Evaluation of New POSIX Real-Time Operating Systems Services for Small Embedded Platforms

Mario Aldea Rivas and Michael González Harbour

Departamento de Electrónica y Computadores
Universidad de Cantabria
39005-Santander, SPAIN
{aldeam,mgh}@unican.es

Abstract[1]*: The ongoing revision of the POSIX.13 standard —real-time profiles for portable operating system interfaces— proposes adding new services to the Minimum Real-Time System Profile that are considered useful to the small embedded applications to which this profile is targeted. Concerns have been raised that these services may introduce too much overhead or may be difficult to implement. In this paper we evaluate the implementation of some of these new services in our MaRTE operating system. The implemented services are the monotonic clock, a high resolution sleep operation with specifiable clock, execution-time clock and timers, the sporadic server scheduling policy, and the timed mutex lock operation. We show that the complexity of these implementations is small, and the overheads introduced by the new services are fully acceptable.*

1. Introduction

POSIX is the acronym for Portable Operating System Interface. It is a proposed operating system interface standard based on the popular UNIX operating system; its main goal is to support application portability at the source-code level. It is being standardized by the Computer Society of IEEE as the IEEE standard P1003, and also by ISO/IEC, as the international standard ISO/IEC-9945 [1].

Because of the need to achieve application portability for real-time systems, a Real-Time System Services Working Group was established in POSIX. This group is developing standards to add POSIX (or UNIX) the OS services that are needed by real-time applications. The charter of the POSIX Real-time Working Group is to “develop standards which are the minimum syntactic and semantic changes or additions to the POSIX standards to support portability of applications with real-time requirements.” The real-time working group has developed several real-time extensions to the POSIX system interfaces, and the threads and trace extensions.

Many real-time applications, such those designed for small embedded systems, have special physical constraints that require operating systems with a reduced set of functionality. For example, many systems exist which cannot have a disk drive, do not have a hardware memory management unit, and have a small amount of memory. For these systems it is necessary that the standard allows implementations to only support a particular subset of the POSIX functions. The subsets necessary for real-time applications are also being addressed by the Real-time System Services Working Group in the POSIX.13 [2] standard. In this standard four realtime application environment profiles have been specified: minimal real-time system (for small embedded systems), real-time controller, dedicated system (for large embedded systems), and multi-purpose real-time system.

Currently there is revision process for the real-time system profiles [3]. One of the main objectives of this revision is to add the new real-time services incorporated into the POSIX standard since 1998, in particular the additional and advanced real-time extensions (POSIX.1d and POSIX.1j), tracing (POSIX.1q), and networking (POSIX.1g). Another important objective is to modify the existing profiles according to field experience on their implementation and use.

Some of the new real-time extensions have been proposed for their inclusion in all of the profiles, including the smaller one. Concerns have been raised that these services may be too complex to implement in the context of a very small kernel, or may introduce too much overhead. It would be interesting to have an evaluation of the impact of implementing these services in a small embedded kernel that follows the POSIX.13 minimal real-time system profile. This paper provides the results of such evaluation, and proposes a particular way of implementing the new OS services.

1. This work has been funded by the *Comisión Interministerial de Ciencia y Tecnología* of the Spanish Government under grant TIC99-1043-C03-03 and by the *Commission of the European Communities* under contract IST-2001-34140 (FIRST project)

The new services have been implemented in our operating system MaRTE OS (Minimal Real-Time Operating System for Embedded Applications) [4], which is a real-time kernel that follows the Minimal Real-Time POSIX.13 profile, providing both the C and Ada language POSIX interfaces. It allows cross-development of Ada and C real-time applications. Mixed Ada-C applications can also be developed, with a globally consistent scheduling of Ada tasks and C threads.

The paper is organized as follows: Section 2 gives a quick overview of the current POSIX real-time profiles and of their proposed revision. Sections 3 to 7 briefly describe each of the new services, propose an implementation for them, and gives the results of the evaluation. Finally, Section 8 gives our conclusions.

2. The POSIX.13 real-time profiles

Because the POSIX standard is so large, subsets are defined to enable implementations for a wide range of systems: from small embedded systems to large general-purpose computers with real-time requirements. The main characteristics of the four real-time profiles defined by POSIX.13 are:

- *PSE51*: *Minimal real-time system profile*. Implementations of this profile are not required to support multiple processes, nor a full featured file system. The unit of concurrency is the thread. Input and output is possible through predefined device files, but no regular files can be created. This profile is intended for small embedded systems. Most of the complexity of a general purpose operating system is eliminated: PSE51 systems can be implemented with a few thousand lines of code, and with memory footprints in the tens of kilobytes range. Our MaRTE operating system [4] is an implementation of this profile.
- *PSE52*: *Real-time controller profile*. It is similar to the PSE51 profile with the addition of a file system in which regular files can be created and read or written. It is intended for systems like a robot controller, which may need support for a simplified file system.
- *PSE53*: *Dedicated real-time system profile*. It is intended for large embedded systems, such as an avionics system. It is an extension of the PSE51 profile adding support for multiple processes. For this kind of system protection boundaries are required between different parts of the application, and processes are required in this profile for that purpose. A file system is not required.
- *PSE54*: *Multi-purpose real-time system profile*. This profile is intended for general-purpose computing systems running a mixture of applications with real-time and non-real-time requirements. It requires most of the POSIX functionality for general purpose systems and, in addition, most of the real-time services.

As a consequence of the approval of new POSIX real-time services, a project as been started by the IEEE to revise the POSIX.13 standard. In addition to extending the profiles, in this revision field experience with the implementation and use of the profiles will be used to make any modifications to the existing profiles.

Although the final outcome may change during the standardization process, the current draft of the revised standard [3] proposes adding most of the new real-time services to all of the profiles. One other major proposed change is to add file system support to the PSE53 profile, in recognition that many real-time systems now have the possibility of implementing file capabilities in flash memory, which has much less stringent mechanical and power requirements than those of rotating magnetic media.

For the minimal real-time profile the proposed additions are:

- Monotonic clock
- High-resolution sleep with specifiable clock
- Execution-time clocks and timers
- Sporadic Server scheduling algorithm
- Timed mutex lock operation

Each of these services has been implemented in our operating system MaRTE OS [4] (Minimal Real-Time Operating System for Embedded Applications), which follows the Minimal Real-Time POSIX.13 subset. This implementation has allowed us to obtain information about the complexity and overheads added to the operating system, thus making it possible to evaluate the impact of these new services in the Minimum Real-Time System Profile. The results of the evaluation appear in the following sections. All the time values have been measured on a 1.1 GHz Pentium III computer.

3. Monotonic Clock

3.1. Description

In the current description of the Minimum Real-Time System Profile there is only one clock defined: `CLOCK_REALTIME.` This clock represents the realtime clock for the system and measures the amount of time, in seconds and nanoseconds, since the Epoch (zero hours, zero minutes, zero seconds, on January 1, 1970 Coordinated Universal Time). It is settable by the application via the `clock_settime()` function, and as such it is subject to sudden changes that could severely affect timing behavior. Just imagine the effects of subtracting one second to the system time (perhaps to synchronize the clock with the official calendar time) while the system is waiting for its next time event that should happen within one millisecond; the next deadline would be missed by one second!

In recognition of the need of real-time applications to have their timing requirements not depend on the official calendar clock, the POSIX.1j standard introduced another clock (`CLOCK_MONOTONIC`), which is defined as a clock whose value cannot be set and which cannot have backward clock jumps. It measures the amount of time elapsed from an arbitrary but fixed time instant, such as the system boot time or the Epoch.

This new clock is very interesting for real-time applications because its use ensures that deadlines and timing requirements are not affected by changes in the system-wide clock time.

3.2. Implementation and Evaluation

Before adding the monotonic clock to MaRTE OS, its internal clock was already monotonic. The implementation of the time events follows the alarm clock model, in which instead of requiring a periodic interrupt, the hardware timer is programmed to interrupt the processor exactly at the time of expiration of the earliest time event. The instants of activation of all the time events, both absolute and relative, were expressed as an absolute value referred to that clock. In this way we were able to order them according to their degree of urgency in a single queue of time events.

The implementation of the internal monotonic clock in a PC architecture (which is currently the only platform for MaRTE OS; others are under development) depends on the underlying processor architecture. If a Pentium processor is available, the measurement of absolute time can be implemented using the “Time-Stamp Counter” (TSC) [7]. This counter (as implemented in the Pentium and P6 family processors) is a 64-bit counter that is set to zero following the hardware reset of the processor. We can use that internal clock directly as the system’s `CLOCK_MONOTONIC` clock.

In 80386 and 80486 processors, the system time is maintained by periodically programming counter zero of the “Programmable Interval Timer” (PIT) [8] so that after each expiration of the counter, the last programmed interval is added to the total time accumulated since the system boot. With this strategy, the value of the monotonic clock is obtained by adding the accumulated time since the system boot plus the current value of counter zero of the PIT.

In any of these implementations, the value of `CLOCK_REALTIME` is obtained by adding to the value obtained from the internal monotonic clock the time of system boot, perhaps modified by the changes to the system time introduced by the application via `clock_settime`().

Because an internal monotonic clock already existed in MaRTE OS, implementing the interface has required a very small modification of the kernel, which only affects the functions that may operate with several clocks, such as `clock_gettime`(), `clock_nanosleep`(), and `pthread_cond_timedwait`(). The modification consists of checking which clock is requested and, in the case of the realtime clock, adding or subtracting the system boot time, as appropriate. The number of new instructions is therefore very small, and has a minimal impact on the overhead of these functions, basically a simple check. The monotonic clock itself is faster than the realtime clock, because there is no need to adapt values representing a calendar time to the internal time format used by the system.

4. High Resolution Absolute Sleep Operation with Specifiable Clock

4.1. Description

Currently, the Minimum Real-Time System Profile includes an operation for a high resolution relative suspension of threads (`nanosleep()`). This operation always uses `CLOCK_REALTIME`.

In the current revision of the profiles a new suspension operation called `clock_nanosleep()` has been proposed. This operation allows a thread to suspend itself until a relative time interval has passed, or an absolute value of time is reached. In addition, the clock to be used for this suspension is specifiable. The absolute time option of this new version of the high resolution sleep allows applications to easily write periodic threads because with absolute time we can avoid the race condition that can occur if the thread is preempted between the thread’s read of the clock and the sleep operation. In addition, the monotonic clock may be used so that there are no dependencies on the application setting the calendar time.

4.2. Implementation and Evaluation

The implementation of this functionality only implies incorporating the `clock_nanosleep()` function in the kernel, and therefore there is no overhead impact for the rest of the services. The size of this function is 35 lines, and thus this represents a minimal impact on the size of the kernel.

Table 1 shows some performance metrics for the `clock_nanosleep`() operation, and their comparison with those of the relative `nanosleep`() function. The first four rows in the table show the time elapsed from the instant when a high priority thread suspends itself with one of these operations, until another lower priority thread that was ready starts executing. The differences between the two functions are due to the larger number of parameters and operation modes of the `clock_nanosleep`() function.

Table 1: Performance of the `clock_nanosleep()` function (Pentium III 1.1GHz)

Description	Time (μs)
Suspension using `nanosleep()`	0.89
Relative suspension using `clock_nanosleep()`	0.93
Absolute suspension using `clock_nanosleep(CLOCK_MONOTONIC)`	0.93
Absolute suspension using `clock_nanosleep(CLOCK_REALTIME)`	0.94
Wake up after suspension operation (`nanosleep()` or `clock_nanosleep()`)	0.75
Resolution of `nanosleep()`	1.0
Resolution of relative `clock_nanosleep()`	1.0
Resolution of absolute `clock_nanosleep(CLOCK_REALTIME)`	0.9
Resolution of absolute `clock_nanosleep(CLOCK_MONOTONIC)`	0.9

The fifth row in Table 1 shows an opposite situation, in which a low priority thread is preempted by a high priority thread that had been previously suspended with one of the high resolution sleep operations. In this case there is no difference between the two functions.

Finally, the last four rows in the table show the difference between the requested thread activation time and the real observed time at which the thread starts executing. These metrics include one context switch, and would include any possible jitter or delay introduced by the timer interrupt mechanism used to handle time events. As it can be seen, both of the high resolution sleep operations are extremely precise, within one microsecond, and with a negligible advantage for the absolute sleep operation.

5. Execution-Time Clocks and Timers

5.1. Description

The proposed revision of the POSIX.13 standard includes the execution-time clocks and timers in all the real-time profiles, including the minimal PSE51. When this functionality is supported in a system, a CPU-time clock is created for each thread (and for each process, if multiple processes are present). These clocks are managed using the standard POSIX interface for clocks and timers, so, it is possible to define timers based on those clocks.

This functionality is very useful for real-time applications, because execution-time timers can be used to detect the consumption of an excessive amount of execution time by a thread, allowing run-time detection of software errors or of errors made in the estimation of the worst-case execution times. Detecting when a thread exceeds the worst-case execution time assumed during the analysis phase is very important in robust time-critical systems, because if the assumptions are violated, the results of the schedulability analysis are no longer valid, and the system could miss its deadlines. Execution-time clocks allow detecting when an execution-time overrun occurs, and activating the appropriate error handling actions. They are also very useful for implementing at the application level various scheduling policies that require execution-time budgeting, such as the constant bandwidth server (CBS) [9].

5.2. Implementation and Evaluation

Execution-time clocks are implemented by accounting at each context switch for the execution time consumed by the running thread. This requires reading the monotonic clock and storing its value as the activation time of the new thread, for later accounting, and increasing the amount of execution time of the old thread by an interval equal to the difference between the current time and the activation time of that old thread.

In addition, because the application may create execution-time timers, there is a new kind of time event that expires when the execution time of a given thread has reached a specific value. In addition to supporting the timers, these events are used in MaRTE OS to implement the round robin and the sporadic server policies, as we will describe later.

Execution-time events are not queued in the regular time-events queue, to prevent multiple queueing and dequeueing operations that would be required each time the system switched to a different running thread. Instead, execution-time events are kept on a thread-by-thread basis. Because in a single processor only one thread is running, the system need only pay attention to the execution-time events of that particular thread. In the MaRTE implementation we have chosen to use a singly linked list for the execution-time events of each thread, ordered by execution time, because usually the number of such events per thread is very low (probably just one), and a singly linked list is faster than a priority queue for a small number of items.

Before the new thread is executed, the hardware timer is programmed to produce an interrupt at the time of the most urgent event in the system. This event is the most urgent between the head of the time-events queue, and the head of the execution-time-events queue of the running thread.

This implementation of clocks and execution-time timers has implied adding around 10 lines of code to the context switch routine (to measure consumed execution time) and another 50 lines added to the clock and timer operations. In addition, we added a list of execution-time events to each thread, which implied a few new fields in the thread control block, and no additional lines for the data structure, because it was already included for implementing other features of MaRTE OS.

MaRTE OS can be configured at kernel compile time to either support or not execution-time clocks and timers. If this functionality is configured out, no measurement of execution time is made, and so an application not interested in this functionality needs not pay its overhead. Table 2 shows the increase in the context switch time that enabling execution-time clocks and timers introduces in threads scheduled under the `SCHED_FIFO` scheduling policy. The context switch time was measured as the time interval elapsed between a thread invoking a yield operation, and the next thread starting to execute

Table 2: Impact of CPU-Time accounting in context switch performance (Pentium III, 1.1GHz)

CPU-time accounting enabled?	**Context Switch Time (μs)**
NO	0.42
YES	0.44

In summary, we have shown that the implementation of execution-time clocks and timers in MaRTE OS has a very small impact on the size of the kernel. The overhead associated with these services is very low, at least in modern architectures in which reading the clock is a very fast operation (like in the Pentium processors when the “Time Stamp Counter” is used).

The use of execution-time clocks and timers allows the application to detect and handle execution-time overruns. In our view, this feature is extremely important in today’s real-time systems in which execution times are very difficult to measure correctly, and the small overhead of the implementation is fully acceptable.

6. Sporadic Server Scheduling Policy

6.1. Description

When the realtime application environment profiles where specified, the POSIX standard only had two scheduling policies: FIFO within priorities (`SCHED_FIFO`) and round-robin within priorities (`SCHED_RR`). A a new scheduling policy was defined recently in the POSIX standard that implements the sporadic server scheduling algorithm (`SCHED_SPORADIC`)[5]. This policy can be used to process unbounded aperiodic events at the desired priority level, while making it possible to guarantee the timing requirements of lower priority threads. The sporadic server provides fast response times and makes systems with aperiodic events predictable. It is also useful to reduce the negative effects that input jitter has on the schedulability of lower priority threads [6].

The sporadic server scheduling policy assigns a limited amount of execution time to a given thread. The thread is allowed to use that amount of execution time at the desired priority level. Once the thread has consumed its available execution time, its priority is switched to a background level, lower than the priorities of any other threads with real-time requirements. Each portion of consumed execution time is replenished at a time equal to the activation time of that computation plus an interval called the replenishment period. In this way, the sporadic server guarantees a bandwidth for its thread equal to the initial execution-time capacity every replenishment period; and it also guarantees that the effects of that thread on lower priority threads are no worse than the effects of an equivalent periodic thread with an execution time and period respectively equal to the initial capacity and replenishment period.

The current proposed revision of the POSIX.13 standard proposes this scheduling policy for all the real-time profiles, and thus also for the Minimum PSE51 profile.

6.2. Implementation and Evaluation

The implementation of the sporadic server policy is rather complex, but part of its complexity comes from the need to compute and limit execution time, which is already available in MaRTE OS. Limiting the execution time of a thread running under the SCHED_SPORADIC policy is easy, by using an execution-time event similar to those described in Section 5 for the execution-time timers. The implementation of this policy has been unified with the round robin policy; in the latter, an execution-time event is used for determining the end of a time quantum. By integrating the treatment of these scheduling policies with the execution-time events, the complexity of the kernel is reduced.

An alternate solution for the execution-time events caused by the sporadic server and round robin policies would have been to include these events among the ordinary (non-execution-time) time events. This solution would imply adding the thread to the time-events queue each time a thread with one of these policies was made runnable, and dequeueing the event every time the thread was blocked or preempted. This would imply significantly increasing the worst-case execution time of the context switch operation.

For this reason we have chosen an integrated approach with the execution-time events.

For those threads scheduled under the SCHED_SPORADIC policy, in addition to limiting the execution time to the available capacity it is necessary to program the replenishment operations, to restore a portion of spent execution capacity at the appropriate time. Because these operations must occur at a particular absolute time not related with the execution time of any thread, they are enqueued in regular time-events queue.

The implementation in MaRTE OS of the sporadic server policy according to the design described above has required adding around 60 lines of code, of which 20 lines are shared with the round-robin policy and the execution-time timers. In addition, we have added 30 lines of code to handle the new scheduling parameters associated with the threads running under this policy.

MaRTE OS has a configuration parameter that allows the application developer to compile the kernel with or without sporadic server support. In this way applications not using the sporadic server policy don't have to pay the overhead associated with it. This configuration parameter makes it easy to determine the overhead effects that the scheduling policy has on the context switch times. Table 3 shows the context switch times between threads scheduled under the SCHED_FIFO policy with or without the sporadic server policy enabled. We can see that enabling the sporadic server policy has a minimal impact on the context switches when no threads are scheduled with the sporadic server policy. The effect is approximately the same as that for adding execution-time clocks and timers, because the actions to be performed by the context switch routine are basically the same in both cases.

If the application has threads running under the sporadic server policy, some context switches may be longer due to the execution of the code implementing the sporadic server rules; in particular, the exhaustion of the execution capacity

Table 3: Impact of the `SCHED_SPORADIC` policy on context switch performance (Pentium III, 1.1GHz)

SCHED_SPORADIC policy enabled?	Context Switch Time (μs)
NO	0.42
YES	0.44

and the replenishment operations increase the context switch times. However, as we can see in Table 4, the increase in the context switch time is always under the one microsecond range, and thus is fully acceptable, even for the most stringent applications. For example for a periodic thread running at a frequency of 1KHz the overhead of a double context switch for each period would not exceed 0.26% of its period.

In summary, we can conclude that the implementation of the sporadic server scheduling policy has a very small complexity, and that the overheads introduced by this policy as compared with those of the regular SCHED-FIFO policy are always under one microsecond, and therefore fully acceptable given the advantages obtained relative to the scheduling of aperiodic activities.

7. Timed Mutex Lock

7.1. Description

Mutexes are the objects defined in POSIX for the mutually exclusive synchronization of threads when accessing shared resources. The basic operations of a mutex are lock and unlock. Initially a mutex is unlocked. When a thread locks a mutex it becomes the owner of that mutex, and no other thread can lock the mutex until unlocked by the owner. The "lock" operation suspends the calling thread if the mutex is not available, i.e., if it is owned by some other thread.

Table 4: Context switch performance for `SCHED_SPORADIC` threads (Pentium III, 1.1GHz)

Thread leaving the processor	Thread taking the processor	Context Switch Time (μs)
`SCHED_FIFO` thread that invokes `sched_yield()`	`SCHED_SPORADIC` thread	0.5
`SCHED_SPORADIC` thread that invokes `sched_yield()`	`SCHED_FIFO` thread	0.45
`SCHED_SPORADIC` thread that invokes `sched_yield()`	`SCHED_SPORADIC` thread	0.47
`SCHED_SPORADIC` thread that reaches the limit imposed to its execution time	`SCHED_FIFO` thread	1.2
`SCHED_FIFO` thread	`SCHED_SPORADIC` thread that replenishes its execution capacity	1.3

Table 5. Performance of the `pthread_mutex_timedlock()` function (Pentium III, 1.1GHz)

Description	POSIX function used	Time (μs)
Lock a free mutex	`pthread_mutex_lock()`	0.15
	`pthread_mutex_timedlock()`	0.15
Attempt to lock an already locked mutex and subsequent context switch	`pthread_mutex_lock()`	0.5
	`pthread_mutex_timedlock()`	0.65
Lock and unlock a free mutex	`pthread_mutex_lock()` and `pthread_mutex_unlock()`	0.35
	`pthread_mutex_timedlock()` and `pthread_mutex_unlock()`	0.35
Unlock a mutex with a thread waiting on it via a `pthread_mutex_lock()` operation	`pthread_mutex_unlock()`	0.570
Unlock a mutex with a thread waiting on it via a `pthread_mutex_timedlock()` operation	`pthread_mutex_unlock()`	0.645
Timeout expiration and context switch	`pthread_mutex_timedlock()`	0.82

If by programming error or because of a software or hardware fault the owner does not unlock the mutex, the thread(s) awaiting at a lock operation would remain blocked forever. For this reason, the new POSIX realtime extensions defined a timed version of the mutex lock operation, in which an absolute timeout can be specified. This timeout is based on the CLOCK_REALTIME clock.

7.2. Implementation and Evaluation

The implementation of this service has been very simple, implying:

- A change to the unlock operation, to eliminate the timeout event if present.
- Adding the new function, `pthread_mutex_timedlock()`.
- Adding the code to execute the timeout, which has to eliminate the waiting thread from the mutex's queue.

No changes to other functions were necessary, except that we added a new field to the time-event data structure, to identify the mutex in which the thread is blocked. The treatment of the new time event is similar to the one implemented for `pthread_cond_timedwait()`. The total number of additional lines required to add the new service is 59.

The performance of the new service compared with the regular mutex lock operations is shown in Table 5, for mutexes with the priority ceiling synchronization protocol enabled. This protocol is called `PRIO_PROTECT` in POSIX and is used to avoid unbounded priority inversion effects. The first two rows show that locking a free mutex takes the same time regardless of the kind of operation used. If the mutex is locked and the function suspends the calling mutex, the time of `pthread_mutex_timedlock()` is longer, because in this case a new time event must be programmed. The results in rows three and four of the table show the time needed to execute the lock operation for this situation, measured until a context switch is made and a new thread starts executing. If we lock and then unlock a free mutex there is no timeout event, and thus there is no difference in performance between the two functions; nor there is a measurable difference before or after implementing the new timed mutex lock operation.

Rows 4 and 5 of Table 5 show that the unlock operation has more overhead in the case of unlocking a mutex with a timeout, because the time event needs to be eliminated from the queue of time events. But the difference is less than 80 ns.

The last row in the table shows the time required to perform a context switch between a low priority thread and a high priority thread that is activated because it's timeout had expired. We can see that this time is comparable with the time spent in the activation of a thread that suspended itself with the `nanosleep()` operation, which is 0.75μs.

In summary, adding the timed mutex lock operation did not cause any measurable overhead to the regular mutex operations. The overhead of the timed mutex operations themselves are similar to those of other timed operations. In addition, the complexity of adding this service to the kernel is very small.

8. Conclusions

We have implemented in our MaRTE OS some of the new real-time operating system services proposed for the Minimum real-time System Profile in the new revision of the POSIX.13 standard. The services implemented are the monotonic clock, the high resolution sleep operation with specifiable clock, execution-time clocks and timers, the sporadic server scheduling policy, and the timed mutex lock operation.

This implementation has allowed us obtaining information about the impact that their incorporation has on the operating system both in terms of size and overhead.

In the cases of the monotonic clock and the high resolution sleep, the impact is almost negligible.

The two services requiring measurement of execution time, i.e., execution-time clocks and timers and the sporadic server scheduling policy, have a very small overhead in the execution time of the context switch operation of regular threads, adding around 20 ns only. They also have a minimal impact in the size of the kernel, requiring the addition of only a few tens of lines. For threads using sporadic server scheduling the overheads are somehow higher, but they remain in the order of one microsecond. We believe that this very small overhead is an acceptable price to pay given the important increase in functionality and flexibility that the incorporation of these services implies.

Finally, the introduction of the timed mutex lock operation did not cause any measurable overhead to the regular mutex operations, and the overheads of the new service are similar to those of other timed operations.

As a consequence our view is that the concerns raised about the complexity and overhead of these services for small embedded systems have been addressed, and as a result of their evaluation our recommendation is to support their incorporation in the revised POSIX Minimal Realtime System Profile.

References

[1] POSIX.1 (2001). *IEEE Std 1003.1:2001. Standard for Information Technology -Portable Operating System Interface (POSIX).* The Institute of Electrical and Electronic Engineers, 2001.

[2] POSIX.13 (1998). *IEEE Std. 1003.13-1998. Information Technology -Standardized Application Environment Profile- POSIX Realtime Application Support (AEP).* The Institute of Electrical and Electronics Engineers, 1998

[3] IEEE Draft Standard P1003.13-Draft 2, *Draft Standard for Information Technology -Standardized Application Environment Profile- POSIX Realtime Application Support (AEP).* The Institute of Electrical and Electronics Engineers, 2002.

[4] M. Aldea and M. González. *"MaRTE OS: An Ada Kernel for Real-Time Embedded Applications"*. Proceedings of the International Conference on Reliable Software Technologies, Ada-Europe-2001, Leuven, Belgium, *Lecture Notes in Computer Science, LNCS 2043*, May, 2001.

[5] B. Sprunt, L. Sha and J.P. Lehoczky. "*Aperiodic Task Scheduling for Hard-Real-Time Systems*". The Journal of Real-Time Systems, Kluwer Academic Publishers, 1, pp. 27-60, 1989.

[6] J.J. Gutiérrez García and M. González Harbour. "*Increasing Schedulability in Distributed Hard Real-Time Systems*". Proceedings of 7th Euromicro Workshop on Real-Time Systems, IEEE Computer Society Press, pp. 99-106, June 1995.

[7] Intel. *Intel Architecture Software Developer's Manual. Vol. 3. System Programming*. (ftp://download.intel.nl/design/pentiumii/manuals/24319202.pdf).

[8] W.A. Triebel, *"The 80386DX Microprocessor. Hardware, Software, and Interfacing"*. Prentice-Hall International Editions, 1992.

[9] L. Abeni and G. Buttazzo. "Integrating Multimedia Applications in Hard Real-Time Systems". *Proceedings of the IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998