Development of Embedded Devices in Real-Time Autonomous Robots

Kristijan Lenac, Enzo Mumolo, Massimiliano Nolich, Massimo Oss Noser

1 DEEI, Università degli Studi di Trieste, 34127 Trieste, Italy
2 A.I.B.S. Lab S.r.l., Via del Follatoio 12, Trieste, Italy
E-mail: klenac@units.it

Abstract. In this work a development of embedded systems suited for the implementation of robotic perceptions is described. The real-time applications are scheduled by a real-time micro-kernel, called YARTOS (Yet Another Real-Time Operating System), designed and developed for that, and running on the Motorola Coldfire micro-controller. In this paper the main features of such embedded system are illustrated and its performances evaluated using classical benchmarks are presented.

Keywords. Embedded system, autonomous robot, real-time system, performance evaluation, map building.

1 Introduction

Autonomous robots can perform desired tasks in unstructured environments without continuous human guidance. In order to operate in unstructured environments, autonomous robots should be equipped with a wide range of sensors and actuators. The raw sensorial data gathered from the sensors must be processed in order to obtain a diagrammatic representations of the perceived environment, therefore a lot of computational power is needed to manage such data. In order to alleviate such computational burden, we developed an embedded device running a real time micro-kernel written by us.

Embedded systems are composed by specialized hardware controlled by an operating system which doesn’t allow interactions with the user - from a programming point of view. Usually these systems are used for observing and controlling industrial devices where time constrains are imposed by physical laws. A system capable of performing the correct response within a fixed time (deadline) is needed in such cases.

In our robotic research, we are studying non-visual sensors which should be processed in real-time and which require high computational power, such as beamforming using microphone arrays, bias reduction in inertial sensors, or map building using sonar arrays. This problem can be solved using embedded devices with sufficient computational power to process such sensors. As a first step toward this goal, we have developed an embedded system that can be used for managing non-visual sensors in the robotic field. As a matter of fact, the time constants of such real-time applications are of several milliseconds. YARTOS was developed by modifying the scheduling module of another tiny operating system, described in [6].

To better explain the considered application, it can be noted that to perform tasks avoiding collisions with obstacles, a mobile robot needs to build an environmental map. Ultrasonic sensor arrays are the low cost sensors of choice for perceiving a map suitable for navigation. To manage an array with a number of easily modifiable sensors and to avoid the overloading of the mobile robot processing device, a real-time embedded system has been developed. It is composed of a micro-controller board equipped with a sonar array. A real-time software environment for managing this type of equipment has been developed and is presented in this paper, together with an evaluation of the real-time system performances.

Our aim was to develop a platform suited for implementing small and light real time applications composed of several tasks: they can be both real-time periodic and real-time aperiodic. Real time task scheduling is per-
formed using EDF (Early Deadline First) algorithm for periodic tasks and TBS (Total Bandwidth Server) for aperiodic tasks as presented in [2]. Moreover the system should be capable of scheduling non real-time tasks if needed. The contribution of this work is the presentation of an embedded device for developing robotic embedded applications that is small and that can be simply reconfigured in order to manage different sort of non-visual sensors.

2 Related work

Generally speaking, the design of systems that respond promptly to external events is interesting for many and diverse applications. Plant regulation, process control, data acquisition, automation systems, virtual reality and robotics, to mention some, require resolution methods that are aware of intrinsic response-time constraints. The solutions in this case yield results whose value does not depend only on the correctness of the performed operations, but also on the moment in time when the results became available.

Systems capable of sensing the external events, processing the collected data and applying the appropriate responses during the actual occurrence and progress of the phenomenon are called real time operating systems [5]. When the time constraints cannot be missed they are referred to as hard real time operating systems, on the other hand if a delay is admissible they are referred to as soft real time. The result which follows the request is more or less useful depending on the time when it becomes available. The usefulness increases rapidly to a maximum time threshold and decreases afterwards more or less rapidly. In hard real time systems the usefulness becomes zero immediately or, even worse, becomes negative as to indicate a damage.

Some frameworks for considering network of real time sensors has been proposed [4].

3 The Real-Time Operating System

The real-time kernel, called YARTOS (acronym for Yet Another Real Time Operating System), was developed for the 683xx family [6] and it has been rewritten for the Coldfire Microcontroller family. The system has been tested on the AvNet mcf5282 board shown in Fig. 1.

![Figure 1: The Coldfire Microcontroller](image)

3.1 YARTOS outlines

The system uses two timers, one for controlling the time slicing and one for the real-time processing, called RTClock; one serial port is also managed, in order to transmit data to other devices. When the system detects an interrupt coming from the serial port, an aperiodic task is scheduled; external interrupts are not served directly but rather through an Interrupt Table which is analyzed and served by the main scheduler.

Yartos allows the creation and running of threads for fast context switching and doesn’t use virtual memory; rather it offers a dynamic memory management using a first-fit criterion. Real-time tasks can be periodic or aperiodic, scheduled with EDF [3] and Total Bandwith Server [7] respectively; non real-time tasks are managed using a priority based criterion. Synchronization tools based on semaphores have also been provided.

In order to improve the versatility of the system, a Ram-disk has been added: it is actually an array defined in main memory and managed using pointers, therefore its operation is very fast. The Ram-disk offers a suitable structure for storing temporary data and executable code which enriches the amount of real-time processes which the kernel can run.
The structure of the kernel is reported in fig. 2.

Figure 2: Internal structure of Yartos.

### 3.2 Concurrent processes

A process is formed by three regions: code, data and stack. A set of system calls for thread management is provided; a thread in YARTOS is basically formed by a stack and a Program Counter. Concurrency is performed using a time-sharing approach, where the threads, which are known at the kernel level, are scheduled on the basis of a Round Robin mechanism at different priorities. Threads in YARTOS are described by a data structure, called Thread Control Block (TCB).

Relevant fields stored in TCB are `start` and `dline`, which are the starting time and the deadline of a real-time task, `period` which is the period of a periodic real-time task, `max-time` and `time` which are the computing time that the task employed in the previous execution slice.

### 3.3 Memory management

An amount of stack and data memory, containing process-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.

- Code and data memory. The data and code regions are allocated contiguously. The data and code allocation mechanism allows only external fragmentation.
- Stack and heap. The stack is organized with a LIFO policy. Every TCB has a stack 32Kb long. A heap area, which is a linked list of 8Kb memory blocks, is needed by the `alloc()` system call. Every process has a pointer to the heap area, and every heap area has a pointer to the next area.
- Ram-disk. It is structured as a flat directory which includes all the files. The virtual disk is formed by 2048 blocks of 512 bytes each. A file is formed by a number of memory blocks allocated contiguously. A directory entry exists, which is basically a table which associates a file name with a pointer to memory and the size reserved for the file.

### 3.4 System calls

A number of system calls has 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.

### 3.5 Scheduling management

Scheduling is managed with a linked list of TCB’s at 4 priority levels. The highest priority queue is the queue number 0, and the lowest priority queue is the queue number 3. The TCBs of real-time tasks are stored in the highest priority queue, while those waiting for execution in the queue number 1.

The scheduling strategy adopted in YARTOS is the Total Bandwidth Server (TBS) [7]. It allows management of both periodic and aperiodic real-time tasks while reducing the response time of aperiodic tasks. The idea of TBS is to compute the earliest possible deadline of aperiodic tasks, taking into account schedulability constraints,
and giving to aperiodic tasks all the available bandwidth as soon as it is requested. Once the deadline of aperiodic tasks is computed, all the tasks, periodic and aperiodic, are scheduled using EDF. The deadlines of aperiodic tasks computed as described above yield a low response time. Calling \( \text{start} \) the first activation instant of a periodic task and \( \text{period} \) its period, the deadline of that task is \( d_{\text{line}} = \text{start} + \text{period} \). The deadline of aperiodic tasks is instead computed as

\[
d(k) = \max[r(k), d(k-1)] + \frac{C(k)}{\mu_{TBS}}
\]  

(1)

where \( r(k) \) is the arrival time of the \( k \)-th aperiodic task, \( d(k-1) \) is the deadline of the \( (k-1) \)-th aperiodic task, \( C(k) \) is the execution time of the \( k \)-th task and \( \mu_{TBS} \) is the server utilization factor, i.e. its bandwidth (\( \mu_{TBS} = 0.1 \) in our implementation). In the actual code the deadline of an aperiodic task is computed using: \( \text{dline} = \max(\text{RTC\!clock, last\!deadline}) + \text{maxtime}/\mu_{TBS} \) where maxtime is the maximum time needed to complete a generic task. The EDF schedulability of \( N \) tasks - both periodic and aperiodic - is assured as usual; calling \( C(k), P(k) \) the execution time and the period of the \( k \)-th task, the following condition must hold:

\[
\sum C(k) / P(k) \leq 1.
\]  

(2)

As reported in [7], the following theorem holds.

**Theorem.** Given a set of \( n \) periodic tasks with processor utilization \( \mu_P \) and a TBS with processor utilization \( \mu_{TBS} \), the whole set of tasks is schedulable if and only if \( \mu_P + \mu_{TBS} \leq 1 \).

A number of low-level procedures are used to perform scheduling; the most representative routines are summarized in the following. The entry point of the kernel is an infinite loop where the interrupt table and the task queue are examined, as described in the following pseudocode:

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

The \( \text{ServiceInterruptTable} \) routine verifies if an interrupt is pending, and in this case it activates the suitable module for serving that interrupt.

If there are no interrupts to serve, the task queue is analyzed to select the task to be scheduled; if there are no tasks in queue 0, an aging operation is performed and a non real-time task is selected. Clearly, a test to determine whether the task has already been executed or is a new one is performed.

During aging operations, the non real-time tasks which wait more than a given value are moved to a higher level queue to give them a chance to be scheduled.

The movement of a TCB from a queue to another at a given priority level is performed with a procedure called \( \text{ConcatTCB} \), which inserts the task itself in the task queue. If the task is real-time, the task is inserted in the highest priority queue, and a sorting operation is performed on the task deadlines.

A real-time process is activated with the low-level routine \( \text{CallRTT} \), described as follows:

```c
CallRTT()
{
    Extract a TCB;
    if(process is real-time periodic){
        if(uCPU >= 0.9)
            return(error);
    }
    else if(process is real-time aperiodic) {
        compute the deadline with TBS;
        if(task termination exceeds deadline)
            return(error);
    } else
        if(there are no errors)
            create & insert a TCB in the Task Queue;
}
```

### 4 Performance evaluation

The performance of the system has been tested using the following parameters:

**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. In Tab. 1 the mean and variance of the jitter time obtained in three different load conditions are reported. The variance of this parameter is nearly constant under different load conditions.
Table 1: Jitter time.

<table>
<thead>
<tr>
<th>Mean [ms]</th>
<th>Variance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.09</td>
<td>0.27</td>
</tr>
<tr>
<td>1.36</td>
<td>0.26</td>
</tr>
<tr>
<td>0.87</td>
<td>0.27</td>
</tr>
</tbody>
</table>

**latency time:** it is the time delay between an interrupt event and the execution of the first instruction of its service routine. In Tab. 2 the values measured using a timer as interrupt generator are reported. Besides the comments on the measured variance reported for the jitter time parameter, it is worth recalling that the variance can be used to estimate confidence intervals.

<table>
<thead>
<tr>
<th>Mean [ms]</th>
<th>Variance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.73</td>
<td>0.69</td>
</tr>
<tr>
<td>1.66</td>
<td>0.69</td>
</tr>
<tr>
<td>1.67</td>
<td>0.69</td>
</tr>
</tbody>
</table>

Table 2: Latency time.

**context switch time:** it is the time spent during the execution of the context switch routine of the scheduler. The results have been obtained forcing a process to perform a context switch (using a system call) and the results are reported in Tab. 3. As expected, the context switch time does not depend on the number of iterations.

<table>
<thead>
<tr>
<th>Nr. of iteration</th>
<th>Mean time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>0.08</td>
</tr>
<tr>
<td>30000</td>
<td>0.08</td>
</tr>
<tr>
<td>50000</td>
<td>0.08</td>
</tr>
</tbody>
</table>

Table 3: Context switch time.

5 Case study: map building application

In mobile robotics several tasks require the strict satisfaction of time constraints, so real-time systems are needed. In this work we have developed a small and simply reconfigurable real-time embedded system capable of operating when the time constant required by the sensors processed is of the order of several milliseconds. The sensors generally used for mobile robot navigation, such as inertial navigation systems, sonar sensor array, GPS, laser beacons, should be processed considering real time constraints. As map building is a fundamental task in mobile robotics, an application of this type has been designed and the main outlines of its implementation using YARTOS on the embedded system are described in the following.

5.1 Map building application

If a mobile robot has to perform a navigation task, an updated map is needed to plan the navigation task avoiding collisions with obstacles. YARTOS has been installed on an embedded system composed by the Coldfire micro-controller board connected to an array of 3 sonar sensors placed in front of a mobile robot. The map building task is a good test environment as it requires real-time constraints in order to avoid collisions.

The application is based on the Elfes algorithm [1] and its operation can be summarized as in Fig. 3.

![Figure 3: Scheduling of real-time map building tasks. P1, P2 and P3 are sensor acquisition tasks; P4 is an Update map task, and A1 is an aperiodic task requiring map transfer.](image)

The application is structured as follows:

- Sensor acquisition tasks. There are 6 real-time tasks that perform the acquisition of the data from the sonar sensor.
- Update map. There is a real-time task that updates the map according to the
acquired sensorial data provided by the sensor acquisition task.

- Map transfer. There is an aperiodic task that allows to obtain the complete map from the embedded system using serial line connection.

This structure is simple to reconfigure: adding one more sensor requires the scheduling of a new real-time acquisition task and the incrementing the variable which keep track of the number of sensors in the array. It is also really simple to decide whether a new task can be scheduled according to the given time constraints.

PERIODIC TASK:
robot.run();
{
    robot.getX();
    robot.getY();
    robot.getTh();
    // odometer readings
    robot.getSonar();
    // sonar sensor readings
    robot.updateMap();
    // update map (memory array)
}

APERIODIC TASK:
robot.sendMap() {
    transferMatrixData()
    // the map stored in memory array
    // is transmitted to serial port
}

In Fig. 4 some examples of maps obtained with the embedded devices 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

A hard real-time kernel, which has been developed by the authors for running real-time sensor measurement applications with time constants of several milliseconds, has been highlighted in this paper. The kernel was designed to execute periodic and aperiodic real-time applications as well as non real-time tasks. Therefore, a development environment must be available externally. Applications developed externally, i.e. using cross-development systems, must be compiled inside the operating system.

Some features of the kernel include an interrupt processing using interrupt tables, 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 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. The system is small and it is highly reconfigurable. In particular, if it is assumed to consider only real-time tasks, the scheduler can be modified and the context switch can be avoided. Almost all the kernel has been written using 6350 lines of C-language; the executable image takes less than 120 kilobytes.

References


