Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 390106, 13 pages
doi:10.1155/2008/390106

Research Article

A Real-Time Embedded Kernel for Nonvisual Robotic Sensors

Enzo Mumolo,[1] Massimiliano Nolich,[1] and Kristijan Lenac[1,2]

[1] *DEEI, Università degli Studi di Trieste, 34127 Trieste, Italy*
[2] *AIBS-Lab S.r.l., Via del Follatoio 12, 34148 Trieste, Italy*

Correspondence should be addressed to Enzo Mumolo, mumolo@units.it

Received 5 April 2007; Revised 4 December 2007; Accepted 11 January 2008

Recommended by Alfons Crespo

We describe a novel and flexible real-time kernel, called Yartek, with low overhead and low footprint suitable for embedded systems. The motivation of this development was due to the difficulty to find a free and stable real-time kernel suitable for our necessities. Yartek has been developed on a Coldfire microcontroller. The real-time periodic tasks are scheduled using nonpreemptive EDF, while the non-real-time tasks are scheduled in background. It uses a deferred interrupt mechanism, and memory is managed using contiguous allocation. Also, a design methodology was devised for the nonpreemptive EDF scheduling, based on the computation of bounds on the periodic task durations. Finally, we describe a case study, namely, an embedded system developed with Yartek for the implementation of nonvisual perception for mobile robots. This application has been designed using the proposed design methodology.

1. INTRODUCTION

Embedded systems are hardware and software devices designed to perform a dedicated function, therefore, generally not allowing users to build and execute their own programs. Moreover, since embedded devices goal is to control a physical system, such as a robot, time factor is important, as are the size and cost. As a matter of fact, generally embedded systems require to be operated under the control of a real time operating system running on processors or controllers with limited computational power.

In this paper, we present a novel, flexible, real time kernel with low overhead and low footprint for embedded applications and describe an embedded application developed with it. The reason behind this development is the unavailability of an open source real-time operating system with the requested features. In fact, our objective was to realize small autonomous embedded systems for implementing real-time algorithms for nonvisual robotic sensors, such as infrared, tactile/force, inertial devices, or ultrasonic proximity sensors, as described for example in [1, 2]. Our requirements are: real-time operation with nonpreemptive scheduling, deferred interrupt mechanism, low footprint, and low overhead. Nonpreemptive scheduling is suitable to process nonvisual sensors, because most of these sensors use time of flight measurements which are cheaper to perform using polling rather than interrupt management. Furthermore, nonpreemption leads to a lower overhead, as requested by low performance microcontrollers. In addition to nonpreemption, also nonreal-time preemptive tasks are needed, especially for the communication with external devices. As a matter of fact, we interact with external devices through a serial port which is managed using interrupts and served by non-real-time tasks.

The kernel is called Yartek (yet another real time embedded kernel) and its source code is freely available online [3]. Yartek has been developed by modifying the scheduling module of another tiny operating system described in [4], and it is suitable for running on microcontrollers, since it uses a small amount of resources. In general, preemptive scheduling should be preferred over nonpreemptive policies in term of utilization factor. However, there are many applications where properties of hardware devices and software configurations make preemption impossible or expensive. In the application described in this paper, in fact, nonpreemptive management of the sensors leads to a cheaper utilization of computing resources. Moreover, the advantages of nonpreemptive scheduling are: an accurate response analysis, ease of implementation, no synchronization overhead, and reduced stack memory requirements. Finally, in general

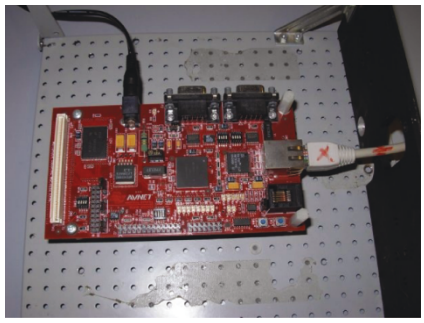

Figure 1: The Coldfire microcontroller.

nonpreemptive scheduling comes with lower-context switch overhead, since there are less interruptions.

In order to operate in unstructured environments, autonomous robots are equipped with a wide range of sensors—visual, like video cameras, and nonvisual, like ultrasonic or inertial sensors—and actuators. The raw sensorial data gathered from the sensors are processed in order to obtain a representation of the perceived environment. However, the robot is controlled by a processor with limited computational power due to the limited power supply of the mobile system. The motivation behind the development of Yartek was the need to build small, autonomous embedded systems which provide the processing requested by nonvisual sensors without imposing a computation burden on the main processor of the robot. In particular, the embedded system described in this paper provides the robot with the environmental map acquired with the ultrasonic sensors.

Yartek allows the creation and running of threads for fast context switch and it is based on a contiguous memory; moreover, it offers a dynamic memory management using a first-fit criterion. The threads can be real-time periodic scheduled with nonpreemptive EDF [5], or nonreal-time. In order to improve the usability of the system, a RAM-disk is included: it is actually an array defined in the main memory and managed using pointers, therefore its operation is very fast. The RAM-disk offers a file system structure for storing temporary data and executable code to enrich the amount of real-time applications which the kernel can run.

Yartek has been developed on a Coldfire microcontroller, in particular on the board having a MCF5282 microcontroller shown in Figure 1.

The main contributions of this paper are the following. The Yartek embedded kernel is introduced, and its performances and comparisons to a different real-time operating system are reported. A simple design methodology for nonpreemptive EDF scheduling is also described, based on bounds on the duration of nonpreemptive tasks. Finally, a real-time application is described, in the field of nonvisual sensor perception in robotics.

This paper is structured as follows. Section 2 summarizes the scheduling policies used in our kernel and proposes a design methodology useful for nonpreemptive tasks. Section 3 describes some technical aspects in the Yartek architecture. Section 4 deals with the performances of this implementation. Section 5 reports a case study where the nonpreemptive design methodology has been applied. Final remarks are discussed in Section 6. Some pieces of source code are reported in Appendices A, B, and C.

2. NONPREEMPTIVE REAL-TIME SCHEDULING

Real-time scheduling of a set of tasks with deadlines means that each task is executed within its deadline. This is a typical requirement of real time kernels for embedded systems, where missing a deadline may lead to an actuator misfunction or missing data during acquisition. We consider a periodic task as a set of instructions periodically invoked: the duration of the ith task is denoted C_i, and its period is denoted p_i.

Remark 1. We assume that time is discrete, and it is indexed by natural numbers because it is measured in clock ticks.

When possible, preemptive EDF is preferred over other strategies because it allows scheduling with high-utilization factors. However, there are many practical situations where nonpreemption of tasks is highly desirable. For example, there are cases where I/O devices make preemption impossible or expensive. Also, nonpreemptive real-time scheduling requires less overhead than preemptive because both synchronization primitives and deadline sorting at each task release are not necessary.

Remark 2. A fundamental parameter in real-time scheduling of n tasks is the utilization factor U: $U = \sum_{i=1}^{n} (C_i/p_i)$.

There are several authors who have presented some results on nonpreemptive scheduling [6]. The main difficulty with nonpreemptive scheduling is that it is, in general, a NP-complete problem [7] for every processor load [8]. In certain constrained cases, the NP-completeness can be broken, as shown by Jeffay et al. in [5] and Georges in [8] for EDF scheduling. In particular, Jeffay et al. show that necessary and sufficient scheduling conditions for a set of n nondecreasing periodic tasks, that is, $p_1 \le p_2 \le \cdots \le p_n$, are the following:

$$\sum_{i=1}^{n} \frac{C_i}{p_i} \le 1, \tag{1}$$

$$t \ge C_i + \sum_{j=1}^{i-1} \left\lfloor \frac{t-1}{p_j} \right\rfloor C_j \quad \forall 1 < i \le n, \ \forall t, \ p_1 < t < p_i. \tag{2}$$

In other words, informally, the EDF scheduling of the set of n periodic tasks is feasible if, according to the first condition, there is enough computational capacity to execute all tasks while, according to the second condition, the total computational demand in a temporal interval t is lower than the length of the interval itself.

2.1. Nonpreemptive design methodology

This section describes a design methodology which is based on the assignment of the computation times C_i of the

nonpreemptive tasks according to the physical requirements and subjected to suitable bounds. In other words, we seek the values of the bounds B_i so that if $C_i < B_i$, for all $i = 1, \ldots, n$, the set of periodic tasks is schedulable.

Starting from the Jeffay conditions (1) and (2), we now derive the bounds for the periodic tasks executions which bring the task set to be schedulable using the EDF nonpreemptive policy.

To this purpose, we can easily prove the following proposition which states a sufficient condition for a set of tasks to be schedulable.

Proposition 1. *If the computation times of a set of n nonpreemptive periodic tasks are bounded by B_i:*

$$B_i = p_1\left(1 - \sum_{j=1}^{i-1}\frac{C_j}{p_j}\right), \tag{3}$$

for $i = 1, \ldots, n$, then the set of tasks is schedulable using nonpreemptive EDF.

Proof. The bounds are a direct consequence of the condition reported in (2), which can be put in the following form:

$$C_i \le t - \sum_{j=1}^{i-1}\left\lfloor\frac{t-1}{p_j}\right\rfloor C_j, \quad \forall i = 2, \ldots, n, \ \forall t : p_1 < t < p_i. \tag{4}$$

If there is only one periodic task, then we can set $B_1 = p_1$. On the other hand, if there are two tasks ($n = 2$), then the condition reported in (4) becomes $C_2 \le t - \lfloor (t-1)/p_1 \rfloor C_1$, for all $t : p_1 < t < p_2$ or, in other words, $B_2 = \min_{p_1<t<p_2}(t - \lfloor (t-1)/p_1 \rfloor C_1)$. Since the possible values for t are: $p_1 + 1, \ldots, p_2 - 1$, we have $B_2 = p_1 + 1 - C_1$. Consider now the situation with i tasks. As before, we have

$$U_i = \min_{p_1<t<p_i}\left(t - \sum_{j=1}^{i-1}\left\lfloor\frac{t-1}{p_j}\right\rfloor C_j\right). \tag{5}$$

By considering the quantity

$$\overline{U_i} = \min_{p_1<t<p_i}\left(t - \sum_{j=1}^{i-1}\frac{t-1}{p_j}C_j\right), \tag{6}$$

which is the same as (5) without the floor operator, then $\overline{U_i} \le U_i$. This means that if $C_i \le \overline{U_i}$, the condition expressed in (2) surely holds. Now, we can easily find out that

$$\begin{aligned}\overline{U_i} &= \min_{p_1<t<p_i}\left(t - \sum_{j=1}^{i-1}\frac{t-1}{p_j}C_j\right)\\ &= \min_{p_1<t<p_i}\left(t\left(1 - \sum_{j=1}^{i-1}\frac{C_j}{p_j}\right) + \sum_{j=1}^{i-1}\frac{C_j}{p_j}\right)\\ &= (p_1+1)\left(1 - \sum_{j=1}^{i-1}\frac{C_j}{p_j}\right) + \sum_{j=1}^{i-1}\frac{C_j}{p_j}\\ &= p_1\left(1 - \sum_{j=1}^{i-1}\frac{C_j}{p_j}\right) + 1.\end{aligned} \tag{7}$$

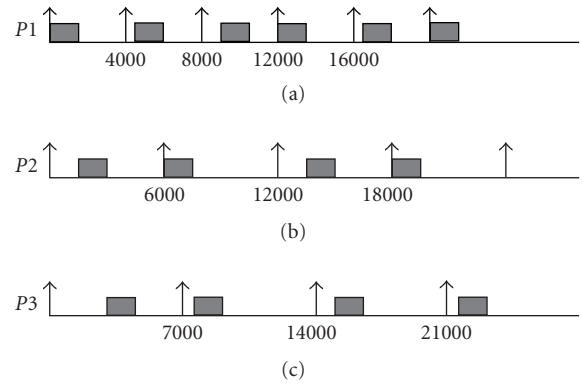

Figure 2: Scheduling example.

On the other hand, the first Jeffay condition, expressed in (1), can be stated as

$$C_i \le V_i = \left(1 - \sum_{j=1}^{i-1}\frac{C_j}{p_j}\right)p_i. \tag{8}$$

In conclusion, we can state that if $C_i \le B_i$, where $B_i = \min(\overline{U_i}, V_i) = \overline{U_i}$ as the p_i's are in nondecreasing order and so $\overline{U_i} \le V_i$, both the Jeffay conditions are satisfied, and the task set is EDF schedulable. It is worth noting that the opposite is not true, namely the condition is only sufficient.

This derivation completes the proof. □

From the proposition, we can immediately plan a design methodology for nonpreemptive real-time scheduling, consisting in finding the bounds of each tasks which guarantee scheduling of the task set, and setting the duration of the tasks within the bounds and according to the physical constraints. In other words, the physical system to be controlled through the real time kernel must have time constants less than the computed bounds. Otherwise, the architecture of the real time solution must be formulated in a different way.

To show how the above conditions can be used in practice, we have worked out the following example.

Example 1. Let us consider three tasks, with $p_1 = 4000$, $p_2 = 6000$, and $p_3 = 7000$. Then, $B_1 = 4000$, and assume that $C_1 = 1500$. Then, the bounds, which guarantee the tasks to be schedulable, are: $B_2 = 4000(1 - C_1/p_1) = 2500$. Assume then that $C_2 = 1500$. In the same way, $B_3 = 4000(1 - C_1/p_1 - C_2/p_2) = 1500$. Assume then that $C_3 = 1500$.

This scheduling is outlined in Figure 2.

Remark 3. The algorithm has a complexity of $O(n^2)$ divisions, where n is the number of tasks.

In fact, for $i = 2$, we have to compute one division, for $i = 3$, we have two divisions, and for the generic $i = n$, we have one product and $n - 1$ divisions. It is worth noting that complexity is not a critical problem, because the scheduling is statically designed.

```
MainLoop()   {
      while(true){
            if (InterruptTable is not empty)
                  ServiceInterruptTable();
            else
                  ServicetaskQueue();
      }
}
```

Pseudocode 1

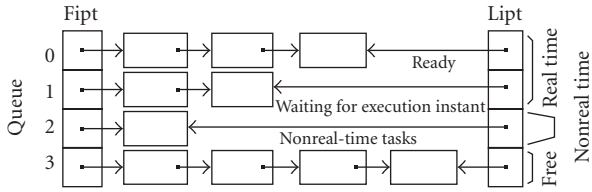

Figure 3: Yartek queues used for scheduling.

3. YARTEK ARCHITECTURE

Yartek has been designed according to the following characteristics:

(i) running on the Freescale MCF5282 Coldfire microcontroller [9];
(ii) nonpreemptive EDF scheduling of periodic real-time threads;
(iii) background scheduling of nonreal-time threads;
(iv) sensor data acquisition with a polling mechanism;
(v) deferred interrupt mechanism;
(vi) contiguous stack and data memory management using first-fit policy;
(vii) RAM-disk management;
(viii) system call primitives for thread, memory, and file management;
(ix) general purpose I/O management;
(x) communication with the external world via serial port.

3.1. Task scheduling

Task scheduling is one of the main activities of the operating system. All the scheduling operations are performed on the basis of a real-time clock, called RTClock, which is generated by an internal timer. Each task is represented using a data structure called thread control block (TCB), which is reported in Appendix A. TCB contains the name, type, and priority of the process, its allocated memory, and fields used to store the processor's state during task execution. For real-time processes, the TCB also contains *Start*, *Dline*, and *Period* fields to store the time when the process starts, its deadline, and its period. Scheduling is managed with a linked list of TCBs with 3 priority levels, as shown in Figure 3. There is one more queue used for storing free TCBs.

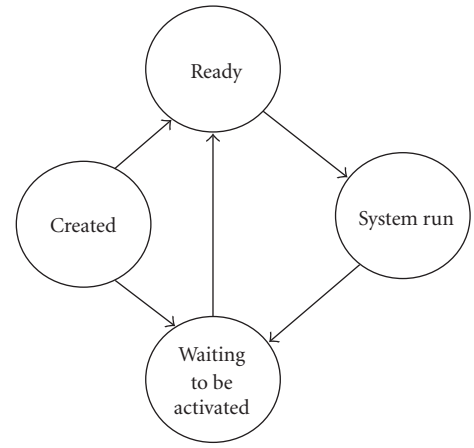

Figure 4: State diagram of a real-time thread in Yartek.

The periodic real-time threads are managed using nonpreemptive EDF scheduling. The TCB queues 0 and 1 are used as follows: queue 0 contains the TCBs of active threads, that is, ready to be scheduled and is ordered by nondecreasing deadline; hence these TCBs are executed according to the EDF policy. The periodic threads awaiting to be activated are inserted in queue 1 ordered by start time. As time lasts, some tasks can become active and the corresponding TCBs will be removed from queue 1 and inserted into queue 0. This operation is performed by the ServiceTaskQueue routine, which analyzes the TCBs on queue 1 to seek start times less than or equal to RTClock, that is, threads to be activated. Moreover, in queue 2 are stored the TCBs of nonreal-time threads. The queue 2 is managed using a FIFO policy.

The entry point of the kernel is an infinite loop where the interrupt table and the task queue are examined, as described in Pseudocode 1.

Interrupts are served using a deferred mechanism: each interrupt raises a flag on an interrupt table, reported in the Appendix A, and the ServiceInterruptTable routine checks the interrupt table to verify if a pending interrupt flag is set. In this case, it activates the suitable nonreal-time thread for serving that interrupt. In Appendix B, we report more detailed code.

The movement of a TCB from a queue to another at a given priority level is performed with a procedure which inserts the task in the task queue.

3.2. Process states

When a real-time thread is created (see Figure 4), it is in *Ready* state when *start time* > *RTClock* (TCB inside queue 0), it is in *Waiting to be activated* state when *start time* > *RTClock* (TCB inside queue 1). The first *Ready* thread will then be selected for execution and will go into *System run* state. When the execution stops, the process will become *Waiting to be activated* as it is periodic, and the scheduler updates its start time and its deadline adding them the thread period.

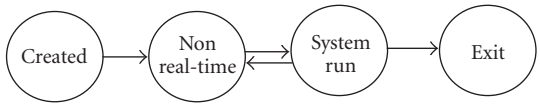

Figure 5: State diagram of a non-real-time thread in Yartek.

```
Interrupt_handler:
        set the flag in the operating system table;
        return from interrupt;
```

Pseudocode 2

The states of a nonreal-time thread are similar and are reported in Figure 5. The main difference is that a nonreal-time thread is preemptive, and it can be interrupted by real-time threads.

To create a process, a TCB is taken with GetTCB() function and filled with process information. The QueueTCB function then inserts the TCB in the requested queue. For queue 0, the TCB is inserted in ascending order according to deadlines, while for the queue 1 the TCB is inserted in ascending order according to activation time. For the other two queues, it is simply enqueued at the end.

3.3. Memory management

An amount of stack and data memory, containing thread-related information such as a local file table and information needed for thread management and user variables, is assigned to each process; furthermore, dynamic memory is also available when requested by system calls. Stack, data, and heap memory are organized in a sequence of blocks managed with first-fit policy.

3.4. System calls

A number of system calls have been implemented using the exception mechanism based on the trap instruction. The system calls are divided into file system management (open, read, write, close, unlink, rewind, chname), process management (exec, kill, exit), heap management (alloc, free), and thread management functions (suspend, resume).

3.5. Timer and interrupts

The microcontroller MCF5282 [9] has 4 programmable timers: one is used as the system's time reference, and it is used as RTClock, and the other timers are used for the measurement of time intervals. The timer is composed of a 16-bit register and a frequency divider. The first timer is used as the system's time reference, and it is used as RTClock. Since four interrupts are used for the timers, there are three interrupt levels for application code. The routines activated by interrupts set a single flag in the interrupt table. Later, the scheduler activates a process to actually manage the request, that is, in deferred mode. The pseudocode of an interrupt service routine is illustrated in Pseudocode 2.

4. PERFORMANCE EVALUATION

Generally speaking, as noted in [10], measuring real-time operating system performance and comparing a real time system to other real-time operating systems are difficult tasks. The first problem is the fact that different systems can have different functionalities, and the second concern is the method used to perform the actual measurements. Many features are worth to be measured: for example, Sacha [10] measures the speed of inter-task communication, speed of context switch, and speed of interrupt handling, while Garcia-Martinez et al. [11] reported measurements of responses to external events, inter-task synchronization and resource sharing, and inter-task data transferring. Finally, Baynes et al. [12] considered what happens when a real-time operating system is pushed beyond its limits; they also report real-time operating system power consumption measurements.

This section reports some measures used to describe the performance of Yartek, namely context switch time, jitter time, interrupt latency time, kernel stability, and kernel overhead. Yartek has been implemented on the Avnet board [13] part number ADS-MOT-5282-EVL, based on the Freescale MCF5282 ColdFire Processor running at 33 MHz. It is equipped with BDM/JTAG interface and has 16 MB SDRAM and 8 MB Flash. The communications are based on general purpose I/O (GPIO) on AvBus expansion connector, two RS-232 serial ports, 10/100 Ethernet. The performances obtained with Yartek have been compared to the performance of RTAI operating system, ported to this board.

4.1. RTAI

The RTAI project [14–16] began at the Dipartimento di Ingegneria Aerospaziale del Politecnico di Milano (DIAPM) as a plug-in which permits Linux to fulfil some real-time constraints. RTAI allows real-time tasks to run concurrently with Linux processes and offers some services related to hardware management layer dealing with peripherals, scheduling, and communication means among tasks and Linux processes. In particular, RTAI port for Coldfire microcontroller is a uCLinux [17] kernel extension that allows to preempt the kernel at any time to perform real-time operations. Unlike the implementations of RTAI for x86, PPC, and MIPS, the Coldfire version is not based on the "deferred interrupt mechanism" but uses the capability of the MC68000 architecture to priorities interrupts in hardware by the interrupt controller. In the current implementation, Linux interrupts are assigned lower-priority interrupt levels than RTAI interrupts. However, RTAI presents drawbacks which restrict the fields of integration:

(1) the preemptive scheduler works with static priorities and there is no built-in nonpreemptive EDF scheduler,
(2) both aperiodic and periodic tasks can be used but the priority of a periodic task bears no relation to its period,
(3) deadlines are not used,
(4) it is a hybrid system with high footprint.

```
void test_context_switch( ) {
  int i;
  CreateRTtask(thread, thread_code);
  ActivateTask(thread);
  t0 = StartTimer();
  for (i = 0, i < LOOPS; + + i){
   Resume(thread);
  }
  t1 = StopTimer();
}

void thread_code( ) {
  while (1) {
   Suspend();
  }
```

Pseudocode 3: Pseudocode of the context-switch test.

```
void test_jitter_time( ) {
  CreateRTtask(thread, thread_code);
  ActivateTask(thread);
}

void thread_code( ) {
  int loops=LOOPS;
  while (loops–) {
   t1[loops] = RTClock();
   t2[loops] = CurrentTCB->StartTime;
  }
}
```

Pseudocode 4: Pseudocode of the jitter time test.

4.2. Performance measures and comparisons

Experimental results with respect to kernel performances are based on the following parameters.

Context switch time

It is the time spent during the execution of the context switch routine of the scheduler.

Jitter time

It is the time delay between the activation time of a periodic real-time process and the actual time in which the process starts.

Deferred interrupt latency

It is the time delay between an interrupt event and the execution of the first instruction of the deferred task scheduling its service routine.

Kernel stability

It establishes the robustness of the kernel.

Kernel overhead

It represents the time the kernel spends for its functioning.

4.2.1. Test for context switch time

The test program used for measuring the context switch time is shown in Pseudocode 3.

The test program creates one thread and measures the time which lasts from the thread execution to subsequent suspension repeating the process *LOOPS* times. More precisely, in Yartek the only thing the thread does is to suspend itself, that is to put its TCB on the queue 1 while the loop calls the resume *LOOPS* times. *Resume*() moves the TCB on the queue 0 so it will be immediately scheduled. It is worth noting that *Resume*() is a blocking primitives, that is, it exits only when the TCB is put indeed in queue 0.

In RTAI, *rt_task_suspend*() suspends the execution of the task given as the argument, *rt_task_resume*() resumes execution of the task indicated as argument previously suspended by *rt_task_suspend*().

In Yartek, the average context switch time is 130 μs, while in RTAI is 124 μs.

4.2.2. Test for jitter time

It is worth recalling the detailed operation of Yartek to manage periodic real-time tasks. The TCB queue 1 contains the periodic threads, ordered by activation time, whose activation time is in the future. The starting time of the periodic tasks that lie on queue 1 is tested to detect the scheduling condition, that is, the starting time in TCB is greater than current time. In this case, the TCB is moved to the queue 0 which is sorted by deadline.

The test program used for measuring jitter time is shown in Pseudocode 4.

In Yartek, TCB is the data structure containing the data related to threads, and CurrentTCB is the pointer to the current thread. The system call RTClock() returns the value of the real-time clock used by Yartek for scheduling real-time tasks. However, in Yartek, real-time threads are activated by the ServiceTaskQueue routine, and therefore the exact starting time does not necessarily correspond to the start time written in the TCB. The delay can vary depending on the state of the routine. As a consequence, the uncertainty of activation time can be quite high. In fact, results of our tests show that Yartek has an average jitter time of 750 μs, while RTAI has an average jitter time of 182 μs.

4.2.3. Test for deferred interrupt latency

Yartek serves the interrupt using a deferred mechanism, that is, only a flag is raised immediately. A nonreal-time thread is activated only when the scheduler processes the Interrupt table.

```
void test_deferred_interrupt_latency( ) {
  CreateRTtask(thread1, thread_code1);
  ActivateTask(thread1);
}

void thread_code1( ) {
  CreateRTtask(Thread2, thread_code2);
  t1 = StartTimer();
  ActivateTask(thread2);
}

void thread_code2( ) {
  t2 = StopTimer();
}
```

Pseudocode 5: Pseudocode of the deferred interrupt latency test.

As RTAI does not implement on the Coldfire porting any interrupt service, we have estimated the time needed to schedule a task, thus implementing a deferred interrupt service.

So, the test program used for measuring deferred interrupt latency is schematically presented in Pseudocode 5. There are no periodic real-time threads in execution during the tests.

By using Yartek, an average deferred interrupt latency of 780 μs is obtained, while using RTAI a latency of 17 microseconds is obtained.

4.2.4. Kernel stability and overhead

Although we did not make a specific stability test, Yartek has shown a good robustness since it run for several hours both for performing applications and for evaluating the performance tests.

The time the kernel spends for itself and not for the application is mainly divided in scheduling time, context switch time, and memory management time. The essential code of scheduling is reported in the Appendix B.

In summary, the worst case complexity for interrupt management is about 110 assembler instructions to deferred schedule a nonreal-time thread for serving one interrupt. The management of the real-time task queues, under the hypothesis of one real-time TCB to be activated, requires about 150 assembler instructions. Regarding memory management, the alloc system call requires about 300 assembler instructions for allocating one block of contiguous memory using first fit and the free system call about 270 assembler instructions. These overheads express in time depend on the actual CPU frequency and for this reason have been left in number of instructions.

As previously computed, the overhead for context switch is 130 μs.

4.3. Discussion

The first important difference between RTAI and Yartek is the operating system footprint. The footprint of an operating system concerns the usage of RAM and flash memory resources. As noted in Section 4.1, the RTAI plug-in works with a Linux kernel. In our tests, we used uCLinux which has a minimum kernel size of 829 kbytes. The RTAI modules have a size of 97 kbytes, so the whole image is of about 900 kbytes. Instead, the footprint of Yartek is about 120 kbytes. This big difference in size is due to the fact that Linux plus RTAI brings some of the powerful tools and features of Unix. However, these tools are not necessary for an embedded system. The second difference between RTAI and Yartek is the nonpreemptive scheduling. It is worth remarking the adequacy of nonpreemptive scheduling in real-time embedded systems on low-power microcontrollers. RTAI does not offer nonpreemptive scheduling. Of course, it could be introduced by coding a new scheduler and integrating it in RTAI, but we instead decided to modify our previous kernel [4] for footprint reasons.

In conclusion, we decided to use Yartek for developing embedded solutions. The performance evaluation tests show that the time performances of Yartek are similar to RTAI for the context switch time and that the time for task creation is much lower for Yartek than RTAI. However, the jitter time for Yartek is much worst than RTAI, due to the Yartek architecture. It has to be noted, however, that the embedded systems for nonvisual sensors is not critical with respect to the jitter time, due to the time constants of such sensors. Yartek allows to manage task allocation that will be much more complex on a nonreal time or a sequential system providing periodic threads scheduled with a nonpreemptive EDF policy; the schedulability can be tested using the simple methodology presented in Section 2.

5. APPLICATION: EMBEDDED MAP BUILDING SYSTEM FOR MOBILE ROBOTS

In mobile robotics, several tasks require the strict satisfaction of time constraints, so real-time systems are needed. The sensors generally used for mobile robot navigation, such as inertial navigation systems, sonar sensor array, and GPS, laser beacons, should be processed considering real-time constraints. As map building is a fundamental task in mobile robotics, we considered an application of this type. Its design is described, and its implementation using Yartek is outlined in the following.

5.1. Previous work in map building for robot navigation

The problem of robotic map building is that of acquiring a spatial model of a robot's environment; this model is used for robot navigation and localization [18]. To acquire a map, a robot must be equipped with sensors that enable it to perceive the outside environment. Commonly used sensors include cameras, range finders using sonar, laser, and infrared technology, tactile sensors, compasses, and GPS. However, none of these sensors can furnish a complete view of the environment. For example, light and sound cannot penetrate walls. This makes it necessary for a robot to build a map of the environment while navigating in it.

Working in fusing multiple sensor readings for map building falls into two broad categories: target tracking models and occupancy grid models. In target tracking, one or more geometric features of the environment are modeled and tracked, that is, their location is estimated at each new sensor reading [19, 20].

Target-tracking methods are appropriate when there is a small number of targets, such as a few landmarks, and their interaction with the sensor is well known. A key issue in the target-tracking paradigm is the data-association problem: how to identify the target that a given sensor reading is associated with. While target-tracking is a good method for navigation using landmarks, in many situations it may be important to determine not just the position of a few landmarks, but the complete surface geometry of the environment.

The occupancy grid method [21–23] provides a probabilistic framework for target detection, that is, determining whether a region of space is occupied or not.

Elfes [21, 22] reformulated the problem as a probabilistic Bayesian updating problem using gaussian noise with a very large variance to account for the gross errors entailed by multiple reflections. He also addressed the problem of geometric uncertainty associated with sensor beam width by considering target detection under all possible configurations of the environment. In practice, given the overwhelming combinatorics of keeping track of data associations for each reading, independence and other simplifying assumptions are made to reduce the computational complexity of Bayesian update. That is, each cell of space is treated as an independent target in the presence of the geometric uncertainties induced by the beam width. This leads to unrealistic estimates for target map updates, for example, all the cells at the leading edge of the beam have their probabilities raised, when in fact usually only one cell is responsible for the echo.

Borenstein and Koren [24, 25] introduced the vector field histogram (VFH) method. They use a spatial histogram of sonar points, along the axis of the sonar beam, to identify areas that are likely to contain obstacles. The histogram is updated rapidly as new returns come in, and older ones are abandoned. The VFH method has the advantage that it can deal with dynamic and noisy environments. Since it is based on a continuous update of the map, this method is particularly suitable for mobile robots. The updating of the map can in fact be performed during the robot movement.

5.2. Implementation

Yartek has been installed on an embedded system composed by the Coldfire microcontroller board connected to an array of 6 sonar sensors placed in front of a mobile robot, and a PC-104 board on top of the robot, as shown in Figure 6.

The embedded system controls the ultrasonic sensors, calculates the robot position from the odometry readings, and updates the internal map. The last N_c sensor readings are stored internally in a circular list and used for updating the map with N_c set according to the odometry error model. The map is a histogram grid [24, 25] which is quite simple for a computational point of view. It is suitable for rapid in-

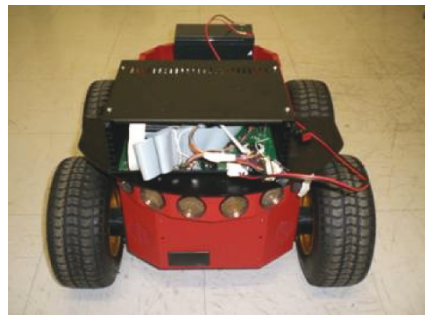

Figure 6: Mobile robot, type PIONEER 3-AT, equipped with the embedded system.

motion sampling of onboard range sensors and modeling of inaccurate and noisy range-sensor data, such as that produced by ultrasonic sensors. The system provides the internal map through a serial port and receives commands such as clear the map, compute the map, and send the map.

The application is designed as follows.

Sensor acquisition tasks

There are 6 real-time periodic tasks that perform the acquisition of the data from the sonar sensor and 2 real-time periodic tasks that read the odometers.

Map updating tasks

There is a real-time periodic task that updates the map according to the acquired sensorial data provided by the sensor acquisition task and antisensor real-time periodic task with larger period for filtering erroneous readings.

Map requests

These are aperiodic tasks scheduled upon requests. There is an aperiodic task that allows to obtain the complete map from the embedded system using serial line connection. There is also a task for clearing the map and resetting the robot's position.

There is one real-time task for each ultrasonic sensor that controls the firing of the sensor and waits for the reflected ultrasonic burst. This is compatible with the sonar array operation since only one sensor fires the ultrasonic burst at a time in order to avoid crosstalk. Crosstalk is a common problem occurring with arrays of multiple ultrasonic sensors when echo emitted from one sensor is received by another sensor leading to erroneous reading. In order to avoid crosstalk only one sensor emits the ultrasonic burst and listens for the echo. Only after the echo has been received or a maximum allowed time has elapsed in the case there is no obstacle in front of the sensor, the process can be repeated for the next sensor, and so on.

The range distance of the sensor is 3 m which is equivalent to a time of flight (TOF) of 17 milliseconds. We assume that the set of sensors is read every 500 milliseconds. This fact poses a physical limit on the maximum robot speed

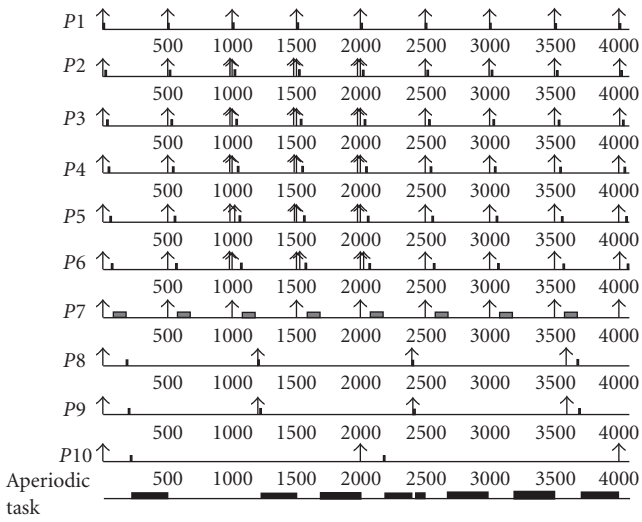

Figure 7: Time diagram of real-time scheduling for the map building application.

```
robot.getSonarN()
{
   fire ultrasonic sensor N (set logical 1 on digital pin)
   while time < maxtime and echo not detected
   {
        listen for echo (check the I/O pin of sensor N to go
                         to logical level 1)
        time++
   }
   calculate distance from time of flight
}
```

Pseudocode 6

```
robot.getOdoN
{
  read OdoN counter;
  diff=counter-previous;
  if diff<0 diff=maxcounter-previous+counter;
  calculate traversed distance from counter value;
  previous=counter;
}
```

Pseudocode 7

to approximately 0.5 m/s (if the robot is not to bump into obstacles and considering that minimum detectable distance is set to 20 cm). This is more than enough for our application. Finally, note that in this application the interrupts are generated only by the serial port.

The design methodology described in Section 2.1 has driven the development of this application. We call:

getSonar1, getSonar2, ..., getSonar6 the periodic tasks which read the sensors with period $p_1 = p_2 = \cdots = p_6 =$ 500 milliseconds; updateMap a periodic task with period $p_7 = 500$ milliseconds; getOdo1, getOdo2 periodic tasks with period $p8 = p9 = 1200$ milliseconds; antiSensor a periodic task with period $p_{10} = 2000$ milliseconds.

Also, we choose the worst case task duration as $C_1 =$ 20 milliseconds ($B_1 =$ 500 milliseconds), then $B_2 =$ 480 milliseconds. Choosing $C_2 = 20$ milliseconds, then $B_3 =$ 460 milliseconds. Choosing $C_3 = 20$ milliseconds, then $B_4 =$ 440 milliseconds. Choosing $C_4 = 20$ milliseconds, then $B_5 =$ 420 milliseconds. Choosing $C_5 = 20$ milliseconds, then $B_6 =$ 400 milliseconds. Choosing $C_6 = 20$ milliseconds, then $B_7 =$ 380 milliseconds. Choosing $C_7 =$ 100 milliseconds, then $B_8 =$ 280 milliseconds. Choosing $C_8 =$ 20 milliseconds, then $B_9 = 500(1 - 220/500 - 20/1200) = 271$ milliseconds. Choosing $C_9 =$ 20 milliseconds, then $B_{10} = 500(1 - 220/500 - 40/1200) = 263$ milliseconds. Finally we choose $C_{10} = 20$ milliseconds.

As the condition holds, then the set of task can be scheduled using nonpreemptive scheduling. In Figure 7, the corresponding time diagram of the nonpreemptive scheduling designed for the map building application is reported. In black is shown the part of times which can be dedicated to the aperiodic tasks.

The pseudocode for sonar real-time tasks is shown in Pseudocode 6.

Two other real-time periodic tasks read the odometers: the odometer increments the counter register for each tick of the wheel encoder. The task reads the register and compares it with the previous reading to find a traversed distance. The wrapping of the counter when passing from maximum value to 0 is easily handled since max wheel speed is such that relative distance to previous reading is limited (see Pseudocode 7).

The real-time periodic process that updates the map reads the sensor readings stored in the internal circular list, calculates the robot's position, and updates the cells in the histogram grid corresponding to sensor readings. The most recent readings increment the histogram grid cells while the oldest ones decrement the cells, that is, only the last Nc readings compare in the map. In this way the portion of the map with revealed obstacles moves with the robot (see Pseudocode 8).

As opposed to the updateMap task a second real-time periodic task called antisensor decrements all cells in a specified area around a robot (an area in front of the robot with range 1.5 m in the implementation). This mechanism is used in order to eliminate moving obstacles and to correct erroneous sonar readings. The antisensor task is scheduled with much larger period than the map updating task allowing for the cells in which real obstacles are present to reach high values before they get decremented.

The map transfer task is a real-time aperiodic task scheduled when a request for a map arrives. The array containing the map is then compressed with a simple (value, count) scheme and transmitted to a serial port running at 9600 bps. The number of bytes to be transmitted does not grow with the array size due to the updating mechanism discussed earlier where only the last readings compare in the map (see Pseudocode 9).

```
robot.updateMap();
{
      getLastOdoReadings();          // the most recent odometry readings in circular list
      calculateRobotPosition();
      getLastSonarReadings(); // the most recent sonar readings
      calculateSonarReadings();      // polar to cartesian based on relative sensor position
      updateMapPositive();           // increments histogram map cells

      getFirstOdoReadings();         // the oldest odometry readings
      calculateRobotPosition();
      getFirstSonarReadings();       // the oldest sonar readings
      calculateSonarReadings();
      updateMapNegative();           // decrements histogram map cells
}
```

Pseudocode 8

```
robot.sendMap()
{
  compressMatrixData();  // the map stored in memory array
  transferMatrixData();  // is transmitted to serial port
}
```

Pseudocode 9

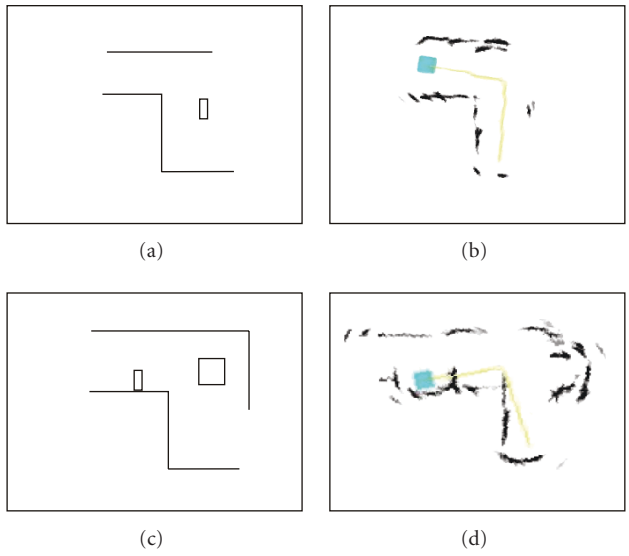

Figure 8: Map building results. On the left the real maps, on the right the maps estimated with the embedded system.

5.3. Experimental results

The bitmap estimated with the *updateMap()* method is referred to absolute coordinates and requires absolute localization results obtained with odometry. In other words, the map is an array of bytes where each element represents a cell of the grid by which the environment is divided. In our case, each cell represents a square of 50 mm long sides.

```
/*Components interfaces */
#include <TCB.h>    /* Thread Control Block */
#include <yartek.h> /* Scheduler */
#include <irq.h>    /* Interrupt management */
```

Algorithm 1

In Figure 8, some examples of maps obtained with the described embedded system are reported: on the left the actual map of the environment and on the right the perceived map obtained from the embedded device.

6. CONCLUDING REMARKS

In this paper, a real-time kernel we called Yartek, designed to implement embedded systems in mobile robotics, is described. It has been used to develop an embedded system for ultrasonic sensors-based environment mapping using the VFH algorithm. The embedded device we developed is currently used in our mobile robots: the ultrasonic sensors are used by the embedded system to compute an environmental map which is transferred to the robot without any other computational cost. The development of Yartek was motivated by the impossibility to find an open source real-time kernel for the Coldfire microcontroller used in our embedded system. Yartek is a small size kernel, which could be simply ported to other microcontroller architectures and implements nonpreemptive EDF scheduling. It also implements a deferred interrupt mechanism; by using this mechanism, interrupts are served by nonreal-time aperiodic tasks, which are scheduled in background.

Applications developed externally, that is, using cross-development systems, must be compiled inside the operating system. Therefore, a development environment must be available externally. Other features of the kernel include a thread management mechanism, dynamic memory management using first-fit, availability of a serial port driver which allows the connection of an external terminal for data

```
/* the TCB (Thread Control Block) type */
struct Tcb {char    SMI,        /* sending module */
            RMI,                /* receiving module */
            PR,                 /* task priority */
            PX,                 /* process index */
            DATA[4],            /* short data field */
            SR[2],              /* Cpu condition codes (SR) */
            *a[8],              /* Cpu address registers: A0–A7 */
            *d[8],              /* Cpu data registers: D0–D7 */
            *PC;                /* Cpu Program Counter */
     short  Stack,              /* Assigned Stack */
            Heap;               /* Assigned Heap */
     int    BornAbsTime,        /* Time of birth of process */
            Mem;                /* Memory address */
     char   ST_PR;              /* Starting task priority */
     void   (* fun)();          /* Pointer to the function to execute */
     unsigned int
            PID,PPID;           /* Process ID, Parent Process ID */
     CommandLine Command;       /* Record field for longer data */
     char   Name[NameDim];      /* Name of command or process */
     struct Tcb *CP;            /* Chain pointer */
     char   Type;
     unsigned int
            Born,               /* Born time */
            Start,              /* Next start time */
            Dline,              /* Deadline */
            Period,             /* Period of the thread */
            Maxtime,            /* Maximum duration of the thread */
            Stat;               /* Duration of the previous execution */
};
typedef struct Tcb TCB;
```

Algorithm 2

exchange and system monitoring purposes, and availability of a RAM-disk which provides a convenient data structure for temporary storage of information and for easily extending the operating system features.

Yartek has been developed for the Coldfire microcontroller family; if a porting to different architecture is to be performed, there are some short pieces of assembler code, mainly for the context switch and for timers management, which must be rewritten. It should be pointed out that Yartek could be simply modified for working both with preemptive and nonpreemptive scheduling. The main difference between nonpreemptive and preemptive scheduling is that in the preemptive modality some more operations should be performed to assure the preemption. In the preemptive case, another table is required to contain the TCB ID, the next activation time, and the task period. Before executing a TCB code, a timer is set in order to execute the scheduling when a periodic task arrives. The interrupt routine first updates this table, adding the period to the next activation time of the current task, it sorts the table by the next activation time in the ascending order, and then the scheduler is called.

In Appendix C, we report what should be done if one wants to adapt Yartek to another application.

The system we present in this paper is small, and it is highly reconfigurable. Almost all the kernel has been written in C language, and the source code can be freely downloaded [3]. The executable image takes less than 120 kbytes. We are currently developing other embedded systems for nonvisual sensors, namely inertial and infrared, for mobile robots.

APPENDICES

A. DATA STRUCTURES

In the following the Yartek data sructures are briefly summarized. To compile the application, a programmer needs to include headers of the required components (see Algorithm 1). The main data structure in Yartek is called TCB and is defined in Algorithm 2.

The Interrupt table is defined as follows:

```
int InterruptTable[3];          /* Interrupt Table */
```

B. EDF SCHEDULING

The scheduling algorithm of Yartek implements an EDF nonpreemptive scheduling. A representative C code of this routine for interrupt and real-time threads is reported in Algorithm 3.

```
void MainLoop()
{
   bool Found;
   register int i,  j;
   TCB *p, *p1, *p2;
mainloop:
      /* ServiceInterruptTable */
      /* deferred mechanism, nonreal-time threads are scheduled using Call */
      /* Call takes a TCB from the free list and put it on queue 2 */
      if (InterruptTable [0]! = 0)
      {
         Call( ServiceInterrupt0 );
      }
      if (InterruptTable [1]! = 0)
      {
         Call( ServiceInterrupt1 );
      }
      if (InterruptTable [2]! = 0)
      {
         Call( ServiceInterrupt2 );
      }

      /* ServiceTaskQueue */
      /* if a task to be activated is found on queue 1
         then concatenate task on queue 0 (using concatTCB function) */
      p=fipt[1];
        while (p!=NULL && p->Start<=RTClock)
      {
         fipt[1] = p->CP;    /* fipt: root of the queue 1 */
         if (fipt[1]==NULL) lipt[1]=NULL;
         p->PR = 0;
         p->ST_PR = 0;
         concatTCB(p);
         p = fipt[1];
      }
      if (p!=NULL)
      while (p->CP!=NULL)
         if (p->CP->Start<=RTClock)
         {
            p1=p->CP;
            p->CP=p1->CP;
            if (p->CP==NULL) lipt[1]=p;
            p1->PR = 0;
            p1->ST_PR = 0;
            concatTCB(p1);
         } else p=p->CP;
      if (Found==TRUE)
      {
       savedTCB=fipt[0];
      }
      if (savedTCB->CP == NULL)
         lipt[0] = NULL;    /* Updates the last element queue pointer */
      /* EXE execute the real-time thread */
      asm{
         jmp EXE;
      }
      /* When the real-time periodic thread has completed its period,
       then it is enqueued on queue 1 */
      concatTCB(savedTCB);
      goto mainloop;
}
```

Algorithm 3

C. HINTS FOR DEVELOPERS

A Yartek application developer should

(i) write the code for the interrupt service routines *ServiceInterrupt0*, *ServiceInterrupt1*, and *ServiceInterrupt2*;
(ii) write the *Init()* routine that schedules the periodic real-time threads;
(iii) write the code of the periodic real-time threads.

REFERENCES

[1] K. Lenac, E. Mumolo, and M. Nolich, "Fast genetic scan matching using corresponding point measurements in mobile robotics," in *Proceedings of the 9th European Workshop on Evolutionary Computation in Image Analysis and Signal Processing (EvoIASP '07)*, vol. 4448 of *Lecture Notes in Computer Science*, pp. 375–382, Valencia, Spain, April 2007.

[2] E. Mumolo, K. Lenac, and M. Nolich, "Spatial map building using fast texture analysis of rotating sonar sensor data for mobile robots," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 19, no. 1, pp. 1–20, 2005.

[3] Yartek, http://www.units.it/smartlab/yartek.html.

[4] E. Mumolo, M. Nolich, and M. OssNoser, "A hard real-time kernel for motorola microcontrollers," *Journal of Computing and Information Technology*, vol. 9, no. 3, pp. 247–252, 2001.

[5] K. Jeffay, D. F. Stanat, and C. U. Martel, "On non-preemptive scheduling of periodic and sporadic tasks," in *Proceedings of the 12th Real-Time Systems Symposium*, pp. 129–139, San Antonio, Tex, USA, December 1991.

[6] W. Li, K. Kavi, and R. Akl, "A non-preemptive scheduling algorithm for soft real-time systems," *Computers and Electrical Engineering*, vol. 33, no. 1, pp. 12–29, 2007.

[7] M. Garey and D. S. Johnson, *Computer and Intractability, A Guide to the Theory of NP-Completeness*, W.H. Freeman Company, San Francisco, Calif, USA, 1979.

[8] L. Georges, P. Muehlethaler, and N. Rivierre, "A few results on non-preemptive real-time scheduling," Research Report 3926, INRIA, Lille, France, 2000.

[9] Freescale, "MCF5282 ColdFire Microcontroller Users Manual," November 2004 http://www.freescale.com/files/32bit/doc/ref manual/MCF5282UM.pdf.

[10] K. Sacha, "Measuring the real-time operating system performance," in *Proceedings of the 7th Euromicro Workshop on Real-Time Systems*, pp. 34–40, Odense, Denmark, June 1995.

[11] A. Garcia-Martinez, J. Conde, and A. Vina, "A comprehensive approach in performance evaluation for modern real-time operating systems," in *Proceedings of the 22nd EUROMICRO Conference (EUROMICRO '96)*, pp. 61–68, Prague, Czech, September 1996.

[12] K. Baynes, C. Collins, E. Fiterman, et al., "The performance and energy consumption of embedded real-time operating systems," *IEEE Transactions on Computers*, vol. 52, no. 11, pp. 1454–1469, 2003.

[13] AVNET, http://www.em.avnet.com/.

[14] P. Mantegazza, "Diapm rtai for linux: why's, what's and how's," in *Proceedings of the Real Time Linux Workshop*, Vienna, Austria, November 1999.

[15] RTAI, http://www.rtai.org/.

[16] M. Silly-Chetto, T. Garcia-Fernandez, and A. Marchand, "Cleopatre: open-source operating system facilities for real-time embedded applications," *Journal of Computing and Information Technology*, vol. 15, pp. 131–142, 2007.

[17] "uClinux Embedded Linux Microcontroller Project," http://www.uclinux.org/.

[18] S. Thrun, "Robotic mapping: a survey," in *Exploring Artificial Intelligence in the New Millenium*, G. Lakemeyer and B. Nebel, Eds., Morgan Kaufmann, San Fransisco, Calif, USA, 2002.

[19] J. Crowley, "World modeling and position estimation for a mobile robot using ultra-sonic ranging," in *Proceedings of IEEE International Conference on Robotics and Automation*, vol. 2, pp. 674–681, Scottsdale, Ariz, USA, May 1989.

[20] J. J. Leonard, H. F. Durrant-Whyte, and I. J. Cox, "Dynamic map building for an autonomous mobile robot," *International Journal of Robotics Research*, vol. 11, no. 4, pp. 286–298, 1992.

[21] A. Elfes, "Occupancy grids: a stochastic spatial representation for active robot perception," in *Autonomous Mobile Robots: Perception, Mapping, and Navigation*, S. S. Iyengar and A. Elfes, Eds., vol. 1, pp. 60–71, IEEE Computer Society Press, Los Alamitos, Calif, USA, 1991.

[22] A. Elfes, "Dynamic control of robot perception using multi-property inference grids," in *Proceedings of IEEE International Conference on Robotics and Automation*, vol. 3, pp. 2561–2567, Nice, France, May 1992.

[23] H. P. Moravec and A. Elfes, "High resolution maps from wide angle sonar," in *Proceedings of IEEE International Conference on Robotics and Automation*, vol. 2, pp. 116–121, St. Louis, Mo, USA, March 1985.

[24] J. Borenstein and Y. Koren, "Histogramic in-motion mapping for mobile robot obstacle avoidance," *IEEE Transactions on Robotics and Automation*, vol. 7, no. 4, pp. 535–539, 1991.

[25] J. Borenstein and Y. Koren, "The vector field histogram-fast obstacle avoidance for mobile robots," *IEEE Transactions on Robotics and Automation*, vol. 7, no. 3, pp. 278–288, 1991.