A Hard Real-Time Kernel for Motorola Microcontrollers

Enzo Mumolo, Massimiliano Nolich and Massimo Oss Noser

DEEI, University of Trieste, Italy

This paper describes a real-time kernel for running embedded applications on a recent family of Motorola microcontrollers. Both periodic and aperiodic real-time tasks are managed, as well as non real-time tasks. The kernel has been called Yartos, and uses a hard real-time scheduling algorithm based on an EDF approach for the periodic task; aperiodic tasks are executed with a Total Bandwith Server.

Keywords: real-time operating system, embedded system, microcontroller

1. Introduction

Embedded systems are formed by a specialized hardware controlled by a real-time operating system which doesn't allow interactions with the user — from a programming point of view.

Motorola offers a family of microcontrollers, the MC683xx, which combines a stripped-down 68020 core with a 16-bit on-chip intermodule bus, which links the cpu with peripherals. The core processor is a 68020 cpu for embedded control that lacks memory-management unit (mmu) or floating-point-unit (fpu) interfaces; its block diagram is reported in Fig. 1.

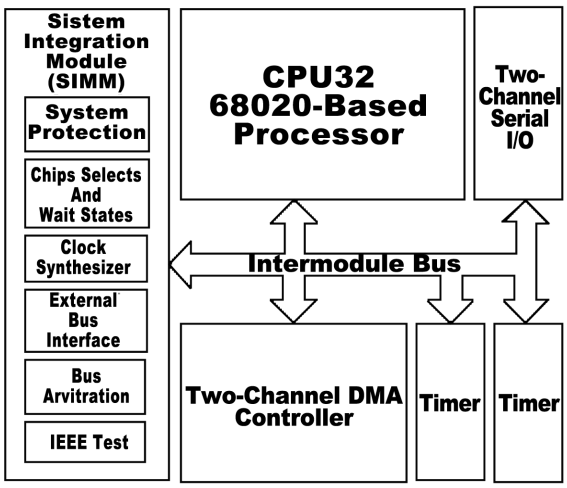

Fig. 1. Block diagram of the 683xx family.

A fundamental issue in embedded application development is thus related to the real-time operating system needed for running embedded applications on these processors. We faced this problem, in fact, during the development of a research project in robotics [4], which was constrained by the necessity to use open source components. Instead of looking for a software suited to our needs, we developed a real-time kernel on our own. We think that the developed system has some interesting features which could make it useful in other embedded applications; thus we decided to share it as an open source.

The contribution of this paper is therefore to highlight some characteristics of our real-time kernel, which we called Yartos (acronym for Yet Another Real Time Operating System) which was developed for the 683xx family. The system uses two timers, one for controlling the time slicing and the other for the real-time processing, called RTClock; one serial port is also managed. An external terminal can be connected to the serial port in order to perform some monitoring tasks, such as analysis of log-files or to run debugging tools. When the system detects an interrupt coming from the serial port, a shell command interpreter is activated; it should be pointed out therefore that in Yartos the shell is not a normal non real-time task, but rather it is an

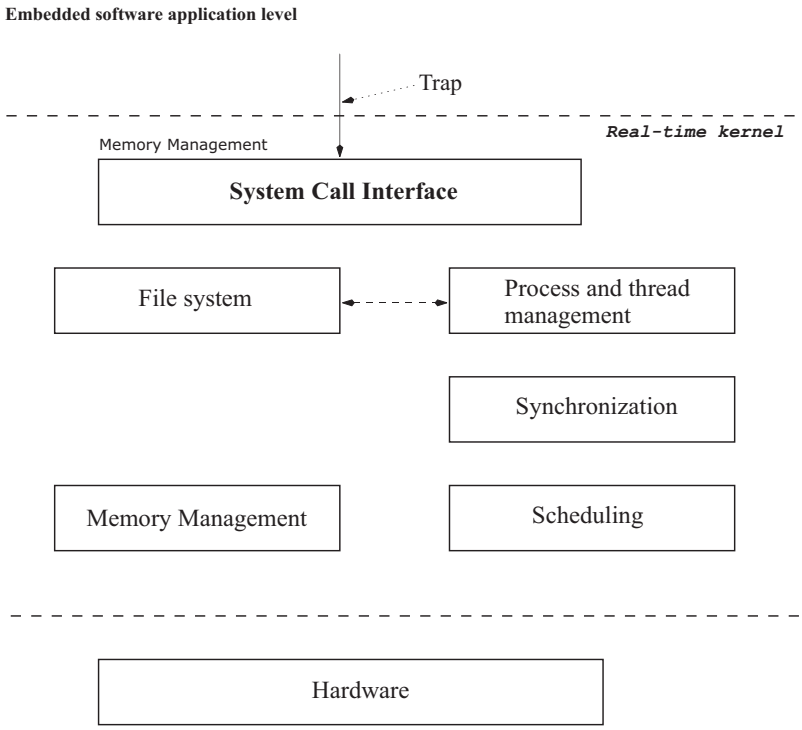

Fig. 2 a) Internal structure of Yartos.

interrupt service routine. It is important to note that external interrupts are not serviced directly but rather through an Interrupt Table which is analyzed and served by the main scheduler.

Moreover, Yartos allows the creation and running of threads for a faster context switch 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 [5] and Total Bandwith Server [1] respectively; non real-time tasks are managed using a priority-based criterion. Synchronization tools based on semaphores have been also developed.

In order to improve versatility of the system, a Ram-disk has been added. The Ram-disk is actually an array defined in the 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 a), and the state diagram of processes in Yartos is shown in Fig. 2 b).

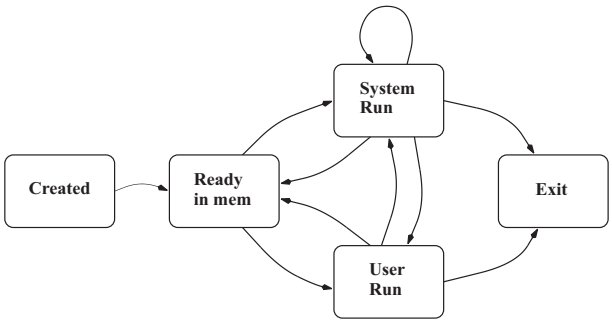

Fig. 2 b) States of a process in Yartos.

2. Concurrent Processes in Yartos

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), which is defined as follows:

```
struct TCB{
   c|har type;               //proces|s type: periodic real-time, aperiodic real-time, non real-time
   int start, dline, period, maxtime, time;
   c|har SMI, RMI, PR, PX, Data[4], SR[2], *a[8], *d[8], *PC;
   short Stack, heap;        //stack and heap IDs
   int BornTime, Mem;        //starting time, pointer to memory area
   c|har St_pr;              //starting priority
   void (*fun)();            //the entry point of the thread
   unsigned int PID, PPID;
   CommandLine Name[Dim];
   struct TCB *CP; //pointer for the linked TCB structure
}
```

A TCB is generated by one process (identified by SMI) and the process the thread belongs to is identified by RMI. The PR field of TCB is the priority, and SR is the status register. Address and Data registers of the microcontroller are referred to by the pointers *a and *d, and PC is the Program Counter.

Other relevant fields 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, maxtime and time which are the computing time that the task employed in the previous execution slice.

Processes are normally embedded in the code of the kernel. However, as suggested in the Introduction, there is also the possibility to have a process allocated in the Ram-disk, which is uploaded from an external system where the application cross-development takes place. Hence, it is possible to dynamically change the number of processes which run in Yartos.

2.1. Memory Management

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

- Code and data memory. The data and code regions are allocated contiguously. If the process is not embedded in the kernel, but comes from an executable file located in the Ram-disk, an amount of memory equal to the file size is allocated after the data region and the code is loaded into it. The data and code allocation mechanism allows only external fragmentation.
- Stack and heap. The stack is organized with a LIFO policy. The access is performed using the A7 register of the microcontroller. 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 that file.

2.2. System Calls

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

2.3. Process Creation/Termination

New processes are created using the exec system call. Basically creating a new process means that a new TCB must be selected, filled up with the related information including PID, and put in the task queue. A task can be terminated using the kill system call. If many threads run in the context of a process, all the threads are

```
void down(bool *sem)
{
#asm
     movea.l (sp),a0
wait:
     tas (a0)        //test and set the semaphore
     beq end         //exit from the loop in case the semaphore was up
     trap #14        //this exception goes to the  main loop of the kernel
     bra wait        //loop if the semaphore was down
end:
#endasm
}
```

terminated. By addressing the threads by name, it is also possible to terminate a selected thread.

2.4. Process Syncronization

Synchronization in concurrent processing could occur when there is a simultaneous usage of resource which cannot be shared. In Yartos this problem is solved using binary semaphores implemented according to the tas instruction; the code for semaphore management is therefore written in assembler. In the example above the code used for implementing the down semaphore primitive is reported.

3. Scheduling Management

Scheduling is managed with the structure represented in Fig. 3, which is a linked list of TCB's at 32 priority levels. The highest priority queue is the queue number 0, and the lowest priority queue is the 31-st. In the highest priority queue the TCBs of real-time tasks are stored, while in queue nr.1 there are the tasks waiting for execution.

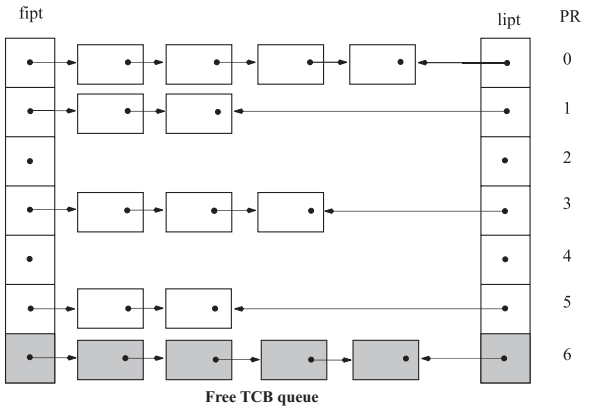

Fig. 3. Data structure for scheduling.

The scheduling strategy adopted in Yartos is the Total Bandwidth Server (TBS) [1]. 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 contraints, and to give 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 the first activation instant of aperiodic task *start* and its period *period*, the deadline of that task is $dline = start + period$. The deadline of aperiodic tasks is instead computed as

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

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 μTBS is the server utilization factor, i.e. its bandwidth. In the actual code the deadline of an aperiodic task is computed using: $dline = max(RTClock, lastdeadline) + maxtime/\mu TBS$ 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:

$$\sum \frac{C(k)}{P(k)} \leq 0.9 \quad (2)$$

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:

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

The 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 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 by a procedure called 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.

When a task terminates, the resources are normally released. If the real-time task is periodic, however, some further processing is needed, to prepare the task for successive executions. The procedure, FreeMod, is described in the following pseudocode:

```
FreeMod()
{
    Release data, code, heap and stack areas;
    if(process is real-time periodic)
    {
        start += period;         //update the starting time instant
        dline += period;         //update the deadline
        move the process's TCB in queue #1; //wait for activation
    } else
    if(process is real-time aperiodic)
    {
        update uCPU;
        release the process's TCB;
    } else                       //non real-time processes
    {
        release the process's TCB;
    }
    GoTo MainLoop;
}
```

A real-time process is activated by the low-level routine CallRTT, described as follows:

```
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. Concluding Remarks

A hard real-time kernel, developed by the authors for running real-time robotics applications, has been highlighted in this paper. The kernel was designed only 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 downloaded into the embedded system.

Some features of the kernel include an interrupt processing by means of 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 debugging and

system monitoring purposes, and availability of a Ram-disk which provides a convenient data structure for temporary storage of information and for an easy extension of the kernel features. Almost the whole kernel has been written using 4590 lines of C-language; the executable image takes less than 70 Kbytes. The code of Yartos is available at [2]. Two activities are currently underway: a porting of Yartos on Intel processors and the development of suitable tools for code development, such as editors, compilers and assemblers to give Yartos general purpose capabilities.

Acknowledgments

This work was partially funded by MURST 40%, Certamen project.

References

[1] M. SPURI AND G. C. BUTTAZZO, "Efficient Aperiodic Service Under Earliest Deadline Scheduling", *Proc. Of 15th IEEE Real-Time System Symposium,* 1994., Puerto Rico.

[2] *Yartos Home Page*, `ftp://serving4.univ.trieste.it/arc_stud/mumolo/sistemi_operativi/yartos`.

[3] P. MCKEE, "68000 Assembler user manual", `www.elec.qmw.ac.uk/staffinfo/eric/courses/mpe/asm.txt2`.

[4] ENZO MUMOLO AND MASSIMILIANO NOLICH AND GIANNI VERCELLI, "Algorithms and Architecture for Acoustic Localization based on Microphone Array in Service Robotics", *International Conference on Robotics and Automation*, **3**, pp. 2966–2971, San Francisco, April 2000.

[5] C. L. LIU AND J. W. LAYLAND, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", *Journal of ACM*, no. 1, vol. **20**, Januarry 1973.

Received: June, 2001
Accepted: September, 2001

Contact address:
Enzo Mumolo
DEEI, University of Trieste
Via Valerio 10
34127 Trieste, Italy
Phone: +39.040.6763861
Fax: +39.040.6763460
e-mail: `mumolo@univ.trieste.it`

Massimiliano Nolich
e-mail: `nolic|h@smartlab.univ.trieste.it`

ENZO MUMOLO received a Dr Eng degree (magna cum laude) in electrical engineering from the University of Trieste, Italy, in 1982, where he conducted research into signal processing algorithms before joining in 1984 the Central Laboratory of Alcatel Italia, FACE Division, formerly FACE Res. Center, in Pomezia, Rome, Italy. In 1985 he was with ITT DCD-West in S. Diego, CA. In 1987 he become responsible for research activities within the Speech Processing Dept. of FACE RC. From 1990 to 1991 he was with Sincrotrone Trieste, Trieste, Italy, as head of the Electronics Group. In 1991 he joined the Computer Science Dept. at DEEI, University of Trieste, as research engineer and assistant professor. His current research interests include nonlinear systems, adaptive filtering, operating systems and speech processing. Member of IEEE, ACM and AEI, he has published more than 80 papers in professional journals and international conferences and holds two United States Patents.

MASSIMILIANO NOLICH was born in Trieste, Italy, in 1971. After graduating in electronic engineering with magna cum laude at the University of Trieste in 1999, he enrolled in a PhD program, at the same University, working on algorithms and software techniques for robotic platforms. His research interests include intelligent autonomous systems, operating systems, software engineering and embedded systems. Ing. Nolich is an IEEE member and has published almost 10 papers in international conferences.

MASSIMO OSS NOSER was born in Venezia, Italy, in 1974. He is currently a student at the EECS Dept. of the University of Trieste.