M2 CORO-ERTS project 2021-2022 : EDF validation & EDH implementation over Xenomai

Advisor: Audrey Queudet

Debus Alexy, Rayella Niranjan

Support

if you encounter any problem, please reach us by mail

alexy.Debus@eleves.ec-nantes.fr,

Table of Contents

1
SECTION

Project organization

1.1 TODO

2

SECTION

Validation for EDF policy

In this section, we will descibe the entire process to validate the EDF policy developped over Xenomai (refer to the github repository for EDF policy source code

2.1 Cobalt core installation with dynamic priority policy

The first step was to compilate/install the cobalt core (I mean the dual-core configuration) for Xenomai 3.1.

In order to do this, you can follow the installation manual available here, respecting few modifications listed below to ensure a proper installation :

If you want to have Xenomai with EDF policy only :

- Replace the github link: https://github.com/BraveMole/Xenomai-EDF.git with this one : https://github.com/skyultime/Xenomai-EDH.git Be sure to move on branch EDFpolicyV1.0 with the following command : **git checkout EDFpolicyV1.0**
- To configure the Kernel (cmd **make menu-config**), load directly the .config file available here (the ipipe patch is also available in the same location if needed)

If you want to have Xenomai with EDF and EDH policy :

- Replace the github link : https://github.com/BraveMole/Xenomai-EDF.git with this one : https://github.com/skyultime/Xenomai-EDH.git. Be sure to move on branch final_version with the following command : **git checkout final_version**
- To configure the Kernel (cmd **make menu-config**), load directly the .config file available here (the ipipe patch is also available in the same location if needed)

In case you want to start from scratch and integrate EDF or EDH policy in default Xenomai 3.1 git repository, you can use the default repository : https://github.com/openil/xenomai-3.git and add patch available here (1 patch for EDF and 1 additional patch for EDH)

2.2 EDF Source code modification

In order to view details of implementation for EDF policy integration over Xenomai 3.1, you can have a look to the EDF.patch file available here
Here is the list of modified files:

Major modifications :

- kernel/cobalt/sched-dyna.c
- kernel/cobalt/sched.c
- kernel/cobalt/posix/thread.c
- kernel/cobalt/thread.c
- lib/alchemy/task.c
- include/cobalt/kernel/sched-dyna.h
- include/cobalt/kernel/sched.h
- include/cobalt/kernel/thread.h
- include/alchemy/task.h

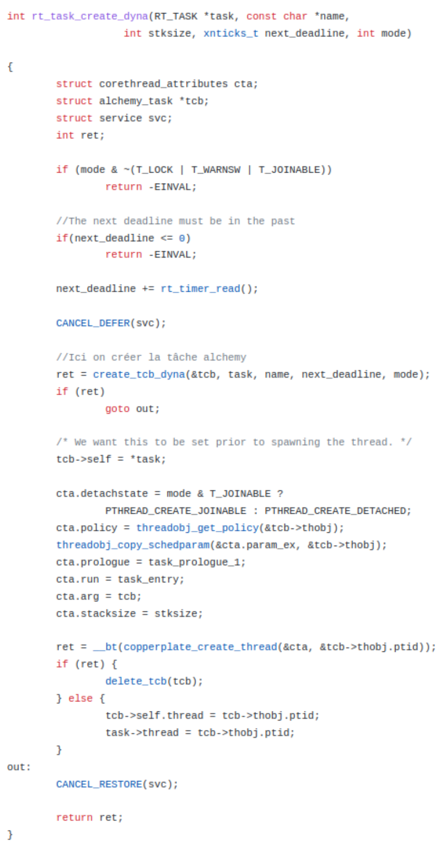
Minor modifications

- include/mercury/boilerplate/sched.h
- kernel/cobalt/posix/sched.c
- include/cobalt/uapi/sched.h
- include/cobalt/kernel/schedparam.h
- include/cobalt/kernel/list.h
- include/cobalt/kernel/schedqueue.h
- lib/copperplate/internal.c
- lib/copperplate/threadobj.c

On top of that, the purpose of the last commit related to EDF policy implementation (see here) is only to have a proper compilation of xenomai source code files without warning/errors.

Basically, the implementation of the EDF policy over Xenomai relies on :

- The creation of a new xnsched_class, called **dyna**, to host tasks scheduled using the EDF policy. (cf. file sched_dyna.c available here)
- Creation of function **rt_task_create_dyna** to pass the deadline of the task. (cf figure below)



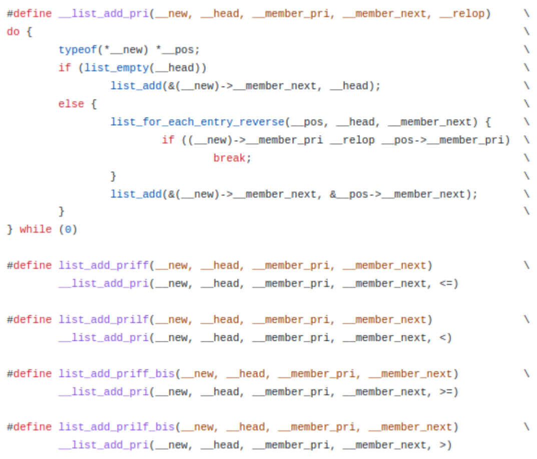
```c
int rt_task_create_dyna(RT_TASK *task, const char *name,
                        int stksize, xnticks_t next_deadline, int mode)

{
        struct corethread_attributes cta;
        struct alchemy_task *tcb;
        struct service svc;
        int ret;

        if (mode & ~(T_LOCK | T_WARNSW | T_JOINABLE))
                return -EINVAL;

        //The next deadline must be in the past
        if(next_deadline <= 0)
                return -EINVAL;

        next_deadline += rt_timer_read();

        CANCEL_DEFER(svc);

        //Ici on créer la tâche alchemy
        ret = create_tcb_dyna(&tcb, task, name, next_deadline, mode);
        if (ret)
                goto out;

        /* We want this to be set prior to spawning the thread. */
        tcb->self = *task;

        cta.detachstate = mode & T_JOINABLE ?
                PTHREAD_CREATE_JOINABLE : PTHREAD_CREATE_DETACHED;
        cta.policy = threadobj_get_policy(&tcb->thobj);
        threadobj_copy_schedparam(&cta.param_ex, &tcb->thobj);
        cta.prologue = task_prologue_1;
        cta.run = task_entry;
        cta.arg = tcb;
        cta.stacksize = stksize;

        ret = __bt(copperplate_create_thread(&cta, &tcb->thobj.ptid));
        if (ret) {
                delete_tcb(tcb);
        } else {
                tcb->self.thread = tcb->thobj.ptid;
                task->thread = tcb->thobj.ptid;
        }
out:
        CANCEL_RESTORE(svc);

        return ret;
}
```

Figure 1: Function rt_task_create_dyna

```c
struct __sched_deadline_param {
        __u64 sched_absolute_deadline;
        __u64 sched_relative_deadline;
};

struct sched_param_ex {
        int sched_priority;
        union {
                struct __sched_rr_param rr;
                struct __sched_deadline_param deadline;
        } sched_u;
};
```

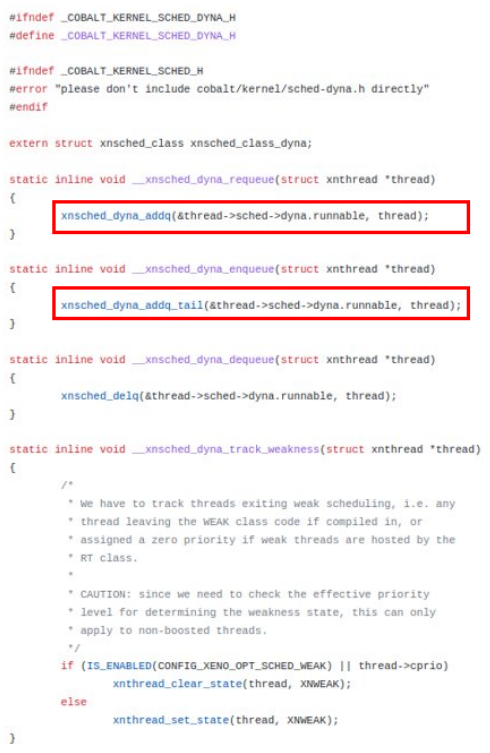
Figure 2: struct sched_deadline_param for EDF policy

- To enqueue a task in a list using round-robin policy, we need to pay attention to the priority of each task. However when we enqueue a task in a list using EDF policy (or EDH policy as well), we need to pay attention to the deadline of each task. While the value 0 correspond to a lowest priority when using round-robin policy (I mean the task with the lowest priority), a task with a small deadline will be the first queued element (lower is the deadline value, higher is the priority of the task).
  Figures below demonstrate how to integrate these modifications. (in folder include /include/cobalt/kernel, cf. here)



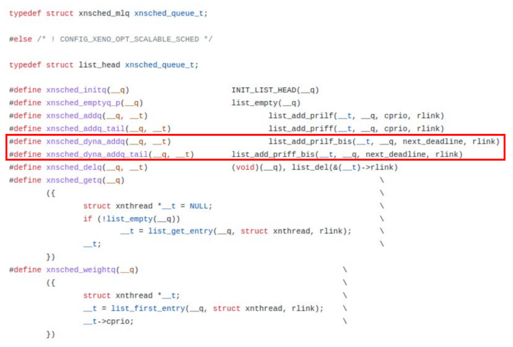
```c
#define __list_add_pri(__new, __head, __member_pri, __member_next, __relop)    \
do {                                                                           \
        typeof(*__new) *__pos;                                                 \
        if (list_empty(__head))                                                \
                list_add(&(__new)->__member_next, __head);                     \
        else {                                                                 \
                list_for_each_entry_reverse(__pos, __head, __member_next) {    \
                        if ((__new)->__member_pri __relop __pos->__member_pri) \
                                break;                                         \
                }                                                              \
                list_add(&(__new)->__member_next, &__pos->__member_next);      \
        }                                                                      \
} while (0)

#define list_add_priff(__new, __head, __member_pri, __member_next)             \
        __list_add_pri(__new, __head, __member_pri, __member_next, <=)

#define list_add_prilf(__new, __head, __member_pri, __member_next)             \
        __list_add_pri(__new, __head, __member_pri, __member_next, <)

#define list_add_priff_bis(__new, __head, __member_pri, __member_next)         \
        __list_add_pri(__new, __head, __member_pri, __member_next, >=)

#define list_add_prilf_bis(__new, __head, __member_pri, __member_next)         \
        __list_add_pri(__new, __head, __member_pri, __member_next, >)
```

Figure 3: File list.h



```
#ifndef _COBALT_KERNEL_SCHED_DYNA_H
#define _COBALT_KERNEL_SCHED_DYNA_H

#ifndef _COBALT_KERNEL_SCHED_H
#error "please don't include cobalt/kernel/sched-dyna.h directly"
#endif

extern struct xnsched_class xnsched_class_dyna;

static inline void __xnsched_dyna_requeue(struct xnthread *thread)
{
        xnsched_dyna_addq(&thread->sched->dyna.runnable, thread);
}

static inline void __xnsched_dyna_enqueue(struct xnthread *thread)
{
        xnsched_dyna_addq_tail(&thread->sched->dyna.runnable, thread);
}

static inline void __xnsched_dyna_dequeue(struct xnthread *thread)
{
        xnsched_delq(&thread->sched->dyna.runnable, thread);
}

static inline void __xnsched_dyna_track_weakness(struct xnthread *thread)
{
        /*
         * We have to track threads exiting weak scheduling, i.e. any
         * thread leaving the WEAK class code if compiled in, or
         * assigned a zero priority if weak threads are hosted by the
         * RT class.
         *
         * CAUTION: since we need to check the effective priority
         * level for determining the weakness state, this can only
         * apply to non-boosted threads.
         */
        if (IS_ENABLED(CONFIG_XENO_OPT_SCHED_WEAK) || thread->cprio)
                xnthread_clear_state(thread, XNWEAK);
        else
                xnthread_set_state(thread, XNWEAK);
}
```

Figure 4: File sched_dyna.h

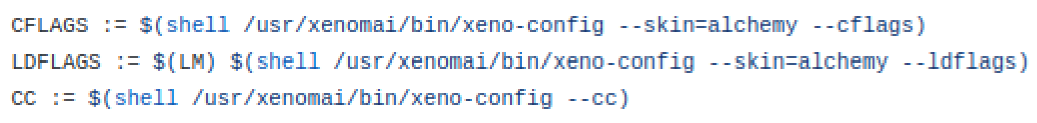


```
typedef struct xnsched_mlq xnsched_queue_t;

#else /* ! CONFIG_XENO_OPT_SCALABLE_SCHED */

typedef struct list_head xnsched_queue_t;

#define xnsched_initq(__q)                      INIT_LIST_HEAD(__q)
#define xnsched_emptyq_p(__q)                   list_empty(__q)
#define xnsched_addq(__q, __t)                          list_add_prilf(__t, __q, cprio, rlink)
#define xnsched_addq_tail(__q, __t)                     list_add_priff(__t, __q, cprio, rlink)
#define xnsched_dyna_addq(__q, __t)                     list_add_prilf_bis(__t, __q, next_deadline, rlink)
#define xnsched_dyna_addq_tail(__q, __t)        list_add_priff_bis(__t, __q, next_deadline, rlink)
#define xnsched_delq(__q, __t)                  (void)(__q), list_del(&(__t)->rlink)
#define xnsched_getq(__q)                                                   \
        ({                                                                  \
                struct xnthread *__t = NULL;                                \
                if (!list_empty(__q))                                       \
                        __t = list_get_entry(__q, struct xnthread, rlink);  \
                __t;                                                        \
        })
#define xnsched_weightq(__q)                                        \
        ({                                                          \
                struct xnthread *__t;                               \
                __t = list_first_entry(__q, struct xnthread, rlink); \
                __t->cprio;                                         \
        })
```

Figure 5: File schedparam.h

2.3 EDF Source code Validation

Now, let's have a look to the validation process to ensure the EDF policy is working well over Xenomai through simple examples.
The source code dedicated to create/start a pre-defined number of real-time tasks scheduled with EDF policy is available here.
A first important thing to do in order to compilate source code using Xenomai Alchemy API is to define the CFLAGS,LDFLAGS and CC Flags as shown below :



```
CFLAGS := $(shell /usr/xenomai/bin/xeno-config --skin=alchemy --cflags)
LDFLAGS := $(LM) $(shell /usr/xenomai/bin/xeno-config --skin=alchemy --ldflags)
CC := $(shell /usr/xenomai/bin/xeno-config --cc)
```

Figure 6: Define flags in Makefile.h

Then the architecture of the source code, **edfAsserts** is really simple.
Indeed we have :

- the main
- the listener (file **listener.c/.h**

- the controller (file **loop**$_t ask.c/.h$
The execution process is really simple:

1. follow the README.md to execute the ./EDFtest binary (link here)

2. Choose to use dynamic priority,enter the number of tasks and desired parameters for each task in the terminal

3. Oberve the results (also display in the terminal).

To validate our results (in order to ensure EDF policy is working well over Xenomai), the 1st step could be to execute again our binary but this time, we will decide to use fixed priority policy (basically here the dafult one is the round-robin policy) and then compare both previous results with EDF and new results with round-robin.

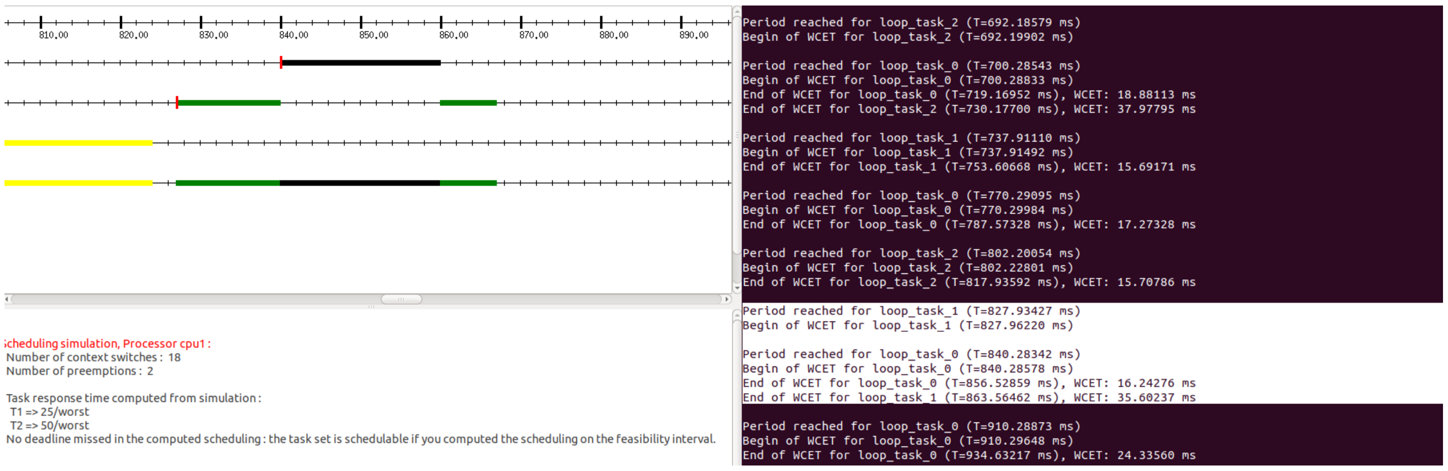
Here is one example with the following parameters :

Tasks	Deadlines	WCET
Task 1	70ms	20ms
Task 2	90ms	20ms
Task 3	110ms	20ms

Table 1: Test case n°1 to validate EDF policy implementation over Xenomai (Period = Deadline

First we could use these values and observe the behaviour with EDF policy and Round-robin policy. Below are the parameters we entered when executing the binary EDFtest (keep in mind the conversion rate => 70 000 000 units = 70 ms):
TODO : Ajouter screenshots des paramètres utilisés avec/sans politique EDF
Then we can compare the 2 execution. The figure 8 exhibits the 3 tasks scheduled with round-robin policy. There is no preemption occuring and the priority of execution is directly linked to the priority of each tasks as describes in table 1.
The figure 9 exhibits the 3 tasks scheduled with EDF policy. We can indeed clearly see the differences with figures 8. We can observe preemptions and tasks being scheduled according to their deadlines.
TODO : Ajouter screenshots de l'exécution des tâches avec politique round-robin / avec politique EDF
In order to clearly observe preemptions (and so be sure to have a proper EDF scheduling activity), we can directly compare our results with a tool, called Cheddar. Cheddar allows you to model software architectures of real-time systems in order to check their schedulability (available here).
Let's simulate with Cheddar using parameters define in table 1 (examples and configuration files for Cheddar are available here if needed) and let's compare with what we obtained using EDF policy over Xenomai:

- In this first view, the task T2 is executing. On both view (Cheddar on the left, Xenomai on the right) at T= 840 ms , T1 which has a shortest deadline than T2 is preempted it.

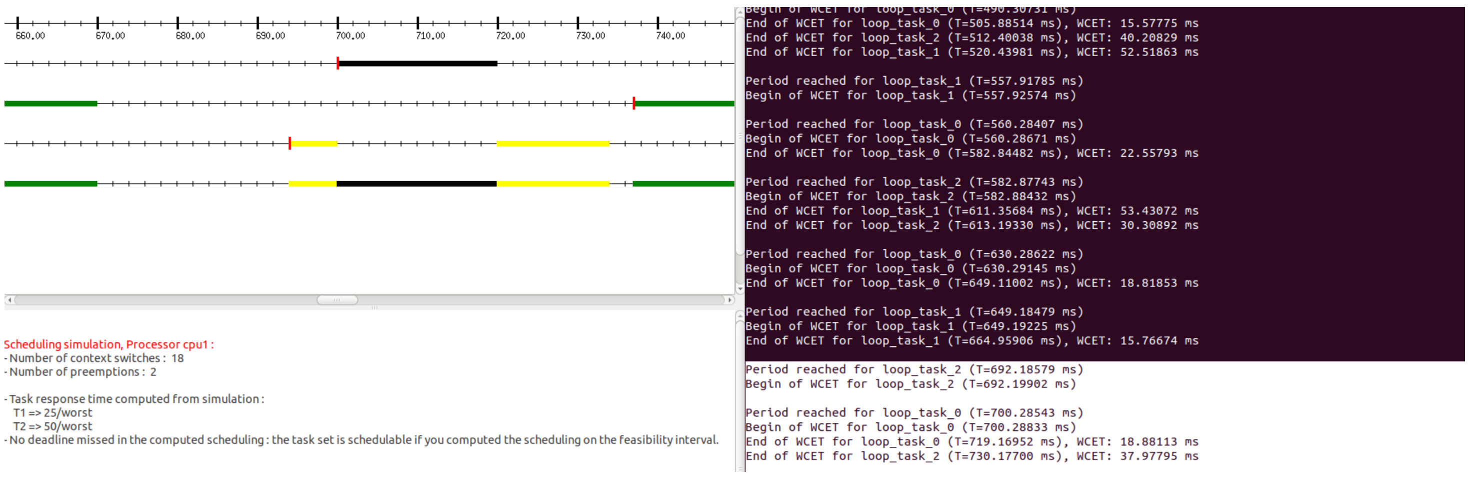


```
Scheduling simulation, Processor cpu1 :
Number of context switches :  18
Number of preemptions :  2

Task response time computed from simulation :
 T1 => 25/worst
 T2 => 50/worst
No deadline missed in the computed scheduling : the task set is schedulable if you computed the scheduling on the feasibility interval.
```

```
Period reached for loop_task_2 (T=692.18579 ms)
Begin of WCET for loop_task_2 (T=692.19902 ms)

Period reached for loop_task_0 (T=700.28543 ms)
Begin of WCET for loop_task_0 (T=700.28833 ms)
End of WCET for loop_task_0 (T=719.16952 ms), WCET: 18.88113 ms
End of WCET for loop_task_2 (T=730.17700 ms), WCET: 37.97795 ms

Period reached for loop_task_1 (T=737.91110 ms)
Begin of WCET for loop_task_1 (T=737.91492 ms)
End of WCET for loop_task_1 (T=753.60668 ms), WCET: 15.69171 ms

Period reached for loop_task_0 (T=770.29095 ms)
Begin of WCET for loop_task_0 (T=770.29984 ms)
End of WCET for loop_task_0 (T=787.57328 ms), WCET: 17.27328 ms

Period reached for loop_task_2 (T=802.20054 ms)
Begin of WCET for loop_task_2 (T=802.22801 ms)
End of WCET for loop_task_2 (T=817.93592 ms), WCET: 15.70786 ms

Period reached for loop_task_1 (T=827.93427 ms)
Begin of WCET for loop_task_1 (T=827.96220 ms)

Period reached for loop_task_0 (T=840.28342 ms)
Begin of WCET for loop_task_0 (T=840.28578 ms)
End of WCET for loop_task_0 (T=856.52859 ms), WCET: 16.24276 ms
End of WCET for loop_task_1 (T=863.56462 ms), WCET: 35.60237 ms

Period reached for loop_task_0 (T=910.28873 ms)
Begin of WCET for loop_task_0 (T=910.29648 ms)
End of WCET for loop_task_0 (T=934.63217 ms), WCET: 24.33560 ms
```

Figure 7: Preemption of Task T2 by T1

- In this second view, the task T3 is executing.At T= 700 ms , T1 which has a shortest deadline than T3 is preempted it.

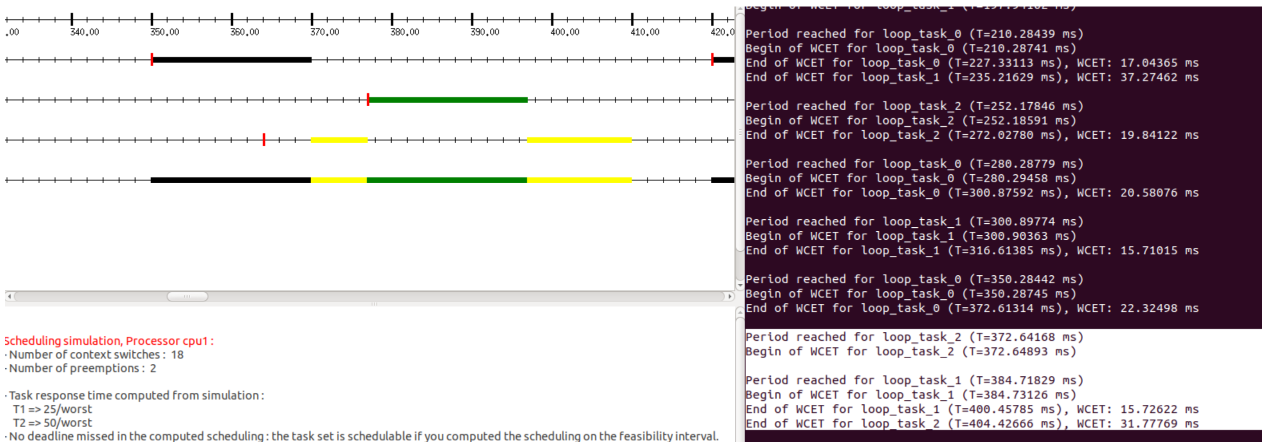


```
Scheduling simulation, Processor cpu1 :
- Number of context switches :  18
- Number of preemptions :  2

- Task response time computed from simulation :
   T1 => 25/worst
   T2 => 50/worst
- No deadline missed in the computed scheduling : the task set is schedulable if you computed the scheduling on the feasibility interval.
```

```
Begin of WCET for loop_task_0 (T=490.30731 ms)
End of WCET for loop_task_0 (T=505.88514 ms), WCET: 15.57775 ms
End of WCET for loop_task_2 (T=512.40038 ms), WCET: 40.20829 ms
End of WCET for loop_task_1 (T=520.43981 ms), WCET: 52.51863 ms

Period reached for loop_task_1 (T=557.91785 ms)
Begin of WCET for loop_task_1 (T=557.92574 ms)

Period reached for loop_task_0 (T=560.28407 ms)
Begin of WCET for loop_task_0 (T=560.28671 ms)
End of WCET for loop_task_0 (T=582.84482 ms), WCET: 22.55793 ms

Period reached for loop_task_2 (T=582.87743 ms)
Begin of WCET for loop_task_2 (T=582.88432 ms)
End of WCET for loop_task_1 (T=611.35684 ms), WCET: 53.43072 ms
End of WCET for loop_task_2 (T=613.19330 ms), WCET: 30.30892 ms

Period reached for loop_task_0 (T=630.28622 ms)
Begin of WCET for loop_task_0 (T=630.29145 ms)
End of WCET for loop_task_0 (T=649.11002 ms), WCET: 18.81853 ms

Period reached for loop_task_1 (T=649.18479 ms)
Begin of WCET for loop_task_1 (T=649.19225 ms)
End of WCET for loop_task_1 (T=664.95906 ms), WCET: 15.76674 ms

Period reached for loop_task_2 (T=692.18579 ms)
Begin of WCET for loop_task_2 (T=692.19902 ms)

Period reached for loop_task_0 (T=700.28543 ms)
Begin of WCET for loop_task_0 (T=700.28833 ms)
End of WCET for loop_task_0 (T=719.16952 ms), WCET: 18.88113 ms
End of WCET for loop_task_2 (T=730.17700 ms), WCET: 37.97795 ms
```

Figure 8: Preemption of Task T3 by T1

- In this last view, the task T3 is executing.At T= 384 ms , T2 which has a shortest deadline than T3 is preempted it.



```
Scheduling simulation, Processor cpu1 :
- Number of context switches : 18
- Number of preemptions : 2

- Task response time computed from simulation :
  T1 => 25/worst
  T2 => 50/worst
- No deadline missed in the computed scheduling : the task set is schedulable if you computed the scheduling on the feasibility interval.
```

```
Period reached for loop_task_0 (T=210.28439 ms)
Begin of WCET for loop_task_0 (T=210.28741 ms)
End of WCET for loop_task_0 (T=227.33113 ms), WCET: 17.04365 ms
End of WCET for loop_task_1 (T=235.21629 ms), WCET: 37.27462 ms

Period reached for loop_task_2 (T=252.17846 ms)
Begin of WCET for loop_task_2 (T=252.18591 ms)
End of WCET for loop_task_2 (T=272.02780 ms), WCET: 19.84122 ms

Period reached for loop_task_0 (T=280.28779 ms)
Begin of WCET for loop_task_0 (T=280.29458 ms)
End of WCET for loop_task_0 (T=300.87592 ms), WCET: 20.58076 ms

Period reached for loop_task_1 (T=300.89774 ms)
Begin of WCET for loop_task_1 (T=300.90363 ms)
End of WCET for loop_task_1 (T=316.61385 ms), WCET: 15.71015 ms

Period reached for loop_task_0 (T=350.28442 ms)
Begin of WCET for loop_task_0 (T=350.28745 ms)
End of WCET for loop_task_0 (T=372.61314 ms), WCET: 22.32498 ms

Period reached for loop_task_2 (T=372.64168 ms)
Begin of WCET for loop_task_2 (T=372.64893 ms)

Period reached for loop_task_1 (T=384.71829 ms)
Begin of WCET for loop_task_1 (T=384.73126 ms)
End of WCET for loop_task_1 (T=400.45785 ms), WCET: 15.72622 ms
End of WCET for loop_task_2 (T=404.42666 ms), WCET: 31.77769 ms
```

Figure 9: Preemption of Task T3 by T2

Now that we have verified our EDF policy over Xenomai is working correctly, we can then have a look to the theory of ED-H policy before implementing this policy over Xenomai too.

3
SECTION

EH Scheduling Algorithms

3.1 Introduction

Since decades batteries are accustomed store energy and providing the necessary power source for mobile, embedded and the remote system application.The development of battery techniques doesn't follow Moore's law . There are constraints like physical size,limited electric quantity, high cost etc. Restrict the performance of application like embedded systems, wireless sensor networks and low power electronics.

There exists ambient number of energy sources from environment (tidal,solar , windmill , etc.) which may be harvested and convert them into electric sources and It can be be later used to power wireless systems, this is often defined as energy harvesting or energy scavenging[**?**].

Energy harvesting embedded systems has been emerged as a significant topic within the field of research.There has been many research to style embedded systems which might efficiently harvest energy. The core objective of designing such systems is just to make embedded systems optimally with the available energy source.The systems are implemented in such the way that managing the power to perform operation supported current energy unit stored within the storage unit/battery and may predict the incoming energy in the future. Below Figure depicts how

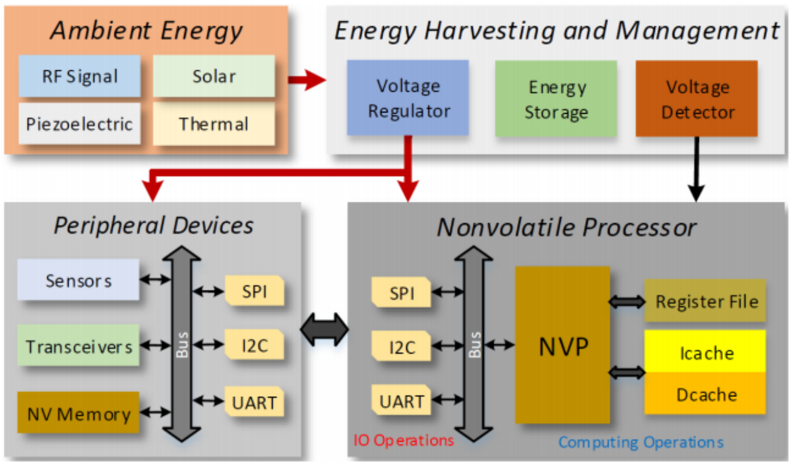


Figure 10: Architecture of Energy Harvesting Powered Sensing System
[**?**]

energy is harvested from renewable environment sources.The harvested energy is regulated by energy management module to power processors and multiple peripherals.The peripheral devices consists of includes sensors,transceivers. Sensors collect the data from environment such to temperature, humidity from the environment and transceivers transmits the collected data to server for further computation.

3.2 Energy storage systems

3.2.1 Battery-Related Concepts

The battery is a main power source from which energy is consumed in application mobile, portable applications thus battery capacity limits the performance.The production of current in a very given time is termed as capacity of the battery. The capacity within the battery is expressed in ampere-hours (Ah) or milliampere -hours (mAh) [**?**].

There are various parameters that affect the capacity of battery like cell constructions period of time,charge and discharge cycles.The battery's capacity is approximately constant under normal operation condition during its lifetime. The lifetime of battery is reduced after they are charged incorrectly, hence they should be charged only under suitable conditions with correct charge current.To deplete the battery completely so recharge it's an acceptable method to extend the lifetime of battery.

3.2.2 Energy Production Model

Energy harvester unit (eg.solar panel) produces energy from ambient renewable environment sources and converts it into electrical power at each time *(t)* which later is used in embedded systems.As the energy produced from the source is unpredictable and not necessarily a constant value , we can still predict the availability in a short term perspective with worst case changing rate (WCCR) on the harvested source power output ,it is characterized by an instantaneous charging rate P_p*(t)* that incorporates all the losses [**?**].Clearly we do not The energy which has been harvested over the time interval *(*t_1, t_2*)*is represented below in the following equation [1].

$$E_p(t_1, t_2) = \int_{t_1}^{t_2} P_p(dt) \tag{1}$$

The energy utilized in any unit time slot is not less than the energy produced in the same unit time-slot.Whenever at time t a job is executed the residual capacity of energy storage is never increased.

3.2.3 Energy Storage model

The energy storage unit used in our systems (battery or supercapacitor) which are recharged up to a nominal capacity *C*,which corresponds to the maximum amount of energy units stored at that time *t*. As we only use an ideal storage unit, we neglect the amount of energy wasted in charging and discharging process.If the energy recharged in the battery C exceeds maximum capacity *(C>*C_{max}*)*, then we consider the energy stored in battery in C_max.The energy level in the battery is remain between two intervals

$$C = C_{max} - C_{min} \tag{2}$$

C_{max} and C_{min} are upper and lower bounds of the storage unit and C_{min} is not zero as there must reserved energy for worst case scenarios
If any task executing on the processor it consumes energy that is drawn from the reservoir/battery. The energy stored in the battery at time t is denoted by *E(t)*.There is no leakage of energy from the reservoir over time. If at time t the energy units in battery is completely discharged $0 \leq E(t) < C_max$ is denoted by $E(t) \approx 0$.The energy units in the battery at *t=0* is fully charged ie *E(0)=C*.

4

SECTION

ED-H Scheduling algorithm

4.1 Introduction

In this section we will discuss about ED-H scheduler.In classic EDF scheduler the processor is never let idle and job is executed as soon as possible thus utilizes all the energy stored in the reservoir while not saving for future energy needs.

Let us assume that task set is time feasible by EDF,the job j_i of τ_i will encounter energy starvation only from execution of j_j of τ_j whose arrival time a_j<a_i with $d_j > d_i$.Hence if the scheduler can predict the future arrival time of jobs and energy production it can help EDF to anticipate energy starvation and deadline violation.Thus in energy constrained systems, it is sometimes necessary not to execute the ready job if it will prevent execution of future arriving jobs due to time constraint or energy constraint [**?**].

The main principle of ED-H is authorize job execution as long as there occurs no starvation in future.Hence ED-H is modified version of EDF which is solely used for energy harvesting constrained jobs.ED-H verifies the timing constraints , energy requirements as well as the replenishment rate of the storage unit for future arriving jobs.If one of these conditions is not fulfilled, the processor has to idle so that the storage unit recharges sufficiently [**?**].

4.2 ED-H scheduling concepts

A Static Analysis In this section we will study the ED-H concepts for understanding the feasibility of a job set with both energy and deadline constraints.

The energy demand of $g_w(t_1,t_2)$ is defined as the total computation time of all the uncompleted task whose a_i and d_i lies between time interval(t_1,t_2). The energy demand of the jobset τ in given time interval *(t_1,t_2)* is calculated by

$$g_w(t_1,t_2) = \sum_{t_1 \le r_k, d_k \le t_2} E_k \tag{3}$$

The static slack energy SSE_τ on a jobset τ for the time interval (t_1,t_2) is calculated by

$$SSE_\tau(t_1,t_2) = C + E_p(t_1,t_2) - g_w(t_1,t_2) \tag{4}$$

where,$E_p(t_1,t_2)$=$\int_{t_1}^{t_2} P\_p(t)dt$ it is the amount of the energy that is produced by the source between t_1 and t_2.The energy demand g_w 3.$SSE_tau(t_1,t_2)$ gives the maximum energy that could be made available within the time interval (t_1, t_2) after executing jobs of τ with arrival time at or after t_1 and deadline at or before t_2 [**?**].

B Dynamic Analysis In this section we will discuss about dynamic slack energy SE with respective to the current time t_c in the schedule produced by τ.
Slack energy of a job τ for the current time t_c is calculated by

$$SE_(t_c) = E(t_c)E_p(t_c, d_i) - g_w(t_c, d_i) \tag{5}$$

$SE_(t_c)$ is calculated to determine the maximum energy that can be consumed between t_c, d_i whilst guaranteeing that future arriving jobs at or after t_c with deadline at or before d_i have sufficient energy for its execution.There will occur energy starvation for job of τ if $SE_(t_c) =$ and its deadline if after d_i between t_c and d_i.Thus ED-H provides clairvoyance on the jobs arrival time and energy production to predict possible energy starvation or deadline miss.

The maximum slack energy that could be consumed by currently active jobs at t_c while still guaranteeing the energy feasibility for the jobs that may preempt it in future is called preemption slack energy PSE_τ

$$PSE_\tau = \min_{t_c < r_i < d_i < d} SE_{\tau i}(t_c). \tag{6}$$

5
SECTION

ED-H Simulator software

5.1 Introduction

In this section we will discuss the core objective here is to develop a simulator software in C language for scheduling according to ED-H algorithm.Simulating the ED-H scheduling policies are studied in this project.

The code is developed in C because as it provides optimized machine instruction for the given input which in result increases the embedded systems performance.Large memory is required when using high-level languages, when compared to high level languages we can directly control the memory using pointers and perform various process with them.We can dynamically allocate the memory, based on how much memory is required.Hence we found optimal to develop the software in C.

5.2 Simulator structure

The simulator software is divided in four parts.

- The first part consists of receiving the data inputs of the task set **(T)** from the user.
- The second part is defined by analyzing all the feasibility checks based on ED-H algorithm.
- The third part comprises of scheduling the task based on the input from the user according to the ED-H algorithm.
- The final part is output displaying the scheduling gantt chart.

5.3 Simulator Packages

The simulator consists of three main packages

- EDH.c
  In this file we check the feasibility of the schedule based upon static slack energy *(SSE)* and static slack time *(SST)*.Also the scheduling of the task according the ED-H algorithm is achieved from the functions defined in this file.

- Task.c
  Here the parameters from the user are received for each task, that is passed to task struct which is defined in this file.

- Schedule_trace.c
  In the file we display the gantt chart produced by the simulator.

5.4 Simulator Algorithm

The simulator software comprises of two algorithm structures.

- First algorithm checks if the taskset **T** is feasible i.e static slack energy *(SSE≥ 0)* and *(SST≥ 0)*.If the taskset **T** is feasible the simulation proceeds to schedule the with the second algorithm ,else the simulation is terminated.

- Once the taskset **T** is feasible the scheduler is called to simulate the tasks. The scheduler is implemented using state machine logic and the simulation is carried out each time interval checking the energy requirements of the scheduler.

5.4.1 Simulator Functions

A Feasibility Checks

As mentioned in the earlier section the the simulator algorithm is differentiated in two parts. In this first part will discuss about the function generate_ftable() which calculates the feasibility test of the taskset **T** that is defined in EDH.c file. Once the hyper period is calculated from all the periods (P_i) in the LCM() function, we pass on the tasks τ_i deadline di and arrival time a_i to the function generate_ftable().

After the deadline d_i and arrival a_i time is passed, we need to calculate the processor demand h_w and energy demand g_w between each release time and each deadline time of every task τ_i in the taskset **T**.

- The processor demand h_w is calculated by summation of the computation time of all the task between the earliest arrival time a_i and deadline d_i (**??**).

- The energy demand g_w is calculated by summation of the energy consumption of all the task between the earliest arrival time a_i and deadline d_i.

Next arrival times of tasks are compared and the earliest one is taken into consideration to calculate the the processor demand h_w and energy demand g_w till the deadline of each task, be it τ_i or τ_j.
For instance, if the earliest arrival time were to be 0, then it would loop as such: 0-d_i,0-d_j 0-d_i+$_1$,0-d_j+$_1$,... till 0-hyperperiod.
Following that, the earliest arrival time will update to the next earliest arrival time possible e.g. $T_{i,nextarrival}$

where $T_{i,nextarrival} = a_i + p_i$.

We set to two flags $pending\_next\_arrive$ and $pending\_same\_arrive$ to true to check if processor demand h_w and energy demand g_w have already been calculated or not for current arrival time a_i.The flags are set in order to check if two or more task have either have same arrival time a_i or deadline d_i , so that the processor demand h_w and energy demand g_w are not calculated twice for same time interval.

$pending\_next\_arrive$ is FALSE when the earliest next arrival exceeds hyperperiod, so loop can be exited and feasibility test can be terminated.

$pending\_same\_arrive$ is FALSE when all possibilities for that arrival time have been checked - i.e. for $next_{ar} = 0$, after 0-6,0-8.....0-24 (hyperperiod), no need to check again for $next_{ar} = 0$; $next_{ar}$ can be incremented by that task's period to start next round of checking e.g. 6-8, 6-12,....6-24.

This also ensures that the arrival time is not checked twice if the next task has same arrival time as first.

Subsequently once we obtain earliest arrival time of task τ_i from the task set **T** we need to calculate the h_w and g_w till the deadlines d_i of each task τ_i in the **T**.

After we acquire the time intervals t_1 and t_2 we pass the value to $calculate\_line()$ function and calculate the processor demand h_w and energy demand g_w of all the tasks who's arrival time a_i and deadline d_i lies between this time interval t_1 and t_2.After the h_w g_w is calculated we can use it find out static slack energy *(SSE)* and static slack time *(SST)* in the function $calculate\_line()$.

The computation is repeated till we receive the all the time intervals$t_1 and t_2$. The $pending\_same\_arrive$ flag is set to FALSE after we receive all the computation for h_w and g_w from time t_1 which is the latest arrival time a_i till the deadline d_i and d_j of each task in the taskset. Then we update the $pending\_next\_arrive$ to compute the values from next arrival time of task τ_{i+1}

If either of *(SSE≥ 0)* and *(SST≥ 0)* this two conditions is not satisfied we exit the simulation process.Once the ED-H feasibility test are passed, We calculate the sufficient schedulability condition and energy feasibility condition in the feasibilityChecks() function. If any of these three tests doesn't pass the simulation process is terminated.

```
function generate-ftable
for each task do
    $Task[i].next_{arrival} \leftarrow Task[i].arrival$
    Task[i].next$_{deadline} \leftarrow Task[i].deadline$
end
Bool pending-next-arrive,pending-same-arrive = TRUE
while pending-next-arrive do
    if Task[i].next$_{arrival} \geq Hyperperiod$ then
        pending-next-arrive = FALSE
    end
    while pending-same-arrive do
        endIndex $\leftarrow find-min-end$                                          end
        StartIndex $\leftarrow find-min-start$
        Earliestarr $\leftarrow Task[StartIndex].next_{arrival}$
        Earliestdeadline $\leftarrow Task[endIndex].next_{deadline}$
        if Earliestdeadline $\leq Hyperperiod$ then
            Feasibility-checkEarliestarr,Earliestdeadline
        end
        if Earliestdeadline $\geq Hyperperiod$ then
            pending-same-arrive = FALSE
        end
         end
        Task[StartIndex].next$_{arrival}+= Task[StartIndex].period$

```

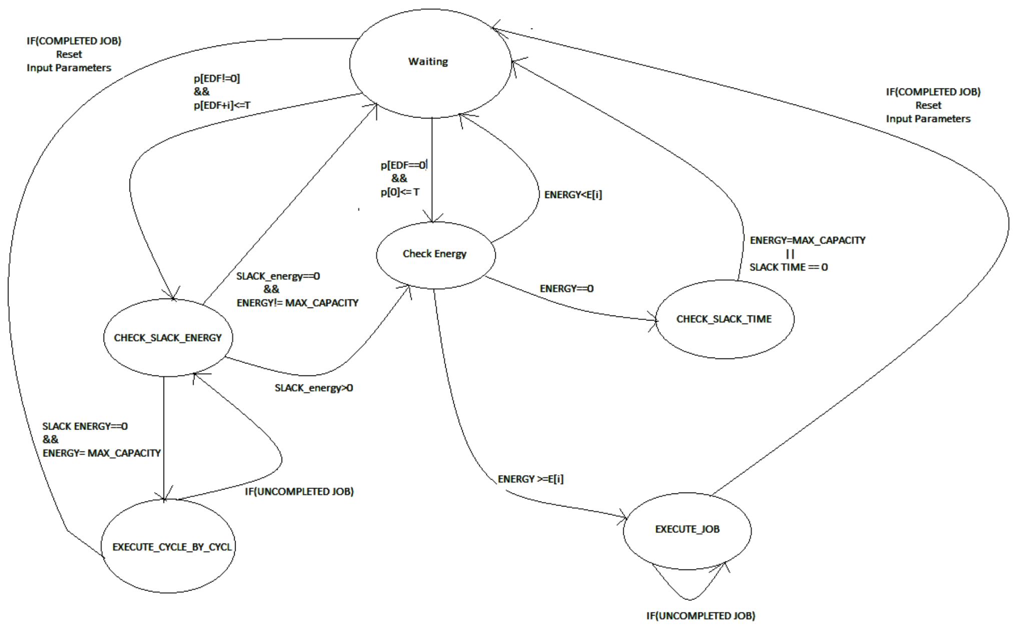
```
function Feasibility-check (Earliestarr, Earliestdeadline)
Calculate $hw, gw, SSE, SST_W$
if $SSE < 0$ or $SST_W < 0$
 then
    Bool feasibility-pass = false;
end
return feasibility-pass
```

B ED-H Scheduler

Once the taskset **T** has passed all the feasibility conditions and tests, the simulation process enters the second part where the simulator schedules according to the ED-H algorithm.The simulation process for the scheduling is developed using state machine.
State machine logic was optimal for task scheduling as we need take note of different parameters - energy consumption of the task τ_i , slack energy *(SE)* and, slack time *(ST)* are calculated for each time interval t over the hyperperiod.

There are total six states used in the simulator:

- WAITING
- CHECK_ENERGY
- CHECK_SLACK_ENERGY
- CHECK_SLACK_TIME
- EXECUTE_JOB
- EXECUTE_CYCLE_BY_CYCLE



1 WAITING

This state is the one that is executed first when the execution starts after checking the feasibility test.In this state, sort() function is called and we pass the tasks' deadlines as input parameters to set the priority of the tasks τ according to earliest deadline first. The highest priority task is stored in the 0^{th} position in run-queue.

Next, within the waiting state itself, scheduling too happens. The scheduler can necessarily only schedule tasks that are released before or at the current time t_c, known as eligible tasks.

To begin, if the highest-priority task τ_0 is eligible, the processor switches to the CHECK_ENERGY state to check if the energy E in battery is sufficient for execution of task τ_i

.

However, if τ_0 is not eligible, the next highest priority task τ_1 is checked for eligibility and if that too is not, then τ_2 and so on. In this way, it goes down the priority list until it finds an eligible task. Since this task is not of highest priority, unlike τ_0, and there is a possibility of pre-emption by a higher priority task in future, processor instead enters CHECK_SLACK_ENERGY state, where it is checked if slack energy is sufficient to authorize the task.

If there is no task ready in the run-queue the scheduler goes to idle state and battery is recharged till the next arrival time a_i of task in the run-queue.

Algorithm 1: WAITING State

```
Input: Tasks ($\tau_0$ to $\tau_{fin}$) and their arrival times $t_{arr}$, current time $t_c$
Output: Tasks sorted by priority for energy state checking
Data: prioritylist
EDF = 0                                                // initialize counter
/* Prioritizing tasks by earliest deadlines                                  */
prioritylist ← $Sort(Tasks)$
if prioritylist[0].$t_{arr} < t_c$ then
    case CHECK ENERGY state;
end
else
    for EDF ∈ [1, fin] do
        if prioritylist[EDF].$t_{arr} < t_c$ then
            case CHECK SLACK ENERGY state;
            break;
        end
        if EDF == fin then
            No task found;
            Enter idle period for recharging cycle;
            break;
        end
        EDF++;
    end
end
```

2 CHECK_ENERGY

After the task has been selected from the run-queue, the scheduler enters the CHECK_ENERGY state. Here, the energy units E in the battery are checked to see if they are sufficient to execute the task which is currently ready. If the energy in battery is completely depleted *(E==0)*, the processor cannot execute tasks – so we need to introduce an idle period during which the battery can recharge, known as the recharge cycle.

Slack time (ST) is the maximum time the processor can be idle to avoid deadline miss (also known as time starvation). Hence, the processor needs to constantly check the slack time to see how long it can idle.

To begin, if ST has not yet been checked at t_c, it switches to CHECK_SLACK_TIME. However, if it has been checked and still, battery level is zero, the scheduler goes to the idle mode and charges for one time period. After this one recharge, it does CHECK_SLACK_TIME again to avoid any deadline miss (in case slack time is 0 at the new t_c and it needs to switch to WAITING state to take in updated tasks).

If battery energy is non-zero *(E≠0)*, there is a possibility to begin a task but that is not enough - the scheduler should also verify that there will be sufficient energy to execute it completely. This brings the need for residual energy forecasting.

2.1 Residual Energy Forecasting

The amount of residual energy in the battery that will be remaining after task completion at a future time depends on both the current energy level and the percentage of the task that has already been executed (this is tracked by a boolean member variable belonging to the struct 'task', known as **ongoing**).

ongoing = 0 if the active task is yet to begin and = 1 if it has already executed some portion.

The scheduler now calculates the remaining energy units E after executing the task. If **ongoing** = 0, residual energy in battery after job will be completed is forecast as:

$$Remaining\_energy = E(t) + (P_p * C_i) - e_i \tag{7}$$

If **ongoing** = 1,residual energy in battery after job will be completed is forecast as:

$$Remaining\_energy = E(t) + (P_p * (C_i - executed\_time)) - (e_i - consumed\_energy) \tag{8}$$

where

- E(t)= Energy Units in the storage unit(Battery) at time t
- P_p=Rechargeable power constant over the hyper-period
- C_i= Computation time of task τ_i
- $executed\_time$= Already executed time of task τ_i
- e_i= Total energy consumption of task start to end τ_i

- $consumed\_energy$= Already consumed energy of task τ_i corresponding to its residual computation time.

Maximum possible battery energy is capped off at Emax.

In the following two cases:

- If remaining energy is forecasted to be < 0 i.e. energy starvation is detected
- If initial energy available is not enough to even execute the task for one computation time.

The processor introduces idle time to recharge the battery for one time period before switching state to WAITING, so that processor can assign an updated schedule based on updated task deadlines (and hence, priorities) and arrival times.

If the available energy is sufficient for executing the job, case EXECUTE JOB commences, where the scheduler executes the ready task τ_i.

```
Algorithm 2: CHECK_ENERGY State
Input: Energy in battery (E),$E_{max}$, $ActiveJob(\tau_i)$, Recharge-rate($P_p$), SlackChecked,
       current time($t_c$)
Output: Residual Energy after Job Completion, $E_{residual}$
1  For each job: Computation time $C_i$, already executed time $t_{Ei}$, total energy $e_i$, already
     consumed energy $E_{ci}$, bool ongoing
2  /* Checking if battery energy is enough to execute task, or if it needs
     recharge                                                              */
3  if $E == 0$ then
4      /* Idle time is needed for recharge; maximum allowable idle time is
         calculated by checking SLACK_TIME                                 */
5      if $SlackChecked == TRUE$    // Checking if SLACK_TIME has been checked since
        t=0
6      then
7          if $SlackCheckedTime == t$          // Check if SLACK_TIME's checked at $t_c$
8          then
9              Recharge by $p_p$ for one time-period and update $t_c$
10         end
11         $state = CHECK\_SLACK\_TIME$                       // Check SLACK_TIME at $t_c$
12         $break$           // exit CHECK_ENERGY once CHECK_SLACK_TIME is called}
13     end
14 end
15 if $\tau_i.ongoing == 0$                // ongoing = 0 means active task has not begun
16 then
17     $E_{residual} = E + (P_p * C_i) - e_i$
18 else
19     /* ongoing $\neq$ 0 means active task has begun                       */
20     $E_{residual} = E + (P_p * (C_i - t_{Ei})) - (e_i - E_{ci})$
21 end
22 if $E_{residual}$ is sufficient then
23     state = EXECUTE_JOB
24 else
25     Recharge for one time period.
26     state = WAITING
27 end
```

3 CHECK_SLACK_TIME

To recall from the previous section, the processor enters the CHECK_SLACK_TIME state from CHECK_ENERGY state When the energy in the battery is completely depleted. It continually loops within this state until battery energy reach Emax or the scheduler cannot be idled further for recharging due to the minimum ST equating to 0.

Slack time (ST) can be defined as the maximum time the scheduler can be idled for recharging and the active task can be delayed without causing deadline miss of other tasks in the taskset. It is calculated via the following steps:

The variable *minSlack*, which stores the shortest ST among all tasks, is updated with the newly calculated ST if this ST < *minSlack*.

Either if energy in battery has reached its max capacity or the calculated Slack time *ST* results to 0, the processor cannot be idled for any further time period and the scheduler enters $WAITING$ state.

Algorithm 3: CHECK_SLACK_TIME State

```
Input: Each task $\tau_i$, each job $J_k$, Release times of the task $r_i$, release time of job is $r_k$ ,
       Partially completed jobs remaining computation time is $C_{k,rem}$ , job not yet
       released has computation time $C_k$ and deadline of job is $d_k$ , deadline of task is $d_i$
       , current time $t_c$
Output: ST for each job of task $\tau_i$
1  for All Jobs in task $\tau_i$ at time $t_c$ do
2  end
3  if $(C_{k,rem}! = 0)$ & $(d_k \leq d_i)$ // If job is incomplete
4  then
5      $C_{tot,rem} + = C_{k,rem}$ end
6      if $(t_c <= r_k$ & $d_k <= d_i)$ // If the job will be released in future ($t_c <= r_k$)
7      then
8          $C_{tot,fut} + = C_k$ end
9          ST$\tau_i$ = $d_i - t_c - C_{tot,rem}$ - $C_{tot,fut}$
```

```
1  /* Dynamic ST is the minimum slack time of all jobs ending after current
      time.                                                              */
2  Dynamic ST$_w$ = $min(all$ ST$\tau_i$ s with $d_i \geq t_c)$
```

Output: ST for each job of task τ_i

4 CHECK_SLACK_ENERGY

The simulator enters this state when high priority task is forecast-ed to be released after the current time t_c, in which case, it will pre-empt the running lower priority job. In this state, SE is calculated for the lower priority job to ensure there is sufficient energy to run a higher priority task in case it arrives in future.

Algorithm 4: CHECK_SLACK_ENERGY State

```
Input: Each task $\tau_i$, each job $J_k$, Release times of the task $r_i$, release time of job is $r_k$ ,
       Instantaneous recharging Pp, deadline of job is $d_k$ , deadline of task is $d_i$ , Energy
       at current time is $E(t_c)$, $e_w$ is total energy consumed by all jobs, $E_k$ is job's energy,
       Preemption consumed by future job, current time $t_c$
Output: Slack energy SE for each job of task $\tau_i$
1 for All Jobs in task $\tau_i$ at time $t_c$ do
2     if ($t_c \leq r_k$
3     & $d_k \leq d_i$) /* If the job will be released in future, Add job's energy $E_k$ to
        $e_w$                                                                        */
4      then
5          e$_w + = E_k$end
6     end
7     SE$_{\tau_i} = E(t_c) + (d_i - t_c) * Pp - e_w$
8
```

```
1 /* Dynamic preemptive SE (PSE) the minimum SE of all tasks releasing after
     current time and with deadline $d_i$ before $d_{max}$.                            */
2 Dynamic PSE$_w = min(all$ SE$\tau_i$ s with $d_{max}$ max$_w(d_i)$
```

5 EXECUTE_JOB

The scheduling of task τ_i is simulated in this state only if

- τ_i is the highest priority task at t_c

  or if it is not the highest priority task,

- SE is positive and residual battery energy E_{res} sufficiently high for τ_i to execute.

THE PROCESS

1. The *ongoing* variable of this task τ_i is set to 1, to acknowledge that this job is about to be executed for at least one time period and is not a fresh task.
2. The executed time t_{Exci} of job τ_i is incremented by 1 each time the state is called, till t_{Exci} is equal to computation time C_i. We store this value to keep track of how many computation units c_i have been executed.
3. If job is incomplete, run EXECUTE JOB again
4. Else, the job is complete, so
   (a) shift deadline and arrival time by one period for next execution
   (b) reset *ongoing* and t_{Exci} to 0

(c) switch to state WAITING to execute next ready task τ_{i+1}.

Algorithm 5: EXECUTE_JOB State

```
Input: Active job's $t_{Exci}, C_i, D_i, t_{arr,i}, P_i$
Output: Desired state
ongoing = 1                                          // Indicates job has started
$t_{Exci} += 1$
/* If job is unfinished at end of cycle                                       */
if $t_{Exci} < C_i$ then
    $State = EXECUTE\_JOB$
end
else
    /* Shift arrival times and deadlines by task period.                      */
    $t_{arr,i} += P_i$
    $D_i += P_i$
    $State = WAITING$
    /* Reset variables to 0                                                   */
    $t_{Exci} = 0$
    $ongoing = 0$
end
```

6 EXECUTE_CYCLE_BY_CYCLE

This state is called from the CHECK_SLACK_ENERGY state, specifically when the slack energy is ≤ 0 while $E = E_{max}$. Recall that CHECK_SLACK_ENERGY is first called when a higher priority task is forecasted to arrive after the current time t_c, such that it will preempt the low priority active task τ_i.

THE PROCESS

In this state, along with the variables *ongoing* and t_{Exci} as defined in the state EXECUTE_JOB, we introduce a new variable $E_{used,i}$ to keep a track on how many energy units e_i have been consumed by a job up to pre-emption.

1. $E_{used,i}$ is the energy consumed by task τ_i after executing some amount of computation time. It is calculated as:

$$E_{used,i} += (e_i / C_i) \tag{9}$$

   where

2. Next, the residual energy in battery after the τ_i is executed for one cycle is calculated as

$$E = E - (e_i / C_i) + P_p \tag{10}$$

3. If job is unfinished at the end of the cycle, it enters the CHECK_SLACK_ENERGY state again to check if

- **SE > 0 & E=E$_{max}$:** it can move to CHECK_ENERGY then EXECUTE_JOB or
- **SE = 0 &** $E = E_{max}$ it needs to continue executing cycle by cycle or
- **SE !=**E_{max}: It needs recharge.

4. Else, it means that the job is complete, so

   (a) shift deadline and arrival time by one period for next execution

   (b) reset *ongoing*, $E_{used,i}$ and t_{Exci} to 0

   (c) switch to state WAITING to execute next ready task τ_{i+1}.

Algorithm 6: EXECUTE_CYCLE_BY_CYCLE State

```
Input: Active job's $t_{Exci}, C_i, D_i, t_{arr,i}, P_i, E_used, i, e_i, P_p$
Output: Desired state
ongoing = 1                                    // Indicates job has started
t$_{Exci}$ += 1
/* Increment energy used by per-cycle-energy consumption of task     */
E$_{used,i}$ += 1
/* If job is unfinished at end of cycle                              */
if $t_{Exci} < C_i$
/* To check values of SE and E                                       */
then
    State = CHECK_SLACK_ENERGY
end
else
    /* Shift arrival times and deadlines by task period.             */
    t$_{arr,i}$ += $P_i$
    D$_i$ += $P_i$
    State = WAITING
    /* Reset variables to 0                                          */
    t$_{Exci}$ = 0
    ongoing = 0
end
```

6
SECTION

Message pipes services

As explained in the previous section, ED-H policy required to have battery information.More particularly, we need the battery capacity and also the energy produced for a given interval in order to compute slack_energy of each task each time the scheduler is called.
However we are using a dual-kernel configuration, I mean Xenomai Real Time tasks are running on the **Cobalt** kernel while non-real time tasks are running on the default Linux kernel.
We will have so :

- Several real-time tasks : tasks scheduled with ED-H policy + the scheduler
- A non real-time task : used to gather battery information from Linux kernel API.

In order for the scheduler to access the battery information from the non real time task, we need to find a way to establish proper communication between real-time task and non real-time task.
Few solutions exist :

- XDDP communication : a native solution using communication pipe, fully supported since Xenomai 2.0
- RT CAN bus or RT USART or RTnet (TCP/IP socket interface) : useful when your Cobalt is on a different board than your linux Kernel
- shared memory

Because XDDP communication is easy to deploy, we will explain and then show how to use this solution in order to retrieve battery information from a RT context.

6.1 XDDP communication : How it works

As explained before, Xenomai runs alongside the default linux kernel. (using an Hardware abstraction layer called **adeos**). In our configuration (I mean a dual Kernel configuration), Xenomai has the higher priority and Linux the lowest. That mean each Xenomai task which use no Linux system calls or APIs within the code have a higher priority (they are in **primary mode** than other tasks using it (these tasks are in **secondary mode**).
On top of that, switching from primary mode to secondary mode could lead to priority inversion problem (a higher priority task being preempted by a task with a lower priority).
So to prevent all this, we need a pipe mechanism called the XDDP. This mechanism can be used directly thanks to the POSIX skin of Xenomai (I mean a subset of POSIX functions for Xenomai only, which include XDDP mechanism).

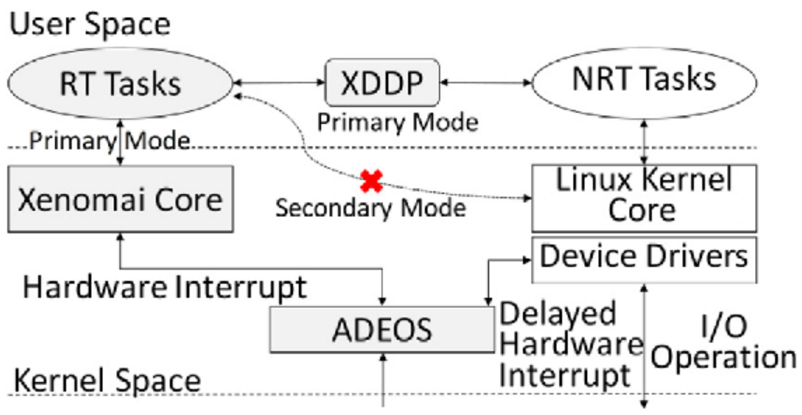


Figure 11: SW Architecture for RT-nRT communication

Basically, the XDDP mechanism use message pipes to communicate between real-time tasks and regular Linux tasks. A communication pipe could only be open by a Xenomai RT tasks.Then a non-RT task could write/read data shared on the pipe by accessing it from the linux standard filesystem (I will explain this later).
The figure below illustrates the RT - non RT communication process using XDDP :

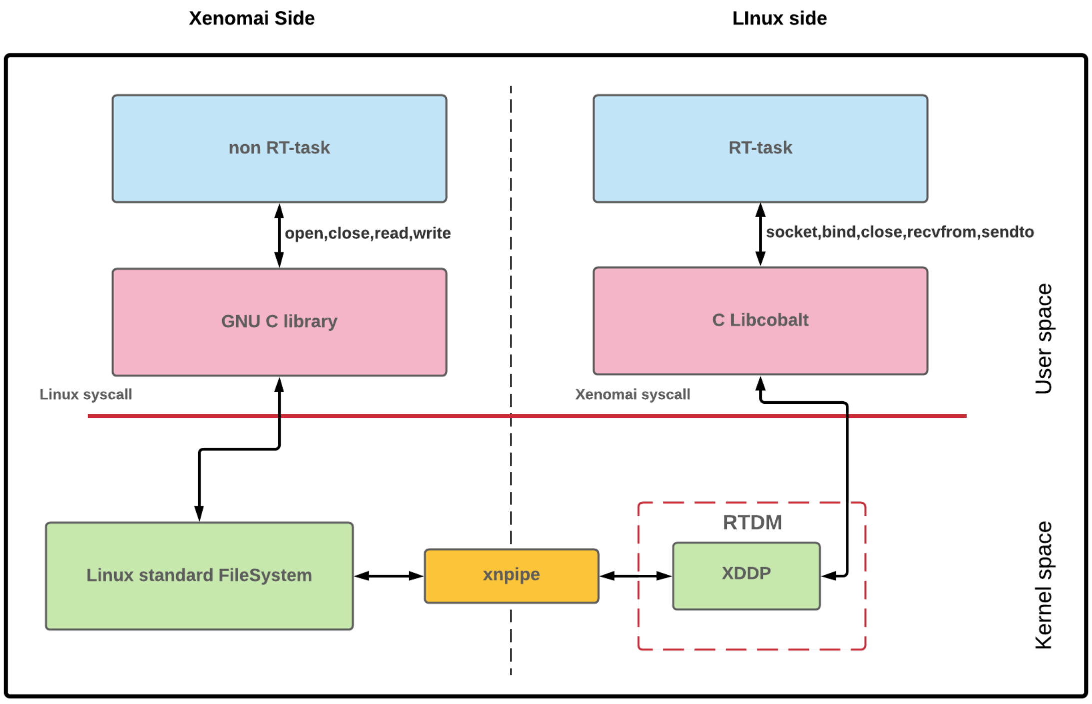
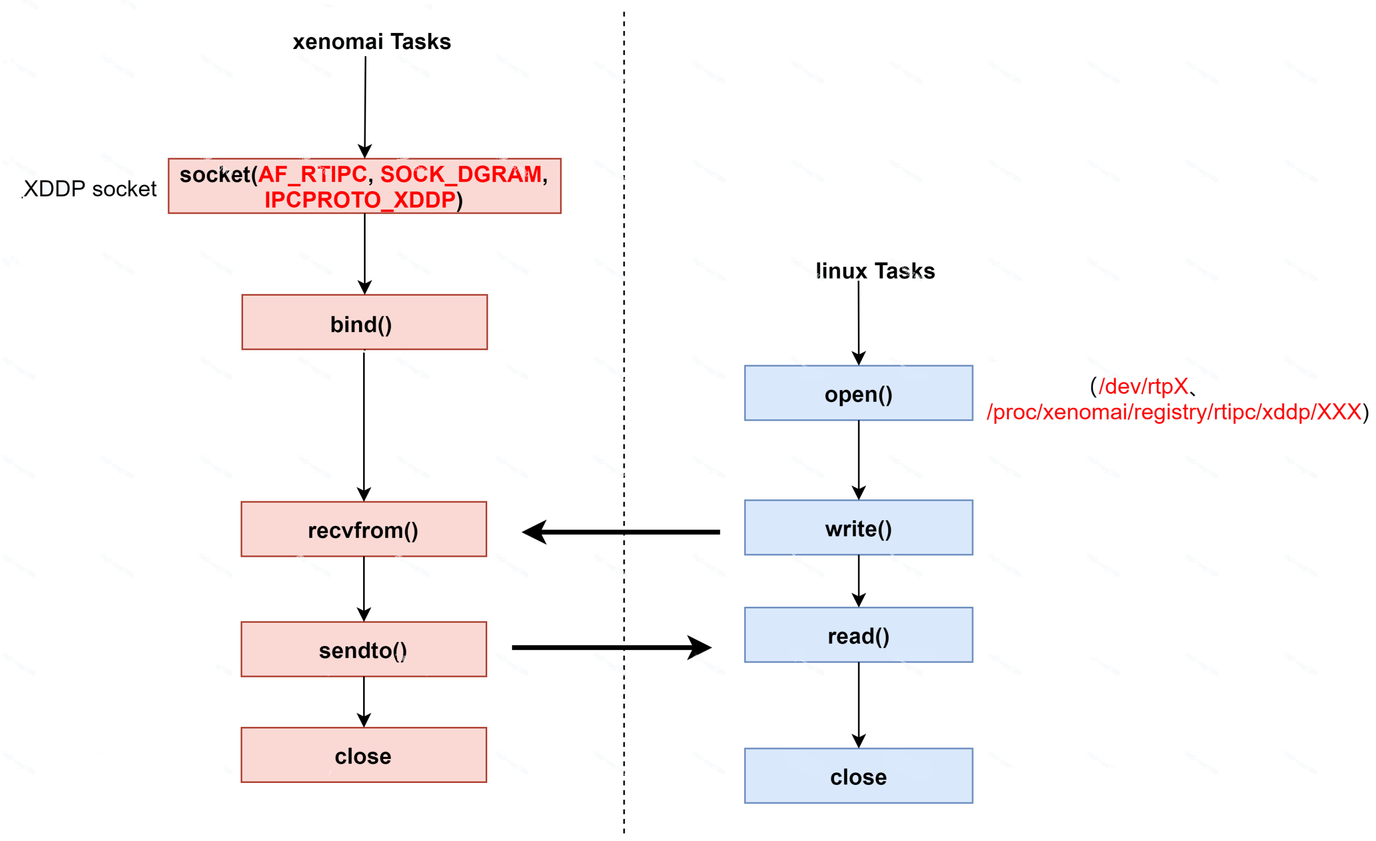


Figure 12: Simplified diagram for XDDP communication

On the Linux domain side, pseudo-device files named /dev/rtp<minor> give regular POSIX tasks access to non real-time communication endpoints, via the linux standard File System service. On the Xenomai domain side, a socket is bound to an XDDP port, which will then act as a proxy to send and receive data to/from the associated pseudo-device file. Keep in mind that XDDP port and minor number are the same and so :

- data sending through a XDDP socket using sendto() function could be received in the linux domain via the standard read() function
- data receiving through a XDDP socket using recvfrom() function have been sent from the linux domain via the standard write() function

Figure below illustrates functions call in linux domain and Xenomai domain :



xenomai Tasks
XDDP socket
socket(AF_RTIPC, SOCK_DGRAM, IPCPROTO_XDDP)
bind()
recvfrom()
sendto()
close
linux Tasks
open()
(/dev/rtpX、/proc/xenomai/registry/rtipc/xddp/XXX)
write()
read()
close

Figure 13: Functions call in both domains

We made a simple example to show you how to bind a socket and read data from RT tasks here: Another similar example, available here, use functions with the RT_PIPE_ prefix. This is simply an API to simplify the whole process (indeed RT_PIPE_CREATE call directly the socket() function for example).

Xenomai also provides example which are available here as well...

6.2 XDDP communication to retrieve battery data

As explained before, we need first a RT task dedicated to create socket and then retrieve information from the battery. Then we need a non-RT task dedicated to send battery information from the linux domain. While the RT task can be directly the one use to schedule all Xenomai tasks (I mean we can retrieve battery information just before selecting the next thread to be executed), we need to create a linux module dedicated to get battery information.

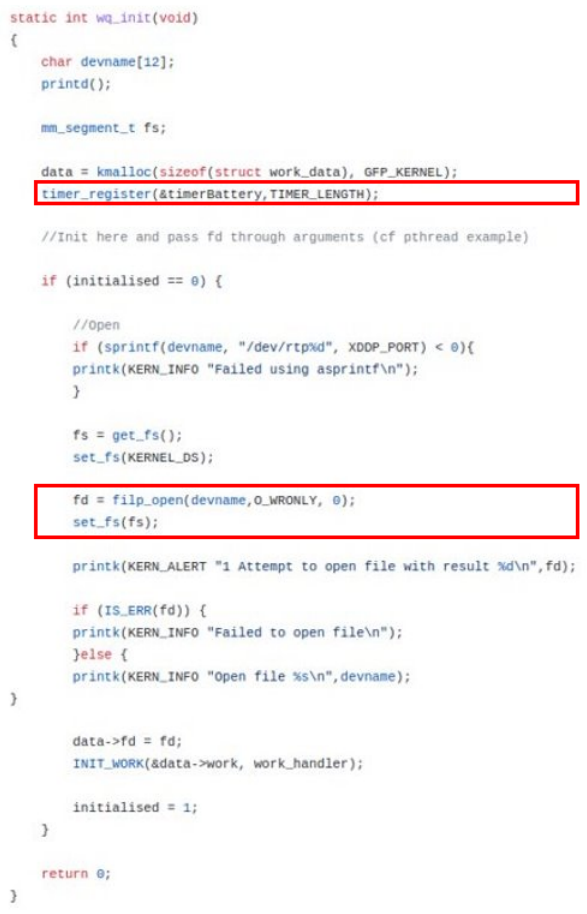
Rather than using a simple application in user space which we can launch each time we need to use, the interest of using a module is :

1. Inserting the module at boot (I mean the module will be automatically started during boot operation).
2. Easy way to load/unload the module dynamically
3. Direct call of kernel space functions

The linux battery module is available here (with README file as well).

In this module, we can observe 3 main parts :

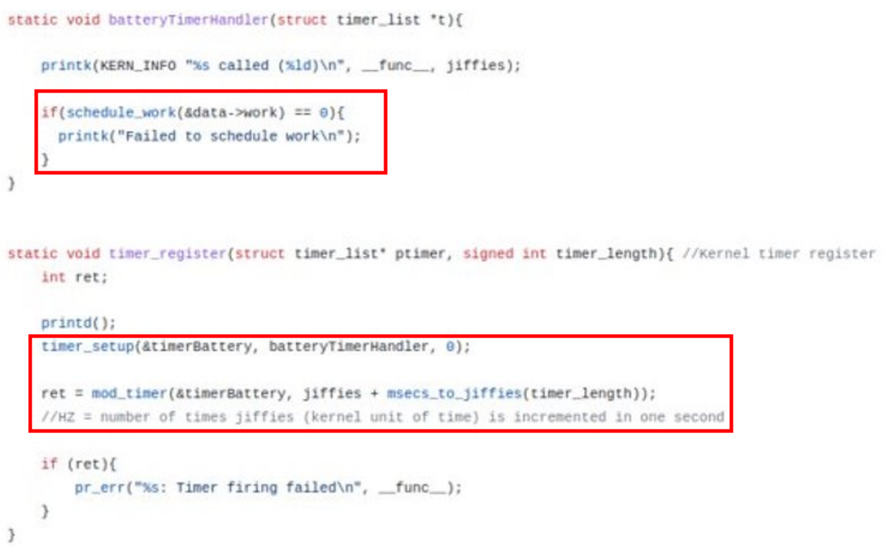
- The init section : Basically, we use the equivalent read() function to access a socket on port 0 (=minor 0) . We also create a timer, of **TIMER_LENGTH** ms duration. Each time the timer expires, a work task dedicated to send battery information through the XDDP port 0 will be called through the timer handler. (cf. here to understand the basis of workqueue and deferred work).



```
static int wq_init(void)
{
    char devname[12];
    printd();

    mm_segment_t fs;

    data = kmalloc(sizeof(struct work_data), GFP_KERNEL);
    timer_register(&timerBattery,TIMER_LENGTH);

    //Init here and pass fd through arguments (cf pthread example)

    if (initialised == 0) {

        //Open
        if (sprintf(devname, "/dev/rtp%d", XDDP_PORT) < 0){
        printk(KERN_INFO "Failed using asprintf\n");
        }

        fs = get_fs();
        set_fs(KERNEL_DS);

        fd = filp_open(devname,O_WRONLY, 0);
        set_fs(fs);

        printk(KERN_ALERT "1 Attempt to open file with result %d\n",fd);

        if (IS_ERR(fd)) {
        printk(KERN_INFO "Failed to open file\n");
        }else {
        printk(KERN_INFO "Open file %s\n",devname);
}

        data->fd = fd;
        INIT_WORK(&data->work, work_handler);

        initialised = 1;
    }

    return 0;
}
```

Figure 14: Init section for battery linux module

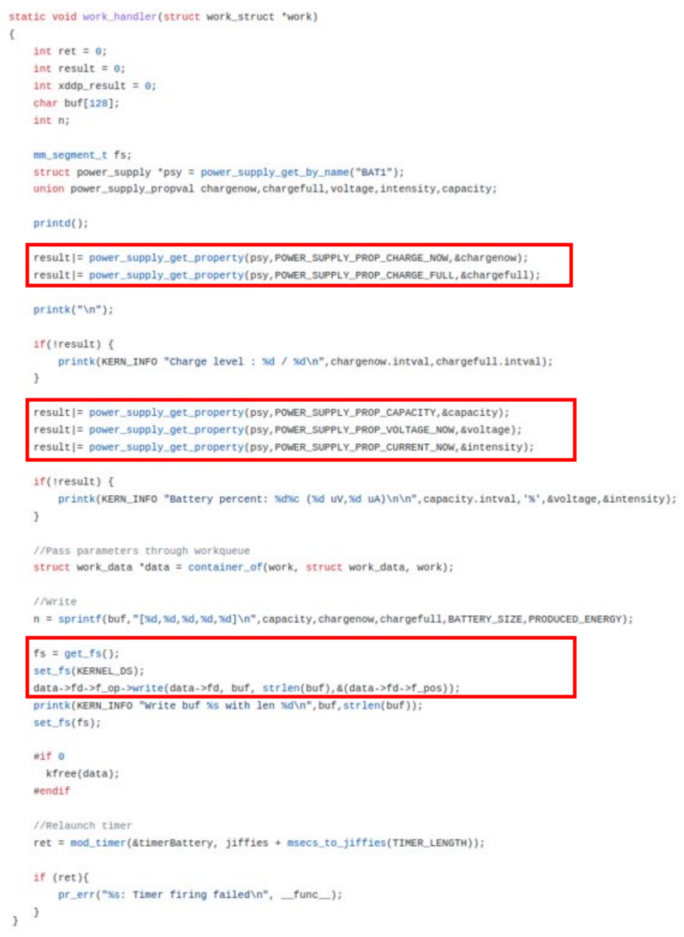
- the timer handler : Each time the timer expires, he will call the work task.



```
static void batteryTimerHandler(struct timer_list *t){

    printk(KERN_INFO "%s called (%ld)\n", __func__, jiffies);

    if(schedule_work(&data->work) == 0){
      printk("Failed to schedule work\n");
    }
}


static void timer_register(struct timer_list* ptimer, signed int timer_length){ //Kernel timer register
    int ret;

    printd();
    timer_setup(&timerBattery, batteryTimerHandler, 0);

    ret = mod_timer(&timerBattery, jiffies + msecs_to_jiffies(timer_length));
    //HZ = number of times jiffies (kernel unit of time) is incremented in one second

    if (ret){
        pr_err("%s: Timer firing failed\n", __func__);
    }
}
```

Figure 15: Tim handler for battery linux module

- the work task : Gather battery information via **<linux/power_supply>** and send it via standard write() function. Once all operations are operated, the work task relaunch the timer.



```
static void work_handler(struct work_struct *work)
{
    int ret = 0;
    int result = 0;
    int xddp_result = 0;
    char buf[128];
    int n;

    mm_segment_t fs;
    struct power_supply *psy = power_supply_get_by_name("BAT1");
    union power_supply_propval chargenow,chargefull,voltage,intensity,capacity;

    printd();

    result|= power_supply_get_property(psy,POWER_SUPPLY_PROP_CHARGE_NOW,&chargenow);
    result|= power_supply_get_property(psy,POWER_SUPPLY_PROP_CHARGE_FULL,&chargefull);

    printk("\n");

    if(!result) {
        printk(KERN_INFO "Charge level : %d / %d\n",chargenow.intval,chargefull.intval);
    }

    result|= power_supply_get_property(psy,POWER_SUPPLY_PROP_CAPACITY,&capacity);
    result|= power_supply_get_property(psy,POWER_SUPPLY_PROP_VOLTAGE_NOW,&voltage);
    result|= power_supply_get_property(psy,POWER_SUPPLY_PROP_CURRENT_NOW,&intensity);

    if(!result) {
        printk(KERN_INFO "Battery percent: %d%c (%d uV,%d uA)\n\n",capacity.intval,'%',&voltage,&intensity);
    }

    //Pass parameters through workqueue
    struct work_data *data = container_of(work, struct work_data, work);

    //Write
    n = sprintf(buf,"[%d,%d,%d,%d,%d]\n",capacity,chargenow,chargefull,BATTERY_SIZE,PRODUCED_ENERGY);

    fs = get_fs();
    set_fs(KERNEL_DS);
    data->fd->f_op->write(data->fd, buf, strlen(buf),&(data->fd->f_pos));
    printk(KERN_INFO "Write buf %s with len %d\n",buf,strlen(buf));
    set_fs(fs);

    #if 0
      kfree(data);
    #endif

    //Relaunch timer
    ret = mod_timer(&timerBattery, jiffies + msecs_to_jiffies(TIMER_LENGTH));

    if (ret){
        pr_err("%s: Timer firing failed\n", __func__);
    }
}
```

Figure 16: Work task handler for battery linux module

We also provide a fake battery module, initially retrieved from this repository : https://github.com/hoelzro/linux-fake-battery-module and whose purpose is to simulate the dynamic behaviour of a battery (Adaptation and more details here)

7

SECTION

EDH over Xenomai

Now let's have a look to the implementation of ED-H over Xenomai. 3 steps are needed in order to modify the Xenomai source code (starting from a working Xenomai with EDF installation):

1. Add socket binding on XDDP port 0 to Xenomai source code to retrieve battery data
2. Modify scheduler algorithm to select next task to run to match ED-H constraints (following the document available here).
3. Modify alchemy API to match ED-H needs.

7.1 XDDP interdomain communication to gather battery data

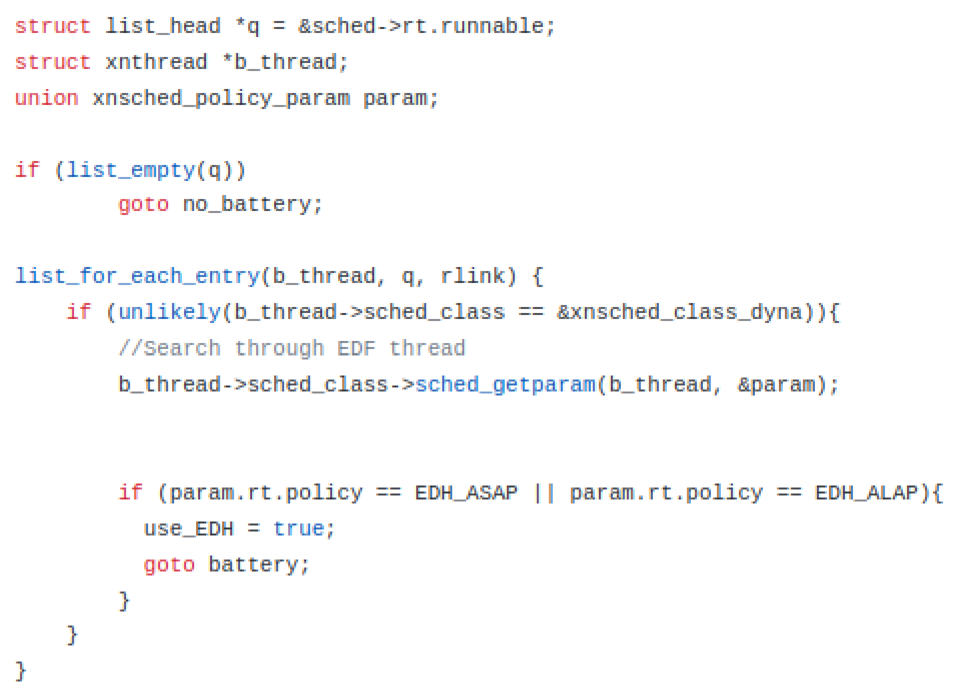
TODO : (Code en cours de validation)

7.2 Scheduler alg.

Let's now talk about how to modify the scheduler algorithm in order to follow ED-H constraints. When we use EDF policy or a round-robin policy, the function **xnsched_pick_next()** available here is responsible to select the next task to be executed (depending on priority or deadlines). If we have at least one task ready to be executed, this task will be executed and the CPU will always be loaded. In case we use the ED-H policy, sometimes we want the system to stay IDLE (we need to keep selecting the IDLE task for a given time) in order for the battery to be filled again.

To do this, we add few modifications to the function **xnsched_pick_next()** :

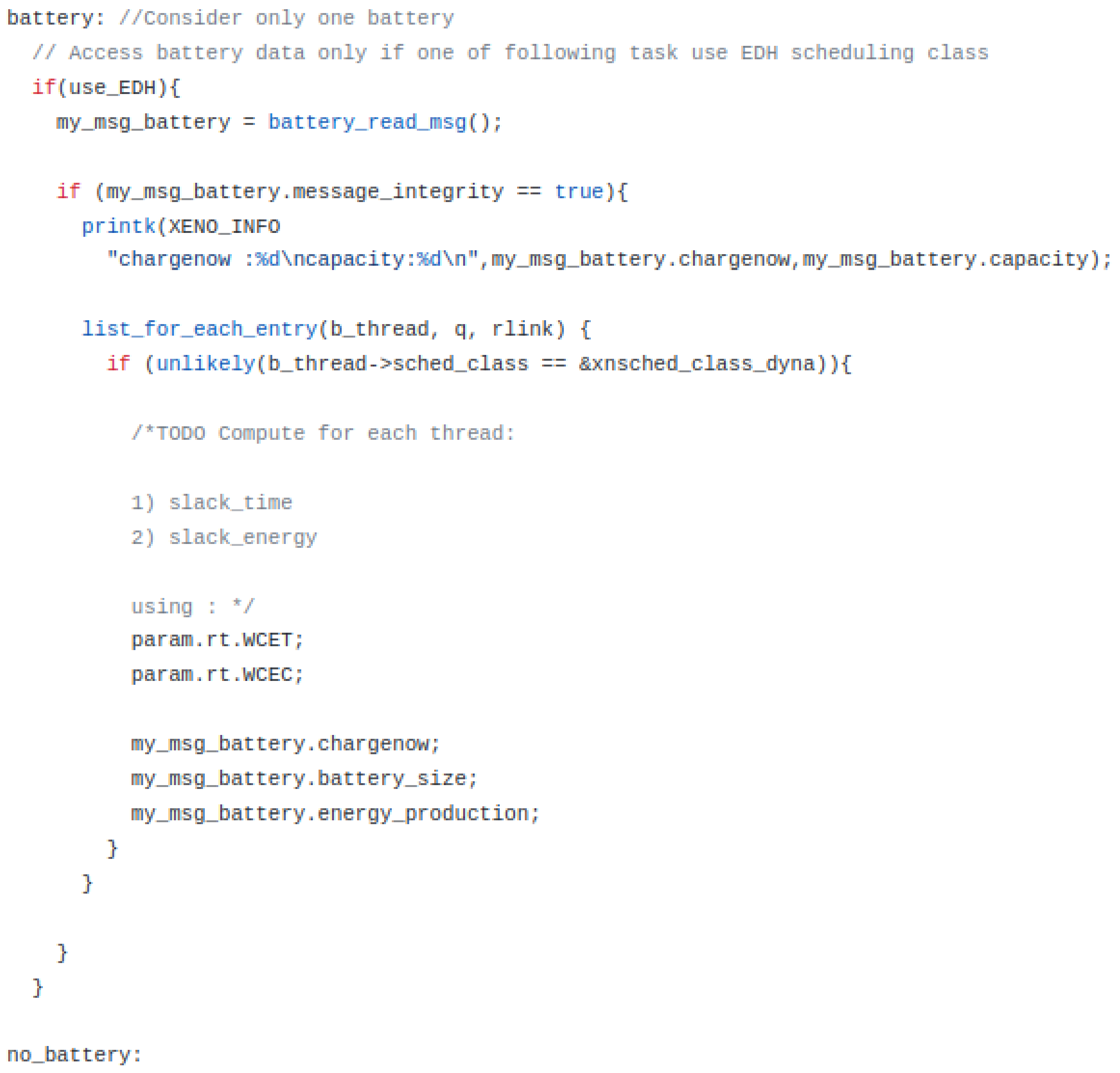
- First we need to check the data **policy** of each task. Among all ready tasks, if we found a task with a **policy** equal to **EDH_ALAP** or **EDH_ASAP**, we need so to schedule the tasks set using ED-H policy.



```
struct list_head *q = &sched->rt.runnable;
struct xnthread *b_thread;
union xnsched_policy_param param;

if (list_empty(q))
        goto no_battery;

list_for_each_entry(b_thread, q, rlink) {
    if (unlikely(b_thread->sched_class == &xnsched_class_dyna)){
        //Search through EDF thread
        b_thread->sched_class->sched_getparam(b_thread, &param);


        if (param.rt.policy == EDH_ASAP || param.rt.policy == EDH_ALAP){
          use_EDH = true;
          goto battery;
        }
    }
}
```

Figure 17: 1st modification in function **xnsched_pick_next()**

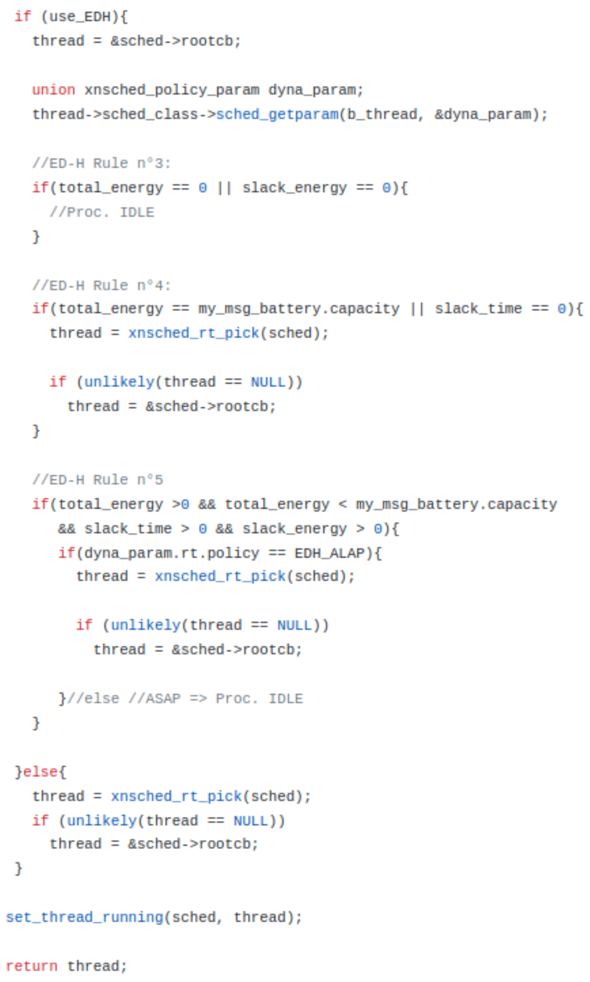
- We retrieve battery data (from XDDP port 0) and we compute then slack_energy and slack_time for each ED-H tasks.



```
battery: //Consider only one battery
  // Access battery data only if one of following task use EDH scheduling class
  if(use_EDH){
    my_msg_battery = battery_read_msg();

    if (my_msg_battery.message_integrity == true){
      printk(XENO_INFO
        "chargenow :%d\ncapacity:%d\n",my_msg_battery.chargenow,my_msg_battery.capacity);

      list_for_each_entry(b_thread, q, rlink) {
        if (unlikely(b_thread->sched_class == &xnsched_class_dyna)){

          /*TODO Compute for each thread:

          1) slack_time
          2) slack_energy

          using : */
          param.rt.WCET;
          param.rt.WCEC;

          my_msg_battery.chargenow;
          my_msg_battery.battery_size;
          my_msg_battery.energy_production;
        }
      }

    }
  }

no_battery:
```

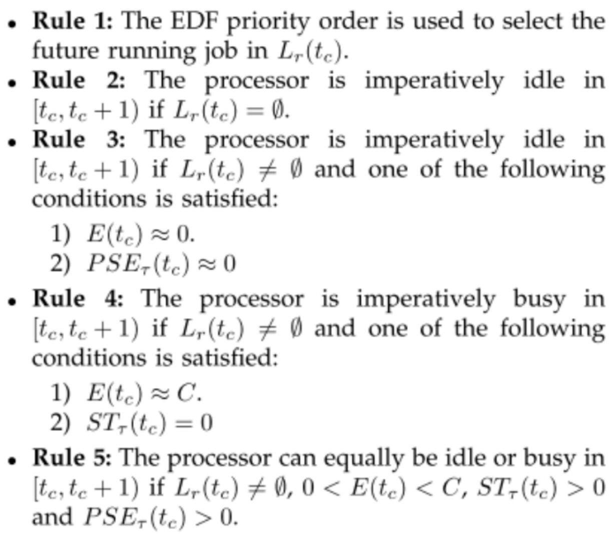
Figure 18: 2nd modification in function **xnsched_pick_next()**

- We follow rules from the figure below. We have 2 possibility : execute a ready task from tasks set (could be the same running tasks) or execute the IDLE task (I mean.



```
if (use_EDH){
  thread = &sched->rootcb;

  union xnsched_policy_param dyna_param;
  thread->sched_class->sched_getparam(b_thread, &dyna_param);

  //ED-H Rule n°3:
  if(total_energy == 0 || slack_energy == 0){
    //Proc. IDLE
  }

  //ED-H Rule n°4:
  if(total_energy == my_msg_battery.capacity || slack_time == 0){
    thread = xnsched_rt_pick(sched);

    if (unlikely(thread == NULL))
      thread = &sched->rootcb;
  }

  //ED-H Rule n°5
  if(total_energy >0 && total_energy < my_msg_battery.capacity
     && slack_time > 0 && slack_energy > 0){
     if(dyna_param.rt.policy == EDH_ALAP){
       thread = xnsched_rt_pick(sched);

       if (unlikely(thread == NULL))
         thread = &sched->rootcb;

     }//else //ASAP => Proc. IDLE
  }

 }else{
  thread = xnsched_rt_pick(sched);
  if (unlikely(thread == NULL))
    thread = &sched->rootcb;
 }

set_thread_running(sched, thread);

return thread;
```

Figure 19: 3rd modification in function **xnsched_pick_next()**



- **Rule 1:** The EDF priority order is used to select the future running job in $L_r(t_c)$.
- **Rule 2:** The processor is imperatively idle in $[t_c, t_c + 1)$ if $L_r(t_c) = \emptyset$.
- **Rule 3:** The processor is imperatively idle in $[t_c, t_c + 1)$ if $L_r(t_c) \neq \emptyset$ and one of the following conditions is satisfied:
  1) $E(t_c) \approx 0$.
  2) $PSE_\tau(t_c) \approx 0$
- **Rule 4:** The processor is imperatively busy in $[t_c, t_c + 1)$ if $L_r(t_c) \neq \emptyset$ and one of the following conditions is satisfied:
  1) $E(t_c) \approx C$.
  2) $ST_\tau(t_c) = 0$
- **Rule 5:** The processor can equally be idle or busy in $[t_c, t_c + 1)$ if $L_r(t_c) \neq \emptyset$, $0 < E(t_c) < C$, $ST_\tau(t_c) > 0$ and $PSE_\tau(t_c) > 0$.

Figure 20: ED-H rules to respect

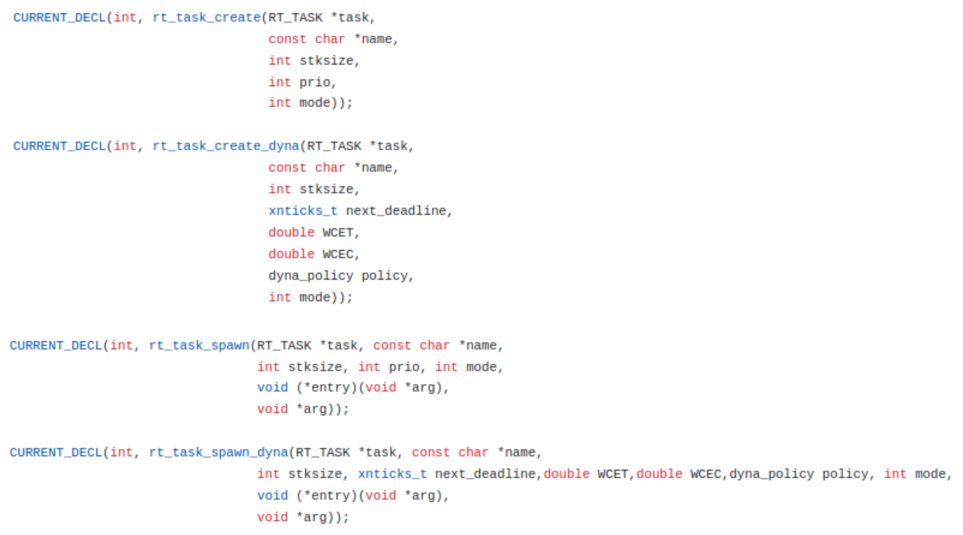
7.3 Alchemy API modifications

Initially, the alchemy API allow us to create RT task using round-robin scheduler. For this purpose, one of the parameters of the function **rt_task_create** is the priority. For EDF and ED-H we don't need it anymore. On top of this, we need more parameters.

We need so for each task:

- EDF policy : the **deadline** of the task
- ED-H policy : **WCET**,**WCEC** of the task (battery information are directly provided from the XDDP port).

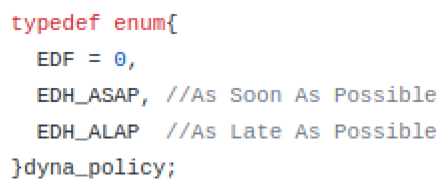
  We add so another function, called **rt_task_dyna**, in order to create EDF/ED-H task. (same idea for **rt_task_spawn_dyna**). This function owns all parameters mentioned above.



```
CURRENT_DECL(int, rt_task_create(RT_TASK *task,
                                 const char *name,
                                 int stksize,
                                 int prio,
                                 int mode));

CURRENT_DECL(int, rt_task_create_dyna(RT_TASK *task,
                                 const char *name,
                                 int stksize,
                                 xnticks_t next_deadline,
                                 double WCET,
                                 double WCEC,
                                 dyna_policy policy,
                                 int mode));

CURRENT_DECL(int, rt_task_spawn(RT_TASK *task, const char *name,
                               int stksize, int prio, int mode,
                               void (*entry)(void *arg),
                               void *arg));

CURRENT_DECL(int, rt_task_spawn_dyna(RT_TASK *task, const char *name,
                               int stksize, xnticks_t next_deadline,double WCET,double WCEC,dyna_policy policy, int mode,
                               void (*entry)(void *arg),
                               void *arg));
```

Figure 21: Alchemy API for task creation in file task.h

Lastly, in order to know if we want to schedule tasks using ED-H or EDF, we add an extra parameter, policy. The structure is presented below :



```
typedef enum{
  EDF = 0,
  EDH_ASAP, //As Soon As Possible
  EDH_ALAP  //As Late As Possible
}dyna_policy;
```

Figure 22: Structure dyna_policy in file uapi/sched.h

Without entering into details of implementation, don't hesitate to look at this commit for example : commit link Looking at this commit, you can see which files are needed to be modified in order to adapt Xenomai source code and more particularly the alchemy API.

7.4 Remaining tasks and amelioration/criticism

TODO: Evoquer:

- Section de code à compléter pour calculer les 2 paramètres suivants : **Slack_time** et **Slack_energy** (corps de fonctions prêt, prêt à coder...)
- Utilisation de Cheddar pour validation EDF
- Utilisation du **fake battery module** pour simplifier les tests

8
SECTION

Planning & project evolution

8.1 TODO

9

SECTION

References

List of Figures

List of Tables