

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

David Haworth

Elektrobit Automotive GmbH  
Am Wolfsmantel 46  
91058 Erlangen, Germany  
[david.haworth@elektrobit.com](mailto:david.haworth@elektrobit.com)

**Abstract.** The integration of safety-relevant software and software developed to normal quality standards in the same embedded system needs a protection mechanism to ensure that the safety-relevant software cannot be adversely affected by failures in the rest the system. The protection mechanism is usually assumed to be provided by the "system software", which implies that the system software must be developed to the same exacting standards as the safety-relevant software. In the AUTOSAR model that may not be possible in practice.

This article explores ways of isolating the safety-relevant software from the bulk of the AUTOSAR system software while retaining the core functionality needed by the system software and the components that run under it.

## 1 Introduction

Developing software to the rigorous standards required for safety-relevant systems is costly and time-consuming. Standards such as ISO26262 [1], as used in the automotive domain, demand that rigorous procedures be applied to the development of all software that could interfere with the safety-relevant functions of an electronic control unit (ECU). These procedures apply to all phases of development, from the definition of requirements through design, implementation, test, verification and final assessment. In practice, it may not be possible to develop a full system to these standards within reasonable timescales.

An approach that is commonly used in many fields is to isolate software components from each other. By means of the isolation, a failing component is prevented from interfering with the behaviour of other components. Interference can be detected and appropriate action can be taken. The action taken depends on the required availability of the systems. The entire system can be switched into a safe state, or an attempt can be made to restart the failed component if its operation is a necessary part of the safe state of the system. This approach is known as providing "freedom from interference" for the safety-relevant components.

Interference can occur in three domains: the time domain, the communication domain and the data processing domain.

Interference in the time domain, such as safety-relevant functions being blocked by faulty software, can be detected by means of a hardware watchdog, perhaps with software extensions.

Interference in the communication domain, whether caused by outside interference or faulty software, can be detected by the use of error-detecting codes such as cyclic redundancy checks.

Interference in the data processing domain occurs when faulty software inadvertently interferes with the processing of safety-relevant functions. Interference can be caused by modification of data while in storage, by modification of variables during processing or by directly affecting the flow of processing.

## 2 The AUTOSAR Standard

AUTOSAR is establishing itself as the standard for an off-the-shelf “core” software used for ECUs in the automotive field.

In the AUTOSAR model, the Operating System module (OS [2]), the “Basic Software” (BSW [3]) and the “Run-Time Environment” (RTE [4]) are combined to form a standard set of modules that can be configured as required for each ECU. The OS provides priority control of tasks and interrupt service routines (ISRs) as well as timing functions using counters, alarms and schedule tables. The BSW includes device drivers, communications stacks, storage management etc. The RTE presents a middleware-like port-based communication interface to “Software Components”

The Software Components are the software modules that perform the required functionality of the ECU. Some of the Software Components may have safety-relevant functions. In the layered architecture model [5] defined by the AUTOSAR standard, the OS, the BSW and the RTE are all assumed to run with the same trusted privilege level because they need access to hardware peripheral devices. The Software Components are gathered together into groups called “OS-Applications”. which can be trusted or non-trusted.

## 3 Practical Solutions

On many modern microcontrollers it is not necessary to have the highest privilege level in order to access hardware peripherals. On processors like some of the Power Architecture derivatives, peripherals can be configured to allow access from software running in user mode. On others, like Infineon’s Tricore range, there is an intermediate privilege level that permits peripheral access while still preventing access to core registers that control the access rights.

On these microcontrollers it is possible to use the AUTOSAR OS module to isolate the BSW, the RTE and the Software Components from each other. Ideally all the BSW, RTE and Software Components run as non-trusted applications. Using this model reduces the amount of software that must be rigorously developed to just the OS plus those portions of the RTE that run alongside the safety-relevant software components.

There remains the problem of the OS module. A complete AUTOSAR OS module is still uncomfortably large. The standard specifies several types of non-executable objects as well as executable objects: tasks, ISRs and hook functions. Each of the executable objects has different characteristics and a traditional design for an embedded operating system typically implements them all in different ways for reasons of efficiency.

An analysis was undertaken to determine whether it would be feasible to implement a smaller module that could be used to isolate the OS from the software components. In the first phase of the project the failure modes of an existing AUTOSAR OS module were studied to determine the types of failure that could be induced in safety-relevant software components if there was an undetected fault in the OS. A complete failure modes and effects analysis for the OS would be too large to discuss here, but some of the most significant failure modes found were:

- Failure to restore a task’s working registers correctly after an interrupt or preemption. This could result in a task’s temporary variables becoming corrupted and an incorrect result being computed from correct input data.
- Failure to restore a task’s program counter correctly after an interrupt or preemption. Large errors here would probably be detected, but small changes would cause a task to omit or repeat a few instructions. The effect could be the same as an error in the working registers, but could result in a task omitting safety measures that it should perform.
- Failure to observe task priority rules. Implicit critical sections could be occupied simultaneously by two or more tasks and stack sharing by tasks would not be reliable.
- Failure to disable interrupts when requested by a task. Explicit critical sections could be occupied simultaneously by two or more tasks.

The second phase of the project attempted to identify methods of detecting the effects of faults in the OS and protecting software components from those effects. During the analysis several possible solutions to this problem were considered; these potential solutions are described in detail in the following sections.

### 3.1 Data integrity based on error-detecting codes

The integrity of safety-relevant data could be verified numerically using methods that are common in communication protocols. One problem with this approach is that the overhead of the data verification increases with the volume of data. The volume of data in a typical communication message is small compared with the data set for a complex computation. Calculating an error-detecting code for a large data set before and after each iteration of the computation could be prohibitive.

Another problem with this approach is that the data-integrity module would need to be integrated with the software component, thus increasing the complexity of the software component slightly. Such an integration would have to be done separately for each software component.

A third problem with this approach is that while the computation is in progress all interruptions and preemptions need to be disabled. This is because an error in the OS or in another task could cause a variable to change its value unexpectedly and thus cause the algorithm to produce incorrect results. Disabling interrupts might mean that higher-priority activities fail to run on time, which might provoke instabilities elsewhere.

### 3.2 Data integrity using stand-alone memory protection

The integrity of safety-relevant data could be assured by using a stand-alone memory protection module. The problems with this approach are the same as the approach using error detecting codes, as described above, with the exception of the overhead of numerical verification.

### 3.3 Adding high-integrity drivers to a standard OS

High-integrity drivers for the memory protection hardware could be added to a standard OS. The drivers would prevent the OS (and the other tasks and ISRs) from being able to modify the stored variables belonging to safety-relevant components. At first glance this might seem to be a viable solution, but the OS is still responsible for the safe keeping of the register values of any task that gets interrupted. Unexpected modification could cause an interrupted algorithm to produce incorrect results.

All interrupts would have to be disabled to prevent this possibility. Such a solution would have the overheads of an OS with memory protection, but would not have the multi-tasking benefits of the OS during the safety-relevant computations.

### 3.4 A minimal high-integrity context switch

One very interesting possibility that was considered was the development of an outer “wrapper” for the OS. This outer wrapper would replace the context switch in a standard OS with a high-integrity version and would implement the memory protection drivers. Access to the memory regions belonging to the tasks (including their stacks and saved register content) would be disabled while the OS is running. In this model the standard OS is responsible for selecting which task should run, but the mechanism of switching to the task is controlled by the high-integrity software.

The cost in CPU time to switch the memory protection mapping for every call to an OS service and back again when returning to the caller could be prohibitive on some architectures. Furthermore, AUTOSAR applications typically rely on the mutual exclusivity of tasks to avoid explicit locking of critical sections. If correct prioritisation were not guaranteed, explicit locking would be needed, with the accompanying overhead. Stack sharing among mutually exclusive tasks would also be unsafe.

To summarise: this approach would provide all the required protection, but with extra CPU time overhead for every OS call and an increased RAM footprint.

### 3.5 A full implementation of the AUTOSAR OS standard

A full implementation of AUTOSAR OS has the memory protection features common in other operating systems that are often used as means to prevent interference between software components. The AUTOSAR OS module is clearly intended to provide the protection mechanisms and would in fact do so. The only problem is the development time and cost.

A full AUTOSAR-OS implementation includes all the counter, alarm and schedule table drivers along with global time synchronisation. The software associated with this is at least as large as the task and ISR management. In addition, there is timing protection using execution-time budgets and rate limits to guarantee schedulability. Unfortunately, the use of this method is not well understood; simpler deadline-based timing protection appears to be preferred.

### 3.6 A minimal operating system kernel

It should be possible to specify a subset of the AUTOSAR-OS standard that provides the essential features within acceptable cost and time budgets. The subset would have to provide the management of executable objects. Such a minimal operating system kernel would provide freedom from interference for all the executable objects under its control, including many features of the standard OS as well as the BSW, RTE and all the software components configured by the system designed.

In fact, such an approach looks very similar to the classical microkernel concept. The analysis concluded that this approach would provide a usable product, the development of which was still achievable within acceptable timescale and cost limits.

## 4 Design of the AUTOSAR-compatible Microkernel

From the outset, it was decided that simplicity of design should be paramount to minimise the size of the microkernel. Reducing the size reduces the development effort and reduces the possibility that a latent error could affect the safety of the system. Where strict conformance with the AUTOSAR specification would add complexity in the form of special cases or extra checks at runtime, the AUTOSAR requirement should be rejected, or relegated to a set of additional checks that are present only during development.

It was decided that the microkernel should retain as little state information as possible. The state of an object is not necessarily stored directly but must often be inferred from the microkernel's state variables. This has the advantages of eliminating multiple state variables and possible conflicts between them. The cost of computing the state of an object is increased, but is offset by the savings of not maintaining that state, and the cost only occurs at the place where the state is requested.

Finally, it was decided that all code, including the microkernel and other code running in supervisor mode, shall run with memory protection enabled and with

the memory protection boundaries set as tightly as possible. This decision means that there is no longer a concept of “absolute trust”; the access rights for every piece of software must be specified explicitly.

#### 4.1 Threads

An AUTOSAR-OS must be able to manage executing instances of tasks subject to the prioritisation requirements specified by AUTOSAR.

The abstract concept of a “thread” as an active instance of a task was developed. A thread queue, consisting of a singly-linked list in order of descending priority, allows the highest priority thread to be determined readily. A thread is inserted before the first thread whose priority is strictly lower than the thread being inserted. The correct position is found using a linear search from the queue head. This insertion algorithm provides the first-in, first-out sequence for tasks of equal priority, as required by AUTOSAR.

In general, removing a thread from the queue also requires a search, but in most cases a thread terminates itself and at that time it is at the head of the queue. A doubly-linked list might be considered for the purpose of eliminating the search when terminating another thread, but that normally only happens under error conditions, so the overhead of maintaining the backward links would probably outweigh the savings.

The microkernel maintains a variable containing a reference to the head of the thread queue. It also maintains a variable containing a reference to the current thread. While a thread is executing on the processor the current thread and the head of the thread queue are the same, but while the microkernel is executing this condition might no longer hold.

Threads typically run in a low-privilege mode of the processor, so services of the microkernel must be called by means of a system call trap.

Given a suitable range of priorities, the other executable objects required by AUTOSAR (ISRs and hook functions) can be implemented using threads too. Thus we have a simple non-reentrant structure: the processor is executing either the microkernel or a thread. The transition from thread to microkernel is by means of a hardware trap. The transition from microkernel to thread is by means of a return-from-trap instruction.

Each thread has its own interrupt locking level, which is written to the interrupt controller whenever the thread becomes active. The level is related to the priority. Using the locking level, ISR threads can guarantee correct nesting, and resources can be shared with ISRs as well as tasks.

A microkernel design that allocated a thread from a pool on demand was considered but rejected in favour of the static configuration policy of AUTOSAR under which a thread is assigned to each task at compile time.

AUTOSAR’s concept of multiple activations would result in a task having as many threads as its activation limit, which would need a lot of memory. Instead, it was decided to add a job queue in the form of a ring-buffer to each thread, and to place subsequent activations of a task into the queue. On termination of a thread, the next job in the queue gets activated. To preserve the correct

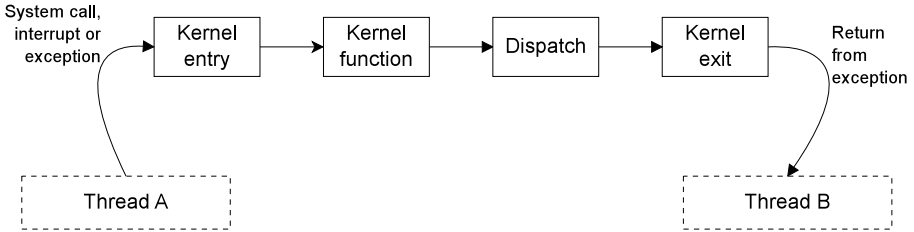
order of execution, if there is a thread with a job queue, all tasks with the same priority must use that thread.

## 4.2 Microkernel structure

The transition from thread to microkernel occurs in the following cases:

- A peripheral device interrupts the processor.
- A processor exception occurs, such as a memory protection or unknown instruction trap.
- A system request is made; this is a special case of a processor exception.

All of these cases are similar, differing only in hardware-specific details. The flow of control in the kernel is depicted in figure 1.



**Fig. 1.** Microkernel control flow

The *kernel entry* routine saves the processor registers then calls the function responsible for handling the request. To simplify the microkernel code (and especially the low-level assembly language part) it is assumed that there is always a thread running, and the microkernel saves the complete processor state at each entry point. The processor state is stored in a structure in the microkernel's space so that the microkernel does not need write access to every thread's stack and does not need to check the validity of the stack pointer.

The *kernel function* performs all internal activities necessary to handle the request. This may result in changes to the thread queue, but not to the current thread variable.

The *dispatcher* selects the most eligible thread and changes the states of the outgoing and incoming threads.

The *kernel exit* routine resumes the new current thread from its previously-saved or newly-initialised state. The same state is saved at every entry point, so the exit routine does not need to determine how the kernel was entered.

The microkernel is not reentrant. This means that the size of the microkernel's stack can be easily calculated. The disadvantage is that an exception that occurs in the microkernel cannot be handled completely and the only available actions are to shutdown or reset the system.

### 4.3 Interrupts

When an interrupt occurs and is accepted by the microkernel, the microkernel identifies the interrupt source and calls a configurable function. For AUTOSAR ISRs, the function is a microkernel function that starts a thread for the ISR.

It is recognised that activating an ISR is a very large overhead, especially when the ISR does little more than acknowledge the hardware and activate a task. Therefore it was decided to add the possibility of specifying a user-defined function to be called in place of the microkernel function that starts the ISR thread. This function is not permitted to use the AUTOSAR services, but can call a small set of the microkernel's internal functions to perform actions such as activating tasks. These user-defined interrupt handler functions must be developed to the same standards as those required of the microkernel.

### 4.4 Resources and interrupt locks

AUTOSAR resources are implemented by raising and lowering the thread's current priority. An interrupt locking level is associated with each resource to permit ISRs to use resources as well as tasks. Each resource has a nesting counter, permitting the same resource to be acquired multiple times by the same thread, up to a predefined limit. The resource is finally released when the nesting counter returns to zero. This extension permits the interrupt locking services to be implemented in terms of resources, thus avoiding the necessity to implement special functionality. Standard AUTOSAR resources have a nesting limit of 1.

In addition to standard resources, AUTOSAR specifies linked resources and internal resources. Linked resources are implemented using multiple resources at the same priority. Internal resources are not present as resources in the microkernel. They are implemented by raising a thread's priority to a higher level when it first gains the processor. The thread's priority remains at that level or higher until it terminates, calls `Schedule()` or waits for an event. Non-preemptive tasks can be considered to use an internal resource whose priority is at least as high as the highest-priority task.

### 4.5 Counters and related services

The AUTOSAR counter, alarm and schedule table objects are implemented using code from an existing AUTOSAR OS, referred to here as the QM-OS, that was not developed using safety standards. To prevent faults in the QM-OS from causing interference the thread abstraction is used. The microkernel provides system calls that each starts a function from the QM-OS in a configured thread. The parameters for the QM-OS function are preloaded into the appropriate registers when activating the thread. The return value of the function is passed back through the microkernel to the original caller. The QM-OS thread runs at a higher priority than the calling thread to guarantee that the QM-OS thread completes before the calling thread continues. To simplify the configuration, two QM-OS threads are available. One is set at a priority higher than the scheduler



priority but lower than ISRs, while the second is set at a priority higher than all the category 2 ISRs.

The configuration of the QM-OS puts pointers to the functions that it provides in its own “system-call” table. For correct operation the function pointers must be in the correct places in the table, but from a safety viewpoint the correctness is not relevant. The microkernel starts a thread with the parameters that it is given and passes back the results. If the QM-OS function violates protection boundaries, the fault is handled just like any other AUTOSAR-OS protection fault. If the QM-OS thread gets terminated the caller is notified.

One problem with the thread mechanism is that some AUTOSAR services return a status code and place the requested information into a referenced variable. The memory protection prevents this from being possible in the QM-OS threads – indeed, it prevents the microkernel from using this mechanism as well. The adopted solution is to return the value in a register. In the first version the requested data replaces the status code. Future versions will use two and maybe more registers, along with structure return values and perhaps a little extra assembler code. The requested data is placed into the referenced variable in a library function that runs in the thread of the caller and therefore with the same protection boundaries. There is no need for a software boundary check on the reference parameters.

## 4.6 Events

An implementation of the AUTOSAR event mechanism in the QM-OS would have been possible but it was decided to implement it in the microkernel. One reason is that the mechanism is used extensively by the RTE, so performance is important. Another reason is that some microkernel support would be needed anyway, to awaken a task from the waiting state. The QM-OS part of the mechanism would be fairly small.

Events can only be used by tasks that are configured for the purpose. A task that is waiting for an event is detached from its thread, but remains active and can only be awakened by `SetEvent()`.

## 4.7 Hook functions

The startup, shutdown, error and protection hook functions, as defined by the AUTOSAR standard, are started in threads with suitably-chosen priorities. Parameters are pre-loaded into the thread’s register storage and, in the case of the protection hook, the return value on termination is retrieved and acted upon. In the first version of the microkernel, the only action available is to shut down. A wider range of actions is envisaged for the future.

## 4.8 The thread/kernel interface

The interface between the code running in threads (the “user” code) and the microkernel is the system-call mechanism of the processor. A C-compatible interface library is provided by the microkernel. This usually consists of a set of

small assembly-language functions that execute a system-call instruction, leaving the parameters intact in their registers. The kernel entry routine saves all the processor registers, so the parameters are available inside the system-call handling functions without having to pass them explicitly as function parameters. On exit from the microkernel the instruction after the system-call is executed; this is normally a return-from-subroutine instruction. This mechanism assumes a processor that passes parameters in registers, but this is normally true for processors used in embedded systems.

Library functions, written in C, are provided where necessary to translate the microkernel's register-based return-value mechanism into the standard AUTOSAR referenced variable mechanism. In some cases the C library function need not make a system call at all; for example, `GetEvent()` can read the pending events for a task without having to enter the microkernel.

All of these library functions run on the thread's side of the thread/kernel interface and are therefore subject to the same memory protection boundaries as the caller. This means that range checking for reference parameters is unnecessary. The library functions are nevertheless developed to the same standards as the microkernel and can be used from safety-relevant threads.

There are some functions in the QM-OS that are directly called. These functions are not safe to be called from safety-relevant threads, but the necessity to do so is not envisaged.

## 5 Conclusion

A prototype AUTOSAR-compatible microkernel has been developed to the outline design described in this paper. The prototype is being used in a real development project alongside a standard AUTOSAR core, including the counter-related features from a standard OS module. Early indications show that the performance of the microkernel itself is comparable with a standard OS and that the counter-related features, while slower, still show acceptable performance.

Work is underway to formalise the design and development of the microkernel. Estimates based on the size of the prototype indicate that the development should be achievable by a small team within reasonable time constraints.

## References

1. ISO/DIS 26262 *Road vehicles – Functional safety* 2009 (draft) International Standards Organization
2. AUTOSAR *Specification of Operating System* v3.1.1 2009, [http://autosar.org/download/R3.1/AUTOSAR\\_SWS\\_OS.pdf](http://autosar.org/download/R3.1/AUTOSAR_SWS_OS.pdf)
3. AUTOSAR *List of BSW Modules* v1.3.0 2009, [http://autosar.org/download/R3.1/AUTOSAR\\_BasicSoftwareModules.pdf](http://autosar.org/download/R3.1/AUTOSAR_BasicSoftwareModules.pdf)
4. AUTOSAR *Specification of RTE* v2.3.0 2010, [http://autosar.org/download/R3.1/AUTOSAR\\_SWS\\_RTE.pdf](http://autosar.org/download/R3.1/AUTOSAR_SWS_RTE.pdf)
5. AUTOSAR *Layered Software Architecture* v2.2.2 2008, [http://autosar.org/download/R3.1/AUTOSAR\\_LayeredSoftwareArchitecture.pdf](http://autosar.org/download/R3.1/AUTOSAR_LayeredSoftwareArchitecture.pdf)

Herausforderungen durch Echtzeitbetrieb

Echtzeit 2011

Halang, W.A. (Hrsg.)

2012, VIII, 140 S. 61 Abb., 29 Abb. in Farbe., Softcover

ISBN: 978-3-642-24657-9