Performing Flexible Control on Low-Cost Microcontrollers Using a Minimal Real-Time Kernel

Ricardo Marau, Pedro Leite, Manel Velasco, Pau Martí, *Member, IEEE*, Luis Almeida, *Member, IEEE*, Paulo Pedreiras, *Member, IEEE*, and Josep M. Fuertes, *Member, IEEE*

***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 multitasking and preemption, 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. We advocate that recent research results on optimal resource management for control tasks can be implemented on simple multitasking preemptive real-time kernels targeting low-cost microprocessors, which can be easily built in-house and tailored to actual application needs. The experimental evaluation shows that significant control performance improvement can be achieved without increasing hardware costs.

***Index Terms*—Adaptive resource management, control systems, embedded systems, microcontroller, real-time kernel.**

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 lifetime, etc.

Most of those approaches focus on theory, whereas practical aspects are omitted. At the processor level, theoretical advances demand flexible real-time kernel support for multitasking and preemption. For example, a recurrent problem is how to assign optimal sampling periods to control tasks such that aggregated control performance is improved within the available resources (e.g., [2]–[4] or [5]). When the systems are distributed, those theoretical advances also demand flexible support for synchronous traffic in the network, for example, to enforce the optimal sampling periods across distributed control transactions or generally to optimize the aggregated quality of service provided to the application (e.g., [6], [7], or [8]).

Conversely 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 approaches.

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 appropriate period to be assigned to each control task depends on the state of the controlled plant. This result suggests that implementing control tasks using the traditional static cyclic approach will not provide the best possible control performance. In fact, the cyclic executive paradigm does not support dynamic task activation 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 [9], [10], or [11] for rate adaptation within available processor capacity, or [12] for an energy aware rate adaptation approach). However, the available real-time kernels or operating systems enhanced with rate adaptation are in general not suitable for simple microprocessor architectures due to resource limitations. Also, simple existing kernels targeting small architectures (e.g., [13]–[18]) do not 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.

Moreover, simple multitasking kernels are relatively easy to implement. In fact, many graduate courses on real-time embedded systems include building them as part of the respective practical assignments. Such kernels are valuable software tools that can be integrated into the applications and tailored to provide just the required services, avoiding excessive overhead and unnecessary complexity.

In this work we use a simple homemade multitasking preemptive real-time kernel targeting a low-cost microprocessor to show the practicality of recent research results on optimal resource management for control tasks. The experimental evaluation shows that a significant control performance improvement can be achieved without increasing hardware costs.

The paper is organized as follows. The next section revisits the sampling control period selection proposed in [2] and discusses its implementation, presenting then the architecture model and design choices underlying to this work. Then,

Manuscript received May 21, 2007; revised January 14, 2008. This work was supported in part by NoE ARTIST2 IST-FP6-2004-004527, NoE ARTISTDESIGN IST-FP7-2008-214373, and in part by Spanish Ministerio de Educación y Ciencia Project ref. CICYT DPI2007-61527. Paper no. TII-08-01-0007.

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

P. 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).

Digital Object Identifier 10.1109/TII.2008.923787

Section III describes the kernel used to support the implementation of the optimal control period policy. Section IV contains the main contribution of this paper showing the practicality of that policy and exhibiting the achievable control performance benefits even with simple resource constrained systems. Finally, Section V presents the conclusions.

II. Background

A. Optimal Sampling Period Selection

As outlined in the previous section, recent works have addressed this problem. Here the focus is on the solution presented by [2], but the discussion will also cover other results. In [2] it was proved that adapting the activation rate of each control task 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 invocation. 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 because of resource constraints, e.g., insufficient CPU bandwidth. Thus, it is not possible to provide the best control performance (equivalent to the one they would provide if they were running in isolation). Therefore, the problem is how to allocate resources to the control tasks so that the overall control performance is maximized.

In particular, we will focus on the computing resources and thus on how to assign CPU capacity to the set of control loops, adapting the respective sampling periods, such that all tasks are schedulable and overall control performance is maximized, knowing that the controllers will provide better performance given more processor share, i.e., when given shorter periods.

The optimal resource allocation policy can be formulated as a constrained optimization problem [2] in which each control task has a minimum guaranteed processor share, i.e., an upper bounded sampling period, and the remaining available processor capacity, herein referred to as *slack*, is assigned to the control loop with largest error, where the error is defined as a function of the plant state. Note that our definition of slack refers to the computing resources that were not statically assigned to any task. Concerning the optimal resource allocation policy we can make the following observations.

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. Architecture Model and Design Decisions

In order to deploy the optimal policy referred above, we can take advantage of a multitasking real-time kernel to provide isolation and modularity among control tasks as well as the timeliness guarantees needed to make sure that the controllers perform as expected. However, a stronger requirement to use such an infrastructure is related to the need for rate adaptation support, which is the core of the optimal policy and cannot be easily achieved with the monolithic cyclic executive approach typically used in embedded control systems.

The optimal policy can be implemented in the kernel or as a task. 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 [3] or [5], a dedicated task performs the optimal resource allocation. In this paper, the second approach is chosen because it is less kernel intrusive and thus more general.

In addition, it is important to point out that the implementation requires specific data exchange between kernel and control tasks. Its correct operation depends 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 requires mechanisms for passing/accessing the decision variables or related information, which are analyzed next. In terms of kernel support, a reflective architecture for real-time systems [19] 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.

Fig. 1 illustrates the reflective architecture. A set of n tasks dedicated to controlling a set of plants is considered. Each ith 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 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 activation period for all tasks h_i is calculated and made available to the kernel as well as to the control tasks.

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

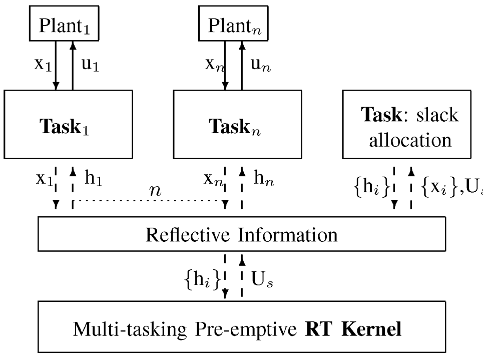


Fig. 1. Conceptual reflective kernel architecture.

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 controller area network (CAN) communications.

In order to provide the required real-time capabilities a simple preemptive multitasking kernel was developed for this platform. For the purpose of this work, two other requirements were considered, namely the capability to adapt the tasks periods online and the support for the exchange of reflective information between tasks and kernel.

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 periodic tasks with offset control, specification of deadlines equal or shorter than the period and temporal policing. The time management is carried out using a periodic tick that can be configured from 2 ms to 65 534 ms. Periods, offsets and deadlines are 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). 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 kernel supports up to 13 periodic tasks plus a background task. This number was chosen because of the execution overhead of the tick handler when using the EDF scheduling policy (see end of Section III-C). It is, nevertheless, an adequate number to many embedded control applications that typically require a low number of tasks.

The kernel code takes 2900 words out of a total of 32 000 words available, i.e., about 8.8% of the program memory. Concerning the more constrained data memory, the kernel takes 39 bytes plus 31 bytes per task from a total of 4000 bytes of RAM, representing a footprint that varies from 101 bytes (2.5%) for one periodic task plus background to 473 bytes (11.6%) for the maximum number of tasks.

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

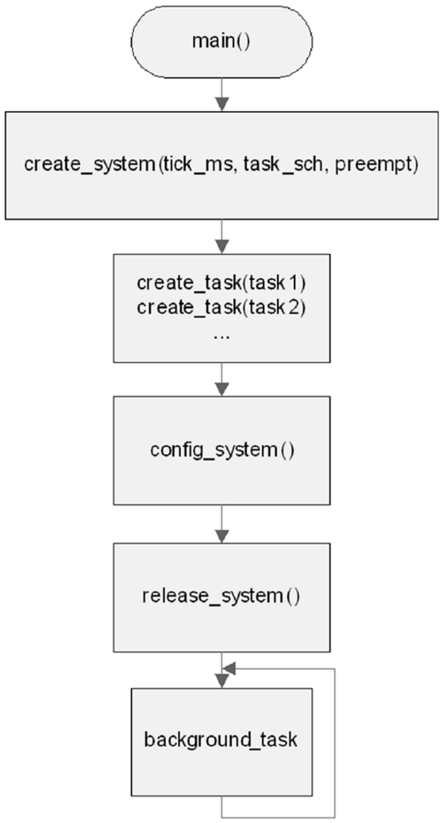


Fig. 2. Structure of the *main* function.

Moreover, to minimize the memory footprint and reduce the 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.

B. Programming Model

The structure of the *main* function in an application is shown in Fig. 2. The program starts invoking the *create_system* system call to specify the duration of the tick in milliseconds, 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.

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 invoking the *release_system* system call, which initiates the tick handling activity and the timing management inside the kernel. All task offsets are counted with respect to this moment in time. Then the system

```
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. 3. Programming model of periodic tasks.

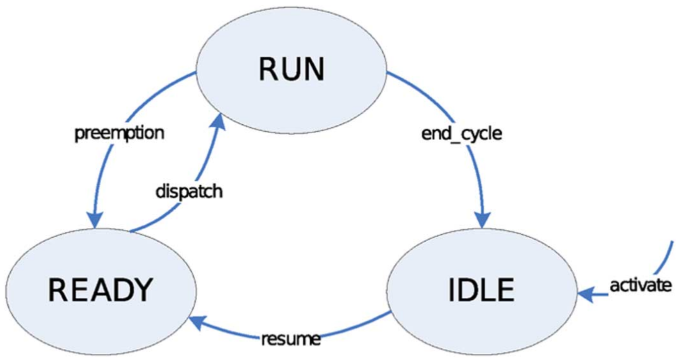


Fig. 4. Tasks state machine.

enters an infinite loop executing the background task, which is preempted for the execution of the periodic tasks, even in nonpreemptive mode.

The programming model of each periodic task is shown in Fig. 3. 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 [20]. 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 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 Fig. 4.

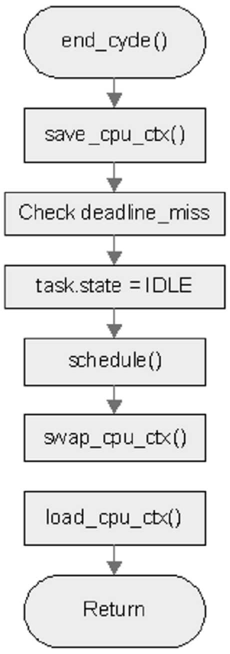


Fig. 5. Internals of the *end_cycle* system call.

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 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. 5).

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 overlapped for all independent functions and exclusive for functions that call one another. Note that with a single execution thread, independent functions never overlap in time and so when a function starts execution any previous function has terminated and the memory space of its local (dynamic) variables released, thus free for the variables of the starting function. This does not hold for dependent functions that can be active at the same time.

In our case, due to preemption, several execution threads can be active at the same time thus, for consistency, is it important that all contexts are exclusive, which is achieved invoking the *recursive_call* system call in the *main*. This call tells the compiler

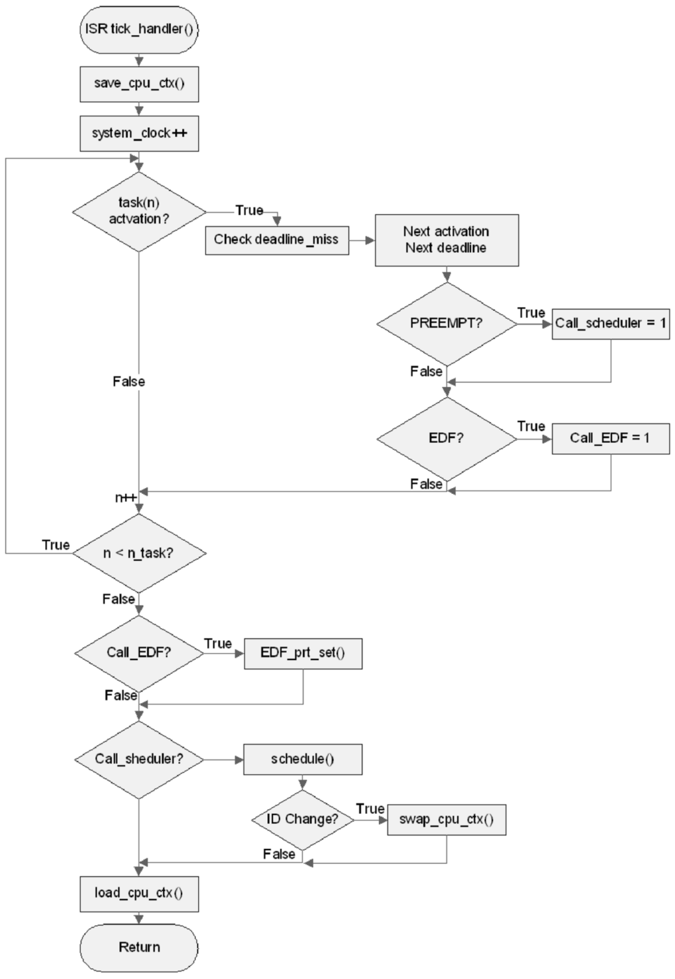


Fig. 6. Structure of the *tick_handler*.

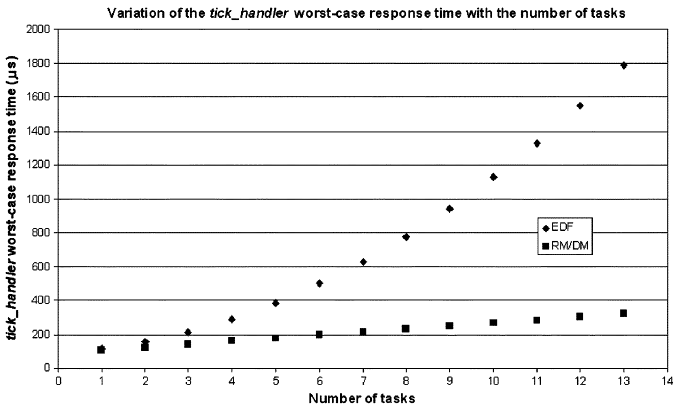


Fig. 7. *Tick_handler* worst-case execution time as a function of the number of tasks activated simultaneously.

that the functions that constitute the tasks must be considered as dependent, despite not calling each other.

The other major role played by the kernel is time management. This is carried out, essentially, within the tick handler. Fig. 6 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 Fig. 7 gives the worst-case execution time of the *tick_handler* for the EDF and RM/DM schedulers as a function of the number of tasks activated simultaneously. The measurements were carried out on a platform based on the 18F258 processor operating at 20 MHz. When using EDF, the dynamic priorities are computed online using the *EDF_prt_set* function, which carries out a sorting algorithm thus causing an execution penalty. This EDF implementation is simple and integrates well within the tick handler, but its efficiency seems to be suboptimal, limiting the maximum number of tasks that can be activated simultaneously. For example, for a tick period of 2 ms the maximum number of tasks is 13. Note, however, that in general not all tasks are activated every tick, thus strongly reducing the overhead it implies. This makes the current solution practical for several applications such as mobile robots and machine control, in which it has been used successfully. Nevertheless, more efficient EDF implementations are currently being considered for future kernel developments.

D. Reflective Layer

The reflective layer shown in Fig. 1 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 nonpreemptive 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, note 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. Implementing the Optimal Sampling Policy

The goal of the paper is to show that advanced theoretical results on control performance and resource optimization for embedded control systems can be implemented using low-cost solutions. In this section we present a simple experiment involving the implementation of the optimal policy on a prototype implementation of the RTKPIC18 kernel built for the PIC18458F, with two control tasks competing for the CPU. The example simply aims at illustrating the practicality of a recent theoretical result using a rather constrained embedded system. Although more control tasks could be added, this would not alter the results herein presented.

A. Experimental Setup

The scheme for the complete hardware and software setup is illustrated in Fig. 8. The kernel concurrently executes two con-

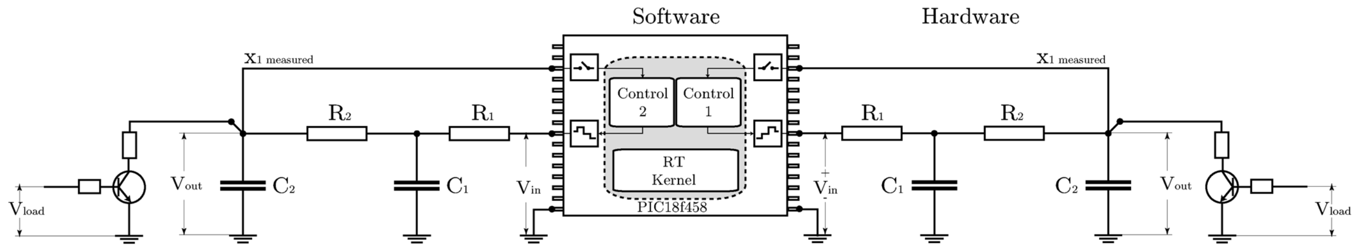


Fig. 8. Hardware and software of the experimental setup.

trol 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 the EDF scheduling algorithm, which is more efficient in terms of CPU utilization for guaranteed schedulability. Moreover, EDF does not introduce any additional asymmetry among tasks, as opposed to fixed priority schemes. The two control tasks, labeled *Control 1* and *Control 2*, execute the code explained in Section IV-C to control each voltage stabilizer.

The electronic components of both stabilizers in the form of RCRC circuits are $R_1 = 330$ k$\Omega, C_1 = 100$ nF, $R_2 = 330$ kΩ, and $C_2 = 100$ nF. Each circuit can be modeled in terms of the currents $\dot{q}_i$ at each R_i by the following equations:

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

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

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

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

$$y = [1 \quad 0]\begin{bmatrix}x_1\\ x_2\end{bmatrix} \tag{4}$$

where u is the control signal, y is the plant output, x_1 corresponds to the voltage in C_2, and x_2 is $\dot{q}_2/C_2$. The control objective is to maintain the output voltage V_{out} of C_2 at 2.4 V regardless of the load perturbations that may affect the system. This is achieved by the control signal generated by each control task sent to the circuit, as illustrated by V_{in}. The sampled controlled variable is V_{out}, as illustrated by $x_{1measured}$. Each circuit is perturbed by the load voltage, as illustrated by V_{load}, which is randomly generated following an uniform distribution.

B. Slack Management

The optimal policy can make use of dynamic slack (reserved but unused resources) or static slack (unreserved resources). In the experimental setup, we limit the application of the optimal policy to static slack and therefore, we consider that the available slack is constant. The primitive *get_CPU_util* computes the CPU utilization from the period and worst case execution time of each task in the system, information that is stored in each task control block. Using only static slack limits achievable control performance improvement. On the other hand, it simplifies operation because using dynamic slack involves performing execution time measurements and associated operations, which complicate the kernel and increase overhead [21]. In any case, the results show that efficiently using static slack can already improve control performance.

Provided that slack is constant, the optimal policy mandates to run each control task at two rates given by a shortest and longest sampling period. Specifically, the experiment is designed to guarantee the minimum rate (longest period) for each control task but both tasks cannot simultaneously run at their maximum rate (shortest period). However, the available slack is enough to run one task at its maximum rate while the other runs at its minimum rate.

The task dedicated to the slack allocation reads the error value stored in the reflective memory by each control task. It also executes *get_CPU_util*, which implies scanning the list of n tasks and computing and accumulating their utilizations. With these values, it sets the new two tasks periods as mandated by the optimal policy. In general, the complexity of the algorithm implemented in the task doing the slack management is $O(n)$, i.e., linear with n, the number of tasks [2].

In this particular implementation, the task implementing the optimal policy assigns the shortest sampling period to the task with greatest error while the other task is given the longest period. The new tasks periods are updated in each task control block. The dedicated task executes at slower rate than the control tasks. In our experiments, its period is set to 200 ms. It is out of the scope of this paper to analyze which optimal period should the slack allocation task have. The shortest its period, the more reactive will the system be, thus improving the error tracking and increasing the potential for better performance. However, this would also imply more overhead due to its execution.

C. Controller Design and Implementation

As mandated by the optimal policy, each control task will have to run with two different sampling periods. At the control algorithm level, this is accommodated by switching controller gains according to the period that applies. Therefore, two controller gains have been calculated. Optimal LQ control [22] has

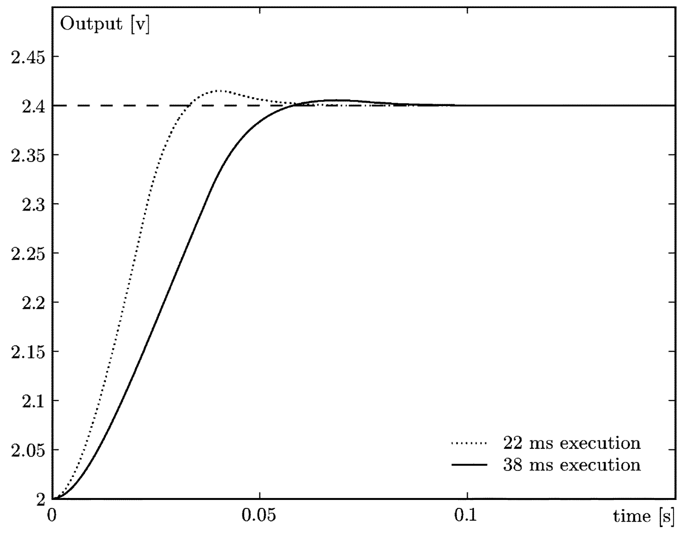


Fig. 9. Systems dynamics for both controller rates.

been the basis for controller design. The quadratic cost function to be minimized is given by

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

where

$$Q = \begin{bmatrix} 400 & 1 \\ 1 & 0 \end{bmatrix}, \quad R = 1. \tag{6}$$

That is, since the focus of the controller is to ensure fast tracking, errors between the output variable and the desired output variable are severely penalized. 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).

In the implementation, the two sampling periods for each control task are 22 ms and 38 ms. With these periods, the two gains are $k_{22} = [4.274\ 0.048]$ and $k_{38} = [2.068\ 0.025]$. In addition to the controller gains, deadbeat reduced observers for estimating the second state variable have been designed for the two task rates. The expected dynamics for the plant with these two gains are illustrated in Fig. 9.

The pseudo-code of the control task is given in Fig. 10. The pseudo-code starts by defining variables, constants, etc. Only the most significant ones are shown: two gains $K22$ and $K38$, 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 task *****************/

void task1(void) {

/*----------- Definitions -------------------------*/

    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=K22;
        }
        else
        {
            ob = ob2;
            K=K38;
        }
        /* 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();

    }
}
```

Fig. 10. Control task pseudo-code.

control signal u is computed, taking into account the estimation given by the observer. It is therefore at this point where the specific controller gain is applied. 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. Here it has been omitted for the sake of clarity.

Differently to approaches like [3], [4], or [5], in which stability is guaranteed by solving the control performance and resource allocation problem, using the optimal policy [2] requires a separate stability check. Many existing stability techniques can be applied. For example, it is easy to check that there is a common Lyapunov function [23] that guarantees stability for each control loop regardless of any changes in tasks periods.

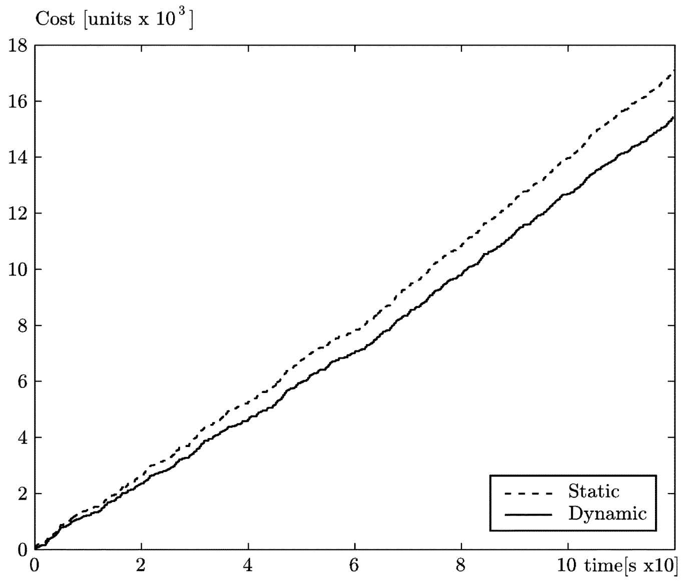


Fig. 11. Static versus dynamic.

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 38 ms and the other at fixed 22 ms. Therefore, they implement the traditional controller, that is, a constant controller gain, which coincides with the optimal gain designed in Section IV-C.

Fig. 11 shows the results in accumulated cost, i.e., control error, of both static and dynamic (optimal) policies. The accumulated cost is the evaluation of the cost function (5) over the experiment time. The lower the curve, the better the performance. Both systems were run for two minutes.

As it can be seen in the Fig. 11, the scenario with slack allocation (dynamic) outperforms the static by more than 10%. Note that the resource utilization in both runs is the same. In the dynamic case, the two control tasks cannot execute at the highest rate, so they alternate between 22 and 38 ms, giving an average for each task of 30 ms. In the static, both tasks always execute at a fixed rate, with average of 30 ms.

V. Conclusion

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 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, with low-cost microcontrollers, namely those of the PIC18Fxx8 family.

The paper presents the referred kernel, the RTKPIC18, which supports preemptive and nonpreemptive 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 online 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 *Proc. 25th IEEE Real-Time Systems Symp.*, Dec. 2004.

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

[4] M.-M. Ben Gaid, A. Çela, Y. Hamam, and C. Ionete, "Optimal scheduling of control tasks with state feedback resource allocation," in *Proc. 2006 Amer. Control Conf.*, Minneapolis, MN, Jun. 2006.

[5] 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 *Proc. 18th Euromicro Conf. Real-Time Systems*, Dresden, Germany, Jul. 2006.

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

[7] G. Cena and A. Valenzano, "On the properties of the flexible time division multiple access technique," *IEEE Trans. Ind. Informat.*, vol. 2, no. 2, pp. 86–94, May 2006.

[8] 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 Trans. Ind. Informat.*, vol. 2, no. 2, pp. 95–102, May 2006.

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

[10] 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 *Proc. 24th IEEE Real-Time Systems Symp.*, Dec. 2003, pp. 396–407.

[11] S. Goddard and L. Xu, "A variable rate execution model," in *Proc. 16th Euromicro Conf. Real-Time Systems*, Jul. 2004, pp. 135–143.

[12] M. Marinoni and G. Buttazzo, "Elastic dvs management in processors with discrete voltage/frequency modes," *IEEE Trans. Ind. Informat.*, vol. 3, no. 1, pp. 51–62, Feb. 2007.

[13] 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.

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

[15] P. Gai, G. Lipari, L. Abeni, M. di Natale, and E. Bini, "Architecture for a portable open source real-time kernel environment," in *Proc. 2nd Real-Time Linux Workshop and Hand's on Real-Time Linux Tutorial*, Nov. 2000.

[16] E. Mumolo, M. Nolich, and M. Noser, "A hard real-time kernel for Motorola microcontrollers," in *Proc. 23rd Int. Conf. Information Technology Interfaces*, Jun. 2001.

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

[18] D. Henriksson and A. Cervin, "Multirate feedback control using the TinyRealTime kernel," in *Proc. 19th Int. Symp. Computer and Information Sciences*, Antalya, Turkey, Oct. 2004.

[19] J. Stankovic and K. Ramamritham*, A Reflective Architecture for Real-Time Operating Systems. Chapter in Advances in Real-Time Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1995.

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

[21] C. Lin and S. A. Brandt, "Improving soft real-time performance through better slack reclaiming," in *Proc. 26th IEEE Real-Time Systems Symp.*, Dec. 2005, pp. 3–14.

[22] K. J. Åström and B. Wittenmark*, Computer-Controlled Systems*. Englewood Cliffs, NJ: Prentice-Hall, Jan. 1997.

[23] D. Liberzon*, Switching in Systems and Control*. Cambridge, MA: Birkhauser, 2003.

Ricardo Marau received the degree in electronics and telecomunications from the University of Aveiro, Aveiro, Portugal, in 2004. Currently, he is pursuing the Ph.D. degree at the IEETA research unit in the University of Aveiro.

His research interests include real-time scheduling in communication systems, dynamic reconfigurability, and quality of service management.

Pedro Leite received the degree in electronics and telecommunications engineering in 2004 from the University of Aveiro, Aveiro, Portugal.

He is currently working as a Project Engineer of Riamolde Engenharia e Sistemas, SA, where he is involved in the development of test and measurement equipments, using artificial vision, for the automotive industry.

Manel Velasco received the maritime engineering degree and the Ph.D. degree in automatic control from the Technical University of Catalonia, Barcelona, Spain, in 1999 and 2006, respectively.

Since 2002, he has been an Assistant Professor in the Department of Automatic Control at the Technical University of Catalonia. He has been involved in research on artificial intelligence from 1999 to 2002 and, since 2000, on the impact of real-time systems on control systems. His research interests include artificial intelligence, real-time control systems, and collaborative control systems, especially on redundant controllers and multiple controllers with self-interacting systems.

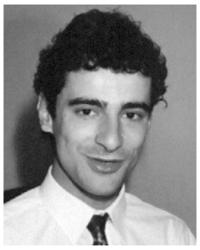
Pau Martí (M'02) received the degree in computer science and the Ph.D. degree in automatic control from the Technical University of Catalonia, Barcelona, Spain, in 1996 and 2002, respectively.

Since 1996, he has been an Assistant Professor in the Department of Automatic Control at the Technical University of Catalonia. During his Ph.D. studies, he was a visiting student at Malardalen University, Västerås, Sweden. From 2003 to 2004, he held a research fellow appointment in the Computer Science Department at the University of California at Santa Cruz. His research interests include embedded systems, control systems, real-time systems, and communication systems.

Luis Almeida (S'86-M'89) received the licenciatura degree in electronics and telecommunications engineering and the Ph.D. degree in electrical engineering both from the University of Aveiro, Aveiro, Portugal, in 1988 and 1999, respectively.

He is an Assistant Professor in the Electronics and Telecommunications Department and a Senior Researcher at the IEETA research unit of the University of Aveiro. Formerly, he was a Design Engineer in a company producing digital telecommunications equipment. His research interests lie in the fields of real-time networks for distributed industrial/embedded systems and control architectures for mobile robots.

Dr. Almeida is a Senior Member of the IEEE Computer Society.



Paulo Pedreiras (M'03) received the licenciatura degree in electronics and telecommunications engineering and the Ph.D. degree in electrical engineering from University of Aveiro, Aveiro, Portugal, in 1997 and 2003, respectively.

Currently, he is an Invited Assistant Professor in the Department of Electronics and Telecommunications at the University of Aveiro. His main research interests include distributed embedded systems, real-time networks, real-time operating systems, and mobile robotics.

Josep M. Fuertes (S'74-M'96) received the Ph.D. degree from the Technical University of Catalonia (UPC), Barcelona, Spain, in 1986.

He became a Permanent Professor at UPC in 1987. He was a Researcher (1975–1986) at the Institut de Ciberntica (Spanish Consejo Superior de Investigaciones Cientficas). In 1987, he spent the year at the Lawrence Berkeley Laboratory working as a Visiting Scientific Fellow on the design of the Active Control System of the W. M. Keck 10-m segmented telescope (Hawaii). From 1990 to 1992, he was responsible for the doctoral program of the Department of Automatic Control and Computer Engineering at UPC. From 1992 to 1999, he was the Director of the University Research Line in Advanced Control Systems and, since 2001, he has been the Director of the Department of Automatic Control and Computer Engineering, UPC. Since 1989, he has coordinated projects at the national and international levels related to his areas of expertise. He has authored or coauthored more than 80 papers on distributed control systems and applications.

Dr. Fuertes has collaborated as an organizer (IEEE-WFCS97, IEEE-ETFA99), chairman, session organizer, and member of program committees for several international conferences. He acted as the Spanish Representative at the Council of the European Union Control Association (EUCA). He was also an AdCom Member of the IEEE Industrial Electronics Society and is a member of several other scientific and technical societies.