A Comprehensive Approach in Performance Evaluation for Modern Real-Time Operating Systems

Alberto García-Martínez, Jesús F. Conde and Angel Viña

CESAT, Open Real-Time Systems Center
Universidad de La Coruña
La Coruña (Spain)
E-mail: [alberto, jconde, avc]@cesat.es

Abstract

In real-time computing is essential the accurate characterization of the performance and determinism that a particular real-time operating system/hardware combination can provide for real-time applications. This issue is not properly addressed by existing performance metrics, mainly due to the lack of completeness and generalization. In this paper we present a set of comprehensive, easy-to-implement and useful metrics covering three basic real-time operating system features: response to external events, intertask synchronization and resource sharing, and intertask data transferring. The evaluation of real-time operating systems using a set of fine-grained metrics is fundamental to guarantee that we can reach the required determinism in real-world applications.

1. Introduction

RTOSs play a key role in most real-time systems. A RTOS must be able to respond to internal/external events in a deterministic timeframe, incorporating features and primitives for multitasking with preemptive priority scheduling, efficient interprocess communication and synchronization, and predictable interrupt response.

A comprehensive performance evaluation of a RTOS may provide useful information in order to obtain the following benefits:

- Choice of the most adequate RTOS to meet the performance and determinism needed by a specific real-time application
- Precise selection of the underlying hardware platform
- Optimal implementation and tuning of the real-time application

In this paper we present a comprehensive methodology and its related metrics for quantitatively measuring real-time performance and determinism of present-day RTOSs. The next section addresses an overview of performance evaluation for RTOSs, presenting a comprehensive approach. Performance measures are discussed in three different sections: response to external events (section 3), intertask synchronization and resource sharing (section 4) and intertask data transferring (section 5). We will end with the conclusions.

2. Performance evaluation for real-time operating systems

There is a growing need for performance measures specifically intended for real-time computer systems. Researchers have attempted to gain insight in RTOSs performance by means of two main approaches:

- **Fine-grained benchmarks**: they investigate a RTOS at a low level, evaluating the efficiency of the hardware and software interaction for the most frequently used services.

  Rhealstone [5] is the best known fine-grained real-time benchmark. The Rhealstone metric is used to obtain a figure of merit from six quantitative measurements: task switching time, task preemption time, interrupt latency time, semaphore shuffling time, deadlock breaking time and datagram throughput time.

- **Application-oriented benchmarks**: they take a much higher level look at a RTOS, usually in terms of the number of deadlines kept or missed and the utilization point at which the system begins to break down. They are often implemented as synthetic

1089-6503/96 $5.00 © 1996 IEEE
Proceedings of EUROMICRO-22

applications running on top of the real-time executive.

Hartstone [8] is the best known application-oriented real-time benchmark suite. It consists of five series of tests that mix periodic and aperiodic tasks, with increasingly frequency requirements. The results are based on the number of missed deadlines.

However, the two benchmarks mentioned above are not adequate for real-time applications that demand a deep knowledge of the underlying RTOS predictability to guarantee that they will meet their requirements.

- *Rhealstone* is not complete. There are situations not considered in its metrics (e.g., time for priority inversion management for a given resource) that can be of special relevance for many present-day real-time applications.

  In addition, Rhealstone has serious drawbacks:

  - It is not focused on providing accurate worst-case measures. It only pretends to compare the average performance of typical real-time operations in different RTOSs.

  - The six measurement categories are somewhat ad-hoc. The metrics leave uncovered particular situations commonly found in real-time applications, that may cause severe predictability lacking.

  - A single figure may not be obtainable for preemption time, since preemption can be quite complicated in a multitasking real-time system due to priority inversion, etc.

  - The deadlock breaking criterion loses importance in the face of algorithms that avoid priority inversion (priority inheritance or priority ceiling) used by well-designed schedulers.

- *Hartstone* is excessively generic. It gives an estimation of the interaction of different scheduling techniques, but lacks details such as response to external events, intertask data transferring, etc. It seems to be a useful test for a very specific type of applications, but not a broadly-oriented benchmark. Hartstone is not satisfactory in predicting the behaviour of a RTOS.

The metrics proposed in the following sections are intended to be comprehensive, useful and easy to implement, thus leading to a fine-grained evaluation of the following essential features of modern RTOSs kernels:

- Response to external events (interrupts)
- Intertask synchronization and resource sharing (object synchronization)
- Intertask data transferring (message passing)

An exhaustive analysis of all the possibilities involved in the use of this features, along with the use of worst-case measurements, will improve understanding and prediction of real-time systems.

3. Response to external events

Response to external events by means of hardware interrupt handling has been deemed as a foremost issue for real-time systems. Not only fast and predictable response is required; a RTOS must also optimize the interaction between application processes and external events, acting in a timely manner and reducing the time consumed to a minimum.

We will first describe the behaviour of a system as a consequence of an interrupt occurrence, continuing with the exposition of the metrics proposed for the characterization of this issue.

3.1. Interrupt processing

Whenever an external device wants to inform of an event occurrence, it sends a signal to the CPU. The signal received is only transformed into an interrupt if the interrupt mask is set to an enabled state for the considered request. The mask is automatically altered when an interrupt arrives, although it can also be modified by the operating system, in order to protect critical code segments (e.g., those involved in context switches). The interval in which interrupts cannot be acknowledged is frequently known as Interrupt Disable Time [6]. The time elapsed until the processor sends an acknowledge signal to the device that caused the interrupt is called Interrupt Latency Time.

Once the interrupt is accepted, the processor has to complete the instruction currently in process, time that can be considerably long for complex CISC instructions. After the instruction has been processed, the processor's state has to be preserved in order to continue with the execution of the application code after the completion of the interrupt service. In modern microprocessors, with sophisticated pipelined and superscalar architectures, storing the state it is not an straightforward issue [7]. The processor's state will be restored after the interrupt code fetches the return instruction. Then, a start-up time must be considered, involving the refilling of the pipeline. This interval is normally short and occurs in a non-critical timeframe.

Subsequently the Interrupt Service Routine (henceforth ISR) is vectored. The RTOS ordinarily requires the provision of a sufficient context for the execution of the ISR, and the preservation of the state of the task that was running when the interrupt arrived. We can call this period the Preprocessing Time for the ISR. The time consumed in this operation may be minimized by limiting the size of the

context needed for the execution of the ISR in the RTOS. There is a compromise between the complexity of the state structure provided for the ISR and the variety of the operations callable inside the interrupt framework (for example, generally the ISR context is not large enough to allow operations leading to block itself).

At this point, the system is ready for performing useful work either for an application waiting for the interrupt, or for the device that caused it. The period elapsed until the first instruction of the responding task begins its execution is known as ISR Dispatch Latency Time.

Finally, when the ISR returns, the processor will execute kernel code that will undo the state changes made in the preprocessing phase. The Interrupt Service Time is the time taken by the ISR to service the interrupt, including this Post-processing Time. It heavily depends on the device serviced, and the application considered. This Post-processing phase is essential for determining the minimum time needed to repeat the execution of the interrupt service.

The Interrupt Service Time should be kept as small as possible, in order not to interfere with the service of other interrupts and with the scheduling of real-time processes, that are handled at a lower scheduling priority. Thus, maintaining the interrupts disabled for a long time, due to an incorrect kernel design or excessively long ISRs, should be avoided. The work related to the external event serviced in the ISR will be limited to a minimum, and the remaining job will be deferred to task context code. The time span up to the beginning of the execution of the first instruction of the task code running in a schedulable context is of paramount concern. In the following paragraphs we will consider several approaches found in modern RTOSs for the switch from an ISR to a schedulable context. The usage of the different options depends on the facilities offered by the operating system and how the designer has conceptually outlined the application.

It is reckoned as basic to ease the synchronization between normal-flow application tasks and external events. Tasks may want to be aware of the occurrence of an event and perhaps perform kernel-mode operations (accessing to the hardware or memory, executing processor's privileged instructions). Although in simple RTOSs this relation can be established using semaphores, sometimes the communication between interrupt level code and schedulable code is restricted. In this case a valuable parameter is the User Task Dispatch Latency Time, that accounts for the time span from the activation of the external event to the execution of the first instruction within the responding task.

Most commonly, drivers are required. A driver, or device controller, is code that encapsulates in a standard and popular interface all the functions needed for the interaction with a device. The usage of drivers combines several advantages: uniform access to different devices, code reusability for similar devices, structured concurrent access management, code modularity, and an appropriate framework for encouraging portability.

For systems in which user tasks do not possess the same privileges as kernel tasks, drivers are the only means for the user level to request kernel features. Driver-related metrics are relevant in order to characterize the operating system's behaviour when performing control issues. One metric that should be considered is the Driver Dispatch Latency Time, defined as the time interval from the instant when the interruption was raised until the first instruction of the driver's code waiting for the occurrence of the event is executed. The synchronization with the interrupt thread should be accomplished in the fastest possible way (usually by means of semaphores). The User Task Dispatch Latency Time (the driver may activate user task code) takes into account the overhead included in the driver facility. Some RTOSs offer kernel-level threads that gather thread benefits with kernel-level features for driver code.

In the evaluation of the time consumed in the switch from the ISR to the newly activated task, we have to note that the change to the schedulable code (supposed of higher priority than all the currently active tasks) can be delayed for RTOSs built upon non-preemptable kernels. If the system were interrupted while servicing a system call, it would need to conclude its related work prior to being able to reschedule the processor.

Additional delays can be found in special situations. Some operations, without being so restrictive to require interrupt disabling, may demand certain degree of privacy that is usually obtained by applying mutual exclusion for the access to common data structures. If an ISR asks for a service that must be accessed in a mutual exclusion fashion, and the task interrupted was making use of the shared resource, the ISR will be forced to stop until the task exits the critical section. Two context switches will be added. We want to stress that this situation will only happen if the operating system provides the ISR with an appropriate context to allow blocking. If the RTOS does not allow blocking operations, some calls like the ones related to object creation (tasks, semaphores, queues) or memory allocation will be forbidden.

3.2. Metrics proposed

The measures will be based on an interrupt-generator device specially adapted for our needs: one of the timers commonly found in hardware platforms. We perform the measure by substracting the value read from the initial interrupt time. Analogous measures have been carried out in [6] and [2], and they are easily portable.

Since the metrics are aimed to offer complete information to the designer/programmer of real-time applications, all the possible situations must be considered,

including driver processing and the deferring of the execution to the activated task. Most of the tests carried out [6, 2] deal exclusively with ISR Latency Dispatch Time, ignoring the subsequent driver and task deferring. While achieving determinism in the ISR response is almost straightforward, maintaining it after the transition to the schedulable context is not as simple, as we have pointed out before.

The underlying software nature of the measures should not be considered as a disadvantage. It should be noted that it reflects what developers can expect from their systems running real-world software applications.

The tests identified, with their corresponding measures are:

- **Interrupt-1**. We will first consider the case in which the RTOS allows the synchronization between a task and an ISR without using the driver's interface, directly awakening a waiting task with a semaphore. In all the tests, the ISR execution code is exclusively devoted to the measure of the times considered, causing a small overhead that can be easily estimated. We distinguish two important values: the ISR Dispatch Latency Time, and the User Task Dispatch Latency Time.

- **Interrupt-2**. In the second test (see Figure 1) we include the usage of a driver. Task A performs a `read()` call, and the corresponding driver routine blocks in a semaphore. After an interrupt arrival, the corresponding ISR is activated and relinquish the semaphore, awakening the driver code. Finally, control is returned to the calling task. To completely characterize the influence of the driver's mechanism, time employed in the `read()` call should be measured. ISR Dispatch Latency Time will be the same for the two tests performed.

4. Intertask synchronization and resource sharing

Synchronization objects are RTOSs services used to control access to shared resources and to signal synchronous events among tasks. Since these objects are frequently used in real-time programming, the impact of its execution times may be determinant for the achievement of the performance required.

4.1. Synchronization objects

The most frequently implemented synchronization objects (counting semaphores, mutexes and condition variables) possess a similar structure consisting of an integer (indicating a state), a task queue, and the necessary primitives to perform adequate actions depending on the state.

A semaphore is the lower level mechanism that a RTOS provides for synchronization purposes between several tasks. The actions carried out by the kernel whenever a semaphore primitive is called are summarized in the following two tables:

TABLE 1. Possible Actions when `request` is invoked

primitive	state>0	state<=0
request	decrement state (*request-decr*)	decrement state and block task (*request-block*)

A mutex is a variation of the semaphore scheme, modified to address the particular problem of serialization in accessing shared resources. Its main characteristic resides in the concept of ownership: the task that acquires

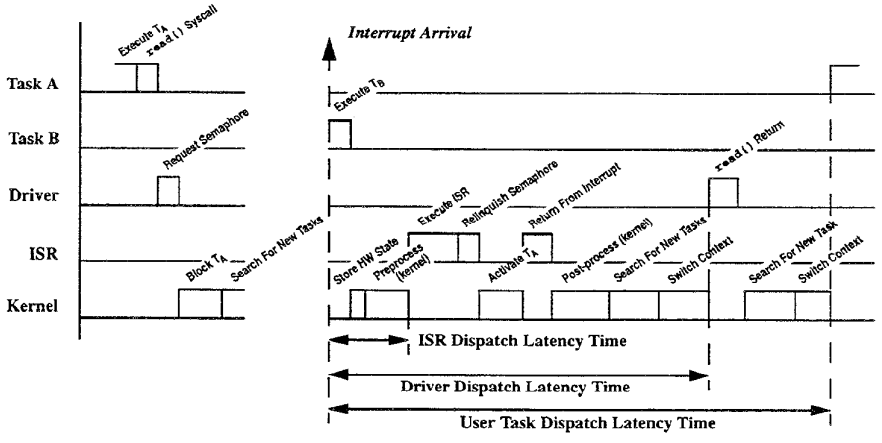

Figure 1. Interrupt-2.

TABLE 2. Possible Actions when `relinquish` is invoked

primitive	state>=0	state<0
relinquish	increment state (*relinquish-incr*)	increment state and unblock blocked tasks (*relinquish-unblock*)

the mutex is the only allowed to release it. This feature allows the implementation of protection and error recovering mechanisms, while providing additional services (implementation of control algorithms, e.g. Basic Priority Inheritance) in order to prevent priority inversion.

Condition variables are useful to solve a problem frequently found in intertask synchronization in a neat and efficient way: the access to a shared resource only when a given condition is satisfied. Condition variables are closely related to mutexes.

4.2. Metrics proposed

In order to account for all the potential interactions among the actions exposed, we propose the following measurements:

- **Semaphore-1** provides an estimation of the time employed by the RTOS in *request-decr* and *relinquish-incr*. Task A requests the semaphore (state>0) and then relinquishes the semaphore (state>=0).

- **Semaphore-2** measures the time employed by the RTOS in *request-block* and *relinquish-unblock* (figure 2). Task A (higher priority) requests the semaphore (state=0) and gets blocked in an empty queue. Task B (lower priority) relinquishes the semaphore (state<0) and the kernel unblocks Task A.

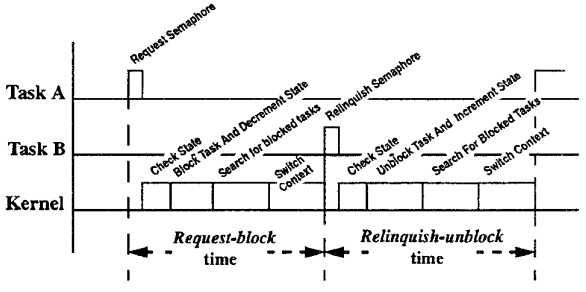

Figure 2: Semaphore-2

- **Semaphore-3** evaluates *request-block* and *relinquish-unblock*, taking into account the effect of the task's relative priorities (figure 3). Task A (higher priority) requests an auxiliary semaphore (state=0) and gets blocked in an empty queue. Task B (medium priority) requests the semaphore (state=0) and gets blocked in another empty queue. Task C activates and relinquishes the auxiliary semaphore, and consequently Task A becomes active. Task A relinquishes the semaphore, and the kernel acts similarly to the *relinquish-unblock* in Semaphore-2 except in that no context switch is made. Finally Task A resumes execution.

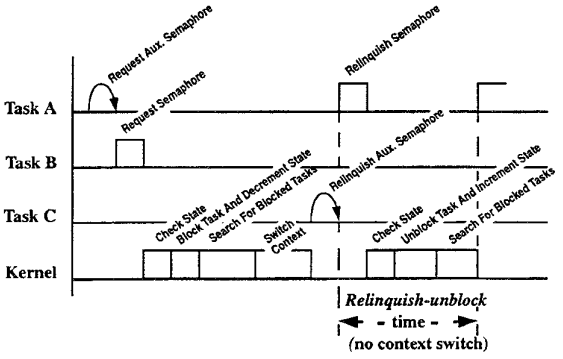

Figure 3. Semaphore-3.

Commercial RTOSs may behave in different ways in this measurement:

1. *Non-deferred unblocking*: as shown in figure 3, when Task A relinquishes the semaphore, the kernel unblocks Task B (although Task A keeps executing). This approach is more intuitive.
2. *Deferred unblocking*: after Task A has relinquished the semaphore, the kernel unblocks Task B only if its priority is higher than priority of Task A. If this is not true, the unblocking of Task B is deferred until the context switch for Task B becomes necessary. This implementation diminishes *relinquish-unblock* time, but it increases context switch time.

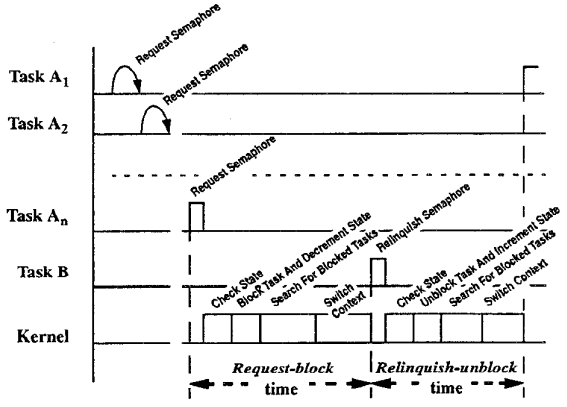

Figure 4. Semaphore-4.

• **Semaphore-4.** This test is similar to **Semaphore-2**, but now the influence of several queued requests is considered. Figure 4 shows it graphically.

• **Semaphore-5** is analogous to **Semaphore-3**, considering the influence of the different queue management policies. Figure 5 shows it graphically.

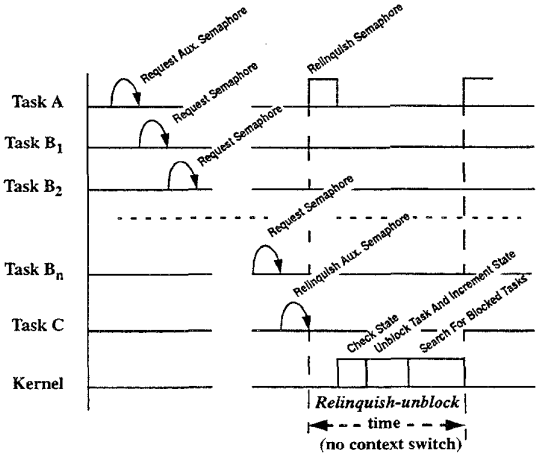

Figure 5. Semaphore-5.

The five former tests can be easily adapted to the corresponding ones for mutexes. One additional test is needed to reflect the influence of a priority inversion control mechanism, usually based in a Basic Priority Inheritance algorithm.

• **Mutex-6.** This measure evaluates the cost of priority inheritance (BPI algorithm). Figure 6 shows it graphically.

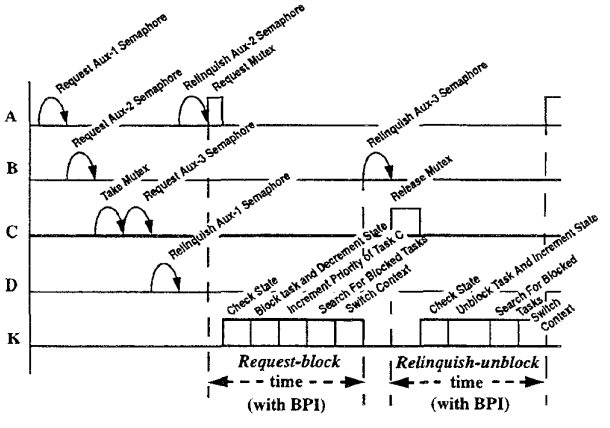

Figure 6. Mutex-6.

The tests just explained can be also applied to condition variables, after the appropriate adaptations in order to take into account its stateless condition and its special relation with mutexes. The metrics are analogous to the exposed above, presenting the following differences:

• **Condition-variable-1**. For this test we will only perform a relinquish operation, that does not cause an increment (as there is no state). If there are no blocked tasks, this relinquish will not produce any profitable action. In this case we can name this action as *null-relinquish*. We want to stress that a mutex operation is always implicit in condition-variable operations, and hence the mutex operation times will be included in the measures.

• **Condition-variable-2**. In this test we will deal with blocking and unblocking operations. The request operation is always blocking (*request-block*). The *relinquish-unblock* operation acts in a more subtle way, unblocking a previously blocked task, but delaying a possible context switch until the releasing of the related mutex. Consequently, the time measured in the unblocking operation should embody the *relinquish-unblock* and the mutex release operations.

• **Condition-variable-3**. We will measure in this test the time elapsed in the activation of a lower priority task by a higher priority task. This time we will only need to measure the *relinquish-unblock* operation, without including the subsequent mutex operation.

In the application of all the measurements just presented, is advisable to consider factors such as: time-outs, model of tasks (processes, threads), total RTOS workload, etc. A complete test would show how the results vary when the factors mentioned above are modified.

5. Intertask data transferring

Message passing is a widely extended mechanism for the interchange of formatted data streams among arbitrary tasks, providing simultaneously synchronization facilities.

5.1. Message passing

The key structures involved in message passing are: a queue, that may contain sent messages waiting for a destination, and two additional task queues (one for sender tasks and another for receiver tasks).

As it happened in the previous section for request and relinquish operations, send and receive operations result in different actions and therefore different measured times, depending on the state of the element. The six basic actions identified are:

• *send-data.* A task delivers a message to a queue with space enough to store it. Data is copied from the task space to the kernel buffer.

• *receive-data.* When a receive operation is signalled and there were previously stored messages, the kernel copies a message from his memory zone to the user's one.

• *send-block.* If a task performs a send operation while the message queue is full, it is blocked and the CPU is granted to another active task.

• *receive-block.* If the running task that asks for a message finds the message queue empty, it is blocked on the receiver's queue.

• *send-unblock.* A message is sent to a queue, with blocked tasks waiting for new messages. One of the waiting tasks is awakened, and data is copied directly from sender to receiver. Once the data has been transmitted, the kernel seeks for the next task to execute, switching contexts if necessary.

• *receive-unblock.* In this case, when a task performs a receive operation, enough space may be released and one (or more) of the tasks that were blocked when trying to send data can be unblocked. Data is copied from the first stored message of the message queue to the receiving task. Afterwards, a sending task is unblocked and the message data is copied into the message queue structure. This last step may be repeated several times, until the message queue is refilled. Some systems may follow different policies (for example, by directly passing a message from a blocked task to the receiver, avoiding unnecessary operations by means of not preserving FIFO posting). The task unblocking may lead to a context switch.

The transference of a message between two tasks always involves costly operations. On one hand, if the *send_data/ receive_data* sequence is performed, the same information is copied twice. On the other hand, with blocking/ unblocking operations, only one copy is made, but the transmission is burdened with expensive task management procedures.

A remarkable property is the duality found between send and receive: there is a symmetry between both functionalities, although the way the unblocking process is implemented may vary substantially, as we have pointed out above. Some advantages may be obtained by working with a queue offering empty slots (and in consequence without blocked sender tasks), and commercial RTOSs are usually tuned under these premises.

5.2. Metrics proposed

As we did for synchronization objects, we will present several tests constructed with the building blocks described in the previous section.

• **Message-passing-1** will account for the *send_data* and *receive_data* functionalities. Task A sends a message to an empty queue; another task receives it. The test can be executed using a single task (fig. 7).

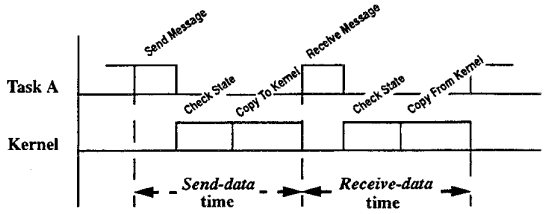

Figure 7. Message-Passing-1.

• **Message-passing-2.** The test previously described may be affected by the number of messages already present in the queue. This second test is similar to the previous one, but the queue is filled with a variable number of messages, for a complete characterization of this effect.

The next tests are devoted to the evaluation of blocking and unblocking operations.

• **Message-passing-3.** We will consider here the time span consumed by Task A when asking for a message in an empty queue, and the amount of time passed when a lower priority task (Task B) unblocks Task A by sending a message (see Figure 8).

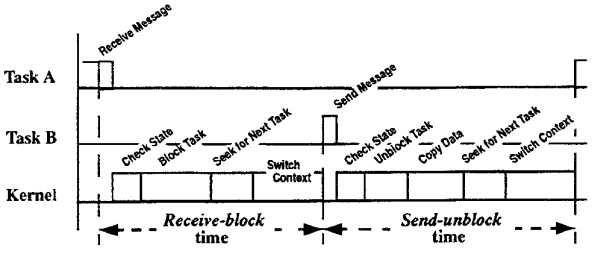

Figure 8. Message-Passing-3.

• **Message-passing-4.** The test is similar to Message-Passing-3, but now Task B (lower priority) is previously blocked (it asked for a message in an empty queue), and it is unblocked by Task A (sending a message). This test (shown in figure 9) will be held with the use of an auxiliary semaphore. As we stated in synchronization objects, there are alternative behaviours that can be

observed for different RTOSs. As the context switch does not take place in the send command, but when a later service is asked, either the unblocking of the task or the copy of the data (or both) can be deferred to the context switch. However, this behaviour is more infrequent in present-day RTOSs than the equivalent one described for semaphores.

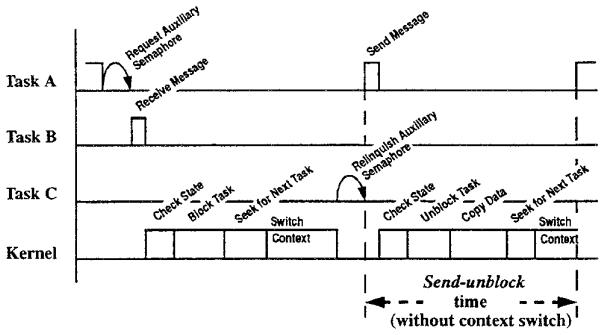

Figure 9. Message-Passing-4.

Due to the symmetry found in the message passing mechanism between send and receive, each test proposed for the evaluation of blocking and unblocking operations will have its corresponding dual counterpart. However, the conceptual duality will not necessarily be transformed into performance duality. Thus, two more metrics (**Message-passing-5** and **Message-passing-6**) are added simply by swapping the words "send" and "receive" for **Message-passing-3** and **Message-passing-4**. In these latter measures, the queue will be pre-filled with the appropriate number of messages.

When blocking and unblocking operations are involved, additional tests should be executed in order to reflect that the number of tasks in the blocking queues may affect the performance measured. Therefore a new set of tests is proposed, adapting **Message-passing-3, -4, -5** and **-6** to the presence of a certain number (that should be variable) of blocked tasks, resulting in **Message-passing-7**, **-8**, **-9** and **-10**.

Another key factor in message passing is the number of transmitted bytes. A complete quantitative description of the message passing mechanism in a RTOS will include results for a wide range of transfer sizes for the ten measures explained above. The comparison of these experimental results with the results obtained via a simple copy primitive will be helpful in evaluating the mechanism. We should recall a principle stating that for large quantities of data the usage of shared memory policies is more suitable than message passing, because the copying phase arises as the major drawback in the message passing process. A thorough discussion on this issue, and an example for a commercial RTOS, can be found in [4].

6. Conclusions

In this paper we have presented several metrics in order to address the comprehensive evaluation of the essential services provided by a RTOS: response to external events, intertask communications and resource sharing, and intertask data transferring. For each of the former features we have described several fine-grained tests, covering all the possibilities that may arise while implementing a real-time application. We want to stress the attainment of a set of tests that pretend to be complete, general, easy to implement, neatly exposed, and last but not least, useful.

The theoretical work presented in this paper has been applied to two of the most popular RTOSs, namely VxWorks and Solaris [3]. The different approaches taken in the implementation of the two RTOSs have contributed to validate the metrics shown. In the application of the tests, useful information (advantages in certain programming strategies, operating system bugs, performance guidelines) concerning the behaviour of the two systems has outcome. The knowledge obtained from the application of the tests to Solaris has been of capital importance in the development of a real-world application [1].

7. References

1 Conde, J. and Viña, A. *A Distributed Real-Time Architecture to Satisfy Hard I/O Throughput Requirements.* Proceedings of the 21st IEEE Euromicro Conference on Design of Hardware/ Software Systems. Como (Italy), 1995.

2 Faller, N. *Measuring the Latency Time of Real-Time Unix-like Operating Systems.* Technical Report TR-92-037. Berkeley University. Junio 1992.

3 García-Martínez, A. *Banco de Pruebas de Sistemas Operativos de Tiempo Real.* Technical Report CESAT-TR-95-06.

4 García-Martínez, A., Juanes R., and Viña, A. *A Fixed-Time Shared Memory Based Solution for Different Size Interprocess Data Transferring.* Technical Report CESAT-TR-95-02.

5 Kar, R. P., and Porter, K. *Rhealstone - A Real-Time Benchmarking Proposal.* Dr. Dobb's Journal, February, 1989.

6 Maechtel, M., and Rzehak, H. *On realtime operating systems: How to compare performance.* 1994 Workshop on Real-Time Programming, Lake Constance. June 1994.

7 Walker, W., and Cragon, H. G. *Interrupt Processing in Concurrent Processors.* Computer, Vol 28, n° 6, June 1995.

8 Weiderman, N. *Hartstone: Synthetic Benchmark Requirements for Hard Real-Time Applications.* Technical Report CMU/SEI-89-TR-23. Carnegie Mellon University, June 1989.