SAE TECHNICAL
PAPER SERIES

960623

ERCOS: An Operating System for Automotive Applications

S. Poledna, Th. Mocken, and J. Schiemann
Robert Bosch GmbH

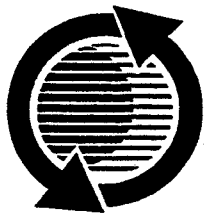
Th. Beck
ETAS GmbH & Co. KG

Reprinted from: Design Innovations in Engine Management and Driveline Controls (SP-1153)

SAE The Engineering Society For Advancing Mobility Land Sea Air and Space®
INTERNATIONAL

International Congress & Exposition
Detroit, Michigan
February 26-29, 1996

400 Commonwealth Drive, Warrendale, PA 15096-0001 U.S.A. Tel: (412)776-4841 Fax:(412)776-5760



ISSN 0148-7191

Printed in USA 96-0049

960623

ERCOS: An Operating System for Automotive Applications

S. Poledna, Th. Mocken, and J. Schiemann
Robert Bosch GmbH

Th. Beck
ETAS GmbH & Co. KG

ABSTRACT

This paper describes the concept of the operating system ERCOS (Embedded Real-time Control Operating System). ERCOS has been specially designed to meet the functionality and performance requirements in the area of automotive applications.

The ever increasing functional requirements for modern electronic control units are introducing considerable complexity in the area of software development. It is well known that real-time operating systems provide powerful means to handle complex functions under real-time constraints. Past experience, however, has shown that the efficiency and flexibility of operating systems was very often inadequate for automotive applications.

To overcome these insufficiencies the operating system ERCOS has been designed with dedicated support for automotive requirements. This has been achieved by supplementing the run-time part of the operating system by powerful off-line tools. The off-line tools support the construction of reusable and modular software in real-time applications by virtue of an object-based model and a strict separation of system-independent functional design and the run-time configuration. Furthermore, they allow the optimization of calls to the operating system to achieve very high efficiency.

The major operating system concepts for scheduling, interprocess communication, guarantee of mutually exclusive access, timer handling and fault-tolerance as implemented in ERCOS are presented.

Bosch will use ERCOS as a standard operating system platform for its automotive products. The operating system is compatible with the OSEK specification [OSE95] and has been passed to an independent software house[1] to make it available to third parties.

1. INTRODUCTION

Over the last decade functionality and complexity of automotive electronics has experienced a dramatic increase. Examples of such systems are electronic engine management systems, anti-lock braking systems, gear box control and others. Supported by the ever growing computing power of microcontrollers this development went alongside with a rapidly increasing size and complexity of the software. In the past the achievable level of functionality was predominantly determined by hardware and the performance of microcontrollers. Currently the limiting factor is shifting more and more from hardware performance to the software development process. Especially in the case of dependable systems, software is the limiting factor for the achievable level of functionality.

It is well known that operating systems support the development of complex software systems. This experience, drawn from larger computer systems, also applies to automotive electronics. There are, however, special requirements for operating systems in the area of automotive electronics. Firstly, the operating system should support handling of hard real-time requirements which are dictated by the environment, e.g., the injection timing of an engine. And secondly, since cost is of utmost importance very high efficiency has to be achieved. Especially the requirement for high efficiency has been in conflict with the performance of general purpose real-time operating systems. This was the reason why commercial operating systems, e.g., [Rea86, ISI93] did not gain acceptance in this field of application. The operating system ERCOS has therefore been specially designed to meet the requirements of automotive electronics. Bosch will use ERCOS as a standard operating system for its automotive products.

The remainder of this article is organized as follows. Section two defines requirements and goals for automotive electronics software. Section three discusses problems of object-oriented software construction in the presence of hard real-time requirements and multi-tasking. The ERCOS object model is introduced. Section four to eight present the major functions of the operating system and its concepts. These are the scheduling strategy, the message concept for interprocess communication, mutually exclusive access to critical resources, timer functions, fault-tolerance features and exception handling. Along with this presentation pos-

[1] ETAS GmbH & Co.KG, Markgröningerstraße 45, D-71701 Schwieberdingen, Germany. FAX +49 711811 3950

sible alternate solutions and the rationale behind the solution implemented in ERCOS are presented. Finally, section nine concludes this article.

2. REQUIREMENTS AND GOALS

The development of ERCOS was influenced by a set of specific goals and requirements. Firstly, it was the goal to create an unified operating system platform for automotive electronic products at Bosch. Further requirements for the operating system were derived from general objectives which focus on software quality and the software development process. Consequently, the catalogue of goals and requirements was not first and foremost targeted at operating system functionality itself but at the development of automotive electronics software and at the development process. An overview of these objectives and requirements is given in the following:

Reusability:
It should be possible to reuse software in different control systems and in different projects. This requires that the functional implementation is independent of timing aspects and global system properties such as priority or scheduling strategy.

Modularity:
Software should be structured and partitioned according to the paradigm of object-orientation to support a modular development and test process.

Efficiency:
Resources of the microcontroller such as RAM, ROM, CPU and peripherals like timers, ports and analog to digital converters have to be utilized efficiently. Since automotive applications are very cost sensitive, efficiency is of utmost importance.

Maintainability and extendibility:
Modifications and extensions to the existing software should be easily possible. These changes should not cross interface boundaries and affect other functions except in the case of interface changes.[2]

Real-time support:
The software has to support real-time requirements. It is therefore necessary that the system responds within a guaranteed latency period to requests. There is a very broad spectrum of timing requirements ranging from 100 milliseconds down to only a few microseconds.

These goals and requirements aim at the management of the steadily increasing software complexity, at shortening development times and improving software quality. Experience shows that rigorous software engineering methods and a dedicated tool support should be applied to reach these goals. As a key principle of modern software engineering, object-orientation addresses all these goals except for real-time support and efficiency. This principle should therefore be considered for the software development process in this field. However, there are problems when applied to multi-tasking real-time systems. These problems are discussed in the next section.

[2] Note that this requirement can only be fulfilled in the domain of values and not in the domain of time since the timing usually changes in a single processor system if a function is changed.

3. THE ERCOS OBJECT MODEL

During the last decade the paradigm of object-orientation [Mey88] has found broad acceptance in the field of non real-time software applications. Its major aims are to improve flexibility, reliability and reuse of software. While these goals are important for real-time software too, classical object-oriented programming languages such as C++ [Str91] are inadequate for hard real-time requirements and multi-tasking. Besides features such as dynamic binding and multiple inheritance the basic problem with object-orientation and multi-tasking real-time systems lies in the object implementation. Object-oriented programming languages do not support concurrent programming and—naively applied—lead to considerable implementation overhead. Thus a suitable adaptation of object-oriented concepts and a dedicated support by operating system mechanisms are needed to draw advantage from object-oriented design within hard real-time systems when efficiency is critical.

An object abstracts an entity which has an internal state (that is invisible from the outside), a well-defined interface and a characteristic functionality. In the context of automotive electronics such an object may be for example a fuel injector or a lookup table. The important aspect of an object is the separation between its implementation—which is encapsulated in the object—and its interface. Typically, an object interface consists of methods (functions) and attributes (variables). Object interfaces with variables, however, cannot be used for real-time systems with multi-tasking because data inconsistency may arise.

3.1. THE PROBLEM OF DATA INCONSISTENCY

Real-time systems typically support preemptive scheduling to guarantee short latency periods. This may lead to cases where the execution of some low priority process is preempted by a higher priority process. Under the assumption that the preempted process reads an object attribute and that the preempting process writes the same object attribute data inconsistency may arise if the preemption occurs between two consecutive read operations, see Figure 1.

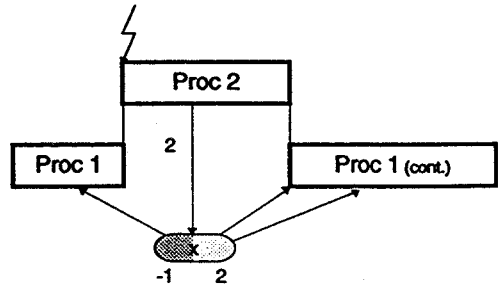


Figure 1: Data inconsistency through object attributes implemented by variables

After preemption through *Proc* 2, *Proc* 1 reads the object attribute x a second and a third time and gets a different result ($x = 2$) than before the preemption ($x = -1$). This example shows that the data accessed by *Proc* 1 becomes inconsistent. As a consequence, it may happen that *Proc* 1 fails. For example let *Proc* 1 implement a function which tests the sign of x and then takes two different execution paths where x is used again. If *Proc* 1 is not preempted the implementation will work correctly. But if the timing of processes changes slightly or if *Proc* 1 is reused in a new application it may easily happen that it gets preempted and does no longer function correctly. Consider the case where *Proc* 1 calculates the absolute value of x by the algorithm:

```
if(x<0)
  {y= -x;}
else
  {y=x;}
```

Since the sign of x may change between the comparison and the assignment operation, *Proc* 1 may fail.

Correct function therefore depends on the timing and the sequence of preemptions in a certain system, as this example shows. This, however, conflicts with the basic requirements for software reuse, modularity, maintainability and extendibility. It also conflicts with the goals of object-orientation where objects should be encapsulated and their correct function should be independent of the environment. To avoid software failures which are timing and system configuration dependent data consistency has to be guaranteed.

Data consistency:
For the duration between start and termination of a process P_1 it has to be guaranteed that all data locations which are accessed by P_1 may change their value if and only if they are changed by P_1.

3.2. ERCOS OBJECT MODEL

To guarantee data consistency ERCOS provides a message communication mechanism (c.f. chapter 5) instead of variables. This mechanism separates the memory areas of different processes and provides functions to exchange information between processes. The interfaces of ERCOS objects therefore consist of functions (methods) and messages (instead of attributes realized by variables).

The basic object classes that are supported by ERCOS are processes, functions, messages and resources. Out of the basic classes only processes are active. Processes change their state autonomously because they are activated exclusively by the operating system. They provide methods for initialization and activation. A function is a passive object which can be called. Messages are basic objects for communication between processes. They provide the methods send and receive. To model resources which can only be accessed exclusively, ERCOS provides resource objects with the methods get and release. Figure 2 shows an example of basic ERCOS objects and their relations. The objects considered here are basic objects at the lowest hierarchy level. ERCOS provides the possibility to construct complex objects with well defined interfaces out of basic objects.

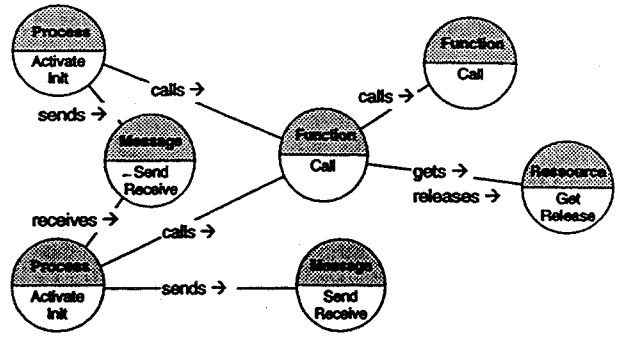


Figure 2: Example of basic objects and their relations

These complex objects are called *subsystems*. Each subsystem defines which objects constitute the interface and which are hidden inside. Figure 3 shows an example of a subsystem which provides a function and a message as an interface. Additionally, to implement the required functionality there are two processes, two functions and one resource which are hidden inside the subsystem.

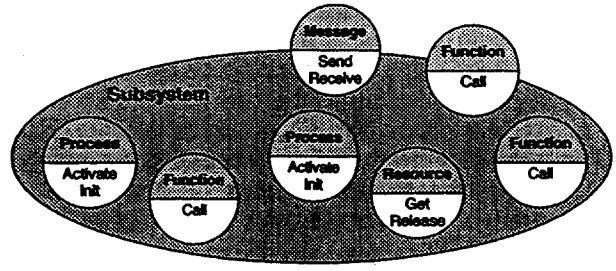


Figure 3: Example of a complex object (subsystem)

Within a subsystem there is no restriction to sequential activations. Subsystems may contain an arbitrary number of processes which can be executed in parallel by means of multi-tasking. Additionally, it is possible to access subsystem interfaces in parallel.

The ERCOS object model is therefore well suited for real-time systems with multi-tasking capabilities. It supports object-based software construction, reuse of software, modularity and extendibility. The ERCOS object model is supported by a set of comprehensive tools which have been developed to allow easy construction of object-based software while providing interface checks for consistency and optimization techniques for the runtime management of objects.

4. SCHEDULING

Scheduling is one of the core functions of a real-time operating system. The scheduler has to decide which process should be started among a set of ready processes [CSR87]. This decision strategy, called scheduling algorithm, is very important since it has influence on the real-time capabilities and efficiency of the system. To achieve the strict requirements for efficiency and real-time performance ERCOS employs a combination of static and dynamic scheduling

together with a mixed preemptive/cooperative scheduling strategy.

4.1. STATIC AND DYNAMIC SCHEDULING

The most fundamental distinction with scheduling is whether it is static or dynamic [SSN95]. With static scheduling, the scheduling algorithm has complete knowledge of all tasks and their constraints. Usual constraints are computation time, deadline, future release times, precedence relations and mutual exclusion. Since all the process constraints are known before the system starts, it is possible to determine the execution order of processes off-line. If such an off-line schedule exists, it is sufficient at runtime to start the processes at the predetermined points in time with a predetermined order. A dynamic scheduling algorithm on the other hand has only knowledge of the ready processes but it has no knowledge of future activation times. Since new processes may become ready spontaneously, the scheduler has to decide at runtime which process has to be selected among the ready processes.

The advantage of dynamic scheduling over static scheduling is its flexibility to react on external events. Especially, the effectiveness of static scheduling decreases with a decreasing latency period [Pol95b]. Disadvantages of dynamic scheduling are higher computational requirements and the memory demand for the management of processes. By supporting static as well as dynamic scheduling, ERCOS allows the implementation of combined strategies which are optimal with respect to the application requirements for response time and memory demand.

4.2. SCHEDULE-SEQUENCES

Object-oriented system construction and modern software engineering methodologies which are targeted at software reuse result in a large number of fine grained processes. Analysis and experience shows that simple sequential precedence relations exists between many of these processes. ERCOS takes advantage of this fact and implements tasks as *schedule-sequences* which represents these precedence relations.

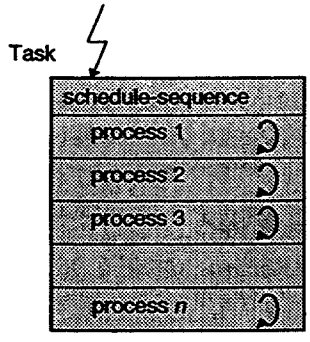


Figure 4: Schedule-sequence

A schedule-sequence is defined as the result of static scheduling. It contains a sequence of processes to be executed in the specified order and at a given priority level upon occurrence of a certain activation event. Dynamic task scheduling (or multi-tasking) only concerns schedule-sequences as a whole and not individual processes. Figure 4 depicts the principle concept of a task defined as a schedule-sequence. Within a schedule-sequence the scheduler does not need to take scheduling decisions since the execution order is constructed statically. This reduces the computational requirements at runtime. Additionally, the memory demand is reduced considerably since it is not necessary to manage a large number of processes (> 100) but a much smaller number of schedule-sequences. This methodology typically reduces the number of entities that have to be managed at runtime by a factor 10-20.

4.3. COOPERATIVE AND PREEMPTIVE SCHEDULING

There are two possible strategies when to switch from an executing task to a ready task with higher priority. The first strategy, called *cooperative* scheduling, switches execution at predefined points in the software. These predefined points are the borders between processes within a task. If necessary, it is also possible to implement switching points by operating system calls. Since the scheduler has to wait until the running process is ready and thus cooperates with the application software, the scheduler is said to be *cooperative*, see Figure 5.

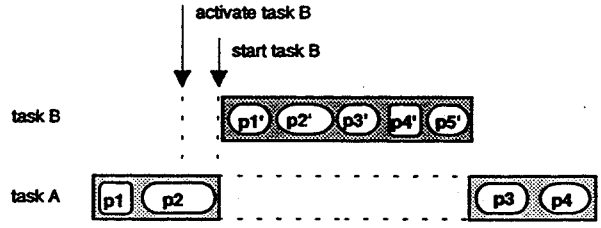


Figure 5: Cooperative task scheduling

With the second strategy, called *preemptive* scheduling, execution can be switched within processes at the boundary of machine instructions (under the assumption that interrupts are not disabled). The scheduler is therefore able to suspend the currently executing process within a task and to start the execution of a task with higher priority, see Figure 6.

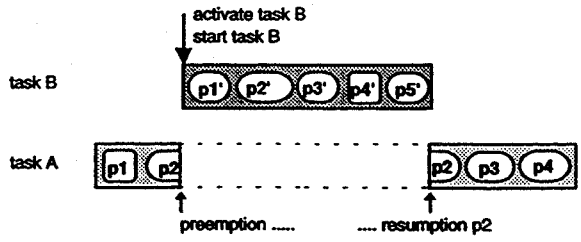


Figure 6: Preemptive task scheduling

The advantage of cooperative scheduling is the efficient utilization of resources. It is guaranteed by design that resources such as stack, registers or messages are accessed exclusively. Additionally, there is no need to save the process context when switching between processes, all processes can execute with the same register bank and with one stack. The disadvantage of cooperative scheduling is its relatively

slow response time that depends on the worst case execution time of processes. For external events (interrupts) and for periodic activities with controlled jitter short response times are necessary. Preemptive scheduling can fulfill the requirement for short response times. The disadvantage of preemptive scheduling are higher memory requirements since it is necessary to save the context of preempted processes and to guarantee data consistency (c.f. chapter 3.1).

ERCOS therefore supports a combination of cooperative and preemptive scheduling. This allows to select the most effective combination for a certain application. Typically, only a small amount of the application has short latency requirements which requires preemptive scheduling while the large remainder can be scheduled cooperatively. The necessary amount of memory resources can thus be minimized while guaranteeing the application specific real-time requirements. This is realized by a hierarchical scheduler concept where the cooperative scheduler is subordinated under the preemptive scheduler. The cooperative scheduler is treated as a single task at the lowest priority level of the preemptive scheduler. Both schedulers use fixed priority assignments as described in [LL73, KRP+93]. Schedulability can therefore be analyzed for the preemptive and the cooperative part according to [Bak91] and [JSM91] respectively. The operating principle of the hierarchical scheduler is shown in Figure 7.

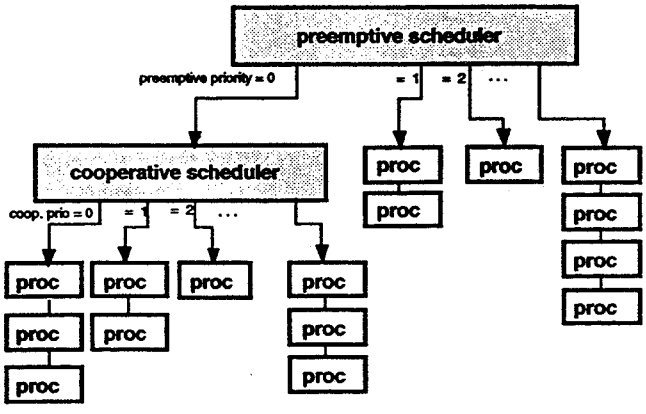


Figure 7: Combined preemptive/cooperative scheduler

The preemptive scheduler provides additional flexibility to handle external events. It is possible to assign an interrupt source to a preemptive priority level. There is no distinction necessary between tasks that are activated by hardware (interrupts) or by software. Thus ERCOS provides a unifying concept for hardware and software activated tasks. It is even possible to have software activated tasks at higher preemptive priority levels than hardware activated tasks.

5. MESSAGES

The ERCOS object model is based on messages for data communication between objects. To achieve data consistency in a real-time system with preemptive scheduling it has to be guaranteed that objects do not share memory. The message mechanism separates the memory space of objects and performs the necessary information exchange.

5.1. EVENT- AND STATE-MESSAGE SEMANTICS

There are two possible types of message semantics which are described in the following.

Event-message semantics:
Event-message semantics [Tan92] is characterized by a consuming receive operation, i.e. if a message is received, it gets consumed such that the next receive operation gets the next message. A message is therefore associated with an event that is processed upon receiving the message. Event-messages are typically implemented by message queues, since it is common that more than one message can be sent before receiving a message. With event-message semantics there is a synchronization between sender and receiver: For each message which has been sent there is exactly one receiver. This 1:1 synchronization relation can be implemented with blocking or non-blocking semantics.

State-message semantics:
The semantics of state-messages [KDK+89] is very similar to that of global variables. If a state-message is received the last value which has been sent is returned. By this the message is not consumed. It is therefore possible to receive a value that has been sent to a state-message more than once. The difference between global variables and state-messages is that global variables can be overwritten at arbitrary points in time whereas the receiver of a state-message gets a message copy that is kept unchanged if no further receive operation is performed which guarantees data consistency. A state-message therefore reflects the last state of some entity which can be read by receiving a message or updated by sending a message. With state messages there is no synchronization relation between sender and receiver. Rather, they can be used for an unsynchronized 1:*n* or *m*:*n* information exchange between processes. State messages are typically implemented by message pools. The sender puts the recent message in the pool while receivers get copies of this message from the pool.

Commercially available real-time operating systems, e.g., [Rea86, ISI93, TSK89] implement event-messages which are not suited for automotive electronics. There are two major problems. Firstly, from a functional point of view most processing activities are carried out periodically. There is a multitude of objects that are activated with different periods or sampling rates. It is therefore inappropriate to require a 1:1 synchronization between sender and receiver of event-messages. And secondly, the efficiency of typical send and receive operation implementations is insufficient. In high-end automotive applications there is a transaction rate of up to 100.000 messages per second which cannot be handled if send and receive operations are consuming between 15 to 100 µs per call. This would result in a theoretical processor load of 150–1000% for communication only.

5.2. ERCOS MESSAGE CONCEPT

ERCOS therefore provides state-message semantics with a highly optimized implementation. Based on the ERCOS

object model, a process receives input data, performs processing actions and sends output data (input-process-output). Send and receive operations are mapped onto message objects. Between processes there is no direct[3] method of information exchange by attributes (or variables). Conceptually, the state-message implementation of ERCOS provides a message copy for each receiver process of a message. During startup of a process, all input messages are copied to the private area for message copies. Having finished, all output messages which are held in a message copy are copied to the global message area. The functioning principle of state-messages is shown in Figure 8.

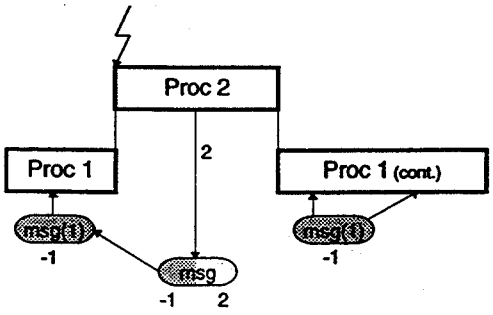


Figure 8: Data consistency through state-messages

Upon start, process *Proc* 1 copies its input message *msg* to the private message copy *msg*(1). All subsequent read operations to the message are done from the private copy. Even though *Proc* 1 gets interrupted by *Proc* 2 which changes the contents of *msg* from -1 to 2 this change does not affect *Proc* 1. It is therefore guaranteed that *Proc* 1 perceives the same contents of its input message during the whole execution. This ensures data consistency. For example the following algorithm will execute correctly regardless of whether *Proc* 1 is interrupted by *Proc* 2 or not (cf. Figure 1).

```
if(x<0)
  {y= -x;}
else
  {y=x;}
```

5.3. OPTIMIZED MESSAGE IMPLEMENTATION

Due to the high rate of up to 100.000 message transactions per second, efficiency of send and receive operations is of utmost importance. Additionally, the memory requirements for message copies should be confined to an absolute minimum since memory is still an important cost factor for automotive electronics. ERCOS therefore provides a powerful set of optimization methods to achieve efficiency. These optimizations are based on static source code analysis. To provide additional information for static analysis a formal description language is used which supplements the C and assembler source code. The implemented optimization strategies are described in the following:

[3] Indirect information exchange between processes is possible by using functions with parameter passing. In this case it is the users responsibility to ensure data consistency.

In-line expansion of send and receive operations:
A general purpose implementation of send and receive operations would have to decide which message copy has to be read or written depending on the actual process context. In the case of static source code analysis send and receive operations can be expanded in-line since the actual process context is known. This reduces the implementation to simple assignment operations, e.g., *msg*(1):= *msg*; for *Proc* 1 in Figure 8. For typical message lengths the execution times of send and receive operations therefore becomes less than 1 μs.

Reduction of message copies to potential cases of data inconsistency conflicts:
Data inconsistency may arise only for two special communication relations between processes. Firstly, if the receiver of a message can get interrupted by the sender, see Figure 8. And secondly, if the sender of a message can get interrupted by the receiver and the sender does not write the message with an atomic operation. It is therefore only necessary to provide message copies and copy operations in these two specific cases. This results in considerable reduction of memory requirements and execution time for message send and receive operations.

Pooling of message send and receive operations:
There is further potential to optimize the execution time for message send and receive operations by pooling. Since the schedule sequence of processes within a task executes strictly sequentially it is possible to pool all receive operations in the header and all send operations in the trailer of a schedule sequence. If for example, a message is received by all four processes in the schedule sequence of Figure 9 then it is sufficient to use only one receive operation and let all the processes share the same message copy. This saves three receive operations and three message copies.

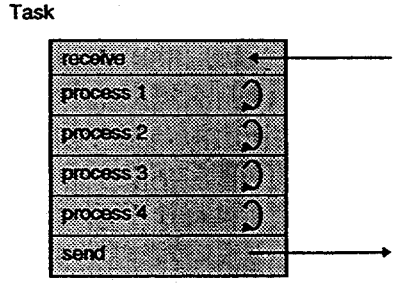


Figure 9: Pooling of message send and receive operations

By using these optimizations the message send and receive rate of 100.000 messages per second could be reduced to 26.000 messages per second in the application mentioned above. Since the execution duration of these operations is less than 1 μs this results in a processor load of 2.6%. This is approximately two orders of magnitude less than other message implementations.

Consequently the ERCOS message implementation can meet the ultra high efficiency requirements for automotive applications while guaranteeing data consistency. This gives strong support for the ERCOS object model which is based

on messages for interprocess communication and which enables software reuse in multi-tasking environments.

6. MUTUALLY EXCLUSIVE ACCESS

The guarantee of mutually exclusive access to resources is an important functional requirement for real-time operating systems. To avoid inconsistencies it is for example necessary to guarantee mutually exclusive access to the error log of a control unit to ensure that an error log update cannot get preempted by a second error log update.

6.1. PROBLEMS WITH SEMAPHORES

Most commonly, this requirement is implemented by semaphores [Tan92]. This solution is not well suited for the requirements of automotive electronics for the following reasons:

(1) Semaphores can block the caller if the requested resource is already accessed. This can lead to deadlock or live-lock conditions which causes severe software failures.

(2) Besides dead- and live-locks semaphores cause the priority inversion problem. This leads to unbounded delays when trying to access a resource. An example of a priority inversion is shown in Figure 10. The low priority process P1 gets exclusive access to the semaphore S. P1 is preempted by the high priority process P3 which tries to access the semaphore S as well. Since the semaphore is already in use by P1, P3 is blocked and has to wait until the semaphore S is released. It is therefore possible for the medium priority process P2 to execute for an arbitrary duration. This phenomenon is called priority inversion since the high priority process P3 in fact has to wait for completion of the medium priority process P2 (even though P2 does not request access to semaphore S).

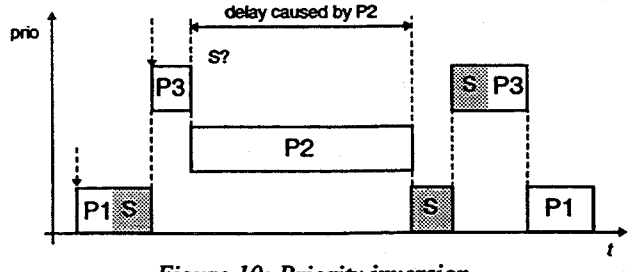


Figure 10: Priority inversion

(3) Since semaphores can block the caller if the requested resource is occupied, the operating system has to provide a context (stack, register, ...) for each process to save its state in the case of blocking. Especially, for object based systems with a high number of processes (>100) this would result in unacceptable resource requirements.

6.2. OPTIMIZED STACK BASED PRIORITY CEILING PROTOCOL

To overcome the problems stated in the previous section, ERCOS supports an optimized variant of the priority inheritance protocol [SRL90] which is called *stack based priority ceiling* protocol. The basic idea of the priority inheritance protocol is to elevate the priority of a process which holds a critical resource when a higher priority process tries to access the same resource. By elevating the priority of the lower priority process to that of the higher priority process it is guaranteed that unbounded priority inversion cannot occur [SRL90]. This protocol, however is still insufficient to guarantee absence of deadlocks and of process blocking.

The priority ceiling protocol [RSL88] is a variant of the priority inheritance protocol where a process inherits the maximum priority of all processes which potentially have access to the critical resource. This inherited process priority is called the priority ceiling of a resource. Application of this protocol guarantees deadlock free execution but process blocking is still possible. The following protocol variant of the priority ceiling protocol guarantees absence of deadlocks and non-blocking execution. Instead of inheriting the ceiling priority only in case of an access conflict the priority ceiling can is inherited immediately upon accessing a resource [Bak91]. This protocol variant is called stack based priority ceiling protocol. It is therefore impossible that a process which occupies a resource is preempted by another process trying to access the same resource, see Figure 11.

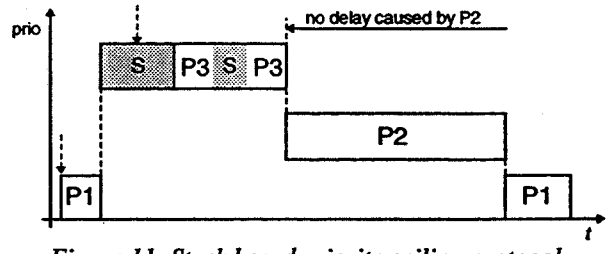


Figure 11: Stack based priority ceiling protocol

ERCOS supports this protocol through off-line tools which are responsible for the static analysis of the priority ceiling for each resource. Additionally, they provide the following optimizations of this protocol. It is detected by static analysis whether a process has the highest priority among all processes potentially accessing a certain resource. Then it is not necessary to change the priority of this process when requesting and releasing the resource. In this case, the off-line tools therefore expand both operating system calls to empty. A second optimization is possible if the duration of the resource access or *critical section* is very short. If this duration is approximately the same as that to perform the priority inheritance and the switch back to the original priority then it is more efficient to disable interrupts during execution of the critical section. Again, this implementation variant is expanded in-line by the off-line tools. The static source code analysis furthermore provides the possibility to detect illegal access to critical resources outside of protected code segments.

7. TIMERS

In the case of automotive applications only a minority of functions is activated by the occurrence of events. The majority of functions is triggered at certain points in time. These time-triggered activations can be subdivided into

activations with fixed repetition periods which are kept unchanged during a whole operating mode and into activations with variable repetition rates. ERCOS provides therefore two types of timer services. Firstly, a *static timer service* for time-triggered task activations with fixed repetition rates and secondly, a *dynamic timer service* for time-triggered activations with varying repetition rates.

Both timer services, however, have the same functioning principle. A timer is characterized by its repetition period *tp* and by its start delay *ts*. The start delay *ts* specifies the delay of the first activation and the repetition period defines the timing of further activations. Upon expiration of a timer a task gets activated, see Figure 12. Both timer services provide a very fine grained resolution in the range of a few micro seconds.

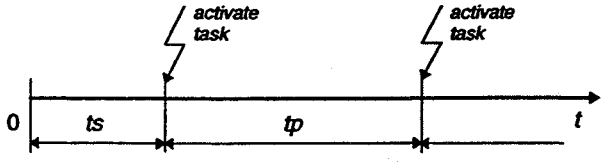


Figure 12: General timer service function

7.1. STATIC TIMER SERVICE

The static timer service supports time-triggered activation with fixed repetition periods. Since the complete timing information for this activation type is known before run-time it is possible to calculate a time schedule for a given set of periodic activations off-line. A corresponding schedule table is generated by the ERCOS off-line tool. It contains a sorted list of all activation times within one overall period. By using this off-line generated schedule table the operating systems run-time effort is considerably decreased. It is not necessary to search the temporarily next timer activation and manage data structures for active timers. Rather the operating system only needs to select the next activation from the pre-sorted schedule table which is very fast. Hence it is possible to support time-triggered activations with very short repetition rates, e.g. two tasks with 1000 μs and 800 μs.

7.2. DYNAMIC TIMER SERVICES

Task activations with varying repetition periods are supported by the dynamic timer service. For example this timer service allows for a single delayed task activation by specifying the start delay and no repetition period. Further possibilities are to change the repetition period of a timer during operation or to start and stop a repetitive timer service depending on the actual operating conditions. To provide the necessary flexibility the dynamic timer service is handled by the operating system completely on-line.

The major advantage of the dynamic timer service is its flexibility which is at the price of higher execution time and memory requirements. On the contrary, the static timer service provides very high efficiency but less flexibility. ERCOS therefore provides a unified concept for timers where it is possible to select the most appropriate service in correspondence to the functional requirements. By supporting static and dynamic timers it is possible to minimize the execution time and memory demand.

8. FAULT TOLERANCE AND EXCEPTION HANDLING

Robustness and fault-tolerance are important properties for real-time systems since they have to respond to state changes in the environment with a guaranteed latency period. Such guarantees, however, can only be given for a specified peak load scenario. The peak load scenario defines the maximum arrival rate and pattern of events that has to be handled by the system. Experience, however, has shown that the assumptions which are made about the peak load scenario are violated often. There are two reasons for this: Firstly, the assessment of the peak load scenario was wrong and the arrival rate of events is higher than assumed. And secondly, faults in the peripherals or in the computer system itself lead to unanticipated *event showers*. The operating system is therefore required to handle the resulting overloads gracefully. ERCOS provides a set of features to handle overload situations and to guarantee timely response. From an operating systems point of view an overload situation is introduced if the number of task activations exceeds the anticipated peek load assumption.

8.1. GUARANTEED MINIMUM INTERARRIVAL PERIOD

For task activations, the peak load scenario can be specified in terms of a minimum interarrival period. Individually for each task this parameter specifies the minimum time interval between two consecutive activations, see Figure 13.

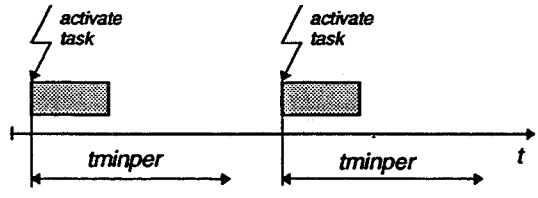


Figure 13: Minimum interarrival period

In the case of sensor faults or a wrongly estimated peek load scenario it is possible that two consecutive task activations are closer together than specified. To prevent system overloads, ERCOS provides a mechanism that rejects task activations which are too early. If tasks are activated by software, the operating system checks the duration since the last activation. It rejects the task activation if the elapsed time is shorter than the minimum interarrival period. In the case of tasks that are activated by hardware interrupts a different strategy has to be used since the activation is not done by an operating system call. To ensure that the minimum interarrival period is not violated, the operating system disables the activating interrupt after the task activation for the duration of the minimum interarrival period. An exact schedulability analysis of this fault-tolerance mechanism is given in [Pol95b]. Figure 14 shows the handling of early interrupts.

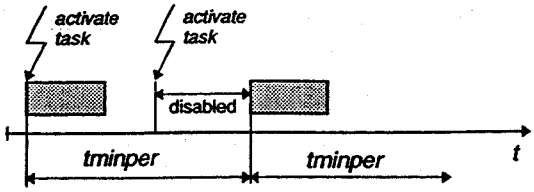


Figure 14: Handling of early interrupts

8.2. BOUNDED NUMBER OF CONCURRENTLY READY TASK INSTANCES

For software activated tasks it is under some circumstances necessary to allow multiple activations of tasks. This results in multiple concurrently ready instances of a task which can be managed by the operating system in FIFO manner. Hence it is possible to implement queuing semantics. Again, a peek load scenario is required to guarantee that task activations are handled correctly by the operating system. ERCOS therefore allows to specify the maximum number of concurrently ready instances of a task. This parameter serves two purposes: Firstly, during the system configuration an analysis tool reads all the task definitions and generates data structures which are sufficient to hold all possible concurrently active task instances. And secondly, the operating system checks upon each task activation whether the maximum number of concurrently active instances will be exceeded. In this case the task activation is rejected. This method guarantees that the operating systems data structures are sufficient to handle the worst case scenario. There is no need for experimental buffer size estimations. Furthermore, the operating system is capable of tolerating software faults at runtime by rejecting excessive task activations.

8.3. DEADLINE MONITORING

Real-time systems are characterized by the fact that they have to respond to events in the environment with a bounded latency period. This latency period, called *deadline*, specifies whether a result is delivered timely, thus being correct or not. ERCOS provides a mechanism to implement deadline checking. This allows the detection of late and incorrect system responses and enables the user to react on behalf of these faults. The deadline mechanism provides functions to start a deadline supervision and to check whether a started deadline has expired. The operating system additionally supervises deadlines by itself. Thus deadline violations are detected even in the case when the application program does not call the function for deadline checking. Furthermore, the operating system provides consistency checks to detect corruption of the internal data structures of the deadline monitoring service.

8.4. EXCEPTION HANDLING

Besides faults in the timing of task activations as mentioned above there is a broad variety of faults which are detected by the operating system at runtime. Among these faults are stack over- and underflows or corruption of data structures. Furthermore, there are possible faults which are detected by the user software. To handle these faults in accordance to the application requirements the operating system provides an exception mechanism [Cri89]. Upon occurrence of a fault an exception is raised either by the operating system or by the user. In accordance to the application requirements it is possible to bind an exception handling routine to an exception. This binding is done statically at system generation time and allows binding of user written exception handlers.

9. CONCLUSION

This paper has presented the concept of the operating system ERCOS which has been specially designed to meet the demands of automotive electronics. Bosch will use ERCOS as a standard operating system platform for its automotive products. Section 2 describes general requirements and objectives for the application software in automotive control units which in turn had influenced the design of the operating system. These goals are reusability, modularity, efficiency, maintainability, extendibility, and real-time support. Modern software engineering practices and especially object-orientation address many of these goals and should therefore be strongly considered for use in the software development process.

Section 3 outlines problems with conventional object implementations in real-time systems and describes the ERCOS object model which overcomes these problems. It is shown that object implementations based on attributes (which are implemented by global variables) are not appropriate to reach the proposed goals of flexibility, reliability and reuse in the context of multi-tasking real-time software. This is due to the inherent parallelism of processes (or functions called by processes) accessing these attributes. Specifically this leads to the problem of data inconsistency. Data inconsistency occurs when a high priority process changes data used by a low priority process during execution of the latter. This problem, which depends on configurational and system aspects, is commonly handled by special solutions which prevent software reuse. To achieve the required goals, it is therefore necessary to guarantee data consistency by a general mechanism. In ERCOS this is provided by message communication. The ERCOS object model is therefore based on processes, functions, messages and resources as basic classes. Functions correspond to methods, messages are replacing attributes while guaranteeing data consistency, processes support the implementation of autonomously active objects, and resources are used to guarantee mutually exclusive access to critical resources. This object model is suitable for parallel execution of processes within multi-tasking real-time environments while supporting the goals as stated in section 2.

The scheduling strategy of ERCOS is described in section 4. To achieve high run-time efficiency, the operating system supports a combination of static and dynamic scheduling. The support for static scheduling reduces the on-line execution time requirements of the scheduler considerably. Additionally, it is possible to use combinations of preemptive and cooperative scheduling to minimize mem-

ory resource requirements. Message objects for interprocess communication are described in section 5. ERCOS implements state-message semantics and provides a set of new optimization techniques to provide very high efficiency with regards to execution time and memory requirements. This optimization concept allows the systematic application of messages and thus guarantees data consistency in general run-time configurations without loss of efficiency. Handling of mutually exclusive access to critical resources is described in section 6. Multi-tasking real-time operating systems typically provide semaphores to provide mutually exclusive access to resources. This mechanism, however, has severe drawbacks since semaphores introduce blocking, they may cause dead- or live-locks and they are the source for unbounded priority inversion. To avoid these problems, ERCOS supports the stack based priority ceiling protocol. This protocol guarantees deadlock free execution, freedom from blocking and bounded resource access delays. Section 7 describes the timer services of ERCOS. Since many functions are activated by time-triggered activation an efficient timer service is necessary. To achieve high efficiency the operating system supports a dynamic and a static timer service. The dynamic timer service is very flexible and allows on-line changes. The static timer service is most efficient but does not allow on-line changes since a statically scheduled task activation table is used. It is therefore possible to select an application specific trade off between flexibility and efficiency.

Applications in the field of automotive electronics often have very demanding safety and reliability requirements. ERCOS therefore provides fault-tolerance and exception handling mechanisms which are described in section 8. Since real-time systems have to respond to relevant events in the environment within a bounded latency it is important to handle system overloads introduced by faulty sensors or software errors. The operating system provides mechanisms to enforce a minimum interrarival period between consecutive activations of a single task. Furthermore, there is a mechanism to limit the number of concurrently active tasks. For the detection of timing faults the operating system provides a deadline monitoring mechanism. Handling of faults is done by the exception mechanism which is supported by the operating system.

This paper has shown that proper selection of operating system mechanisms, static source code analysis and optimization methods allow the construction of a highly efficient operating system for automotive applications without compromising flexibility, reliability, maintainability and support for software reuse. With its support for static scheduling and state-messages the operating system is well suited for extensions to a distributed fault-tolerant real-time system by adding appropriate services [KG94, MP96].

REFERENCES

[Bak91] T.P. Baker. Stack-Based Scheduling of Realtime Processes. The Journal of Real-Time Systems. Nr. 3, 1991, pp. 67–99.

[Cri89] F. Cristian. Exception handling. In *Dependability of Resilient Computers*, T. Anderson (Ed). Blackwell Scientific Publications, Oxford. 1989.

[CSR87] S. Cheng, J. Stankovic, and K. Ramamritham. Scheduling Algorithms for Hard Real-Time Systems—A Brief Survey. In *Tutorial of Hard Real-Time Systems*. J. Stankovic and K. Ramamritham (eds.), 1987, pp. 150–173.

[ISI93] Integrated Systems, Inc. *pSOS System—System Concepts*. Release 2.0, PS2-000-003, PSM2000-MAN, 6. Dec. 1993.

[JSM91] K. Jeffay, D.F. Stanat, and C.U. Martel. On Non-Preemptive Scheduling of Periodic and Sporadic Tasks. In *Proceedings of the Real-Time Systems Symposium*. San Antonio, Texas, 1991, pp. 129–139.

[KDK+89] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, C. Senft and R. Zainlinger. The MARS approach. *IEEE Micro*. Vol. 9, Nr. 1, Feb. 1989, pp. 25–40.

[KG94] H. Kopetz and G. Grünsteidl. TTP—A Protocol for Fault-Tolerant Real-Time Systems. *IEEE Computer*. Vol. 27, No. 1, Jan. 1994, pages 14–23.

[KRP+93] M.H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. Gonzales Härbour. A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Scheduling for Real-Time Systems. Kluwer Academic Publishers. 1993.

[LL73] C.L. Liu and J.W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, Vol. 20, Nr. 1, Jan. 1973, pp. 46–61.

[Mey88] B. Meyer. Object-Oriented Software Construction. Prentice Hall Book Co., Inc. 1988.

[MP96] H.-J. Mathony and S. Poledna. Real-Time Software for In-Vehicle Communication. To appear in SAE International Congress, Michigan, USA. 1996.

[OSE95] OSEK (Open Systems and the Corresponding Interfaces for Automotive Electronics), Operating System. 1995.

[Pol95a] S. Poledna. Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism. Kluwer Academic Publishers. 1995.

[Pol95b] S. Poledna. Tolerating Sensor Timing Faults in Highly Responsive Hard Real-Time Systems. *IEEE Transactions on Computers*. Vol. 44, Nr. 2, Feb. 1995, pp 181–191.

[Rea86] J.F. Ready. VRTX: A Real-Time Operating System for Embedded Microprocessor Applications. *IEEE Micro*. Vol. 4, Nr. 6, Jun. 1986, S. 8–17.

[RSL88] R. Rajkumar, L. Sha, and J. Lehoczky. Real-Time Synchronization Protocols for Multiprocessors. In *Proceedings of the IEEE Real-Time Systems Symposium*. 1988, pp. 259–269.

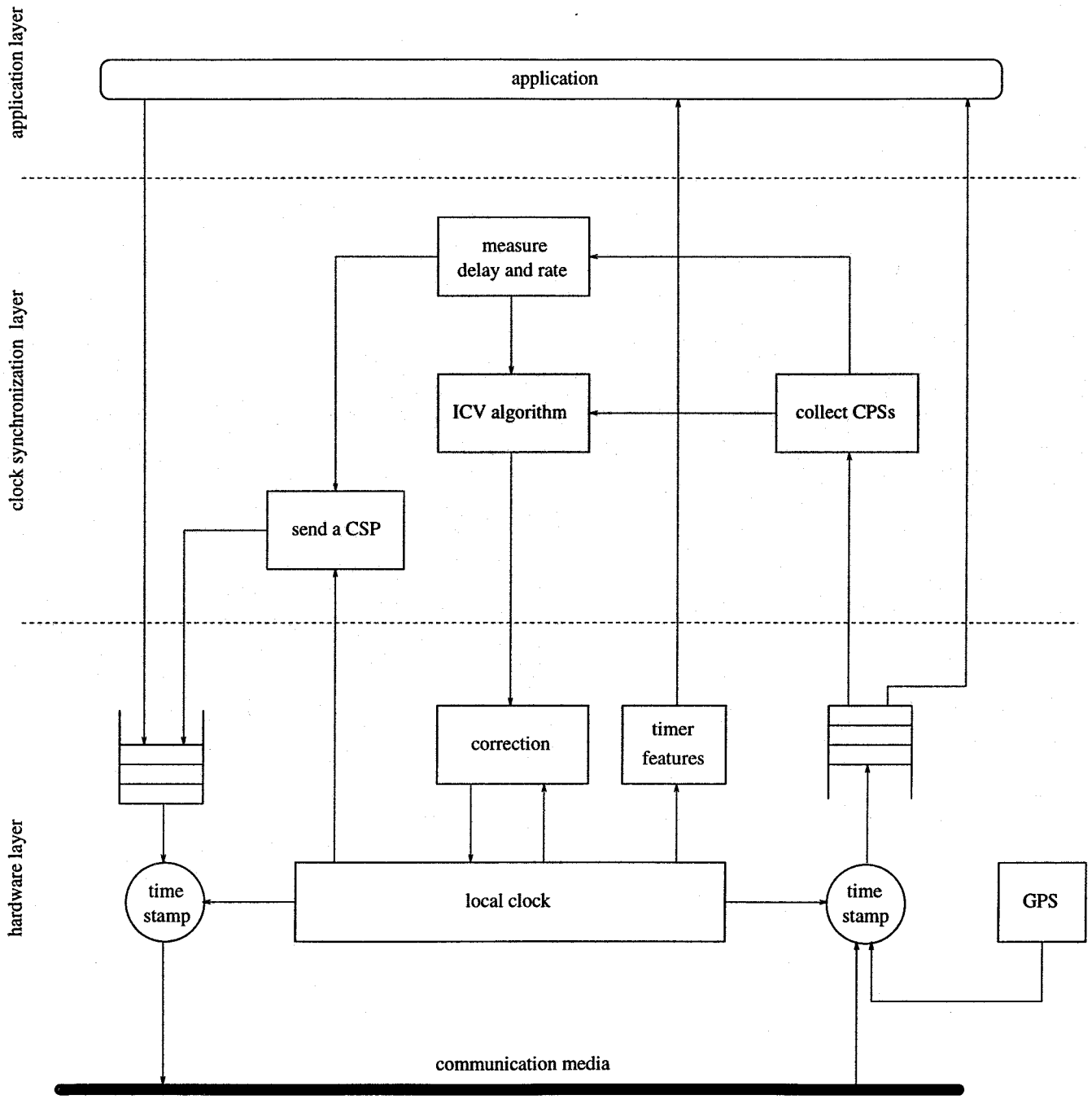
[SRL90] L. Sha, R. Rajkumar, and J. Lehoczky. Priority Inheritance Protocol: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*. Vol. 39, Nr. 9, 1990, pp. 1175–1185.

[SSN95] J. Stankovic, M. Spuri, and M. Di Natale. Implications of Classical Scheduling Results for Real-Time Systems. *IEEE Computer*. Vol. 28, Nr. 6, Jun. 1995, pp. 16–25.

[Str91] B. Stroustrup. The C++ Programming Language. Second Edition. Addison-Wesley. 1991.

[Tan92] A.S. Tanenbaum. Modern Operating Systems. Prentice-Hall. 1992.

[TSK89] H. Takeyama, T. Shimizu, and M. Kobayakawa. Design Concept and Implementation of μITRON Specification for the H8/500 Series. In *Proceedings of the 34th IEEE Computer Society International Conference—COMPCON*. San Francisco, CA, 1989, pp. 48–53.


application layer
clock synchronization layer
hardware layer
application
measure
delay and rate
ICV algorithm
collect CPSs
send a CSP
correction
timer
features
time
stamp
local clock
time
stamp
GPS
communication media