Marte OS: An Ada Kernel for Real-Time Embedded Applications

Conference Paper *in* Lecture Notes in Computer Science · May 2001

DOI: 10.1007/3-540-45136-6_24 · Source: CiteSeer

2 authors:

Mario Aldea-Rivas
Universidad de Cantabria
47 PUBLICATIONS **340** CITATIONS
SEE PROFILE

Michael González Harbour
Universidad de Cantabria
175 PUBLICATIONS **3,342** CITATIONS
SEE PROFILE

MaRTE OS: An Ada Kernel for Real-Time Embedded Applications

Mario Aldea Rivas and Michael González Harbour

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

Abstract: MaRTE OS (Minimal Real-Time Operating System for Embedded Applications) is a real-time kernel for embedded applications that follows the Minimal Real-Time POSIX.13 subset, 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. Details on the architecture and implementation of the kernel are described, together with some performance metrics.

Keywords: Real-Time Systems, Kernel, Operating System, Embedded Systems, Ada 95, POSIX.

1 Introduction[1]

MaRTE OS (Minimal Real-Time Operating System for Embedded Applications) is a real-time kernel for embedded applications that follows the Minimal Real-Time POSIX.13 subset, providing both the C- and Ada-language POSIX interfaces. Although POSIX [3] is a fairly large interface, standard subsets of POSIX have been defined in the IEEE 1003.13 standard [6]. 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.

Most existing POSIX real-time operating systems are commercial systems that do not provide their source code. Open-source implementations exist, like RTEMS [12], but because they were not built following the POSIX model from the beginning they offer POSIX just as an external interface layer. Another popular open-source implementation is RT-Linux [14]. Although this implementation currently offers a partial Minimal Real-Time POSIX.13 interface, it is not suitable for embedded systems, because it requires support for a full Linux operating system.

For these reasons, our research group decided to design and implement a real-time kernel for embedded applications that could be used on different platforms, including

1. This work has been funded by the *Comisión Interministerial de Ciencia y Tecnología* of the Spanish Government under grants TIC99-1043-C03-03 and 1FD 1997-1799 (TAP)

microcontrollers, and that would follow the Minimal Real-Time POSIX.13 subset. This kernel, called MaRTE OS, is usable 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. In addition, it serves as a proof that a POSIX operating system can be implemented in Ada, while providing a C-language interface that allows multithreaded C applications to run on top of it.

This paper presents an overview of the architecture and internal details of MaRTE OS. In Section 2 we describe the objectives and basic requirements of our kernel. Section 3 describes its general architecture, and Section 4 describes some of the most relevant aspects of the implementation of its different components. Section 5 describes some performance metrics. Finally, Section 6 describes the current status of the project and gives our conclusions and future work.

2 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 specified in POSIX.13. In addition to the services defined in this profile, we have implemented some services from the POSIX.1d [4] and POSIX.1j [5] newly approved standards, which we think are very important for the kind of real-time applications at which MaRTE OS is targeted. Among these services are execution-time clocks and timers, and the absolute high resolution sleep.

The applications that we plan for this kernel are industrial embedded systems, such as data acquisition systems and robot controllers. We also plan on using the kernel as a research tool for investigating in operating systems and scheduling mechanisms. Based upon these objectives, the main requirements that we have formulated for our kernel are:

- 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. For efficiency purposes, 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 implemented for multiple target platforms, using cross-development tools.

- Multilanguage: The kernel shall support Ada and C as well as mixed Ada/C applications. In the future, addition of a Java virtual machine will be explored, to allow also real-time Java applications [8].

3 Kernel Architecture

MaRTE OS allows software cross-development of both Ada and C applications using the GNU compilers `gnat` and `gcc`. Most of its code is written in Ada with some C and assembler parts. Multithreaded C applications can make use of the Ada kernel through a C-language interface that has been created on top it.

The kernel has a low-level abstract interface for accessing the hardware. This 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 PC) some of the functions of this hardware abstract interface come from a publicly available toolset called *OSKit* [1], 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 is 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, including its run-time system called GNARL [2]; this is extremely important for us because we need to modify part of that run time system to adapt it to our kernel.

The `gnat` distribution that we have adapted is the version developed for Linux, and compiled to use the Florida State University (FSU) threads, which are a library implementation of the POSIX threads. Adapting the run-time system has implied modifying part of the GNARL lower level layer, called GNU Low-Level (GNULL). In particular, we have modified packages `System.Os_Primitives`, `System.OS_Interface`, and `System.Task_Primitives.Operations`, which define the interface between GNARL and the POSIX interface provided by the FSU threads and the Linux operating system, now replaced by the MaRTE kernel.

The modifications made are very minor, and mainly consist of type conversions between the internal types used by `gnat` (some of which are private), and their equivalent types in MaRTE. When a new version of the compiler is available, if we wish to migrate MaRTE to the new version, a careful check needs to be made to make sure that the modifications are correctly applied, and that the changes introduced in the new version are not incompatible with them. But because the modifications are very minor, if the changes in the compiler itself are not significant this migration process takes just a few hours, as we have experienced when migrating from `gnat` 3.12 to `gnat` 3.13.

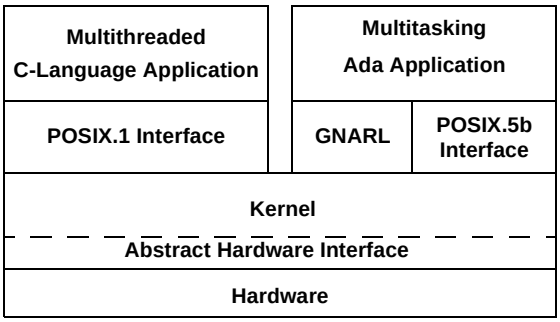

Fig. 1. Layers for applications in C and Ada

The kernel interface has been developed mainly according to the POSIX C-language interface, instead of the POSIX.5b specification [7], even though it is written in Ada. There were two reasons behind this decision. One was that GNULL is layered on top of the POSIX C interfaces. The other reason is that, because we had to provide both the Ada and C-language interfaces, we were not able to use exceptions inside the kernel, because C programs running on top of it would not know how to handle them. Therefore, the kernel interface uses only function return values to notify errors. The POSIX-Ada interfaces are implemented using an additional layer, that maps those interfaces to MaRTE , and raises the appropriate exceptions to the Ada applications. Fig. 1 shows the layered architecture for applications using our kernel and written in the Ada and C languages.

MaRTE OS works in a cross development environment. The host computer is a Linux PC with the `gnat` and `gcc` compilers, and the `gdb` debugger. The target platform is any 386 PC or higher, with a floppy disk (or equivalent) for booting the application, but not requiring a hard disk. An ethernet link is recommended for speeding the uploading of the application, and a serial line is required if remote debugging is desired. Fig. 2 shows the architecture of the cross-development platform. The *bootp* and *nfs* protocols are used to remotely boot an application stored in the host, through the ethernet link. The final application can be downloaded directly from the floppy (or

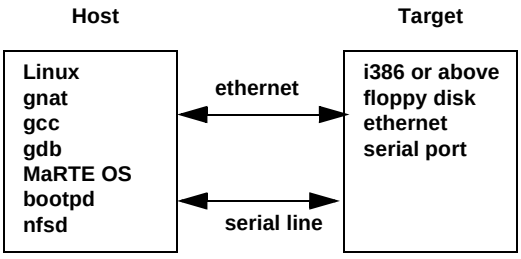

Fig. 2. Cross-Development Platform

a functionally equivalent device, such as a flash memory), without requiring the net, nor the serial line.

4 Implementation

Currently MaRTE OS implements the following POSIX functionalities. They cover the POSIX.13 minimal profile, plus some extensions from POSIX.1j and POSIX.1d:

- *POSIX Thread Management*: thread creation, finalization, attributes, etc. All threads and thread stacks are preallocated, so thread creation is extremely fast. The number of threads and the sizes of the stacks are configurable.
- *Priority Scheduling*. The two preemptive priority scheduling policies required by POSIX are implemented: FIFO within priorities, and round robin within priorities.
- *Mutexes*. For mutually exclusive synchronization among threads. Both priority inheritance and immediate priority ceilings are implemented for avoiding unbounded priority inversions.
- *Condition Variables*. For condition wait synchronization.
- *Signals*. They are the basic event notification mechanism. In particular, this mechanism is used by the timers to notify about timer expirations.
- *Clocks*. The POSIX real-time clock is implemented, for measuring time.
- *Timers*. They are software objects that measure time intervals or detect when a clock reaches a given time.
- *Execution-Time Clocks and Timers*. They enable CPU-time accounting and budgeting. This is an extremely interesting feature for detecting and limiting CPU-time overruns in real-time applications, that would make the results of real-time schedulability analysis invalid.
- *Console I/O*. For I/O using the PC console. Other I/O drivers will be provided in the future.
- *Time Services*: Thread suspension with absolute and relative high-resolution delays. Absolute delays are not yet part of POSIX.13, but were included because they are extremely useful; otherwise, a heavier-weight timer needs to be used.
- *Dynamic Memory Management*. Although not part of the POSIX interfaces, this functionality is required by the programming languages.

These POSIX functionalities support the whole `gnat` run time library, so the current MaRTE OS implementation allows running complex C and Ada applications with the restriction that they do not use a file system. This limitation is imposed by the Minimal Real-Time POSIX.13 subset.

4.1. Implementation Details

Implementation details on some of the most relevant services are given next.

- *Mutexes*. Each thread contains in its thread control block the list of mutexes that it owns, in order to keep track of priorities or 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. Under the immediate priority ceiling protocol, usually no thread will be queued in the mutex queue as a consequence of normal mutual exclusion. 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 (with their associated queue reordering), 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 is faster. The implementation chosen has a priority queue built with a linked list for each priority level, and a priority bit map to rapidly determine the highest non-empty priority level. To make the implementation of the "Signal" and "Broadcast" operations for condition variables faster, 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.

- *Signals*. Suspension inside a signal handler has been restricted to make signals more efficient. If a signal handler was allowed to suspend itself, and the interrupted thread was already suspended at some other OS service, the thread would be suspended twice, in different places. This makes the implementation very complex. Since we can use a user-defined thread to handle signals, such complexity for signal handlers is not really justified. With the proposed restriction, we can use a single special-purpose thread to execute all signal handlers, and the

implementation becomes very simple and efficient. This restriction is being proposed for the next revision of POSIX.13.

- *Dynamic Memory Management*. The 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 [9], which has a bounded response time.

In addition to these application-visible services, there are other internal services in the kernel:

- *Time services*. 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 underlying timer has sufficient resolution. Details on the low-level implementation of the clocks are given in Subsection 4.2.

- *Ready 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.

- *Timed 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 timed events roughly similar to the number of threads, between 20 and 50.

- *Execution-Time Events Queue*. Each thread has a queue of pending execution-time events which can represent the expiration of execution-time timers, or the expiration of their round robin quantum in case the thread is scheduled using the round robin policy. This queue is implemented as a singly-linked list, ordered by execution time, because the number of elements in it is usually very small. When a thread is made a running thread, the implementation calculates the absolute time at which the execution-time event found at the head of the queue should expire.

The hardware timer is then programmed to generate an interrupt at the smallest of that time and the time associated with the head of the Timed Events Queue. When the thread is blocked or preempted, the consumed execution time is added to its registered execution time.

4.2. Low-Level Timing Services

The implementation of clocks and timing services in MaRTE OS depends on the underlying processor architecture. If the processor is an 80386 or 80486, the Programmable Interval Timer (PIT) is used [13], both for the clocks (i.e., for measuring absolute or system time) and timing services (i.e., those requiring the generation of a timed event, at the requested time). The PIT is a standard device in the PC architecture that has three 16-bit counters driven through a hardware clock signal of 838.1 ns period. The main problem with the PIT is that its registers are accessed through the old I/O bus in the PC architecture, which makes accessing any of these registers a very slow operation, typically taking several microseconds.

The strategy used for programming the PIT is the same used in RT-Linux [11]. Counter 0 is used both to generate the timer interrupts required by the timing services, and to implement the clocks used in the OS to measure time. For this purpose, if an interrupt is required to occur within the next 50 ms, the counter is programmed with that interval; otherwise, if there are no urgent timer interrupts required, Counter 0 is programmed to generate an interrupt after 50 ms. After each interrupt occurs, the programmed interval is added to the total time count, and the counter is reprogrammed. The time required to reprogram the counter is measured using Counter 2 (which is commonly used in PCs to produce sounds of different tones through the speaker). This time is also added to the total time count. In this way, we make sure that the delays caused by reprogramming the counters do not cause a cumulative error in the measurement of time for the system clock.

With this strategy, an invocation of the get-time operation to obtain the current time consists of adding the total system time to the current value of Counter 0. The total system time variable is initialized at boot time with the value read from the PC's Real-Time Clock (RTC). The resolution of the clock implemented with this strategy is the same as for the underlying counters (838.1 ns).

If a Pentium processor is available, the measurement of absolute time can be implemented using the time-stamp counter (TSC). 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. Following reset, the counter is incremented every processor clock cycle, even when the processor is halted by the HLT instruction or the external STPCLK# pin [10]. Reading the value of this counter requires only a single machine instruction and, because this counter is internal to the processor and the I/O bus is not used, the operation is very fast (89 ns in a Pentium III at 550 MHz).

The system time is obtained by adding the initial boot time of MaRTE OS (read from the Real-Time Clock) to the number of cycles counted by the TSC at the present time minus the number of cycles already counted at boot time, and multiplied by the corresponding conversion factor. The clock resolution with this strategy is 1.8 ns in a Pentium III at 550 MHz.

In this implementation, timer interrupts are still generated with the PIT's Counter 0. In this case there is no need to use Counter 2 to measure reprogramming times, because the current time is measured with the TSC. Because Counter 0 cannot be programmed to measure intervals greater than 50 ms, if the next timed event is more that 50 ms away from the current time, one or more intermediate activations are programmed. With this strategy, we still have to pay the price of using the PC I/O bus to access the PIT counter.

For P6 processors (Pentium II or higher) the overhead can be greatly diminished by using the timer included in the Advanced Programmable Interrupt Controller (Local APIC). The local APIC is included in all P6 family processors. Although its main function is the dispatching of interrupts, it also contains a 32-bit programmable timer for use by the local processor whose time base is derived from the processor's bus clock. It can be configured to interrupt the local processor with an arbitrary vector. The frequency of the processor's bus clock in the Pentium III-500 MHz that we have used for testing is 100 MHz, and thus the timers in this machine have a resolution of 10 ns.

The local APIC is disabled after the hardware reset of the processor. Enabling the APIC requires two steps: first, a bit needs to be set in the model-specific register of the APIC [10], which enables access to the APIC registers; then, the APIC must be enabled explicitly by writing the appropriate value in its Spurious-Interrupt Vector Register.

Access to the local APIC registers is achieved by reading from and writing to specific memory addresses. With the APIC enabled, accesses to those memory addresses are mapped onto the APIC registers. Read and write access times for those registers are very short. For example writing to the Initial Count Register of the APIC Timer takes 62 ns in our Pentium III at 550MHz.

Table 1 shows some performance metrics comparing the three implementations of the time services. It can be seen that the overheads of using the local APIC are at least an order of magnitude less than when using the PIT.

Table 1. Comparison of the overheads of the time services implementations (ns)

Operation	PIT	TSC+PIT	TSC+APIC
Get time	3100	105	105
Program timer	12900	3000	324

5 Performance Metrics

Table 2 shows performance metrics for some of the most important services, measured on a Pentium III at 550 MHz.

Table 2. Performance metrics

Service	Time (µs)
Context switch, after `yield` operation, low priority thread	0.748
Context switch, after `yield` operation, high priority thread	0.734
Read the clock	0.089
Send signal followed by context switch and await signal	0.927
Send signal followed by context switch and await signal, with deferred priority change	1.316
Mutex lock followed by unlock (with deferred priority change)	0.399
Signal a condition variable on which a high priority thread is waiting, followed by context switch and end of condition wait call	1.262
Minimum Ada rendezvous, including two context switches	6.8
Two ada rendezvous, passing an integer from a producer task through a buffer task to a consumer task	15.1
Relative `delay` operation, until lower priority task starts running, including one context switch	5.0
Absolute `delay until` operation, until lower priority task starts running, including one context switch	5.0
Wake up from a relative `delay` operation, including one context switch	8.7
Wake up from an absolute `delay until` operation, including one context switch	8.8
Delay resolution (difference between requested delay and actual delay)	11.0
Minimum period thread (C program using `nanosleep`)	7.9 (126 KHz)
Minimum period task (Ada program using `delay`)	11.5 (87 KHz)

The number of lines of our implementation is around 10300, which gives an idea that the POSIX minimal real-time profile is really suited for embedded systems.

6 Conclusions and Further Work

With the kernel described in this paper we are able to develop real-time embedded applications running on a bare PC. These applications can be written in Ada, C, or both languages. More important, we can use this kernel as a vehicle for research in new

scheduling mechanisms for real-time, and also as a teaching platform for real-time operating systems and programming.

The implementation of the kernel is now complete for a bare Pentium-PC platform. The next steps in the development of this kernel will be to add services for managing application-defined scheduling policies, application-defined interrupt management, network communications (ethernet and CAN bus), and additional I/O drivers. In addition, we need to port the implementation to other platforms, such as microcontrollers, and Power PC boards.

MaRTE OS is available under the GNU General Public License, and can be found at:
`http://ctrpc17.ctr.unican.es/marte.html`

References

[1] 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)

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

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

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

[11] Real-Time Linux operating system web page (http://luz.cs.nmt.edu/~rtlinux)

[12] “Real-Time Executive for Multiprocessor Systems: Reference Manual”. U.S. Army Missile Command, redstone Arsenal, Alabama, USA, January 1996.

[13] Triebel W.A., “The 80386DX Microprocessor”, Prentice Hall, 1992.

[14] Yodaiken V., “An RT-Linux Manifesto”. Proceedings of the 5th Linux Expo, Raleigh, North Carolina, USA, May 1999.