Real-Time Server Oriented Operating System for Embedded Applications

David R. Donari, Leo Ordinez, Rodrigo Santos, and Javier Orozco

Instituto de Investigaciones en Ingeniería Eléctrica
Universidad Nacional del Sur - CONICET
Av. Alem 1253 - (8000) Bahía Blanca
Buenos Aires - Argentina
{ddonari,lordinez,ierms,jorozco}@uns.edu.ar

Abstract. In recent years, the use of embedded systems has been multiplied creating the necessity of new tools for the development of software applications running on them and in most cases with real-time requirements. The speed at which microcontrollers improve their performances and capabilities oblige a continuous update in their software counterpart that not always can keep the pace. In this paper, a new approach to the design of embedded systems applications is presented based on an easily portable real-time operating system and a development framework.

1 Introduction

Embedded systems (ES) have been defined as a computer that forms part of a bigger system and that depends on its own microprocessor [1]. In the last years, with the advances in electronic integration, the use of embedded systems has been multiplied. They are present in industrial plants, in the power train of cars, cell phones, laser and inject printers, in many of the house garments like DVD players, microwave ovens, etc. Although at first sight these applications seem very different, they share a common structure, which is the collection of data from sensors, its processing and the command of actuators. An additional characteristic is the capacity of communication with other devices.

Most current ES are implemented with system-on-a-chip (SoC) technology. In a single chip, the input/output devices, memory and microprocessor are integrated reducing the layout, weight, size and energy cost. As these systems interact with the environment, many times they have real-time requirements.

Software design for ES has been an *ad-hoc* process customized for each particular application. This implies that the software development was oriented to the particular underlying hardware architecture. This has the disadvantage of poor compatibility and portability of the generated code for each particular case.

Operating systems (OS) are an important part of any modern system. They provide the necessary hardware abstraction so software development can be built without considering any particular features of the microprocessor. Most OSs are service oriented. That means they provide a set of services to the applications

XXXIV Conferencia Latinoamericana de Informática (CLEI 2008), pp. 1425-1434, 2008.

like a file-system, I/O drivers, etc. In addition, OS are in charge of scheduling and dispatching processes and keep the system running. In the particular case of embedded systems, they pursue the objective of providing the necessary portability between the different platforms on which the system may run [2].

In this paper the use of the Resource Reservation Framework (RRF) [3] is proposed as the scheduling paradigm for the OS. In this framework, each task is associated a *reservation* characterized by a budget Q and a period P. Roughly speaking, a reservation *transforms* the soft real-time task into a sporadic task with worst-case execution time no larger than Q and minimum inter-arrival time equal to P, regardless of the actual task requirements. If the set of reservations is schedulable, the RRF provides the important property of *temporal isolation*: the ability of a task scheduled by a reservation to meet its timing constraints depends only on the reservation parameters and not on the presence of other tasks in the system. The task executes approximately as if it was on a virtual dedicated processor of speed $U = Q/P$ times the speed of the real processor.

Examples of such algorithms are the Constant Bandwidth Server running on top of Earliest Deadline First (CBS+EDF)[4, 5]; the Sporadic Server running on top of Fixed Priority (SS+FP) [6, 5].

The use of servers provides also the possibility of negotiate the quality of service of the application. If the server resembles exactly the minimum interarrival time and worst case execution time, the task is guaranteed a hard real-time treatment. If this is not the case, a soft real-time guarantee is provided. The OS does not check the feasibility of the system but guarantees an appropriate behavior provided the tasks and server are properly configured.

Contributions: In this paper, the design and implementation of a Server Oriented Operating System (S.O.O.S.) for embedded systems is proposed. There is a careful description of its architecture and the way in which the different parts interact. Most of the advantages provided by S.O.O.S. are based in the use of a framework that provides the necessary tools to ensure their proper functioning. At the end of this work, this framework-OS interaction is explained.

The rest of the paper is organized as follows. Section 2 presents the state of the art in the design and implementation of real-time operating systems. In Section 3, the design goals of a new OS implementation are expounded. In Section 4, the architecture of the OS is described in detail. Finally, Section 5 describes the contribution of this paper and the direction of future works.

2 Related Works

In recent years, the development of OSs for embedded systems has attracted interest from both academy and industry. Many efforts have been made to provide temporal guarantees to this kind of OS but no standard solution have been reached yet.

Embedded operating systems have been developed in [7] [8] for the special case of the Hitachi H8/300. In the first one, the design includes some support

tools for the development and debugging of applications. It works only with fixed priorities. The contention method for mutex is based on the Ceiling Protocol [9]. The second one also works with fixed priorities. There are a set of examples that work properly but the implementation is not independent of the hardware. In this aspect, the purpose of the OS is somehow violated as the hardware is not transparent to the application. In ERIKA, both ceiling protocol [9] and stack resource policy [10] are implemented for resource contention.

The introduction of an operational framework to generate a scheduler and a dispatcher within an operating system has been outlined by Wang in [11]. In [12] a hybrid real-time scheduler that supports non real-time tasks is proposed. The scheduling is done in a hierarchical architecture; tasks are grouped under Constant Use Servers (CUS) that perform the admission test. Then, all servers are scheduled in the same level. Pawan, also proposed in [13] a self-tuned hierarchical scheduler through a QoS manager.

In this paper, a resource reservation mechanism based on the Behavioral Importance Dual-priority Servers (BIDS) [14] is proposed for an embedded operating system. It differs from the previous work, in the way the servers are implemented natively and schedule by the kernel. Based on modules and layers the system is easily reconfigurable with very little intrusion from the application level into the kernel one.

3 Design Goals

Most OSs for embedded systems share a common set of requirements and objectives including real-time guarantees, inter-task communication, mutual exclusion and reliability. Not all implementations fulfill all of them and in just a couple are taken into account. In the case of the S.O.O.S project, the accent is put in the facilities provided by the server mechanism to schedule tasks.

Small generic real-time Kernel It is important for the S.O.O.S. to have a small generic real-time kernel since it is intended for embedded systems with memory restrictions. In this way, the developer of the application to be implemented in the embedded system has both freedom and responsibility to choose the best configuration for the scheduling policy from a set of possibilities.

Reservation mechanism In addition to the conventional scheduling policies, the OS should provide software tools capable of implementing mechanisms to guarantee a bandwidth reservation to tasks. The reservation mechanism should guarantee isolation preventing that an overload on one task affects other tasks in the system.

Framework support The inclusion of a development framework is important for the case of embedded systems because it allows a systematization of the design procedure with basic blocks that can be easily understood. The existence of upper levels of abstraction facilitates the migration of the system among different hardware platforms. This important feature allows the

developer to base his programming in logical entities and to benefit himself with a predictable behavior based on a formal validation of this components. This predictable behavior involves a logical/temporal correctness.
From a practical point of view, the framework must support a configuration file, allowing the developer to prove different system parameters in a simple way. Thus, is easy to evaluate which one is the best for the system under construction.

4 S.O.O.S. Architecture

In this section, the architecture of S.O.O.S. is presented. In the first place, the general aspects of the OS are described. Then, a detailed explanation of how resource reservation mechanisms are implemented through servers. Finally, an overview of the supporting development framework is made.

4.1 General architecture

S.O.O.S. is an operating system design, able to work on multiple platforms. It differs from general purpose OS in the way the spaces for the OS and the application are separated. In general purpose OS, like Windows or GNU/Linux, executing on a microprocessor like Intel Celeron or similar, the areas are clearly defined and a common user cannot access the reserved space. In microcontrollers, there is not such a distinction. An application may eventually access the space reserved to the OS. Usually, application and OS are compiled together to create an image that is burnt in a flash ROM memory. It is important then to provide as much protection as possible at design time to avoid unnecessary intrusions in the OS area. To reach this objective, S.O.O.S. has a modular structure based on layers [15]. Each layer and module provides a set of functions that facilitates the communication and hides the actual implementation. In this way, it is possible to port the application to another hardware platform just by changing the appropriate layers. Figure 1 shows the architecture of S.O.O.S.

Going bottom-up, the first layer is the Hardware Adaptation Layer (HAL). It is in charged of managing the I/O ports, interrupts handlers and the RT-Clock. It exports to the upper layers a set of functions that allow an application designer to access the I/O ports in a simple standard way. The memory management is important for the context switch operation. Implemented at this level, it is important to preserve the state of the processor whenever a new task is executed or the control of the microcontroller is transfer for example to an interrupt handler. The S.O.O.S. project is based on the Time Triggered Protocol (TTP) [16]. This approach is the most used in real-time systems as it provides better predictability than the Event Triggered Protocol (ETP) [17]. In TTP, there is a real-time clock that states a time base to the system. The slot (time slice or timer tick) is the unit of time, which duration depends on many factors ranging from the precision of the timer implementing it to the granularity of the system.

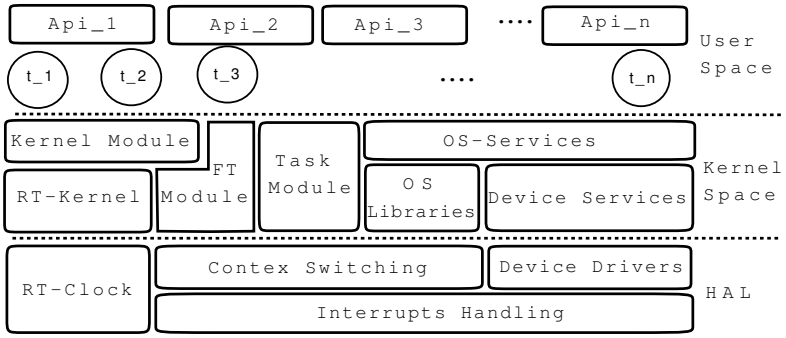


Fig. 1. System Model

As the HAL is dependent on the hardware platform, most of it is written in assembly language.

Upon the HAL, the kernel layer implements the basic services of the S.O.O.S.: load and execute tasks, manage software and hardware calls and provide a set of OS tools. Different modules are built in this layer. The RT-kernel is a set of procedures that schedule and dispatch the tasks in the system. It is in charged of executing the policy defined. It can be either fixed priorities or dynamic ones. The implementation of this is made by means of hooks that provide a set of primitives to be run before compilation actually is done. The flexibility to change from one policy to another is as simple as changing a definition among the directives to the compiler. To do this, the functions associated with the scheduler and dispatcher modules are implemented in a generic way.

The RT-kernel does not test if the system is feasible or not, it simply follows the policy set at compilation time as the one to be used. Each policy may have different advantages and disadvantages. The designer of the system has the responsibility of choosing the one that best satisfies the requirements of the tasks involved. Some of the properties to contemplate are for example: jitter, worst case response time, average response time, latency, etc. In this sense, S.O.O.S. provides a tool for logging the state of the tasks for analyzing the performance.

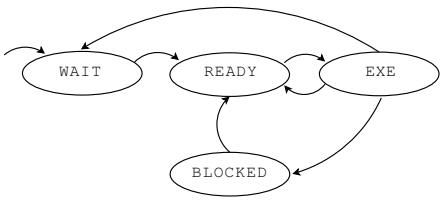


Fig. 2. Task States

The Task Module is a set of procedures to manage the task behavior. The RT-Kernel uses this module to handle different situation in the life-cycle of a task. The core of the task is a sequence of instructions that end with a system call to indicate S.O.O.S. that it has finished its execution, such as the *yield()* instruction [18]. If the task is periodic, this system call activates a timer that will re-activate the task later.

```
void SOOS_task_i(){
                    SOOS_Sensor S1;
                    SOOS_ReadSensor(S1);
                    if(checkThreshold(S1) == 0)
                    then
                    ...
                    else
                    ...
                    SOOS_taskEnds();
                   }
```

Fig. 3. Typical Task's structure

The basic three types of tasks, hard, soft and non real-time share the same structure, see Figure 3. Tasks can be in the usual states, Idle, Ready and Executing as shown in Figure 2.

A special module is included to provide fault tolerance. Basically, if this module is included at compilation time, a bandwidth collector is created. Its functioning is as follows, it accumulates the unused tasks bandwidth and in case a task can not complete its execution because of an error has occurred (for example time out waiting for a sensor response), this module can provide the extra bandwidth needed to recover the task. Of course, as the S.O.O.S. works with real-time constraints, this is not always possible and some extra requirements are needed to guarantee the feasibility of the rest of the tasks.

The kernel layer is completed with the OS services module that is based on the OS libraries and device services modules. They have the objective of exporting from the HAL to the Application layer the I/O ports drivers, interrupt handlers and time administration. The implementation was made through inline functions, avoiding the need to produce unnecessary branches whenever an application requires access to the hardware resources by means of a system call.

4.2 Servers

The implementation of the RRF is made through a special software entity named server. In it, the reservation has a budget that corresponds to the execution time of the task allocated to the server and a period which is associated to the minimum interarrival time of the task. In the case of hard real-time tasks, the server parameters should be equal to the worst case in order to provide the temporal guarantees needed. For soft tasks, the servers containing them should have parameters selected according to some optimization mechanism. However, this does

not form part of S.O.O.S. but of the design of the system. The server implementation is supported by a special module that follows the generic implementation of the RT-kernel. Tasks are loaded into servers by indicating in a special field of them that they are going to run in a server. No additional information is needed from the programmer since the servers are managed natively by the kernel where the user should not interfere. In this sense, the use of servers is transparent for the programmer.

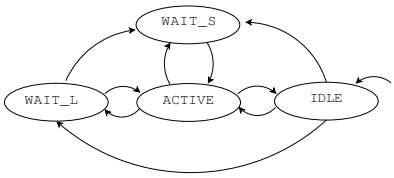


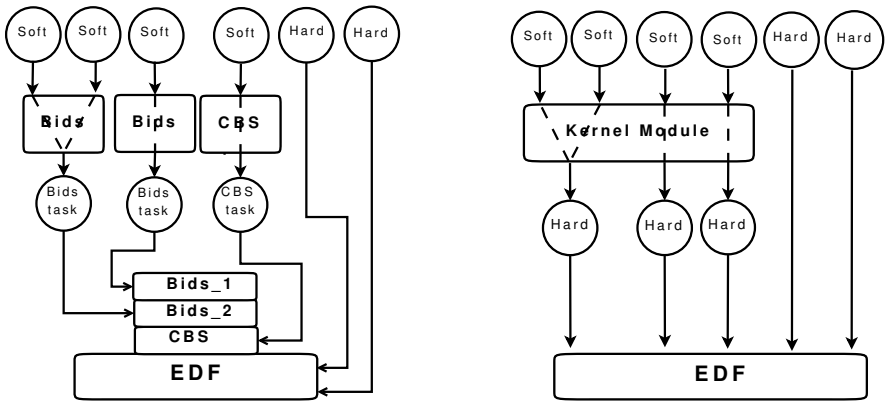
Fig. 4. State Diagram of Bids Server

Each server generated in the kernel module, creates a virtual hard task in the RT-Kernel. In this sense, all servers are guaranteed their execution with a worst case execution time equal to their budget and minimum interarrival time equal to the period. In this manner, the RT-Kernel manages a server in the same way it would manage a hard task that is, following the diagram in Figure 2. However, inside the server, the real task may have a different behavior that depends on the kind of server scheme used and its own governing rules. For example, if BIDS [14] is used, it may be as the one presented in Figure 4. These states are characteristic of the server and evolve inside the Kernel Module. Thus, the Kernel Module acts like an intermediary between the RT-Kernel and the actual task. This is, the corresponding states of a particular server are mapped by the Kernel Module to the corresponding ones of the RT-Kernel.

In Figure 5 two different approaches to the implementation of servers in a RTOS are presented. Both are logically correct but the second one has certain advantages that are going to be discussed. The first one, corresponds to the traditional implementation that can be found in some OS like Linux in its real-time version [18]. In this one, the whole Linux kernel is embedded into a soft task that is schedule by the real-time scheduler that is mounted upon. The implementation is quite straightforward but has little portability when there is an evolution in the original kernel. Instead, in S.O.O.S. the RT-Kernel always deals with hard tasks holding the different types of tasks whether they are hard, soft or non real-time.

4.3 Framework Support

To carry out a great scale software project it is necessary to have support tools that allow a better handling of the operating system. For this reason, the use of a development framework with diverse utilities is necessary. There are two groups



(a) Server Implementation on Linux

(b) S.O.O.S.

Fig. 5. Server implementations.

of software tools, according to the user point of view. On the one hand, the tools that belong to the *front-end* of the system, which are used by the programmer of the system. On the other hand, the *back-end* tools support those requirements. This structure has been proposed in [19].

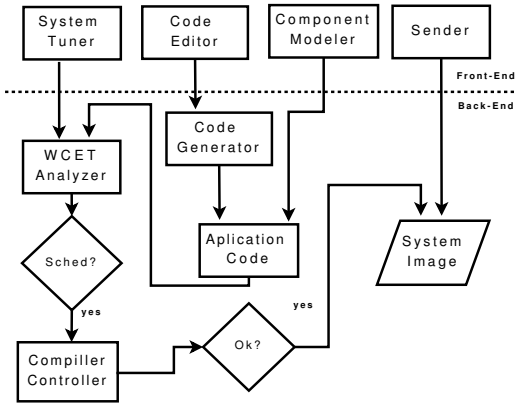


Fig. 6. Framework Support

Figure 6 shows the interaction of the different framework components. As it can be observed, the designer deals with a Code Editor and a Component Modeler that interfaces the software and hardware components that S.O.O.S.

supports. Both tools generate an Application Code. The information provided by the designer about the requirements of the system (*i.e.*, scheduler, servers, architecture) is gathered by the System Tuner. The Worst Case Execution Time (WCET) Analyzer evaluates the schedulability of the system based on the generated application code and checks its feasibility. If it is correct, the Compile Controller generates a compilation file. That file is then used to compile and link the files corresponding to the application and system. If there is no error, an image to be burnt on the microcontroller is built. Finally, once the image is built the sender can address the destination and send it.

Although the system can be executed without the framework, it is necessary to highlight that its usage guarantees a correct behavior. Framework aided programming provides the developer with a high degree of abstraction. In addition, it enables that a future system upgrade does not represent a significant change in the OS, but only the framework specification [20]. Thus, the system can be update easily by just maintaining the framework tools as part of the process.

5 Conclusions and future works

This paper presented S.O.O.S. as an OS for embedded systems. The implementation of a generic kernel allows an OS to meet application developers'requirements. The use of generic routines also facilitates the inclusion of new scheduling policies.

The implementation of servers as a resource reservation mechanism offers the developer the ability to execute tasks with soft time constraints, guaranteeing a series of properties mentioned in the course of this work. This mechanism implemented in the Kernel Module does not alter the performance of tasks with hard restrictions. The servers are implemented in a transparent manner to the programmer, which allows his programming to be focused on the application dynamics.

At last, the implementation of the framework that supports it was explained. This is a complementary design to the OS that offers a number of features to the developer of applications, among which highlights a system configuration tool, a high-level modeler, a code generator and a WCET analysis tool, among others. This framework-OS scheme is the new trend in embedded systems design, since it improves productivity, adaptability, flexibility and debugging of the whole system. Although not obligatory the set of tools provided by a framework, they are necessary to take advantage of all the characteristics offered by S.O.O.S.

As future work, new capabilities will be added to the model like a module for energy awareness and one to allow distributed communications. Also, the fault-tolerance module will be deeply study. Even though, still in a prototype form, S.O.O.S is being implemented on the H8/3292 microcontroller to control the Lego Mindstorms robot. The S.O.O.S. is distributed under the GPL license, and it can be found at http://www.ingelec.uns.edu.ar/rts/soos.

References

1. Wolf, W.: Embedded computing - what is embedded computing? IEEE Computer **35**(1) (2002) 136–137
2. Dostal, W., Michelmann, T.: Eingebettete systeme bilden den mikrokosmos einer soa. Javaspektrum (11 2006) 56–59
3. Rajkumar, R., Juvva, K., Molano, A., Oikawa, S.: Resource kernels: a resource-centric approach to real-time systems. In: Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking. (1998)
4. Abeni, L., Buttazzo, G.C.: Integrating multimedia applications in hard real-time systems. In: RTSS. (1998) 4–13
5. Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in hard real time environment. ACM **20** (1973) 46–61
6. Sprunt, B.: Aperiodic task scheduling for real-time systems. PhD thesis, Dept. of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburg, PA (August 1990)
7. Thane, H., Pettersson, A., Sundmark, D.: The asterix realtime kernel. In: Proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS'01), Industrial Session, Delft. (June 2001)
8. Gai, P., Cantini, D., Cirinei, M.: E.R.I.K.A. http://erika.sssup.it.
9. Sha, L., R.R., Lehoczky, J.P.: Priority inheritance protocols: An approach to real-time synchronization. IEEE Trans. Comput. (9 1990) 1175–1185
10. Baker, T.P.: Stack-based scheduling of realtime processes. Real-Time System Symposium, Proceedings (1990)
11. Wang, Y.C., Lin, K.J.: Implementing a general real-time scheduling framework in the RED-linux real-time kernel. In: IEEE Real-Time Systems Symposium. (1999) 246–255
12. Pengliu Tan, H.J., Zhang, M.: A hybrid scheduling scheme for hard, soft and non-real-time tasks. In: Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06)
13. Goyal, P., Guo, X., Vin, H.M.: A Hierarchical CPU Scheduler for Multimedia Operating Systems. In: Usenix Association Second Symposium on Operating Systems Design and Implementation (OSDI). (1996) 107–121
14. Ordinez, L., Donari, D., Santos, R., Orozco, J.: A behavior priority driven approach for resource reservation scheduling. In: Proceedings of the 2008 ACM Symposium on Applied Computing, Fortaleza, Ceará, Brazil
15. Tanenbaum, A.S.: Structured Computer Organization. Cuarta edn. Prentice Hall International (1999)
16. Kopetz, H., Grnsteidl, G.: Ttp- a protocol for fault-tolerant real-time systems. Computer **27**(1) (1994) 14–23
17. Kopetz, H.: Should responsive systems be event-triggered or time-triggered ? Institute of Electronics, Information, and Communications Engineers Transactions on Information and Systems **E76-D**(11) (1993) 1325–1332
18. Mantegazza, P., Bianchi, E., Dozio, L., Angelo, M., Beal, D.: Rtai programming guide 1.0.
19. Ordinez, L., Donari, D.: Flemi: an approach to a new real-time framework. In: Proceedings of the 9th Workshop on Real-Time Systems, Belém, Brazil, Wip session. (2007)
20. Hsiung, P.A., Lin, S.W., Tseng, C.H., Lee, T.Y., Fu, J.M., See, W.B.: Vertaf: an application framework for the design and verification of embedded real-time software. IEEE Transactions on Software Engineering **30**(10) (Oct. 2004) 656–674