CLEOPATRE: Open-source Operating System Facilities for Real-time Embedded Applications

Maryline Chetto, Thibault Garcia, Audrey Marchand

Journal of Computing and Information Technology - CIT 15, 2007, 2, 131–142
doi:10.2498/cit.1000806

CLEOPATRE: Open-source Operating System Facilities for Real-time Embedded Applications

Maryline Silly-Chetto, Thibault Garcia-Fernandez and Audrey Marchand

IRCCyN, UMR CNRS 6597 / University of Nantes, France

In this paper, we present the facilities offered by CLEOPATRE, a new framework devoted to respond to real-time application developers needs, more particularly for open-source environments. The optional components of the operating system library cover a large range of needs in terms of scheduling, synchronization, fault-tolerance and aperiodic task servicing. And we describe some results of a performance evaluation study focusing on overhead and footprint.

Keywords: real-time, operating system, open-source, scheduling, Linux, fault-tolerance

1. Introduction

In any intelligent control system, there are three hierarchical levels. The servo level involves reading data from sensors, analyzing the data and controlling electromechanical devices. The timing is critical and often involves periodic tasks ranging from 1 Hz to 1000 Hz. At a higher level, orders like *move mobile robot to position A* are issued but are not as critical as for the servo level. Finally, at the highest level, tasks establish a plan for a pre-defined time horizon. At every level of control, the problem of scheduling operations has to be considered. Generally, scheduling will consist in the allocation of available resources (files, CPU, machines, vehicles...) to operations or parts over time with respect to specified constraints including timing constraints. At the servo-level, real-time constraints imposed on the tasks are specified by strict deadlines by which they need to be fulfilled. Real-time means getting the task done on time and every time. This means that the primary concern is not average response time, but worst-case response time.

So, the major goal in designing a real-time control system is to ensure adherence to all the timing constraints during the whole lifetime of the application in order to avoid a system failure. To meet these timing requirements, coordination, scheduling, resource management and control are functions that must be adequately implemented with a multi-tasking and flexible real-time operating system (RTOS). Computer control systems are embedded in a large and growing group of products such as automotive vehicles, aircrafts, and industrial robots. A reason for designing an easy usable and complete real-time operating system is that the applications are becoming increasingly more complex due to the inclusion of more functionality.

This paper focuses on the description of a real-time library aiming to support the reactive layer for control applications. We present CLEOPATRE, a highly configurable system providing the means for the application designer to choose the needed functionality in a library of open-source software components that provides real-time facilities. CLEOPATRE claims not to be restricted to any application area [1].

The library hides to the designer some of the complexity by providing a simplified interface. Furthermore, the library allows on-line modification of real-time parameters. This feature can be used to adapt the computational requirements of the real-time system at run time.

The software framework was developed as part of the French national R&D project CLEOPATRE. It is an offshoot of a larger project to

develop open-source software dedicated to reconfigurable robots. Section 2 deals with weakness of Linux as an open-source operating system for real-time applications and gives some background materials about RTAI, a real-time variant of Linux. In Section 3, we discuss the requirements of an RTOS to be compliant with a large panel of embedded applications which are proposed by the CLEOPATRE library. These real-time facilities are described in Section 4. Section 5 reports evaluation results in terms of overheads and footprints, and Section 6 describes a simple development example. We summarize the paper in Section 7.

2. Linux and Real-time

2.1. Real-time Concepts

The use of state-of-the-art real-time techniques in planning and control applications is still rare. The lack of competence in real-time theory among engineers and the lack of commercially available tools are the major reasons for this.

Real-time systems can be classified into two major categories: hard and soft real-time systems, depending on the consequences of a deadline miss. The deadline is derived from the latest point in time when a response to an event must be generated. Hard real-time systems are computer systems in which all task deadlines must be met. On the other hand, in soft real-time systems, a number of deadlines can be missed without serious consequences. Real-time systems must be able to handle not only periodic tasks, but also aperiodic tasks. Periodic tasks are used to implement off-line pre-planned requests. While periodic tasks have hard deadlines, aperiodic tasks may have soft, hard or no deadlines at all.

When aperiodic tasks have hard deadlines, the goal of the operating system is to allow the execution of aperiodic tasks without jeopardizing the schedulability of hard periodic tasks. It clearly comes that the scheduler is the central element of the operating system and must be flexible enough to cope with all task models and constraints.

2.2. Linux Features

Linux presence is rapidly increasing in advanced control applications, replacing proprietary real-time operating systems. In fact, Linux is now the dominant operating system within the embedded market. The embedded 32-bit processor market has over twenty times more deployments than desktop computers, and scales a wide range of applications from MP3 players to telecommunication systems.

Linux initially attracts designers because it is free to download, comes with full source code, and is compatible with a wide range of processors. A detailed examination reveals a modular architecture, a user-friendly licensing, a worldwide support community, a reputation for reliability, a standard programming interface, free tools, and a mature, well-tested code base. Although these features are ideal for a large segment of the embedded-system world, the lack of precise timing in response to external events has been the biggest weakness for real-time applications.

The standard Linux kernel has been suitable for providing the best average response time. As Linux is loaded down, it gracefully degrades the performance of all tasks. This is not acceptable for real-time computing. For example, when a process calls a kernel service, such as the scheduler or a device driver, this call disables interrupts and makes it impossible to preempt Linux until the service completes execution. Likewise, the stock Linux scheduler uses a fairness algorithm that guarantees even the lowest priority process some CPU time, even though a higher priority process may be waiting.

To deal with these problems and make the kernel more responsive, the Linux support community has devised patches and workarounds that deliver deterministic performance. One approach has been to add preemption points within the kernel to reduce process latency and still protect critical code sections.

A second approach is to add a small real-time kernel to handle the high-priority tasks while Linux runs as the lowest priority to schedule the remaining, non-real-time tasks. This approach delivers the best real-time performance because it doesn't allow Linux to disable interrupts. RTLinux and RTAI (Real Time Application Interface) are two open-source projects

that their developers based on the dual-kernel approach [2] [3]. The response to this is to write a minimal RTOS, then run Linux as a background task under the minimal RTOS.

3. CLEOPATRE Framework

3.1. Background Material on RTAI

The RTAI project began at the *Dipartimento di Ingegneria Aerospaziale del Politecnico di Milano* (DIAPM) about ten years ago. RTAI is a plug-in which permits Linux to fulfil some real-time constraints (few milliseconds deadline, no event loss). It is based on a Hardware Abstraction Layer (HAL). This concept is also known in Windows NT. The HAL defines a clear interface which exports some Linux data and functions close related to hardware (see Figure 1). RTAI modifies them to get control over the hardware platform. RTAI is basically an interrupt dispatcher which mainly traps the peripherals interrupts and if necessary re-routes them to Linux (e.g. disk interrupts).

That allows RTAI real-time tasks to run concurrently with Linux processes. It leads to simple adaptation in Linux Kernel and easy RTAI port from version to version of Linux.

RTAI offers some services related to hardware management layer dealing with peripherals, scheduling and communication means among tasks and Linux processes.

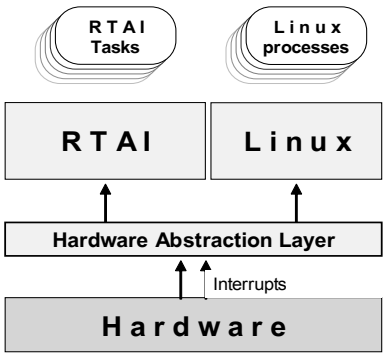

Figure 1. Basic principles of RTAI.

The scheduler of RTAI is in charge of distributing the CPU to different tasks present in the system (including Linux). The scheduler makes it elected the first highest priority task in a ready state. RTAI considers the priority 0 as the highest priority and 0x3fffFfff the lowest, given to Linux.

With RTAI, the Linux Trace Toolkit [4] and the GNU debugger as helper development tools can be used.

Nevertheless, it presents drawbacks which restrict the fields of integration:

1) the preemptive scheduler works with static priorities,
2) both aperiodic and periodic tasks can be used but the priority of a periodic task bears no relation to its period,
3) deadlines are not used,
4) resource access protocols and fault-tolerance mechanisms are not present.

3.2. Motivations

Consequently, our approach in the CLEOPATRE project was first to add real-time capabilities to Linux, yet keeping all its existing capabilities and second, to design an adaptive, reconfigurable and extensible operating system in free open-source code. One of the most important reasons for us to choose Linux as the foundation of our project is due to the open source policy behind it that allows it to grow constantly. One of the key technological innovations in CLEOPATRE is the configuration system. Essentially, this enables CLEOPATRE developers to create their own application-specific operating system and makes CLEOPATRE suitable for a wide range of embedded uses. Configuration also ensures that the resource footprint of CLEOPATRE is minimized as all unnecessary functionalities and features are removed. The configuration system also presents CLEOPATRE as components architecture. This provides a standardized mechanism for component suppliers to extend the functionality of CLEOPATRE.

3.3. Interfacing CLEOPATRE to RTAI

Our purpose was to enrich the real-time facilities of a Linux-based operating system, but we did not want to reinvent the wheels. So we have exploited the idea used by the operating system

community for structuring effective hierarchical operating systems consisting of different layers of operating systems. Such approach provides a flexible environment for sharing hardware resources among multiple operating systems.

In the CLEOPATRE framework, multiple prioritised domains co-exist simultaneously on the same hardware where Linux is the lowest priority domain and CLEOPATRE is the highest priority domain which always processes interrupts before the RTAI domain (see Figure 2). Let us note that RTAI could be replaced by RT-Linux by adapting a single interfacing component. Several projects have pursued the same goal such as the European IST project, OCERA (Open Components for Embedded Real-time Applications), also initiated in 2002, which uses a similar approach and relies on RT-Linux (see `http://www.ocera.org/`).

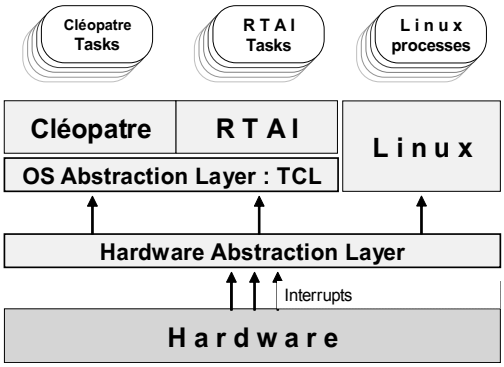

Figure 2. Basic principles of CLEOPATRE.

To interface CLEOPATRE with RTAI, a specific Linux module has been created, namely TCL (Task Control Layer) which is an enhancement of the native RTAI scheduler. TCL provides an *internal interface* for the CLEOPATRE services. It is responsible for low-level mechanisms and data structures including generation of periodic events and lists of task descriptors. For example, *TCLCreateType* is a structure that gathers all the parameters required to create tasks for any RTOS.

The CLEOPATRE library contains this specific module which is necessarily loaded to run a CLEOPATRE application. This module constitutes an OS abstraction layer which makes the library of generic services since it is possible to use them with Linux through different versions of RTAI or other kernel such RTLinux by only adapting this software layer. Consequently, this abstraction model makes it possible to integrate new real-time facilities and development tools, in a clear and efficient way.

Moreover, native RTAI applications still run under the CLEOPATRE environment, with a view to keeping compatibility. RTAI is no longer needed after the introduction of Cleopatre tasks. But it was decided to keep RTAI as a background task which makes available a minimal set of functionalities for users. Cleopatre tasks and RTAI tasks may coexist whereas never interfering except for interrupts handlers which necessarily will require attention. Nevertheless, it is also possible and easy to manually convert RTAI C code to Cleopatre code.

4. Real-time Facilities of CLEOPATRE

4.1. Library of Components

Advanced sensor-based control systems are dynamically reconfigurable. As a result, they require many special features, including highly adaptive schedulers which are currently not found in classical real-time operating systems. Several features the authors have developed and implemented as part of CLEOPATRE are presented here. They are dispatched in multiple shelves of the library and provide selectable components (see Figure 3).

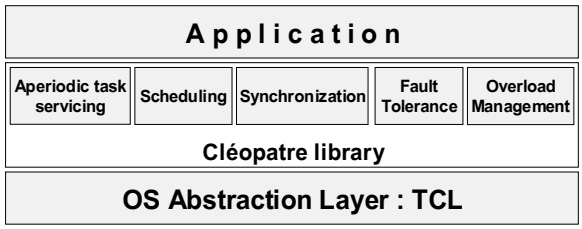

Figure 3. The CLEOPATRE framework.

We define a component as a piece of software that brings some new functionality or feature at different levels in some of the following fields: scheduling, fault tolerance, aperiodic task servicing and overload management. A component is dynamically loadable, and runs in kernel space. It is installed or uninstalled via the well-known *insmod* and *rmmod* Linux commands respectively.

4.1.1. Scheduling

One of the key aspects in real time systems is concerning to the task scheduling policies. A strong work has been done during the last years in static and dynamic scheduling theory. Whereas static scheduling has been incorporated in commercial operating systems due to its simplicity, dynamic scheduling has been relegated to a research framework. However, new advances in scheduling theory and new applications working with dynamic environments emphasize its advantages.

It is well known that with dynamic priority scheduling policies it is possible to achieve higher utilization levels of the system resources than with fixed priority policies. In addition, there are many systems for which their dynamic nature makes it necessary to have very flexible scheduling mechanisms, such as robotics systems, complex control systems and multimedia systems, in which different quality of service measures need to be traded against one another.

CLEOPATRE supports both static and dynamic preemptive scheduling of real-time tasks including Deadline Monotonic (DM) and Earliest Deadline First (EDF) [5] [6]. With DM, tasks with shorter deadlines are given higher priorities, at design time. The DM priority assignment is optimal meaning that if any static priority scheduling algorithm can meet all the deadlines, then the DM algorithm can too. A problem with DM is that is does not support dynamically changing periods.

With EDF, the task with the earliest deadline is always executed first. When scheduling periodic tasks that have deadlines equal to their periods, EDF has a utilization bound of 100%. That is, EDF can guarantee that all deadlines are met provided that the total CPU utilization is not more than 100%. So, compared to fixed priority scheduling techniques like DM scheduling, EDF can guarantee all the deadlines in the system at higher loading. Tasks may be periodic or not and characterized by a deadline less than or equal to the period. Programmers may change from one scheduler to another in loading the corresponding component without involving recompilation.

4.1.2. Synchronization

Synchronization refers here to resource access policy that is a set of rules that govern when and under what conditions each request for resource is granted and how tasks requiring resources are scheduled. Exclusive access to a shared resource is typically achieved with the use of a semaphore.

Problems with semaphore designs are well known: priority inversion and deadlocks. In priority inversion, a high priority task waits because a low priority task has a semaphore. In a deadlock, two tasks lock two semaphores, but in the opposite order. Research has yielded solutions respectively based on non-preemption, priority inheritance and priority ceiling. The classical policies in CLEOPATRE use P and V operations on a semaphore with an associated queue of waiting tasks for the resource and ordered according to some selectable rule such as FIFO or task priority. But to prevent from priority inversion and deadlocks, the following policies have been implemented and made compatible with any fixed or dynamic scheduling strategy:

- The simplest way is to disable all interrupts during critical sections which correspond to execute the task holding a resource at the highest priority. We call this protocol the Super-Priority Protocol (SPP). This scheme is simplistic, but requires clever programming to keep the critical sections very brief.
- The Priority Inheritance Protocol (PIP) mandates that a lower-priority task inherits the priority of any higher-priority task pending on a resource they share [7]. This priority change takes place as soon as the high-priority task is pended; it ends when the resource is released. However, deadlocks may still occur.
- The Priority Ceiling Protocol (PCP) is an extension of PIP to prevent both deadlocks and priority inversions [8]. The scheduling algorithm will increase the priority of a task to the maximum priority of any task waiting for any resource the task has a resource lock on. Once a task finishes with the resource, its priority returns to normal.

CLEOPATRE users may opt for, either protocols that avoid synchronization anomalies and

guarantee predictability with high overheads including PCP, or less complex protocols that may lead to a non-deterministic behaviour, for example using the conventional semaphore-based primitives. Note that PCP and SPP are the only synchronization protocols for which maximum blocking times can be precisely evaluated and inserted in an off-line test to verify feasibility of the application. That means that the choice for a synchronization mechanism must be tuned in accordance with the application profile and requires attention especially for basic protocols that involves unpredictable blocking times. However, as mentioned in Subsection 4.2, Cleopatre provides a timeout mechanism which enables to properly cancel every running task when its execution time exceeds a given time interval.

For the moment, only semaphore-based synchronization mechanisms are supported under CLEOPATRE, but doubtless, the shelf shall be enriched.

4.1.3. Fault-tolerance

The interactions between the computer control system and the instrumentation are more and more complex and the number of functions delegated by the human operator to the computer tool imposes to verify its functioning on the exactness of the results which it produces and on the respect for the temporal constraints attached to each of them.

Even when carefully designed a real-time system can be subject to disturbances which are the effects of subtile errors in software coding, malfunctions in input channel. Moreover, estimating the execution time of tasks is often difficult and under-estimating worst-case execution times can lead to deadline failure.

Consequently, it is necessary to provide any real-time operating system with techniques which are able to make the results available on a timely basis even if this one leads to a functioning in *degraded mode*. This is a quality prerequired by the embedded real-time systems.

Fault-tolerance (FT) is achieved via time and/or spatial redundancy and hence its design methodologies are characterized by the trade-off between these two types of redundancy. In real-time environments, a fault-tolerance policy should be selected and implemented to recover from errors within a certain time limit.

In CLEOPATRE, we consider software redundancy and time redundancy strategies. Software redundancy means that the system will be provided with different software versions of tasks, so that when a version of a task fails under certain inputs, another version can be used. With time redundancy, the task schedule has some slack in it, so that some tasks can be rerun, possibly with less precision, and still meet critical deadlines.

With the so-called Deadline Mechanism, each fault-tolerant task is implemented as two distinct tasks (primary and backup copy) [9]. The primary version produces good quality results, but its execution is prone to a timing failure because of its high level of complexity and resource usage. The backup version, on the other hand, only contains the minimum required functions and produces less precise, but acceptable results in a time bounded interval. Since it is simpler and requires less resources, we assume that its reliability has been fully tested a priori and its WCET (Worst Case Execution Time) can be known. That is why a feasibility test based on WCETs of backup tasks can be performed off-line in order to certify that, in the most degraded mode, all back-up tasks can be feasibly scheduled and the system is never overloaded, even transiently.

Hence, whenever a primary copy tries to execute for an interval of time longer than its reserved execution time, it is aborted and the scheduler is able to guarantee the execution of the backup copy while meeting its timing constraint. The backup-copy provides a degraded performance solution but never executes unnecessarily when the associated primary copy successfully completes, and this by applying the so-called Last chance strategy.

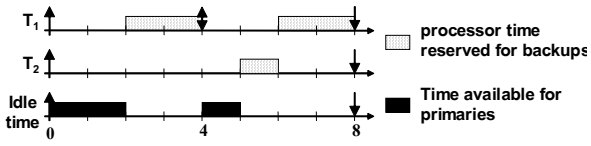

Figure 4. Static EDL schedule for backup copies.

In our implementation, the processor time reserved for the execution of the backup copy is

realized with EDL (Earliest Deadline as Late as possible) algorithm and is reclaimed as soon as the primary task executes successfully (see Figures 4 and 5). EDL is a dynamic slack stealing algorithm which consists of making spare processing time available for the primary tasks as soon as possible by stealing slack from the backup tasks.

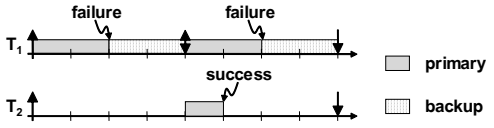

Figure 5. FT schedule with the Deadline Mechanism.

This technique, above all, is proving an efficient one for failure recovery in particular when execution times are under-estimated or when programming errors produce unbounded computation times like infinite loops, leading to deadline missing.

4.1.4. Overload Management

Often, in a so-called dynamic or reconfigurable system, neither the number of tasks to be executed on the processor nor the characteristics of these tasks are known a priori, possibly leading to transient overload. The method of Imprecise Computation (IC), introduced by Lin, Liu and Natarajan, is a flexible technique for the design of such real-time systems [10]. The technique is motivated by the fact that one can often trade off precision for timeliness. It prevents missed deadlines and provides graceful degradation during a transient overload.

A task based on this model consists of two or more logical parts: a mandatory part and at least one optional part. The mandatory part should include all the operations necessary to produce a logically correct result. The optional part, on the other hand, refines the output of the mandatory part within the limits of the available computing capacity. Since the mandatory parts have hard deadlines, provisions should be taken against faults which may occur during execution. That is why Imprecise Computation may be used in conjunction with Fault-tolerance in CLEOPATRE.

4.1.5. Aperiodic Task Servicing

An aperiodic task server aims to schedule soft and hard aperiodic tasks together with periodic tasks. Whenever a hard aperiodic task occurs, an acceptance test is performed in order to verify the feasibility of the resulting schedule. If the aperiodic task is rejected, a message is printed, and the application is informed via error codes so that it can react if necessary. Whenever a soft aperiodic task occurs, it is scheduled so as to minimize its response time. Soft aperiodic tasks are served on a FCFS (First Come First Serve) basis.

Three servers are available in the library: BG (Background server), EDL (Earliest Deadline as Late as possible server) and TBS (Total Bandwidth Server) [11] [12]. EDL is based on the dynamic Slack Stealing approach while TBS is based on the dynamic computation of a virtual deadline for the aperiodic task. Both EDL and TBS have been proved optimal in the sense that they minimize the mean response time for the soft aperiodic tasks and they maximize the acceptance ratio for the hard aperiodic tasks while guaranteeing that periodic tasks still meet their timing requirements.

An additional shelf of components called "Quality of Service" is under development. It aims at providing scheduling solutions for firm real-time applications [13] where missing deadlines under some precise conditions is tolerated. More precisely, it will enrich the CLEOPATRE library with enhanced scheduling components based on the Skip-Over model [14].

4.2. CLEOPATRE Tool Set

The strength of CLEOPATRE lies also in a set of tools which are highly desirable for developing real-time applications, but are not available with the other RTOS. Following is a list of the advanced features implemented as optional components in the CLEOPATRE library:

- A software automatic stop mechanism to safely terminate applications even in emergency situations. This mechanism avoids that programming faults lead to a computer crash and keeps the integrity of the system.
- A watchdog mechanism to protect from infinite loops. At initialization time, the user can

specify a maximal duration during which a task can be executed. If the execution time of a task exceeds this parameter, the real-time application is automatically ended.
- A communication mechanism, namely KLI (Kernel Linux Interface) for data exchanges between user space and kernel space in order to notably use the standard Linux interface.
- A simulation tool set able to generate interrupts. This tool can be used to simulate executions of periodic tasks as well as occurrences of aperiodic events. It can be viewed as a debugging tool for CLEOPATRE components developers.
- And aspect-oriented programming interfaces to help tracing and debugging, with minimal intrusion.

4.3. Properties of the Library

The CLEOPATRE library is organized into six shelves that contain the real-time services. The CLEOPATRE Off-the-Shelf components are optional except the OS abstraction layer (TCL) and the scheduler.

At most one component per shelf can be selected. Since all components of a given shelf have the same programming interface, they are interchangeable. Any component is inter-operable with any other one. The only exception is the Dynamic Priority Ceiling Protocol which necessarily is loaded with the Earliest Deadline scheduler. For example, the user may select the Earliest Deadline scheduler and the Background aperiodic task server to run with the overload management and fault-tolerance mechanisms.

Components are totally independent from the kernel and the hardware. Reusability of the components with another hardware and OS is made possible by just adapting the OS abstraction layer in the TCL component. This component hides the specific features of each platform, so that the run-time components can be implemented in a portable fashion and adapt to the target's processor architecture and board.

CLEOPATRE framework started from two kernels:
- Linux kernel 2.2.14 with the addition of a set of patches to improve the real-time behaviour
- RTAI 1.3

Initial developments of CLEOPATRE relied on the Real-Time Hardware Abstraction Layer (RTHAL). Note that this concept was subject to the RTLinux patent, thus reassessing the open-source features of both RTAI and CLEOPATRE.

Nowadays however, CLEOPATRE framework is running under Linux/RTAI 3.0 and Linux kernel 2.4.22, and is based on the ADEOS (Adaptive Domain Environment Operating System) interrupt low-level mechanisms [15]. The components compliance was tested on the last free RTLinux version too. Results showed the adaptability of the CLEOPATRE components in another environment. Experiments were confined to the validation of the functional aspect of each component under RTLinux.

5. Implementation

5.1. License

CLEOPATRE is royalty and buyout free since distributed under the LGPL license which permits proprietary application code to be linked with CLEOPATRE without being forced to be released under the GPL license.

CLEOPATRE offers a lot of useful documentation (User's Guide, Programmer's guide, and several articles) which are being continually updated as the system develops. Everything needed to use and develop CLEOPATRE can be downloaded from the web site of the CLEOPATRE project:
`http://cleopatre.rts-software.org/`.

5.2. Accessibility

Success of real-time systems comes from both ease of use and performances. Writing code to run with CLEOPATRE is as simple as writing a C language program to run under Linux.

The name of every CLEOPATRE primitive is composed of two words separated by ".". The first word identifies the component which the primitive belongs to and the second one identifies the primitive. For example, *TCL.begin* refers to the primitive, named *begin* in the TCL component.

Our aim was to define an API which is compatible with the largest possible set of components on a given shelf. This API is partly compliant to the POSIX standard. While POSIX is a very recognized standard API, it does not permit to implement all scheduling and fault-tolerant strategies. So, partial compliance with POSIX and its real-time extensions is the price to pay for permitting both total interoperability between components and extensibility of the library.

5.3. Overheads and Footprints

5.3.1. Performance Evaluation

To get an idea of the performance of CLEOPATRE, several critical operating system features were timed. The timings shown hereafter were performed on a 1,7 GHz Pentium 4 with the Time Stamp Counter (TSC), available with every modern Intel processor. The performance has not been yet evaluated on other targets.

A task can be created in 1.7μs, and destroyed in 0.36μs. These operations are generally done during initialization and termination, and not during time critical code. As a result, performance of these routines is sometimes sacrifised, in favour of performing any computations beforehand, so that the performance of run-time operations, such as context switching, can be improved.

One of the important performance criteria of a multitasking kernel is the reschedule and context switch time. Upon the expiration of a time quantum, a timer interrupts is generated, forcing a time-driven reschedule. If the currently running task has the highest priority, it remains running, and the timer interrupt results in no task swapping. The scheduling time in this case is 96 ns, resulting in a peak CPU utilization of 99.84 percent with a one millisecond time quantum.

At the other extreme, a full context switch is needed, which executes in 120 ns (note that this measurement does not include the storing and restoring of FP registers), thus providing minimum CPU utilization of 98.8 percent. The scheduling time includes updating dynamic priorities, and selecting a new task if the highest priority task changes. A full context switch involves suspending the current task by saving its entire context, and resuming the new task by restoring its entire context.

5.3.2. Footprints

CLEOPATRE kernel is effective and consumes a minimal memory footprint because allowing to load dynamically only the required components.

Table 1 shows that a CLEOPATRE component, when loaded in memory (by the *insmod* command) has a size approximately 33% lower than that on the hard disk. This phenomenon is mainly due to the information which is contained on the disk and used by Linux to dynamically assemble the interfaces of the components at the loading stage.

Mechanism	Component	Hard disk size (KB)	Memory size (KB)
Minimal RTAI	RTAI	27,2	23
Interfacing	TCL	34,8	27,1
Central scheduling	DDM DEDF	3,1 3,1	2,1 2,1
Aperiodic task servicing	BGS EDL TBS	4,2 19,7 3,4	2,9 15,2 2,6
Synchronization	FIFO PRIO SPP PCP DPCP	1,8 1,8 1,5 3,6 4,3	0,8 1,0 0,6 2,3 2,9
Fault tolerance	FT	4,9	20
Overload management	IC	2,5	1,3

Table 1. Footprints of CLEOPATRE components.

We also show that the minimal footprint of the CLEOPATRE system is 30 KB in memory (38 KB in hard disk), excluding RTAI and necessarily including the interface and the central scheduler. CLEOPATRE can then be easily stored on a floppy disk or a compact flash memory.

6. Integration

The following example brings to light some of the features of CLEOPATRE and shows the different steps in developing under the CLEOPATRE environment. The application, which is

only an academic one for simplicity, aims to count the number of interruptions which are generated by the mouse.

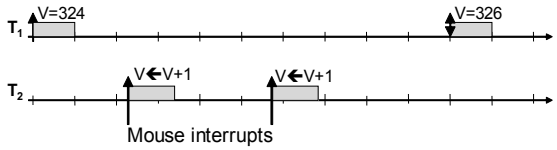

Figure 6. General view of the example.

In this example, an interrupt handler is tied to the mouse signal and releases aperiodic task T_2. T_2 keeps up to date variable V that contains the number of interrupt occurrences. Periodic task T_1 reads V and displays it every second. Access to V is protected by a mutual exclusion semaphore.

6.1. Source Code

To compile the application, a programmer needs to include headers of all required components.

```
/* —— Components interfaces —— */
#include <TCL.h> /* Task Control Layer */
#include <Dsch.h> /* Scheduler */
#include <sem.h> /* Semaphore */
#include <irq.h> /* Interrupt management */
```

Parameters are declared with #define directives.

```
/* —— Parameters —— */
#define TIMERTICKS 1e6 /* Timer period: 1ms */
#define irq_twelve 12 /* Mouse interrupt */
#define HEAP 0 /* Heap size */
#define STACK 2000 /* Stack size */
#define NO_FPU 1 /* 0 to use FPU */
```

Descriptors for tasks, semaphores and interrupts are declared as global static variables.

```
/* —— Descriptors —— */
static DSchTaskType t_mouse; /* Tasks */
static DSchTaskType t_report;
static irqType irq_mouse; /* Interrupt */
static SemType sem; /* Semaphore */
/* —— variables —— */
static unsigned v; /* number of mouse interrupts */
static unsigned i=0; /* number of printings */
```

Infinite loop and explicit *wait next period* primitive are not required to implement a periodic task in the CLEOPATRE environment, in contrast of most RTOSes, which results in more conciseness.

```
/* —— Periodic report task —— */
void report() {
sem.P(&sem); /* semaphore locking */
print("report %4i: %u\n",++i,v);
sem.V(&sem); /* semaphore unlocking */
}
```

Similarly, a semaphore is not required to release an aperiodic task.

```
/* —— Aperiodic count task —— */
void mouse() {
sem.P(&sem);
v++;
sem.V(&sem);
}
```

CLEOPATRE real-time handlers begin with the macro-command *IRQ_begin* and terminate with *IRQ_end*. These macro-commands protect handlers from multiple interrupts during their execution and give hand to Linux handlers if needed.

The following handler just releases the aperiodic task.

```
/* —— Interrupt Handler —— */
void handler() {
IRQ_begin(&irq_mouse);
Dsch.wakeup(&t_mouse,TCL.time);
IRQ_end(&irq_mouse);
}
```

Linux runs the *init_module()* function that initializes tasks, semaphores, interrupt handlers and starts running in real-time mode.

```
/* —— Application initialization —— */
int init_module(void) {
/* O.S. Abstraction Layer parameters */
TCLCreateType creat = {HEAP,STACK, NO_FPU,0};
/* Initializations: Tasks, semaphore, interrupt */
Dsch.create(&t_report,report,1000,1000,creat);
Dsch.create(&t_mouse, mouse, 0, 0,creat);
sem.create(&sem,1);
IRQ.create(&irq_mouse, irq_twelve,handler);
/* periodic task start */
Dsch.wakeup(&t_report,1000);
/* Real time mode */
TCL.begin(TIMERTICKS,20);
return 0;
}
```

Cleanup_module() function is executed when unloading. *TCL.end()* function safely deletes every object initially declared by *init_module()*.

```
/* —— Application deletion —— */
void cleanup_module(void) {
TCL.end();
}
```

6.2. Results

This application enables us to state that, on average the mouse generates 300 interrupts per second. But, this result is not as important as the way to get it. The main objective of this basic example was to give an overview about programming under CLEOPATRE.

7. Conclusion

The word "real-time" is defined as follows: the ability of the operating system to provide a required level of service in a bounded response time. The real-time functions include not only guarantee of worst case but also being lightweight, small footprint, high speed and so on.

CLEOPATRE is a portable, open-source, free to download and royalty free RTOS that can be used in commercial applications through the LPGL license. It can be viewed as an "a la carte" kernel that provides system software components, allowing for a wide variety of services. Developers can reduce their time to market by using CLEOPATRE as a debugging tool during development as well as using its services in the final product deployment. CLEOPATRE applications are highly portable to any new CPU architecture thanks to its OS abstraction layer which makes the library of services generic.

We have proved the applicability and interoperability of the components, first by simulation tests and then by integration. CLEOPATRE has been used with several different systems, both at Nantes University in a mobile robotic application and elsewhere, including at the LRV Laboratory (*Laboratoire de Robotique de Versailles*) to realize a real-time vision system and to control a redundant manipulator with 7 degrees of freedom [16].

Current works aim at completing the functionalities of the library with scheduling facilities that consider Quality of Service requirements.

8. Acknowledgment

The CLEOPATRE project was supported by the French Research Office, grant number 01 K 0742. We would like to thank all partners of this project.

References

[1] T. GARCIA, A. MARCHAND, M. SILLY-CHETTO, CLEOPATRE: A R&D project for providing new real-time functionalities to RTAI Linux. *5th Real Time Linux Workshop* (2003), Valence, Spain.

[2] V. YODAIKEN, The RTLinux Approach to Real-Time. *FSMLabs Inc.*, (August 2004).

[3] P. MANTEGAZZA, DIAPM RTAI for Linux: Why's, what's and how's. *Real Time Linux Workshop*, (1999), University de Technology of Vienna.

[4] B. B. RAMYA, V. PAVITHRA, B. THANGARAJU, Process Tracing with the Linux Trace Toolkit. *Sys Admin magazine*, (2004).

[5] N. C. AUDSLEY, Deadline monotonic scheduling. *Report YCS.146*, (1990), Dept. of Computer Science, University of York.

[6] C. LIU, J. W. LAYLAND, Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of ACM*, (1973), 20(1) pp. 46–61.

[7] L. SHA, R. RAJKUMAR, J. P. LEHOCZKY, Priority inheritance protocol: An approach to real-time synchronization. *IEEE Transactions on Computers*, (1990), 39(9), pp. 1175–1185.

[8] M. I. CHEN, K. J. LIN, Dynamic priority ceilings: a concurrency control protocol for real-time systems. *Real-time systems journal*, (1990), 2(4), pp. 325–346.

[9] H. CHETTO, M. SILLY-CHETTO, An adaptive scheduling algorithm for a fault-tolerant real time system. *Software Engineering Journal*, (May 1991), 6(3), pp. 93–100.

[10] J. W. S. LIU, J. K. LIN, S. NATARAJAN, Scheduling algorithms for multiprogramming in a hard real-time environment. *Proceeding of the 8th real-time system symposium*, (December 1987), pp. 252–260, San Francisco, CA.

[11] M. SILLY-CHETTO, The EDL server for scheduling periodic and soft aperiodic tasks resource constraints. *The journal of real-time systems*, Kluwer academic publishers, 17 (1999), pp. 1–25.

[12] G. C. BUTTAZZO, F. SENSINI, Optimal Deadline Assignment for Scheduling Soft Aperiodic Tasks in Hard Real-Time Environments. *IEEE Transactions on Computers*, (1999), 48(10), pp. 1035–1052, ISSN:0018-9340.

[13] A. MARCHAND, M. SILLY-CHETTO, RLP: Enhanced QoS Support for Real-Time Applications. *In Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications* (August 2005), Hong-Kong.

[14] G. KOREN, D. SHASHA, Skip-Over algorithms and complexity for overloaded systems that allow skips. *In Proc. of the 16th IEEE RTSS'95*, (1995) Pisa, Italy.

[15] K. YAGHMOUR, Adaptive Domain Environment for Operating Systems. *Opersys Inc.*, (2001).

[16] A. DE CABROL ET AL. , Video rate color region segmentation for mobile robotic applications. *Proc. of SPIE, Applications of Digital Image Processing XXVIII*, (2005), Vol. 5909.

Received: February, 2006
Revised: June, 2006
Accepted: July, 2006

Contact addresses:
Maryline Silly-Chetto
IRCCyN, UMR CNRS 6597
Research Group on Real-time systems
1, Rue de la Noé – Nantes, France
e-mail: maryline.chetto@univ-nantes.fr

Thibault Garcia-Fernandez
IRCCyN, UMR CNRS 6597
Research Group on Real-time systems
1, Rue de la Noé – Nantes, France
e-mail: thibault.garcia-fernandez@univ-nantes.fr

Audrey Marchand
IRCCyN, UMR CNRS 6597
Research Group on Real time systems
1, Rue de la Noé – Nantes, France
e-mail: audrey.marchand@univ-nantes.fr

MARYLINE SILLY-CHETTO is a Professor at the University of Nantes, France. Her research is conducted in the Group of Real-time systems of the Research Institute of Communications and Cybernetics (IRRCyN). Her area of interest includes scheduling and fault-tolerance in real-time systems. She has published more than 60 journal articles and conference papers in the area of real-time operating systems. She has been the leader of a French national R&D project, namely CLEOPATRE, supported by the French government, which aims to provide free open source real-time solutions.

THIBAULT GARCIA-FERNANDEZ is an Assistant Professor at the University of Nantes, France. His research field is real-time operating systems. In 2005, he received the PhD degree in Computer Sciences from the University of Nantes.

AUDREY MARCHAND graduated in Computer Engineering from the Ecole polytechnique of the University of Nantes (France), in 2002. She is currently a PhD student at the University of Nantes. Her research interests include scheduling theory and quality of service for real-time systems, and Linux-based real-time operating systems.