Implementing Flexible Embedded Control on a Simple Real-Time Multitasking Kernel

Ricardo Marau, Pedro Leite, Luis Almeida, Manel Velasco, Pau Martí, and Josep M. Fuertes

Abstract

In recent years, approaches to control performance and resource optimization for embedded control systems have been receiving increased attention. Most of them focus on theory, whereas practical aspects are omitted. Theoretical advances demand flexible real-time kernel support for multi-tasking and pre-emption, thus requiring more sophisticated and expensive software/hardware solutions. On the other hand, embedded control systems often have cost constraints related with mass production and strong industrial competition, thus demanding low cost solutions.

In this paper it is shown that these conflicting demands can be softened and that a compromise solution can be reached. First, a simple multi-tasking pre-emptive real-time kernel targeting low cost microprocessors is presented. Second, the implementation on the kernel of recent research results on optimal resource management for control tasks is reported. The experimental evaluation shows that significant control performance improvement can be achieved without increasing hardware costs.

I. Introduction

In recent years, approaches to control performance and resource optimization for embedded control systems have been receiving increased attention. As outlined in [1], these efforts are consequence of the demands created by applications that impose several resource constraints in terms of memory, processing capacity, battery, etc.

Most of those approaches and solution focus on theory, whereas practical aspects are omitted. At the processor level, theoretical advances demand flexible real-time kernel support for multi-tasking and pre-emption. For example, a recurrent problem is how to assign optimal sampling periods to control tasks such as aggregated control performance is improved within the available resources (e.g., [2], [3], [4], [5] or [6]). At the network level, theoretical advances demand flexible support for synchronous and asynchronous traffic. For example, a recurrent problem is how to combine time-triggered and event-triggered communication in a timeliness fashion (e.g., [7], [8] or [9]).

Contradictorily to the initial problem targeted by these solutions, that is, minimize resource requirements to meet tight cost constraints related with mass production and strong industrial competition, research advances seem to require more sophisticated and expensive software/hardware solutions.

In this paper it is shown that this contradiction is fictitious, and that careful implementation of recent research results can be achieved by low cost solutions without compromising the potential benefits offered by the theory.

Looking at the solutions to optimal sampling period selection for controllers, it has been shown that the most appropriated period to be assigned to each control task depends on the state of the controlled plants. This result suggests that implementing control tasks using the traditional static cyclic executive approach [10] can not provide the best possible control performance. Cyclic executives do not support dynamic task processing rates.

The announced benefits of those solutions can be achieved by a computing platform with real-time kernel support for dynamic resource management (e.g., see [11], [12], or [13] for rate adaptation within available processor capacity, or [14] for an energy aware rate adaptation approach). However, the existing real-time kernels or operating systems enhanced with rate adaptation are in general not suitable for simple microprocessor architectures due to resource limitations. And simple existing kernels targeting small architectures (e.g., [15], [16] [17], [18], [19], [20]) do not

This work was partially supported by NoE ARTIST2 IST-2004-004527.

R. Marau and L. Almeida are with the Electronics, Telecommunications and Informatics Department, University of Aveiro, Aveiro, Portugal (e–mail: marau@det.ua.pt; lda@det.ua.pt).

Pedro Leite is with RiaMoldes, Aveiro, Portugal

M. Velasco, P. Martí, and J. M. Fuertes are with the Automatic Control Department, Technical University of Catalonia, Barcelona, Spain (e–mail: manel.velasco@upc.edu; pau.marti@upc.edu; josep.m.fuertes@upc.edu).

provide support for task rate adaptation or their application to adaptive embedded control applications has not been reported. In any case, they are complementary to our work.

The contribution of the paper is twofold. First, a simple multi-tasking pre-emptive real-time kernel targeting low cost microprocessors is presented. Second, the implementation on the kernel of recent research results on optimal resource management for control tasks is reported. The experimental evaluation shows that significant control performance improvement can be achieved without increasing hardware costs.

II. Preliminaries

In this section the result on optimal sampling period selection for real-time control tasks presented in [2] is reviewed, and the required kernel support for its implementation analyzed. Afterward it is analyzed whether a cyclic executive approach can meet the demanded kernel support. From the analysis, preliminary guidelines for designing a simple but effective adaptive real-time kernel are derived.

A. Optimal sampling period selection

As outlined in the introduction, recent works have addressed this problem. Here the focus is in the solution presented by [2], but the discussion also will cover other results. In [2] it was proved that adapting the rate of progress of each control task via changing tasks' periods according to each plant dynamics improved overall control performance.

The embedded control system model considered in this work is the following. A set of n control tasks have to execute on a single processor. Each control task (or controller) and plant constitute a control loop. Each controller performs sampling, control algorithm computation and actuation sequentially at each task execution. Within a range of sampling periods, each controller fulfills the control specifications.

The key assumption is that controllers can not all simultaneously run at their highest possible sampling frequency, and so cannot provide the best possible control performance (equivalent to the one they would provide if they were running in isolation). Therefore, processor is the scarce resource that must be appropriately allocated among the control tasks.

The problem to be solved is how to assign the scarce processor capacity, i.e. how to assign sampling periods, to the set of control loops such that all tasks are schedulable and overall control performance is maximized, knowing that the controllers will provide better performance given more processor, or alternatively, when given shorter periods.

In [2] is it shown that after a minimum processor share given by the longest sampling period is guaranteed to each control task, the optimal resource allocation policy can be formulated as a constrained optimization problem whose solution dictates that the remaining available processor, here refereed as *slack*, should be assigned to the control loop with largest error, where the error is defined as a function of the plant state.

Observation 1. *In terms of tasks periods, the solution states that all controllers will run at their longest sampling periods (slowest frequencies) except for the controller whose plant is experiencing the biggest error, which will run at its shortest sampling period (highest frequency).*

Observation 2. *The application of the optimal policy requires the implementation of controllers capable of running with different sampling frequencies given different resource allocations (for further details on controller design and stability analysis, see [2] and references therein).*

B. Cyclic executive

The traditional cyclic executive (also known as interrupt control loop) has been traditionally the skeleton to build embedded real-time control systems. Its name is derived from the way that it cycles through predetermined paths in the code in a highly predictable manner. It is a conceptually simple architecture, with minimal "executive" code, minimal system overhead, requiring only a single regular (periodic) interrupt, and allowing for very reliable testing.

Figure 1 illustrates the operation of a cyclic executive. After performing the system initialization, which includes start-up processing and enabling interrupts, the software enters an infinite loop. Each pass through the loop (after the first) is initiated by the periodic interrupt. During the cycle, any required inputs are read, processing on those

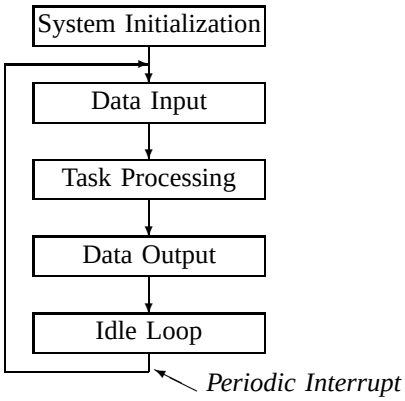

Fig. 1. Cyclic executive operation

inputs accomplished, and the resulting outputs generated. At that point, the software enters an idle loop waiting for the interrupt to initiate the next cycle.

A first limitation of the cyclic executive approach is that it can handle simple systems where all processing is done at the rate established by the periodic interrupt. With several tasks, this means that all tasks must run at a rate that is a harmonic of the periodic interrupt. Therefore, cyclic executives do not support arbitrary task processing rates. Because cyclic executives are non-preemptive, a second limitation is that the software engineer must divide the lower rate processing into sometimes unnatural "tasks" that can be run each cycle because preemption is not allowed.

These limitations prevent implementing the optimal resource allocation policy summarized in observation 1. The optimal policy states that controllers rate of progress has to be changed via changing task periods at run-time according to the controlled plant dynamics and available slack. This demand conflicts with the first limitation imposed by cyclic executives. In addition, since tasks rates depend on plant errors, preemption will permit quick task periods' changes in response to perturbation arrivals, thus providing more responsiveness. This desirable behavior can not be met without preemption, thus impairing the cyclic executive approach.

C. Architecture model and design decisions

The optimal policy can be implemented at the kernel level or at the task level. The first approach implies specific code in the kernel that may not be used in other scenarios. In the second approach, similar to the feedback scheduling approach presented in [4] or [6], a dedicated task performs the optimal resource allocation. In this paper, the second approach is chosen because it is less kernel intrusive.

In addition, it is important to point out that the implementation requires specific data exchange between kernel and control tasks. Its correct operation depend on two decision variables: available slack and plant states. Slack is an information that belongs to the kernel space and plant states is an information that belongs to the applications space. Therefore, the optimal policy also demands mechanisms and support for passing/accessing the decision variables or related information, which is analyzed next. In terms of kernel or operating systems support, a reflective architecture for real-time systems [21] is required.

A reflective system can react in a flexible manner to changing dynamics within the system itself as well as the environment. Reflection is a mechanism by which a program becomes "self-aware", checks its progress and is capable of adapting itself or its own behavior. This is achieved by allowing applications to access kernel data structures to obtain and modify information about the current system state.

Figure 2 illustrates the reflective architecture. A set of n tasks dedicated to controlling a set of plants is considered. Each i^{th} task controls a plant, constituting a control loop. The control is performed by means of the control signal u_i, that is computed by the task considering the plant state vector x_i and the current sampling period h_i. Solid arrows illustrate control operations within each control loop. The dedicated task in charge of performing the slack redistribution and allocation is also illustrated.

The interface between the RT kernel and tasks, i.e., reflective information, is used to exchange data between kernel and tasks, illustrated with dashed arrows. The dedicated task needs to know the available slack U_s, which is

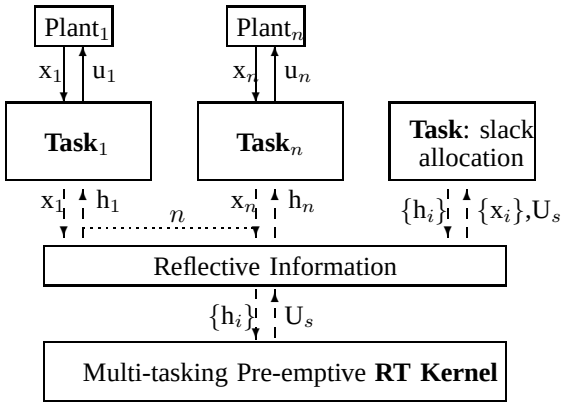

Fig. 2. Conceptual reflective kernel architecture

made accessible by the RT Kernel, as well as, each plant state vector, x_i, which is made accessible by all control tasks. With both informations, the new rate of progress for all tasks, in the form of new sampling periods h_i, is calculated. Each newly calculated h_i is made available to the kernel as well as to the control tasks.

In summary, the optimal policy demands a pre-emptive multitasking real-time kernel based on a reflective architecture capable of 1) providing the required flexibility to accommodate task rate changes, and 2) facilitating communication mechanisms between kernel and tasks for information exchange.

III. Kernel description

The hardware platform used in the experiments is based on the Microchip PIC18FXX8 micro-controllers family[1]. These are 28/40-Pin microcontrollers, enhanced with flash, and with CAN (Controller Area Network) communications. The latter interface allows building distributed real-time embedded systems without compromising timeliness, as analyzed for example in [22] or [23].

In order to provide the required preemptive multi-tasking capabilities a simple kernel was developed for the referred platform. A complete description of the kernel can be found in [24]. For the purpose of this work, two other requirements are important, namely the capability to adapt the tasks periods on-line and the support for the exchange of reflective information between tasks and kernel (and vice-versa).

A. Properties

The kernel was named RTKPIC18 and fully written in the C language, compiled using PICC-18 v8.20PL4 of HI-TECH Software. It supports task creation, periodic activation with offset control, specification of deadlines equal or shorter than the period and tasks temporal policing. The time management is carried out using a periodic *tick* that can be configured from $2ms$ to $65534ms$. Periods, offsets and deadlines and expressed as integer multiples of the tick duration.

The scheduling policy can be specified at system startup. Three policies are already predefined, namely Earliest Deadline First (EDF), Rate-Monotonic (RM) and Deadline-Monotonic (DM) [25]. Other scheduling policies require the addition of the corresponding function to the kernel. Beyond the scheduling policy, it is also possible to specify the preemption mode in which the system will operate, i.e., whether preemption will be allowed on not.

The system supports up to 13 periodic tasks plus a background task. The respective code takes 2900*words* out of a total of 32K*words* available, i.e. about 8.8% of the program memroy. Concerning the more constrained data memory, the kernal takes 39 bytes plus 31 bytes per task from a total of 4K*bytes* of RAM, representing a footprint that varies from 101 bytes (2.5%) for 1 periodic task plus background to 473 bytes (11.6%) for the maximum number of tasks.

Moreover, to minimize the memory footprint and computational overhead, the kernel was provided with the minimum functionality to meet the referred requirements. In particular, task synchronization features were reduced to simple preemption control with a global flag and task communication was reduced to the use of global variables.

[1]http://www.microchip.com/

B. Programming model

The structure of the *main* function in an application is shown in Figure 3. Initially the program must call the *create_system* system call to specify the duration of the tick in miliseconds, the scheduling policy (EDF, RM or DM) as well as the preemption mode (PREEMPT or NOPREEMPT). This system call also creates and initializes all the internal structures of the kernel, still holding the time management off.

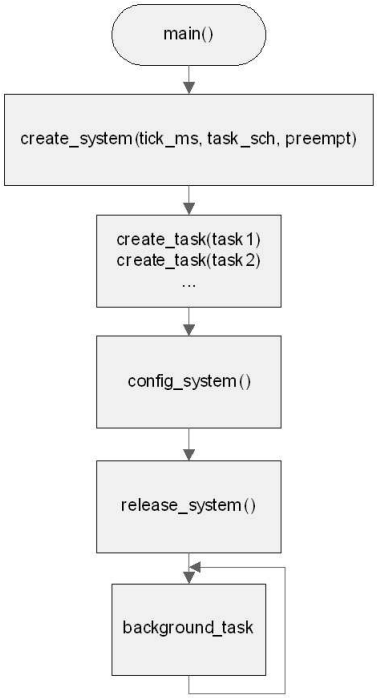

Fig. 3. Structure of the *main* function

The following step corresponds to the actual creation of the tasks, one by one, using the *create_task* system call, which associates the code of each task to a *task control block* (tcb) internal to the kernel and to a block of memory to save the task context, allows specifying the task timing attributes and executes the initialization code inside the task, leaving it ready for the periodic execution. Then, the *config_system* system call must be invoked to initialize the priorities of the tasks according to the scheduling policy chosen.

Finally, the actual start of operation of the whole application, synchronously, is carried out by involking the *release_system* system call, which initiates the tick handling activity and the timining management inside the kernel. All task offsets are counted with respect to this moment in time. Then the system enters an infinite loop executing the background task, which is preempted for the execution of the periodic tasks, even in non-preemptive mode.

The programming model of each periodic task is shown in Figure 4. It follows the common model with an initialization part followed by an infinite loop that corresponds to the recurrent execution of the periodic instances. The initialization part includes the initialization of the task local variables and finishes with a call to the *task_init* system call, which initializes the task context (*tcpuctx*) and prepares the task for the periodic execution that will start at this point, i.e., right after the call to *task_init*. The infinite loop contains the code that will be executed in each periodic instance and must terminate by invoking the *end_cycle* system call, which transfers the execution control to the kernel so that another ready task, if any, can be dispatched.

C. Implementation details

The RTKPIC18 kernel was developed following the guidelines provided in [26]. It carries out two main activities, i.e., tasks and time management. The former makes use of an array of task control blocks that contain, each, the timing attributes of the respective task, a pointer to the associated code, its current execution status and its context

```
void task_example(void)
{
/* Declaration of task dynamic variables */
    /* Initialization of task variables */
    (...)
    task_init();
    /* task_create() executes the task code up to the
       end of the task_init() system call, thus
       the actual periodic execution starts from here */
    while(1)
    {
        /* Code of each periodic instance */
        (...)
        end_cycle();
    }
}
```

Fig. 4. Programming model of periodic tasks

that includes all task dynamic variables and current CPU registers, particularly *program counter* and *status register*. The execution status can be one of three, IDLE (interval of time between the end of the previous instance and the beginning of the next), READY (upon periodic activation and waiting for CPU time) and RUN (executing). The state machine is shown in Figure 5.

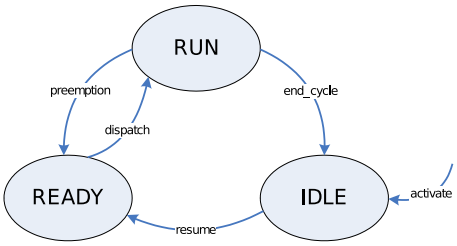

Fig. 5. Tasks state machine

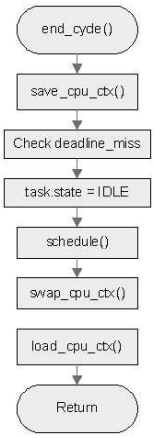

Fig. 6. Internals of the *end_cycle* system call

When preemption is disabled, a task with higher priority, which becomes ready while another one with lower priority is executing, must wait for the termination of the latter, thus becoming blocked. This is normally represented by a specific *blocked* state. However, in our case we avoid this extra state keeping the tasks in such circumstances in the READY state and simply not allowing preemption to take place. Tasks will execute by priority order as soon as the CPU becomes free or preemption is re-enabled. On the other hand, when preemption is enabled, it takes place synchronously with the ticks since it is the tick handler that causes the periodic tasks activation, i.e., transition from IDLE to READY states. The rescheduling points, i.e., where the scheduling function is invoked

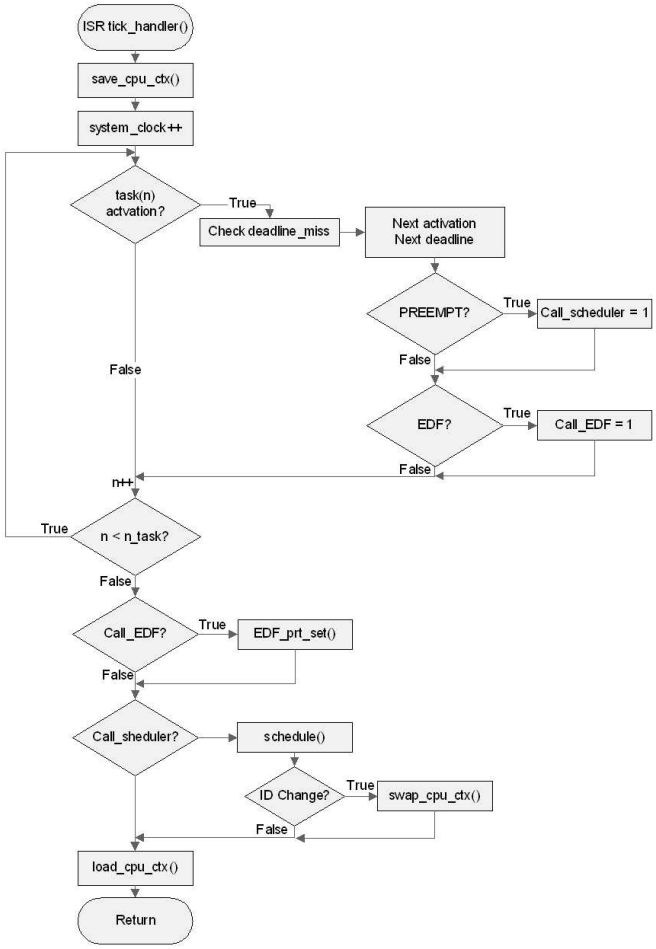

Fig. 7. The structure of the *tick_handler*

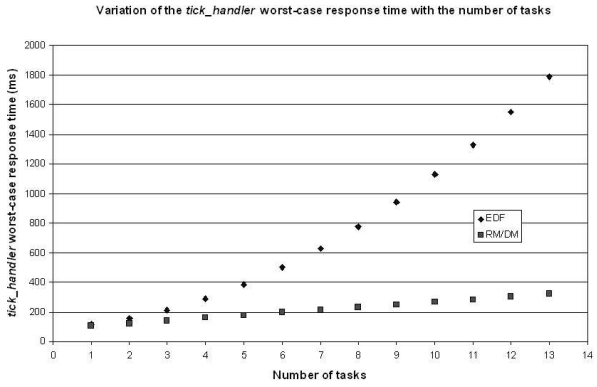

Fig. 8. *Tick_handler* worst-case execution time

to retrieve the highest priority task in the READY state, are the ticks handling, as just mentioned, and the tasks terminations within the *end_cycle* system call (Fig. 6).

One interesting aspect concerns the implementation of the tasks contexts. These data structures are frequently implemented in one or multiple *stacks*. However, the PIC18FXX8 microcontroller has only a very limited stack of a few bytes to store the return addresses from routines. To work around this limitation, the compiler allocates the memory space for the dynamic variables statically in RAM, when linking the code. Such memory space is

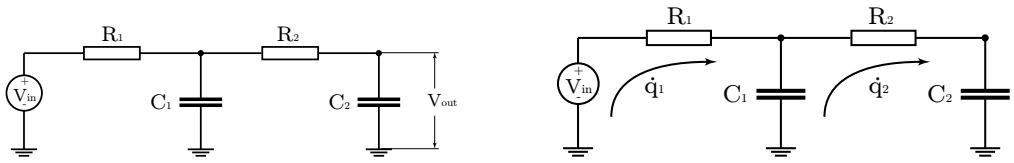

Fig. 9. Electronic circuit scheme (left) and its dynamics (right)

overlapped for all independent functions and exclusive for functions that call one another. In our case, because of preemption, is it important that all contexts are separate, which is achieved invoking the *recursive_call* system call in the *main*.

The other major role played by the kernel is time management. This is carried out, essentially, within the tick handler. Figure 7 shows the flowchart of this handler that starts by accounting for the passage of time, then scanning for periodic activations and finally rescheduling and dispatching a new task if needed. The detection of missed deadlines just sets a flag in the task context that allows it to take reflective measures.

The graphic in Figure 8 gives the worst-case execution time of the *tick_handler* for the EDF and RM/DM schedulers, which, in both cases, happens when all tasks are released simultaneously. When using EDF, the dynamic priorities are computed on-line using the *EDF_prt_set* function, which causes a penalty in the execution time. The measurements were carried out on a platform based on the 18F258 processor operating at 20MHz.

D. The reflective layer

The reflective layer shown in Figure 2 is implemented with an adequate shared structure that contains the required information not included in the kernel, namely the control plants state x_i. By means of this structure, such information is made available to the task that redistributes the slack available in the system which, using the current system slack obtained from the kernel (U_s), sets the adequate sampling periods for the control tasks h_i.

In order to operate on this structure, a few functions were developed and made available to the tasks, namely *mutex_write* and *mutex_read*, which write and read information from the reflective layer in a non-preemptive way, respectively.

The reflective information that is maintained by the kernel is directly accessed with the associated primitives, as shown earlier in the text, namely *get_CPU_util*, which returns the current CPU utilization and is used to compute the current system slack, *get_sys_time*, *get_my_id*, *get_deadl_stat*, *get_period* and *set_period*. In particular, notice that the changes in the tasks periods are enforced on the next periodic activation of the respective tasks, only, which prevents the potential occurrence of transient overloads during the period switching.

IV. Experimental results

This section shows several practical experiments to evaluate the implementation of the optimal policy.

A. Controlled plants

The control tasks implemented on the kernel are in charge of controlling voltage stabilizers, in the form of $RCRC$ circuits (Fig. 9). The electronic components are $R_1 = 330K\Omega$, $C_1 = 100nF$, $R_2 = 330K\Omega$ and $C_2 = 100nF$. The electronic circuit are modeled in terms of the currents, as illustrated in 9 (right), by equations

$$\dot{q}_1 R_1 + (q_1 - q_2)\frac{1}{C_1} = V_{in} \quad (1)$$

$$\dot{q}_2 R_2 + (q_2 - q_1)\frac{1}{C_1} + q_2\frac{1}{C_2} = 0. \quad (2)$$

Taking into account the components values, a state-space form is given by

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -918.27 & -90.90 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 918.27 \end{bmatrix} u \quad (3)$$

$$y = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad (4)$$

where x_1 corresponds to the voltage in C_2, and x_2 is $\dot{q}_2$. For each voltage stabilizer, the control objective is to maintain stable the output voltage V_{out} of C_2 according to a reference value, regardless of the load perturbations. In our setup, we a reference value of $\bar{u} = 2.4v$. Therefore, the desired equilibrium point $x_{desired}$ for each control loop is $\bar{x}_1 = 2.4v$ and $\bar{x}_2 = 0A$ at the circuit output. Figure 10 illustratres the feedback scheme.

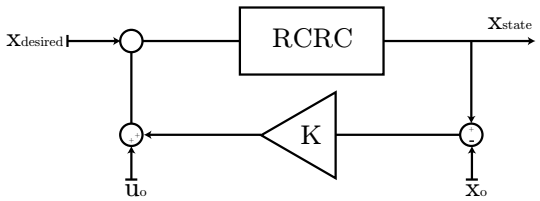

Fig. 10. Feedback control loop scheme

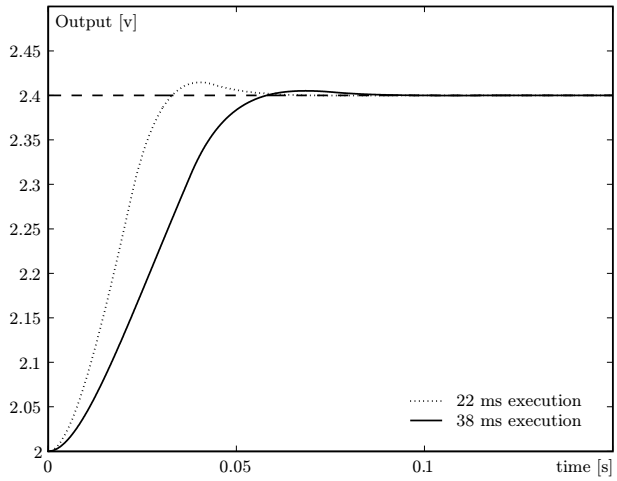

Fig. 11. Systems dynamics for both controller rates

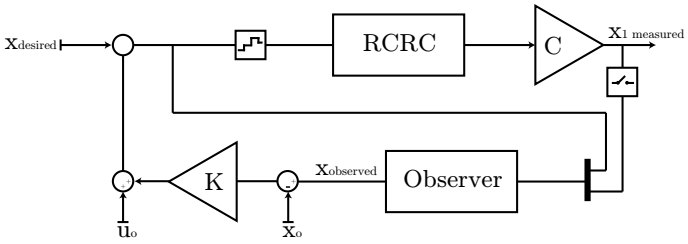

Fig. 12. Feedback scheme with observers

B. Controller design and implementation

Optimal LQ control [27] has been the basis for controller design. The quadratic cost function to be minimized is given by

$$J = \int_0^\infty \left(x^T Q x + u^T R u\right) dt \tag{5}$$

where

$$Q = \begin{bmatrix} 400 & 1 \\ 1 & 0 \end{bmatrix} \tag{6}$$

$$R = 1 \tag{7}$$

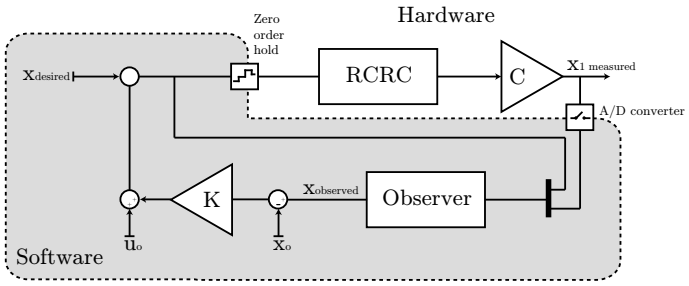

Fig. 13. Hardware and software

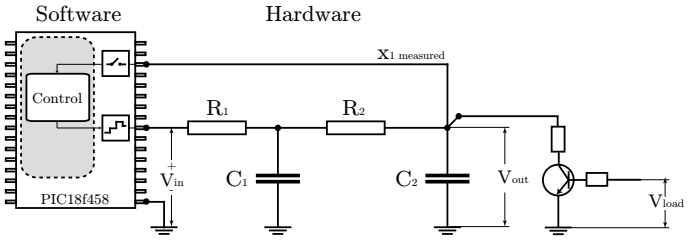

Fig. 14. Physical system

That is, the focus of the controller is to ensure fast tracking of the output variable given the desired voltage output. This is achieved by penalizing errors between the output variable and the desired output variable.

In terms of pole location, the previous specification is equivalent to localize the continuous closed-loop poles at $-103.93 + j87.1$ and $-103.93 - j87.1$. The corresponding discrete closed-loop poles depend on the control task period (i.e., sampling period).

The optimal policy mandates to design two control gains, corresponding to the maximum and minimum rate. In the implementation, this is specified by $22 \cdot 10^{-3}s$ and $38 \cdot 10^{-3}s$. With these periods, the two gains are $k_{22} = [4.274\ \ 0.048]$ and $k_{38} = [2.068\ \ 0.025]$. The expected dynamics for the plant with these two gains are illustrated in Figure 11.

In addition to the controller gains, deadbeat reduced observers for estimating the second state variable have been designed for the two task rates, as illustrated in Figure 12. The matrices defining the observer dynamics are

$$\begin{aligned} \begin{bmatrix} \hat{x}_1(k+1) \\ \hat{x}_2(k+1) \end{bmatrix}_{22ms} &= \begin{bmatrix} -0.045 & 0.009 \\ -0.225 & 0.045 \end{bmatrix} \begin{bmatrix} \hat{x_1}(k) \\ \hat{x_2}(k) \end{bmatrix} + \begin{bmatrix} 0.138 & 0.906 \\ 8.246 & -8.020 \end{bmatrix} \begin{bmatrix} u(k) \\ x_1(k) \end{bmatrix} \\ \begin{bmatrix} x_1(k) \\ x_2(k) \end{bmatrix}_{Observed} &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \hat{x}_1(k) \\ \hat{x}_2(k) \end{bmatrix} \end{aligned} \tag{8}$$

and

$$\begin{aligned} \begin{bmatrix} \hat{x}_1(k+1) \\ \hat{x}_2(k+1) \end{bmatrix}_{38ms} &= \begin{bmatrix} 0.052 & 0.008 \\ -0.314 & -0.052 \end{bmatrix} \begin{bmatrix} \hat{x_1}(k) \\ \hat{x_2}(k) \end{bmatrix} + \begin{bmatrix} 0.254 & 0.693 \\ 8.064 & -7.749 \end{bmatrix} \begin{bmatrix} u(k) \\ x_1(k) \end{bmatrix} \\ \begin{bmatrix} x_1(k) \\ x_2(k) \end{bmatrix}_{Observed} &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \hat{x}_1(k) \\ \hat{x}_2(k) \end{bmatrix} \end{aligned} \tag{9}$$

The hybrid model of the implementation is shown in Figure 13. The shadow part inside of the dotted line corresponds to the software algorithm to be executed in the microcontroller. This illustration of the feedback control loop corresponds to the physical setup illustrated in Figure 14 that includes the source of perturbations affecting the system.

The simplified version of the pseudo-code of the control task is given next:

```
/***************** Control task *****************/

void task1(void) {

/*------------ Definitinos -------------------------*/

    K1,K2: Controller gains for fast and slow periods;
    Ob1, Ob2: Observers for fast and slow periods;
    ref: set point reference;
    fast_period: constant;

/*------------ Start Up ----------------------------*/

    task_init();        //Real-Time Command
    t1=get_my_id();
    while(1)
    {
        inc = READPORTI1;    //Read Input

/*---- Select the Right Control and Observer ------*/

        if(get_period()==fast_period)
        {
            /* use right observer and right gain */
            ob = ob1;
            K=K1;
        }
        else
        {
            ob = ob2;
            K=K2;
        }
        /* Observer computation */
        x_obs = f_ob(ob,x_obs_old, u_old,inc);
        /* Control Computation  */
        u = f_ctrl(x_obs, ref, inc, K);
        WRITEPORTO1(u);         //Write Output

/*----- Update States -----------------------------*/

        x_obs_old = x_obs;     //Observer
        u_old = u;             //Output
        mutex_write(t1,f(x_obs)); //Write reflective inf.

        end_cycle();

    }
}
```

The pseudo-code starts by defining variables, constants, etc. Only the most significants are shown: two gains $K1$ and $K2$, observer matrices for the two reduced observers $Ob1$ and $Ob2$, the reference set point, and constant "fast_period" that is used to differentiate between the two possible rates. The reference in our experiments is constant, but it could be changed within the software. After this, the task is initialized.

Then, an infinite loop contains the main code for the task. First, the output variable is read and its value stored in inc. With the value and the reference (generated by software), and after checking the current period with *get_period*, the appropriate control signal u is computed, taking into account the estimation given by the observer. Index $t1$ is the task identifier obtained with *get_my_id*. This control signal is written to the output port, state updates are performed, and the system call *end_cycle* notifies the kernel that the control task has finished execution of a periodic instance.

Note that in the state updates, the error of the controlled plant is written in the reflective memory shared between tasks and kernel via *mutex_write*. This value will be read by the task performing the slack allocation. The error is a quadratic function of the circuit states.

The actual code of the task also prevents saturation on the control signal. A more complete code including observers and saturation is listed next:

```
/******************* User tasks definition *******************/
// Tasks are defined as functions, there is always a WHILE(1) bucle
// the suspension of the bucle is performed using end_cycle()
// primitive

void task1(void) {

    char inc;                       // For reading AD

/*-----------Controllers----------------------------------*/

    int k[2]={400 , 4};             //Controller Gain for fast period
    int k2[2]={206 , 2};            //Controller Gain for slow period

/*-----------Observers------------------------------------*/

    float F[2][2] = {
                {-0.0720,    0.0089},
                {-0.5846,    0.0720}
            };                          //Observer Dynamics for fast period

    float F2[2][2] = {
                {   0.0526,    0.0088},
                {  -0.3149,   -0.0526}
            };                          //Observer Dynamics for slow period


    float G[2][2] = {
                { 0.1222,    0.9498},
                { 8.1394,   -7.5548}
            };                          //Observer Gain for fast period


    float G2[2][2] = {
                    {0.2542,    0.6932},
                    {8.0645,   -7.7495}
                };                      //Observer Gain for slow period

/*------- Start Up----------*/
    float x[2] = {0,0};
    float x_[2] = {0,0};
    float ua=0;
    char v=0;

    task_init();        //Real-Time Command

    while(1)
    {

        inc = READPORTI1; //Read AD

        /* Select the Right Control  and Observer Gains*/

        if(tcb1[t1].period==period_fast)
        {
            /* Observer computation */

            x[0] = F[0][0] * x_[0] + F[0][1] * x_[1]+ G[0][0]*ua+G[0][1]*inc;
            x[1] = F[1][0] * x_[0] + F[1][1] * x_[1]+ G[1][0]*ua+G[1][1]*inc;

            /* Control Computation  */

            v=(-(inc-consigna[0])*k[0])/100-(x[1]*k[1])/100+  consigna[0];

            /* Saturate Output */

            if ((-(inc-consigna[0])*k[0])/100-(x[1]*k[1])/100+consigna[0]>127)v=127;
            if ((-(inc-consigna[0])*k[0])/100-(x[1]*k[1])/100+consigna[0]<0)v=0;
        }
        else
        {
            /* Observer computation */

            x[0] = F2[0][0] * x_[0] + F2[0][1] * x_[1]+ G2[0][0]*ua+G2[0][1]*inc;
            x[1] = F2[1][0] * x_[0] + F2[1][1] * x_[1]+ G2[1][0]*ua+G2[1][1]*inc;
```

```
            /* Saturate Output */

            v=(-(inc-consigna[0])*k2[0])/100 -(x[1]*k2[1])/100 +consigna[0];

            /* Saturate Output */

            if ((-(inc-consigna[0])*k2[0])/100-(x[1]*k2[1])/100+consigna[0]>127)v=127;
            if ((-(inc-consigna[0])*k2[0])/100-(x[1]*k2[1])/100+consigna[0]<0)v=0;
        }

        /* Write Output*/

        WRITEPORTO1(v);

        /* UPDATE STATES */

        x_[0]=x[0];     //Observed state 1
        x_[1]=x[1];     //Observed state 2
        ua=v;           //Output
        end_cycle();

    }
}
```

C. Experimental setup

To assess the optimal policy and its correctness operation in the kernel, a prototype implementation of the kernel was built for the PIC18F458. Two voltage stabilizer circuits are plugged to the chip pins. The scheme for the complete hardware and software setup is illustrated in Figure 15.

The kernel concurrently executes two control tasks, as well as the task in charge of performing the slack allocation, which is not illustrated in this figure. All tasks are scheduled using EDF scheduling algorithm. The two control tasks, labeled *Control 1* and *Control 2* execute the code explained in section IV-B to control each stabilizer. The control signal generated by each control tasks is sent to the circuit, as illustrated by V_{in}. The sampled controlled variable is V_{out}, as illustrated by $x_{1measured}$. The circuit is perturbed by the load voltage, as illustrated by V_{load}.

Note that in the experimental set-up, the available slack is constant. That is, the minimum guaranteed rate for each task is $38ms$. But both tasks cannot simultaneously run at their maximum rate given by $22ms$. However, the available slack is enough to guarantee to run one task with its maximum rate while the other runs at its minimum rate.

The dedicated task doing the slack allocation reads the error value ($f(x_{obs})$) stored in the reflective memory by each control task. With these values, and taking into account the available slack, sets the two tasks periods as mandated by the optimal policy. Taking into account constant slack, in this particular implementation, the task implementing the optimal policy assigns the shortest sampling period $22ms$ to the task with greatest error while the other task is given $38ms$. The new tasks periods are updated in each task control block. The dedicated task executes at slower rate than the control tasks. In our experimentation, its period is set to $200ms$. It is out of the scope of this paper to analyze which optimal period should have the task doing slack allocation. The shortest its period, the more reactive will be the system, that is, the better the performance. However, this would also imply more overhead due to its execution.

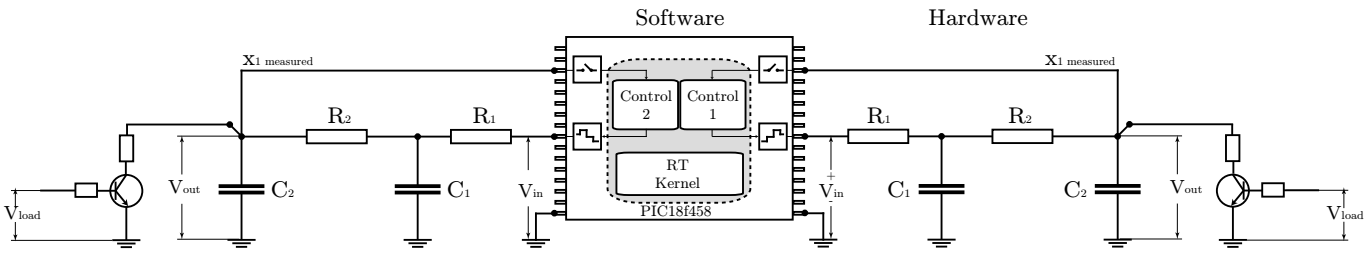

Fig. 15. Hardware and software of the experimental setup

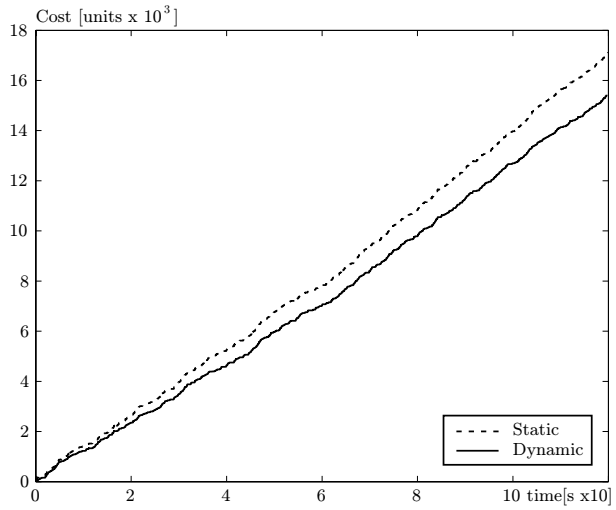

Fig. 16. Static vs. dynamic

Both circuits are subject to randomly generated load perturbations, following an uniform distribution. However, in average, their periodicity is $200ms$. This facilitates the performance analysis.

D. Results

To assess the benefits of the optimal policy, a baseline static policy has been also implemented. In the static policy both controllers run at a fixed rate, and no slack allocation is performed. One controller runs at fixed $38ms$ and the other at fixed $22ms$. Therefore, they implement the traditional controller, that is, a constant controller gain, which coincides with the optimal gain designed in section IV-B.

Figure 16 shows the results in terms of accumulated cost, i.e. control error, of both policies, the static and the dynamic (optimal). The accumulated cost is the evaluation of the cost function (5) over the experimentation time. The lower the curve, the better the performance. Both systems were running for two minutes.

As it can be seen in the Figure 16, the scenario with slack allocation (dynamic) outperforms the static by more than 10%. Note that the resource utilization in both runs are the same. In the dynamic, the two control tasks cannot execute at the highest rate, so they alternate between 22 and $38ms$, giving an average for each task of $30ms$. In the static, both tasks always execute at a fixed rate, with average of $30ms$.

V. CONCLUSIONS

Recently, approaches to control performance and resource optimization for embedded control systems have been receiving substantial attention. Nevertheless, such approaches have essentially focused on theory, without sufficient concern with practical aspects. Several of the theoretical advances require the support of a software infrastructure such as real-time kernels with support for multi-tasking and pre-emption, thus requiring more sophisticated and expensive software/hardware solutions. This requirement conflicts with the frequently severe cost constraints that embedded control systems are subject to, for example related with mass production and strong industrial competition.

In this paper, however, we showed that using a minimal kernel just supporting task and time management with reflectivity allows achieving the desired goal of supporting modern and efficient control techniques, particularly with rate adaptation and multiple closed-loops integration on low cost microcontrollers, namely those of the PIC18Fxx8 family.

The paper presents the referred kernel, the RTKPIC18, which supports preemptive and non-preemptive multitasking, automatic management of periodic task releases, EDF, RM and DM task scheduling, and access to reflective information to support application adaptation. Then, the paper discusses a case study that illustrates the integration of multiple control loops with on-line rate adaptation, how they can be implemented using the referred kernel, and the benefits of rate adaptation by means of practical experiments and quantitative results. Further work will focus on improving specific performance aspects of the kernel, namely reducing the worst-case execution time of the

EDF scheduler, and on analyzing how to tune the period of the adaptation task for an optimal balance between the reactivity of the adaptation, the overhead imposed and the overall performance gains.

REFERENCES

[1] G. Buttazzo, "Research trends in real-time computing for embedded systems," *SIGBED Rev.*, vol. 3, no. 3, pp. 1–10, 2006.

[2] P. Martí, C. Lin, S. Brandt, M. Velasco, and J. M. Fuertes, "Optimal state feedback based resource allocation for resource-constrained control tasks," in *25th IEEE Real-Time Systems Symposium*, Dec. 2004.

[3] A. Antunes, P. Pedreiras, and A. M. Mota, "Adapting the sampling period of a real-time adaptive distributed controller to the bus load," in *10th IEEE Conference on Emerging Technologies and Factory Automation*, Sept. 2005.

[4] D. Henriksson and A. Cervin, "Optimal on-line sampling period assignment for real-time control tasks based on plant state information," in *44th IEEE Conference on Decision and Control and European Control Conference ECC 2005*, Seville, Spain, Dec. 2005.

[5] M.-M. Ben Gaid, A. Çela, Y. Hamam, and C. Ionete, "Optimal scheduling of control tasks with state feedback resource allocation," in *In Proceedings of the 2006 American Control Conference*, Minneapolis, Minnesota, USA, June 2006.

[6] R. Castañé, P. Martí, M. Velasco, A. Cervin, and D. Henriksson, "Resource management for control tasks based on the transient dynamics of closed-loop systems," in *18th Euromicro Conference on Real-Time Systems*, Dresden, Germany, July 2006.

[7] P. Pedreiras and L. Almeida, "The flexible time-triggered (FTT) paradigm: An approach to qos management in distributed real-time systems," in *International Parallel and Distributed Processing Symposium (IPDPS'03)*, Apr. 2003.

[8] G. Cena and A. Valenzano, "On the properties of the flexible time division multiple access technique," *IEEE Transactions on Industrial Informatics*, vol. 2, no. 2, pp. 86 – 94, May 2006.

[9] J. Ferreira, L. Almeida, J. A. Fonseca, P. Pedreiras, E. Martins, G. Rodríguez-Navas, J. Rigo, and J. Proenza, "Combining operational flexibility and dependability in ftt-can," *IEEE Transactions on Industrial Informatics*, vol. 2, no. 2, pp. 95 – 102, May 2006.

[10] C. D. Locke, "Software architecture for hard real-time applications: cyclic executives vs. fixed priority executives," *Real-Time Systems*, vol. 4, no. 1, pp. 37–53, 1992.

[11] G. C. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni, "Elastic scheduling for flexible workload management," *IEEE Transactions on Computers*, vol. 51, no. 3, pp. 289–302, Mar. 2002.

[12] S. A. Brandt, S. Banachowski, C. Lin, and T. Bisson, "Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes," in *24th IEEE Real-Time Systems Symposium*, Dec. 2003, pp. 396–407.

[13] S. Goddard and L. Xu, "A variable rate execution model," in *16th Euromicro Conference on Real-Time Systems*, July 2004, pp. 135–143.

[14] M. Marinoni and G. Buttazzo, "Elastic dvs management in processors with discrete voltage/frequency modes," *IEEE Transactions on Industrial Informatics*, vol. 3, no. 1, pp. 51–62, 2007.

[15] J. A. Stankovic and K. Ramamritham, "The spring kernel: a new paradigm for real-time operating systems," *SIGOPS Oper. Syst. Rev.*, vol. 23, no. 3, pp. 54–71, 1989.

[16] K. M. Zuberi, P. Pillai, and K. G. Shin, "Emeralds: a small-memory real-time microkernel," in *7th ACM symposium on Operating Systems Principles*, 1999, pp. 277–299.

[17] P. Gai, G. Lipari, L. Abeni, M. di Natale, and E. Bini, "Architecture for a portable open source real-time kernel environment," in *Proceedings of the Second Real-Time Linux Workshop and Hand's on Real-Time Linux Tutorial*, November 2000.

[18] E. Mumolo, M. Nolich, and M. Noser, "A hard real-time kernel for motorola microcontrollers," in *23rd International Conference on Information Technology Interfaces*, June 2001.

[19] P. Gai, L. Abeni, M. Giorgi, and G. Buttazzo, "A new kernel approach for modular real-time systems development," in *13th IEEE Euromicro Conference on Real-Time Systems*, June 2001.

[20] D. Henriksson and A. Cervin, "Multirate feedback control using the TinyRealTime kernel," in *Proceedings of the 19th International Symposium on Computer and Information Sciences*, Antalya, Turkey, Oct. 2004.

[21] J. Stankovic and K. Ramamritham, *A Reflective Architecture for Real-Time Operating Systems. Chapter in Advances in Real-Time Systems*. Prentice–Hall, 1995.

[22] S. Cavalieri, "Meeting real-time constraints in can," *IEEE Transactions on Industrial Informatics*, vol. 1, no. 2, pp. 124–135, 2006.

[23] R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien, "Controller area network (can) schedulability analysis: Refuted, revisited and revised," *Real-Time Syst.*, vol. 35, no. 3, pp. 239–272, 2007.

[24] P. Leite, R. Marau, and L. Almeida, "Rtkpic18 - real-time kernel pic18fxx8." Electronics and Telecommunications Department, University of Aveiro, Portugal" Technical Report, 2004. [Online]. Available: http://sweet.ua.pt/ lda/str/kerpic18.zip

[25] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the Association for Computing Machinery*, vol. 20, no. 1, pp. 46–61, Jan. 1973.

[26] G. Buttazzo, *Hard Real-Time Compuing Sysytems: Predictable Scheduling Algorithms and Applications*, 2nd ed. Springer, 2005.

[27] K. J. Åström and B. Wittenmark, *Computer-Controlled Systems*. Prentice Hall, Jan. 1997.