Enhancing Linux/RTAI with Open Source Software Components

Thibault Garcia and Maryline Silly-Chetto
LINA (Laboratoire d'Informatique de Nantes Atlantique)
IUT de Nantes / Dept OGP
La Chantrerie - Rue Christian Pauc BP 50609 - 44306 Nantes cedex 03
France
{garcia,chetto}@iut-nantes.univ-nantes.fr

Abstract

Cleopatre is an open source and flexible real-time operating system, that supports many components, features and optional dynamics aspects. The Cleopatre components are classified in four groups: the scheduling components (including static and dynamic schedulers), the synchronization components (including priority inheritance and ceiling protocols), the aperiodic tasks servicing components (including Earliest Deadline Late and Total Bandwidth Servers) and the fault tolerance components (with Deadline Mechanism and Imprecise Computation).

1 Introduction

General-purpose operating systems are not designed to meet the timing constraints of real-time processes. Since its birth, Linux has undergone extensive growth that has made it one of the most robust and efficient operating systems and besides a very good candidate for a potential RTOS.

Recently, we have been working on a RTOS project, namely Cleopatre[1] (Open components for real-time applications) based on Linux[2]. 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. Several projects have pursued the same goal such as RT-Linux and RTAI which both implement a small real-time kernel underneath but outside of the Linux kernel. Our approach is to add new real-time capabilities to Linux, yet keep all existing Linux and RTAI capabilities. Cleopatre can be viewed as a library of components that provides selectable real-time facilities and in addition a specific module which makes it possible to use the components with Linux through RTAI.

The Cleopatre components are contained in dynamic modules of Linux and provide an external interface for the applications. These components have been tested in the integration phase of the project and the demonstrator has been a mobile robotic platform. To facilitate the integration of the application software or additional components, we have developed several features from the last year[1], which will be detailed in the paper.

Cleopatre proposes four categories of components respectively dedicated to scheduling, synchronization, fault tolerance and aperiodic servicing. Only the first one is necessary. The other ones are optional and selectable by the application user. The main problem encountered during the implementation of these components was their separation in different Linux dynamic modules.

Stand alone RTAI is not able to support Cleopatre components. To make that possible, RTAI must be patched with software called Task Control Layer (TCL). TCL provides an "internal interface" for the components, low level mechanisms and data structures such as the followings:

- Creation and destruction of tasks adapted to Linux/RTAI,
- Switching to real time mode and getting back from it,
- Lists of task descriptors,

[1] work supported by the French research office, grant number 01 K 0742
[2] http://www.cleopatre-project.org

- Watchdog for every task,
- A system clock for generating periodic events.

Native RTAI applications can still run under Cleopatre environment; nevertheless, tasks that have been created with the native RTAI interface cannot use Cleopatre primitives. And tasks issued from the Cleopatre interface cannot use the native RTAI primitives.

RTAI schedules Linux as a background task. As a result Linux processes are scheduled whenever there is no activity from RTAI. And TCL allows RTAI to run its tasks whenever there is no activity from Cleopatre.

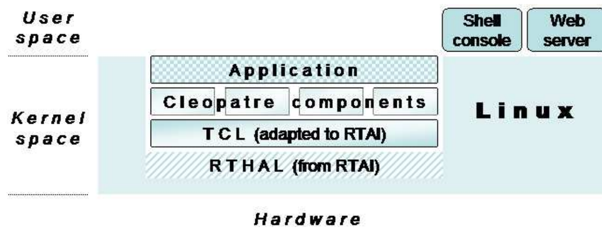

FIGURE 1: *Cleopatre architecture*

TCL is the abstraction layer that permits to make the components as generic as possible. Using components with other versions of RTAI or other RTOSes is made possible by only modifying TCL software. TCLCreateType is a Cleopatre structure that gathers all the parameters needed to create tasks for any RTOS. Usually, the components do not use these data, but they can transmit them to TCL without loosing genericity.

2 Library of components

2.1 Scheduling

Static and dynamic priority driven schedulers, including Deadline Monotonic[2] and Earliest Deadline First[3] are available in the Cleopatre library. Tasks may be periodic or not and characterized by a critical delay less than or equal to the period. Programmers may change from one scheduler to another in loading the corresponding component without involving recompilation of their application.

2.2 Semaphores

Three components in this category are easy to implement: SPP (Super Priority Protocol) that gives a super priority to the task that locks a semaphore, FIFO which releases the task that wait for a semaphore the longest, and "priority" which releases the blocking task with the highest priority. Moreover, the library contains a Priority Inheritance Protocol, namely PIP, usable with a static or a dynamic priority scheduler.

Finally, to avoid both deadlocks and priority inversions, Cleopatre provides priority ceiling protocols: PCP[4] (Priority Ceiling Protocol) and DPCP[5] (Dynamic Priority Ceiling Protocol) respectively designed for a static and a dynamic priority driven scheduler. Both need to know the list of semaphores possibly locked by the tasks.

2.3 Fault tolerance

Two fault tolerance mechanisms have been implemented in Cleopatre: The Imprecise Computation and the Deadline Mechanism.

The imprecise-computation technique is a way to deal with transient overloads. 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.

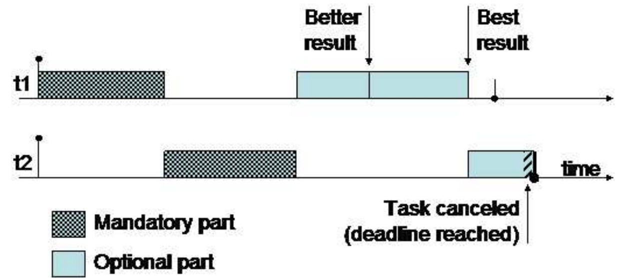

FIGURE 2: *Imprecise Computation*

The mandatory part should include all the operations necessary to produce a logically correct result. The optional part, on the other hand, includes all the other operations. In other words, operations included in the optional part only affect the quality of result. In order to ensure this, there is a rigid precedence constraint between these two parts: the mandatory part must always complete before the corresponding optional part starts its execution. In our implementation, a task may have more than one optional part.

With the Deadline Mechanism[6], each fault-tolerant task is implemented as two distinct tasks (primary and backup copy). Hence, whenever a task tries to execute for an interval of time longer than its reserved execution time, it is suspended and the scheduler is able to guarantee the execution of the backup copy which never executes unnecessarily, using the so-called Last Chance strategy.

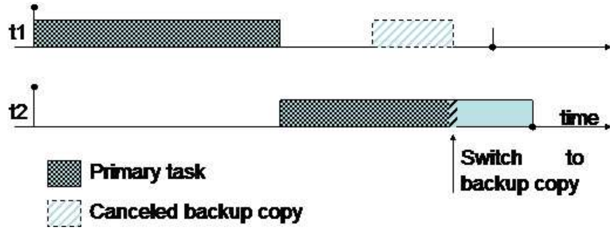

FIGURE 3: *The Deadline Mechanism*

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. This technique, above all, permits to recover failures coming from an under-estimated evaluation of the execution time required by primary tasks or programming errors which produce unbounded computation times like infinity loops.

2.4 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. 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[7][8] (Earliest Deadline as Late as possible server) and TBS[9] (Total Bandwidth Server). 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 that 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.

3 Cleopatre features

Implementation of additional components and application software has been made easy with the following facilities: A software automatic stop which permits to safely terminate applications even in emergency situations, a communication mechanism between user space and kernel space, and aspect oriented programming.

3.1 Software automatic stop

This mechanism registers tasks descriptors, semaphores descriptors, interrupt handlers and user safe ending functions. When the primitive *TCL.end()* is invoked, the context switches to a safe environment in which every descriptor is destroyed, interrupt handlers are unlinked and user safe ending functions are called.

FIGURE 4: *A safe environment(RT1) to destroy real time tasks - LTT capture*

For developers, this mechanism is more reliable than manually destroying descriptors and unlinking handlers, since it avoids to forget executing one of these primitives that often lead to a computer crash. "TCL.end" can be used in case of emergency stop, as the watchdog does when it detects an infinite loop.

3.2 Communication between user and kernel spaces

Applications and components are contained in modules which are loaded in the kernel space of Linux. These modules cannot use standard C libraries and Linux drivers. While RTAI proposes unidirectional FIFOs to establish communications between user space and kernel space, bidirectional FIFOs buffers are proposed in Cleopatre with KLI module (Kernel Linux Interface).

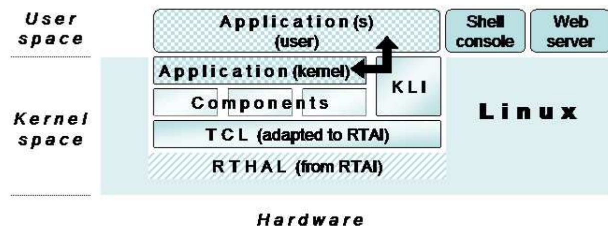

FIGURE 5: *The Kernel Linux Interface*

KLI offers an interface similar to a Linux driver (*open/close/read/write* prefixed by *Cleo_buf*). Each buffer is identified by a string. LXRT (LinuX Real-Time) allows Linux processes present in user space to access to RTAI real-time primitives. LXRT is still usable with Cleopatre environment, but only with RTAI interface.

3.3 Aspect Oriented Programming

AOP[10] (Aspect Oriented Programming) is a recent software engineering technique that facilitates maintenance of large programs. AOP is able to add, modify or even remove some functionalities from a program, following rules.

For example, a security aspect can be weaved to a web server application to encode every messages send, even if the sending primitives are dispatched in most of its modules.

Technically, in our case, every component contains a data structure that gathers the addresses of its primitives. To call a primitive, an application or another component refers to these structures.

An aspect is a Linux module. When this module is weaved to a component, the structure addresses are modified to refer to substitute primitives from the aspect module itself. Most often, these substitute primitives call the primitive they replace, after adding new functionalities as logging or measuring time.

The Cleopatre system contains logging aspects that can be weaved to trace both tasks executions and calls to Cleopatre real-time primitives. Other utilisations have been also considered.

4 Example

The application described in this part aims to count the number interruptions generated by the mouse. This example gathers the fourth main features useful to implement a real-time application: periodic tasks, aperiodic tasks, interruptions and semaphores.

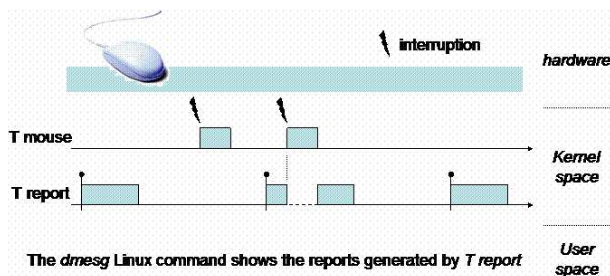

FIGURE 6: *Example - general view*

An interruption handler is linked to the mouse signal to release an aperiodic task. This aperiodic task keeps up to date a variable that contains the number of interruptions. A periodic task read this variable every second to report it. The variable is protected by a mutual exclusion semaphore.

4.1 Source code

To compile the application, a programmer needs to include every component headers.

```
/* ----- 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* directive. The *stack* and the *heap* are memories to store local variables and for dynamic memory allocation. The *FPU* (Floating Point/Processor Unit) is the arithmetic processor that the computer uses to perform arithmetic operations with floating point data.

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

Task, semaphore and interruption descriptors are declared as global and *static* variables.

```
/* ----- Descriptors ----- */
static DSchTaskType t_mouse; /* Tasks */
static DSchTaskType t_report;
static irqType irq_mouse;  /* Interrupt */
static SemType sem_nb;     /* Semaphore */
```

A global variable must be protected by a semaphore if it is used by several tasks (for example, *nb* is protected by *sem_nb*)

```
/* ----- variables ----- */
static unsigned nb;   /* Interrupt's number */
static unsigned i=0;    /* report number */
```

Infinite loop and explicit "wait next period" primitive aren't needed to implement periodic tasks with the Cleopatre system, because they are directly present in the scheduler itself. As a result, the program is easier to read and understand, but it spends a little time for the missing primitives (about 10ns with a 1.7GHz Pentium). To get this time back, a programmer has to add an infinite loop around the code of the task and a *Dsch.wait()* primitive just before the end of the loop.

```
/* ----- Periodic report task ----- */
void report() {
 sem.P(&sem_nb);   /* variable nb protected */
 print("report %4i: %u\n",++i,nb);
 sem.V(&sem_nb);
}
```

Infinite loop and semaphore aren't required to program aperiodic tasks too. To save time, a programmer can add an infinite loop and the *Dsch.wait()* primitive just before the end of the loop as in periodic tasks. The *Dsch.wakeup()* primitive releases aperiodic tasks without semaphore to avoid spending time in managing semaphore descriptors.

```
/* ----- Aperiodic count task ----- */
void mouse() {
 sem.P(&sem_nb);   /* variable nb protected */
 nb++;
 sem.V(&sem_nb);
}
```

Cleopatre real-time handlers begin with the macro-command *IRQ_begin* and end with *IRQ_end*. These macro-commands protect handlers from new occurrences of their interruptions during executions 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 when the application is loaded into the kernel by the *insmod* command. This function creates tasks, initializes semaphores, attaches handlers to interruptions, switches the system to real-time mode, sets the watchdog timer and releases periodic tasks for their first execution.

TCLCreateType gathers all the parameters needed to create tasks adapted to the Linux/RTAI environment.

```
/* ----- Application initialization ----- */
int init_module(void) {
 /* Linux/RTAI specific parameters */
 TCLCreateType creat = {HEAP,STACK, NO_FPU,0};

 /* Creations: Tasks, semaphore, interrupt */
 Dsch.create(&t_report,report,1000,1000,creat);
 Dsch.create(&t_mouse, mouse,    0,   0,creat);
 sem.create(&sem_nb,1);
 IRQ.create(&irq_mouse,12,handler);

 /* Run periodic task */
 Dsch.wakeup(&t_report,1000);

 TCL.begin(TIMERTICKS,20);   /* Real time */

 return 0;
}
```

Linux runs the *cleanup_module()* function when the application is removed from the kernel by the *rmmod* command.

The *TCL.end()* function deletes every descriptor safely. The interrupt descriptor is used here to detach the handlers from their interruption.

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

Every primitive name is composed by two words separated by ".". The first word identifies the component to which the primitive belongs and the second word identifies the primitive itself. For example, *TCL.begin* is the *TCL* component *begin* primitive.

Technically, *TCL* is a C structure that gathers the addresses of every *TCL* component primitive. These addresses may be modified by aspects during their weaving to add new functionalities before and/or after *TCL* primitives.

4.2 Results

The application produces two kinds of results: Text result and graphical result throw LTT (Linux Trace Toolkit). LTT shows the execution of every task and system calls thanks to a graphical interface.

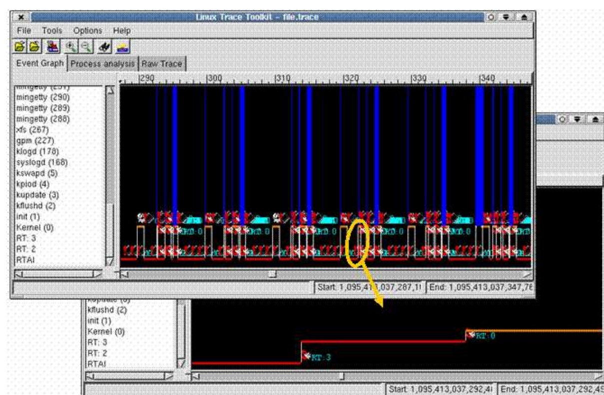

FIGURE 7: *LTT - Aperiodic task execution*

The periodic task prints text reports. These results can be viewed with the *dmesg* Linux command.

```
Terminal - Terminal
Fichier Sessions Options Aide

***** RTAI UNMOUNTED (MOUNT COUNT 0) ******

RTAI-TRACE: Trace facility removed
RTAI-TRACE: Initilialization complete

***** STARTING THE REAL TIME SCHEDULER WITH NO LINUX *****
***** FP SUPPORT AND READY FOR A PERIODIC TIMER *****
***<> LINUX TICK AT 100 (HZ) <>***
***<> CALIBRATED CPU FREQUENCY 1715298000 (HZ) <>***
***<> CALIBRATED 8254-TIMER-INTERRUPT-TO-SCHEDULER LATENCY 7999 (ns) <>***
***<> CALIBRATED ONE SHOT SETUP TIME 3000 (ns) <>***

----- Cleopatre -----
Beginning (0 = Ok) : 0
***** RTAI NEWLY MOUNTED (MOUNT COUNT 1) ******

Tracer: Initialization complete
Tracer: Buffer Size Order is 9
rapport    1: 30
rapport    2: 333
rapport    3: 634
rapport    4: 936
----- Cleopatre -----
Ending  (0 = Ok) : 0
***** THE REAL TIME SCHEDULER HAS BEEN REMOVED *****

***** RTAI UNMOUNTED (MOUNT COUNT 0) ******

RTAI-TRACE: Trace facility removed
[root@localhost ex]#
[root@localhost ex]# dmesg
Nouveau  Terminal n°1
```

report 1: 30
report 2: 333
report 3: 634
report 4: 936

FIGURE 8: *dmesg - The periodic task reports*

This application shows that a mouse is able to generate about 300 interruptions per second. But, this result is not as important as the way to get it.

5 Conclusion

We have implemented a modular and flexible RTOS based on Linux/RTAI in the framework of a national R&D project, Cleopatre, which will end in June 2005. It is public, freely downloading (http://www.cleopatre-project.org/) and protected by the GNU/LGPL licence.

Components have been designed so as to be independent from any RTOS and compatible with a RTOS by adapting a software module called TCL. Cleopatre components have been tested on a mobile robotic platform, an Automated Guided Vehicle that executes orders which are transmitted through wireless communications.

FIGURE 9: *Cleopatre system demonstration mobile robotic platform*

In the future, we aim first to adapt LXRT to Cleopatre and second, to develop Cleopatre components for multiprocessor architectures.

References

[1] T. Garcia, A. Marchand, M. Silly-Chetto, 2003, *CLEOPATRE: A R&D Project for Providing New Real-Time Functionalities to Linux/RTAI*, Proceedings of the Fifth Real-Time Linux Workshop

[2] J.Y.T. Leung, M.L. Merril, 1980, *A note on preemptive scheduling of periodic real-time tasks*, Information processing letters, vol. 20 n3 pp. 115-118.

[3] C. Liu, J.W. Layland, 1973, *Scheduling algorithms for multiprogramming in a hard real-time environment*, Journal of ACM, vol.20

[4] R. Rajkumar, 1991, *Synchronization in real-time systems : a priority inheritance approach*, Boston : Kluwer Academic Publishers, ISBN: 0792392116.

[5] M.I. Chen, K.J. Lin, 1990, *Dynamic Priority Ceilings: A Concurrency Control Protocols for Real-Time Systems*, Real-Time Systems Journal, 2(4), pp. 325-346.

[6] A.L. Liestman, R.H. Campbell, 1986, *A fault tolerant scheduling problem*, IEEE Trans. on software engineering, vol.12 pp.1089-1095.

[7] H. Chetto, M. Chetto-Silly, 1989, *Some results of earliest deadline scheduling algorithm*, IEEE Trans. on SW Eng., 18(8), pp.736-748.

[8] J.P. Lehoczky, L. Sacha, Y. Ding, 1992, *An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority pre-emptive systems*, Proceedings of the IEEE Real-time Systems Symposium, pp.110-123.

[9] M. Caccamo, G. Lipari, G. Buttazzo, December 1999, *Sharing Resources among Periodic and Aperiodic Tasks with Dynamic Deadlines*, Proceedings of the IEEE Real-Time Systems Symposium, Phoenix, Arizona. n1 pp.46-61.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W.G. Griswold, 2001, *An Overview of AspectJ*, European Conference for Object-Oriented Programming.