Working with Real-Time Systems
Technical Report

Mauricio Araya
Universidad Técnica Federico Santa María
Av.España 1680
Valparaíso, Chile
maray@inf.utfsm.cl
21 September 2007

Abstract
This report is a description of the important concepts and reasons that working with real-time requirements introduce. The document introduce the real-time concept, including a state of art discussion about hardness, deadline handling, embedded systems and real-time operating systems. Also, a practical experience is proposed to realize which are the complications of a real-time system, and how unsuitable is a general-purpose operating system.

1 Introduction

Real-Time Systems are spread everywhere nowadays, from specific-purpose embedded/dedicated systems to very large distributed controlling frameworks. However, the common knowledge about these systems is very limited, and sometimes simply wrong. The real-time concept, because of its colloquial use, commonly produce misgivings. The word real its often confused with fast or in user time. Even more, in the early days of computer science the real-time term was used to describe interactive or time-sharing systems [17]. Regardless, a RTS have an unmistakable property that define its nature: the logical correctness is as important as the time correctness [22]. This means that its not a sufficient condition to produce a correct answer to a given computation, but to deliver the results with correct timing. There is no direct relation with how quickly a computation is delivered, because an early delivery can be as ruinous as a delayed response. Also, there is no relation with user-time psychological perception, because real-time systems are used for controlling critical or embedded systems, with machine timing constraints that are almost completely incompatible with interactive systems.
When the time constraints are so important, that a catastrophic event occurs if the time window is not met, the system has hard real-time requirements [22]. For instance a military defense system cannot miss a time deadline in a missile trajectory calculation, because the consequences are indeed deathly situations. This catastrophic events includes not only life and dead situations, but heavy economical looses, whole system collapse or loosing unique opportunities. There are other kind of real-time systems that have less critical constraints called Soft Real-Time Systems. These systems executes the real-time tasks according to a desired schedule “on average” (but with deadlines). Some authors [3], state that there is no division line between a hard and a soft real time, because there is not such thing like a 100% deterministic CPU, so all real-time systems have to satisfy an “on the average” restriction. The hardness or softness of a real-time system will be then a fault tolerance level and not a tight classification of systems.

Regardless of whether the RT system is hard or soft, they have common desing challenges that always should be considerated[15].

- **Predictability**: well-defined, or deterministic operations with independent timing of surrounding environment.

- **User Control**: The user (programmer) has an ultimate control of the behavior of the system. The user can change dynamically the scheduling policy, the priorities of the RT tasks, the general throughput, etc.

- **Timeliness**: The temporal correctness must be ensure on a RTS, so the system must provide mechanisms which takes this issues in account.

- **Mission Oriented**: A RTS has multiple tasks, including system maintenance, desktop application, etc. But in a time period, a RTS must have only one (or very few) critical operations with an other status: very high priority tasks with a direct access to system hardware and peripherals [21].

### 1.1 Real-Time Programming

The bibliography of programming with real-time requirements is often related to a hardware-specific task [3] or closely embedded at the application area [28]. Due this fact, it is very difficult to distinguish between general contributions and system specific achievements. But, some efforts were made during the past decades on establishing general guidelines for RT Programming, in pursuit of a strong paradigm applicable to several systems. For instance, the very basic concept of priority-interrupt-driven programming [1], it is the general method (currently known as preemptiveness) that allows deadline handling using resource expropriation. Some formal approaches such the asynchronous message passing model [14] can be found in the early bibliography, but RT systems are more a practical challenge than a theoretical problem. From the practical riverside, ADA language is the most prolific early contributor for RT Programming, establishing a whole list of real-time requirements for ADA 9X.
such task scheduling, queuing order, asynchronous transfer of control, IPC and synchronization, interrupt entry binding, time specifications, etc. Modern approaches are more related to software engineering than direct RT programming, such using UML for real-time development [5][20], well defined interfaces for RT components [24], modular object oriented systems for developing distributed RT systems [4], high-level component programming and configuration [19], or end-to-end engineering of dynamic RT systems [18]. For more specific details on the C/C++ and Java real time APIs please refer to [13, 6, 7, 15].

1.2 Embedded Systems

An embedded system is a purpose-specific computer system that belongs to a larger system [2]. Embedded systems are commonly semi-autonomous entities of a complex system, that takes care of a specific hardware or human interface with very few responsibilities. A complex system should be understood as a system that manages many variables at the same time, such a nuclear power plant, an intelligent vehicle highway system, an air-traffic control system, or an antenna array. For instance, an embedded system of an antenna array can be a local control unit that controls the axis encoders and motors of one specific antenna. Embedded systems usually are considered as small, resource limited, and purpose-specific systems, but the increasing sophistication of microprocessors and memory devices is allowing to design configurable and programmable embedded systems as generic reallocable units rather than specifically manufactured drives [22]. Even more, off the shelf general purpose computers can be used for more soften real-time requirements, included as an embedded machine of a larger system [6]. John Stankovic et al [23] describes this last phenomena as the Open Real-Time Systems, over the general idea of developing general purpose embedded systems that permit a dynamic mix of multiple, and independently developed hardware and applications.

As embedded systems usually controls hardware, most of them are Real-Time systems, mainly because of the timeliness restrictions and the critical operations that hardware manipulation involves. In fact, there is a strong bond between both concepts; embedded operating systems usually support real-time requirements, and more over embedded hardware usually are designed to fulfill real-time requirements. A controlling system, such embedded systems that controls hardware, interacts with the environment using sensors and actuators. Sensors are used to get a snapshot of the reality and setup a model that drives the actuators behavior. The monitoring of the sensors and the actuators actions can be periodic, aperiodic or sporadic [22], defining the Real-Time requirements of the system. For instance, periodic requirements need more confiability on meeting the time window than aperiodic requirements, that need more straight-forward access to do the operations fast enough.
1.3 Real-Time Operating Systems

The main difference between a General Purpose Operating System (GPOS) and a Real-Time Operating System (RTOS), is that the former do not support a direct control of the microprocessor (neither peripherals) for timing issues. Then, the user (programmer) is unable to develop a program that meets timeliness requirements, because he has no control over the process scheduling, sleeping deadlines or buffer flushing. The part of an Operating System that controls the process scheduling and the direct access to hardware is the Kernel[21]. Therefore, a Real-Time Kernel must provide basic support for predictably satisfying real-time constraints, thread/process preemptibility, thread/process prioritity, deterministic synchronization mechanisms, predefined latencies and a specialized API for accessing this functionality (i.e. RTAI [6]).

There is a common standard for portable Real-Time systems as a part of the POSIX standard (1003.1) [11]. Most of the new RTOSs designers have adopted this standard, so previous work like Dragon Slayer [26], HARTOS [12], or ARTS [25] will be not discussed in the current paper. A detailed survey on the technical details that involved in early RTOS (before 1994) can be found in [9]. The POSIX extension for Real-Time (1003.1b) [10], includes the technical definitions regarding developing a RTOS, such process primitives, process environment, files/directory access, I/O primitives, device functions, synchronization, memory management, scheduling, clock & timers, and message passing functions. Indeed, these are the essential parts of a general-purpose OS [21]. As an example of this standard, in the Clocks and Timers section, the CLOCK_REALTIME identifier is defined, including the minimum POSIX resolution of 20[ms]. Obviously this number can be much lesser in a specific implementation. This standard not only motivates portable applications and compatible RTOS, but provides a research base with practical implications. For instance, Li et al presents a formal verified framework for Real-Time Scheduling manipulation on POSIX compliant RTOSs [8]. A survey on the most popular commercial POSIX compliant RTOSs can be found in [2].

2 Fundamental Real-Time Concepts

The concept that defines a RT Systems is the deadline awareness, understanding a deadline as a critical time point, when the conditions of the system or environment changes if the proper response has not been sent. In the general design of computer systems, deadlines can be slightly violated meanwhile the violations are not too much for a certain period, because systems usually can continue, restart or at least roll back to a recovery point [?]. In hard RT systems, there is no such recovery or restart from a deadline violation. Regardless, RT systems are still exposed to hardware or other failures, so the system not only have to delivers the normal operation results at the correct timing, but do so when the hardware (or other component) fails. Then, a hard real-time
system should always have fault tolerance mechanisms to avoid deadlines violations at any cost. To do so, the system have to be aware of the deadlines and handle them in a proper way. Unfortunately, proper deadline handling always depends on the requirements, so there is not a general method to manage deadlines. However, a proper analysis of the system requirements could help to not “rebuilding the wheel” every time.

A formal approach to deadline analysis is the service utility function (SUF), that describes how useful is a given response in terms of the time slice from the deadline. For instance, a possible SUF can be:

\[
SUF(t_i) = \begin{cases} 
\frac{t_i}{t_{dl}} & \text{if } t_i \leq t_{dl} \\
\frac{t_i - t_{dl}}{t_{dl}} & \text{if } t_i < t_{dl} \\
0 & \text{if } t_i > t_{max} 
\end{cases} 
\]  

Where \( t_i \) is the response time, \( t_{dl} \) the given deadline, and \( t_{max} \) the maximum time when the response is completely useless. \( SUF(t_i) = 1 \) means that the response is 100% useful, and \( SUF(t_i) = 0 \) that is useless. This particular SUF is isochronal, meaning that an early response is as much unwanted as a delayed response. Also, this function is clearly from a soft RT requirement, because delayed responses are unwanted, but acceptable for a time while. Hard real-time requirements will always have a \( SUF(t_i) = 0 \) when \( t_i > t_{dl} \), but more complex functions can be found when \( t_i < t_{dl} \).

Another important concept on deadline analysis, is the degree of timing precision (DTP) \([13]\), that takes into account the response precision of the RT requirement. This quantity helps to select the proper hardware and software for a given problem. For instance, if the DTP of a problem is near a minute, almost any desktop PC, even over a very slow network, will be capable to manage it. But if the requirements are near microseconds, as we will demonstrate later, no general-purpose PC is capable of that resolution with accuracy. More over, if the DTP is near the nanoseconds, no classic computer system can be used, and the designer should think in a specialized electronic device that supports that speed.

2.1 RTOS Concepts

Modern computer systems are interrupt-driven, meaning that the operating system services and mechanisms usually works using the interrupt hardware support. Real-time Systems are event-driven, so, theoretically, a direct mapping can be made with an event and an interrupt. The basic principle of an interrupt is to expropriate the processor for a while to manage hardware calls, task switching, exceptions or user-space system calls. A non RT task, that do not cooperates with other tasks, can always be interrupted with no further consequences; unfortunately, a RT task needs a special OS support to ensure a response before a deadline, like at least have a deterministic prediction of the interrupt timing. Even more, in cooperative non RT tasks, an interrupt is not
always welcome, because some concurrent operations need to disable the inter-
rupts to ensure data integrity. This problem is widely called the Critical Section
Problem, and most OS APIs (including POSIX), offers semaphores mechanisms
to solve this problem. Disabling the interrupts is a very harmful operation
for real-time tasks, so special safe-semaphores also should be supported by the
RTOS.

Modern OSs are also multi-tasking systems, having a task scheduler as the
main part of the OS. In general terms, this means that at some point a task is
forced to leave the CPU, but the registers, program counter, stack, and other
task variables are saved to restore the task to the CPU in the future. The process
of changing a task for another, saving and restoring the tasks data, is called
Context Switching. As every single service on a RTOS, the context switching also
should be a deterministic process, to successfully predict the system overhead
on changing tasks.

2.2 Task Scheduling

In General-Purpose OSs, usually task scheduling is focused on fairness, to
give the perception that all the tasks are executing simultaneously on the CPU.
Real-Time Systems needs a completely different approach, focused on task pri-
ority, low overhead and full preemptiveness. The first RT scheduling research
publication, was the rate-monotonic theory: a deterministic preemptive algo-
rithm family, based on static service priorities (highest request frequency). The
main problem of rate-monotonic algorithms and other static approaches, is the
resource utilization limit, or so called scheduling bound. This bound may be
much less than 100%, such 70% for rate-monotonic algorithms, in real condi-
tions.

[More on Task Scheduling]

3 Working with Real-Time Systems

3.1 Why We Need RTOS: A Practical View

Imaging you are controlling a device with real-time requirements. A common
requirement is to generate a periodic call of \(X [Hz]\) with a maximum error of
\(e(X) = \frac{1}{X} \cdot 10^{-2}[s]\). If the device has hard real-time requirements, the maximum
error \(e(X)\) is applicable to every call, meaning that if a single call have a greater
error than \(e(X)\), then the system is not suitable for hard real-time. If it’s a soft
requirement, then we can use the standard deviation as an “on average” statistic
that must be lesser than the maximum error. If our timing precision is over a
second (\(X = 1[Hz]\)), almost any general-purpose OS is capable to do this task,
but as we increase \(X\) we will need more specific RTOS, and maybe a suitable
machine to support that requirements. As an example, Figure \ref{fig:RTOS}
presents a C
program that setups a periodic call using a POSIX timers and prints the real
time difference between each timer call.
int count;
struct timeval new,last;

void call(int foo){
    gettimeofday(&new,NULL);
    /* put into a buffer the difference
    between last and new times*/
    last=new;
    count++;
}

void periodic_call_setup(int max,int interval){
    struct itimerval it;
    gettimeofday(&last,NULL);
    it.it_interval.tv_usec=interval);
    signal(SIGALRM,call);
    setitimer(ITIMER_REAL,&it,NULL);
    while(count<max);
}

Figure 1: POSIX timer example. This code setup a standard POSIX timer to a given interval in microseconds. When the system timer is met, the call() function is called and the current time is saved. The differences are sent to a buffer to check the timer behavior. The gettimeofday(), signal() and setitimer() functions are conforming to POSIX.1-1003b.

Figure 2: 100 samples of 1[Hz] timer differences (∆1_i) on a general-purpose OS and CPU (Linux). $\Delta T \approx 1000012[\mu s]$, $\sigma(\Delta 1) \approx 493[\mu s]$. 
This code was compiled on a common IBM compatible PC (2\text{GHz} processor with 1\text{GB} of RAM) using a standard configured Linux Kernel, the GNU LibC, and the GNU C compiler. This same hardware configuration is the one will be used through this report as the general-purpose CPU; as well, the general-purpose OS will be the same software configuration described above.

If we setup the timer to $X = 1\text{Hz}$ (\texttt{interval=1000000}), the difference values are very near the $1 \leq 10^6 [\mu s]$. Figure 2 shows the behavior of this first case, with no much perturbation between the expected values. As $\text{max}_i(|\Delta_1| - 10^6) = 2420[\mu s] << e(1) = 10^4[\mu s]$, then the hard real-time requirement is fulfilled.

If we do the same experiment, but with $X = 100\text{Hz}$ (\texttt{interval=10000}), we have some huge delays due the context switching as the Figure 3 shows. After each delay, we can see that the next difference is quite smaller than the expected time, because the timer “realize” that is late, so it sleeps less than the given \texttt{interval}. Probably, lesser errors are not caused by the context switching problem, but to the simple fact that the timers are not designed with a deterministic-fashion criteria. As $\text{max}_i(|\Delta_{100}| - 10^4) = 3632[\mu s] > e(100) = 100[\mu s]$, then the hard real-time requirement is not met. As regards to the soft requirement, the $\sigma(\Delta_{100}) \approx 3852[\mu s]$ is also much greater than $e(100) = 100[\mu s]$, so “on average” the requirement is not met either. Figure 4 shows a quick summary of the several tests with different timer periods.

Linux implements the POSIX time-related syscalls, such \texttt{usleep()}, \texttt{nanosleep()}, \texttt{gettimeofday()}, etc. Most of those calls have a theoretical resolution of 1[\mu s], besides \texttt{nanosleep()} that have a 1[ns] resolution. But in the practice, the resolution of the time-related syscalls are very much lesser. In a default Linux kernel configuration, timer values are represented in jiffies, as a minimal unit for any
Figure 4: This table contains the common statistics for 3 independent tests of 4 different timer intervals. This tests were made on a general-purpose CPU and OS, with no special system load rather the desktop environment.

O.S. procedure (i.e. scheduling). A jiffy is set to 0.004[s], so the resolution of the sleeping syscalls are between 5 to 10 milliseconds. This means that the sleeping syscalls cannot ensure a precise sleeping time; indeed nanosleep manual page says “nanosleep() delays the execution of the program for at least the time specified in req.”, meaning that the nanosleep() syscall will always complete the jiffy time before waking the process.

4 Acknowledgments

This work was supported by project ALMA-CONICYT 31060008, and could not been done without the support of ALMA, ESO, NRAO and UTFSM. Mauricio Araya’s work was supported by a UTFSM Iniciación Científica grant.

References


