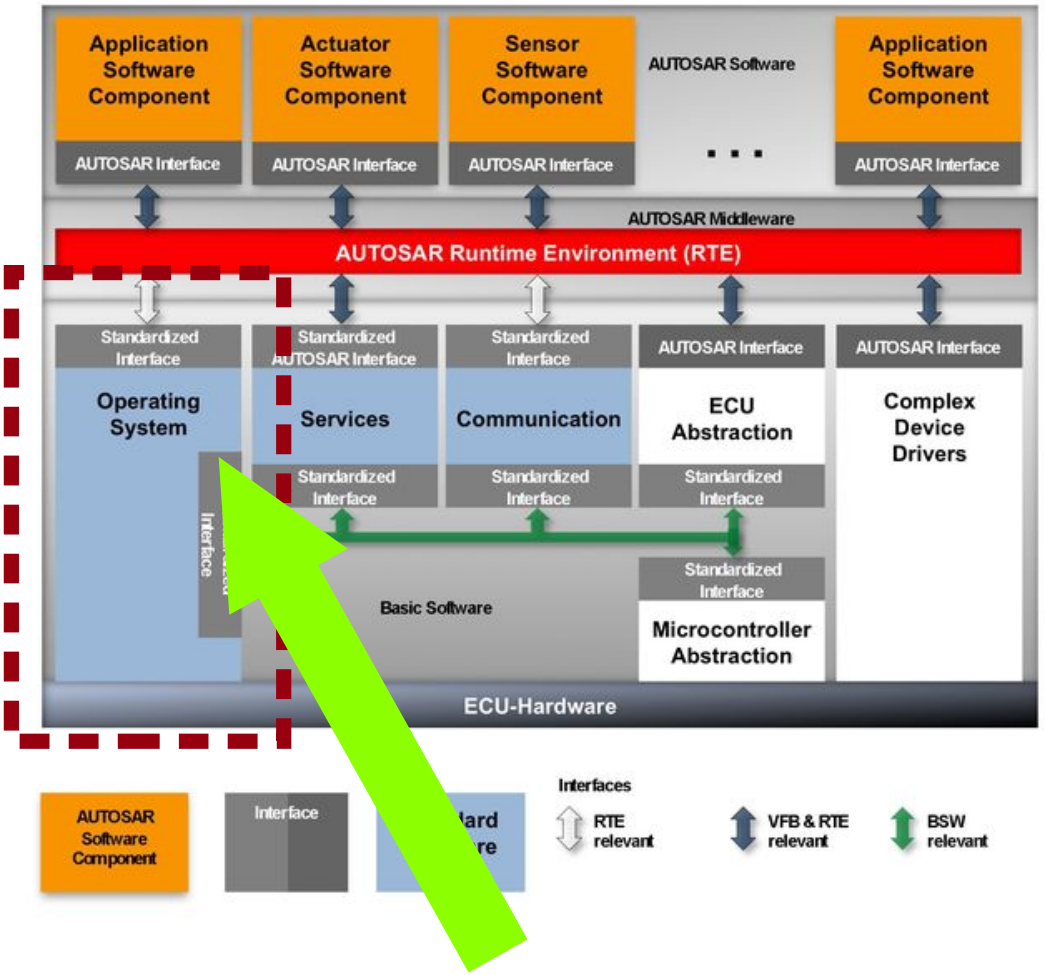
- AUTOSAR and ARINC 653
  - Short Introduction
  - Task Models
  - Partitioning Concepts
- AUTOBEST
  - System Architecture
  - Technical Concepts
  - Implementation
- Research Topics
- Conclusion & Outlook

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

AUTOSAR

Look & Feel

System Configuration

```
TASK myTask1 {
        PRIORITY = 42;
        STACK = SHARED ;
        SCHEDULE = FULL;
};

COUNTER myCounter1 {
        MINCYCLE = 2;
        MAXALLOWEDVALUE = 100;
        TICKSPERBASE = 1;
};

ALARM myAlarm1 {
        COUNTER = "myCounter1";
        ACTION = ACTIVATETASK {
                TASK = "myTask1";
        };
        AUTOSTART = TRUE;
};
```

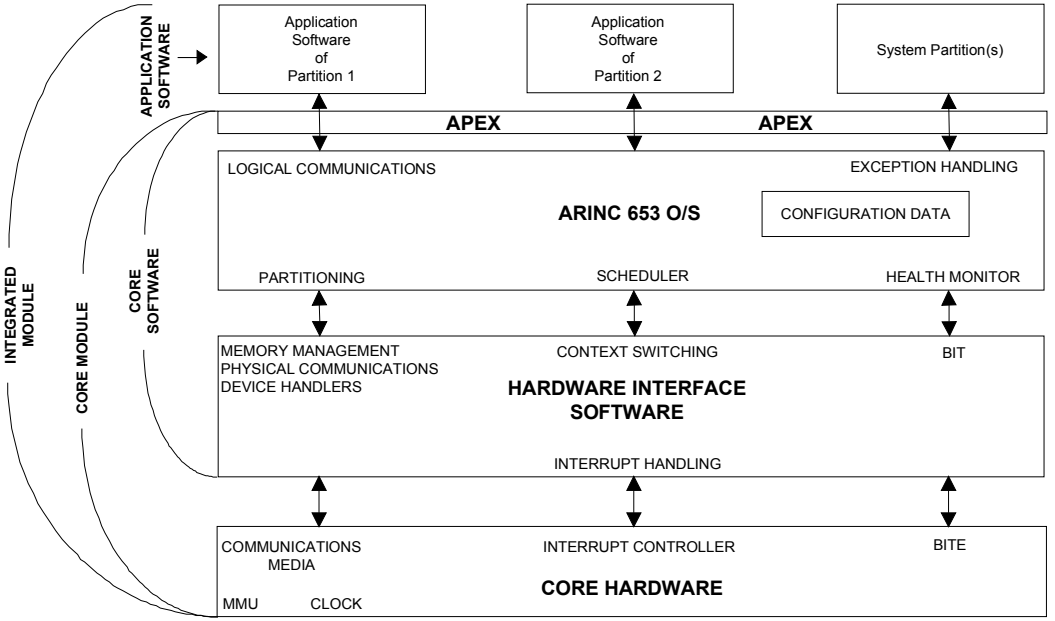
Application Code in C

```
int main(void)
{
        /* Start the OS. */
        StartOS(OSDEFAULTAPPMODE);
        /* Does not return. */
        return 0;
}

TASK(myTask1)
{
        /* toggle LED */
        static int led_state = 0;
        led_state = !led_state;
        set_led(15, led_state);

        TerminateTask();
}
```

ARINC 653

ARINC 653 Standard:

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

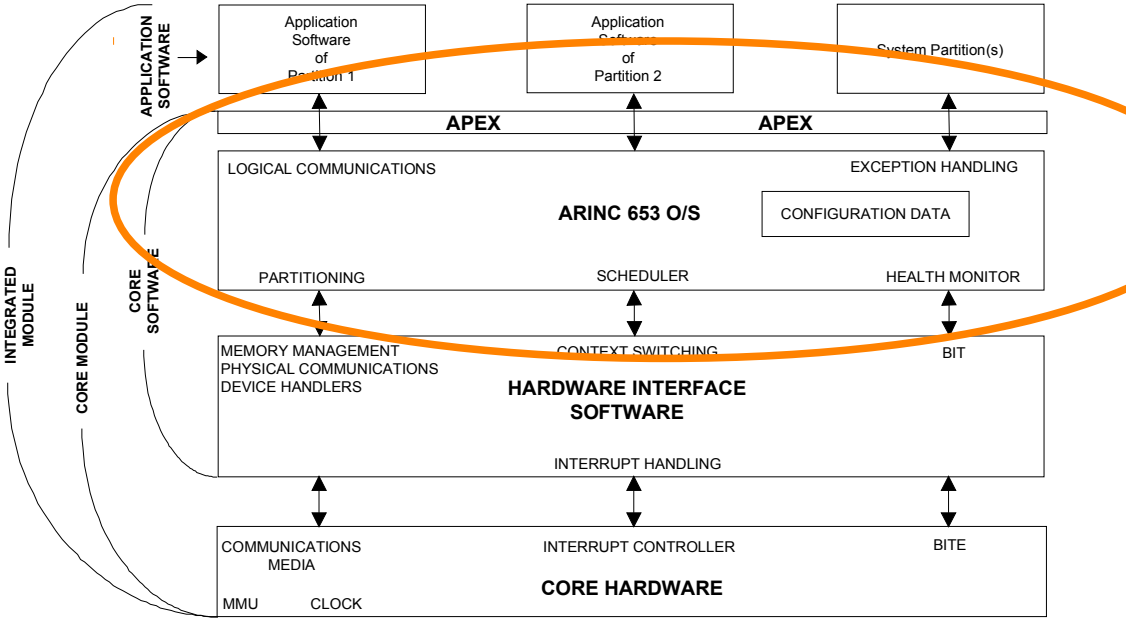


Source: ARINC 653 specification

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

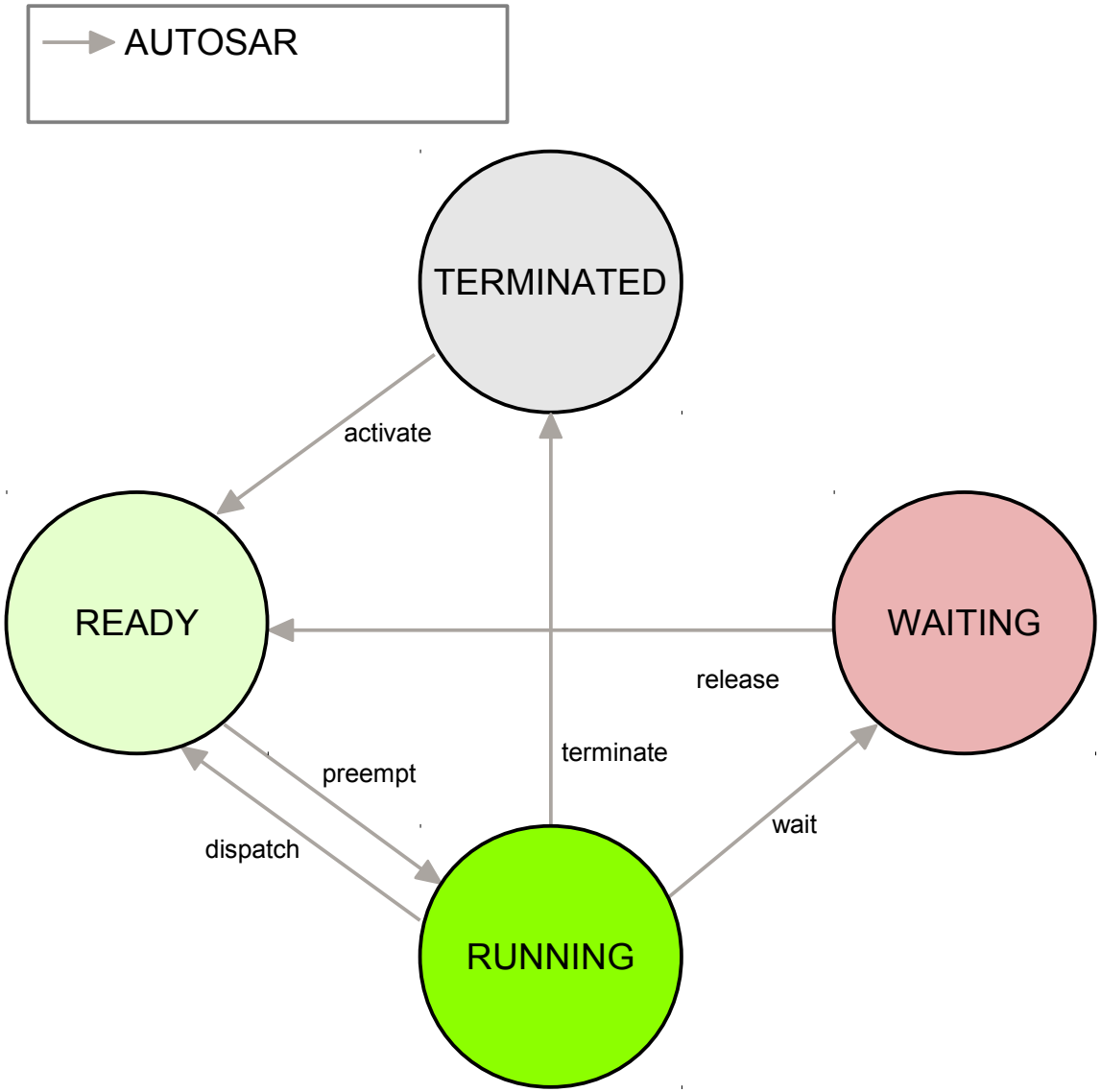
ARINC 653

Look & Feel

- XML configures partitioning:
  - How much memory?
  - Communication ports between partitions
- Application startup code:
  - Opens all communication channels
  - Creates all internal OS objects

Source: http://www.wikipedia.org

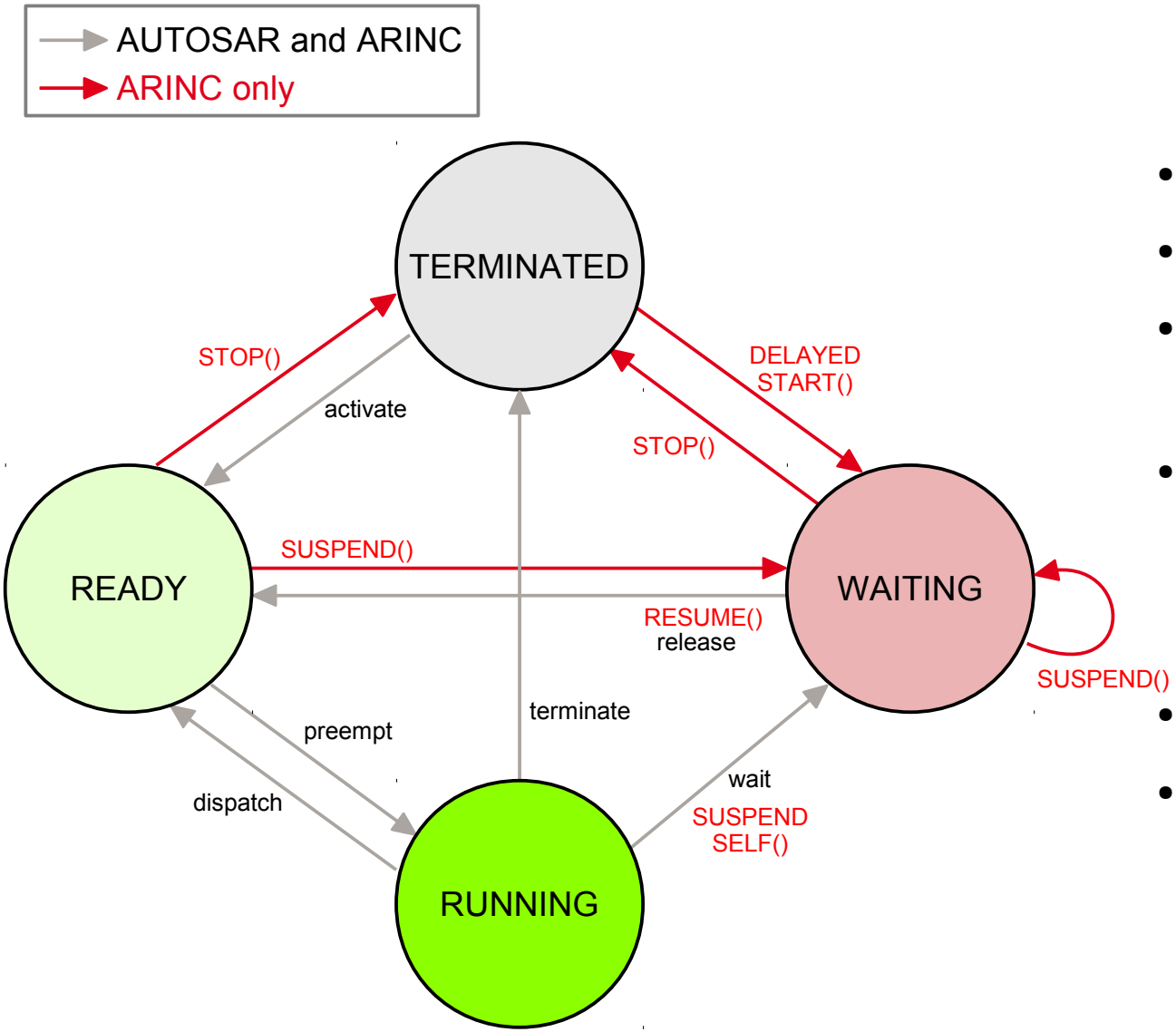
Task Model



AUTOSAR

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

Task Model



AUTOSAR

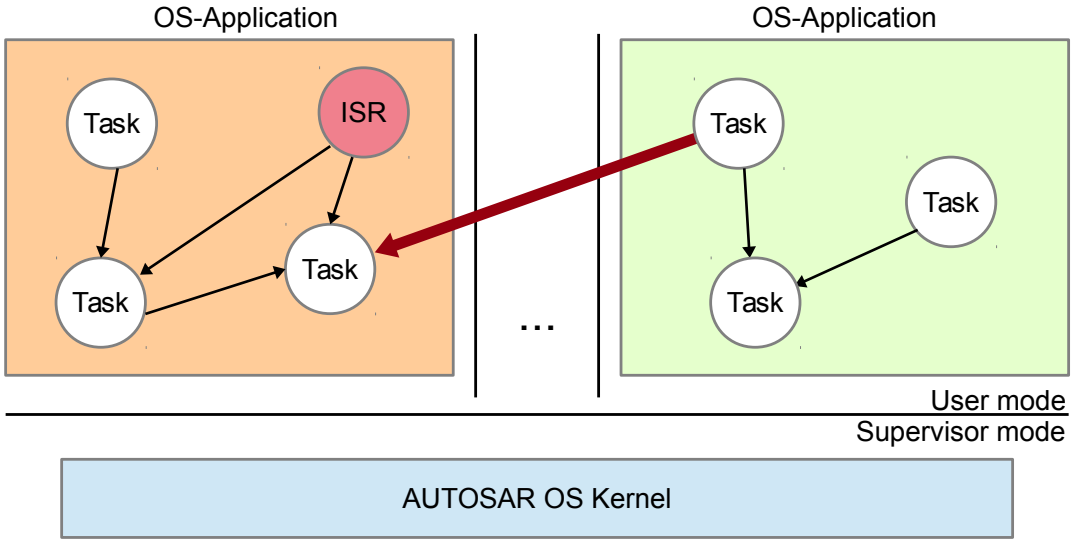
- 4 task states
- Preemptive scheduling
- Waiting:
  - per-task event bitmask
- A task terminates when its job is complete

ARINC 653

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

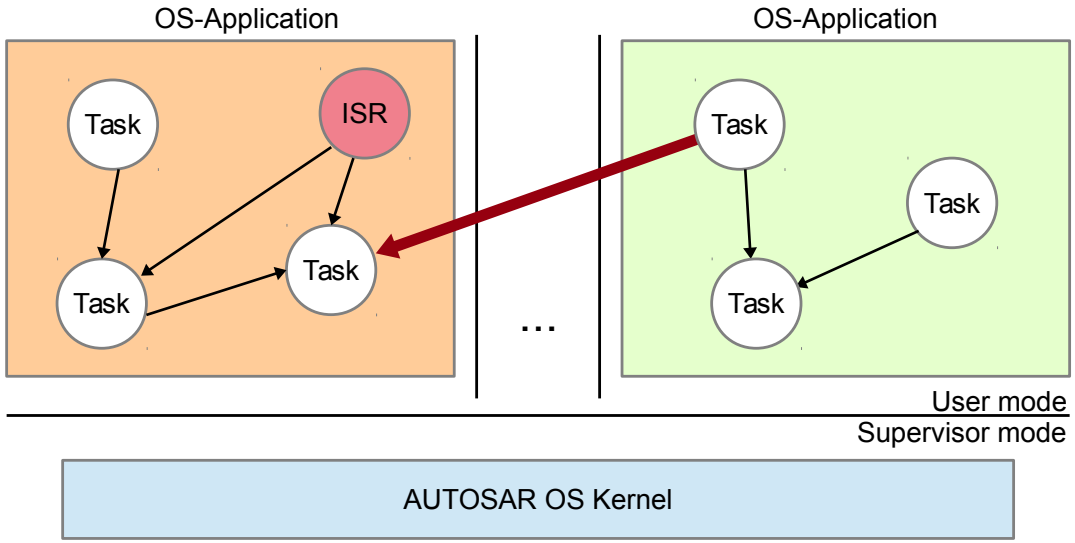
Separation & Isolation

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

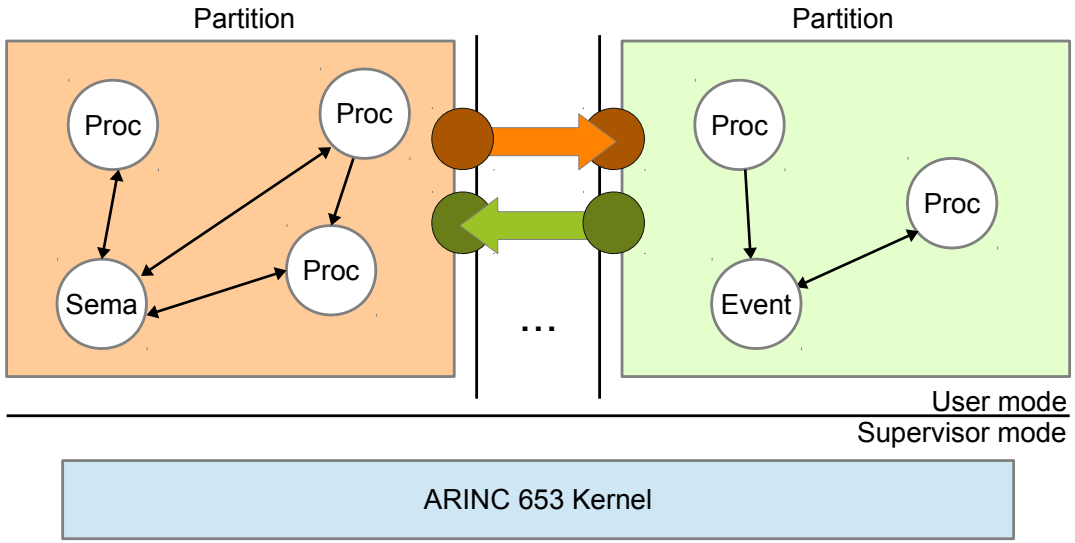


Separation & Isolation

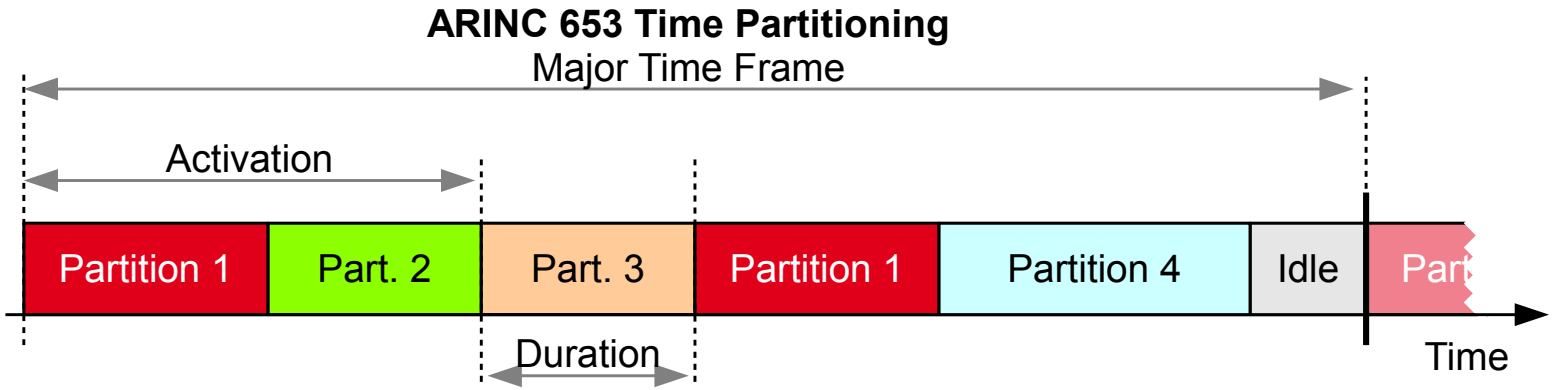
- AUTOSAR
  - *OS-Applications*
  - Optional concept
  - Memory protection
  - Configurable access to objects in other Applications



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

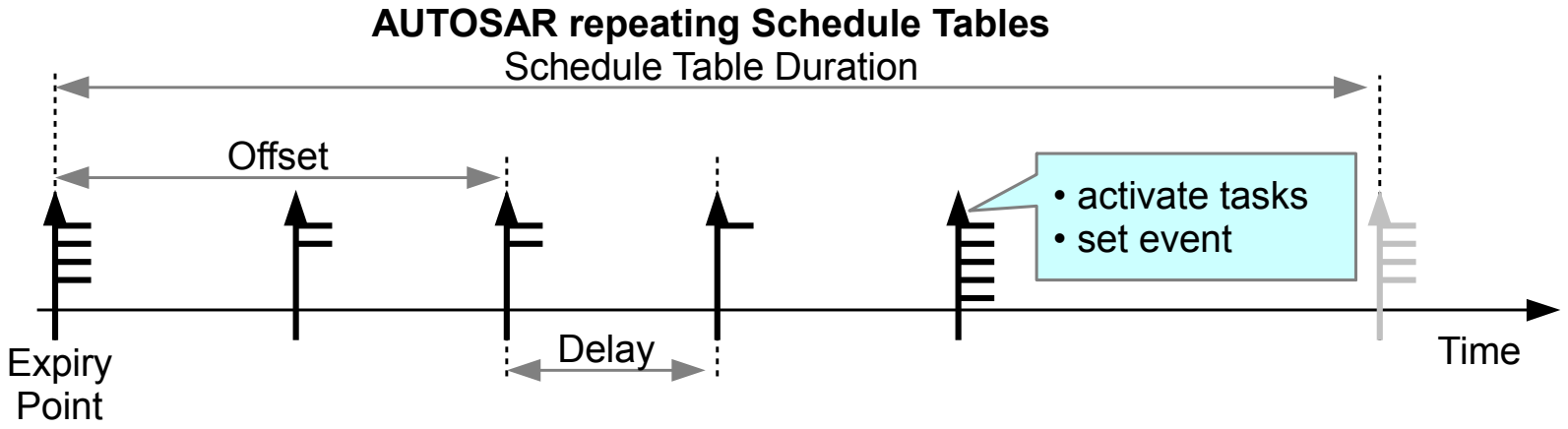


Time Partitioning



Time partitioning separates partitions and drives time-triggered tasks

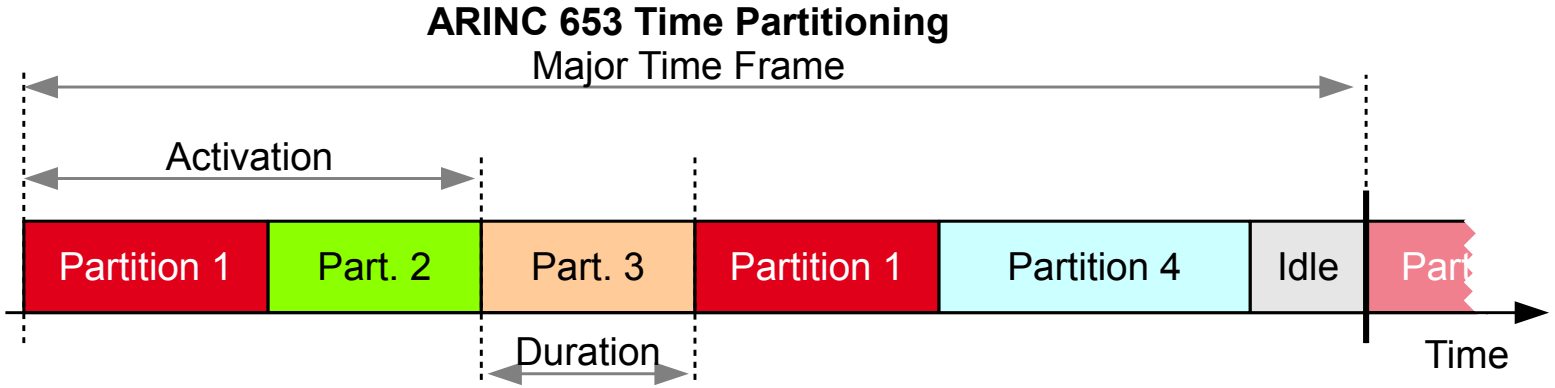
Time-Triggered



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

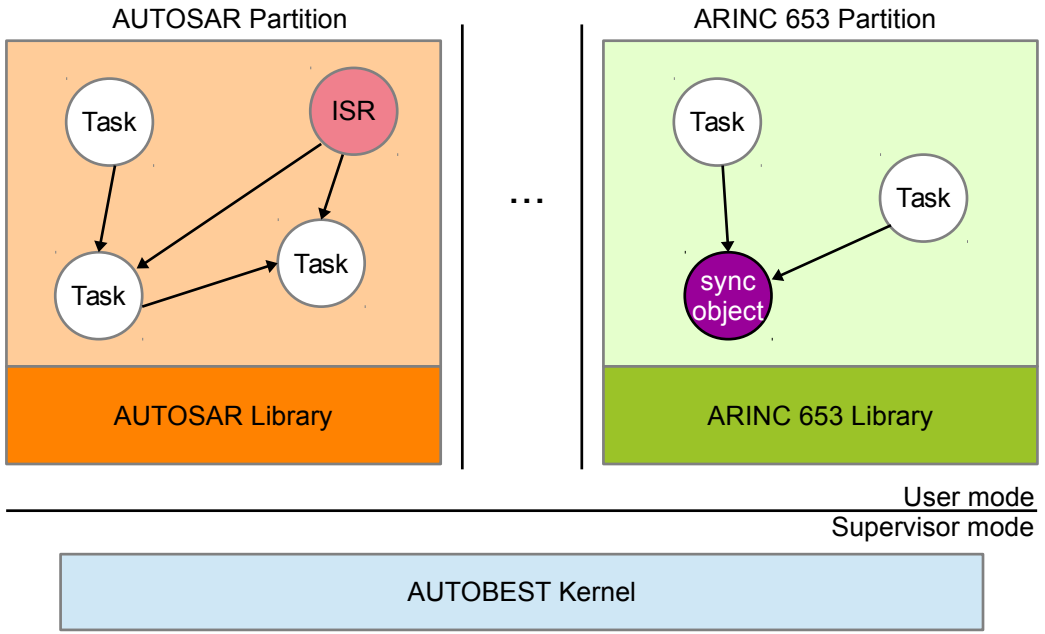
AUTOBEST

AUTOBEST Architecture

Statically configured microkernel

Strict Partitioning

- Space partitioning
- Time partitioning
- Driven by ARINC 653



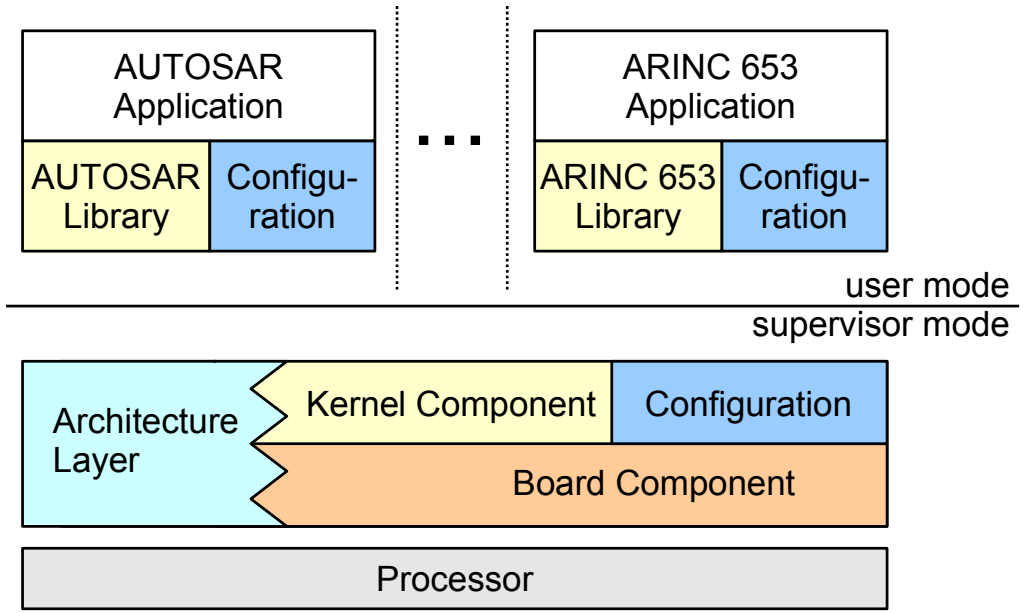
Tasks

- 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

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

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

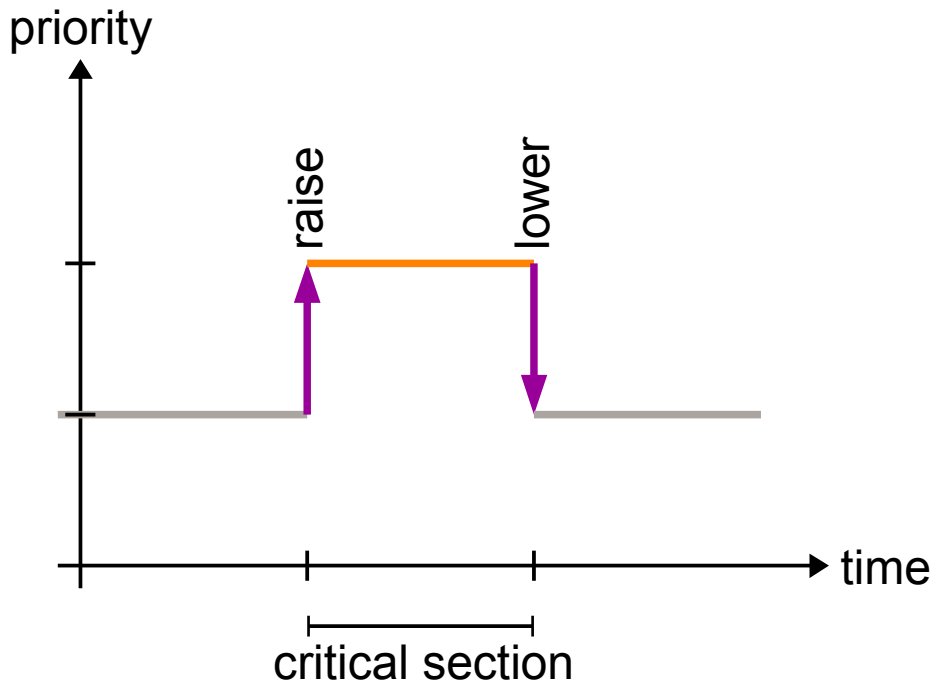
- Futex wait queues for ARINC sync objects
- 64-bit Nanosecond Timeout API
- Health Monitoring
  - Strict error handling using HM tables
  - Error process is mapped to a hook (like for AUTOSAR)
- Partitioning API
  - Start & Shutdown of other partitions
  - Privileged system calls
- Task deadline monitoring

Lazy Priority Switching

Lazy Priority Switching

Typical critical section

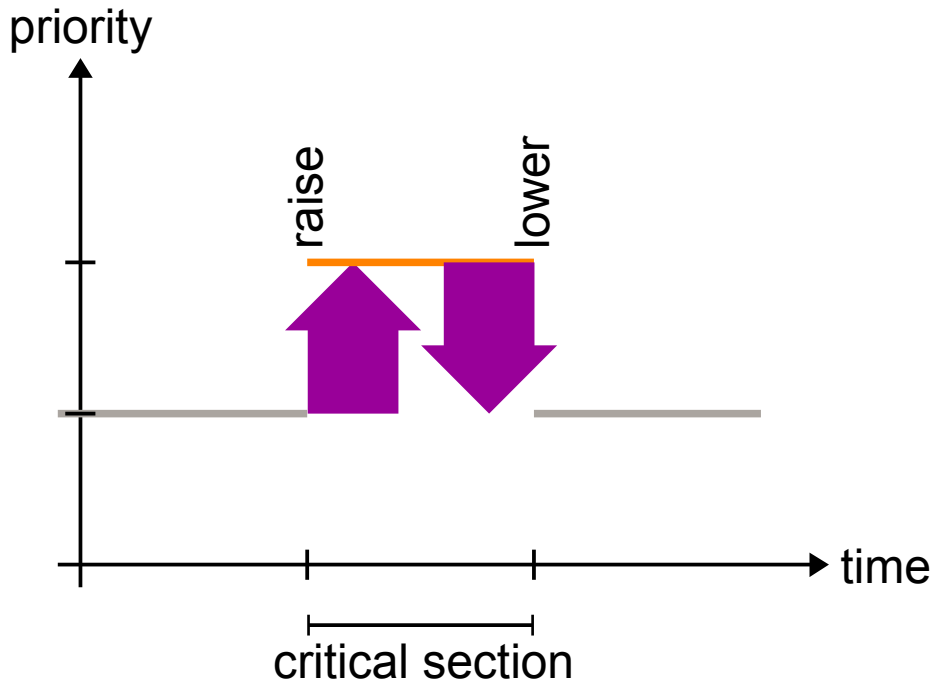
(using the AUTOSAR priority ceiling protocol)



Lazy Priority Switching

Typical critical section

(using the AUTOSAR priority ceiling protocol)

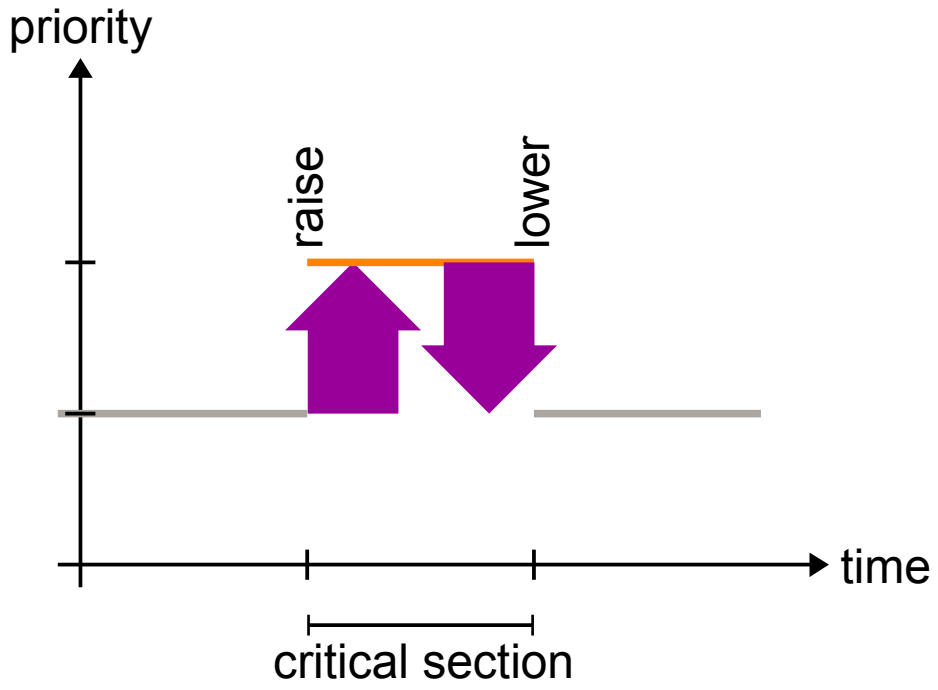


If critical sections are
short and frequent,
the
overhead
to change the caller's
scheduling priority
dominates
runtime costs!

Lazy Priority Switching

Typical critical section

(using the AUTOSAR priority ceiling protocol)



Optimize it:

1. increase priority by
by writing to a variable
in user space

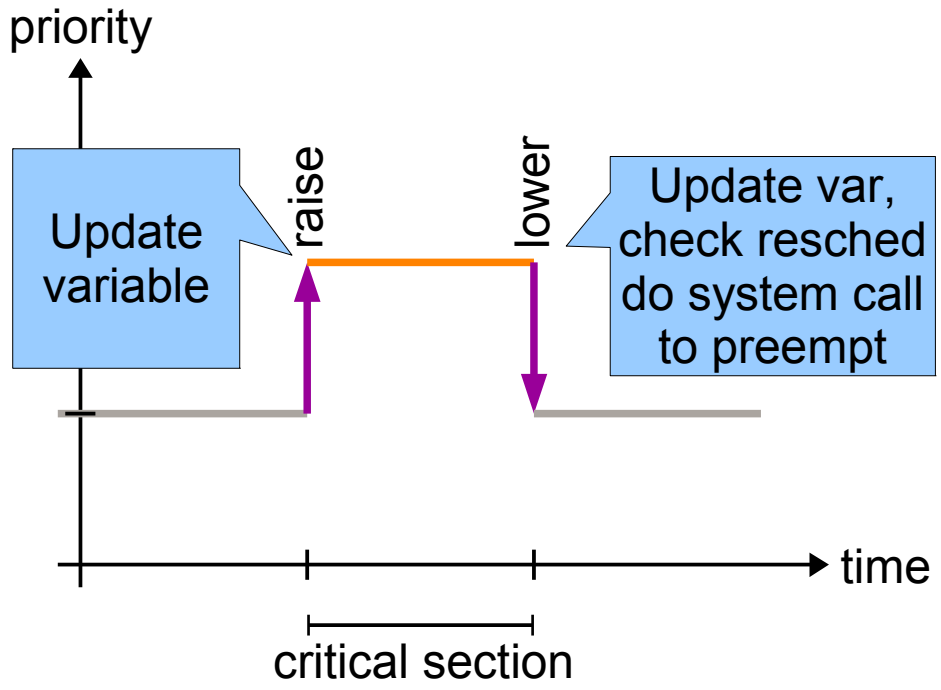
2. decrease priority
by updating the variable

3. check for preemption
in a second variable

Lazy Priority Switching

Optimized critical section:

One system call in the worst case (preemption)



Best case performance (no preemption):

MPC5646C: 688 → 31
TMS570: 843 → 94

(CPU cycles)

Futex Wait Queues

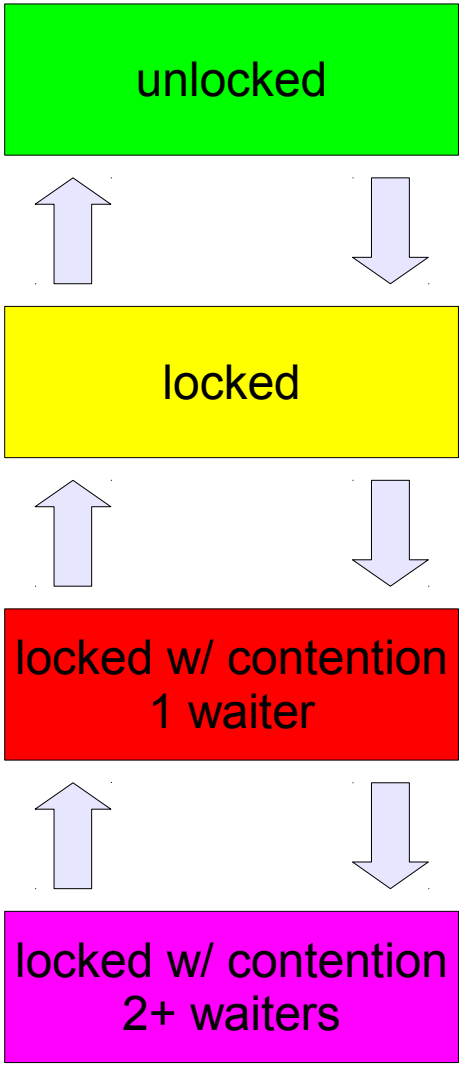
Futex Wait Queues

ARINC sync 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

Idea: use Futexes, like in Linux

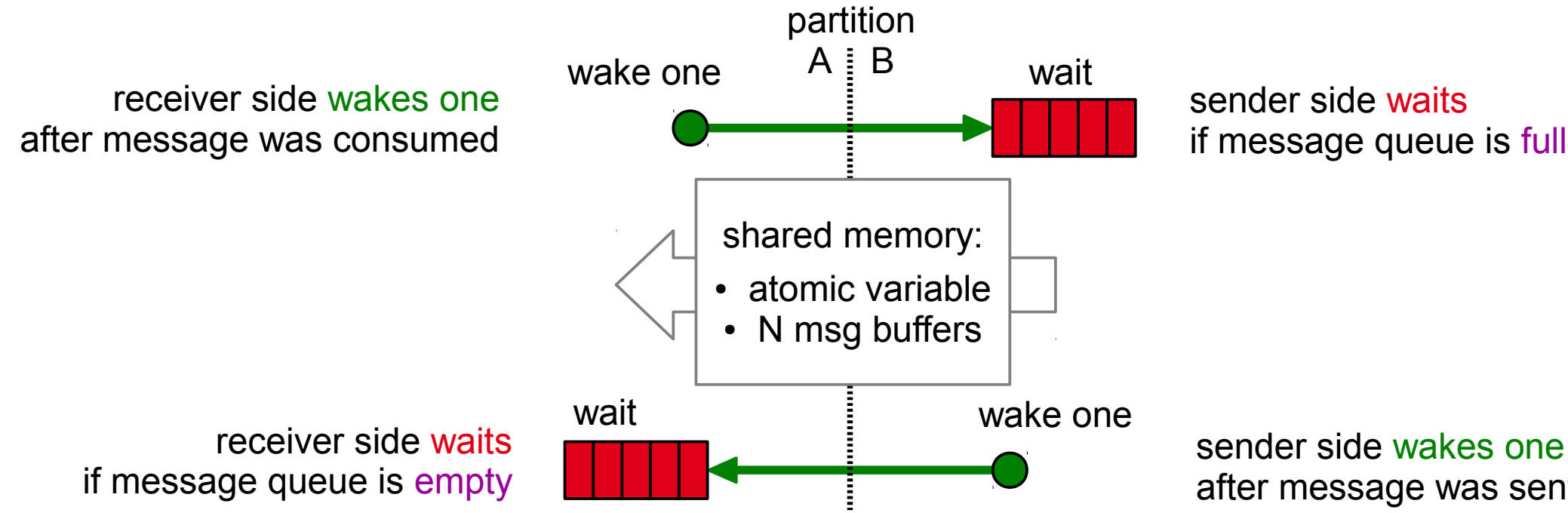
- Atomic operations in the fast path
- Syscalls to wait() and wake() tasks
- Wait queue statically configured
- Build all ARINC sync objects in user space



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



Implementation

Implementation

- Implementation in C99 with some GCC extensions
- Compiler: GCC, CodeWarrior (PowerPC VLE)
- Supported Architectures:
  - ARM v7
    - Cortex-R4: Texas Instruments TMS570 “Hercules”
    - Cortex-A8: BeagleBone Black (for testing), QEMU, Multicore
  - ARM v7-M
    - Cortex-M3/4: STM32F4 “Discovery”, LPC1768 “mbed”, QEMU
  - PowerPC e200
    - MPC5646c (Bolero3M), QEMU
  - Tricore TC1.6E/P
    - TC27x eval board, TSIM

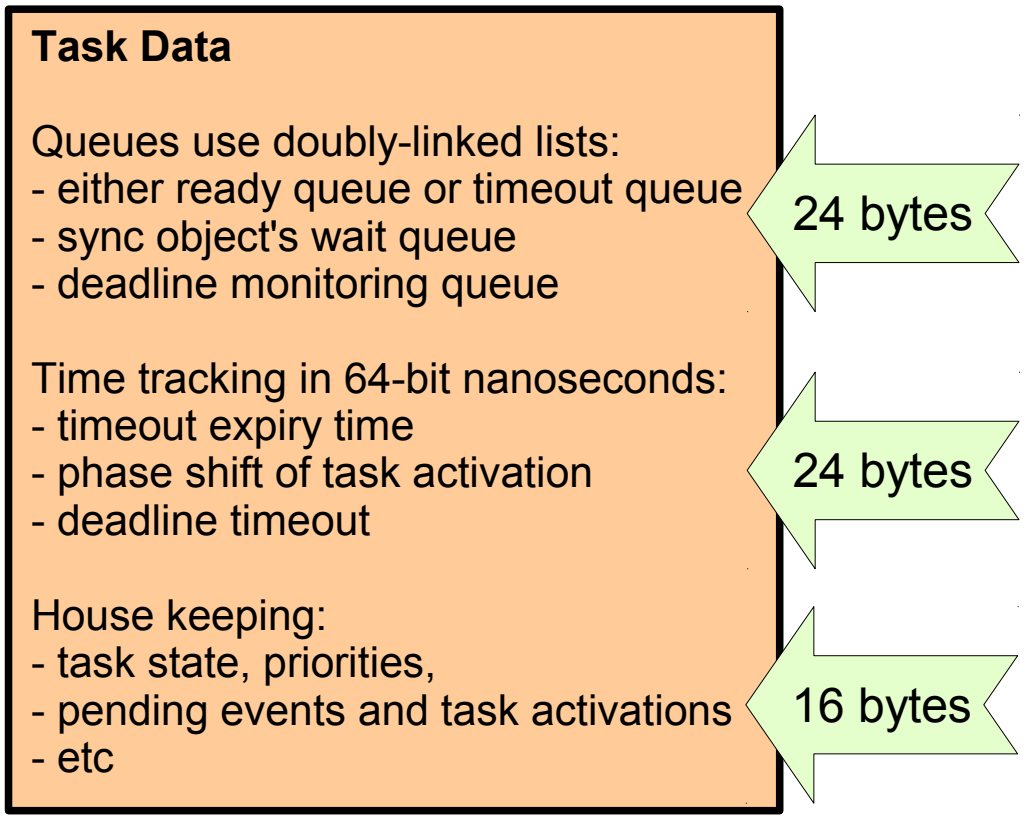
Implementation

- Current Status / Done (July 2015):
  - Full OSEK API + AUTOSAR extensions
  - Full ARINC 653 Part 1 Supplement 3 support
  - Resource partitioning + MPU support
  - Multicore support
  - ARINC Health Monitoring + AUTOSAR Error/Protection Hook
  - ARINC Time Partitioning
  - Deadline monitoring for ARINC and AUTOSAR Time Protection
- Work in progress:
  - AURIX multicore
  - Reservation-based Time Partitioning Concept for Automotive

Implementation

Resource Efficiency: RAM is precious!

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



Implementation

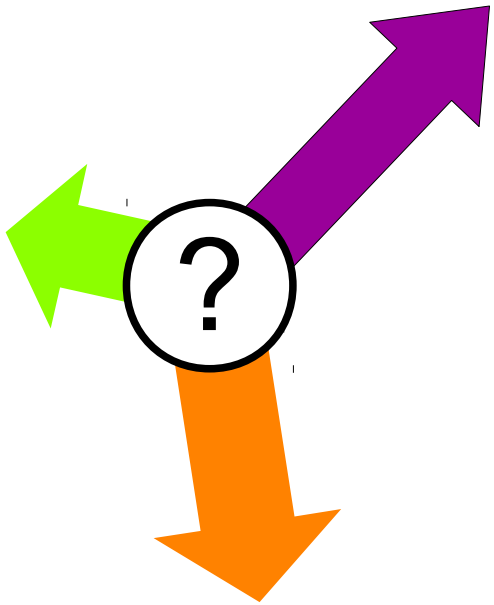
Resource Efficiency: And flash as well!

- Example Configuration:
  - 7 resource partitions, 3 time partition
  - 32 tasks, 12 hooks, 1 ISR
  - 4 schedule tables, 2 alarms
- Kernel memory usage on TMS570:
  - 14.9 KiB code (ARM thumb code, gcc -Os)
  - 9.5 KiB config
  - 15.0 KiB data (1.4 KiB stack, 3.6 KiB regs, 6.3 KiB rdyqs)

Research

Research Topics

- Engineering challenges
  - Make it safe
  - Make it fast
  - Low memory consumption



- Research challenges
  - Techniques to mitigate costs of partitioning
  - Interrupt-Handling ↔ Strict Temporal Isolation
  - Bounded Interference on Multicore

Research Topics

Temporal Isolation

- ISO 26262 requires temporal isolation
  - An erroneous partition must not interfere with other, probably higher critical, partitions!
- ARINC Time Partitioning
  - Fixed slots for partitions in a TDMA schedule
  - Conservative, System Utilization often <50%
  - (Usually) No Interrupts

Research Topics

Temporal Isolation

- Automotive System
  - Interrupts do not fit well into TDMA schedules
  - System utilization is pushed >90%
  - ARINC is too strict
- Idea
  - Reservation-Based Time Partition Scheduling

Conclusion & Outlook

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

Our research project was successfully …

- Partitioning concepts do not cost that much:
  - 64 Bytes RAM per task for ARINC 653 support
  - 2 KiB RAM per time partition for ready queues
  - Performance costs dominated by MPU reload

… but more research is required!

- Reservation-based Time Partition Scheduling
- Map AUTOSAR RTE communication to Futexes
- Safe & Secure OS for *Internet of Things*
- Integrate in real-time modeling tools for *Industry 4.0*

Thank you for your attention!