EARLY EXPERIENCE WITH AN IMPLEMENTATION OF THE POSIX.13 MINIMAL REAL-TIME OPERATING SYSTEM FOR EMBEDDED APPLICATIONS

Mario Aldea Rivas and Michael González Harbour

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

Abstract: Although the real-time POSIX operating system standards define system services for large systems, standard subsets have been defined that allow small implementations for embedded systems. In this paper we present the architecture and internal details of an implementation of the standardized POSIX Minimal Realtime System profile. This implementation constitutes a kernel for high-efficiency small embedded systems, under which applications with hard-real time requirements can be built.

Keywords: Real-Time, POSIX, Operating Systems, Embedded Systems

1. INTRODUCTION[1]

The family of Unix-based operating system standards called POSIX (ISO/IEC 9945-1:1996) includes real-time and threads interfaces that allow support for portable applications with real-time requirements. Although POSIX is a fairly large interface, standard subsets of POSIX have been defined in the IEEE 1003.13 standard (POSIX.13, 1998). The smallest of these subsets requires only a reduced set of the system services, which can be implemented as a small and very efficient kernel that can be used to effectively implement embedded systems with real-time requirements.

Although there are many implementations of real-time operating systems and kernels that are compliant to the POSIX standards, such as LYNX, HP-RT, VxWorks, QNX, etc., they are not available for most of the special-purpose platforms used in embedded systems, such as microcontrollers. In addition, they do not provide the source code and thus they cannot be used as a research vehicle for testing new concepts in thread scheduling and real-time services. The RTEMS real-time kernel (RTEMS, 1996) would be a good candidate for such purpose because its sources are available; however, although it offers a POSIX interface, its internal design was not done following the POSIX threads model, and thus the POSIX layer represents a source of inefficiency, than can be avoided in a kernel designed with the POSIX thread model from the beginning.

Consequently, in our research group we decided to design and implement a real-time kernel for embedded applications that could be used on different platforms, including microcontrollers, and that would follow the Minimal Real-Time POSIX.13 subset. This kernel will serve both as a vehicle for the development of real-time applications such as robot controllers, and as a research tool on which we can prototype new OS interfaces, such as user-defined real-time scheduling, interrupt control, etc.

This paper presents our early experience with the implementation of this kernel. In Section 2 we present an overview of the POSIX Minimal Real-time subset. In Section 3 we describe the objectives and basic requirements of our kernel. Section 4 describes its

[1]This work has been funded by the *Comisión Interministerial de Ciencia y Tecnología* of the Spanish Government under grant TIC99-1043-C03-03

general architecture, and Section 5 describes some of the most relevant aspects of the implementation of its different components. Section 6 describes the current status of the project, together with some performance metrics. Finally, Section 7 gives our conclusions and future work.

2. THE POSIX MINIMAL REAL-TIME SYSTEM SUBSET

POSIX is the acronym for Portable Operating System Interfaces. It is a family of standards based on the popular UNIX operating system. The POSIX standards define application program interfaces for the operating system services. Only the interfaces themselves, with their associated semantics, are defined; the standard contains no implementation details, which are left to the implementors to encourage innovation. Since 1996, the POSIX standard contains the real-time and threads extensions (ISO/IEC 9945-1: 1996), that allow developing portable applications with real-time requirements.

The POSIX base standards define all the system interfaces using the C language, but the POSIX family of standards also includes Ada bindings for these services, including the real-time and threads extensions (POSIX.5b, 1996). This allows us building real-time Ada applications running on a POSIX compliant operating system.

Although the complete set of POSIX services is useful to large real-time applications, it is too large for most embedded systems, which usually have tight memory requirements, may not have memory management capabilities, and may not have a secondary memory for implementing the UNIX file system. For these reasons the POSIX standard recognizes the need for the creation of subsets of the operating system services, yet standard subsets that allow portability of applications from one implementation to another.

The IEEE 1003.13 (POSIX.13, 1998) contains four real-time subsets, called application environment profiles. The smallest of these subsets, called the Minimal Realtime System Profile, is intended for small embedded applications. Among other things, it does not require support for multiple UNIX processes, nor support for a full-featured file system. These two simplifications by themselves take away from the OS most of the complexity of UNIX, and allow building a small efficient kernel for embedded real-time applications. The benefit is that an application conforming to this minimal profile can be ported to a larger real-time POSIX system. The main services provided in the Minimal Real-Time Profile are:

- *Threads*. They are the concurrency mechanism. Services include thread creation and termination, managing thread attributes and specific data, waiting for thread termination, etc.
- *Thread Priority Scheduling*. Priority preemptive scheduling for threads, with two different flavours for equal-priority threads: FIFO or round robin. Services include setting scheduling policies and parameters and yielding the processor to other equal-priority threads.
- *Thread Synchronization*. The synchronization mechanisms are mutexes (for mutual exclusion), condition variables (for signal & wait synchronization), and counting semaphores (for both). Mutexes include priority inheritance and priority ceiling protocols for avoiding priority inversion.
- *Signals*. They are used as a mechanism to notify the application of the occurrence of an event. Although traditional UNIX signals have been used to execute asynchronous signal handlers, the preferred approach in multi-threaded applications is to use signals in a synchronous way, by means of an `await_signal` operation. Services include the ability to mask signals, send signals and wait for the arrival of a signal.
- *Device I/O*. Although a general file system is not required in the Minimal Realtime System, basic read, write, open, and close operations are provided for using device drivers. The open operation is restricted to opening only existing files, defined at system configuration time and, therefore, it cannot create new regular files.
- *Time services*. Time services include the high resolution sleep operation plus clock and timer operations. Timers can be created by the user to measure time intervals and be notified by the system when such interval has elapsed, or when a given time has been reached. A signal is used as the notification mechanism.
- *Message passing*. Contains operations to use a message queue system, in which messages carry a priority field that is used to retrieve messages in priority order.
- *Configuration services*. Operations are provided to allow the application obtaining configuration information.
- *Dynamic memory management*. Although POSIX does not define an interface for dynamic memory support, it requires supporting the language-specific dynamic memory operations, such as `malloc` or `new`.

In addition to these services, the POSIX.13 standard requires support for some other services for upward compatibility of applications:

- *Memory locking*. These services allow preventing the unbounded memory access time caused by virtual memory implementations, by locking the address space of a realtime process into physical memory. Although most minimal realtime system implementations will not have virtual memory facilities, the interface is provided for upward compatibility to other POSIX systems. Applications are encouraged to use these services, even though in a system without virtual memory they will have a null implementation, since by default all the address space is locked in physical memory anyway.
- *File synchronization and synchronized I/O*. Again, in small embedded systems most I/O is unbuffered and thus synchronized by default. The application is encouraged to use the services though and request synchronized I/O, to preserve compatibility with other POSIX systems.
- *Shared memory objects*. They may be used to implement memory-mapped I/O, which is rather common in many embedded architectures. Although the memory may be directly accessible, applications are advised to perform memory-mapped I/O through shared memory objects, so that they can be ported to larger systems in which direct access to memory-mapped devices is forbidden to the application. Implementation of shared memory objects in a non-protected architecture where all memory is directly addressable is straightforward.

With the services described, a kernel that is compliant with the POSIX.13 Minimal Real-Time System Profile can be built with a very small size, and as a highly efficient implementation.

With the recent approval of the POSIX additional real-time extensions (POSIX.1d, 1999) and (POSIX.1j, 2000), it is forecasted that a revision of the real-time profiles will take place, to include the relevant services defined in these new real-time extensions. In order to anticipate some of this work, we have decided to include in our kernel those services from the new standards that we feel are most useful for embedded real-time applications. These services are:

- *Absolute high-resolution sleep*. Although periodic tasks can be created using timers, an absolute high-resolution sleep operation is simpler to use and more efficient, because it does not involve the use of signals.
- *Monotonic clock*. A clock whose value cannot be changed explicitly; it only changes monotonicaly with the passage of time. Such a clock is interesting to prevent the effects that setting the system clock has on the application scheduling.
- *Timeouts*. Bounded wait operations for mutexes, semaphores, and message queues.
- *Execution-time clocks and timers*. Execution-time clocks are useful for measuring the execution time of the threads. More important to real-time are execution-time timers, that allow detecting execution-time overruns. This is a very interesting feature for hard real-time, because the results of real-time schedulability analysis are only valid if the estimation of worst-case execution times are accurate. Execution-time timers enable us to validate this information on-line, and take corrective actions if an overrun occurs.
- *Sporadic server scheduling*. This scheduling policy allows processing aperiodic activities with low response times, while bounding their effect on lower priority threads, even in the presence of unbounded aperiodic requests. It is also useful for eliminating the negative effects that jitter has on the schedulability of lower priority threads.

3. OBJECTIVES AND BASIC REQUIREMENTS

The main objective is to develop a real-time kernel for embedded systems that conforms to the POSIX minimal real-time system profile in POSIX.13. In addition to the services defined in this profile, we plan to implement the services from the POSIX.1d and POSIX.1j newly approved standards that we mentioned above.

The applications that we plan for this kernel are industrial embedded systems, such as data acquisition systems and robot controllers. We also plan to use the kernel as a research tool for investigating in operating systems and scheduling mechanisms. In particular, we plan to implement new OS services that we feel are useful in many real-time applications: application-level interrupt management, and user-defined scheduling.

Based upon these objectives, the main requirements that we have formulated for our kernel are:

- Conformance to the POSIX.13 Minimum Real-Time System Profile, with the addition of some services from POSIX.1d and POSIX.1j
- Targeted for applications that are mostly static, with the number of threads and system resources well known at compile time. This allows these resources (i.e., threads, mutexes, thread stacks, number of priority levels, timers, etc.) to be preallocated at system configuration time, thus

saving a lot of time when the application requests creation of one of these objects.

- All services with bounded response times, for hard real-time performance.
- Non protected. No protection boundaries will be established between the application and the kernel. This means that a misbehaved application may corrupt kernel data, but this should be no problem in thoroughly tested static systems, like the targeted applications.
- Multiplatform. The kernel shall be able to run in multiple platforms, using cross-development tools.

For the initial version of our kernel, we have restricted the set of services required in POSIX.13, because some of these services are redundant and are thus not essential for applications. The main restrictions that we plan for the initial implementation are:

- No suspension inside signal handlers. Signals will be used in a synchronous way, through application threads acting as signal handlers, using the `await_signal` operation. Therefore, we do not need signal handlers for C applications. For Ada applications, we need signal handlers for the asynchronous select statement (asynchronous transfer of control), but these signal handlers do not suspend themselves. Therefore, we have decided to implement this restricted version of signal handlers, which is much simpler to implement than the general model described in POSIX. In our restricted implementation we use a special-purpose thread to serve all signal handlers, instead of executing the signal handler in the context of the thread to which the signal is delivered. This simplifies many of the kernel functions. For example, if a thread is waiting for a mutex, we don't need to eliminate it from the mutex queue while the signal handler is executing.
- No semaphores, because mutexes and condition variables are already a complete set of synchronization primitives. Semaphores are useful for synchronizing with a signal handler, but we are restricting them as well.

4. KERNEL ARCHITECTURE

The implementation language for our kernel is Ada 95, which we choose because it allows building applications more reliably than with other languages. We use the Gnat compiler which is integrated in the *gcc* compilation environment, that provides facilities for many targets, as well as cross compilation.

The kernel has a low-level abstract interface for accessing the hardware. The interface encapsulates operations for interrupt management, clock and timer management, and thread context switches. Its objective is to facilitate migration from one platform to another. Only the implementation of this hardware abstraction layer needs to be modified. For our initial platform (a Pentium PC) some of the functions of this hardware abstract interface come from a publicly available toolset called *OSKit* (Ford et al., 1997), which is intended to ease the low-level aspects of the development of an operating system. They are written in assembly and C language. We also use the facilities of *OSKit* for booting the application from a diskette or from the net

The kernel will be directly usable as the basis for the *Gnat* run-time system, and thus applications may be programmed in Ada using its language-specific tasks. The *Gnat* compiler is free software and provides the sources; this is extremely important for us because we need to replace part of the run time system, called GNARL (Giering and Baker, 1994).

The kernel interface has been developed according to the low level interface (GNULLI) defined between the Gnat run-time system and the POSIX underlaying implementation. This interface has been extended according to the POSIX.5b specification to cover the services required in POSIX.13.

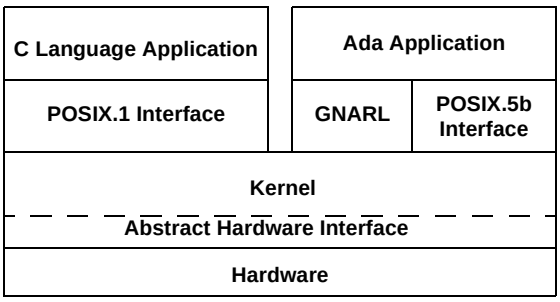

Fig. 1. Layers for applications in C and Ada

Because other application developers may wish to use other languages, we have developed a C-Language POSIX interface for C or C++ applications. In addition, we plan to develop in the future an implementation of the Java Virtual Machine for real time, now that the real-time specification for Java is being developed (RTSJ, 1999). This would allow applications written in Java to also use our kernel for embedded systems. Fig. 1 shows the layered architecture for applications written in these languages using our kernel.

Internally, the kernel has been divided into the following modules, as shown in Fig. 2:

- `Task_Operations`: basic thread management operations such as thread creation and finalization, priority changes, thread suspension, etc.

- `Mutexes`: operations for mutex management, conforming to the POSIX.5b interface, and including the priority protect protocol.
- `Signals`: Signal handling; currently only the functionality required by the GNARL run-time system is supported.
- `Condition_Variables`: operations for managing condition variables, with a POSIX.5b interface.
- `Timed_Events_Queue`: a queue with all the time-related events (such as sleep or delay expirations, timeouts, etc.), ordered by time, and with operations to insert and delete events form the queue.
- `Ready_Queue`: a queue with all the threads that are ready to execute, organized according to their priorities; it has operations to queue and dequeue threads from the queue.
- `Scheduler`: Implements the scheduling policies and the context switches. It is the only module that uses the operations defined in `Ready_Queue`.

5. IMPLEMENTATION DETAILS

Time Management. The system timer chosen is of the "alarm clock" kind, in which the timer is programmed at every scheduling point to expire at the next nearest scheduling point. This implementation reduces the overhead of the traditional "ticker" or periodic timer, and reduces the jitter in determining the scheduling points, provided that the underlaying timer has sufficient resolution. The clock resolution in the PC architecture that we have used as our first platform is 0.838 μs.

Ready Thread Queue. This is one of the most fundamental objects in the kernel, because its performance is crucial to the overall performance of the system. For this reason, the queuing and dequeuing operations must be extremely fast, and with bounded execution time. We have tested different implementations of priority queues for numbers of threads between 20 and 50, which we estimate typical for the kind of embedded applications that we target. We have focused on systems in which the number of threads per each priority level is very low, because with normal Rate Monotonic or Deadline Monotonic scheduling each thread usually has a distinct priority. As a result of these tests, we have chosen an array of ordered singly linked lists, one for each priority level, together with a map of bits that indicates whether there is an active thread at any given priority. This map of bits can be tested in a very short time (typically one instruction) to determine the highest active priority level.

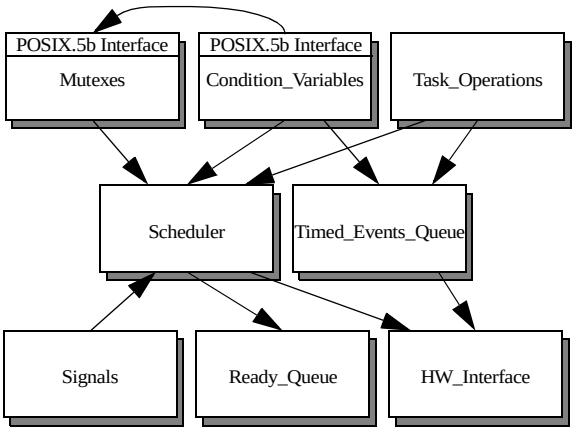

Fig. 2. Internal architecture of the kernel

Time Events Queue. This is another structure that is crucial to the performance of the overall system. We cannot use the same priority queue implementation as for the ready queue, because this queue is ordered by time values, and the time may have very many different values. For this queue we have chosen the heapform heap structure, which provided the best worst-case results for a number of time events roughly similar to the number of threads, between 20 and 50.

Signals. As we mentioned above, we have not implemented in this first version general signal handlers, because signal handlers are not allowed to suspend. Therefore, the signals used represent a simple event mechanism. The POSIX realtime signals behavior has been adopted for all signals; in this behavior signals are queued, and they may carry an additional field of information.

Mutexes and Priority Inversion Avoidance. Each thread contains in its thread control block the list of mutexes that it owns, in order to keep track of priority ceilings that need to be inherited. In turn, each mutex has a queue with the threads that are waiting on that mutex. Despite what would seem natural, the mutex queues are not implemented as priority queues, although they behave as such as required by the POSIX specification. The reason is that we assume that for real-time threads the priority protection protocol will be used and, under this protocol, if threads do not block themselves while holding a mutex, threads will always find their mutexes unlocked and therefore no thread will be queued in the mutex queue as a consequence of normal mutual exclusion. The queue is only provided for the case when a "broadcast" operation is invoked for a condition variable in conjunction with a mutex, of for the rare case in which a thread would block while holding a mutex. By not providing a priority queue, we avoid having to reorder it whenever priorities change.

We have optimized the priority protection mechanism for mutexes by implementing a “deferred priority change”. In this mechanism, when a thread becomes the owner of a mutex and thus inherits its ceiling priority, the priority is not immediately raised, but a flag is set to indicate that the priority change is pending. In most of the cases, the critical section during which the thread holds the lock is very short, and thus the priority is returned back to its normal level very soon, with no other thread being scheduled in between. In that case, the deferred priority change flag is reset and nothing else needs to be done. If indeed a scheduling point occurs after setting the flag, then the scheduler would check the flag and make the required priority change. The effect of this optimization is that in most cases we save two full priority changes, and consequently the average-case response time for the priority protection protocol is extremely low, similar to that of the priority inheritance protocol.

Condition Variables. Each condition variable needs a queue of waiting threads. If the number of threads simultaneously waiting is under four, a simple linked list is the best implementation for the queue. However, for a larger number of threads, a priority queue like the one used for the ready queue (with a linked list for each priority, and a priority bit map) is faster. This is the implementation that we have chosen.

One possible implementation of the “Signal” and “Broadcast” operations for condition variables would be to put the activated thread or threads into the ready queue, and let them “fight” for the associated mutex. Although this is the simplest implementation, we have chosen to complicate the implementation to make it more efficient. As a result, we check the priorities of the set of activated threads; within this set, and if the mutex is not locked, the thread with the highest priority is made the owner of the mutex, and put into the ready queue, while the other threads, if any, are inserted in the mutex wait queue. If the mutex was already locked, all threads are put in the mutex wait queue.

Dynamic Memory Management. Current implementation is very simple, and is optimized for real-time threads that make the allocations at initialization time, and then do not release the allocated memory. For each allocation (i.e., `malloc`), a consecutive block of memory is reserved. The `free` operation has no effect. Another possibility would be to use the Buddy algorithm for memory allocation and deallocation (Rusling, 1999), which has a bounded response time.

6. PERFORMANCE METRICS

The implementation of the kernel is now complete with the restrictions mentioned in Section 4, for a bare Pentium-PC platform. We have replaced the implementation of the low level GNULLI interface in the Gnat run time system by our kernel, so we can now run Ada tasks on top of it. C language applications can also run on top of the POSIX interface.

Table 1 shows performance metrics for some of the most important services, measured on a Pentium III at 550 MHz. Context switch tests have been performed with a large number of tasks (60 tasks), but only one task per priority.

Table 1. Performance of some kernel services

Service	Time (µs)
Context switch, after `yield` operation, low priority thread	0.675
Context switch, after `yield` operation, high priority thread	0.668
Read the clock	3.100
Send signal followed by context switch and await signal	0.843
Send signal followed by context switch and await signal, with deferred priority change	1.213
Mutex lock followed by unlock (with deferred priority change)	0.433
Signal a condition variable on which a high priority thread is waiting, followed by context switch and end of condition wait call	1.410
Minimum Ada rendezvous, including two context switches	6.7
Two ada rendezvous, passing an integer from a producer task through a buffer task to a consumer task	14.8

The number of lines of our implementation is around 4500, which gives an idea that the POSIX minimal real-time profile is really suited for embedded systems.

7. CONCLUSIONS AND FUTURE WORK

With the kernel described in this paper we are able to develop real-time embedded applications running on a bare PC. More important, we can use this kernel as a vehicle for research in new scheduling mechanisms for real-time.

In order to complete our work we need to eliminate the restrictions imposed, and we need to finish the implementation of newly approved POSIX services such as the sporadic server or execution time clocks

and timers. In addition, we need to port the implementation to other platforms, such as microcontrollers, and Power PC boards.

REFERENCES

Ada (1995). *International Standard ISO/IEC 8652:1995, Information Technology, Programming Languages, Ada Reference Manual.*

Ford, B., G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. (1997). The Flux OSKit: a Substrate for OS and Language Research. *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Saint Malo, France (http://www.cs.utah.edu/flux/oskit)

Giering, E.W. and T.P. Baker (1994). The GNU Ada Runtime Library (GNARL): Design and Implementation. *Wadas'94 Proceedings*.

ISO/IEC 9945-1 (1996). *ISO/IEC Standard 9945-1:1996. Information Technology -Portable Operating System Interface (POSIX)- Part 1: System Application Program Interface (API) [C Language]*. Institute of Electrical and electronic Engineers.

POSIX.1d (1999). *IEEE Std. 1003.d-1999. Information Technology -Portable Operating System Interface (POSIX)- Part 1: System Application Program Interface (API) Amendment: Additional Realtime Extensions [C Language]*. The Institute of Electrical and Electronics Engineers.

POSIX.1j (2000). *IEEE Std. 1003.j-2000. Information Technology -Portable Operating System Interface (POSIX)- Part 1: System Application Program Interface (API) Amendment: Advanced Realtime Extensions [C Language]*. The Institute of Electrical and Electronics Engineers.

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.

POSIX.5b (1996). *IEEE Std 1003.5b-1996, Information Technology—POSIX Ada Language Interfaces—Part 1: Binding for System Application Program Interface (API)—Amendment 1: Realtime Extensions.* The Institute of Electrical and Engineering Electronics.

Lehoczky, J. and S. Ramos-Thuel (1992). An optimal algorithm for scheduling soft aperiodic tasks in fixed-priority preemptive systems. *Proceedings of the 13th IEEE Real-Time System Symposium*, pages 110 123, Phoenix, Arizona.

RTSJ (1999). The Real-Time for Java Experts Group, "Real-Time Specification for Java", Version 0.8.2, November 1999 (http://www.rtj.org).

Rusling, D.A. (1999). *The Linux Kernel*, Version 0.8-3 (http://www.linuxhq.com/guides/TLK/tlk.html).

Sprunt, B., L. Sha, and J.P. Lehoczky (1989). Aperiodic Task Scheduling for Hard Real-Time Systems. *The Journal of Real-Time Systems*, Vol. 1, pages 27-60.