PharOS, a multicore OS ready for safety-related automotive systems: results and future prospects

C. Aussaguès[1], D. Chabrol[1], V. David[1], D. Roux[2], N. Willey[2], A.Tournadre[2], M. Graniou[3]

1: CEA, LIST, Embedded real-time systems laboratory
2: Delphi France SAS
3: PSA Peugeot Citroen

Abstract: Automotive electrical/electronic architectures need to perform more and more functions that are mapped onto many different electronic control units (ECU) because of their different safety levels or different application domains (body, powertrain, multimedia, etc.). Freedom of interference is required to comply with the upcoming ISO 26262 standard for mixing different ASIL levels on the same ECU and is also required to cope with the safe integration of software from different suppliers. PharOS provides dedicated software partitioning mechanisms as well as controlled and efficient resource sharing by construction, from the design to the implementation stages. The main features of PharOS, contributing to this property, are presented in this paper as well as the results on its application an industry-driven case study and associated future prospects.

Keywords: dependability and safety, real-time systems, multi-core, automotive embedded software.

1. Introduction

Automotive electrical/electronic architectures need to perform more and more functions that are mapped onto many different electronic control units (ECU) because they have different safety levels or different application domains (body, powertrain, multimedia, etc.). Next-generation vehicles need reduced development costs and reduced energy consumption and this can be achieved by reducing the number of embedded ECUs [1].

This evolution of automotive architectures requires, firstly, ECUs with more computing power. Today microcontroller suppliers can offer multi-core architectures, where computing power is increased by adding cores rather than by over-clocking the CPU and so for the same CPU capability, this solution increases much less the power consumed. However, carmakers and automotive suppliers lack the appropriate support tools to obtain the full benefit from these new paralleled architectures without compromising safety requirements.

Secondly, non-interference among different functions of “merged” ECUs must be ensured. It is required also to comply with the upcoming ISO 26262 standard and to solve the problem of mixing different ASIL levels on the same ECU. Freedom from interference is also a property required to cope with the actual way of integrating software from different suppliers in automotive industry. It calls for dedicated software partitioning mechanisms as well as controlled and efficient resource sharing.

Standard approaches in automotive software architectures, like the AUTOSAR-based ones, address some of these requirements partially, such as software modularity and portability, but do not yet provide complete and proved solutions to safety aspects related to non-interference and optimal use of multi-core microcontrollers.

To this aim, CEA LIST has developed, in collaboration with Delphi, PharOS, a technology for design and implementation of safe embedded real-time multi-core systems. It shares the same parallel time-triggered and safe-by-construction paradigms of the OASIS technology [2] but coping with the specific automotive requirements. Such approaches allow conciliating efficient parallelism management and real-time determinism [3].

PharOS provides safety-assisted real-time design, temporal & spatial partitioning mechanisms and dual-core support, optimized in terms of memory footprint and computing performance required by the highly-constrained automotive environment.

The icing on the cake is that PharOS can effectively reduce the development cost by simplifying the validation of components and limiting the need of global system revalidation.

This paper will present on one hand the main features of PharOS easing the design and implementation of safe-by-construction and highly available embedded real-time systems and on the other hand the results and future perspectives in the automotive domain.

2. PharOS execution and protection features

A real-time system coordinates elementary activities (such as data acquisitions/actuations from/to the environment or computations) that are performed by processing units. Each elementary activity has its associated temporal constraints, which can be derived or specified explicitly: an *activation rhythm* (not necessarily periodic), a time interval when the activity has to be performed (i.e. an *earliest start date* and a *deadline*), and dependencies with others activities (communications, atomic requests). The coordination among all activities implies synchronization on the physical (i.e. real) time—to ensure the temporal coherence—and synchronization of the sequential and conditional activities. Thus, temporal distribution of different activities must be controlled and, consequently, defined by the task model.

Furthermore, to allow valid offline verification (e.g. proofs, tests, etc.), the system must be *deterministic*. Indeed, predictability and reproducibility of software behavior in both temporal and logical domains guarantees the coherence between, offline analysis and tests, on one hand, and actual execution, on the other. To ensure such determinism, sources of asynchronism, such as scheduling (e.g. preemption), variation of the execution time, and communication delays must be taken into account at the design level in order to control interactions among tasks. Therefore, all temporal constraints on interactions must be explicitly provided by the tasks model.

2.1. Observability Principles

When asynchronism is not accounted for at design level, interactions can result in out-of-date or inconsistent data being manipulated, which can lead to non-reproducible behavior or failures. Indeed, in absence of synchronization mechanisms, it is impossible to know whether production processing is complete or not. Similarly, consulted data must be consistent throughout the associated consumption processing. Furthermore, expiration rules must be provided in order to compute bounds for the size of communication buffers.

An *observation of a temporal variable* is a couple *(X,V)* where *X* is the value of the variable and *V* is the formal visibility instant of this value. Communications are then based on the following principles (see Figure 1):

- A variable *X* is visible only from its visibility instant *V*.
- An observed value of a variable *X* is not modifiable, neither by its producer nor by consumers.
- An observed value of a variable *X* must remain available until a consumer can use it.

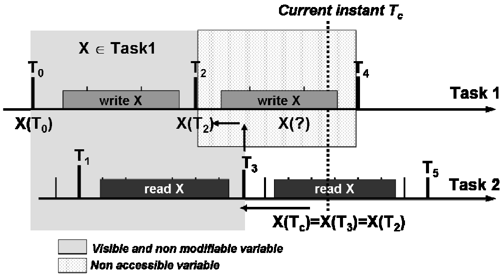


Figure 1. Observability principles

Consider two communicating tasks shown in Figure 1. Assume that Task 1 produces a new value for the variable *X* once in the interval $[T_0,T_2[$ and once in the interval $[T_2,T_4[$; Task 2 consults the value of *X* at some current date $T_c \in [T_3,T_5[$. As the date T_c is a priori unknown, the determinism principle requires that the value of the variable *X* at the date T_c is equal to the value that was the last one observed at the date T_3. Similarly, this last value is given by the observation (X, T_2). In other words, the consulted value of a variable is always a past one and equal to its value at the last formal visibility date less than or equal to the earliest start date of the current time interval of the consulting task.

2.2. Time- and Event-Triggered Model

PharOS is based on a Time-Triggered (TT) execution paradigm [2][5]. The system observes the environment and initiates its activities at some predetermined points of the globally synchronized time. For each task, its time-scale, observations, and interactions are defined at the design step, allowing a precise control at the execution of the system.

In the automotive domain, many activities (e.g. signal capture) have temporal constraints with durations smaller than few microseconds. Hardware performance may not allow addressing such activities with Time-Triggered paradigm. Therefore, in addition to Time-Triggered execution, PharOS also extends the principles described in the previous section to Event-Triggered (ET) activities.

In PharOS, the decomposition in tasks corresponds exactly to the processing that should be executed in parallel (i.e. activities that are not directly dependent). Moreover, a task can be defined in time-triggered domain or in event-triggered one. The design in PharOS imposes no constraint on the decomposition of an application in tasks, nor triggering domains.

2.2.1. Time-Triggered tasks

A TT task—also called an *agent*—is an autonomous entity defined by a deterministic labeled transition system, where labels represent *elementary activities* (denoted EA) representing sequential computation. Each EA has an associated *deadline D*, i.e. its latest end instant and a *quota Q*, its maximum authorized execution duration. An EA is executed within a temporal window *w* defined by an interval $[T_s,T_t[$ (see Figure 2), where T_s is the earliest start date, and T_t is the latest termination date of the activity.

All dates belong to a predetermined subset of ticks of *real-time clocks* defined in the system. Each clock is a couple (*P*, *d*), where *P* is the period and the *d* initial phase.

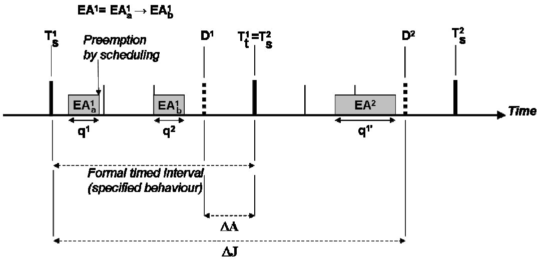


Figure 2. TT tasks model

By default the deadline *D* is equal to T_t or computed according to different execution paths. It can also be explicitly specified by the designer for a finer control of jitter and activation delay. The quota *Q* is the maximal authorized execution duration, i.e. the inequality $Q > \sum_{i=1}^{N} q^i$ is controlled at runtime, where q^i is the execution time of the executed EA part and *N* is the total number of executed parts (preemptive system).

In the PharOS approach, the quota has no impact on application design, but is only used for monitoring and, by offline tools, to perform the CPU sizing analysis: to verify whether hardware performance is sufficient for the application requirements or not (and to compute the associated maximum CPU load). Thus, in order to ensure timeliness of elementary activities, quotas represent upper bounds of their WCET. Quota violation is therefore an error and will be managed as described in section 3.3.

An execution of an agent is an infinite trace of its defining transition system. An execution defines a sequence of temporal windows (denoted *W*) derived from the constraints specified at design phase and such that $\mathrm{T_s^w}$ of the current temporal window $w \in W$ is equal to $\mathrm{T_f^{w-1}}$ of the previous window denoted $\mathrm{w}-1$.

The following temporal constraints can be expressed, monitored, and guaranteed by the system (see Figure 2):

- *Activation delay* ($\Delta\mathrm{A}$)—the minimum delay between executions of two consecutive elementary activities: at the latest, the first activity must finish before $\mathrm{D^w}$, whereas the second cannot start before $\mathrm{T_s^{w+1}}$. Hence, $\Delta A = \min\{T_s^{w+1} - D^w \mid w \in W\}$.
- *Jitter* ($\Delta\mathrm{J}$)—the maximum delay between executions of two consecutive elementary activities: at the earliest, the first activity can finish at $\mathrm{T_s^w}$ (zero, i.e. negligible, execution time); similarly, the second must start before D^{w+1}. Hence, $\Delta J = \max\{D^{w+1} - T_s^w \mid w \in W\}$.

The basic temporal constraints, such as the deadlines and sizes of temporal windows (i.e., $T_t - T_s$), are directly incorporated in the task specification allowing to describe behaviors according to external events (e.g. received network frame) or internal (e.g. state of local variable). Therefore, each agent can adopt dynamic behaviors, periodic or not, regular or not for which all possible sequence combinations are known and incorporated in a graph (see section 3.1). Although each TT task must have an associated basic clock, all the clocks are defined globally for the application and can be used by all TT tasks for synchronization. Thus, tasks can be synchronized (see Figure 3):

- At system start (potentially with a bounded phase shift)—first of all, the global reference of the application is specified as a formal instant called *inittime*; then a start delay $\Delta\mathrm{S}$ relative to *inittime* is associated to each task. It is important to observe that *inittime* and $\Delta\mathrm{S}$ need not necessarily belong to the same clock. Hence, the starting instant of a task is given by $T_0 + \Delta\mathrm{S}$, where T_0 is the last formal instant on the clock of $\Delta\mathrm{S}$ before *inittime*.
- At a specified instant on any given clock. For example, the synchronization between Agent 1 and Agent 2 in Figure 3 is obtained by Agent 1 advancing to the fourth and Agent 2 to the second next tick on the clock C_{10}.

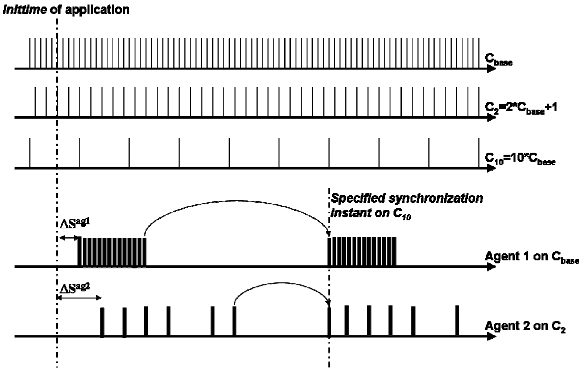


Figure 3. Tasks synchronization

2.2.2. Event-Triggered tasks

An ET task—also called a *handler*—runs one elementary action when an external event is raised and enables a hardware interrupt. The associated computation is executed in a growing temporal window $\left[T_e, T_f\right[$, where $\mathrm{T_e}$ is the earliest date when the event can occur, whereas $\mathrm{T_f}$ is the termination instant unknown until the event does occur and defined by $T_f = T_o + \Delta_o$ with $\mathrm{T_o}$ the measured effective occurrence date and Δ_o the minimum interval with at most *N* occurrences of the same event. Similarly to TT tasks, a quota *Q* (an upper bound for the WCET) can be specified for the execution time of a handler.

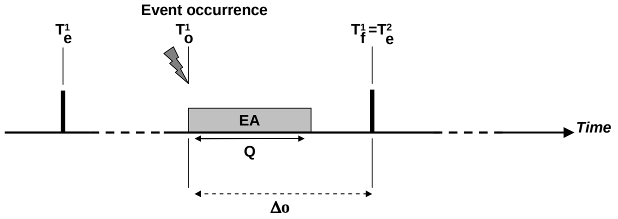


Figure 4. ET tasks model

Altogether, the above parameters specify the largest acceptable workload.

2.3. Temporal communication

All communications/interactions among tasks are explicitly described at design phase and incorporate temporal aspects. The following temporal constraints can be specified and controlled:

- *Age delay* is the maximum delay from which the consumed or sent value (the earliest value) has been produced.
- *Delay before injection* is the maximum delay between effective event occurrence and its processing by the system. This delay also comprises the injection of data from ET domain to the TT one.
- *Expedition delay* is the maximum delay between data production and its delivery to the external environment.
- *Expiration delay* is the maximum delay from which the data becomes obsolete.

All these constraints (based on the observability principles) associated with temporal behaviors of each task provide means for expression and control of end-to-end temporal constraints, i.e. maximum delay between data acquisition and associated actions.

On the basis of the communication principles presented above, PharOS provides the following communications mechanisms within the TT domain:

- *Temporal variables* are real-time data flows: values, available to all agents, stored and updated by a single writer—the owner agent—at a predetermined temporal rhythm.
- *Temporal messages* represent a mechanism for sending typed messages with associated visibility dates.
- *Temporal flags* are a special case of temporal messages for notifying only event (e.g. implicit boolean value) without associated data (for optimization purposes).
- *Temporal blackboards* generalize the first two above mechanisms to allow several agents to contribute to the same temporal variable at a bit granularity.

ET <=> TT (1 agent to 1 handler, and reciprocally) define mechanisms derived from temporal variables and messages to exchange data between the two domains.

3. Temporal and spatial partitioning

To implement partitioning, the execution is structured in different contexts (called execution context) in PharOS:

- An application layer (compounded of different tasks), that executes the main functions (there is one applicative execution context per task),
- A system layer that manages task execution control and inter-task communications (there is one system execution context per task),
- A micro-kernel that manages time and task context switching (it is the only atomic execution context).

The kernel is the association of the micro-kernel and the system layer and is designed to be independent of any application. The application layer is compounded of two execution domains:

- Time-triggered (TT) tasks, where processing is activated only by time evolution,
- Event-triggered (ET) tasks (also called handlers), where processing is activated on the occurrence of an associated event (Input/Output interrupt). This execution paradigm is typically required when the temporal rhythm of events is too tight to be met by TT tasks (limited hardware performance).

The kernel implements generic spatial and temporal partitioning mechanisms [4] to protect each execution context. It uses the protection runtime information (e.g. MPU descriptors, timers, temporal diagram) automatically extracted from functional and sizing constraints provided at software design phase. The protection mechanisms are configured for: earliest/nearest error detection and confinement, non-interference between application and system layers, nor between domains, nor among applications, i.e. TT and ET tasks.

3.1. Temporal partitioning

From the design step, offline analysis of the global software sizing can be performed in order to verify whether provided hardware performance is adequate to satisfy application constraints [2][6]. The guarantee that all the tasks can meet their temporal constraints is provided for both nominal and failure contexts. For this analysis to be relevant the execution of the system must conform to the analyzed design, including the associated temporal constraints. The following errors must be detected:

- *Local sizing error*: task execution consumes more time than expected. This error must have no impact on other tasks.
- *Global sizing error*: a task reaches its deadline. Required CPU load is therefore greater than hardware capabilities. An increase of CPU quartz would suffice in the case of an error in the global CPU load calculation performed during offline analysis. Such operation is performed without changing the design of the application.
- *Flow execution error*: a task attempts to take an execution path inconsistent with its specification.

In PharOS, detection of these errors is performed by monitoring the deadlines, quotas, and execution sequences for each task.

3.1.1. Deadline monitoring

In the TT domain, the system observes its environment and initiates processing operations at recurring, although not necessarily periodic, predetermined instants in time. Therefore, effective execution takes place between two instants—an earliest start date and a latest termination date—split potentially into several execution parts due to pre-emptive scheduling policy (see Figure 5). Execution support uses therefore a hardware timer to ensure EA termination at the associated deadline, otherwise an error is raised.

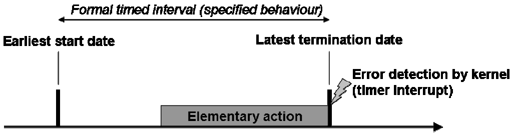

Formal timed interval (specified behaviour)
Earliest start date
Latest termination date
Error detection by kernel (timer interrupt)
Elementary action

Figure 5. Deadline monitoring

The same—hardware timer—mechanism is used to ensure timely termination of handlers in the ET domain. The main difference is that the deadline for a handler is determined at the occurrence of its associated interrupt.

3.1.2. Execution time monitoring

CPU is a shared resource for which scheduling policy determines, at every instant, which tasks can use it. Only one task can execute on a given CPU at any given moment; other tasks are waiting. It is therefore important to ensure that CPU is correctly released by the tasks.

Tasks execution time is monitored in order to ensure that each task uses no more than the quota it has been assigned at design step. If a task attempts to use more time, an exception is raised by the kernel (see Figure 6). Notice that this implies that the specified execution time of each activity must be strictly positive, whether it belongs to the applicative or the system layer.

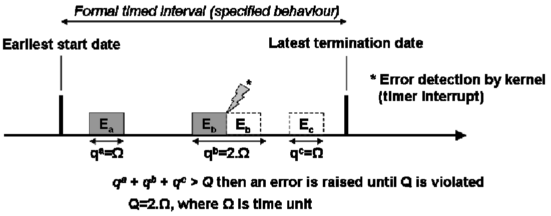

Formal timed interval (specified behaviour)
Earliest start date
Latest termination date
* Error detection by kernel (timer interrupt)
*
E_a
E_b
E_b
E_c
$q^a=\Omega$
$q^b=2.\Omega$
$q^c=\Omega$
$q^a + q^b + q^c > Q$ then an error is raised until Q is violated
$Q=2.\Omega$, where Ω is time unit

Figure 6. Quota monitoring

Each fraction of the execution time consumed by the system layer (communication and monitoring) is attributed to the task for which the processing is performed. Observe, in particular, that a specific time margin, attributed to tasks, is reserved for the monitoring of the quotas by the micro-kernel.

Similarly to deadlines, the monitoring of quotas is implemented by a hardware timer which is set when a task is executed.

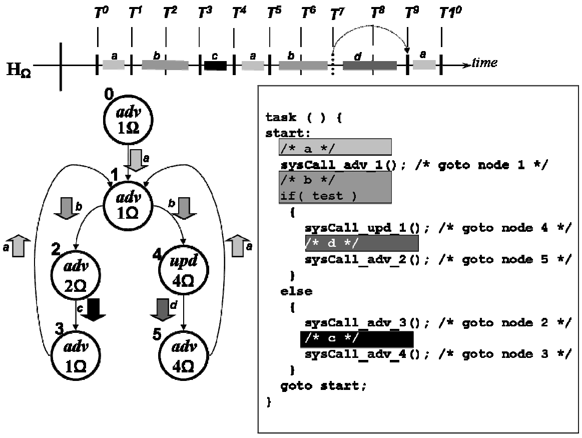
3.1.3. Execution sequence monitoring

The temporal behavior of tasks and handlers is monitored in order to verify that it conforms to the specified one. To avoid misinterpretation by developers, the temporal model is automatically extracted from the application design and stored in a runtime table; this information supplies online verification mechanisms to check correctness of processing sequence and their associated temporal constraints.

The labeled transition system describing a task encompasses its complete temporal behavior and includes all synchronization points that this task can reach. For each task, an extended representation, independent from other tasks, is obtained by unfolding this transition system and computing a temporal congruence on its states. The resulting temporal diagram (illustrated in Figure 7) is an optimized model for behavior used by the system layer to control the execution of a task. This diagram allows one to precisely place each elementary activity in a corresponding temporal window.

The temporal diagram in Figure 7 illustrates a cyclic task for which temporal behavior alternates according to a conditional test. This diagram has two types of nodes: *adv* and *upd*, corresponding respectively setting and updating the temporal window of the current activity.

A system call function sysCall_xxx_n(), generated separately for each transition (cf. the generated code in the right-hand side of Figure 7), passes the index of the target node of the temporal diagram to the system layer. The diagram itself is stored in read-only memory and cannot be modified. The system layer verifies whether the task can move from the current node to the target one while respecting the associated constraints thereby controlling the conformity of the execution to the design specifications.



```
task ( ) {
start:
  /* a */
  sysCall_adv_1(); /* goto node 1 */
  /* b */
  if( test )
   {
     sysCall_upd_1(); /* goto node 4 */
     /* d */
     sysCall_adv_2(); /* goto node 5 */
   }
  else
   {
     sysCall_adv_3(); /* goto node 2 */
     /* c */
     sysCall_adv_4(); /* goto node 3 */
   }
  goto start;
}
```

Figure 7. Temporal behavior monitoring

In the ET domain, all useful interrupt sources are explicitly identified; other unused interrupts are disabled. Therefore, apart from timer interrupts used for time management, each interrupt must start an appropriated processing, encapsulated in a known and controlled temporal window: the interrupt must be expected. For this, minimum time interval (the shortest acceptable in terms of CPU load) between a given number of occurrences of the same interrupt must be specified at the design stage. Then, a system automaton ensures that their executions are performed according to occurrence's order of the respective events. Based on this information, support tools generate appropriate code to perform occurrence monitoring through dedicated hardware timer channel.

3.1.4. Timer implementation

Timer implementation in the Time-Triggered domain of PharOS is straightforward. Indeed, one timer is sufficient for monitoring both deadlines and quotas of task activities. Hence, a hardware timer can always be used. This timer is always enabled and reset each time the execution leaves the microkernel.

In the Event-Triggered domain, each handler has a specified quota used for monitoring. Hence a separate timer might be necessary for each interrupt type in order to optimize their execution time. However, timer resources may be insufficient or microcontroller performance too tight. Different implementations are possible. In the optimal case, a dedicated or shared timer can be set while ensuring time countdown coherency policy: time must only be deducted when the handler is effectively executed. In some other cases, a software mechanism must be implemented by checking whether, when handler is executed, its previous execution was already terminated or not.

3.2 Spatial partitioning

3.2.1. Code and data partitioning

An error occurring during the execution of a task should impact neither the other tasks, nor the kernel. This can be ensured by the hardware memory protection, provided clear and explicit segmentation of the executable code: text and data sections of object files must be identified in order to clearly separate variable and constant data, and the application code.

In PharOS, data naming rules are used in order to assist this identification. In addition, specific information for the backend compiler can be included in the source code to specify membership of sections. Sections can then be aggregated into uniform ones depending on the execution context:

- produced data and buffers,
- consumed data and buffers,
- functions.

These sections are then placed in the memory according to the following criteria:

- Access performance of the memory support (code or data is frequently used).
- Shared memory between cores when multi-core architecture is targeted,
- Implicit protection provided by the memory support such as flash memory. (indeed, such memory requires use of specific driver to distort it),
- Similarity of the required memory access rights for different elements.

All the above operations depend on the memory protection abilities provided by the hardware unit.

3.2.2. Execution support protection

Hardware execution mode mechanism is used to prevent applications from interfering with the system layer. Such mechanism authorizes execution of specific instructions and access to memory regions (through configuration of memory protection unit descriptors) only according to specific microcontroller mode. At least two execution modes must be distinguished by the hardware:

- *Supervisor mode*—to restrict access to the internal registers (e.g. stack pointer store/load instructions, memory management unit instructions) and peripheral device registers (INT, MPU, timers, etc.) used for the execution.
- *User mode*—for the application software.

System access is only possible through a unique interface: by the application, based on trap calling (automatically generated at compilation stage), and by the micro-kernel via timer interrupt to manage scheduling as illustrated in Figure 8 (SL means System layer and µN means Micro-Kernel in the following figures).

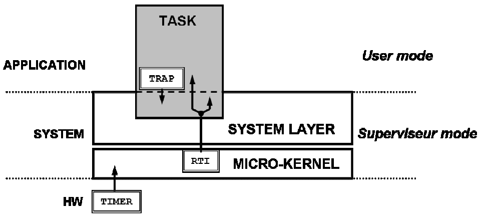


Figure 8. Execution modes

3.2.3. Memory access protection

A strict memory protection policy must be implemented in order to protect data and code sections against spurious read, write or/and execute attempts. This protection is essential to ensure predictable and reproducible behavior even for the degraded modes. This protection is defined with the memory protection unit (MPU), where rights associated with execution modes provided by the microprocessor are applied dynamically and swapped upon context switch to allow appropriate memory access for the enabled context, as illustrated in Figure 9 (the current task is shown on the left-hand side for each context).

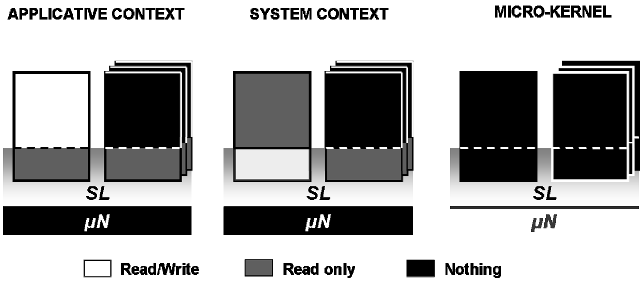


Figure 9. Memory rights

Write access protection

Write access protection is the most important one. Indeed, task or system state alteration can only be confirmed by writing of data in memory or register. Applicative right accesses must therefore restrict at the necessary and sufficient need whatever protection abilities level.

Read and execution access protection

A priori, write access protection is sufficient for some safety-related purposes. However, read and execute accesses should not be restricted to increase the execution determinism, particularly in case of failure. Task should not be able to access unrelated memory area. Hence, for optimal protection, memory area consulted or executed by a context must be clearly identified. However, depending on hardware capacity, degraded protection can be used, where read access rights are extended: first for constant data and code, then for variable data. To be coherent with hardware capabilities, memory protection is highly correlated with the code and data segmentation.

3.3. Failure management

Whenever an error is detected and confined, a specific failure management policy must be applied. Based on the presented protection mechanisms, PharOS ensures that errors are detected and confined in their source execution context. The system then remains stable and continues its execution in a degraded mode. Degraded behavior is specified at design stage with the same requirements as in the nominal behavior. PharOS allows performing specific recovery procedures.

3.3.1. Degraded behavior

When a faulty task is shut down upon detection and confinement of an error, other tasks can still be affected if their behaviors depend on the data produced by this task. In order to control such error propagation, PharOS provides two kinds of mechanisms to specify degraded behavior:

- *Task grouping*—confinement area is defined over a set of tasks. Thus, if one task fails, all tasks of the group are shut down. This is necessary when several tasks compose the same application or specification of the degraded behavior for correct tasks is impossible.
- *Task state broadcasting*—for tasks producing real-time data flows, data extrapolation can be realized by the system to ensure data coherency. Consumer tasks are therefore notified of the state of the producer (faulty or correct) and the pertinence of data. To ensure real-time coherence of the data-state pair, the state information is broadcasted to consumers through the real-time data flow mechanisms.

3.3.2. Recovery policy

PharOS provides a recovery mechanism that is performed at the faulty task (or group) level without affecting the functions of the other tasks. Moreover, it is realized with temporal constraints to ensure restarting within a controlled delay.

This recovery mechanism is provided by the kernel. The first step consists in reinitializing the context of the task (stacks, registers, etc.) and communication mechanisms without impacting actual consumed values. The second step consists in re-integrating tasks in scheduling lists. For this, a restart instant is computed from a delay specified over a congruent cycle, i.e. in a synchronized way consistent with the offline computed CPU load.

When a task is restarted, relevant information are provided to the application level to enable choosing between the nominal behavior (executed at the system initialization and start) or a specific (degraded) one, as all behaviors, including those after a retart, are described in the design. In particular, this allows correct re-initialization of local variables.

3.3.3. Logging mechanisms

Safe and controlled resource sharing allows reporting the error source for later diagnostic operation.[1] For TT tasks failures, the nature of the error as well as the corresponding location in code (given by the temporal execution diagram) can be reported; for ET tasks—information about the interrupt vector. The global determinism ensured by PharOS allows enhanced error identification, and better verification and validation support during software development.

4. Dual-core support

The dual-core hardware architectures used for microcontrollers dedicated to automotive ECU provides intrinsic execution parallelism.

4.1. PharOS execution model

In order to optimize the allocation of the CPU time to application execution, PharOS provides an execution model based on the following two principles:

- Task execution must be interrupted only when necessary.
- Ready tasks must be executed whenever possible.

The microkernel is split between cores[2] in order to perform system execution in parallel with application execution:

- The *control core* manages time evolution and scheduling (lists of ready and pending TT tasks).
- The *computing core* executes TT tasks, with appropriate task context switching management, according to the list of ready tasks.

The control core is also in charge of the execution of handlers, thus avoiding any perturbation of the execution of TT tasks. Therefore, CPU time is preserved for the execution of TT tasks; no time is consumed for task's context switching and system process execution.

Figure 10 illustrates the PharOS dual-core execution principle. The computing core runs TT tasks from the list of ready tasks (contexts), which is updated by the control core. Assume that two agents AG1 and AG2 are to be executed in the temporal window [T_0, T_1[. When AG1 terminates its elementary action the computing core switches execution to AG2. Apart from changing the task contexts (by the micro-kernel layer), this operation involves operations such as, for example, updating the list of ready tasks. In the single-core context, all these operations are performed by the same core, which implies, in particular, a number of additional context switches. The separation of control from computing reduces the number of context switches and control computations on the computing core, saving time for execution of application tasks.

To summarize, the control core relieves the computing one from managing the time evolution and TT scheduling, as well as processing the interrupts from the I/O operations. Moreover no context switching is required between agents and handlers.

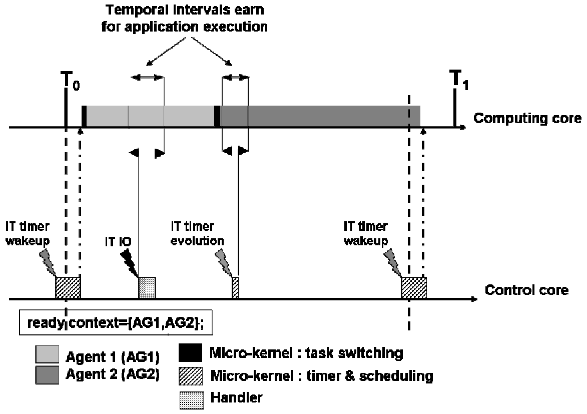


Figure 10. Dual-core execution principles

4.2. Programming model

The dual-core architecture underlying the system execution is completely transparent for developers, thus simplifying the design phase. In particular, no API is needed for specifying processing on the control core or mapping of tasks. Developers describe only nature of tasks, that is whether a task is Time-Triggered or Event-Triggered, and provide the associated temporal and functional constraints. The mapping is implicit and automatically determined from the design description.

5. Case study and results

An ECU developed for PSA Peugeot Citroen with an OSEK OS on the S12XS (Freescale microcontroller, single HC12 core) has been redeveloped partially, using PharOS on the S12XE (double core, pin to pin compatible with S12XS).

The case study is a subset of an automotive application that includes: some functions for system outputs command, CAN communication bus, sensors signal measurements and voltage control. The whole application is composed of 7 TT tasks (including CAN LS driver) and 2 handlers. TT tasks have different dynamic behaviors and scales from hundreds of microseconds to several seconds, ET ones from a few microseconds to several milliseconds. Handlers indeed measure the duty

[1] The failure reports are stored in an EEPROM.

[2] These microkernel features for multi-cores are integrated into a patent.

cycle of a 400 Hz PWM signal and catch the signal state after a precise exclusion time with a filtering policy. CAN driver is totally integrated in the TT domain, even though the CAN frames are received in an event way

The sample application that is described below covers some representative functions extracted from this automotive industrial case study.

5.1. Sample application

This application includes some functions of system outputs commands, communication through a CAN LS bus and sensors signal measurements.

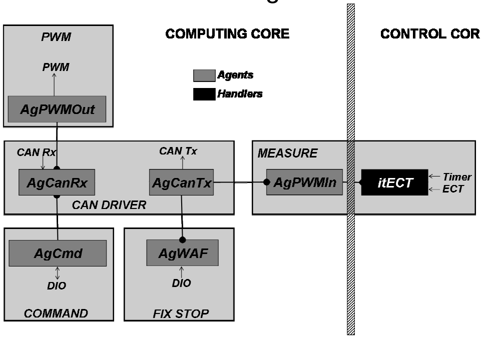


Figure 11. Sample application

The application consists of 6 TT tasks (including CAN LS driver) and 1 ET handler (see Figure 11 and Figure 15):

- AgCanRx: Reception management of CAN LowSpeed (125 kbits/s) driver. Its design is implemented to ensure by construction that there is no loss of received messages due to hardware buffer overflows and no work overload due to CAN bursts. AgCanRx has a period of 1 ms followed by a 500 µs deadline in order to ensure a maximum jitter of message extraction processing under 1.5 ms.
- AgPwmOut: Production of the duty cycle according to an external value. This process is performed only if beams command is set to on (15 ms period); otherwise AgPwmOut periodically checks for a presence of a new command (5 ms period for improved reactivity).
- AgCmd: Management of the output signals for the light beams and the wiper according to the command received from CAN.
- AgCanTx: Transmission management of CAN LowSpeed driver. Its design is implemented to ensure by construction that message transmission is precise and guaranteed. AgCanTx has a period of 5 ms to satisfy transmission constraints of the new values produced by AgPwmIn and AgWAF.
- AgWAF: Retrieval of the fix stop position of the wiper and its communication over the CAN network via AgCanTx
- itECT: ET task for measurement of a duty cycle of a 400 Hz PWM signal. Time scale (from some microseconds to 2.5 ms) required for this signal processing is too small (compared to the provided frequency range) to be treated in the TT domain. Thus, itECT extracts timers registers value (e.g. instants) noted by hardware on occurrence of rising and falling edges of the measured PWM signal (400 MHz). Then, it communicates these acquisitions to AgPwmIn by blocks of 5. System is then configured to tolerate 2 interrupts by period of 2.5 ms.
- AgPwmIn: Computation of the duty cycle based on the instants provided by itECT and communication of the result over the CAN network via AgCanTx.

TT tasks have different dynamic behaviors, according to internal or external state/command, and different time scales from 500 microseconds to several milliseconds.

5.2. Error injection results

To stress the error detection mechanisms and the recovery management, 3 kinds of errors were injected at AgWAF level:

- A wrong memory access (see Figure 12), to attempt distortion of a critical function such as light beams command (AgCanRx) by setting the beams to off when they must be set on.

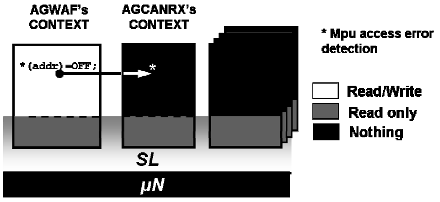


Figure 12. Memory access error injection

- An infinite loop (see Figure 13), for example, to simulate hardware failure which diverts the exit condition of a test.

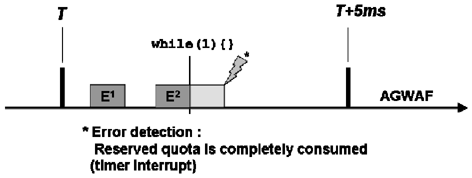


Figure 13. Quota error injection

- And interferences on interrupt occurrences (see Figure 14), to simulate interrupt burst or bad plug connector.

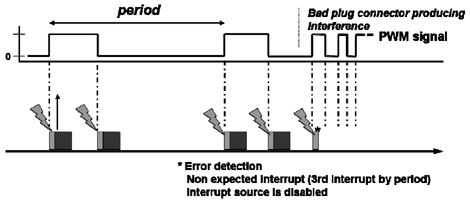

period
Bad plug connector producing interference
PWM signal
0
*
* Error detection
Non expected interrupt (3rd interrupt by period)
Interrupt source is disabled

Figure 14. Interference error injection

In actual automotive controllers, such errors can lead to undesirable failures and, in the worst case, a loss of all functions as a result of restarting the controller. With PharOS, each error was detected and confined at the level of the task responsible for the failure (see Figure 15); the sources and nature of errors were logged in the EEPROM memory for later diagnostics. Subsequently, a recovery procedure was engaged for the faulty task in order to restart it after a specified delay of 2 seconds (see Figure 15). No other functions were disturbed during this recovery process.

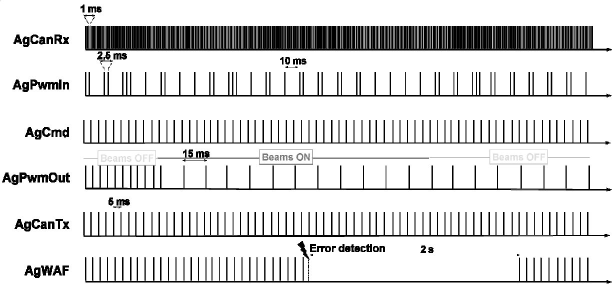


Figure 15. Application temporal diagram

The development of the industrial case has established the feasibility of PharOS in an automotive real case. So, ongoing work is now focused on achieving integration between PharOS and AUTOSAR. This paves the way for a modular software architecture where components reuse and modification is easier among vehicles: software component interactions and composition can be controlled to improve cost-efficiency without making any compromise with respect to quality and safety.

6. Conclusions and prospects

PharOS is a solution for next-generation of automotive controllers that enables implementation of the upcoming automotive safety standard (ISO 26262). It provides a new approach for the design and implementation of embedded real-time systems fulfilling also other important automotive requirements: generic software architecture, adaptability to hardware features such as multi-core one, reuse of safety-based mechanisms, software integration from different suppliers. As it is possible to define and control different temporal behaviors and operational modes of applicative and basic software designed with the PharOS kernel and its associated code generation tool-chain, development times and costs can be improved. Moreover, it enables integration of functions with different ASIL level on a same ECU, as the embedded protection mechanisms ensure freedom from interference between them and so paves the way to future multi-domain ECUs.

7. References

[1] Leteinturier, P., “Multi-Core Processors: Diving the Evolution of Automotive Electronics Architecture”, EETimes, Embedded.com online journal, 2007

[2] David, V., Delcoigne, J., Leret, E., Ourghanlian, A., Hilsenkopf, Ph., Paris, Ph., “Safety properties ensured by the OASIS model for safety critical real time systems”, IFIP-SAFECOMP’98 Conf., Heidelberg, Germany, 1998.

[3] Halang, W. A., Gumzej, R., Colnaric, M. and Druzovec, M. (2000), "Measuring the Performance of Real-Time Systems", The International Journal of Time-Critical Computing Systems 18: 59-68

[4] Rushby, J., “Partitioning in Avionics Architectures: Requirements”, Technical Report, Computer Science Laboratory, SRI International, 1998.

[5] Kopetz, H., “The time-triggered approach to real-time system design”, Predictability Dependable Computing Systems, Springer-Verlag, pp. 53-78, 1995.

[6] Aussaguès, C., David, V., “Guaranteeing timeliness in safety critical real-time systems”, 15th IFAC Workshop on Distributed Computer Control Systems, Como, Italy, 1998.