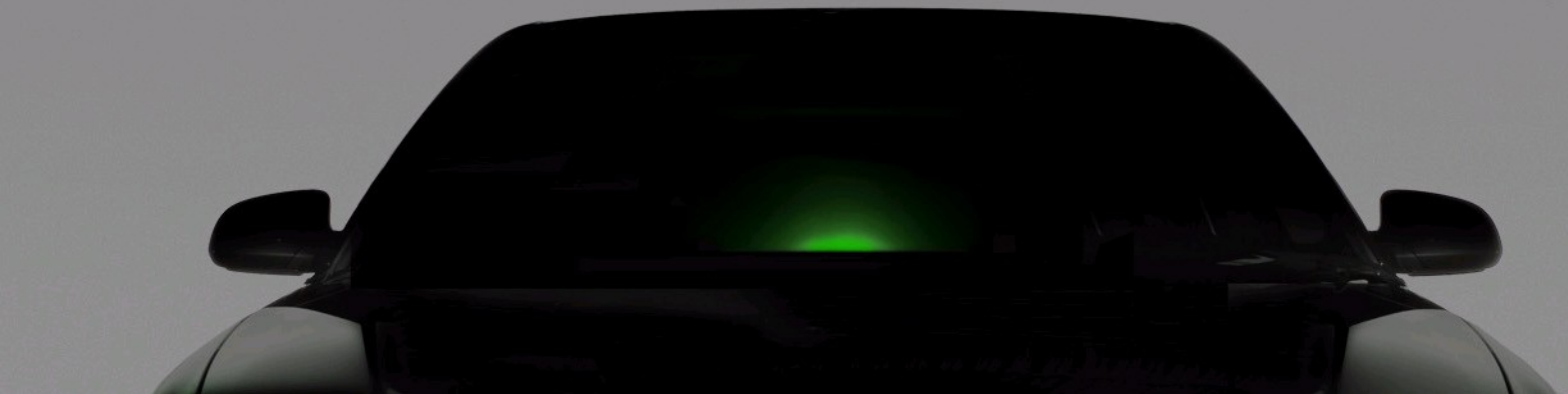
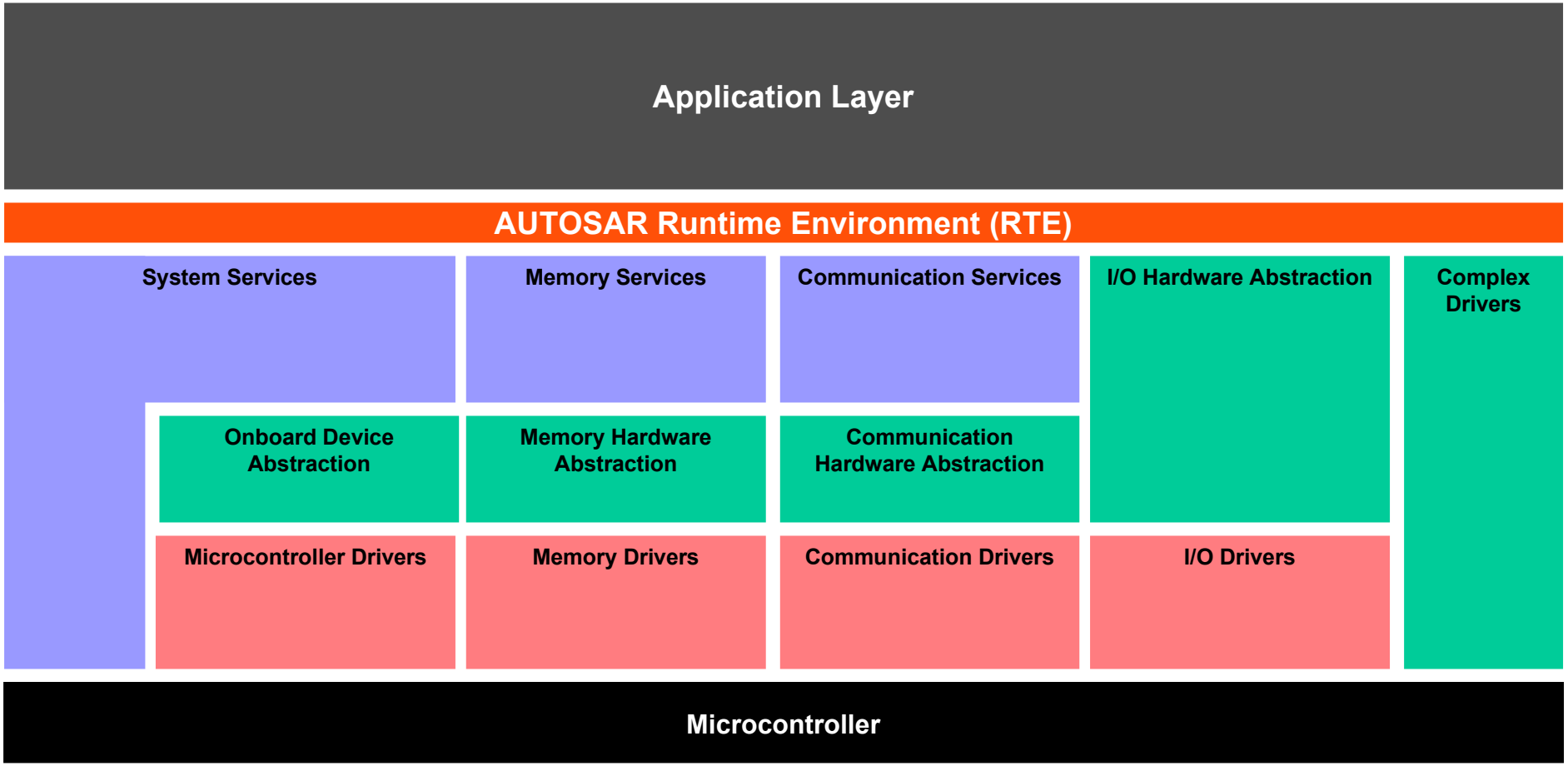
An AUTOSAR-compatible microkernel
for systems with safety-relevant components

David Haworth
Elektrobit Automotive GmbH
David.Haworth@elektrobit.com



The AUTOSAR layered architecture



The AUTOSAR OS module

- Tasks
- ISRs
- Hook functions
- Interrupt locking
- Resources
- Counters
- Alarms
- Schedule Tables
- Synchronization for Schedule Tables
- Memory protection
- Timing protection
- Error handling

A huge amount of code to develop to safety standards

Protection outside the OS

Task provides its own protection

- Error-detecting codes (EDC)
- Memory protection

Procedure:

- Disable interrupts
- Verify or enable access to data
- Perform computation
- Disable access to data, or recalculate EDC
- Enable interrupts

Problems:

- Must be implemented separately in each task
- Performance (computation of EDC)
- Locks out all other activity (may disrupt network activity)

High-integrity MPU driver

Add a high-integrity MPU driver to a standard OS (without memory protection)

- Added in hook functions in all kernel entry and exit points
- Prevents OS and other tasks from modifying critical data

Problems:

- OS is still responsible for register values (local variables)
- Tasks must therefore lock interrupts while processing critical data
- Locks out all other activity (may disrupt network activity)

Minimal high-integrity context switch

Create a high-integrity context switch for a standard OS

- Implements context switch and memory protection
- Task's memory is protected against modification by other tasks
- Task's memory is protected against modification by standard OS
- Standard OS still selects which task is “most eligible”

Problems:

- High overhead of switching MPU twice for each OS service
- Fault in standard OS could select wrong task
- Critical sections could fail
- Stacks cannot be shared among equal-priority tasks

Full implementation of AUTOSAR-OS

A full implementation of AUTOSAR-OS would certainly satisfy the requirements:

- Task's memory is protected against modification by other tasks
- OS can be trusted not to modify tasks' variables
- Critical sections and stack sharing is possible

Problems:

- Very high development costs
- Some features (e.g. timing protection) not well understood

Solution: a microkernel

In the microkernel:

- Task, ISR and hook function management
- Resources and interrupt locking for critical sections
- Interface (with memory protection) to functions of a standard OS
- Events (could be in standard OS, but need speed for RTE)

In the standard OS:

- Counters
- Alarms
- Schedule Tables (including synchronization)

Not implemented (for now)

- Timing protection
- OS-Applications

Design of EB microkernel

Microkernel manages “threads”:

- A “thread” is a generalized abstraction of a task
- Threads can have parameters and can return values
- All other executable objects (ISRs, hooks etc.) run in threads
- OS functionality outside the microkernel (StartScheduleTable(), SetRelAlarm() etc.) runs in threads too
- Even `main()` and the idle loop run in threads
- Most threads run in an unprivileged mode of the processor

Having just one type of executable object leads to simplicity of design

Design of EB microkernel

Microkernel is not re-entrant:

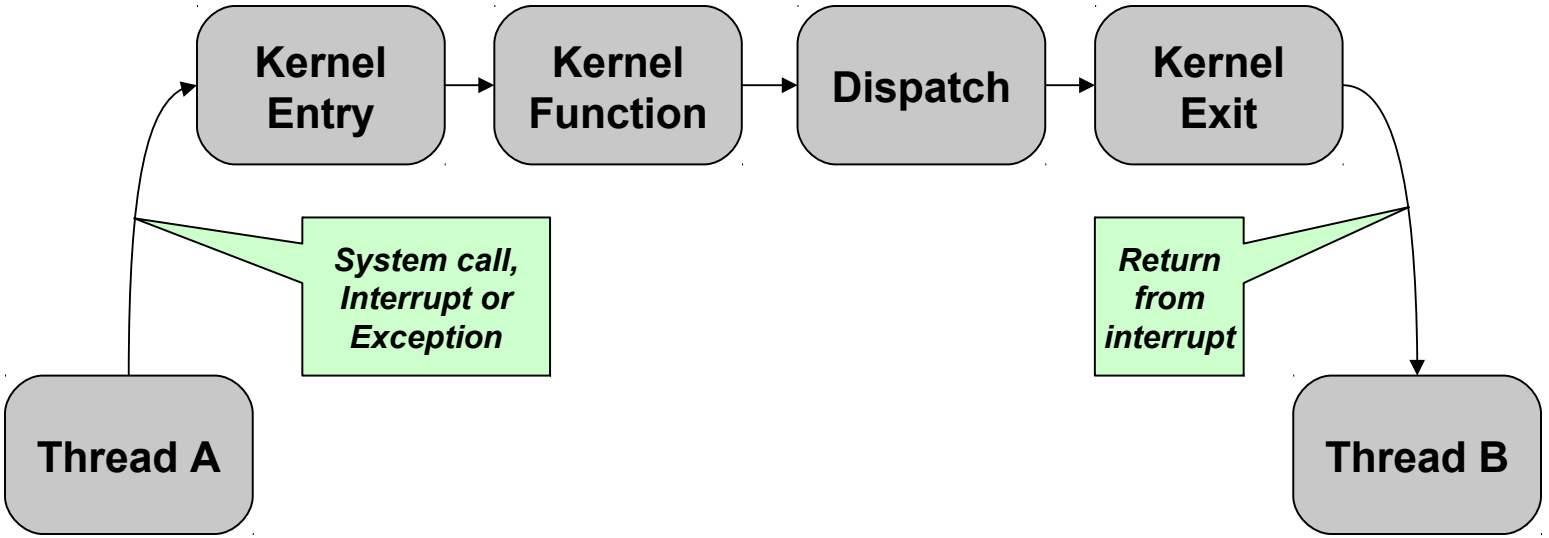
- Interrupts are completely disabled during microkernel execution
- System state is held in explicit structures, not in multiple nested stack frames
- Exception during microkernel execution should not happen
  - If it does, a simple response of shutdown.

Microkernel runs with strict memory protection:

- Does not have access to threads' data or stacks
- “Get” API implemented by passing results back in registers and placing into referenced variables in a library function

Easier to verify the correctness of the design

Microkernel control flow



Kernel entry and exit

Kernel entry (typically assembly language):

- One entry point for each interrupt/exception type
- Save all registers into “register store” of current thread
- Create valid C environment (stack pointer, etc.)
- Call the kernel function associated with this entry point
  - If the kernel function returns: endless loop (shouldn't happen)

Kernel exit (typically assembly language):

- One exit point (“called” by dispatcher)
- Restore all registers from “register store” of current thread
  - (Including PC, so thread resumes where it left off)

There is always a thread available for execution

A new thread has its “register store” initialized as though it were interrupted just before the first instruction of its defined top-level function

Kernel function

Three types of kernel function:

- System call
  - handle internally (tasks, resources, etc.)
  - activate thread in non-safety OS (alarms, schedule tables etc.)
- Interrupt
  - handle internally
  - activate thread for ISR
- Exception handler
  - hardware-dependent; typically activate thread for ProtectionHook

Dispatcher

Selects most eligible thread and lets it run.

- Old thread to “READY” state
- New thread to “RUNNING” state
  - Note: new and old could be the same thread!
- Load memory partition for new thread into MPU
- “Return” to new thread; kernel exit

Microkernel services

- Task control
  - ActivateTask(), ChainTask(), Schedule()
  - TerminateSelf() (implements TerminateTask, but behaves identically in other threads)
- Resources
  - GetResource(), ReleaseResource()
- Events
  - SetEvent(), WaitEvent(), ClearEvent()
- Information
  - GetTaskID(), GetTaskState(), GetISRID()
- OS control
  - StartOS(), ShutdownOS(), ReportError()

All other implemented services (alarms, counters, schedule tables) are delegated to the non-safety OS

What? No interrupt locks?

Problems with interrupt lock functions:

- Not “fast”; needs a system call anyway
- Extra API (up to 6) – more verification
- API calls may unlock interrupts unexpectedly
- AUTOSAR solves this by introducing error checks
  - performance hit
  - cannot use API inside critical section (e.g. SyncScheduleTable)

Solution: use resources instead!

- Two global resources configured with ceiling priority of highest-priority ISR of category 1 and 2, respectively
- Extension to permit nesting (in same thread)

It is now safe to call APIs with interrupts locked (except WaitEvent)

Summary

We have implemented a microkernel that is:

- Functionally compatible with AUTOSAR-OS for correctly written systems
- Implements safety-relevant task, ISR and hook management
- Developed for use up to ASIL-D (ISO26262)
- Delegates non-safety-relevant activities to a non-ASIL subsystem
- Prevents interference from the non-ASIL subsystem (with respect to data, including statically and dynamically allocated variables and local variables in registers)