# New Real-Time Extensions to the MINIX operating system

**Pablo J. Rogina** **Gabriel Wainer**

{pr6a,gabrielw}@dc.uba.ar

*Departamento de Computación*
*Facultad de Ciencias Exactas y Naturales*
*Universidad de Buenos Aires*
*Pabellón I - Ciudad Universitaria*
*Buenos Aires (1428) – ARGENTINA*

## ABSTRACT

The present work presents the inclusion of several real-time services provided in real-time operating systems, and incorporates these features in a recent version of the MINIX operating system. The most recent extensions allow the incorporation of fault tolerant schemes. The services are being used in a bottle-filling plant prototype to test real-time capabilities of the operating system.

**Keywords:** Operating systems, Real-time systems, Scheduling Algorithms, Fault Tolerance.

## 1. INTRODUCTION

**Real-time systems** are those systems in which the correctness depends not only on the results obtained, but also on the time at which these results are produced. These timing constraints are usually attached to processes (or tasks).

Real-time systems span from microcontrollers in automobile engines to very complex applications, such as aircraft flight control or process control in manufacturing plants. Nonetheless, a real-time system consists generally of a control system and a controlled system. Information about the environment is provided via sensors, and the system can in turn modify the state of the environment through actuators. Failure in meeting the tasks' deadlines can lead to catastrophic consequences.

As many other computer applications, real-time systems are usually built by using the services offered by an operating system. In this case, the services provided should be slightly different than the case for traditional applications. It should provide basic support for predictability, satisfaction of real-time constraints, fault tolerance and integration between time-constrained resources and scheduling.

Existing real-time operating systems (RTOS) can be divided in two categories:

- Systems implemented using somewhat stripped down and optimized (or specialized) versions of conventional timesharing OS
- Systems starting from scratch, focusing on predictability as a key design feature.

Research projects falling in the first category include RT Mach [1]; RT-Linux [2] and KURT [3]. Operating systems like Spring [4], Maruti [5] and YARTOS [6] were developed using the second approach. Though several commercially available systems, including LynxOS [7] and QNX [8], offer real-time performance and services to applications, they are too costly and proprietary to be used by research or academic institutions.

Task **scheduling** in multitasking systems has been extensively studied in the operating system literature. Nevertheless, the traditional scheduling techniques used in general purpose systems (e.g. FIFO, Shortest Job First, Round Robin, etc.) are not adequate to be used in time constrained systems. These scheduling policies attempt to reduce certain performance metrics (i.e., the average response time), and do not deal with the timing constraints of the processes to be scheduled.

On the other hand, scheduling policies for real-time systems need to guarantee that tasks will meet their deadlines in all circumstances. Such a set of tasks is called *schedulable*, with each task having a *predictable* behavior. Scheduling algorithms can be divided in two major models: preemptive, and non-preemptive. The first one assume that any task can be interrupted during its execution, while non-preemptive algorithms do not allow a running task to be interrupted.

Most scheduling algorithms divide the schedulable tasks into two different classes: periodic and aperiodic (sporadic). The periodic tasks must run repeatedly, and within fixed times (known as period). The aperiodic tasks run sporadically, and only once, when they are invoked.

Two well-known policies are broadly accepted for Real-Time scheduling: RMS (Rate Monotonic Scheduling) was shown to be optimal for scheduling fixed priority task sets. In dynamic priority systems, using EDF (Earliest Deadline First) policy, full processor utilization can be achieved. Real-time scheduling algorithms are a field of continuing research.

Taking this base into account, the present project shows the results obtained building a new version of an extended Real-Time operating system. The Minix 2.0 [9] operating system was taken as a base, and it was extended it with several real-time services. The most important include task management capabilities (both for periodic an aperiodic tasks), real-time scheduling algorithms; new device drivers allowing A/D conversion, and improved fault tolerance features, specially, robust sensing algorithms incorporated inside the kernel.

The rest of this work is divided as follows: Section 2 describes the MINIX OS and the real-time extensions done to that operating system. Each added feature is presented and explained in detail. Section 3 is devoted to introduce current applications using the modified OS, while future work possibilities are listed in Section 4.

## 2. REAL-TIME EXTENSIONS TO MINIX

MINIX [9] (name that stands for Mini-UNIX), is a complete, timesharing, multitasking operating system. Inspired by UNIX, it was written from scratch by A. Tannenbaum. Though it is copyrighted, the source has been made widely available to universities for study and research in computer science courses.

The work presented in [10] showed the results obtained in a research project devoted to use MINIX to implement real-time scheduling. Several changes was made to source code of the kernel, in order to provide the user with a set of system calls to create and manage tasks, both periodic or aperiodic.

The project was devoted to provide programming facilities to develop hard real-time software. Under the changed MINIX OS, the programmer was allowed to define timing constraints for the tasks, letting the OS to execute them in a timely fashion. In this way, productivity, security and development costs can be improved.

Several real-time services were added. First, RM and EDF scheduling were included. These strategies were later combined with other traditional strategies, such as Least Laxity First, Least Slack First and Deadline Monotonic. At present new flexible schedulers are being included.

To allow these chenges several data structures in the operating system were modified (to consider tasks period, execution time and criticality). A new multiqueue scheme was defined, so as to accommodate real-time tasks along with interactive and CPU-bound tasks.

A new set of signals was added to indicate special situations, such as missed deadlines, overload or uncertainty of the schedulability of the task set.

All these services were made available to the programmer as a complete set of new system calls. A long list of tests demonstrated the feasibility of MINIX as a workbench for real-time development.

Several work was done using the tool, spanning from the testing of new scheduling algorithms to kernel modifications. In despite of this fact, several additional features were identified to be added to original environment. Recently, the need to integrate the previous work in a new version for the operating system arised. This happened because new MINIX versions were released in the meantime. Some of those extensions are presented in the following paragraphs.

### Analogic - Digital Conversion

The first group of changes was related with the need to acquire analogic data from the environment. As stated earlier, many real-time systems are used to control a real process, such as a production line or a chemical reaction. This implies a 'sense and act' attitude, i.e., sensing the environment and then changing it if necessary to keep control of the whole process. To sense the real world, a long list of sensors can be used, ranging from thermometers, pressure, infrared, etc.; many of them providing analogic signals.

The game port interface in the PC allows connecting up to four analog and four digital inputs. Providing the OS with the ability to directly read the game port enhances the chance to connect different analog sensors. The possibility to use this feature from within MINIX was tested [11], and a device driver for the game port was written.

When the new solutions were tested, it showed ppor performance when doing the readings. The device driver had to be completed rewritten, this time following the same framework used under Linux [12], with slightly changes. Resistive inputs (coordinates XY) and digital inputs (buttons) are aligned together in a byte (8 bits) that can be read at address 201h. Input pins from D-connector relates with that byte the following way:

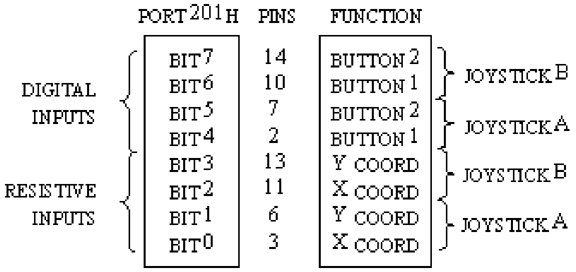

*Figure 1. Game Port Data bus and pins correlation*

The device driver adds a new kernel task that provide the programmer with three basic operations (open, read, close) to access the game port as character devices (/dev/js0 and /dev/js1, for joystick A and joystick B, respectively). To read the axis, the task sends any value to that port (201h) and cycles reading the port, waiting for any of the resistive inputs to become 0. The number of times the cycle is run is proportional to the resistance (and thus position) of the joystick. Some scripts were also modified to make device creation a simple step.

At present we ar working into the addition of new drivers for different A/D – D/A controllers.

### Joined Scheduling Queues

A second set of changes was related with the task scheduler management. The original task scheduler of MINIX used three queues, in order to handle task, server and user processes in that order of priority. Each queue was scheduled using the Round Robin algorithm.

The next figure shows the MINIX structure related to processes and message passing and the ready process queuing and handling:

| | | | | |
|---|---|---|---|---|
| *Level 3* | INIT | User 1 | User 2 | User n |
| *Level 2* | Memory Manager (MM) | | File System (FS) | |
| *Level 1* | Disk Task | Clock Task | Printer Task | Other Tasks |
| *Level 0* | Process Manager | | | |

*Figure 2. Processes structure in MINIX* [9]

Each of these levels are described below:

- *Level 0* is in charge of three fundamental duties: process management; message passing and interrupti management.
- *Level 1* includes I/O processes or *tasks* (known also as device drivers).
- *Level 2* contains only two processes, FS and MM, bringing an extended machine able to manage system calls of certain complexity.
- *Level 3* comprises all the processes below the INIT process, the place for applications (like compilers, shell, editors) and user processes.

The basic idea considered in joining the queues was related with the goal that a real-time task should not be interfered by low level interrupts (and its associated servers). The work presented in [13] worked on the hypothesis that server and user queues can be joined, allowing File System (FS) and Memory Manager (MM) processes to be moved from server to user process category.

The expected result of such change is getting better response time from the operating system. The union of the queues avoids intereference of the Operating

System tasks in the most critical real-time tasks. Several examples of possible scenarios are introduced. Through these case studies and their impacts in processing time, it became clear that the unification was feasible. Reducing the number of queues is also a step towards fault tolerance.

When the availability of shared resources (such as FS or MM) are diminished, a deadlock problem is likely to appear quite often. A deadlock occurs whenever a process is blocked waiting for a second process, while the later is also waiting for the first one.

Under the original scheduler in MINIX 2.0, a process requiring a service from FS or MM had it delivered immediately. This was that way because FS or MM had enough priority to start at any time without being preempted. An in-depth analysis was made to check the possibility of deadlock between FS and MM, first revisiting the semantics of them and then trying to measure the impact of the new scheduler (with the joined queues).

The only possible communication between FS y MM (under the original source code) is done during system initialization, and that connection is unidirectional, thus avoiding the circular waiting case. A conclusion from that scheme is that FS and MM work independently, having relation only with processes of task category (the kernel itself or device drivers). Task level processes have higher priority and are not preempted because of that condition, with their execution being considered instantaneous (and atomic) regarding a user process.

The final conclusion is that deadlocks are not probable to occur due to the changed scheduler. User processes cannot communicate each other; FS does not communicate with MM; and the management of the task queue was not altered from the original code. This is a very good feature to achieve fault tolerance.

## Real-time Metrics

Once the OS was extended with real-time services, the need arose to have several measuring tools. It is needed to test the evolution of the executing tasks according with the different scheduling strategies. The impact of the different workloads should be also considered.

To do so, the kernel is in charge to keep a data structure that is accessible to the user via a system call. The structure includes the following items:

```
struct rt_globstats {
      int actpertsk;
      int actapetsk;
      int misperdln;
      int misapedln;
      int totperdln;
      int totapedln;
      int gratio;
      clock_t idletime;
};
```

with:
*actpertsk, acrtapetsk*: number of active (running) real-time tasks, both periodic and aperiodic.

*misperdln, misapedln*: number of missed deadlines, both periodic and aperiodic.

*totpertsk, totapetsk*: number of total scheduled real-time tasks instances, both periodic and aperiodic.

*gratio*: guarantee ratio, i.e., the relationship between number of instances and deadlines met.

*idletime*: time (in clock ticks) not used as compute time.

Statistics also can be monitored online by means of a function key displaying all that information.

## Replicated Sensors

Sensor replication is an area of growing interest in real-time processing. It enhances the fault tolerance potential of the whole system by exploiting redundancy. As earlier explained, MINIX has been expanded with sensor reading capabilities, and the existing serial and parallel ports can be connected to data acquisiton hardware. The main goal was to include standard fault tolerant strategies, allowing to to check the validity of different available sensing algorithms.

The work presented in [14] introduces an important concept in order to tolerate sensor failure: the use of *abstract sensors*. An abstract sensor is a set of values that contains the present value of a physical variable of interest. Each abstract sensor is implemented using a *concrete sensor* (a physical device that reads a physical variable, i.e. a thermometer). The concrete sensor does not need to sense the physical variable of interest. For example, a temperature abstract sensor can be constructed using a manometer to sense pressure and then applying the Boyle's law.

Another important aspect of sensor replication is the ability to enhance the expected accuracy from a set of replicated sensors far beyond the obtainable using only one sensor. This leads to multisensor environments or the use of a distributed network of sensors.

Data coming from the physical system may be faulty due to sensor's failure, communication problems or noise. When using sensor replication, a method to combine data from several different sensors is needed. This action is called information integration, and it can be *competitive* or *complementary*.

In the first approach, each sensor theoretically provides identical information (though this is not the case in practice). Complementary information integration is done when partial information is available from each sensor: that information is combined to get the necessary knowledge about the environment.

Another advantage provided by the concept of abstract sensor is the capacity of data abstraction. A strategy of fault tolerance algorithms is to employ different kinds of redundant sensors. Thus, a real application could arrange different sensors (i.e., infrared, microwaves, and radar) that are not vulnerable to the same type of interference. To specify such a real-time system, only abstract sensors are considered, without concern of the type.

Using the algorithms studied under [15], the idea was to extend RT-MINIX with the possibility to use several sensors from a fault tolerance perspective. First of all, the four algorithms were coded as a user application. The next step was to incorporate the ability to use real data. In this case, the environment was sensed by means of four potentiometers (using the four analogic inputs from the joystick port). The inputs were arranged as a set of concrete sensors (acting as position sensors for a simulated robotic arm).

The algorithms worked as expected, providing a unique value from the replicated sensors and although one of them were faulty (the user had the chance to change data varying the potentiometers as desired).

Finally, the algorithms were combined in the kernel, providing the programmer with a set of functions to work with abstract sensors. It is possible to create (indicating physical devices, such as /dev/js0 and type of algorithms) and then read an abstract sensor, even in the presence of faulty concrete sensors.

## 3. CURRENT APPLICATIONS

The present section is mainly devoted to show several applications that have been developed using this Operating System, and a new set of programs being built at present.

### Supervisory Control And Data Acquisition

The first developed application was a SCADA program developed with academic purposes. It was written previously to run under MINIX and later adapted to execute in a real-time environment.

The SCADA is built as as a general application used to supervise a set of industrial processes. Different parameters can be defined for each process, including ports to be read, values to be recorded and alarms to be raised. Data acquired by the program can also be monitored from another computer through the serial ports. A history log file is generated, allowing the revision (and printout) of the activity that occurred during program execution.

A SCADA tool is a good application to test RT-MINIX with real processing conditions. It is composed of several periodic and sporadic real-time tasks running concurrently. It also includes a set of soft real-time tasks combined with interactive processes.

### A Model of a Bottle-filling Line

A prototype of a bottle-filling system (as described in [16]) is currently under construction, with the aim of using RT-MINIX as the RTOS to control such a real process.

The proposed system modeled in that work consists in a number of bottle-filling lines fed by a single vat containing the liquid to be bottled. The bottle size may differ from line to line. The tasks of the control system are to control the level, the pH and the temperature of the liquid in the vat, to manage the movement and filling of bottles in the various lines, and to exchange and log information with human operators working with the individual lines and a supervisor monitoring the entire system.

With several concurrent tasks (both periodic and aperiodic), this prototype will impose RT-MINIX with real-world constraints to play with.

## 4. PRESENT WORK

The sensor integration problem and tolerance of failures from replicated (redundant) sensors can now be studied in depth with help of RT-MINIX thanks to the incorporated sensing algorithms. A possible work line is deal with multidimensional sensors (replacing each interval corresponding to a physical value by a vector of intervals).

The algorithms presented in section 1 are only two examples of a long and growing list of scheduling algorithms. Real-time guarantees in the presence of faults along with fault tolerant scheduling strategies are very interesting fields to extend the present state of RT-MINIX. Feasible Shortest Path (FSP) and Linear Time Heuristic (LTH) are models that can be studied and compared, with a future implementation in RT-MINIX depending on results to be obtained.

One of the problems associated with scheduling algorithms is priority inversion. [17] presents a very clear example to definitely understand priority inversion, a case that occurred during the NASA Mars Pathfinder mission in 1997.

Any task within RT-MINIX can have a priority: if new scheduling algorithms to be implemented will consider that value to pick a task instead of another one, care must be taken in order to handle this characteristic properly. It is possible that a task with medium priority be scheduled while a high priority task is waiting for a resource that is blocked by a low priority task. A solution to that dilemma known as priority inheritance was identify and proposed in [18]. Tasks should inherit the right value to avoid priority inversion and furthermore deadline missing, thus improving the overall performance of the scheduling algorithms.

## 5. CONCLUSION

MINIX proved to be a feasible testbed for OS development and real-time extensions that could be easily added to it.

This “new” operating system (a MINIX 2.0 base with real-time extensions) has a rich set of features, which makes it a good choice to conduct real-time experiences. The added real-time services covered several areas:

- *Task creation*: tasks can be created either periodic or aperiodic, stating their period, worst execution time and priority

- *Clock resolution management*: the resolution (grain) of the internal clock can be changed to get better accuracy while scheduling tasks.

- *Scheduling algorithms*: both RMS and EDF algorithms are supported, and can be selected on the fly.

- *Statistics*: several variables about the whole operation are accessible to the user to provide data for benchmarking and testing new developments.

- *Supervisory Control and Data Acquisition*: as a user application, it makes full use of real-time services.

- *Sensor Integration*: tolerance of failures from replicated (redundant) sensors will be achieved due to the sensing algorithms added to RT-MINIX.

With these extensions, RT-MINIX can be used as a platform for real-time processing or as a starting point for adding more real-time services.

## 6. ACKNOWLEDGEMENTS

This work was partially supported by the UBA-SECYT research project TX-004, "Concurrency in Distributed Systems".

All the related source code can be obtained via FTP at http://www.dc.uba.ar/people/proyinv/cso/rt-minix, together with downloading and installation instructions.

## 7. REFERENCES

[1] H. Tokuda, T. Nakajima, P. Rao. Real-time MACH: Towards a predictable real-time system. *Proceedings of USENIX MACH Workshop*, volume 1, 1990.

[2] V. Yodaiken. The RT-Linux approach to hard real-time. Online publication found at http://luz.cs.nmt.edu/~rtlinux/whitepaper/short.html

[3] B. Srinivasan. KURT: The KU Real-Time Linux. Online publication found at http://hegel.itc.ukans.edu/projects/kurt/

[4] J. Stankovic, K. Ramamrithman. The design of the Spring kernel. In Proc. of 8th RealTime Systems Symposium. 1991.

[5] M. Saksena, J da Silva, A. Agrawala. Principles of Real-Time Systems, chapter Design and Implementation of Maruti. Prentice-Hall, 1994.

[6] K. Jeffay, D. Stone, D. Poitier. Kernel support for efficient, predictable real-time systems. Proceedings of the IEEE Workshop on RTOS, pp. 8-31, 1991.

[7] LynxOS – Hard Real-time OS Features and Capabilities, online at http://www.lynx.com/products/ds_lynxos.html

[8] QNX Realtime OS, online at http://www.qnx.com/product/qnxrtos.html

[9] A. Tannenbaum, "A Unix clone with source code for operating systems courses", *ACM Operating Systems Review*, 21:1, January 1987.

[10] G. Wainer, "Implementing Real-Time Scheduling in a Time-Sharing Operating System", *ACM Operating Systems Review*, July 1995.

[11] D. Polakoff, P. Rogina, W. Ruaro, E. Szulsztein, Wainer, G. "Real-time modificatioins of the Minix Operating System" (in Spanish), Internal Report, CS Dept., FCEyN, UBA, December 1997.

[12] V. Paulik, Joystick device driver for Linux, online at ftp://atrey.karlin.mff.cuni.cz/pub/linux/joystick/joystick-0.8.0.tar.gz

[13] N. Wolowick, M. Cuenca Acuña, Wainer, G. "Joining the scheduling queues in Minix Operating System" (in Spanish), Internal Report, CS Dept., FCEyN, UBA, July 1998.

[14] K. Marzullo, "Tolerating failures of continuous-valued sensors", ACM Transactions on Computer Systems, 8(4):284-304, November 1990.

[15] R. Brooks, S. Iyengar, "Robust Distributed Computing And Sensing Algorithm", IEEE Computer, June 1996, pp. 53-60.

[16] P. Ward, S. Mellor, "Structured Development for Real-Time Systems", Appendix B, Yourdon Press, 1985.

[17] M. Jones, What happened on Mars?, online at http://www.cs.cmu.edu/afs/cs/project/art-6/www/mars.html

[18] L. Sha, R. Rajkumar, J. Lehoczky, Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Trans. on Comp.*, 39:1175-1185, Sep. 1990