

On Designing Software Architectures for Next-Generation Multi-Core ECUs

Thomas Fuhrman and Shige Wang
General Motors LLC

Marek Jersak and Kai Richter
Symtavision GmbH

ABSTRACT

Multi-core systems are promising a cost-effective solution for (1) advanced vehicle features requiring dramatically more software and hence an order of magnitude more processing power, (2) redundancy and mixed-IP, mixed-ASIL isolation required for ISO 26262 functional safety, and (3) integration of previously separate ECUs and evolving embedded software business models requiring separation of different software parts. In this context, designing, optimizing and verifying the mapping and scheduling of software functions onto multiple processing cores becomes key. This paper describes several multi-core task design and scheduling design options, including function-to-task mapping, task-to-core allocation (both static and dynamic), and associated scheduling policies such as rate-monotonic, criticality-aware priority assignment, period transformation, hierarchical partition scheduling, and dynamic global scheduling. To support ISO 26262 “freedom-from-interference” requirements, sets of tasks can be grouped into “partitions” which can be protected from each other both spatially and temporally, and different scheduling techniques are demonstrated to be capable of ensuring temporal isolation (e.g., hierarchical partition scheduling, or static task-to-core allocation based on partition). In support of navigating through these numerous design options, advanced timing analysis tools are imperative. Many design options can already be evaluated and verified using commercially available timing analysis tools, while tool support for newer design options such as dynamic global scheduling will continue to be developed in the coming years.

CITATION: Fuhrman, T., Wang, S., Jersak, M., and Richter, K., "On Designing Software Architectures for Next-Generation Multi-Core ECUs," *SAE Int. J. Passeng. Cars – Electron. Electr. Syst.* 8(1):2015, doi:10.4271/2015-01-0177.

INTRODUCTION

Future automotive electronic control systems will continue to demand ever increasing computing power to accommodate growing software functionality in powertrain, hybrid, electric, and advanced driver assistance systems. Due to inherent limitations in semiconductor technologies, increasing the clock frequency of processors is running out of steam as a way to increase processing performance. Instead, computing architectures are turning to parallelism in the form of multiple processing cores as the best way to address increasing performance demands.

At the same time, the trend in automotive electronic architectures is to allocate this increasing software functionality onto a reduced number of physical computing nodes in an effort to reduce size, weight, and power consumption. Together, these two trends lead to new design challenges. An increasing number of software-based features and functions must be sequenced and grouped into operating system tasks, and these tasks must in turn be allocated to the processing cores, whether statically (at design time) or dynamically (at run time). Increasingly, the software intellectual property (IP) assigned to a given ECU will be developed by multiple development organizations, such as the OEM and one or more Tier1 suppliers (mixed-IP integration), and may reflect different levels of ISO 26262

safety-criticality (mixed-ASIL integration). To ensure correct operation of critical functionality that shares processing resources with less-critical functionality, ISO 26262 requires mechanisms for “freedom-from-interference.” [1] Accordingly, the task of mapping and scheduling tasks to cores must take into account multiple factors such as balancing the workload across the cores and sharing information across the cores while at the same time ensuring safety isolation and fault containment.

Multi-core design and optimization challenges include the following: (1) select suitable software mapping and scheduling strategies that offer the most potential for optimizing ECU performance (by maximizing parallelism without introducing crippling communication overhead) while also meeting safety, availability and separation requirements; (2) design software safety architectures that can shut down some applications while continuing availability of others on the same system to achieve freedom from interference; and (3) adjust the development process to realize this potential and take the right engineering steps during different development phases.

The above new challenges in designing multi-core software further impose corresponding analysis challenges. Analysis must pre-verify that the software architecture will meet functional, performance and data-consistency requirements on one hand, and ISO 26262 safety

and freedom-from-interference requirements on the other hand. Analysis should also provide guidance to configuring the scheduler (mapping of runnables to tasks to cores, sequencing runnables, specifying task scheduling parameters like periods and priorities), and configuring timing protection mechanisms like execution time monitoring or deadline monitoring.

FUNCTION-TO-TASK MAPPING

Function-to-task mapping is a design step to determine which software functions (runnables in the case of AUTOSAR [2]) to execute within which task. As a scheduling and execution unit, all executions of functions are through task invocation. Each task is defined with an invocation service, either using time or using an event, and a sequence of functions (runnables) within it, often following data dependencies derived from signal processing flows in control design.

The ECU software integration engineer must first decide how many tasks to allocate on a given ECU. In automotive ECUs that use OSEK or AUTOSAR based operating systems, the tasks are statically allocated as part of the OS configuration, either in an OIL file (for OSEK) or XML file (for AUTOSAR). In addition, the selective use of interrupts may be chosen as the way to trigger software functions that need to execute very frequently in response to aperiodic events or with very short response time. Static tasks are typically activated in a periodic fashion by alarms or timers.

On a single-core implementation, one style of task design is to statically allocate a set of tasks with harmonic rates, with only one task running at each rate. As a hypothetical example, the integration engineer may allocate tasks with periods of 10ms, 20ms, 40ms, and 80ms, with a single task at each period. Use of harmonic task periods generally facilitates higher processor utilization, as the tasks may “pack” into the processor more efficiently much like boxes will pack into a bin with less wasted space if the box dimensions are multiples of each other. With this style of task design, the mapping of functions (runnables) to tasks is determined by the desired loop time of the control function as determined by the control algorithm engineer.

While the function-to-task mapping based on control loop time is relatively straightforward, the sequencing of the functions within each task is very complex, as it must comprehend data dependencies between the functions. In some cases the functions mapped to a given task may actually belong to different unrelated software features, with the result that the functions belonging to these different features will be interleaved with each other.

To target a multi-core implementation, the task design must change because the operating system task is the fundamental unit of scheduling on a given core. On a two-core implementation, for instance, a hypothetical task design could consist of *two* tasks at each harmonic period, one running on each core. The determination of which functions are mapped to which task at a given rate (and accordingly executing on which core) is a very complex design problem that must take into account information exchange between the tasks (and hence between the cores) as well as synchronization and concurrency issues arising due to the independent and concurrent execution of the cores. Compared to the single-core task design, more comprehensive analysis

is required to better understand the emerging behaviors caused by parallel execution, for example potential data races and synchronizations due to true parallel task executions on different cores. The analysis should also assist selection of adequate architecture services, such as those for data locking and task scheduling, and balancing design trade-offs such as running a function in a different order, or in a different task, or on a different core.

An additional dimension of complexity in multi-core designs is mixed-IP or mixed-ASIL integration. If the software functions were developed by different IP sources or relate to different ASIL levels, then the mixed-IP and mixed-ASIL software functions may be interleaved arbitrarily within the same tasks executing on a given core unless care is taken to separate them. An operating system task turns out to be a very convenient unit of granularity for the purposes of fault detection, fault mitigation, and fault containment. For example, if a task is misbehaving in terms of its memory accesses or timing behavior, then that task may be terminated while at the same time allowing other independent tasks to continue operating. If the terminated task itself contains mixed-IP or mixed-ASIL functionality, then considerable functional availability may be lost. For this reason, it may be desirable to design tasks at least partially on the desire for IP source and/or ASIL level separation. Such a design offers the opportunity for selective fault containment on the basis of IP source or ASIL level.

As a hypothetical example, consider a design in which there is a desire for selective fault isolation between OEM vs. Tier1 IP, as well as between high-ASIL and low-ASIL functions. One possible task design to enable such fault isolation might be to statically allocate multiple tasks at each harmonic rate, one each for the four combinations of OEM-vs.-Tier1, and high-vs.-low-ASIL. On the other hand, if either IP separation or ASIL separation is desired, but not both, then only two sets of tasks would need to be allocated at each harmonic rate.

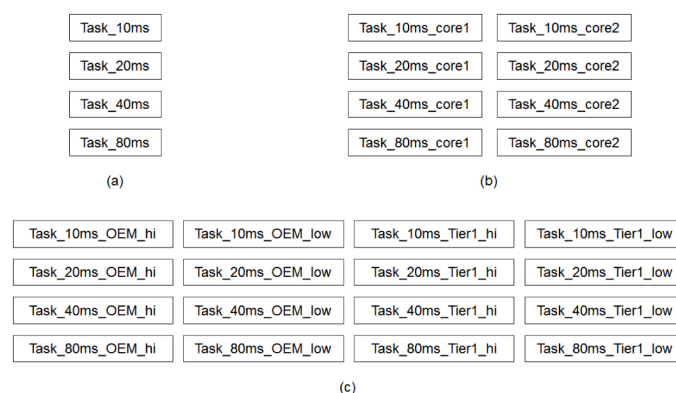


Figure 1. Hypothetical harmonic task sets for (a) single-core implementation, (b) two-core implementation, and (c) four-way separation between OEM-Tier1 and high-low-ASIL.

Some possible harmonic task sets for single-core and two-core implementations without IP and ASIL separation, and for simultaneous four-way OEM-Tier1 and high-low-ASIL separation, are depicted in Figure 1. To ensure correct system behavior, different scheduling strategies may be required to execute the tasks for these three different configurations.

TASK-TO-CORE ALLOCATION

Once the tasks are defined and software functions mapped to them, the tasks must be allocated to cores. In a single-core implementation, as depicted in [Figure 1 \(a\)](#), this allocation is trivial as there is only one core available upon which to allocate tasks. In a multi-core implementation, most automotive operating systems such as AUTOSAR OS would comprehend only static (design-time) task-to-core allocation. In this case, the tasks need to be designed ahead of time with explicit intent to execute on a given core. This situation is depicted in [Figure 1 \(b\)](#).

In case a mixed-IP or mixed-ASIL separation-based task design is desired, as depicted in [Figure 1 \(c\)](#), there is more flexibility in the task-to-core allocation. There are 16 tasks that need to be allocated to cores in that example. In the case of a four-core implementation, a natural way to allocate tasks to cores would be by IP and ASIL category. All OEM_hi tasks could be allocated to core1, all OEM_low tasks could be allocated to core2, all Tier1_hi tasks could be allocated to core3, and all Tier1_low tasks could be allocated to core4. The advantage of this allocation scheme is that it provides a natural fault-containment mechanism between IP and ASIL categories due to the physical independence of the cores. Suppose for example that Task_10ms_Tier1_low misbehaves by consuming too much processing time. Although this misbehaving task may interfere with the other Tier1_low tasks executing on core4, it will not interfere with the OEM_hi, OEM_low, or Tier1_hi tasks running on the other cores due to their physical independence.

In addition to the interferences from processor sharing, the tasks may also interfere with each other through memory. Memory interference can be prevented as well by using the same partitioning scheme. If each of the IP-ASIL task sets are placed in their own memory partitions that are protected by a memory management scheme such as AUTOSAR OS-Applications, then any invalid memory accesses will be blocked across these partitions (but not within a partition). For example, suppose Task_Tier1_low misbehaves by attempting to write into a memory address belonging to partition OEM_hi, OEM_low, or Tier1_hi. This invalid memory reference will be blocked by the memory protection scheme. On the other hand, if Task_Tier1_low incorrectly overwrites a memory location located *within* its own Tier1_low partition, then this memory access will not be blocked.

SCHEDULING ANALYSIS FOR MIXED-CRITICALITY SYSTEMS

Future ECUs will increasingly be mixed-criticality systems for two main reasons:

1. Integration of separate ECUs into centralized multifunction ECUs such as domain control units (reduces HW cost)
2. ASIL-decomposition to minimize the amount of software that needs to be developed in compliance with higher ASILs (reduces SW cost)

ISO 26262 Part 6 (clauses 7.4.9 and 7.4.11) states clear rules for how to handle mixed-criticality systems: freedom-from-interference has to be ensured.

Let's use the hypothetical single-core multi-function ECU shown in [Figure 2](#) an illustrating example of mixed-criticality integration.

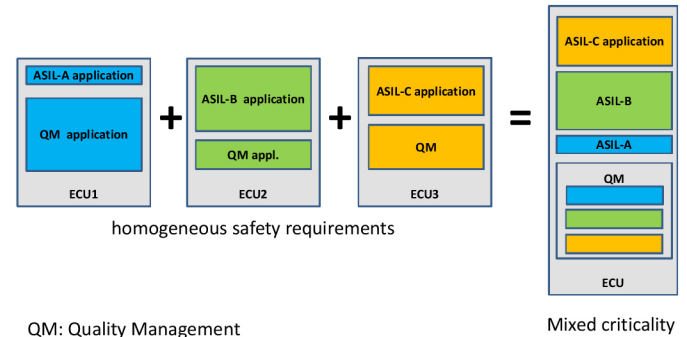


Figure 2. Hypothetical single-core multi-function ECU exhibiting mixed criticality.

Let us assume that we are integrating the set of functions into four tasks:

- Task T1 is ASIL-C and needs to run every 25 ms
- Task T2 is ASIL-B and needs to run every 10 ms
- Task T3 is ASIL-A and needs to run every 1 ms
- Task T4 is QM and needs to run every 10 ms

How should we assign task priorities to schedule these four tasks using an AUTOSAR OS? Without considering safety requirements, a typical design choice would be to assign rate monotonic scheduling [3] (RMS) task priorities, i.e., the shortest period gets the highest priority. This is shown in [Figure 3](#) (generated with Symtvision's scheduling analysis tool SymTA/S) [4].

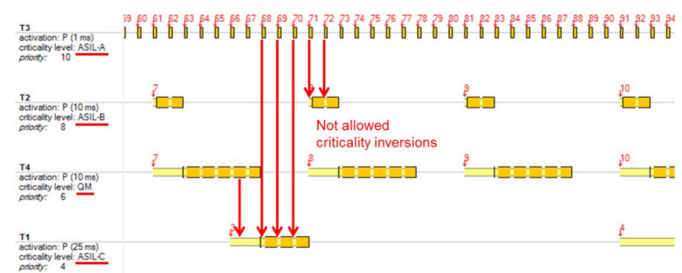


Figure 3. Straight-forward RMS task priorities exhibiting not-allowed criticality inversions.

As can be seen, while all tasks complete before their next activation in the *nominal case*, there is no guarantee that a lower-critical task (here T3 and T4) could not prevent a higher-critical task (here T2 and T1) from executing in the error case (for example, if a lower-critical task goes into an infinite loop). Freedom from interference is not ensured; the schedule cannot be certified according to ISO 26262.

What about assigning priorities in accordance with ASIL level? This is shown [Figure 4](#).