Journal of Applied Sciences

ISSN 1812-5654

science alert

ANSInet
an open access publisher
http://ansinet.com

Journal of Applied Sciences 8 (18): 3083-3095, 2008
ISSN 1812-5654

Development of Real Time Multitask Kernel

B. Sahli and A. Bouaza
Ibn Khaldoun University of Tiaret, BP 78, Route Zaaroura, 14000 Tiaret, Algeria

Abstract: This study describes real-time kernel essential mechanisms and deals with the implementation of a real-time multitasks executive. To make more advantage of microprocessors in applications involving many functions simultaneously, our real-time kernel provides a multiprogramming environment in which many independent multitasking application programmes may execute and provides facilities to manage efficiently the processes (tasks) and communicate between them. These facilities are provided by system calls that handle data structures namely tasks, semaphores, messages, events flag, resources, mail-boxes, queues and interruptions. Present kernel is preemptive and priorities assigned to tasks are dynamic, the kernel manages up to 63 task levels (63 is the lowest priority level assigned to the idle task). Round robin scheduling is not used here.

Key words: Real-time kernel, data structures, resources, communication, synchronization

INTRODUCTION

Software development tools are programs that help developers create other programs and automate mundane operations while bringing the level of abstraction closer to the application engineer. In practice, software development tools have been in wide use among safety-critical system developers. Typical application areas include space, aviation, automotive, nuclear, railroad, medical and military (Kornecki and Zalewski, 2005; Beccari *et al*., 2005).

Emerging parallel or distributed, heterogeneous real-time computer systems with many disparate constraints and requirements would benefit from a unifying and comprehensive systems engineering support in the design, testing and deployment phases, which interfaces with a human at a very high level and efficiently handles the system complexity (Baki'c and Mutka, 2005).

Industrial control applications are usually developed in two phases: control design and real-time system implementation. In the control design stage a regulator is obtained and later it is translated into an algorithm in the implementation phase. Traditionally, these two phases have been developed in separate ways (Balbastre *et al*., 2004).

The design of an industrial real-time application always presents a certain specific number of difficulties. If consequent progress was made on the level of the design methods/specification, the passage to the detailed design remains a current problem. Such an application is in direct catch with the physical world. On the design level, the application division in two sub-systems became current (Harel and Pnueli, 1989) thus distinguishing the transformational systems (tactical level of decision or supervision) from the reactive systems (level reflex in contact with the physical devices). The first produce information starting from other information. The seconds react to events resulting from the physical environment associated to them and have, in theory, the critical time constraints most severe (Fohler and Buttazzo, 2002; Real and Crespo, 2004; Racu *et al*., 2007). The programming of such applications rests then primarily on a multi-task data-processing basis that it is mono or multiprocessors (Baker, 2006) and the tools available are those introduced to describe the parallel treatments in opposition to the sequential treatments. They are on the one hand numerical models of the operations partially or completely parallels and on the other hand the computer tools (operating systems and/or languages) facilitating the implementation of such treatments. For the applications of average complexity, which are also most current, one of the tools most widespread is the executive multitasks real-time. For the strictly reactive parts of a system, the use of synchronous languages can prove to be useful.

In this study, we will exclusively devote our attention to the implementation of an executive multitasks real-time.

The increasing complexity of real-time distributed applications demands the use of more sophisticated and diversified communication services. For instance, communication services that enforce strong consistency properties such as ordering and agreement at the communication are used in many infrastructures to simplify the development of fault tolerance application constraints. On the other hand, the necessity to provide a diversified set of properties to applications demands an increased flexibility in communication systems

Corresponding Author: B. Sahli, Ibn Khaldoun University of Tiaret, BP 78, Route Zaaroura, 14000 Tiaret, Algeria

(Rodrigues *et al*., 2007). The role of this one is to ensure various tasks coherent scheduling of an application and to place at the disposal of the programmer the essential elements to manage the communication/synchronization and mutual exclusion functions. They make it possible to cover, on a more or less raised level, the needs for indication, synchronization, communication and mutual exclusion, using objects such as tasks, events, mail-boxes, messages, semaphores, easy to handle via a set of primitives.

Independent environments flexibility allows the application programmers to manage separately each application resource during development time and test phases. Resources include processor scheduling, interruption resources and I/O devices (Abeni and Buttazzo, 2004). Managing each of these system resources and the way of sharing them will be discussed further (Hamidzadeh *et al*., 1999; Liu *et al*., 2007).

The multi-task kernel real-time, associated or not with a complete system with file management are currently the engines of applications in fields as varied as automation, robotics, medical electronics, process control, instrumentation, telecommunication and video.

For questions of portability, the major part of the kernel is written in C language with a specific code of the target microprocessor written in assembly language, kept at least.

KERNEL STRUCTURES

This kernel can manage more than 60 tasks. The stack size can be specified independently for each task, reducing the necessary quantity of RAM. The kernel is with ordering of priority and always executes the Higher Priority Task (HPT), it is also completely preemptive. The interruptions can suspend the execution of a task and if a HPT is waked up, following the interruption, the task at higher priority will be treated as of the completion of the interruption. The interruptions can be overlapping until a depth of 255 levels.

Critical sections: When the kernel is in treatment to update the critical data, it must guarantee not to be interrupted to avoid deteriorating the data. To guarantee this, the kernel has the exclusive access to the critical section of the code; it disables interruptions before executing this code and enables them when it is done. Macros OS_ENTER_CRITICAL () and OS_EXIT_CRITICAL ( ), which are specific to the target microprocessor, are used to disable and enable interruptions.

Interruptions are never disabled for more than 500 CPU clock cycles. An additional clock with 200 cycles is necessary to save the processor context and to notify with the kernel that an interruption was treated. The interruption response is thus around 700 clock cycles of the CPU. The execution times of all the kernel services are deterministic.

Tasks: A task is a function with infinite loop, or a function which autodestruct when it is executed. The infinite loop can, possibly, being preempted by an interruption which will allow a higher priority task to be executed. The task can also call upon one of the following kernel services: OSTaskDel ( ), OSTimeDly ( ), OSSemPend ( ), OSMbxPend ( ) or OSQPend ( ).
A task can thus be declared as follows:

```
Void far Task(void *data)
{
 User code;
 OSTaskDel(task's priority);
}
Or
Void far Task(void *data)
{
 While (1) {
      Optional user code;
      Call Kernel service to DELAY or PEND;
      Optional user code;
 }
}
```

The application can have up to 63 of these functions. With each task only one level of priority from 0 to 63 is assigned. The lower priority level corresponds to the highest priority of the task (the most important). The priority number is also used to identify the task. The priority number is used by the kernel services OSTaskChangePrio ( ) and OSTaskDel ( ).

Task states: The SLEEPING state corresponds to a task which lies in the EPROM, but is not put of availability for the kernel. A task is put of availability for the kernel by call of OSTaskCreate ( ). When a task is created, it is put READY for execution. Tasks can be created before the beginning of multitasks, or dynamically by a task in execution. If created task has a higher priority than its creative task, the control of the CPU is given immediately to the task created. A task can even turn over by it or another task to the state sleeping by call of OSTaskDel ( ).

Multitasks begin with OSStart ( ) which executes created HPT. This task is then placed in the EXECUTION state. The task in execution can be delayed for a certain time by call of OSTimeDly ().

This task is then placed in the DELAYED state and the HPT (higher priority task) following takes the control of the CPU immediately. The delayed task is put at the READY state for execution by OSTimeTick ( ) when the desired delay expires. The task in execution needs also to await the occurrence of an event, while calling is OSSemPend ( ), OSMbxPend or OSQpend ( ). The task is then on standby of an event. As a task awaits an event, the following HPT takes the control of the CPU immediately. The task is put at the ready state when the event arrives. The occurrence of an event can be announced either by another task or an ISR. A task in execution can always be INTERRUPTED, unless it disables the interruptions. When an interruption occurs, the execution of the task is suspended and the ISR takes the control of the CPU. The ISR can put one or more tasks at the state ready to execute, by announcing one or more events. In this case, before returning from ISR, the kernel determines if the interrupted task is always the HPT ready for execution. If a new HPT is put at the ready state to execute by the ISR, then it is taken again. Otherwise, the interrupted task begins again.

When all tasks are, that is to say on standby for events, or delayed for a number of clock tops, the kernel execute the idle task OSTaskIdle ( ).

Task control blocks: A task control block is assigned to a task when it is created: OS_TCB, which is used by the kernel to maintain the state of a task when it is preempted. When a task takes again the control of the CPU, the OS_TCB makes it possible the task to take again the execution exactly where it left. A kernal task control bolck is shown as:

```
typdef struct os_tcb {
 void            *OSTCBStkPtr;
 UBYTE           OSTCBStat;
 UBYTE           OSTCBPrio;
 UWORD           OSTCBdLY;
 UBYTE           OSTCBX;
 UBYTE           OSTCBY;

 UBYTE           OSTCBBitX;

 UBYTE           OSTCBBitY;

 struct os_tcb   *OSTCBNext;

 struct os_tcb   *OSTCBPrev;
 OS_EVENT        *OSTCBEventPtr;
}
```

Here a description of each field in the data structure of the OS_TCB:

OSTCBStkPtr: It contains a stack pointer of the task. The kernel makes it possible each task to have its own stack. OSTCBStkPtr is the only field in the data structure OS_TCB reached by the assembly language (of context commutation code).

OSTCBStat: It contains task state. When OSTCBStat is 0, the task is ready to execute. Other values can be assigned in OSTCBStat, these values are described in KERNEL.C appendix.

OSTCBPrio: It contains the task priority. A HPT has a low value (smallest is the number, highest is the current priority).

OSTCBDly: It is used when a task needs to be delayed for a number of clock ticks, or if it needs to await the arrival of an event, with a timeout. In this case, the field contains the clock ticks number for which the task is authorized to await the occurrence of an event. When this value is zero, the task is not delayed; there is no timeout. OSTCBX, OSTCBY, OSTCBBitX and OSTCBBitY: are used to accelerate the process to put the task at the ready state to execute. The values of these fields are calculated when the task is created, or when the priority of the task is changed as follows:

```
OSTCBX      = priority and 0×07;
OSTCBBitX   = OSMapTbl [priority and 0x07];
OSTCBY      =  priority >> 3;
OSTCBBitY   = OSMapTbl [priority >> 3];
```

OSTCBNext and OSTCBPrev: They are used to link the OS_TCB doubly. This chain of the OS_TCB is used by OSTimeTick ( ) to update the OSTCBDly fields for each task.

A doubly dependant list is used to make it possible an element of the chain to be removed quickly (OSTaskDel ( ) and OSTaskChangePrio ( ).

OSTCBEventPtr: It is a pointer towards a valve block of event.

The maximum number of tasks, therefore the maximum number of the OS_TCB is declared in the user code. All the OS_TCB are placed in OSTCBTbl [] and to manage by the kernel.

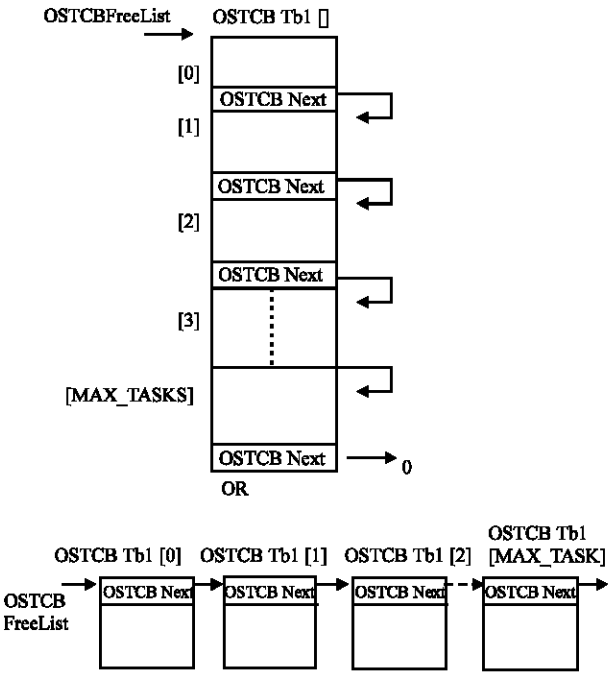

Fig. 1: OS_TCB free list

An extra OS_TCB is allocated to the idle task and thus, if a memorizing is allocated for 20 tasks, OSTCBTbl [] will have 21 entries. When the kernel is initialized, all the OS_TCB, in this table, are linked in a single linked list of free OS_TCB (Fig. 1).

Create a task: Tasks are created by call of OSTaskCreate ( ). This function is specific to the target microprocessor. The code for microprocessor 80486/80188 is shown below. Tasks can be created, either before the beginning of multitasks, or by a task mother during the execution time. A task cannot be created by an ISR (interrupt service routine). The OSTaskCreate () function made pass four arguments:

The task: It is a pointer towards the task code.

The data: It is a pointer towards a definable data surface of the user who is used to pass the arguments. For example, a task can be written to manage multiple communication channels. When a task is created, it passes a pointer towards a structure which defines the channel specificities.

pstk: It is a pointer towards the task stack surface which is used to store the local variables and the registers of the CPU during an interruption.

p: It is the task priority.

```
UBYTE OSTaskCreate(void(far *task)(void *pd), void
*pdata, void pstk, UBYTE p)
{
    UWORD *stk;
    UBYTE err;

    OS_ENTER_CRITICAL();
    If (OSTCBPrioTbl[p] == (OS_TCB *)0) {

      OS_EXIT_CRITICAL()
       Stk = (UWORD *)pstk;
      *--stk = (UWORD)FP_OFF(pdata);
      *--stk = (UWORD)FP_SEG(task);
      *--stk = (UWORD)FP_OFF(task);
      *--stk = (UWORD)0x0200;
      *--stk = (UWORD)FP_ SEG(task);
      *--stk = (UWORD)FP_OFF(task);
      *--stk = (UWORD)0×0000;
      *--stk = (UWORD) 0×0000;
      *--stk = (UWORD) 0×0000;
      *--stk = (UWORD) 0×0000;
      *--stk = (UWORD) 0×0000;
      *--stk = (UWORD) 0×0000;
      *--stk = (UWORD) 0×0000;
      *--stk = (UWORD) 0×0000;
     err = OSTCBInit(p, (void *)stk);
     if (err == OS_NO_ERR) {
       If (OSRunning) {
        OSSched();
       }
     }
     Return (err);
    } else {
     OS_EXITCRITICAL();
     Return (OS_PRIO_EXIST);
    }
```

Before the task is created, OSTaskCreate ( ) makes sure that it was not already created. The stack format of a task created for an 80486/80188 is shown in Fig. 2.

OSTaskCreate ( ) then calls OSTCBInit ( ), which obtains an OS_TCB (Task Control Block) starting from free OS_TCB. If all the OS_TCB were used, the function returns an error code. IF an OS_TCB is available, it is initialized. A pointer towards the OS_TCB of the task is placed in the TCB priorities table of the task OSTCBPrioTbl [], using the priority of the task as index. The OS_TCB of the task is then inserted in a doubly dependant list OSTCBList pointing towards the OS_TCB of the task most recently created. The task is then inserted in the list of the tasks which are ready with the treatment, i.e., the ready list.

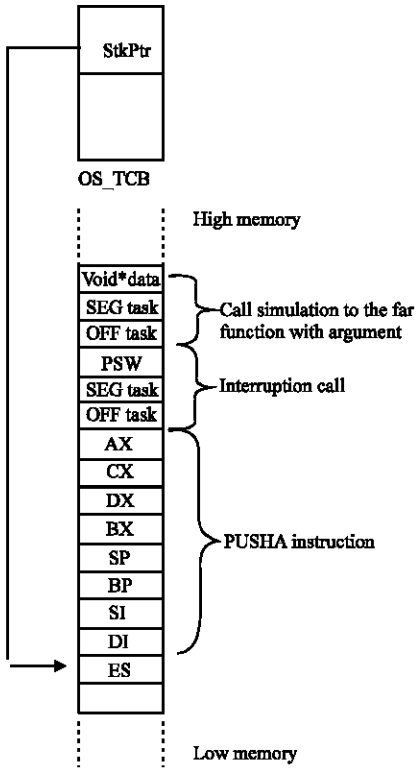

Fig. 2: Stack Format when tasks are created (80186/80188)

If the task is created after multi-task began it (thus created by the task), the scheduler is called to determine if the task created has a higher priority than its creative task; if it is the case, the new task is carried out immediately, if not, OSTaskCreate ( ) turns over to its caller. Initialize the TCB is shown as:

```
UBYTE OSTCBInit(UBYTE p, void *stk)
{
    OS_TCB *ptcb;
    OS_ENTER_CRITICAL();
    ptcb = OSTCBFreeList;
    if (ptcb !=(OS_TCB *)0) {
        OSTCBFreeList       = ptcb->OSTCBNext;
        OSTCBPrioTbl[p]     = ptcb;
        ptcb->OSTCBStkPtr = stk;
        ptcb->OSTCBPrio     = (UBYTE)p;
        ptcb->OSTCBStat     = OS_STAT_RDY;
        ptcb->OSTCBDly      = 0;
        ptcb->OSTCBX        = p and 0x07;
        ptcb->OSTCBBitX     = OSMapTbl[p and 0×07];
        ptcb->OSTCBY        = p >> 3;
        ptcb->OSTCBBitY     = OSMapTbl[p >> 3];
        ptcb->OSTCBEventPtr= (OS_EVENT *)0;
        ptcb->OSTCBNext           = OSTCBList;
        ptcb->OSTCBPrev           = (OS_TCB *)0;
        if (OSTCBList != (OS_TCB *)0 {
         OSTCBList->OSTCBPrev = ptcb;
        }
        OSTCBList         = ptcb;
        OSRdyGrp | = OSMapTbl[p >> 3];
        OSRdyTbl[p >> 3] | = OSMapTbl[p and 0x07];
        OS_EXIT_CRITICAL();
        Return (OS_NO_ERR);
      } else {
        OS_EXIT_CRITICAL();
        Return (OS_NO_MORE_TCB);
      }
}
```

Suppression of a task: A task can turn over, by itself or another task, with the state SLEEPING by call of OSTaskDel (). The code for OSTaskDel () is shown below. The priority of the task to be removed passed to OSTaskDel ().

```
UBYTE OSTaskDel (UBYTE p)
{
    register OS_TCB *ptcb;
    register OS_EVENT *pevent;
    if(p == OS_LO_PRIO) {
        return (OS_TASK_DEL_IDLE;
    }
    OS_ENTER_CRITICAL();
    if((ptcb = OSTCBPrioTbl[p] != (OS_TCB *)0) {
        OSTCBPrioTbl[p] = (OS_TCB *)0;
        If((OSRdyTbl[ptcb->OSTCBY] and= ~ptcb-
        >OSTCBBitX) == 0) {
                OSRdyGrp and= ~ptcb->OSTCBBitY;
        }
        If (ptcb->OSTCBPrev == (OS_TCB *)0;
          ptcb->OSTCBNext->OSTCBPrev = (OS_TCB
          *)0;
           OSTCBList     = ptcb->OSTCBNext;
        } else {
           ptcb->OSTCBPrev->OSTCBNext = ptcb-
           >OSTCBNext;
           ptcb->OSTCBNext->OSTCBPrev = ptcb-
           >OSTCBPrev;
        }
        If ((pevent = ptcb->OSTCBEventPtr) !=
        (OS_EVENT *)0) {
        If ((pevent ->OSEventTbl[ptcb->OSTCBY] and=
        ~ptcb->OSTCBBitX) == 0) {
              Pevent ->OSEventGrp and= ~ptcb-
            >OSTCBBitY;
```

```
            }
        }
        ptcb->OSTCBNext = OSTCBFreeList;
        OSTCBFreeList = ptcb;
        OS_EXIT_CRITICAL();
        OSSched();
        return (OS_NO_ERR);
    } else {
        OS_EXIT_CRITICAL();
        return (OS_TASK_DEL_ERR);
    }
```

OSTaskDel ( ) checks that the priority of the task to be removed is not OS_LO_PRIO, to prevent an application from erasing the idle task of the kernel. OSTCBDel ( ) also checks that the task to be removed was created. If the task were created, it is initially withdrawn from the ready list (ready task to execute), then the OS_TCB is detached from the OS_TCB chain.

If the OSTCBEventPtr field in the OS_TCB of the task is not zero, the task to be removed is on standby event and thus must be withdrawn from the waiting list of event. Before finding the next HPT (Higher Priority Task) to be executed, the OS_TCB is turned over to the list of the free OS_TCB in order to be used by another task.

Note: OSTaskDel () should not be called by an ISR (Interrupt Service Routine).

Task scheduling: The kernel always executes the HPT ready to execute. The scheduler determines the task which at the highest priority and thus which execute soon. The scheduling of the task is carried out by OSSched (). The code of this function is shown as:

```
Void OSSched(void)
{
    Register UBYTE y;

    OS_ENTER_CRITICAL();
    If((OSLockNesting | OSIntNesting == 0) {
        Y = OSUnMapTbl[OSRdyGrp];
        OSTCBHighRdy = OSTCBPrioTbl[(y << 3) +
        OSUnMapTbl[OSRdyTbl[y]]];
        If(OSTCBHighRdy != OSTCBCur) {
            OSCtSwCtr++;
            OS_TASK_SW();
        }
    }
    OS_EXIT_CRITICAL();
}
```

With each task only one level of priority between 0 and 63 is assigned. Priority 63 is always assigned with the idle task of the kernel when it is initialized.

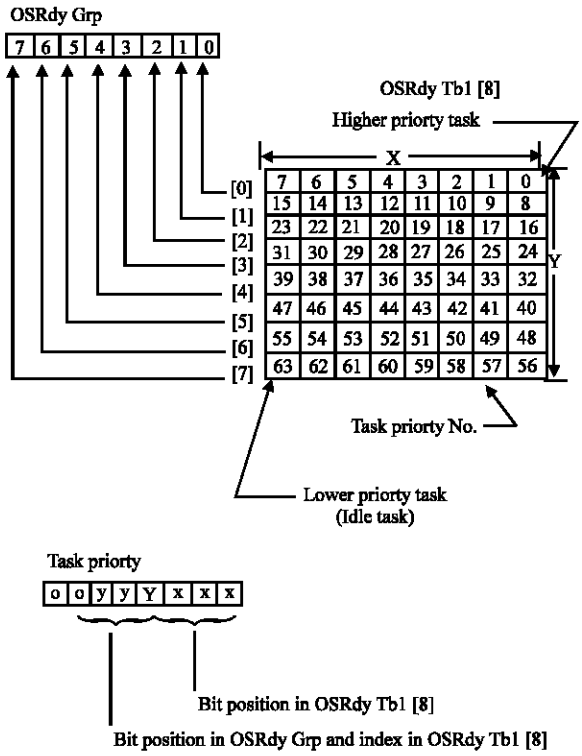

Fig. 3: Ready list

The duration of the scheduling of a task of the kernel is constant, independently of the number of tasks created in an application. Each task which is ready to execute is placed in a ready list containing two variables, OSRdyGrp and OSRdyTbl [8]. The priorities of the tasks are grouped in OSRdyGrp (8 tasks by group). Each bit in OSRdyGrp is used in indicating all the times that a task, in a group is ready to be executed. When a task is ready to be executed, it puts also its corresponding bit in the ready table, OSRdyTbl [8]. To determine the following priority (and thus the task) which will be executed, the scheduler determines the number of the lowest priority which has its bit placed in the table OSRdyTbl [8]. The relation between OSRdyGrp and OSRdyTbl [8] is shown in Fig. 3 and is given by the following rules:

Bit 0 in OSRdyGrp is 1 when any bit in OSRdyTbl [0] is 1.
Bit 1 in OSRdyGrp is 1 when any bit in OSRdyTbl [0] is 1.
....
....
Bit 7 in OSRdyGrp is 1 when any bit in OSRdyTbl [0] is 1.

The following code is used to place a task in the ready list (Fig. 3):

```
OSRdyGrp              |= OSMapTbl [p >> 3];
OSRdyTbl [p >> 3]     |= OSMapTbl [p and 0×07];
```

p is the task priority.

As can be shown in the Fig. 3, the three least significant bits of the task priority are used to determine the bit position in OSRdyTbl [8], whereas the three following most significant bits are used to determine the index in OSRdyTbl [8]. To notice that OSMapTbl [8] is a table in the ROM memory, used to equalize an index from 0 to 7 with a bit mask as shown it the following table:

Index	bit mask (binary)
0	00000001
1	00000010
2	00000100
3	00001000
4	00010000
5	00100000
6	01000000
7	10000000

A task is withdrawn from the ready list by conversely process. The following code is executed in this case:

```
if ((OSRdyTbl [p >> 3] and = ~OSMapTbl[p and 0×07]) == 0)
        OSRdyGp and = ~OSMapTbl[p >> 3];
```

This code erases the ready bit of the task in OSRdyTbl [8] and erases the bit in OSRdyGrp only if all the tasks in a group are not ready to execute, i.e., all bits in OSRdyTbl [] are 0. Another table look-up is carried out, rather than to explore the table while starting with OSRdyTbl [0] to find the task at higher priority ready to execute. OSUnMapTbl [256] is a table of resolution (definition) of priority. Eight bits are used to represent tasks which are ready in a group. The least significant bit has the highest priority. The following code section determines the priority of the task at higher priority to execute:

```
y    = OSUnMapTbl [OSRdyGrp];
X    = OSUnMapTbl [OSRdyTbl [y]];
p    = (y << 3) + X;
```

Again, p is the task priority and having a pointer towards block OS_TCB of this task, (OSTCBHighRdy) is carried out by indexing in OSTCBPrioTbl [64] using the task priority.

Once the HPT found, OSSched () checks that it is not the current task in order to avoid a useless context switch. The whole code in OSSched () is regarded as a critical section. Interruptions are invalidated to prevent ISR to indicate the ready bit of one or more tasks during the process of research of the HPT to be executed. OSSched ( ) could be entirely written in assembly language to reduce the duration of scheduling. OSSched ( ) is written in language C for legibility and the portability and also to minimize the assembly language.

Count of mapping to correspond bit position to the bit mask: The index in the table is the position of the wished bit, 0...7. The indexed value corresponds to the mask of bit.

```
UBYTE const OSMapTbl [] = {0×01, 0×02, 0×04, 0×08,
                           0×10, 0×20, 0×40, 0×80}
```

Priority resolution table: The index in the table is the bit configuration to solve the highest priority. The indexed value corresponds to the bit position of the highest priority.

```
UBYTE const UMapTbl [] = {
        0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
        4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
        5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
        4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
        6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
        4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
        5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
        4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
        7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
        4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
        5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
        4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
        6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
        4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
        5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
        4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0
};
```

INTERRUPTION TREATMENT

Here, covers the assumption of responsibility of the interruption, the importance of the response times of the interruption and clock ticks interruptions.

The kernel requires an ISR written in assembly language. For the 80486/80188, the ISR must be written as follows:

```
ISR×PROC FAR
    STI
    PUSHA
    PUSH ES
    CALL_OSIntEnter
    CALL_UserISRCode
    CALL_OSIntExit
    POP ES
    POPA
    IRET
ISR×ENDP
```

Interruptions are validated early in this ISR if other interruptions have a higher priority. However, our ISR does not have to validate the interruptions prematurely. We could carry out a certain code with the interruptions invalidated as shown low, but should save to us the remainder CPU context before carrying out any code.

```
ISR×PROC FAR
    PUSHA
    PUSH ES
    CALL_UserCode
    STI
    CALL_OSIntEnter
    CALL_UserISRCode
    CALL_OSIntExit
    POP ES
    POPA
    IRET
ISR×ENDP
```

By executing the code before validating the interruptions, one increases the interruption waiting duration (latency) for other interruptions.

The kernel requires that the OSIntEnter ( ) function is called before any system call. OSIntEnter ( ) keep the trace of the interruption overlap level. The kernel makes it possible the interruptions to be overlapping up to a level of 255. With the return of OSIntEnter ( ), the code of the ISR of the user is called upon. It should be noted that the user code is responsible for the interruption suppression.

The completion of the ISR is marked by the call of OSIntExit ( ) which decrement the level of the interruption overlap. When the level of overlap is zero, all the interruptions are completed, the kernel determines if a task at higher priority were awaked by an ISR (or any other event which was to interrupt this ISR). If a HPT is ready to be executed, the ISR will turn over to the HPT rather than to the interrupted task (unless scheduling was invalidated: OSLockNesting! = 0)).

Introducing ISR

```
Void OSIntEnter(void)
{
    OS_ENTER_CRITICAL();
    OSIntNesting++;
    OS_EXIT_CRITICAL();
```

Leaving ISR

```
Void OSIntExit(void);
{
  OS_ENTER_CRITICAL();
  If ((--OSIntNesting | OsLockNesting) == 0) {
      OSIntExitY = OSUnMapTbl[OSRdyGrp];
      OSTCBHighRdy = OSTCBPrioTbl[(OSIntExitY << 3)+
      OSUnMapTbl[OSRdyTbl[OSIntExitY]]];
      If (OSTCBHighRdy != OSTCBCur) {
          OSCtxSwCtr++;
      OSIntCtxSw();
      }
  }
  OS_EXIT_CRITICAL();
}
```

OSIntExit() calls OSIntCtxSw function in assembly language instead of OSCtxSw. There are two reasons for that: firstly, half of work is already achieved, because the interruption saved the PSW, CS (Code Segment) and the IP (Instruction Pointer) of interrupted task and all the other registers (saved at the beginning of the ISR). Secondly, OSIntExit () allocates local variables (register SO for the 80486/80188) on the stack of interrupted task and calls upon OSIntCtxSw.

To make the stack format such a simple context commutation is in progress, the stack pointer of the processor needs to be adjusted as shown in Fig. 4. The number to be added to SP register depends on the compiler used and the choice on the options on the compiling duration.

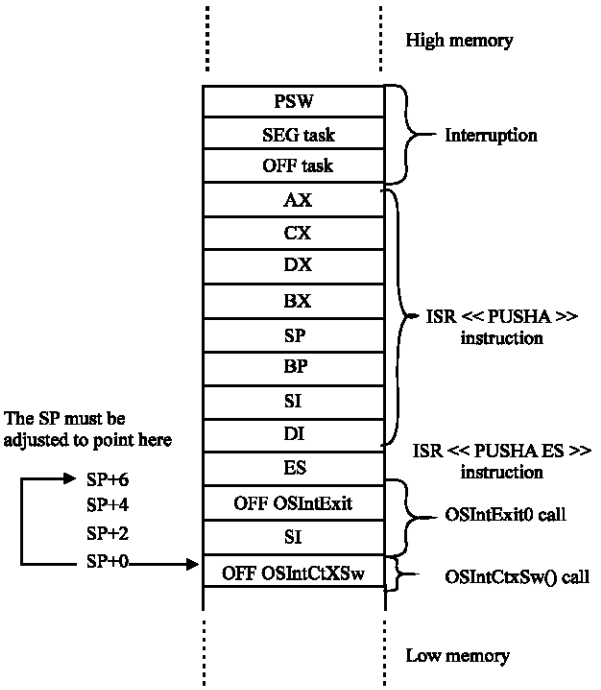

Fig. 4: Stack format during an interruption (80486/80188)

Carry out a context switch (starting from an ISR)
Void OSIntCtxSw(void)

Total execution time: 142 bus cycles
_OSIntCtxSw proc NEAR

```
ADD SP,6                    ; 4~, be unaware of the calls to OSIntExit and OSIntCtxSw
MOV BX,[_OSTCBCur]          ; 9~, save the stack pointer in the old TCB
MOV [BX],SP                 ; 12~
MOV BX,[_OSTCBHighRdy]      ; 9~, point towards TCB of the ready HPT to execute MOV
[_OSTCBCur],BX              ; 12~, this update the TCB now
MOV SP,[BX]                 ; 9~, obtain the new task stack pointer
MOV ES                      ; 8~
POPA                        ; 51~
IRET                        ; 28~, return to the new task
```

_OSIntCtxSw ENDP

Interruption latency, response and recovery: The duration necessary to answer an interruption and to really begin the execution of the user code to deal with the interruption (Fig. 5) is an important point in real-time systems.

Our kernel disables interruptions for a maximum of 500 cycles of CPU clock (80486/80188).

The latency is given by:

Maximum duration of invalidation of interruption (500) + lasted of routing towards the ISR (50)

In the worst case, interruption latency is thus 550 cycles of CPU clock. Before the execution of the user code, the context of the processor must be saved and if the services of the kernel are required, OSIntEnter () must be called. The response time is given by:

Latency of interruption (550) +
time of saving all registers (60) +
execution time of OSIntEnter () (75).

For this kernel, the response time, in the worst case, is thus of 685 cycles of CPU clock.

Since, the kernel disables interruptions for 500 cycles of CPU clock (80486/80188), an application can make the same thing without carrying out the interruption latency

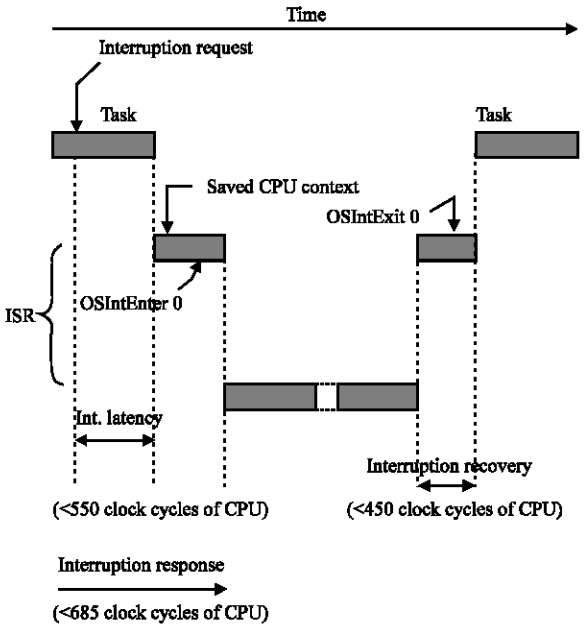

Fig. 5: Interruption latency (timeout), response and recovery

(in the worst case). For example, the application could invalidate the interruptions for 500 cycles of CPU clock to reach the critical sections of the code or to reach the shareable data.

Recovery time is the duration necessary to the processor to restore its context and to turn over interruption. The recovery of the interruption is carried out by OSIntExit (). If a HPT were put at the ready state to execute (result of the ISR), the recovery time of the interruption, in the worst case, is of 450 cycles of CPU clock (Fig. 5).

Clock tick: The kernel allows tasks either to suspend the execution for a number clock ticks or to wait until the occurrence of an event with a timeout. A clock tick is typically gotten by a periodic interruption and can be regarded as the beat of the system heart. The duration between interruptions (ticks) is specific to the application and is typically between 10 and 200 m sec. Faster is the flow of ticks, highest are the overhead.

The code to treat a tick interruption can be established as follows:

```
TickISR PROC FAR
        STI
        PUSHA
        PUSH ES
        CALL_OSIntEnter
        CALL_OSTimeTick
```

```
;
;               user code to erase tick interruption
;
                CALL_OSIntExit
                POP ES
                POPA
                IRET

    TickISR ENDP
```

The tick ISR calls OSTimeTick () which decreases the field of OSTCBDly for each OS_TCB if it is not zero. OSTimeTick () follows chain OS_TCB starting with OSTCBList until reaching the idle task. When the OSTCBDly field of a task OSTCB is decreased to zero, the task is put at the ready state to execute. The execution time of OSTimeTick is directly proportional to the number of tasks created in an application. System tick treatment is shown as:

```
Void OSTimeTick(void)
{
    Register OS_TCB *ptcb;
    Ptcb = OSTCBList;
    While (ptcb->OSTCBPrio != OS_LO_PRIO) {
        OS_ENTER_CRITICAL();
        If (ptcb->OSTCBDly != 0){
                If(--ptcb->OSTCBDly == 0){
                     OSRduGrp != ptcb->OSTCBBitY;
                      OSRdyTbl[ptcb->OSTCBY] != ptcb-
                     >OSTCBBitX;
                }
        }
        OS_EXIT_CRITICAL();
        Ptcb = ptcb->OSTCBNext;
    }
    OS_ENTER_CRITICAL();
    OSTime++
    OS_EXIT_CRITICAL();
}
```

If one does not wish to have the ISR as long as they would owe the being, OSTimeTick () can be called on the level of the task. For thus making, we allocate priority 5 with the task to serve the tick as follows (the priorities from 0 to 4 are reserved for the future use).

```
 TimeTask ()
 {
    while (1) {
            OSSemPend (...);
              OSTimeTick ();
    }
 }
```

We could create a semaphore (initialized to 0) to announce to the task that a tick interruption arrived. The execution time of the tick interruption could have been very short as shown low (80486/80188):

```
    TickISR PROC FAR
                STI
                PUSHA
                PUSH ES
                CALL_OSIntEnter

                MOV AX,_TickSem
                PUSH AX
                CALL_OSSemPost
                ADD SP,2
;               user code for erasing the tick interruption
;
                CALL_OSIntExit
                POP ES
                POPA
                IRET

    TickISR ENDP
```

OSTimeTick () accumulates also the number of klock ticks (variable not signed of 32 bits called OSTime). The current price of OSTime can be read by a task by calling OSTimeGet (). OSTimeGet prevents the application from directly handling OSTime. OSTime can also be forced with any value during the execution of the program by call of OSTimeSet () is shown as:

```
ULONG OSTimeGet(void)
{
    ULONG ticks;

    OS_ENTER_CRITICAL();
    Ticks = OSTime;
    OS_EXIT_CRITICAL();
    Return (ticks);
}
```

To put the system clock

```
Void OSTimeSet(ULONG ticks)
{
        OS_ENTER_CRITICAL();
        Ticks = OSTime;
        OS_EXIT_CRITICAL();
}
```

COMMUNICATION, SYNCHRONIZATION AND COORDINATION

Here, we give a short description without going in the details. The details of this section will be the subject of another article which will be given later on.

The kernel uses mail boxes of message and the files (queues) for the communication. It uses the semaphore for synchronization and coordination. Under the kernel, these services are regarded as events; one announces is the occurrence of an event (POST) or one awaits the arrival of an event (PEND).

Only the tasks are authorized to await the occurrence of events, indeed, an ISR should never wait for an event. More than one task is authorized to await the occurrence of the same event, when the event arrives, the kernel puts at the ready state to execute the HPT on standby event. The ISR and the tasks can announce the arrival of the events.

Event Control Block (ECB): A data structure called Event Control Block (ECB) is used to maintain the state of an event. The state of an event is composed of:

The event itself:

- A semaphore counter
- A message for the mail-boxes
- A file of message for files (queues)

A waiting list for the tasks awaiting the occurrence of events

With each semaphore, mail-boxes and file, is assigned an ECB
An ECB contains four fields
OSEventGrp is similar to OSRdyGrp except that it indicates that any task in a group of eight tasks is on standby of the arrival of an event
OSEventTbl [8] is also similar to OSRdyTbl [] except that it contains an addressable bit of waiting tasks for events
OSEventCnt is used to maintain the counting of the semaphore (when the ECB is used for a semaphore)
OSEventPtr contains the mailboxes message or a pointer towards the queue data structure, respectively (when the ECB is used for mailboxes or a file)

Semaphores: The semaphores of the kernel are signed entities with 16 bits which must be initialized with a value between 0 and 32767 before their use. The semaphore can be only handled through three functions:

- OSSemCreate ()
- OSSemPend ()
- OSSemPost ()

OSSemCreate () allocates a valve block of event to be used by the semaphore. It positions also the initial value of the semaphore.

OSSemCreate () return a pointer to the ECB allocated to the semaphore. This pointer will need to be assigned with a variable in the application, because it is used like the pointer of the semaphore. If all the ECB are used, OSSemCreate () return a NULL pointer. The field of OSEventCnt () of the ECB includes/understands the current semaphore value and can be between -63 and 32767. A positive value indicates the number of tasks which can reach the resource at the same time, or the number of times that an event arrived. When the value of the semaphore is zero, the resource is not available, or the event did not arrive. When the value is negative, it indicates the number of tasks waiting for a resource to becoming available, or for the occurrence of an event.

If a task calls OSSemPend () and the semaphore value is higher than zero, then OSSemPend () decreases the semaphore count and turns over to its caller. If the semaphore value is lower or equal to zero, OSSemPend () decreases the semaphore count and place the calling task in the semaphore waiting list.

A semaphore is announced by call of OSSemPost (). If the semaphore value is equal to or higher than zero, the semaphore count is increased and OSSemPost () turns over to its caller. If the semaphore count is negative, tasks waiting for semaphore are to be announced. In this case, OSSemPost () withdraws the HPT waiting for semaphore of the waiting list and puts this task at the state ready to execute by OSTaskResume ().

Mailboxes: Kernel authorizes a task or an ISR to send a message (a pointer size variable) to one or many tasks through a mailbox. The application decides towards what the pointer points. A mailbox can be handled through three functions:

- OSMboxCreate()
- OSMboxPend()
- OSMboxPost()

OSMboxCreate() allocates an ECB for use by the mailbox and allows to initialize its contents. OSMbox () return a pointer to the ECB allocated to the mailbox. This pointer will need to be assigned with a variable in the application, because it is used like the pointer of the mailbox. If all the ECB are used beforehand, OSMboxCreate () return a NULL pointer.

If a task calls OSMboxPend() and the mailbox contains a non NULL pointer, then OSMboxPend() take out the message from the mailbox and erase it. The withdrawn message is returned to the OSMboxPend() caller. If the mailbox contains a NULL pointer, OSMboxPend() place the calling task in the waiting list for mailbox. The task will wait until another task or an ISR deposits a message in the mailbox.

A message is sent to a task by calling OSMboxPost(). If the mailbox contains already a message, OSMboxPost() return an error code to his caller. If the mailbox is empty, the message is deposited there and OSMboxPost() determines if there is a waiting task for a coming message. If there is a waiting task, OSMboxPost() withdraw the HPT from the waiting list and put this task at the ready state for execute by calling OSTaskResume().

Queues: Queues are similar to mailboxes. A mailbox permits to a task or an ISR to send a single message of pointer size to one or many tasks. Queues are used to send a user finite messages number to one or many tasks. As for mailboxes, contents of sent messages are specific to the application. A queue may be handled through three functions:

- OSQCreate()
- OSQPend()
- OSQPost()

OSQCreate() allocate an ECB to be used by the queue. OSQCreate() return a pointer to the ECB allocated to the queue. This pointer will need to be assigned to a variable in the application, because it is used as queue pointer. If all ECB are used, OSQCreate() return a NULL pointer.

If a task calls OSQPend and the queue contains one or many messages, then OSQPend() withdraws the pointed message by OSQOut from the queue and return to the calling task. However, if the queue is empty, OSQPend() place the calling task in a queue waiting list. The task will wait until a task or an ISR deposits a message in the queue. When the current task is placed in the waiting list, rescheduling occurs and the following HPT ready to execute take CPU control.

A message is sent to a task by calling OSQPost(). An error occurs if the queue is full. In this case, OSQPost() return an error code to his caller. If the queue is not full, the message is deposited in the queue and OSQPost() determines if there is a waiting task for a coming message. In this case, OSQPost() withdraw the HPT waiting for a message in a waiting list and put this task in the ready state to execute by calling OSQResume().

CONCLUSIONS

This study proposes a development framework for the industrial applications by giving to the designer architectural elements for the general organization of its software. The proposed services at the higher level (modules of service) make it possible to facilitate, by guiding the passage of the specification to the preliminary design, the design of a large variety of industrial applications.

The system architecture has changed from a federated system architecture to an integrated system architecture and then to system of systems. Each shift has brought about enormous challenges to the available technological infrastructure.

In integrated system architecture, sensors, communication channels and processors are extensively shared. The large number of possible configurations becomes a challenge for system architects. The current generations of schedulability analysis tools offer inadequate support for system architects. They must manually create the alternative options and then check the schedulability of each option. They would like to have tools to automatically search the design space and perform sensitivity analysis regarding the uncertainty of task parameters. During the system engineering phase, the values of task parameters are often educated guesses. Finally, from the perspective of runtime reconfiguration, dynamic priority scheduling theory has potential advantages (Chandra *et al*., 2003). In addition to the potential of higher schedulability, the feasibility analysis of dynamic priority scheduling is often faster. We are looking forward to the maturing and subsequent use of dynamic scheduling theory in practice (Chen and Mok, 2004).

In modern integrated systems, there are substantial high volume and high variability imaging data streams. Depending on the nature of applications, they have either hard or soft end-to-end deadlines (Abdelzaher *et al*., 2004; Yeung and Lehoczky, 2001). If the images are used for steering a vehicle, they will have a hard end-to-end deadline and tight jitter tolerance. Such hard-deadlines streams pose challenges to traditional scheduling theory using worst-case assumptions. A system of systems is often a large distributed system, where keeping distributed views and actions timely and consistent is at the heart of collaborative actions. Ideally, we would like to keep distributed views, state transitions and actions consistency with each other. In business systems, the consistency of a distributed system is managed by atomic operations. Simply put, atomic operations wait for every working component to be ready and then commit the operations. However, this may not be viable for real-time systems. How to handle the interactions between timing

constraints, consistency requirements and re-synchronisation in a large distributed system is a challenge. As a networked embedded system of systems grows larger and the coordination becomes tighter, so will be the impact of this technological challenge. The re-synchronization loop is a form of feedback control. Feedback is a powerful technique which has yet to be fully exploited in the control of the behaviour of computing systems in the face of uncertainty (Eker *et al*., 2000; Gandhi *et al*., 2001).

Another characteristic of a system of systems is that a wide variety of real-time, fault tolerance (Lima and Burns, 2003) and security protocols are used in different systems, because most of the systems of systems are integrated, not built from scratch. Priority inversion is an example of pathological interaction between independently developed synchronization protocol and priority scheduling protocol. This is not an easy problem to solve because the scope of modern technologies is so large and complex. To advance any area, one must specialize. As a result, we have developed mechanisms with little attention on how separately developed protocols may interact. Research is needed to formally verify that protocols do not invalidate each others' pre-conditions when they interact.

ACKNOWLEDGMENTS

This study was elaborated at the University of Sciences and the Technology of Oran (faculty of electronic engineering) within the framework of a research program in the field of the industrial data processing and the multi-task real-time applications. This study was supported by the Ministry of higher education and scientific research Project CNEPRU

REFERENCES

Abdelzaher, T., G. Thaker and P. Lardieri, 2004. A feasible region for meeting aperiodic end-to-end deadlines in resource pipelines. IEEE International Conference on Distributed Computing Systems, March 2004, Tokyo, Japan, pp: 1-10.

Abeni, L. and G. Buttazzo, 2004. Resource reservation in dynamic real-time systems. Real-Time Syst., 27: 123-167.

Baker, T.P., 2006. An analysis of fixed-priority schedulability on a multiprocessor. Real-Time Syst., 32: 49-71.

Baki'c, A.M. and M. Mutka, 2005. Integrating on-line performance visualization and real-time system design. Real-Time Syst., 30: 163-185.

Balbastre, P., I. Ripoll, J. Vidal and A. Crespo, 2004. A task model to reduce control delays. Real-Time Syst., 27: 215-236.

Beccari, G., S. Caselli and F. Zanichelli, 2005. A technique for adaptive scheduling of soft real-time tasks. Real-Time Syst., 30: 187-215.

Chandra, R., X. Liu and L. Sha, 2003. On the scheduling of flexible and reliable real-time control systems. Real-Time Syst., 24: 153-169.

Chen, D. and A.K. Mok, 2004. The Pinwheel: A Real-Time Scheduling Problem. In: Handbook of Joseph, Y.T.L. (Ed.) CRC Press, Chapman Hall .

Eker, J., P. Hagander and K.E. Arzen, 2000. A feedback scheduler for real-time control tasks. Control Eng. Pract., 8: 1369-1378.

Fohler, G. and G.C. Buttazzo, 2002. Introduction to the special issue on flexible scheduling. Real-Time Syst., 22: 5-7.

Gandhi, N., S. Parekh, J. Hellerstein and D.M. Tilbury, 2001. Feedback control of a lotus notes server: Modeling and control design. Am. Control Conf., 4: 3000-3005.

Hamidzadeh, B., Y. Atif and K. Ramamritham, 1999. To schedule or to execute: Decision support and performance implications. The Int. J. Time-Crit. Comput. Syst., 16: 281-313.

Harel, D. and A. Pnueli, 1989. On the Development of Reactive Systems. Computer and Systems Sciences, 1st Edn., Springer-Verlag New York, ISBN: 0-387-15181-8 pp: 477-498.

Kornecki, A.J. and J. Zalewski, 2005. Experimental evaluation of software development tools for safety-critical real-time systems. Innovat. Syst. Softw Eng., 1: 176-188.

Lima, G. and A. Burns, 2003. An optimal fixed-priority assignment algorithm for supporting fault-tolerant hard real-time systems. IEEE Trans. Comput. Syst., 52: 1332-1346.

Liu, P., M. Cai, T. Fu and J. Dong, 2007. An EDF Interrupt Handling Scheme for Real-Time Kernel. In: Design and Task Simulation, Shi, Y. *et al*. (Eds.): ICCS 2007, Part IV, LNCS 4490, Springer-Verlag, Berlin Heidelberg, pp: 969-972.

Racu, R., A. Hamann and R. Ernst, 2007. Sensitivity analysis of complex embedded real-time systems. Real-Time Syst. 10.1007/s11241-007-9039-9

Real, J. and A. Crespo, 2004. Mode change protocols for real-time systems: A survey and a new proposal. Real-Time Syst., 26: 161-197.

Rodrigues J., J. Ventura, A.M. de Campos and L. Rodrigues, 2007. Implementation and analysis of real-time communication protocol compositions. Real-Time Syst., 37: 45-76.

Yeung, S.N. and J.P. Lehoczky, 2001. End-to-end delay analysis for real-time networks. 22nd IEEE Real-Time Systems Symposium (RTSS'01), Dec. 2001, IEEE, Computer Society, pp: 299-309.