# TiCOS: a Time-Composable OS

## 1. Introduction

The aim of this document is to present TiCOS, a real-time operating system developed within the EU FP7 PROARTIS project [*http://www.proartis-project.eu/, visited on October 2012*]. The main goal of TiCOS is enabling composability in time dimension in Integrated Modular Avionics (IMA) systems where timing composability is a fundamental assumption behind temporal and spatial isolation among software partitions.

TiCOS originated from POK [*http://pok.safety-critical.net/, visited on October 2012*], an open source, real-time embedded kernel designed for safety-critical systems. In this document we report on our attempt in designing TiCOS; most of this activity has been performed re-implementing part of POK with a view to timing composability between OS and application layers. For readers who are not very familiar with this notion, we recall the definition of timing composability and the requirements it poses on OS and applications in the following section. In the rest of the document we focus on the basic principles that guided the implementation on TiCOS, as well as implicit requirements and applicability issues with respect to legacy applications.

### 1.1 What is timing composability? How can be obtained?

Broadly speaking, *timing composability* (or composability in the time dimension) refers to the fact that the execution time of a software component (typically a task, which is the unit of scheduling on a computer system) observed, by analysis or measurement, in isolation, is not affected by the presence of other software components.

Unfortunately, it is usually very difficult to guarantee composability in the time dimension. In general, the timing behaviour of a software module highly depends on the internal state of history-dependent hardware, which in turn is affected by other modules. Timing composability is in fact a system property that originates from the underlying hardware and must be preserved across other layers, including the operating system.

Enabling and preserving time-composability at the operating system layer poses two main requirements on the way an OS or ARINC service should be delivered:

***Zero-disturbance***. In the presence of hardware features that exhibit history-dependent timing behaviour, the execution of an OS service should not have disturbing effects on the application. Some kind of separation is needed to isolate the hardware from the polluting effects of OS or ARINC services. The approach we have implemented consists to simply inhibiting all the history-dependent hardware at once when OS services are executed. This approach, however, comes at the cost of a relevant performance penalty that, though not being the main concern in critical real-time systems, could be still considered unacceptable. Also the execution frequency of a service is relevant with respect to disturbance: services triggered on timer expire (such as, for example, the PowerPC DEC interrupt handler) or an event basis can possibly have even more disturbing effects on the APPLICATION SW level, especially with respect to the soundness of timing analysis. The *deferred preemption* mechanism in combination with the selection of predetermined preemption points [*G. Yao, G. C. Buttazzo, and M. Bertogna. Feasibility analysis under fixed*

*priority scheduling with limited preemptions. Real-Time Systems, 47(3):198–223, 2011*] could offer a reasonable solution for guaranteeing minimal uninterrupted executions while preserving feasibility.

**Steady timing behaviour**: jittery timing behaviour of an OS service complicates its timing composition with the user-level application. Timing variability at the OS layer depends on a combination of multiple interacting factors: (i) the hardware state, as determined by history sensitive hardware features; (ii) the software state, as determined by the contents of its data structures and the algorithms used to access them; and, (iii) the input data. Whereas the first aspect can be treated similarly and contextually with the specular phenomenon of disturbance, the software state instead is actually determined by more or less complex data structures accessed by OS and ARINC services and by the algorithms implemented to access and manipulate them. The latter should thus be re-engineered to exhibit a constant-time – O(1) – and steady timing behaviour, like, for example, constant-time scheduling primitives (e.g., $O$(1) Linux scheduler [*I. Molnar. Goals, Design and Implementation of the new ultra-scalable O(1) scheduler, Jan. 2002. Available on-line at http://casper.berkeley.edu/, visited on April 2012.]).* Besides the software state, the timing behaviour of an OS service may be influenced by the input parameters to the service call (so-called input data dependency). This is the case, for example, of ARINC IO services that read or write data of different size. This form of history dependence is much more difficult to attenuate as the algorithmic behavior (e.g., application logic) cannot be completely removed, unless we do not force an overly pessimistic constant-time behaviour.

An OS layer that meets the above requirements is time-composable in that it can be seamlessly composed with the user-level APPLICATION SW without affecting its timing behaviour. In the following we present the implementation of a set of KERNEL-level services that exhibit a steady timing behaviour and do not disturb the timing behaviour of the user-level code.

## 2. Time-composable kernel and ARINC services

The original POK release provided a set of kernel and ARINC services that were not developed with time composability in mind as the implementation of these services was clearly oriented towards an optimisation of the average-case performance. The approach we implemented in TiCOS aims at injecting timing composability in the OS layer services, starting from the baseline provided by the POK framework. Further details concerning our approach can be found in the paper [*A. Baldovin, E. Mezzetti, and T. Vardanega, A Time-composable Operating System, 12th International Workshop on Worst-Case Execution-Time Analysis, 2012*].

In the next sections we summarize the basic principles that guided the redesign and implementation of TiCOS, as well as implicit requirements and applicability issues with respect to legacy applications.

We start our description with the basic kernel design choices on time management and system scheduling and then proceed with considerations on some ARINC services we re-implemented. In doing so, we refer to ARINC-specific concepts such as processes, partitions, scheduling slots, etc., whose detailed description is out of the scope of this document: the interested reader can referred to [*APEX Working Group. Draft 3 of Supplement 1 to ARINC Specification 653: Avionics Application Software Standard Interface. 2003*.].

## 2.1. Base principles

In the following we enumerate the guiding principles of the TiCOS implementation and focus on the requirements or modifications they may require to the design of ARINC653-compliant applications.

### 2.1.1. Run-to-completion

We enforce a run-to-completion semantics to minimize direct interference in the execution of a process. According to this design choice, within the boundaries of a partition, each job (i.e., execution instance) of a task (process) is guaranteed to complete its execution without being preempted by any other task, as scheduling decisions are deferred until the end of the job. In other words, the process scheduler (partition-wide) now relies on fixed-priority scheduling policy where preemption has been effectively deferred until the end of the every job run.

The first practical consequence of this approach is that we restrict the possible scheduling points to the end of an execution of a periodic/aperiodic process or a partition switch: when a task with a higher priority than that of the running task is inserted in the ready queue, the current executing task is not preempted but the scheduling of the new task is deferred to the end of the activities of the running task. We realized this solution simply deferring all scheduling decision at the end of the execution of the running task. Accordingly, any scheduling event (occurring as a result of either a periodic trigger or an asynchronous event) cannot have any effect in between two successive scheduling points, although it will be possibly acknowledged and taken into account. The effects of this preemption deferral mechanism on the overall system schedulability can be analyzed by means of mainstream analysis techniques.

***Guidelines***

Breaking up those processes characterized by longer execution into smaller run-time entities may be required to guarantee adequate responsiveness to asynchronous events.

***Requirements and limitations***

- No relevant requirement.

***Effects on legacy application***

In general, the way we implemented the run-to-completion semantics in TiCOS does not compromise the execution of legacy applications. Although non-preemptive systems in general reduce the system schedulability as compared to preemptive ones, the risk of breaking the feasibility of a legacy task set is negligible.

Different issues may arise when the functional behavior of an application relies on specific patterns of preemptions and interleaving between processes (e.g., via potentially blocking calls or EVENTS): in this case one cannot guarantee that the behavior of legacy applications is always strictly preserved.

Under the run-to-completion semantics, we do not need to worry about mutual exclusion as no process can be preempted while holding a lock so long as every job releases any lock that it has previous acquired. Hence, there is no point in using the ARINC SEMAPHORE primitives as a semaphore lock is always available by construction. We currently provide a void implementation of SEMAPHORE services (simply return a NO_ERROR value) so that legacy applications that make use of the SEMAPHORE API will still be able to run on top of TiCOS.

### 2.1.2. Single invocation event

Allowing more than a single invocation event for a process (task) complicates timing analysis in that the functional behavior of each process cannot be studied as a simple sequential program unit. In particular, if the functional specification of a process includes any call to potentially blocking procedures, then the process itself is characterized by as many additional invocation events as the number of potentially blocking calls.

As an example, the invocation of the SEND_BUFFER and RECEIVE_BUFFER operations, as well as the WAIT_EVENT service may, under certain conditions, block the execution of the invoking process: these conditions themselves coming true represent an additional invocation event. Enforcing all run-time entities in the application to exhibit a single invocation event (time- or event-triggered) eases the timing analysis of user applications.

In order to avoid multiple invocation events per process, all processes including potentially blocking calls should be broken down into as many sub-processes as needed to enforce the "single invocation event per process" rule. In Figure 1 is shown a simple example where a periodic process A is split into two sub-processes A' (predecessor) and A'' (successor): whereas A' performs all the activities of the original process up to the potentially blocking call, A'' is responsible for the remaining activities, inclusive of the invocation

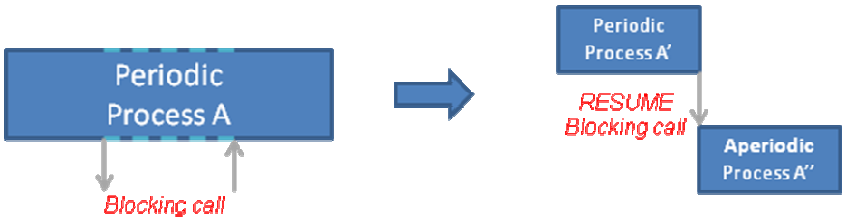


**Figure 1**: Splitting of a periodic process

of the blocking service.

In practice A'' should be implemented as an aperiodic process whose activation event is just the combination of a start event (triggered by A') and the potentially blocking condition.

In order to implement this new pattern we propose two different solutions. In the first solution the original behavior of A is further enforced by a careful assignment of process priorities: process A' in the example would be assigned the same priority as A, while A'' would be assigned the immediately lower priority value.

In the second solution (that we named O(1) scheduling with task split) we propose the successor task instead is activated (i.e. start its execution) as result of an event which is the combination of the condition: the original blocking condition and the precedence condition which binds the successor with its predecessor (i.e. the predecessor has executed). The implementation of this solution however has required to redesign the tasks synchronization mechanism used by the kernel, i.e. the *lock objects* data structure and the *Wait* and *Signal* procedures. The Wait procedure is used to check if the calling task (i.e. the task which want to wait for an event using a lock object) may execute (i.e. the blocking barrier of the lock object is open) or if it can not. In the former the task continues its execution, in the latter it must be blocked and enqueued to wait for the blocking condition to be met. The Signal procedure is used to set the blocking condition end eventually unlock the task with higher priority which is queued waiting for it.

We modified the original lock object implementation to achieve two goals: to execute the Signal operation in constant time and to introduce the precedence condition in the lock object barrier used by the Wait procedure.

As regards to the first goal we use the same solution already seen for the scheduling primitives: we replace the array of tasks state with a bitmask where each bit represents a task defined in the system and where the bits are ordered by task priority. As regards to the second goal we introduce a new bitmask, called executed predecessors to say, for each successor task, if its predecessor task has executed.

Using the executed predecessors bitmask we can add the precedence condition to the lock objects barrier simply using the bitwise AND operation; the new barrier implementation is:

*(executed predecessors AND task bitmask) AND original barrier*

where task bitmask is the bitmask corresponding to the calling task (i.e. the task which call the Wait procedure) and original barrier is the original lock object barrier replaced with a bitmask containing all ones if the barrier is open and all zeros if it is closed.

***Guidelines***

A sketchy representation of the sub-processes implementation is as follows:

First solution:

- *Process A'*

```
while (true) {
  // do something
  // WAIT_EVENT (event_id, 0, &(ret));
  RESUME (process A'');
  PERIODIC_WAIT (&(ret));
}
```

- *Process A''*

```
while (true) {
  SUSPEND_SELF (infinite, &ret);
  // on resumption wait for the event
  WAIT_EVENT (event_id, 0, &(ret));
  // do something
}
```

In this solution the successor task does not really have a single activation point; in fact, when it resumes after the SUSPEND_SELF call, it can be blocked again on the WAIT_EVENT call (this is the case when the event that is waiting for is down).

Second solution:

- *Process A'*

```
while (true) {
  // do something
```

```
    // WAIT_EVENT (event_id, 0, &(ret));
    PERIODIC_WAIT (&(ret));
}
```

- *Process A''*

```
while (true) {
      // single activation point
      WAIT_EVENT (event_id, 0, &(ret));
      RESET_EVENT (event_id, &(ret));
      // do something
}
```

- In this second solution the *Process A''* is resumed only when the event that is waiting for is up. As a consequence, the WAIT_EVENT call is not blocking and the task has a single activation point.

***Requirements and limitations***

- No relevant requirement (except for following the guideline above).

***Effects on legacy application***

- Besides being able to separate the effect of potentially blocking and highly variable IO service calls, the application of the above pattern allows to preserve the ARINC signature and semantics for all the potentially blocking services.
- Process splitting may lead to an explosion in the number of application processes. To counter this issue, one could reserve a fixed amount of processes to serve as parametric sporadic processes that take part into potentially blocking calls.
- Although legacy application will still be able to run on top of TiCOS, we should be aware of potentially different behaviour, mainly due to the inhibition of preemptions.

### 2.1.3. Scheduling

The scheduling and dispatching primitives have been redesigned to enforce a run-to-completion semantics for process execution as described in Section2.1.1; additionally, their implementation has been conceived to incur constant-time overhead on the executing user application.

The new O(1) scheduler is implemented by means of bitmasks and queues: the idea is that every thread is associated to a bit in a bitmask representing its state, whereas queues are needed to handle asynchronous events. The required data structures are the following:

One bitmask per partition (named `runnables`) tracks the evolving state of the processes in the partition, a value of "1" in position N meaning that the N-th process of the partition is runnable. This mask is updated and looked up at every dispatching point, to set the appropriate bit for the terminating process and to select the next running process, respectively.

- One bitmask per each scheduling slot defined in a MAF is statically built at system startup to identify which processes are runnable in the corresponding slot; such mask will handle the activation of periodic processes which is performed immediately at partition switch.

- One queue of asynchronous events tracks those activation events which are programmed to fire while some user process is running[1], when nor scheduling nor dispatching activities are programmed; queues are reordered as needed at every dispatching point, and every asynchronous event is guaranteed to be serviced (i.e. activated) at the first dispatching point after its programmed occurrence time.

Using bitmasks has the advantage of performing bitwise operations in constant time, but requires a fundamental assumption to hold, i.e. all processes should have distinct priorities; processes will then be sorted in decreasing priority order at creation time for efficiency reasons. Requiring distinct priorities is no restrictive assumption according to the ARINC specification, but might obviously impact legacy applications; process reordering instead is fully transparent to user applications, since no assumption on the creation order is made and processes can be still referred to via their PROCESS_ID as per ARINC API specification.

***Guidelines***

- Assign distinct priorities to user application processes. A process with priority level N has higher priority over a process with priority M iff N>M.
- User application processes should be assigned priorities greater or equal than 2 (being 0 and 1 priority levels assigned to partition main threads and the idle thread).
- The sub-processes created as a consequence of breaking down a potentially blocking process (as explained in section 2.1.2) should be assigned contiguous priority levels, in order for their sequential execution to be preserved in case the blocking does not occur.

***Requirements and limitations***

- Priority levels in ascending order and tasks with distinct priorities;
- Using the O(1) scheduler in combination with other schedulers for different partitions is not supported.

***Effects on legacy applications***

- Priority levels of user-level processes may need to be redefined in accordance to the guidelines above.
- A higher latency could be experienced by user application processes as a direct consequence of enforcing the run-to-completion semantics.

### 2.1.4. Time management

The original POK tick-based time management scheme has been discarded to avoid user process interruption, in favor of a timer-based mechanism supporting run-to-completion. The DEC PPC register is loaded at every partition switch with the time value assigned to the upcoming scheduling slot, so that a time interrupt will fire no earlier than the next partition switch. Within a partition thread dispatching is managed by the O(1) scheduler presented in section 2.1.3 with no need of timing information, except in the presence of asynchronous events. Whenever a user application may ask for the value of system time during its execution, it will be read from the PPC TB and compared to the value read at the last DEC interrupt, so that the actual time can be calculated.

In case sampling or queuing ports are used (see section 2.2.1) a configurable portion of each scheduling slot can be allocated to perform deferred write operations on output ports; in this case the DEC register will be configured to fire an interrupt earlier than partition switch, accounting for this reserved amount of time.

1 Asynchronous events are generated by the invocation of the DELAYED_START and TIMED_WAIT ARINC routines.

***Guidelines***

- The timer-based time management mechanism is automatically enabled when the O(1) scheduler is selected.

***Requirements and limitations***

- In case of inter-partition communications, a reserved time for writing on output ports should be configured as needed.

***Effects on legacy applications***

- Inter-partition communication in legacy applications will be broken when no or insufficient slack time is defined to accommodate deferred write operations on sampling and queuing ports.

## 2.2. ARINC APEX and Issues

In the following we discuss the most relevant assumptions in the implementation of TiCOS that may affect the design of a ARINC653-compliant applications. We highlight the set of prerequisites that must be fulfilled in order to have an application running on top of TiCOS.

### 2.2.1 Inter-partition communication

The execution time of IO services intrinsically depends on the size of the message that has to be read or written. Although this variability could be bounded forcing every IO service to transmit the maximum amount of data, this would entail an intolerable performance loss. The alternative solution we propose consists in removing from the IO service the variable part of the read/write operation and executing it on the partition switch.

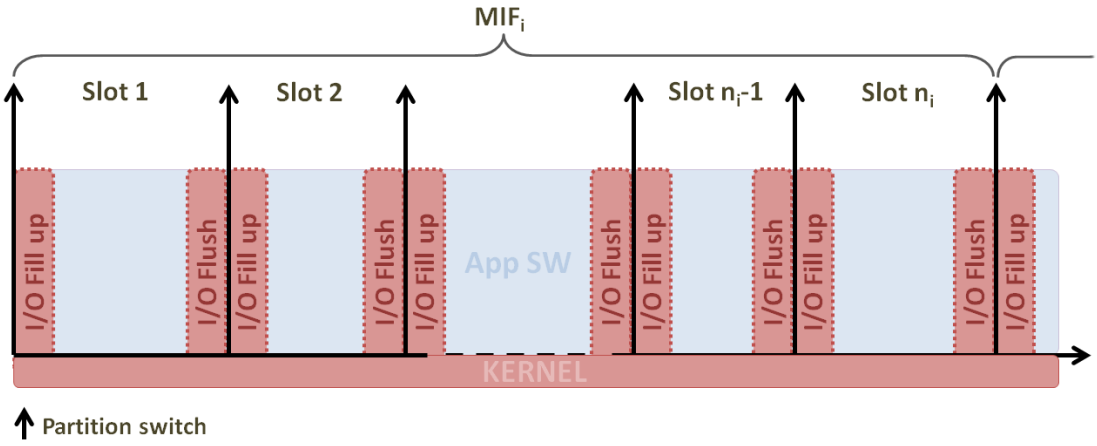


**Figure 2**: *Inter-partition IO management.*

As depicted in Figure 2, the messages written (or sent) by the application are copied to the output ports not during the IO service calls but at the end of the partition slot. For the same reason, the messages read (or received) from the input ports are pre-loaded in the input buffers (at the user space) at the beginning of each partition slot.

However, this technique if from one hand allows to execute IO services in constant time, from the other hand imposes to the application developers some constrains about the way the application need to be

implemented in order to be compliant with the time composable IO services implementation. In the following subsections we provide a rough sketch of the implementation of the IO services and outline the guidelines for designing applications that make use of the time composable Sampling and Queuing ports services.

### *Output ports*

In the output ports implementation we propose, in the WRITE_SAMPLING_MESSAGE and the SEND_QUEUING_MESSAGE services actually is not implemented any message transfer, but the address of the message to be written (contained in the MESSAGE_ADDR input parameter) is saved in the token associate to the port (dashed arrow in the Figure 3. The message will then be copied from the user message

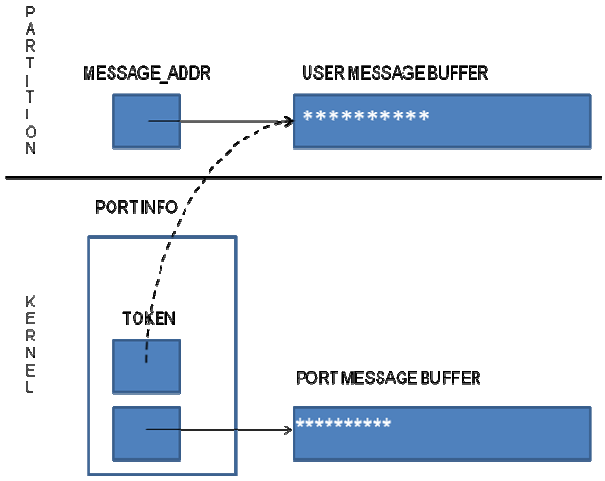


**Figure 3**: *WRITE/SEND implementation.*

buffer to the port message buffer at the end of the partition slot (as depicted in Figure 3).

#### *Guidelines*

- The only relevant guideline for applications that make use of WRITE_SAMPLING_MESSAGE or SEND_QUEUING_MESSAGE service calls is that of meeting the requirement below.

#### *Requirements and limitations*

- As the message is not flushed to the port message buffer during the WRITE_SAMPLING_MESSAGE/SEND_QUEUING_MESSAGE service call, the (user) buffer containing the message to be written or sent (i.e. the buffer pointed by the MESSAGE_ADDR input parameter) should not be overwritten until the end of the partition slot, when it will be really flushed to the port message buffer;
- Timeouts are not supported yet.

#### *Effects on legacy applications*

All calls to the WRITE_SAMPLING_MESSAGE/SEND_QUEUING_MESSAGE services from within legacy applications that do not meet the above requirement will result in an inconsistent behaviour. In fact, if the

(user) buffer containing the message to be written or sent is overwritten by the application code before the end of the partition slot (i.e. before the message will be really flushed to the port message buffer), the "correct" message will not be written in the port message buffer and will be lost. Moreover, the content of the port message buffer after the flush operation will be corrupted.

### *Input ports*

In case of input ports, the implementation of time composable IO services anticipates at the beginning of each partition slot the message transfer from the port message buffer (at the kernel layer) to the input message buffer (at the partition layer). The preload phase is executed at the beginning of the partition slot in which the port will be read; during this phase, the messages are copied from the port message buffers to the input message buffers allocated in the partition memory area at the deployment time (see "Requirements and limitations" below) and associated to each port at port creation (see dashed arrow in Figure 4).

Owing to the message preload in the READ_SAMPLING_MESSAGE and RECEIVE_QUEUING_MESSAGE services, the only operation necessary to make the application accessing the message is to set the MESSAGE_ADDR variable to the address of the input message buffer in which the message has been preloaded (see the dotted arrow in Figure 4).

In order to allow the READ/RECEIVE service to modify the content of the MESSAGE_ADDR variable, two solutions are possible: (i) passing to the READ/RECEIVE the **address** of the MESSAGE_ADDR variable; (ii) specifying, at deployment time, a static mapping between the input port and the **address** of the variable used in the READ/RECEIVE operations. The first solution is not strictly compatible with the ARINC653 standard, in fact, although it does not modify the ARINC653 API signatures, it imposes the strong requirement of passing to the READ/RECEIVE the address of the MESSAGE_ADDR variable instead of MESSAGE_ADDR variable itself. We named this solution "dynamic" because MESSAGE_ADDR variable whose address is passed as input parameter to the READ/RECEIVE function does not have to be the same in all READ/RECEIVE function invocations; moreover, it is not required to specify at deployment time any static mapping between input ports and variables addresses (as instead necessary in the second solution). The second solution, that we named "static", seems to be more compatible with the ARINC653 standard but imposes that all the READ/RECEIVE service calls that read messages from the same input port (even if invoked by different processes) must always use, as MESSAGE_ADDR input parameter, the variable whose address has been specified, at deployment time, in the static mapping between input ports and variable address.

It is worth underlining that when a READ/RECEIVE service is invoked, the address that is possibly contained in the MESSAGE_ADDR input parameter is not used at all by the READ/RECEIVE service calls (hence the MESSAGE_ADDR input parameter could even contain a NULL pointer). Moreover, the MESSAGE_ADDR variable, after a READ_SAMPLING_MESSAGE or a RECEIVE_QUEUING_MESSAGE service call, will contain the address of the input message buffer in which the message to be read/received has been copied during the preload phase.

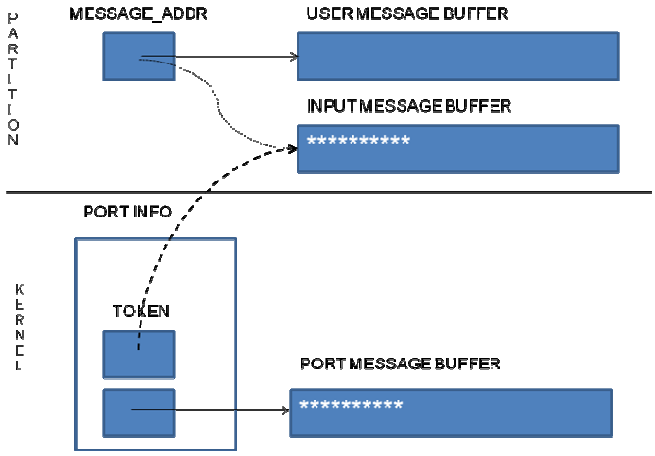


**Figure 4**: *READ/RECEIVE implementation*.

***Guidelines***

- Input message buffers must be allocated at deployment time: see "Requirements and limitations" subsection.
- In case of the dynamic solution is adopted, the **address** of the MESSAGE_ADDR variable instead of MESSAGE_ADDR variable itself must be passed to the READ_SAMPLING_MESSAGE and RECEIVE_QUEUING_MESSAGE service call as input parameters (this does not required any modification of the READ/RECEIVE APIs signature ).
- In case a dynamic solution is not adopted, the MESSAGE_ADDR variable passed as input parameter to the READ/RECEIVE service call must be always the same for all READ/RECEIVE service calls on the same port. Moreover, the address of that variable must be specified at deployment time and associated to a given port.

***Requirements and limitations***

- At deployment time we must: (i) statically allocate the input message buffers where the messages will be preloaded; (ii) specifying a mapping between each input port and the **address** of the variable used in the READ/RECEIVE operations on that port (only in case the dynamic solution is not adopted);
- Moreover, in case of queuing port in order to support several messages (with different size) at a time, the input message buffer in Figure 4 must be replaced with an array of input message buffers;
- Timeouts are not supported yet.

***Effects on legacy applications***

- If input message buffers are not allocated (and, in case of static solution, the mapping port – variable address is not specified) legacy applications including READ_SAMPLING_MESSAGE and RECEIVE_QUEUING_MESSAGE service calls will not compile.
- In case of dynamic solution, if the MESSAGE_ADDR variable itself (instead of its address) is passed to the READ/RECEIVE service call, the READ/RECEIVE semantics will become the following: the buffer pointed by the MESSAGE_ADDR variable will not contain the message, but the address of the input message buffer containing the message.
- In case of static solution, if (i) the MESSAGE_ADDR variable passed as input parameter to the READ/RECEIVE service call is NOT always the same for all READ/RECEIVE service calls on the same ports or (ii) the MESSAGE_ADDR variable passed as input parameter to the READ/RECEIVE service call is not that specified in the "port – variable address" static mapping, then when the READ/RECEIVE service call returns, the buffer pointed by the MESSAGE_ADDR variable does not contains the message to be read (unless the MESSAGE_ADDR variable – passed as input parameter- already points to the input message buffer assigned, at port creation, to that port);

### 2.2.2 Events

We currently provide an almost standard implementation of ARINC EVENT services. The main characteristics of our implementation are rather direct consequence of the design choices at the level of process scheduling. The run-to-completion semantics, in fact, would suggest a reinterpretation of an EVENT within the process split mechanism introduced in Section 2.1.2. The WAIT_EVENT service may in fact block the invoking process until the EVENT is eventually set.

***Guidelines***

- According to our reasoning in Section2.1.2, we maintain that all processes including potentially blocking calls should be broken down into as many sub-processes as needed to enforce the "single invocation event per process" rule;
- A process that is calling the WAIT_EVENT service should be split into two sub-processes so as to make the WAIT_EVENT call as the invocation event for a sporadic process;
- A process that is calling the SET_EVENT service could also be specularly split into two sub-processes so as to enforce a rescheduling after the event is set up. In this case, SET_EVENT would be performed by the first sub-process as its last action before self suspending, while the remaining activities would be performed by a sporadic process.

***Requirements and limitations***

- The FIFO queuing policy is not supported;
- Timeouts are not supported yet.

***Effects on legacy applications***

- Legacy applications that assume a FIFO ordering on the process waiting on an EVENT would exhibit a different behavior;

Similarly to SEMAPHORE services, since we assume that every job runs to completion, we do not need to guarantee mutual exclusive access to the object implementing an EVENT. Mutual exclusion is thus currently not supported.

### 2.2.3 Buffers

We currently provide a basic implementation of ARINC BUFFER services. Again, the main characteristics of our implementation are rather direct consequence of the design choices at the level of process scheduling. The run-to-completion semantics, in fact, would suggest a reinterpretation of buffer-related service within the process split mechanism introduced in Section2.1.2. The RECEIVE_BUFFER service may in fact block the invoking process until the BUFFER is not empty; similarly, a SEND_BUFFER request may block the invoking process until the BUFFER has some room to accommodate a new message.

***Guidelines***

- Again (see Section2.1.2), we suggest to split all those processes that include potentially blocking calls into as many sub-processes as needed to enforce the “single invocation event per process” rule;
- Processes that are invoking the RECEIVE_BUFFER, SEND_BUFFER services should be split into two sub-processes so as to make these both calls as the invocation event for as many sporadic processes.

***Requirements and limitations***

- Similarly to events, the FIFO queuing policy is not supported;
- Timeouts are not supported yet.

***Effects on legacy applications***

- Legacy applications that assume a FIFO ordering on the process waiting on an RECEIVE_BUFFER would exhibit a different behavior;
- Similarly to SEMAPHORE services, since we assume that every job runs to completion, we do not need to guarantee mutual exclusive access to the object implementing a BUFFER. Mutual exclusion is thus currently not supported.

### 2.2.4 Blackboards

We currently provide a basic implementation of ARINC BLACKBOARD service. As in case of BUFFER service, the main characteristics of our implementation are a direct consequence of the design choices at the level of process scheduling. The run-to-completion semantics, in fact, would suggest a redesign of the blackboard-related services along with the process split mechanism introduced in Section 2.1.2. The READ_BLACKBOARD service may in fact block the invoking process until the BLACKBOARD is not empty.

***Guidelines***

- Again (see Section2.1.2), we suggest to split all those processes that include potentially blocking calls into as many sub-processes as needed to enforce the “single invocation event per process” rule;
- Processes that are invoking the READ_BLACKBOARD service should be split into two sub-processes so as to make each call as the invocation event for as many sporadic processes;
- A process that is calling the DISPLAY_BLACKBOARD service could also be secularly split into two sub-processes so as to enforce a rescheduling after a message is displayed in a blackboard. In this case,

DISPLAY_BLACKBOARD would be performed by the first sub-process as its last action before self suspending, while the remaining activities would be performed by a sporadic process.

***Requirements and limitations***

- Similarly to events, the FIFO queuing policy is not supported;
- Timeouts are not supported yet.

***Effects on legacy applications***

- Legacy applications that assume a FIFO ordering on the process waiting on an READ_BLACKBOARD would exhibit a different behavior;
- Similarly to SEMAPHORE services, since we assume that every job runs to completion, we do not need to guarantee mutual exclusive access to the object implementing a BLACKBOARD. Mutual exclusion is thus currently not supported.