We are IntechOpen, the world's leading publisher of Open Access books Built by scientists, for scientists

4,800
Open access books available

122,000
International authors and editors

135M
Downloads

Our authors are among the

154
Countries delivered to

TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities

CLARIVATE ANALYTICS BOOK CITATION INDEX INDEXED

WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com

5

Real-Time Operating Systems and Programming Languages for Embedded Systems

Javier D. Orozco and Rodrigo M. Santos
Universidad Nacional del Sur - CONICET
Argentina

1. Introduction

Real-time embedded systems were originally oriented to industrial and military special purpose equipments. Nowadays, mass market applications also have real-time requirements. Results do not only need to be correct from an arithmetic-logical point of view but they also need to be produced before a certain instant called deadline (Stankovic, 1988). For example, a video game is a scalable real-time interactive application that needs real-time guarantees; usually real-time tasks share the processor with other tasks that do not have temporal constraints. To organize all these tasks, a scheduler is typically implemented. Scheduling theory addresses the problem of meeting the specified time requirements and it is at the core of a real-time system.

Paradoxically, the significant growth of the market of embedded systems has not been accompanied by a growth in well-established developing strategies. Up to now, there is not an operating system dominating the market; the verification and testing of the systems consume an important amount of time.

A sign of this is the contradictory results between two prominent reports. On the one hand, The Chaos Report (*The Chaos Report*, 1994) determined that about 70 % had problems; 60 % of those projects had problems with the statement of requirements. On the other hand, a more recent evaluation (Maglyas et al., 2010) concluded that about 70% of them could be considered successful. The difference in the results between both studies comes from the model adopted to analyze the collected data. While in The Chaos Report (1994) a project is considered to be successful if it is completed on time and budget, offering all features and functions as initially specified, in (Maglyas et al., 2010) a project is considered to be successful even if there is a time overrun. In fact, in (Maglyas et al., 2010) only about 30% of the projects were finished without any overruns, 40% have time overrun and the rest of the projects have both overruns (budget and time) or were cancelled. Thus, in practice, both studies coincide in that 70 % of the projects had some kind of overrun but they differ in the criteria used to evaluate a project as successful.

In the literature there is no study that conducts this kind of analysis for real time projects in particular. The evidence from the reports described above suggests that while it is difficult to specify functional requirements, specifying non functional requirements such as temporal constraints, is likely to be even more difficult. These usually cause additional redoes and errors motivated by misunderstandings, miscommunications or mismanagement. These

errors could be more costly on a time critical application project than on a non real time one given that not being time compliant may cause a complete re-engineering of the system. The introduction of non-functional requirements such as temporal constraints makes the design and implementation of these systems increasingly costly and delays the introduction of the final product into the market. Not surprisingly, development methodologies for real-time frameworks have become a widespread research topic in recent years.

Real-time software development involves different stages: modeling, temporal characterization, implementation and testing. In the past, real-time systems were developed from the application level all the way down to the hardware level so that every piece of code was under control in the development process. This was very time consuming. Given that the software is at the core of the embedded system, reducing the time needed to complete these activities reduces the time to market of the final product and, more importantly, it reduces the final cost. In fact, as hardware is becoming cheaper and more powerful, the actual bottleneck is in software development. In this scenario, there is no guarantee that during the software life time the hardware platform will remain constant or that the whole system will remain controlled by a unique operating system running the same copy of the operating embedded software. Moreover, the hardware platform may change even while the application is being developed. Therefore, it is then necessary to introduce new methods to extend the life time of the software (Pleunis, 2009).

In this continuously changing environment it is necessary to introduce certainty for the software continuity. To do such a thing, in the last 15 years the paradigm Write Once Run Anywhere (WORA) has become dominant. There are two alternatives for this: Java and .NET. The first one was first introduced in the mid nineties and it is supported by Sun Microsystems and IBM among others (Microsystems, 2011). Java introduces a virtual machine that eventually runs on any operating system and hardware platform. .NET was released at the beginning of this century by Microsoft and is oriented to Windows based systems only and does not implement a virtual machine but produces a specific compilation of the code for each particular case. (Zerzelidis & Wellings, 2004) analyze the requirements for a real-time framework for .NET.

Java programming is well established as a platform for general purpose applications. Nevertheless, hardware independent languages like Java are not used widely for the implementation of control applications because of low predictability, no real-time garbage collection implementation and cumbersome memory management (Robertz et al., 2007). However, this has changed in the last few years with the definition and implementation of the Real-Time Specification for Java. In 2002, the specification for the real-time Java (RTSJ) proposed in (Gosling & Bollella, 2000) was finally approved (Microsystems, 2011). The first commercial implementation was issued in the spring of 2003. In 2005, the RTSJ 1.0.1 was released together with the Real-Time Specification (RI). In September 2009 Sun released the Java Real-Time System 2.2 version which is the latest stable one. The use of RTSJ as a development language for real-time systems is not generalized, although there have been many papers on embedded systems implementations based on RTSJ and even several full Java microprocessors on different technologies have been proposed and used (Schoeberl, 2009). However, Java is penetrating into more areas ranging from Internet based products to small embedded mobile products like phones as well as from complex enterprise systems to small components in a sensor network. In order to extend the life of the software, even over a particular device, it becomes necessary to have transparent development platforms to the

hardware architecture, as it is the case of RTSJ. This is undoubtedly a new scenario in the development of embedded real time systems. There is a wide range of hardware possibilities in the market (microcontrollers, microprocessors and DSPs); also there are many different programming languages, like C, C++, C#, Java, Ada; and there are more than forty real-time operating systems (RTOS) like RT-Linux, Windows Embedded or FreeRTOS. This chapter offers a road-map for the design of real-time embedded systems evaluating the pros and cons of the different programming languages and operating systems.

Organization: This chapter is organized in the following way. Section 2 describes the main characteristics that a real-time operating system should have. Section 3 discusses the scope of some of the more well known RTOSs. Section 4 introduces the languages used for real-time programming and compares the main characteristics. Section 5 presents and compares different alternatives for the implementation of real-time Java. Finally, Section 6 concludes.

2. Real time operating system

The formal definition of a real-time system was introduced in Section 1. In a nutshell these are systems which have additional non-functional requirements that are as important as the functional ones for the correct operation. It is not enough to produce correct logical-arithmetic results; these results must also be accomplished before a certain deadline (Stankovic, 1988). This timeliness behavior imposes extra constraints that should be carefully considered during the whole design process. If these constraints are not satisfied, the system risks severe consequences. Traditionally, real-time systems are classified as hard, firm and soft. The first class is associated to critical safety systems where no deadlines can be missed. The second class covers some applications where occasional missed deadlines can be tolerated if they follow a certain predefined pattern. The last class is associated to systems where the missed deadlines degrade the performance of the applications but do not cause severe consequences. An embedded system is any computer that is a component of a larger system and relies on its own microprocessor (Wolf, 2002). It is said to work in real-time when it has to comply with time constraints, being hard, firm or soft. In this case, the software is encapsulated in the hardware it controls. There are several examples of real-time embedded systems such as the controller for the power-train in cars, voice processing in digital phones, video codecs for DVD players or Collision Warning Systems in cars and video surveillance cam controllers.

RTOS have special characteristics that make them different to common OS. In the particular case of embedded systems, the OS usually allows direct access to the microprocessor registers, program memory and peripherals. These characteristics are not present in traditional OS as they preserve the kernel areas from the user ones. The kernel is the main part of an operating system. It provides the task dispatching, communication and synchronization functions. For the particular case of embedded systems, the OS is practically reduced to these main functions. Real-time kernels have to provide primitives to handle the time constraints for the tasks and applications (deadlines, periods, worst case execution times (WCET)), a priority discipline to order the execution of the tasks, fast context switching, a small footprint and small overheads.

The kernel provides services to the tasks such as I/O and interrupt handling and memory allocation through *system-calls*. These may be invoked at any instant. The kernel has to be able to preempt tasks when one of higher priority is ready to execute. To do this, it usually has the maximum priority in the system and executes the scheduler and dispatcher periodically

based on a timer tick interrupt. At these instants, it has to check a ready task queue structure and if necessary remove the running task from the processor and dispatch a higher priority one. The most accepted priority discipline used in RTOS is fixed priorities (FP) (*eCosCentric*, 2011; *Enea OSE*, 2011; *LynxOS RTOS, The real-time operating system for complex embedded systems*, 2011; *Minimal Real-Time Operating System*, 2011; *RTLinuxFree*, 2011; *The free RTOS Project*, 2011; *VxWorks RTOS*, 2011; *Windows Embedded*, 2011). However, there are some RTOSs that are implementing other disciplines like earliest deadline first (EDF) (*Erika Enterprise: Open Source RTOS for single- and multi-core applications*, 2011; *Service Oriented Operating System*, 2011; *S.Ha.R.K.: Soft Hard Real-Time Kernel*, 2007). Traditionally, real-time systems scheduling theory starts considering independent, preemptive and periodic tasks. However, this simple model is not useful when considering a real application in which tasks synchronize, communicate among each other and share resources. In fact, task synchronization and communication are two central aspects when dealing with real-time applications. The use of semaphores and critical sections should be controlled with a contention policy capable of bounding the unavoidable priority inversion and preventing deadlocks. The most common contention policies implemented at kernel level are the priority ceiling protocol (Sha et al., 1990) and the stack resource policy (Baker, 1990). Usually, embedded systems have a limited memory address space because of size, energy and cost constraints. It is important then to have a small footprint so more memory is available for the implementation of the actual application. Finally, the time overhead of the RTOS should be as small as possible to reduce the interference it produces in the normal execution of the tasks.

The IEEE standard, Portable Operating System Interface for Computer Environments (POSIX 1003.1b) defines a set of rules and services that provide a common base for RTOS (IEEE, 2003). Being POSIX compatible provides a standard interface for the system calls and services that the OS provides to the applications. In this way, an application can be easily ported across different OSs. Even though this is a desirable feature for an embedded RTOS, it is not always possible to comply with the standard and keep a small footprint simultaneously. Among the main services defined in the POSIX standard, the following are probably the most important ones:

- Memory locking and Semaphore implementations to handle shared memory accesses and synchronization for critical sections.
- Execution scheduling based on round robin and fixed priorities disciplines with thread preemption. Thus the threads can be waiting, executing, suspended or blocked.
- Timers are at the core of any RTOS. A real-time clock, usually the system clock should be implemented to keep the time reference for scheduling, dispatching and execution of threads.Memory locking and Semaphore implementations to handle shared memory accesses and synchronization for critical sections.

2.1 Task model and time constraints

A real-time system is temporally described as a set of tasks $S(m) = \{\tau_1, \ldots, \tau_i, \ldots, \tau_m\}$ where each task is described by a tuple $(WCET_i, T_i, D_i)$ where T_i is the period or minimum interarrival time and D_i is the relative deadline that should be greater than or equal to the worst case response time. With this description, the scheduling conditions of the system for different priority disciplines can be evaluated. This model assumes that the designer of the system can measure in a deterministic way the worst case execution time of the tasks. Yet,

this assumes knowledge about many hardware dependent aspects like the microprocessor architecture, context switching times and interrupts latencies. It is also necessary to know certain things about the OS implementation such as the timer tick and the priority discipline used to evaluate the kernel interference in task implementation. However, these aspects are not always known beforehand so the designer of a real-time system should be careful while implementing the tasks. Avoiding recursive functions or uncontrolled loops are basic rules that should be followed at the moment of writing an application. Programming real-time applications requires the developer to be specially careful with the nesting of critical sections and the access to shared resources. Most commonly, the kernel does not provide a validation of the time constraints of the tasks, thus these aspects should be checked and validated at the design stage.

2.2 Memory management

RTOS specially designed for small embedded system should have very simple memory management policies. Even if dynamic allocations can provide a better performance and usage, they add an important degree of complexity. If the embedded system is a small one with a small address space, the application is usually compiled together with the OS and the whole thing is burnt into the ROM memory of the device. If the embedded system has a large memory address space, such as the ones used in cell phones or tablets, the OS behaves more like a traditional one and thus, dynamic handling of memory allocations for the different tasks is possible. The use of dynamic allocations of memory also requires the implementation of garbage collector functions for freeing the memory no longer in use.

2.3 Scheduling algorithms

To support multi-task real-time applications, a RTOS must be multi-threaded and preemptible. The scheduler should be able to preempt any thread in the system and dispatch the highest priority active thread. Sometimes, the OS allows external interrupts to be enabled. In that case, it is necessary to provide proper handlers for these. These handlers include a controlled preemption of the executing thread and a safe context switch. Interrupts are usually associated to kernel interrupt service routines (ISR), such as the timer tick or serial port interfaces management. The ISR in charge of handling the devices is seen by the applications like services provided by the OS.

RTOS should provide a predictable behavior and respond in the same way to identical situations. This is perhaps the most important requirement that has to be satisfied. There are two approaches to handle the scheduling of tasks: time triggered or event triggered. The main characteristic of the first approach is that all activities are carried out at certain points in time known a prori. For this, all processes and their time specifications must be known in advance. Otherwise, an efficient implementation is not possible. Furthermore, the communication and the task scheduling on the control units have to be synchronized during operation in order to ensure the strict timing specifications of the system design (Albert, 2004). In this case the task execution schedule is defined off-line and the kernel follows it during run time. Once a feasible schedule is found, it is implemented with a cycle-executive that repeats itself each time. It is difficult to find an optimum schedule but onces it is found the implementation is simple and can be done with a look-up table. This approach does not allow a dynamic system to incorporate new tasks or applications. A modification on the number of executing tasks requires the recomputation of the schedule and this is rather complex to be implemented on

line. In the second approach, external or internal events are used to dispatch the different activities. This kind of designs involve creating systems which handle multiple interrupts. For example, interrupts may arise from periodic timer overflows, the arrival of messages on a CAN bus, the pressing of a switch, the completion of an analogue-to-digital conversion and so on. Tasks are ordered following a priority order and the highest priority one is dispatched each time. Usually, the kernel is based on a timer tick that preempts the current executing task and checks the ready queue for higher priority tasks. The priority disciplines most frequently used are round robin and fixed priorities. For example, the Department of Defense of the United States has adopted fixed priorities Rate Monotonic Sheduling (priority is assigned in reverse order to periods, giving the highest priority to the shortest period) and with this has made it a *de facto* standard Obenza (1993). The event triggered scheduling can introduce priority inversions, deadlocks and starvation if the access to shared resources and critical sections is not controlled in a proper manner. These problems are not acceptable in safety critical real-time applications. The main advantage of event-triggered systems is their ability to fastly react to asynchronous external events which are not known in advance (Albert & Gerth, 2003). In addition, event-triggered systems possess a higher flexibility and allow in many cases the adaptation to the actual demand without a redesign of the complete system (Albert, 2004).

2.4 Contention policies for shared resources and critical sections

Contention policies are fundamental in event-triggered schedulers. RTOSs have different approaches to handle this problem. A first solution is to leave the control mechanism in hands of the developers. This is a non-portable, costly and error prone solution. The second one implements a contention protocol based on priority inheritance (Sha et al., 1990). This solution bounds the priority inversions to the longest critical section of each lower priority task. It does not prevent deadlocks but eliminates the possibility of starvation. Finally, the Priority Ceiling Protocol (PCP) (Sha et al., 1990) and the Stack Resource Policy (SRP) (Baker, 1990) bound the priority inversion to the longest critical section of the system, avoid starvation and deadlocks. Both policies require an active kernel controlling semaphores and shared resources. The SRP performs better since it produces an early blocking avoiding some unnecessary preemptions present in the PCP. However, both approaches are efficient.

3. Real time operating system and their scope

This section presents a short review on some RTOS currently available. The list is not exhaustive as there are over forty academic and commercial developments. However, this section introduces the reader to a general view of what can be expected in this area and the kind of OS available for the development of real-time systems.

3.1 RTOS for mobile or small devices

Probably one of the most frequently used RTOS is Windows CE. Windows CE is now known as Windows Embedded and its family includes Windows Mobile and more recently Windows Phone 7 (*Windows Embedded*, 2011). Far from being a simplification of the well known OS from Microsoft, Windows CE is a RTOS with a relatively small footprint and is used in several embedded systems. In its actual version, it works on 32 bit processors and can be installed in 12 different architectures. It works with a timer tick or time quantum and provides 256 priority levels. It has a memory management unit and all processes, threads, mutexes, events

and semaphores are allocated in virtual memory. It handles an accuracy of one millisecond for SLEEP and WAIT related operations. The footprint is close to 400 KB and this is the main limitation for its use in devices with small memory address spaces like the ones present in wireless sensor networks microcontrollers.

eCos is an open source real-time operating system intended for embedded applications (*eCosCentric*, 2011). The configurability technology that lies at the heart of the eCos system enables it to scale from extremely small memory constrained SOC type devices to more sophisticated systems that require more complex levels of functionality. It provides a highly optimized kernel that implements preemptive real-time scheduling policies, a rich set of synchronization primitives, and low latency interrupt handling. The eCos kernel can be configured with one of two schedulers: The Bitmap scheduler and the Multi-Level Queue (MLQ) scheduler. Both are preemptible schedulers that use a simple numerical priority to determine which thread should be running. The number of priority levels is configurable up to 32. Therefore thread priorities will be in the range of 0 to 31, with 0 being the highest priority. The bitmap scheduler only allows one thread per priority level, so if the system is configured with 32 priority levels then it is limited to only 32 threads and it is not possible to preempt the current thread in favor of another one with the same priority. Identifying the highest-priority runnable thread involves a simple operation on the bitmap, and an array index operation can then be used to get hold of the thread data structure itself. This makes the bitmap scheduler fast and totally deterministic. The MLQ scheduler allows multiple threads to run at the same priority. This means that there is no limit on the number of threads in the system, other than the amount of memory available. However operations such as finding the highest priority runnable thread are a slightly bit more expensive than for the bitmap scheduler. Optionally the MLQ scheduler supports time slicing, where the scheduler automatically switches from one runnable thread to another when a certain number of clock ticks have occurred.

LynxOS (*LynxOS RTOS, The real-time operating system for complex embedded systems*, 2011) is a POSIX-compatible, multiprocess, multithreaded OS. It has a wide target of hardware architectures as it can work on complex switching systems and also in small embedded products. The last version of the kernel follows a microkernel design and has a minimum footprint of 28KB. This is about 20 times smaller than Windows CE. Besides scheduling, interrupt, dispatch and synchronize, there are additional services that are provided in the form of plug-ins so the designer of the system may choose to add the libraries it needs for a special purposes such as file system administration or TCP/IP support. The addition of these services obviously increases the footprint but they are optional and the designer may choose to have them or not. LynxOS can handle 512 priority levels and can implement several scheduling policies including prioritized FIFO, dynamic deadline monotonic scheduling, prioritized round robin, and time slicing among others.

FreeRTOS is an open source project (*The free RTOS Project*, 2011). It provides porting to 28 different hardware architectures. It is a multi-task operating system where each task has its own stack defined so it can be preempted and dispatched in a simple way. The kernel provides a scheduler that dispatches the tasks based on a timer tick according to a Fixed Priority policy. The scheduler consists of an only-memory-limited queue with threads of different priority. Threads in the queue that share the same priority will share the CPU with the round robin time slicing. It provides primitives for suspending, sleeping and blocking a task if a

synchronization process is active. It also provides an interrupt service protocol for handling I/O in an asynchronous way.

MaRTE OS is a Hard Real-Time Operating System for embedded applications that follows the Minimal Real-Time POSIX.13 subset (*Minimal Real-Time Operating System*, 2011). It was developed at University of Cantabria, Spain, and has many external contributions that have provided drivers for different communication interfaces, protocols and I/O devices. MaRTE provides an easy to use and controlled environment to develop multi-thread Real-Time applications. It supports mixed language applications in ADA, C and C++ and there is an experimental support for Java as well. The kernel has been developed with Ada2005 Real-Time Annex (*ISO/IEC 8526:AMD1:2007. Ada 2005 Language Reference Manual (LRM)*, 2005). Ada 2005 Language Reference Manual (LRM), 2005). It offers some of the services defined in the POSIX.13 subset like pthreads and mutexes. All the services have a time bounded response that includes the dynamic memory allocation. Memory is managed as a single address space shared by the kernel and the applications. MaRTE has been released under the GNU General Public License 2.

There are many other RTOS like SHArK (*S.Ha.R.K.: Soft Hard Real-Time Kernel*, 2007), Erika (*Erika Enterprise: Open Source RTOS for single- and multi-core applications*, 2011), SOOS (*Service Oriented Operating System*, 2011), that have been proposed in the academic literature to validate different scheduling and contention policies. Some of them can implement fault-tolerance and energy-aware mechanisms too. Usually written in C or C++ these RTOSs are research oriented projects.

3.2 General purpose RTOS

VxWorks is a proprietary RTOS. It is cross-compiled in a standard PC using both Windows or Linux (*VxWorks RTOS*, 2011). It can be compiled for almost every hardware architecture used in embedded systems including ARM, StrongARM and xScale processors. It provides mechanisms for protecting memory areas for real-time tasks, kernel and general tasks. It implements mutual exclusion semaphores with priority inheritance and local and distributed messages queues. It is able to handle different file systems including high reliability file systems and network file systems. It provides the necessary elements to implement the Ipv6 networking stack. There is also a complete development utility that runs over Eclipse.

RT-Linux was developed at the New Mexico School of Mines as an academic project (*RTLinuxFree*, 2011)(RTLinuxFree, 2011). The idea is simple and consists in turning the base GNU/Linux kernel into a thread of the Real-Time one. In this way, the RTKernel has control over the traditional one and can handle the real-time applications without interference from the applications running within the traditional kernel. Later RT-Linux was commercialized by FMLabs and finally by Wind River that also commercializes VxWorks. GNU/Linux drivers handle almost all I/O. First-In-First-Out pipes (FIFOs) or shared memory can be used to share data between the operating system and RTCore. Several distributions of GNU/Linux include RTLinux as an optional package.

RTAI is another real-time extension for GNU/Linux (*RTAI - the RealTime Application Interface for Linux*, 2010). It stands for Real-Time Application Interface. It was developed for several hardware architectures such as x86, x86_64, PowerPC, ARM and m68k. RTAI consists in a patch that is applied to the traditional GNU/Linux kernel and provides the necessary real-time primitives for programming applications with time constraints. There is also a

toolchain provided, RTAI-Lab, that facilitates the implementation of complex tasks. RTAI is not a commercial development but a community effort with base at University of Padova.

QNX is a unix like system that was developed in Canada. Since 2009 it is a proprietary OS (*QNX RTOS v4 System Documentation*, 2011). It is structured in a microkernel fashion with the services provided by the OS in the form of servers. In case an specific server is not required it is not executed and this is achieved by not starting it. In this way, QNX has a small footprint and can run on many different hardware platforms. It is available for different hardware platforms like the PowerPC, x86 family, MIPS, SH-4 and the closely related family of ARM, StrongARM and XScale CPUs. It is the main software component for the Blackberry PlayBook. Also Cisco has derived an OS from QNX.

OSE is a proprietary OS (*Enea OSE*, 2011). It was originally developed in Sweden. Oriented to the embedded mobile systems market, this OS is installed in over 1.5 billion cell phones in the world. It is structured in a microkernel fashion and is developed by telecommunication companies and thus it is specifically oriented to this kind of applications. It follows an event driven paradigm and is capable of handling both periodic and aperiodic tasks. Since 2009, an extension to multicore processors has been available.

4. Real-time programming languages

Real-time software is necessary to comply not only with functional application requirements but also with non functional ones like temporal restrictions. The nature of the applications requires a bottom-up approach in some cases a top-down approach in others. This makes the programming of real-time systems a challenge because different development techniques need to be implemented and coordinated for a successful project.

In a bottom-up approach one programming language that can be very useful is assembler. It is clear that using assembler provides access to the registers and internal operations of the processor. It is also well known that assembler is quite error prone as the programmer has to implement a large number of code lines. The main problem however is that using assembler makes the software platform dependent on the hardware and it is almost impossible to port the software to another hardware platform. Another language that is useful for a bottom-up approach is C. C provides an interesting level of abstraction and still gives access to the details of the hardware, thus allowing for one last optimization pass of the code. There are C compilers developed for almost every hardware platform and this gives an important portability to the code. The characteristics of C limits the software development in some cases and this is why in the last few years the use of C++ has become popular. C++ extends the language to include an object-oriented paradigm. The use of C++ provides a more friendly engineering approach as applications can be developed based on the object- oriented paradigm with a higher degree of abstraction facilitating the modeling aspects of the design. C++ compilers are available for many platforms but not for so many as in the C case. With this degree of abstraction, ADA is another a real-time language that provides resources for many different aspects related to real-time programming as tasks synchronization and semaphores implementations. All the programming languages mentioned up to now require a particular compiler to execute them on a specific hardware platform. Usually the software is customized for that particular platform. There is another approach in which the code is written once and runs anywhere. This approach requires the implementation of a virtual machine that deals with the particularities of the operating system and hardware platform. The virtual machine

presents a simple interface for the programmer, who does not have to deal with these details. Java is probably the most well known WORA language and has a real-time extension that facilitates the real-time programming.

In the rest of this section the different languages are discussed highlighting their pros and cons in each case are given so the reader can decide which is the best option for his project.

4.1 Assembler

Assembler gives the lowest possible level access to the microprocessor architecture such as registers, internal memory, I/O ports and interrupts handling. This direct access provides the programmer with full control over the platform. With this kind of programming, the code has very little portability and may produce hazard errors. Usually the memory management, allocation of resources and synchronization become a cumbersome job that results in very complex code structures. The programmer should be specialized on the hardware platform and should also know the details of the architecture to take advantage of such a low level programming. Assembler provides predictability on execution time of the code as it is possible to count the clock states to perform a certain operation.

There is total control over the hardware and so it is possible to predict the instant at which the different activities are going to be done.

Assembler is used in applications that require a high degree of predictability and are specialized on a particular kind of hardware architecture. The verification, validation and maintenance of the code is expensive. The life time of the software generated with this language is limited by the end-of-life of the hardware.

The cost associated to the development of the software, which is high due to the high degree of specialization, the low portability and the short life, make Assembler convenient only for very special applications such as military and space applications.

4.2 C

C is a language that was developed by Denis Ritchie and Brian Kernighan. The language is closely related to the development of the Unix Operating System. In 1978 the authors published a book of reference for programming in C that was used for a 25 years. Later, C was standardized by ANSI and the second edition of the book on included the changes incorporated in the standardization of the language (*ISO/IEC 9899:1999 - Programming languages - C*, 1999). Today, C is taught in all computer science and engineering courses and has a compiler for almost every available hardware platform.

C is a function oriented language. This important characteristic allows the construction of special purpose libraries that implement different functions like Fast Fourier Transforms, Sums of Products, Convolutions, I/O ports handling or Timing. Many of these are available for free and can be easily adapted to the particular requirements of a developer.

C offers a very simple I/O interface. The inclusion of certain libraries facilitates the implementation of I/O related functions. It is also possible to construct a Hardware Adaptation Layer in a simple way and introduce new functionalities in this way . Another important aspect in C is memory management. C has a large variety of variable types that

include, among others, char, int, long, float and double. C is also capable of handling pointers to any of the previous types of variables and arrays. The combination of pointers, arrays and types produce such a rich representation of data that almost anything is addressable. Memory management is completed with two very important operations: `calloc` and `malloc` that reserve space memory and the corresponding `free` operation to return the control of the allocated memory to the operating system.

The possibility of writing a code in C and compiling it for almost every possible hardware platform, the use of libraries, the direct access and handling of I/O resources and the memory management functions constitute excellent reasons for choosing this programming language at the time of developing a real-time application for embedded systems.

4.3 C++

The object-oriented extension of C was introduced by Bjarne Stroustrup in 1985. In 1999 the language received the status of standard (*ISO/IEC 14882:2003 - Programming languages C++*, 2003). C++ is backward compatible with C. That means that a function developed in C can be compiled in C++ without errors. The language introduces the concept of Classes, Constructors, Destructors and Containers. All these are included in an additional library that extends the original C one.

In C++ it is possible to do virtual and multiple inheritance. As an object oriented language it has a great versatility for implementing complex data and programming structures. Pointers are extended and can be used to address classes and functions enhancing the rich addressable elements of C. These possibilities require an important degree of expertise for the programmer as the possibility of introducing errors is important.

C++ compilers are not as widespread as the C ones. Although the language is very powerful in the administration of hardware, memory management and modeling, it is quite difficult to master all the aspects it includes. The lack of compilers for different architectures limits its use for embedded systems. Usually, software developers prefer the C language with its limitations to the use of the C++ extensions.

4.4 ADA

Ada is a programming language developed for real-time applications (*ISO/IEC 8526:AMD1:2007. Ada 2005 Language Reference Manual (LRM)*, 2005). Like C++ it supports structured and object-oriented programming but also provides support for distributed and concurrent programming. Ada provides native synchronization primitives for tasks. This is important when dealing with real-time systems as the language provides the tools to solve a key aspect in the programming of this kind of systems. Ada is used in large scale programs. The platforms usually involve powerful processors and large memory spaces. Under these conditions Ada provides a very secure programming environment. On the other hand, Ada is not suitable for small applications running on low end processors like the ones implementing wireless sensors networks with reduced memory spaces and processor capacities.

Ada uses a safe type system that allows the developer to construct powerful abstractions reflecting the real world while the compiler can detect logic errors. The software can be built in modules facilitating development of large systems by teams. It also separates interfaces from

implementation providing control over visibility. The strict definition of types and the syntax allow the code to be compiled without changes on different compliant compilers on different hardware platforms. Another important feature is the early standardization of the language. Ada compilers are officially tested and are accepted only after passing the test for military and commercial work. Ada also has support for low level programming features. It allows the programmer to do address arithmetic, directly access to memory address space, perform bit wise operations and manipulations and the insert of machine code. Thus Ada is a good choice for programming embedded systems with real-time or safety-critical applications. These important features have facilitated the maintainability of the code across the life time of the software and this facilitates its use in aerospace, defense, medical, rail-road and nuclear applications.

4.5 C#

Microsoft's integrated development environment (.NET) includes a new programming language C# which targets the .NET Framework. Microsoft does not claim that C# and .NET are intended for real-time systems. In fact, C# and the .NET platform do not support many of the thread management constructs that real-time systems, particularly hard ones, often require. Even Anders Hejlsberg (Microsoft's C# chief architect) states, "I would say that 'hard real-time' kinds of programs wouldn't be a good fit (at least right now)" for the .NET platform (Lutz & Laplante, 2003). For instance, the Framework does not support thread creation at a particular instant in time with the guarantee that it will be completed by a certain in time. C# supports many thread synchronization mechanisms but none with high precision.

Windows CE has significantly improved thread management constructs. If properly leveraged by C# and the .NET Compact Framework, it could potentially provide a reasonably powerful thread management infrastructure. Current enumerations for thread priority in the .NET Framework, however, are largely unsatisfactory for real-time systems. Only five levels exist: AboveNormal, BelowNormal, Highest, Lowest, and Normal. By contrast Windows CE, specifically designed for real time systems has 256 thread priorities. Microsoft's ThreadPriority enumeration documentation also states that "the scheduling algorithm used to determine the order of thread execution varies with each operating system." This inconsistency might cause real-time systems to behave differently on different operating systems.

4.6 Real-time java

Java includes a number of technologies ranging from JavaCard applications running in tens of kilobytes to large server applications running with the Java 2 Enterprise Edition requiring many gigabytes of memory. In this section, the Real-time specification for Java (RTSJ) is described in detail. This specification proposes a complete set of tools to develop real-time applications. None of the other languages used in real-time programming provide classes, templates and structures on which the developer can build the application. When using other languages, the programmer needs to construct classes, templates and structures and then implement the application taking care of the scheduler, periodic and sporadic task handling and the synchronization mechanism.

RTSJ is a platform developed to handle real-time applications on top of a Java Virtual Machine (JVM). The JVM specification describes an abstract stack machine that executes

bytecodes, the intermediate code of the Java language. Threads are created by the JVM but are eventually scheduled by the operating system scheduler over which it runs. The Real-Time Specification for Java (Gosling & Bollella, 2000; Microsystems, 2011) provides a framework for developing real-time scheduling mostly on uniprocessors systems. Although it is designed to support a variety of schedulers only the `PriorityScheduler` is currently defined and is a preemptive fixed priorities one (FPP). The implementation of this abstraction could be handled either as a middleware application on top of stock hardware and operating systems or by a direct hardware implementation (Borg et al., 2005). RTS Java guarantees backward compatibility so applications developed in traditional Java can be executed together with real-time ones. The specification requires an operating system capable of handling real-time threads like RT-Linux. The indispensable OS capabilities must include a high-resolution timer, program-defined low-level interrupts, and a robust priority-based scheduler with deterministic procedures to solve resource sharing priority inversions. RTSJ models three types of tasks: Periodic, Sporadic and Aperiodic. The specification uses a FPP scheduler (`PriorityScheduler`) with 28 different priority levels. These priority levels are handled under the `Schedulable` interface which is implemented by two classes: `RealtimeThread` and `AsyncEventHandler`. The first ones are tasks that run under the FPP scheduler associated to one of the 28 different priority levels and are implementations of the `javax.realtime.RealtimeThread`, `RealtimeThread` for short. Sporadic tasks are not in the FPP scheduler and are served as soon as they are released by the `AsyncEventHandler`. The last ones do not have known temporal parameters and are handled as standard `java.lang.Thread` (Microsystems, 2011). There are two classes of parameters that should be attached to a schedulable real-time entity. The first one is specified in the class `SchedulingParameters`. In this class the parameters that are necessary for the scheduling, for example the priority, are defined. The second one, is the class `ReleaseParameters`. In this case, the parameters related to the mode in which the activation of the thread is done such as period, worst case computation time, and offset are defined.

Traditional Java uses a Garbage Collector (GC) to free the region of memory that is not referenced any more. The normal memory space for Java applications is the `HeapMemory`. The GC activity interferes with the execution of the threads in the JVM. This interference is unacceptable in the real-time domain as it imposes blocking times for the currently active threads that are neither bounded nor can they be determined in advance. To solve this, the real-time specification introduces a new memory model to avoid the interference of the GC during runtime. The abstract class `MemoryArea` models the memory by dividing it in regions. There are three types of memory: `HeapMemory`, `ScopedMemory` and `InmortalMemory`. The first one is used by non real time threads and is subject to GC activity. The second one, is used by real time threads and is a memory that is used by the thread while it is active and it is immediately freed when the real-time thread stops. The last one is a very special type of memory that should be used very carefully as even when the JVM finishes it may remain allocated. The RTSJ defines a sub-class `NoHeapRealtimeThread` of `RealtimeThread` in which the code inside the method `run()` should not reference any object within the `HeapMemory` area. With this, a real-time thread will preempt the GC if necessary. Also when specifying an `AsyncEventHandler` it is possible to avoid the use of `HeapMemory` and define instead the use of `ScopedMemory` in its constructor.

4.6.1 Contention policy for shared resources and task synchronization

The RTSJ virtual machine supports priority-ordered queues and performs by default a basic priority inheritance and a ceiling priority inheritance called priority ceiling emulation. The priority inheritance protocol has the problem that it does not prevent deadlocks when a wrong nested blocking occurs. The priority ceiling protocol avoids this by assigning a ceiling priority to a critical section which is equal to the highest priority of any task that may lock it. This is effective but it is more complex to implement. The mix of the two inheritance protocols avoid unbounded priority inversions caused by low priority thread locks.

Each thread has a base and an active priority. The base priority is the priority allocated by the programmer. The active priority is the priority that the scheduler uses to sort the run queue. As mentioned before, the real-time JVM must support priority-ordered queues and perform priority inheritance whenever high priority threads are blocked by low priority ones. The active priority of a thread is, therefore, the maximum of its base priority and the priority it has inherited.

The RTSJ virtual machine supports priority-ordered queues and performs by default a basic priority inheritance and a ceiling priority inheritance called priority ceiling emulation. The priority inheritance protocol has the problem that it does not prevent deadlocks when a wrong nested blocking occurs. The priority ceiling protocol avoids this by assigning a ceiling priority to a critical section which is equal to the highest priority of any task that may lock it. This is effective but it is more complex to implement. The mix of the two inheritance protocols avoid unbounded priority inversions caused by low priority threads locks.

Each thread has a *base* and an *active* priority. The base priority is the priority allocated by the programmer. The active priority is the priority that the scheduler uses to order the run queue. As mentioned before, the real-time JVM must support priority-ordered queues and perform priority inheritance whenever high priority threads are blocked by low priority ones. The active priority of a thread is, therefore, the maximum of its base priority and the priority it has inherited.

4.7 C/C++ or RTJ

In real-time embedded systems development flexibility, predictability and portability are required at the same time. Different aspects such as contention policies implementation and asynchronous handling, are managed naturally in RTSJ. Other languages, on the other hand, require a careful programming by the developer. However, RTSJ has some limitations when it is used in small systems where the footprint of the system should be kept as small as possible. In the last few years, the development of this kind of systems has been dominated by C/C++. One reason for this trend is that C/C++ exposes low-level system facilities more easily and the designer can provide ad-hoc optimized solutions in order to reach embedded-system real time requirements. On the other hand, Java runs on a Virtual Machine, which protects software components from each other. In particular, one of the common errors in a C/C++ program is caused by the memory management mechanism of C/C++ which forces the programmers to allocate and deallocate memory manually. Comparisons between C/C++ and Java in the literature recognize pros and cons for both. Nevertheless, most of the ongoing research on this topic concentrates on modifying and adapting Java. This is because its environment presents some attributes that make it attractive for real-time developers. Another interesting attribute from a software designer point of view is that Java has a powerful, portable and continuously

updated standard library that can reduce programming time and costs. In Table 1 the different aspects of the languages discussed are summarized. *VG* stands for very good, *G* for good, *R* for regular and *B* for bad.

Language	Portability	Flexibility	Abstraction	Resource Handling	Predictability
Assembler	*B*	*B*	*B*	*VG*	*VG*
C	*G*	*G*	*G*	*VG*	*G*
C++	*R*	*VG*	*VG*	*VG*	*G*
Ada	*R*	*VG*	*VG*	*VG*	*G*
RTSJ	*VG*	*VG*	*VG*	*R*	*R*

Table 1. Languages characteristics

5. Java implementations

In this section different approaches to the implementation of Java are presented. As explained, a java application requires a virtual machine. The implementation of the JVM is a fundamental aspect that affects the performance of the system. There are different approaches for this. The simplest one, resolves everything at software level. The jave bytecodes of the application are interpreted by the JVM that passes the execution code to the RTOS and this dispatches the thread. Another option consists in having a Just in Time (JIT) compiler to transform the java code in machine code and directly execute it within the processor. And finally, it is possible to implement the JVM in hardware as a coprocessor or directly as a processor. Each solution has pros and cons that are discussed in what follows for different cases. Figure 1 shows the different possibilities in a schematic way.

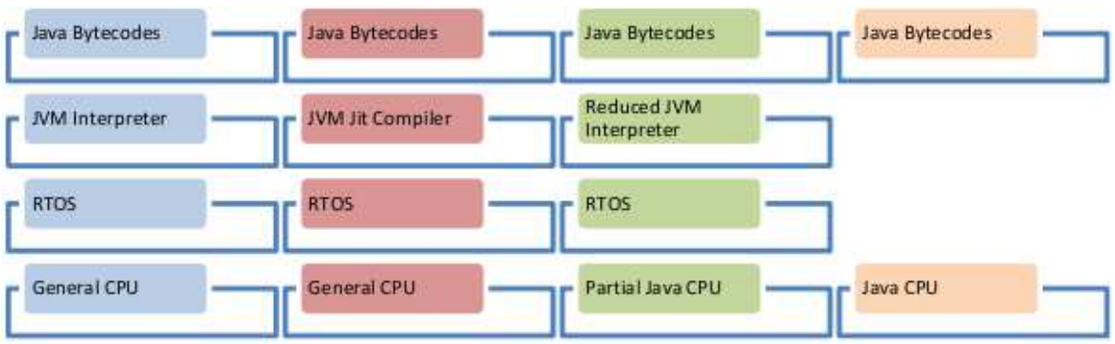


Fig. 1. Java layered implementations

In the domain of small embedded devices, the JVM turns out to be slow and requires an important amount of memory resources and processor capabilities. These are serious drawbacks to the implementation of embedded systems with RTSJ. In order to overcome these problems, advances in JIT compilers promote them as the standard execution mode of the JVM in desktop and server environments. However, this approach introduces uncertainties to the execution time due to runtime compilation. Thus execution times are not predictable and this fact prevents the computation of the WCET forbidding its use in hard real-time applications. Even if the program execution speeds up, it still requires an important amount of memory. The solution is not practical for small embedded systems.

In the embedded domain, where resources are scarce, a Java processors or coprocessors are more promising options. There are two types of hardware JVM implementations:

- A coprocessor works in concert with a general purpose processor translating java byte codes to a sequence of instructions specific to this coupled CPU.
- Java chips entirely replace the general CPU. In the Java Processors the JVM bytecode is the native instruction set, therefore programs are written in Java. This solution can result in quite a small processor with little memory demand.

In the embedded domain, where resources are scarce, a Java processors or coprocessors are more promising options. There are two types of hardware JVM implementations:

- A coprocessor works in concert with a general purpose processor translating java bytecodes to a sequence of instructions specific for this coupled CPU.
- Java chips entirely replace the general CPU. In the Java Processors the JVM bytecode is the native instruction set, therefore programs are written in Java. This solution can result in quite a small processor with little memory demand.

Table 2 shows a short list of Java processors.

Name	Target technology	Size	Speed [MHz]
JOP	Altera, Xilinx FPGA	2050 LCs, 3KB Ram	100
picoJava	No realization	128K gates, 38KB	
picoJava II	Altera Cyclone FPGA	27.5 K LCs; 47.6 KB	
aJile	aJ102 aJ200 ASIC	0.25μ	100
Cjip	ASIC 0.35μ	70K gates, 55MB ROM, RAM	80
Moon	Altera FPGA	3660 LCs, 4KB RAM	
Lightfoot	Xilinx FPGA	3400 LCs	40
LavaCORE	Xilinx FPGA	3800 LCs 30K gates	33
Komodo		2600 LCs	33
FemtoJava	Xilinx FPGA	2710 LCs	56

Table 2. Java Processors List

In 1997 Sun introduced the first version of picoJava and in 1999 it launched the picoJava-II processor. Its core provides an optimized hardware environment for hosting a JVM implementing most of the Java virtual machine instructions directly. Java bytecodes are directly implemented in hardware. The architecture of picoJava is a stack-based CISC processor implementing 341 different instructions (O'Connor & Tremblay, 1997). Simple Java bytecodes are directly implemented in hardware and some performance critical instructions are implemented in microcode. A set of complex instructions are emulated by a sequence of simpler instructions. When the core encounters an instruction that must be emulated, it generates a trap with a trap type corresponding to that instruction and then jumps to an emulation trap handler that emulates the instruction in software. This mechanism has a high variability latency that prevents its use in real-time because of the difficulty to compute the WCET (Borg et al., 2005; Puffitsch & Schoeberl, 2007).

Komodo (Brinkschulte et al., 1999) is a Java microcontroller with an event handling mechanism that allows handling of simultaneous overlapping events with hard real-time

requirements. The Komodo microcontroller design adds multithreading to a basic Java design in order to attain predictability of real time threads requirements. The exclusive feature of Komodo is the instruction fetch unit with four independent program counters and status flags for four threads. A priority manager is responsible for hardware real-time scheduling and can select a new thread after each bytecode instruction. The microcontroller holds the contexts of up to four threads. To scale up for larger systems with more than three real-time threads the authors suggest a parallel execution on several microcontrollers connected by a middleware platform.

FemtoJava is a Java microcontroller with a reduced-instruction-set Harvard architecture (Beck & Carro, 2003). It is basically a research project to build an -application specific- Java dedicated microcontroller. Because it is synthesized in an FPGA, the microcontroller can also be adapted to a specific application by adding functions that could includes new Java instructions. The bytecode usage of the embedded application is analyzed and a customized version of FemtoJava is generated (similar to LavaCORE) in order to minimize resource usage: power consumption, small program code size, microarchitecture optimizations (instruction set, data width, register file size) and high integration (memory communications on the same die).

Hardware designs like JOP (Java Optimized Processor) and AONIX PERC processors currently provide a safety certifiable, hard real-time virtual machine that offers throughput comparable to optimized C or C++ solutions (Schoeberl, 2009)

The Java processor JOP (Altera or Xilinx FPGA) is a hardware implementation of the Java virtual machine (JVM). The JVM bytecodes are the native instruction set of JOP. The main advantage of directly executing bytecode instructions is that WCET analysis can be performed at the bytecode level. The WCET tool WCA is part of the JOP distribution. The main characteristics of JOP architecture are presented in (Schoeberl, 2009). They include a dynamic translation of the CISC Java bytecodes to a RISC stack based instruction set that can be executed in a three microcode pipeline stages: microcode fetch, decode and execute. The processor is capable of translating one bytecode per cycle giving a constant execution time for all microcode instructions without any stall in the pipeline. The interrupts are inserted in the translation stage as special bytecodes and are transparent to the microcode pipeline. The four stages pipeline produces short branch delays. There is a simple execution stage with the two top most stack elements (registers A and B). Bytecodes have no time dependencies and the instructions and data caches are time-predictable since ther are no prefetch or store buffers (which could have introduced unbound time dependencies of instructions). There is no direct connection between the core processor and the external world. The memory interface provides a connection between the main memory and the core processor.

JOP is designed to be an easy target for WCET analysis. WCET estimates can be obtained either by measurement or static analysis. (Schoeberl, 2009) presents a number of performance comparisons and finds that JOP has a good average performance relative to other non real-time Java processors, in a small design and preserving the key characteristics that define a RTS platform. A representative ASIC implementation is the aJile aJ102 processor (*Ajile Systems*, 2011). This processor is a low-power SOC that directly executes Java Virtual Machine (JVM) instructions, real-time Java threading primitives, and secured networking. It is designed for a real-time DSP and networking. In addition, the aJ-102 can execute bytecode extensions for custom application accelerations. The core of the aJ102 is the JEMCore-III

low-power direct execution Java microprocessor core. The JEMCore-III implements the entire JVM bytecode instructions in silicon.

JOP includes an internal microprogrammed real-time kernel that performs the traditional operating system functions such as scheduling, context switching, interrupt preprocessing, error preprocessing, and object synchronization. As explained above, a low-level analysis of execution times is of primary importance for WCET analysis. Even though the multiprocessors systems are a common solution to general purpose equipments it makes static WCET analysis practically impossible. On the other hand, most real-time systems are multi-threaded applications and performance could be highly improved by using multi core processors on a single chip. (Schoeberl, 2010) presents an approach to a time-predictable chip multiprocessor system that aims to improve system performance while still enabling WCET analysis. The proposed chip uses a shared memory statically scheduled with a time-division multiple access (TDMA) scheme which can be integrated into the WCET analysis. The static schedule guarantees that thread execution times on different cores are independent of each other.

6. Conclusions

In this chapter a critical review of the state of the art in real-time programming languages and real-time operating systems providing support to them has been presented. The programming lan guages are limited mainly to five: C, C++, Ada, RT Java and for very specific applications, Assembler. The world of RTOS is much wider. Virtually every research group has created its own operating system. In the commercial world there is also a range of RTOS. At the top of the preferences appear Vxworks, QNX, Windows CE family, RT Linux, FreeRTOS, eCOS and OSE. However, there are many others providing support in particular areas. In this paper, a short list of the most well known ones has been described.

At this point it is worth asking why while there are so many RTOSs available there are so few programming languages. The answer probably is that while a RTOS is oriented to a particular application area such as communications, low end microprocessors, high end microprocessors, distributed systems, wireless sensors network and communications among others, the requirements are not universal. The programming languages, on the other hand need to be and are indeed universal and useful for every domain.

Although the main programming languages for real-time embedded systems are almost reduced to five the actual trend reduces these to only C/C++ and RT Java. The first option provides the low level access to the processor architecture and provides an object oriented paradigm too. The second option has the great advantage of a WORA language with increasing hardware support to implement the JVM in a more efficient.

In the last few years, there has been an important increase in ad-hoc solutions based on special processors created for specific domains. The introduction of Java processors changes the approach to embedded systems design since the advantages of the WORA programming are added to a simple implementation of the hardware.

The selection of an adequate hardware platform, a RTOS and a programming language will be tightly linked to the kind of embedded system being developed. The designer will choose the combination that best suits the demands of the application but it is really important to select one that has support along the whole design process.

7. References

Ajile Systems (2011). http://www.ajile.com/.

Albert, A. (2004). Comparison of event-triggered and time-triggered concepts with regard to distributed control systems, *Embedded World 2004*, pp. 235–252.

Albert, A. & Gerth, W. (2003). Evaluation and comparison of the real-time performance of can and ttcan, *9th international CAN in Automation Conference*, p. 05/01–05/08.

Baker, T. (1990). A stack-based resource allocation policy for realtime processes, *Real-Time Systems Symposium, 1990. Proceedings., 11th* pp. 191–200.

Beck, A. & Carro, L. (2003). Low power java processor for embedded applications, *12th IFIP International Conference on Very Large Scale Integration*.

Borg, A., Audsley, N. & Wellings, A. (2005). Real-time java for embedded devices: The javamen project, *Perspectives in Pervasive Computing*, pp. 1–10.

Brinkschulte, U., Krakowski, C., Kreuzinger, J. & Ungerer, T. (1999). A multithreaded java microcontroller for thread-oriented real-time event-handling, *Parallel Architectures and Compilation Techniques, 1999. Proceedings. 1999 International Conference on*, pp. 34 –39.

eCosCentric (2011). http://www.ecoscentric.com/index.shtml.

Enea OSE (2011). http://www.enea.com/software/products/rtos/ose/.

Erika Enterprise: Open Source RTOS for single- and multi-core applications (2011). http://www.evidence.eu.com/content/view/27/254/.

Gosling, J. & Bollella, G. (2000). *The Real-Time Specification for Java*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

IEEE (2003). *ISO/IEC 9945:2003, Information Technology–Portable Operating System Interface (POSIX)*, IEEE.

ISO/IEC 14882:2003 - Programming languages C++ (2003).

ISO/IEC 8526:AMD1:2007. Ada 2005 Language Reference Manual (LRM) (2005). http://www.adaic.org/standards/05rm/html/RM-TTL.html.

ISO/IEC 9899:1999 - Programming languages - C (1999). http://www.open-std.org/JTC1/SC22/WG14/ www/docs/n1256.pdf.

Lutz, M. & Laplante, P. (2003). C# and the .net framework: ready for real time?, *Software, IEEE* 20(1): 74–80.

LynxOS RTOS, The real-time operating system for complex embedded systems (2011). http://www.lynuxworks.com/rtos/rtos.php.

Maglyas, A., Nikula, U. & Smolander, K. (2010). Comparison of two models of success prediction in software development projects, *6th Central and Eastern European Software Engineering Conference (CEE-SECR), 2010*, pp. 43–49.

Microsystems, S. (2011). Real-time specification for java documentation, http://www.rtsj.org/.

Minimal Real-Time Operating System (2011). http://marte.unican.es/.

Obenza, R. (1993). Rate monotonic analysis for real-time systems, *Computer* 26: 73–74. URL: *http://portal.acm.org/citation.cfm?id=618978.619872*

O'Connor, J. & Tremblay, M. (1997). picojava-i: the java virtual machine in hardware, *Micro, IEEE* 17(2): 45 –53.

Pleunis, J. (2009). Extending the lifetime of software-intensive systems, *Technical report*, Information Technology for European Advancement, http://www.itea2.org/innovation_reports.

Puffitsch, W. & Schoeberl, M. (2007). picojava-ii in an fpga, *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, JTRES '07, ACM, New York, NY, USA, pp. 213–221.
URL: *http://doi.acm.org/10.1145/1288940.1288972*

QNX RTOS v4 System Documentation (2011). http://www.qnx.com/developers/qnx4/documentation.html.

Robertz, S. G., Henriksson, R., Nilsson, K., Blomdell, A. & Tarasov, I. (2007). Using real-time java for industrial robot control, *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, JTRES '07, ACM, New York, NY, USA, pp. 104–110.
URL: *http://doi.acm.org/10.1145/1288940.1288955*

RTAI - the RealTime Application Interface for Linux (2010). https://www.rtai.org/.

RTLinuxFree (2011). http://www.rtlinuxfree.com/.

Schoeberl, M. (2009). *JOP Reference Handbook: Building Embedded Systems with a Java Processor*, number ISBN 978-1438239699, CreateSpace. Available at http://www.jopdesign.com/doc/handbook.pdf.
URL: *http://www. jopdesign.com/ doc/handbook.pdf*

Schoeberl, M. (2010). Time-predictable chip-multiprocessor design, *Signals, Systems and Computers (ASILOMAR), 2010 Conference Record of the Forty Fourth Asilomar Conference on*, pp. 2116 –2120.

Service Oriented Operating System (2011). http://www.ingelec.uns.edu.ar/rts/soos.

Sha, L., Rajkumar, R. & Lehoczky, J. P. (1990). Priority inheritance protocols: An approach to real-time synchronization, *IEEE Trans. Comput.* 39(9): 1175–1185.

S.Ha.R.K.: Soft Hard Real-Time Kernel (2007). http://shark.sssup.it/.

Stankovic, J. A. (1988). Misconceptions about real-time computing, *IEEE Computer* 21(17): 10–19.

The Chaos Report (1994). www.standishgroup.com/sample_ research/PDFpages/Chaos1994.pdf.

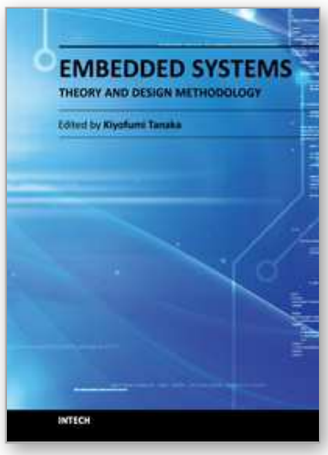
The free RTOS Project (2011). http://www.freertos.org/.

VxWorks RTOS (2011). http://www.windriver.com/products/vxworks/.

Windows Embedded (2011). http://www.microsoft.com/windowsembedded/en-us/develop/windows-embedded-products-for-developers.aspx.

Wolf, W. (2002). What is Embedded Computing?, *IEEE Computer* 35(1): 136–137.

Zerzelidis, A. & Wellings, A. (2004). Requirements for a real-time .net framework, *Technical Report YCS-2004-377*, Dep. of Computer Science, University of York.



Embedded Systems - Theory and Design Methodology
Edited by Dr. Kiyofumi Tanaka

ISBN 978-953-51-0167-3
Hard cover, 430 pages
Publisher InTech
Published online 02, March, 2012
Published in print edition March, 2012

Nowadays, embedded systems - the computer systems that are embedded in various kinds of devices and play an important role of specific control functions, have permitted various aspects of industry. Therefore, we can hardly discuss our life and society from now onwards without referring to embedded systems. For wide-ranging embedded systems to continue their growth, a number of high-quality fundamental and applied researches are indispensable. This book contains 19 excellent chapters and addresses a wide spectrum of research topics on embedded systems, including basic researches, theoretical studies, and practical work. Embedded systems can be made only after fusing miscellaneous technologies together. Various technologies condensed in this book will be helpful to researchers and engineers around the world.

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen