New Real-Time microkernel and facilities for MINIX (RT-MINIX version 2.0.2)

Pablo A. Pessolani

Departamento de Sistemas de Información

Facultad Regional Santa Fe

Universidad Tecnológica Nacional - Argentina

Gerencia de Ingeniería de Servicios

Telecom Argentina S. A.

ppessolani@hotmail.com

ABSTRACT

Tanenbaum's MINIX operating system [1] was extended by Wainer with Real-Time services to comform RT-MINIX[2, 3]. This work add new extensions that includes a Real-Time microkernel, more flexible Interprocess Communication facilities, basic priority inheritance protocol, statistics and Real-Time metrics collection, timer and event driven interrupt management and a new scheduling algorithm, keeping backward compatibility with standard MINIX versions. This report discuss the design decisions, explain the programming interfaces and added system calls, system data structures, sample pseudocode and evaluation results of RT-MINIX version 2.0.2.

TABLE OF CONTENTS

1. INTRODUCTION

In these days the existing number of Hard Real-Time systems is growing rapidly and the functionality that Real-Time applications require of their operating system is much different from the functionality required by non time constrained timesharing applications.
Engineers and Computer Science professionals working on Real-Time projects need to have a deep knowledge of every software component and the interactions with hardware devices considering timing constraints. The experience earned in well-planed assignments and projects in OS course enhance their knowledge and personal performance.

1.1. Motivation

MINIX[1] is a general-purpose time sharing Operating System broadly used in OS degree courses [11, 23, 24, 25].
The aim of the project is to provide an educational tool like MINIX but for Real-Time OS courses. Several reasons lead us to select MINIX among other operating systems as the base for RT-MINIXv2. These are:

- Our previous experience.
- Existing documentation.
- Hardware platform requirements.
- It's clear and modular design.

The RT-MINIXv2 keeps MINIX modular design to let Real-Time OS teachers make easily a multiplicity of grade courses assignments, laboratory tests, projects and other academic uses with a well documented and known Real-Time OS.

1.2. Terminology and Notation

Before describing RT-MINIXv2, it is useful to clear some Computer Science terminology.

In MINIX terminology, a *process* is an instance of a program in execution and a *task* is as a special *process* type used in the implementation of MINIX device drivers. In Real-Time terminology, *task* is the term used for *process.*
MINIX distinguish among three kinds of processes:

- *USER:* to refer to a MINIX user's process.
- *SERVER:* is special *process* type used to serve requests from user's *processes* as the Memory Manager Server (MM) or the File System Server (FS).
- *TASK:* to refer to a MINIX device driver task

We will use this words in uppercase to refer to MINIX terminology.

Other confusing term is the IBM-compatible PCs device that can produce interrupts at regular periods (ticks). MINIX routines refer it as the *clock device* but the correct term is *timer device*.

To simplify the notation, for the following paragraphs all Real-Time related words will be preceded by "RT-" and Non Real Time words will be preceded by "NRT-".

1.3. MINIX Time Sharing Features

MINIX is a complete, time sharing, multitasking operating system developed from scratch by Andrew S. Tanenbaum. Though it is copyrighted, the source has been made widely available to universities for study and research in computer science courses. Its design is highly modular, clear and elegant.
Its main features are:

- ***Microkernel based:*** Provides process management and scheduling, basic memory management, interprocess communication, interrupt processing and low level I/O support.
- ***Multilayer system:*** Permits a modular design and clear implementation.

- ***Client/Server model:*** All system services and device drivers are implemented as server processes with their own execution environment.
- ***Message Transfer Interprocess Communications (IPC):*** Used for process synchronization and data sharing among processes.
- ***Interrupt hiding:*** Interrupts are converted in message transfers.

1.4. RT-MINIX Features

Wainer[2, 3] changed the standard MINIX operating system to support Real-Time processing named it "RT-MINIX". Its features are:

- ***Different Scheduling Algorithms Can Be Selected***
- ***Joined Scheduling Queues***
- ***Real-Time Metrics collection***
- ***Timer Resolution Management***

1.5. RT-MINIXv2 Features

Existing real-time operating systems (RTOS) can be divided in two categories:

- Systems implemented using somewhat stripped down and optimized (or specialized) versions of conventional timesharing OS
- Systems starting from scratch, focusing on predictability as a key design feature.

RT-MINIXv2 is based on the first design approach using MINIX as the conventional OS.
This special version offers predictable Real-Time computing environment at lower cost than propietary RTOS used to teach Real-Time systems and other academic purposes.

The major features of RT-MINIXv2 are summarized as follows:

- ***Real-Time microkernel***
- ***Modular Architecture that simplify enhancements and changes***
- ***Timer/Event Driven Interrupt Management***
- ***Periodic and Sporadic processing***
- ***Timer Resolution Management***
- ***Priority Based Real-Time Scheduling***
- ***Synchronous/Asynchronous Message Transfer using ports.***
- ***Basic Priority Inheritance Protocol support in Message Transfer IPC.***
- ***Receive and Synchronous Send Timeout support.***
- ***Statistics and Real-Time Metrics Collection.***
- ***Two levels of Interrupt Handling with Software IRQs***

Is widely believed that microkernel based systems are inherently inefficient and a multilayer message transfer kernel have performance disadvantage over monolitic kernel. But in [26] is presented evidence that the inefficiency is not inherited from the basic idea but from improper implementation. Current CPU's speeds, system modularity and the academic purposes let us to focalize our design in concepts other than performance like flexibility, schedulability, preemptability, timing precision, etc.

The advantage of using a microkernel for Real-Time applications is that the preemptability of the kernel is better, the size of the kernel becomes much smaller, the addition/removal of services is easier.

RT-MINIXv2 provides the capability of running special realtime tasks and interrupt handlers on the same machine as standard MINIX. These tasks and handlers execute when they need to execute no matter what MINIX is doing.

RT-MINIXv2 works by treating the MINIX OS kernel as a task executing under a small realtime

operating system. In fact, MINIX is like the idle task for the realtime operating system, executing only when there are no realtime tasks to run.

The MINIX task can never block interrupts or prevent itself from being preempted. The technical key to all this is a software emulation of interrupt control hardware. When MINIX tells the hardware to disable interrupts, the realtime system intercepts the request, records it, and returns to MINIX.

MINIX is not ever allowed to really disable hardware interrupts. No matter what state MINIX is in, it cannot add latency to the realtime system interrupt time.

RT-MINIXv2 is a general purpose Real-Time Operating System with a default Fixed Priority scheduler that can be used for:

- *Interrupt driven devices:* As MINIXv2 is designed for 32 bits INTEL PCs [9], the interrupt levels of the standard devices like keyboard, serial ports, etc cannot be changed and may produce priority inversion problems with other devices.
- *Timer driven devices:* the devices are polled or some job is done on the device at regular periods. As in standards PCs the timer has the top priority level, no priority inversion problems can occurs.

The current version of RT-MINIX is based on MINIX version 2.0.2 for 32 bits INTEL [9] processors therefore it requires the same hardware platform.

2. ARCHITECTURE AND INTERRUPT HANDLING

2.1 MINIX System Architecture

MINIX is structured in four layers as it can see in Figure 1.

1. The microkernel
2. Input/Output Tasks
3. Server Processes
4. User Processes

Interrupts are the heartbeats of the operating system. An interrupt is usually defined as an event that alters the sequence of instructions executed by a processor [28].
In MINIX, when a hardware device interrupts the CPU, an *Interrupt Service Routine (ISR)* or an *interrupt handler* is called too partially process the interrupt. If the interrupt needs more time to complete its job, the ISR sends a message to the Interrupt Service Task (IST) and returns calling the process scheduler. An IST is like a kernel thread that share kernel address space and have processing attributes. We call this approach *Two Layer Interrupt Handling*.

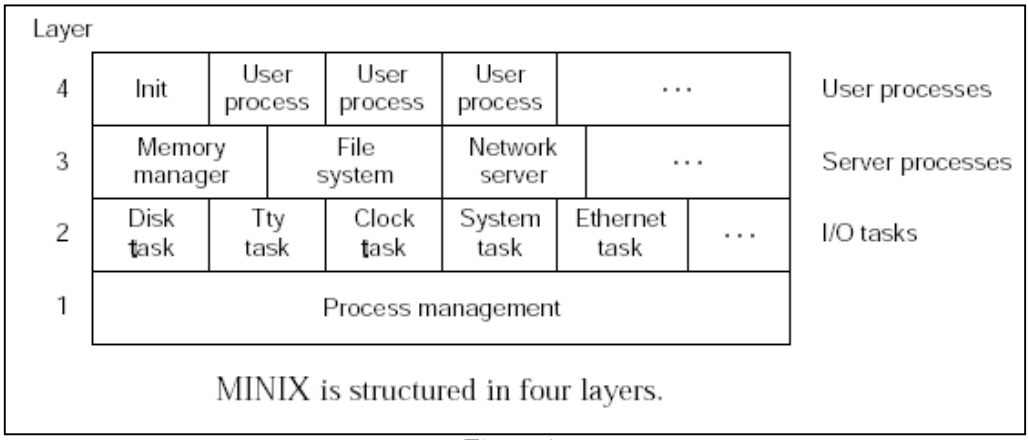

Figure 1
from Operating Systems: Design and Implementation, 2nd ed.

2.2 RT-MINIXv2 System Architecture

As RT-MINIXv2 intends to be used in an academic environment, we decide to be as least intrusive as possible in the standard MINIX source code. Yodaiken and Barabanov [8] have proposed that approach for RT-LINUX. The key idea is how interrupt management is done.

As result, one Real-Time Operating System (RT-MINIXv2) hosts a standard time sharing Operating System (MINIX). These two OS have their own set of system calls.

The RT-MINIXv2 effectively puts in place a new scheduler that treats the MINIX operating system kernel as the lowest priority process executing under the RT-kernel. Under that design, MINIX only executes when there are no RT-process to run, and the RT-kernel is inactive. Thus, the MINIX process can never block interrupts or prevent itself from being preempted, yielding all resources to a RT-process. MINIX kernel may be preempted by a RT-process even during a system call, so no MINIX routine can be safely called from a RT-process.

Some problems must be solved.

1. The interrupts must be captured by a RT-kernel.
2. Real-Time schedulers and services must be implemented.
3. Real-Time applications need an interface layer to interact with the Real-Time kernel.
4. RT-applications may need transfer data and synchronize with NRT-applications.
5. Full process and interrupt handler preemptability is needed

2.3 Interrupt Handling

RT-MINIXv2 uses Virtual Machine (VM) concept limited to interrupt emulation o virtualization. Its microkernel is underneath of MINIX and the scheduler runs NRT-processes when there are not RT-processes ready to run.

The task of capturing and redirecting the interrupts was addressed by creating a small Real-Time microkernel, which captures all hardware interrupts and redirects them to either standard MINIX handler or to RT-MINIXv2 handler. The Real-Time microkernel provides a framework onto which RT-MINIXv2 is mounted with the ability to fully preempt MINIX.

A key component in the architecture is the Interrupt Descriptor Table (IDT). The IDT is an array of 8 byte interrupt descriptors in memory devoted to specifying (at most) 256 interrupt service routines. The first 32 entries are reserved for processor exceptions, and any 16 of the remaining entries can be used for hardware interrupts. The rest are available for software IRQs.

MINIX defines a table called *irq_table[]* that has the pointers to C language written interrupt handlers. This table lets easily change among Real-Time and standard handlers.

At start, RT-MINIXv2 virtual machine sets all *irq_table[]* pointers to the address of a common interrupt handler named *RTM_IRQ_dispatch()* that uses an interrupt descriptor table to dispatch the interrupt processing to a Real-Time or to a standard MINIX handler. The interrupt descriptor table, *RTM_desc_table[]* is an array of a *RTM_irq_desc_t* data structures.

All MINIX standard kernel functions that handle interrupts and the PIC are replaced with virtualized ones to avoid that MINIX could not be preemptive when a Real-Time interrupt occurs.

RT-MINIXv2 defines three types of interrupt handlers:

1. Standard MINIX interrupt handlers or NRT-handlers.
2. Real-Time Timer-Driven interrupts handlers.
3. Real-Time Event-Driven interrupts handlers.]

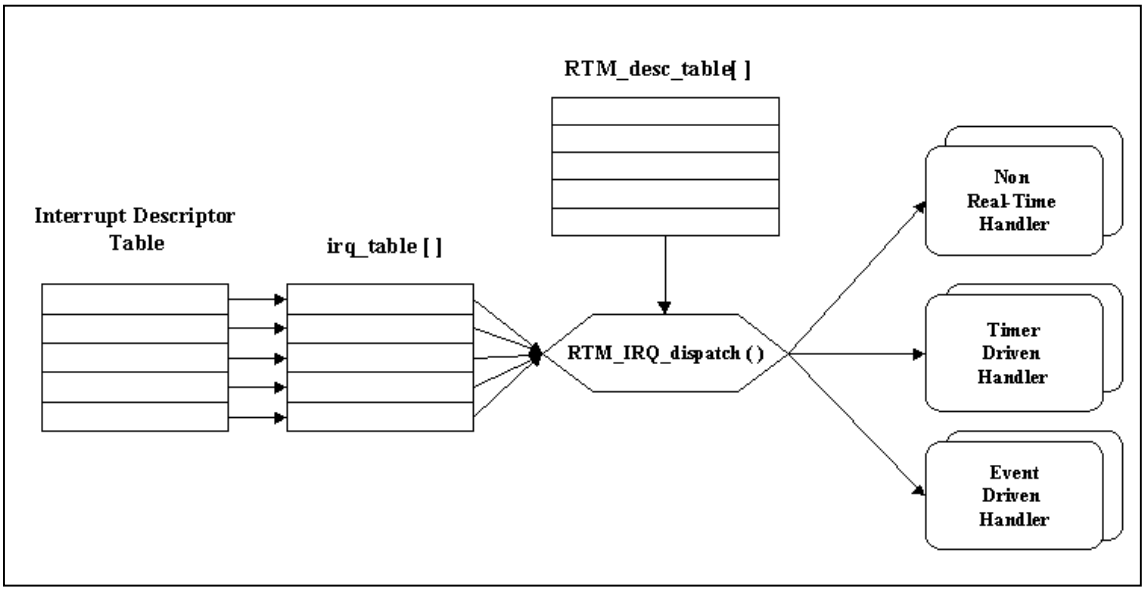

Figure 2

Changes to standard MINIX are minimal with the Virtual Machine approach. This low level of intrusion on the standard MINIX kernel improves the code maintainability to keep the Real-Time modifications up-to-date with the latest release of the MINIX kernel.

RT-MINIXv2 avoids disabling interrupts for extended periods of time to improve system responsivity. Its design algorithms and data structures can be used with interrupts enabled or they are disabled only for very short intervals.

RT-interrupt handlers can easily be replaced with standard ones. This is especially useful in certain debugging situations.

A drawback is that the MINIX kernel suffers a slight performance loss when RT-MINIXv2 Virtual Machine is added due to the redirection (through pointers), to the interrupt mask/unmask functions. In consideration of both strengths and weaknesses, this technique has shown itself to be both efficient and flexible because it removes none of the capability of standard MINIX, yet it provides guaranteed scheduling and response time for critical tasks.

2.3.1. RT-MINIXv2 microkernel interrupt dispatcher

RT-MINIXv2 microkernel traps all interrupts and, depends on the interrupt type, dispatchs the apropiated handler or registers its occurence for delayed processing.

2.3.2. Standard MINIX Non Real-Time interrupts

When the RT-interrupt dispatcher is invoked by a NRT-interrupt, it inserts the interrupt descriptor in interrupt pending queue *RTM_intQ[]*. There are *RTM_NR_PRTY* queues, one for each priority level.
The RT-scheduler is not invoked when the handler returns if a RT-process or a RT-ISR was interrupted by a non RT-interrupt.
When the running RT-process or RT-ISR leaves the CPU, a routine named *RTM_flush_int()* is called. All delayed interrupts are checked and the handlers are executed in priority order.
Suppose that the keyboard is specified as a NRT-device with priority 14. The keyboard uses IRQ level 1 in a standard PC hardware. When the user do a keystroke, the keyboard hardware interrupts the CPU an the RT-interrupt dispatcher is called that insert the interrupt descriptor *RTM_desc_table[1]* in the 14th interrupt pending queue *RTM_intQ[14]*.
Next, when all RT-processes of higher priority will be blocked, all NRT-interrupts are flushed, the Standard MINIX keyboard interrupt handler is called and descriptor is removed from the queue.

2.3.3. Real-Time Timer-Driven Interrupts

Some devices can be attended in a Timer-Driven manner. If the device does not raise an interrupt, a periodic process can be created to poll the device checking the status and taking an action.
Some other devices will raise interrupts but the interrupt processing may be delayed to be managed by a periodic process in the next schedule.
When the RT-interrupt dispatcher is invoked by a Timer-Driven interrupt, it inserts the interrupt descriptor in the interrupt pending queue *RTM_intQ[]*.
On the next timer interrupt, the Timer RT-handler calls *RTM_flush_int()* that scans the interrupt pending queue in priority order. The Timer Driven interrupt handler is called only at the end of each period and if the priority of the interrupt is greater than the process/handler that has been interrupted. Otherwise it must wait until all process/handlers with higher priorities will finish their processing or at the end of its current period .

2.3.4. Real-Time Event-Driven Interrupts

When the RT-interrupt dispatcher is invoked by a Event-Driven interrupt, it checks if the priority of the interrupted process/handler is greater than the priority of the interrupt..
If the interrupted process/handler priority is greater than the interrupt priority, the dispatcher inserts the interrupt descriptor in interrupt pending queue *RTM_intQ[]*.
Next, when all higher priority processes/handler will be blocked, all Event Driven interrupts are flushed in priority order until another higher priority process will be ready.

If the interrupt priority is greater than the interrupted process/handler priority, its RT-handler is invoked with minimal latency.

2.4 Interrupt Handler Dispatcher

The function RTM_IRQ_dispatch() attends all hardware interrupt types running the interrupt handler or registering the interrupt for delayed processing.

```
/*===========================================================================*
 *                              RTM_IRQ_dispatch                             *
 * This is the common Hardware Interrupt Handler pointed by all IRQ vectors  *
 * for Timer Interrupts:                                                     *
 *      increase RTM_counter.ticks                                           *
 *      enqueues the interrupt descriptor in a interrupt pending queue       *
 * for Event Driven Interrupts:                                              *
 *      runs the handler only if it's priority is greater than the           *
 *      current interrupted process                                          *
 * for Timer Driven and Non Real Time Interrupts:                            *
 *      enqueues the interrupt descriptor in a interrupt pending queue       *
 *===========================================================================*/
PRIVATE int RTM_IRQ_dispatch(irq)
int irq;
{
        int pty,  retval;
        scounter_t before;
        RTM_irq_desc_t  *d;

        if (irq < 0 || irq >= NR_IRQ_VECTORS)  panic("invalid call to RTM_dispatch. IRQ= ", irq);

        /* interrupts are enabled, disabled them */
        RTM_lock();                     /* ENTER_CRITICAL_SECTION */
        d = &RTM_desc_table[irq];

        if( RTM_RTswitch == RTM_STDMODE)    /* Processing MODE?  */
                {                           /* For Standard processing mode */
                RTM_unlock();               /* enable interrupts */
                retval = d->nrthandler(d->irq);  /* run the Non Real-Time Handler */
                RTM_lock();                 /* disable interrupt */
                return(retval);             /* return to restart() */
                }

        RTM_counter.interrupts++;           /* update the global interrupt counter */
        d->count++;                         /* update the irq descriptor interrupt counter */

        if ( irq == CLOCK_IRQ)              /* for timer Interrupts... */
                {
                RTM_IRQ_latency = RTM_hz_elapsed();     /* saves the IRQ latency */

                RTM_IRQ_maxlat  =
                        MAX(RTM_IRQ_latency,RTM_IRQ_maxlat); /* update the maximun latency */

                RTM_counter.ticks++;                /* update the timer interrupt counter */

                if(RTM_counter.ticks == 0 )         /* Does the counter walk arond  */
                        RTM_counter.highticks++;    /* uptdate the more significative timer int counter */
                d->timestamp = RTM_counter.ticks;   /* stamp the tick counter in the int. descriptor */
```

```
                if(RTM_flush_lock == RTM_FLUSH_ENABLED)  /* Can flush pending interrupts ? */
                RTM_irqQ_insert(d);             /* enqueue the descriptor for delayed processing */
                RTM_flush_int();                /* YES, flush then */
                return(RTM_IRQ_DISABLED);       /* return to restart */
                }

        /* for all interrupt types, except timer interrupt */
        d->timestamp = RTM_counter.ticks;       /* stamp the tick counter in the int. descriptor */

        if( (d->irqtype & RTM_EDINT)                    /* Event Driven interrupt with   */
                && (d->priority < RTM_p_epri(proc_ptr))) /* higher priority than interrupted process */
                {
                RTM_do_handler(d);              /* run the handler and more */
                if( d->flags & RTM_RESCHED )/* does the handler could change the current process */
                        RTM_pick_proc();        /* call the Real-Time Scheduler */
                }
        else    /*  Timer Driven ints, Non Real-Time Ints and Event Driven (with lower priority)  */
                {
                RTM_irqQ_insert(d);             /* enqueue the descriptor for delayed processing */
                }

        if(RTM_flush_lock == RTM_FLUSH_ENABLED)     /* Can flush pending interrupts ? */
                RTM_flush_int();                    /*YES, flush then */

        /* hwint00 - 15 wait interrupts to come disabled */
        return(RTM_IRQ_DISABLED);               /* return to restart */
}
```

2.5 Returning from Interrupts, Exceptions and System Calls

We will examine the termination phase of interrupt/exception handlers and system calls. The main objetive is to execute the highest priority process, but several issues must be considered before doing it:

- Pending Event Driven Interrupt handlers to be executed
- Pending Software IRQ handlers to be executed
- Pending Timer Driven Interrupts handlers
- Pending Standard MINIX interrupt handlers

Only that pending issues that have a greater priority than the current running process are executed. The kernel function that accomplish with this issues is *RTM_flush_int()*.

INSERTAR DIAGRAMA DE FLUJO Y EXPLICARLO

TAMBIEN PONER PARTE DE CODIGO ASSEMBLER DE _restart()

2.6 Real-Time Processing

In [4] the authors classify Real-Time implementations in two categories:

1. ***Event Driven Implementation***

- *Integrated Interrupt Event Driven Scheduling:* is integrated in the sense that hardware interrupt priorities are matched with the software process priorities. All process are initiated by external interrupts.
- *Non-Integrated Interrupt Event Driven Scheduling:* the priority of the interrupt associated with process arrival has no correspondence to the software priority of the process, an is thus non-integrated.

2. ***Timer Driver Implementation***
   - *Timer Driven Scheduling:* A timer expires every T_{tic} seconds causing a non-maskable interrupt that force a scheduling point. The scheduler moves all process that have next scheduling points greater or equal than the current time to the ready queue.
   - *Timer Driven Scheduling with counter:* The Timer handler decrements a counter on every timer interrupt and will only invoke the scheduler when the counter expires. The counter limits the scheduler to run only on timer interrupts that correspond to process arrivals.

RT-MINIXv2 does not match strictly in any of these categories but it depends on the set of process running on the system and the type of hardware where it runs.
We could consider RT-MINIXv2 as an *Event Driven with Non-Integrated Interrupts Scheduling and Timer Driven with counter* system.

- *Event Driven:* On an Evend Driven interrupt the handler is called without delay.
- *Non-Integrated Interrupts:* Hardware interrupts priorities could no match with processes priorities.
- *Timer Driven with counter:* The RT-Timer interrupt handler invokes the scheduler only when a Timer Driven process must be scheduled or when Timer Driven interrupt have occured in the last timer period.

2.7 Real-Time Process Dispatch Latency Time

A Real-Time process dispatch latency time is the agreggation of [29]:

1. Interrupt response time that includes:
   - Hardware delay
   - Completion of current instruction
   - Interrupt latency
2. Interrupt routine
   - *preprocessing:* This time includes the processing costs of *RTM_IRQ_dispatch()*.
   - *interrupt servicing*: This is the time consumed by the handler itself.
   - *post processing*: this time includes the processing costs of *RTM_flush_int()*.
3. Software recognition of the need for context switch: this time includes the processing costs of *RTM_pick_proc()*.
4. Context switch

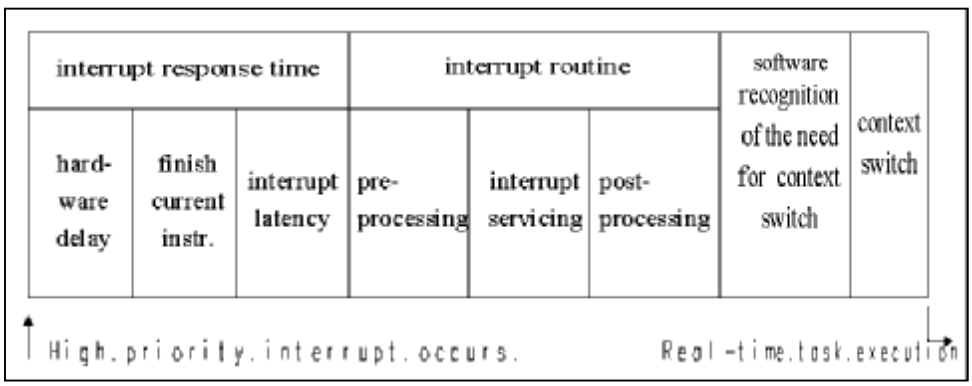

Figure 3– (from [29])

2.8 Timer Driven Latency and Processing Costs

The next could be a sequence of events showing the processing costs and source of latency for a Timer Driven interrupt.
The processes priorities assumed are:

$$P_{RT\text{-}Timer} < P_{TD} < P_{USER}$$

$P_{RT\text{-}Timer}$ = RT-Timer interrupt handler priority
P_{TD} = Timer-Driven interrupt handler priority
P_{USER} = A standard USER process priority
Note: higher priority number indicates lower priority level

1. A USER process is running.
2. The CPU receives a Timer-Driven interrupt and the kernel saves USER process's context.
3. *RTM_IRQ_dispatch()* inserts the interrupt descriptor in the interrupt pending queue *RTM_intQ[P_{TD}]*.
4. On return, *RTM_flush_int()* is called and because there are no higher priority interrupts pending the kernel restore USER process's context.
5. The USER process continues running.
6. The CPU receives a timer interrupt and the kernel saves the USER process's context.
7. *RTM_IRQ_dispatch()* increases the tick counter *RTM_counter.ticks* and calls the RT-Timer handler.
8. On return, *RTM_flush_int()* is called. For each Timer Driven pending interrupt priority two conditions must be met to invoke the handlers:
    A. The priority of the interrupt handler must be greater than the interrupted USER process priority.
    B. The last period of the Timer Driven interrupt priority has finished.

    For this sample the Timer Driven Interrupt handler is called.
9. After each call, the RT-scheduler is invoked to compare priorities among the handler and the next current process (*proc_ptr*). If the next current process priority is greater than all pendings interrupts, its suspends the invocation of the remainding handlers. All pending interrupts will be flushed by *RTM_flush_int()* later.

The interrupt blocking time is reduced because the processing is done out of interrupt time delayed until the next Timer interrupt. This fact could increase latency but enhance preemptability.

2.9 Event Driven Latency and Processing Costs

The next is a possible sequence of events to show the processing costs and source of latency of a Event-Driven interrupt.
The processes priorities assumed are:

$$P_{RT\text{-}Timer} < P_{ED} < P_{USER}$$

$P_{RT\text{-}Timer}$ = RT-Timer interrupt handler priority
P_{ED} = Event-Driven interrupt handler priority
P_{USER} = A standard USER process priority
Note: higher priority number indicates lower priority level

1. A USER process is running.
2. The CPU receives an Event-Driven interrupt and the kernel saves the USER process's context.
3. *RTM_IRQ_dispatch()* compares priorities among the current process (*proc_ptr*) and the Event-Driven Interrupt handler. If the priority of the interrupt handler is greater than the priority of the current process the handler is called, else inserts the interrupt descriptor in the interrupt pending queue *RTM_intQ[P_{ED}]* to be processed later when pending interrupts are flushed by *RTM_flush_int()*.

4. If the priority of the current process is greater than the Event-Driven interrupt, the blocking time is reduced because the processing is done out of interrupt time delayed until all higher priority interrupts and processes are blocked. This fact reduce the latency and enhance preemptability.

2.10 PIC and Interrupt Masking Virtualization

One of the problems with doing hard real-time on a standard MINIX system is the fact that the kernel uses disabling interrupts as a means of synchronization and to avoid race conditions on kernel variables. Promiscuous use of disabling and enabling interrupts inflicts unpredictable interrupt dispatch latency.

In RT-MINIXv2, this problem is solved by putting a layer of emulation software between the MINIX kernel and the interrupt controller hardware.

These virtualizations are quite simples because MINIX use the following functions:

- *lock():* to disable maskable interrups (CLI for Intel x86)
- *unlock():* to enable maskable interrupts (STI for Intel x86)

RT-MINIXv2 change the behavior of this functions and simulate then for MINIX kernel. Whenever an NRT-interrupt happens, a bit in a bitmap is set by *RTM_IRQ_dispatch()* as is explained in Standard MINIX Non Real-Time interrupts. Later, *RTM_flush_int()* is called, and if MINIX has virtualy disabled interrupts using *lock()*, or the IRQ has been disabled in the virtual PIC with *disable_irq()*, the standard MINIX handler of the interrupt will not be executed.
When MINIX kernel re-enables interrupts using *unlock()*, all pending interrupts are executed in the next *return from system call* or *return from interrupt.*

2.11Interrupt Priorities

The PIC treats interrupts according to their priority level. This is a function of which interrupt line they use to enter the interrupt controller. For this reason, the priority levels are directly tied to the interrupt number. Higher-priority IRQ may improve the performance of the devices that use them.
The PC hardware have assigned priorities for standard interrupts related to IRQ number as shown in Figure 4. A lower IRQ number implies higher priority.

RT-MINIXv2 manages interrupts using a priority field in the interrupt descriptor *RTM_irq_desc_t*.
When the kernel services a hardware interrupt, it sets the interrupt controller to mask out lower priority interrupts using the mask field.
If the interrupt cannot be serviced because it has interrupted a higher priority process, the interrupt descriptor is inserted in one of the interrupt pending queues (*RTM_intQ[irq_priority])* and a bit is set in *RTM_irqQ_bitmap* as show in Figure 5.

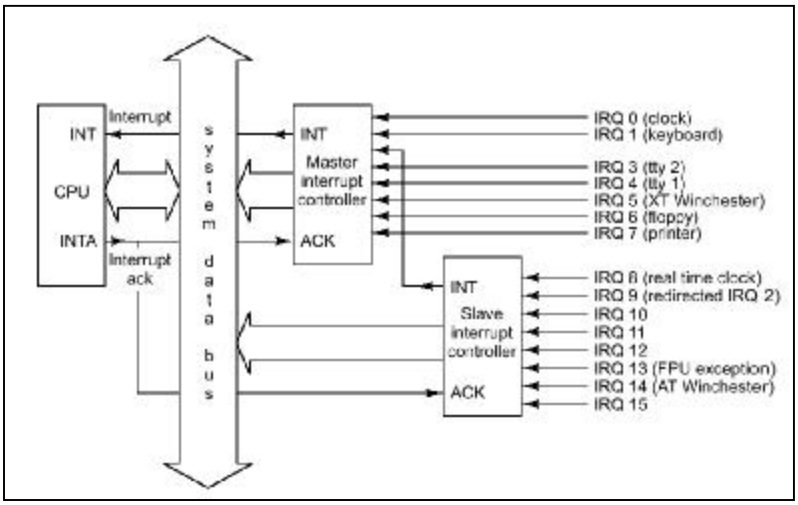

Figure 4
from Operating Systems: Design and Implementation, 2nd ed.

RTM_flush_int() scans *RTM_irqQ_bitmap* for the first non zero bit until the priority of the current process. If it finds pending interrupts, it invokes their handlers and the bit in the bitmap is reset.

Not all pending interrupts handlers in the queue are executed by *RTM_flush_int(),* exceptions are:

- Non Real Time pending interrupts when MINIX has disabled interrupts (virtually).
- Non Real Time pending interrupts when MINIX has disabled the IRQ in the PIC.
- Timer Driven pending interrupts that have not reach their launching period.
- Periodic software IRQs that have not reach their launching period.
- All pending interrupts with lower priority than the current process *proc_ptr*.

The "next" current process (*proc_ptr)* is evaluated (*RTM_pick_proc()*) after running each handler because it could change the state of a higher priority process from RT-BLOCKED to RT-READY.

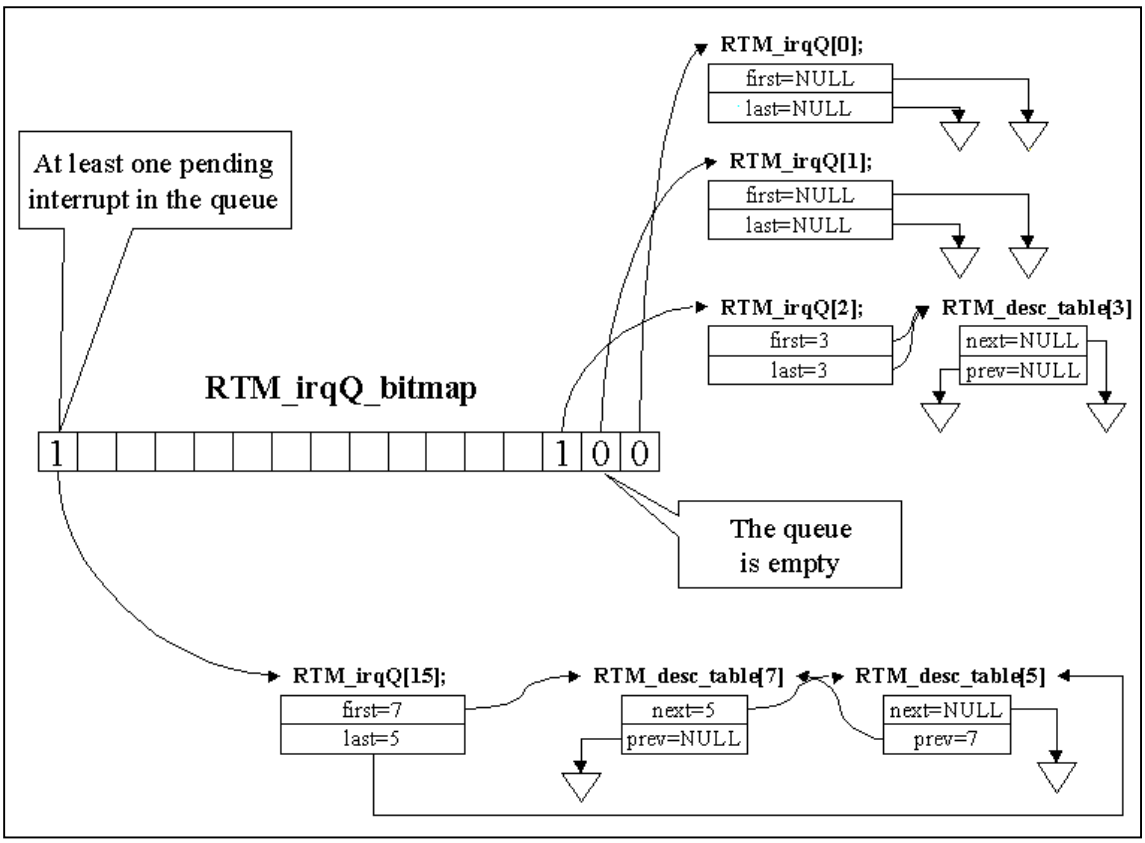

Figure 5

2.12 Preventing from interrupt showers

In Real-Time systems unpredictability is introduced by interrupts from some devices.
A type of unbounded priority invertion is produced when a higher priority process is executing and a lower priority hardware device produce an interrupt shower.

We describe the behavior of some common PC interrupts.

- ***Timer Interrupt:*** It is a periodic interrupt type generated by the PIT (Programmable Interrupt Timer). The kernel only need to set the period once.
- ***Kerboard Interrupt:*** It is an aperiodic, asynchronous, and input interrupt type. It's occurs when the user press or release a key. It is common to use a buffer to keep the typed keys until the keyboard interrupt handler can process it.
- ***Diskette/Disk Interrupt:*** Is an aperiodic input/output interrupt. The interrupts occurs when the hardware has finished with a diskette/disk command. The commands sent to the device are as requests and the interrupts are as replies, then they are synchronized with the operating system.
- ***RS232 Interrupts:*** produce tree types of interrupts when:
  - ü a character is received in the communication port (sporadic).
  - ü an error or a status change occurs in the communication port (sporadic).
  - ü a character has been sent by the communication port. If no character is sent to the RS232 port, no interrupts of this type occurs. Therefore they are syncronized with the operating system.
- ***Network Interrupts****:* produce tree types of interrupts:
  - ü when a packet is received from the network device (sporadic).
  - ü when an error or a status change occurs in the network device (sporadic).
  - ü when a packet has been sent by the network device. If no packet is sent to the network, no interrupts of this type occurs. Therefore they are syncronized with the operating system .

Because it is common that drivers tasks are not reentrant, they will not send new requests to the hardware until they finish with the last interrupt servicing, therefore no synchronous interrupts can occur and the system could only receive asynchronous interrupt showers.

To limit interrupts showers all processes descriptors have a hardware interrupt mask. To reduce the priority inversion all interrupt descriptors have an interrupt mask too. These masks could filter some hardware interrupts types avoiding that they could occurs while the process or interrupt handler are running. The default value for that masks is set to permit all types of hardware interrupts.

2.13 Interrupt Timestamps

RT-MINIXv2 includes a timestamp field in the interrupt descriptor *RTM_irq_desc_t* that is set by *RTM_IRQ_dispatch()*.
The units of the timestamp field are Real-Time ticks since the system startup or since the last *rtm_restart()* system call execution.
RTM_flush_int() uses the interrupt timestamp field to decide for the invocation of Timer-Driven interrupt handlers.

2.14 Software IRQs

One of the main problems with interrupt handling is how to perform longish tasks within a handler.
Often a substantial amount of work must be done in response to a device interrupt, but interrupt handlers need to finish up quickly and not keep interrupts blocked for long. These two needs (work

and speed) conflict with each other, leaving the driver writer in a bit of a bind. Therefore it is desirable that the interrupt handlers could delay the execution of some tasks so that they don't block the system for too long.

As it is explained in 2.1, MINIX uses splitted interrupt management where an interrupt handler partially process the interrupt and then send a message to a device driver task to resume the interrupt processing. This approach implies at least a context switch to restore the TASK state.

LINUX kernel resolves this problem by splitting the interrupt handler into two halves. The so-called top half is the routine that actually responds to the interrupt. The bottom half is a routine that is scheduled by the top half to be executed later, at a safer time.

In RT-MINIXv2, device driver writers could decide among three approaches:

- Complete interrupt processing in the handler.
- Splitted interrupt management like MINIX using TASKs and message transfers.
- Splitted interrupt management using Software IRQs

Software IRQs are kernel routines that are invoked by *RTM_flush_int()* as it do with pending hardware interrupts.

Software IRQ descriptors are the same as Hardware Interrupts descriptors, therefore they have priority, mask, counter and timestamps fields.

The motivation for introducing software IRQs is to allow a limited number of functions related to interrupt handling to be executed in a deferred manner. This increase the system responsiveness because some work is executed out of interrupt time. Addicionally the processing overhead is lower than using the TASK model because it avoids the context switch among the interrupted process and a TASK and does not need of message transfers.

The kernel functions that operate on software IRQs are:

- *RTM_set_softirq(p_irqd):* assigns a software IRQ descriptor and set it with the parameters passed in *p_irqd*.
- *RTM_free_softirq(irq):* frees a software IRQ.

An interrupt handler could activate a software IRQ inserting its descriptor in the interrupt pending queue *RTM_intQ[]* as *RTM_irq_dispatch()* do with hardware interrupts.

3. PROCESS MANAGEMENT AND SCHEDULING

3.1 Real-Time Process Creation

RT-MINIXv2 does not have new system calls to create a Real-Time process. All Real-Time processes are created transforming a standard MINIX processes into a RT-MINIXv2 process using the *rtm_setproc()* system call.

To convert a standard MINIX process in a RT-MINIXv2 process, one of the Real-Time bits is set in the process's *p_flags*, and the descriptor is removed from any MINIX ready queue using the *unready()* function when the process calls *rtm_setproc()*.

Because there are two process schedulers that run on RT-MINIXv2 (the Real-Time and the standard MINIX scheduler), there must be a way to distinguish among Real-Time and Non Real-Time processes.

3.2 Process States and States Transitions

As there are two kernels that can operate on the same set of processes, there are two dimensions of states: one for standard MINIX processes and other for Real-Time ones.

The following are the possible process states for a standard MINIX process(Figure 6):

- READY: The process is ready to run and waiting to be selected by the standard MINIX process scheduler.
- BLOCKED : The process is blocked trying to send/receive a message.
- BLOCKED REALTIME: The process is a Real-Time one. It must be ignored by the MINIX process scheduler.
- RUNNING: The process is running under MINIX kernel control.

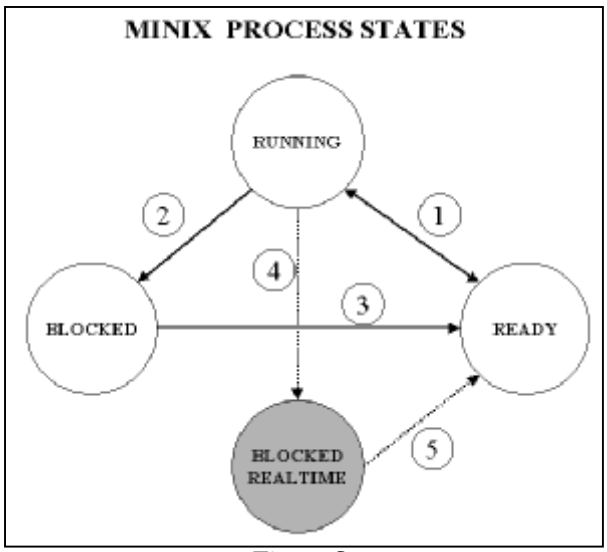

Figure 6

The process state transitions are:

1. READY to RUNNING: The process has been selected to run by the MINIX process scheduler. RUNNING to READY: The process has been preempted by a higher priority process or by the kernel when the process's timeslice reaches zero.
2. RUNNING to BLOCKED:The process has done a blocking system call.
3. BLOCKED to READY: The system has finished the process system call.

4. READY to BLOCKED REALTIME: The process has done a *rtm_setproc()* system call and has changed the process type to *RTM_P_PERIODIC* or *RTM_P_SPORADIC*.
5. BLOCKED REALTIME to READY: The process has done a *rtm_setproc()* system call and has changed the process type to *RTM_P_STANDARD*.

The following are the possible process states for a RT-MINIXv2 process (Figure 7):

- RT-READY: The process is ready to run and waiting to be selected by the RT-MINIXv2 process scheduler.
- RT-BLOCKED : The process is blocked trying to send/receive a RT-message.
- RT-BLOCKED STANDARD: The process is a standard one. It must be ignored by the RT-MINIXv2 process scheduler.
- RT-RUNNING: The process is running under RT-MINIXv2 kernel control.

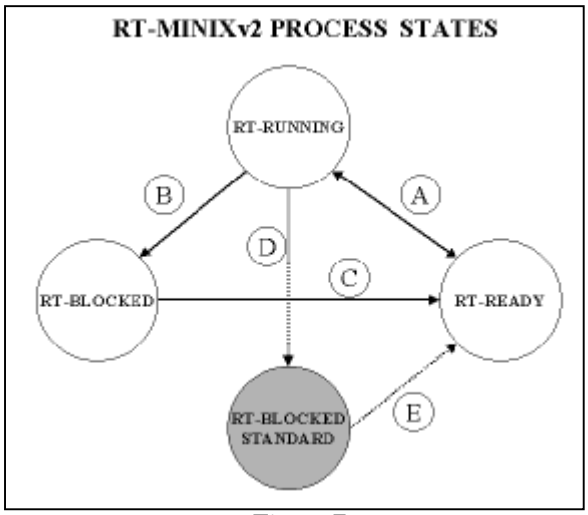

Figure 7

The process state transitions are:

A. RT-READY to RT-RUNNING: : The process has been selected to run by the RT-MINIXv2 process scheduler. RT-RUNNING to RT-READY: The process has been preempted by higher priority process.
B. RT-RUNNING to RT-BLOCKED:The process has done a blocking Real-Time system call.
C. RT-BLOCKED to RT-READY: The system has finished to process the Real-Time system call.
D. RT-READY to RT-BLOCKED STANDARD: The process has done a *rtm_setproc()* system call and has changed the process type to *RTM_P_STANDARD*.
E. RT-BLOCKED STANDARD to RT-READY: The process has done a *rtm_setproc()* system call and has changed the process type to *RTM_P_PERIODIC* or *RTM_P_SPORADIC*.

Really, two states does not exist as shown in Figure 8. They are RT-BLOCKED STANDARD and BLOCKED REALTIME.

- BLOCKED REALTIME: is the set of RT-MINIXv2 process states.
- RT-BLOCKED STANDARD: is the set of MINIX process states.

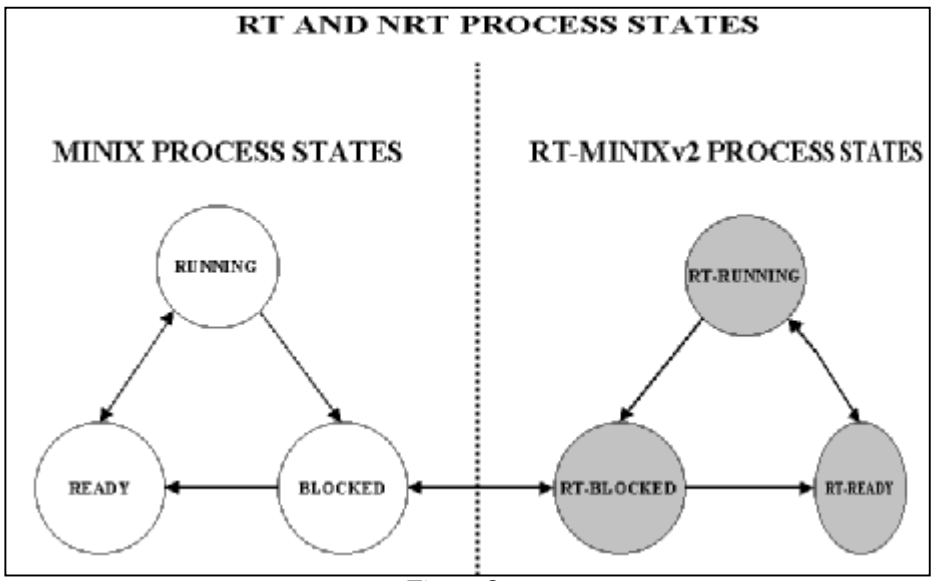

Figure 8

3.3 Process Descriptor Real-Time fields

The standard MINIX kernel uses a process descriptor table to keep the status information of every process in the system.

Each process descriptor have a field named *p_flags* to indicates the reason why a process is blocked. If *p_flags = 0*, the process can be scheduled by the standard MINIX process scheduler.
The meanings of each *bit in p_flags* are listed in *Appendix A*.

RT-MINIXv2 defines *two new bits* in *p_flags* for Real-Time processes to avoid that they can be scheduled by the standard MINIX kernel. Therefore, the RT-MINIXv2 process scheduler can only schedule processes with one of this new bits in ON(1) and the other bits in OFF(0).

To check for a RT-process the RT-kernel uses the macro *RTM_is_realtime(X)*.
To check for a ready RT-process the RT-kernel uses the macro *RTM_is_rtready(X)*.

RT-MINIXv2 adds *new fields* to the process data structure for Real-Time process management and statistics collection. Some of these fields are RT-process characterization *parameters*:

- The scheduling period for a RT-periodic process
- A limit for the number of RT-schedulings.
- The process deadline.
- The process latency.
- The process laxity.

RT-MINIXv2 kernel only uses the period field for RT-periodic processes.
The period and deadline fields are specified in Real-Time timer ticks or RT-ticks.
The maxlat and minlax fields are filled by the system on each process scheduling. More about this parameters is detailed in the *Statistics Collection* chapter.

When a RT-process does not complete it's work until it's deadline, the RT-kernel can send a MT_WATCHDOG message to a watchdog process specified in the process descriptor.

3.4 The Real-Time Process Scheduler

The process scheduler is the component of the kernel that selects which process to run next. The scheduler can be viewed as the code that divides, using a defined policy, the finite resource of processor time between the runnable processes on a system.

Policy is the behavior of the scheduler that determines what runs when. A scheduler's policy often determines the overall feel of a system and is responsible for optimally utilizing processor time.
The policy behind a Real-Time scheduler is simple:

"A priority scheduled Real-Time system must ensure that the highest priority runnable process can start to run in a bounded time—and the bound needs to be small.".

The RT-MINIXv2 process scheduler always selects the highest priority runnable process for execution. All unvoluntary context switches are triggered by interrupts. Timer interrupts can cause preemption due to Timer-Driven process activation. If the priority of the activated process is higher than the priority of the current process, the current process is preempted. When there is no Real-Time process ready to run, the MINIX scheduler is invoked.

The RT-MINIXv2 process scheduler (*RTM_pick_proc()*) tries to find a ready RT-process with the highest priority. If there are not such process, the standard MINIX process scheduler is called (*pick_proc()*).

The RT-scheduler uses an optimized process-selection algorithm, based on a set of 16 priority queues and a 16 bits bitmap [9]. Each bit in the bitmap represents a priority queue. If a bit is set, it means that at least one process is ready in that queue.
Typically, the bit-map is scanned for the highest priority non-empty queue, and the first process in that queue is selected to run.

The RT-scheduler implements fully O(1) scheduling. The algorithm completes in constant-time, regardless of the number of running processes.

In standard MINIX (and other time-sharing operating systems), the timeslice is the numeric value that represents how long a process can run until it is preempted.
RT-MINIXv2 does not use a timeslice for preempt a process. Only a higher priority process can preempt the running process or it must relinquish the CPU by itself.
When a higher priority process enters the RT-READY state, the kernel checks whether its priority is higher than the priority of the currently executing process. If it is, the scheduler is invoked to pick a new process to run (presumably the process that just became runnable).

3.5 Process Priority.

A common type of scheduling algorithm is priority-based scheduling. The idea is to rank processes based on their worth and need for processor time. Processes with a higher priority will run before those with a lower priority, while processes with the same priority are scheduled round-robin (one after the next, repeating).

The standard MINIX kernel uses 3 priority queues to schedule a process as it is shown in Figure 9:

- ***TASK_Q:*** for device drivers or TASKS.
- ***SERVER_Q:*** for SERVERs like Memory Manager (MM) and File System Manager (FS).
- ***USER_Q:*** for USER processes.

The *TASK_Q* has more priority than the *SERVER_Q* that has more priority than the *USER_Q*.
All process in the same queue have the same priority and the process scheduler selects the next process to run in FCFS order for TASKs and SERVERs and in FIFO order for USER processes.

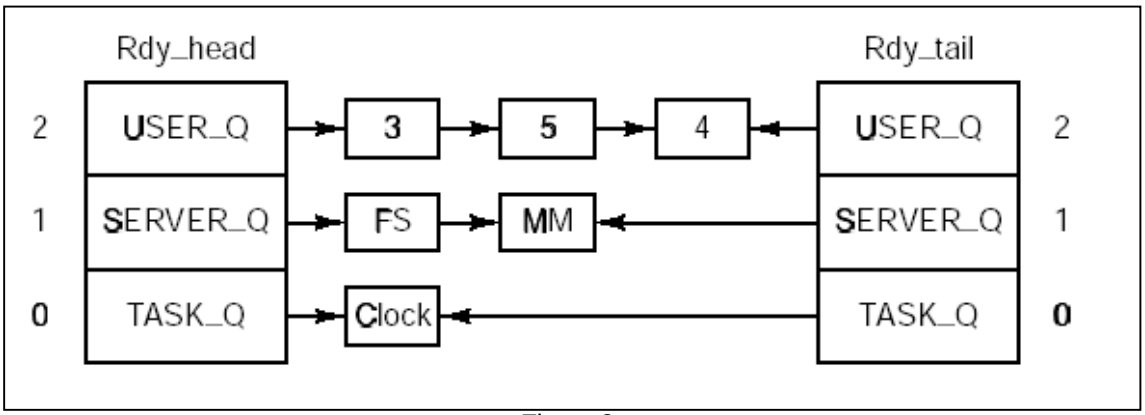

Figure 9
from Operating Systems: Design and Implementation, 2nd ed.

RT-MINIXv2 provides static priority-based scheduling with *RTM_NR_PRTY* (16) priority queues, therefore a new field was added in the process descriptor named priority. This field is used for:

A. Store the Real-Time ready queue number for the process in the RT-READY state.
B. Reduce the Interrupt Blocking time for the process because only interrupt handlers with higher priorities are executed during process running.
C. To implement the Priority Inheritance Protocol for RT-IPC.

The number of the Real-Time ready queue is defined by the last four bits of priority field named its Effective Priority(*epri*).

7	6	5	4	3	2	1	0
bpri = Base Priority				**epri = Effective Priority**			

The first four bits of the priority define the process Base Priority (*bpri*). On process creation, *epri* = *bpri*.. The RT-kernel only considers the Effective Priority (*epri)* to put a process in one of it's ready queues. The Base Priority (*bpri)* is used by IPC primitives to return the Effective Priority *(epri)* at is original value after using the priority inheritance protocol (PIP). More about IPC are detailed in the Real-time Interprocess Communications chapter.

3.6 Priority Queues Management

RTM_NR_PRTY is the number of priority levels on the system. By default, this is 16. Thus, there is one RTM_priQ_t for each priority.

RTM_priQ_bitmap is a *unsigned int* variable that have one bit for each valid priority level.
Initially, all the bits are zero. When a process of a given priority becomes runnable (that is, its state becomes RT-READY), the corresponding bit in the bitmap is set to one.
Finding the highest priority process on the system is therefore only a matter of finding the first set bit in the bitmap. Because the number of priorities is static, the time to complete this search is constant and unaffected by the number of running processes on the system.

Each priority queue descriptor RTM_priQ_t have two pointers, one for the head and one for the tail of the queue. The insertions in the queue can be in FIFO or LIFO order. Processes of the same priority will be managed under a FIFO policy.
Each descriptor also contains a counter, *inQ*. This is the number of runnable process in this priority queue.

A process that inherits priority by RT-IPC must be inserted and removed from the queues in LIFO order as it is explained in the RT-IPC chapter [10].

These kernel functions help to manage priority queues.

- *RTM_priQ_fifo()* : FIFO insertion in a priority queue.
- *RTM_priQ_lifo() :* LIFO insertion in a priority queue.
- *RTM_priQ_rmv() :* remove a process from a priority queue.

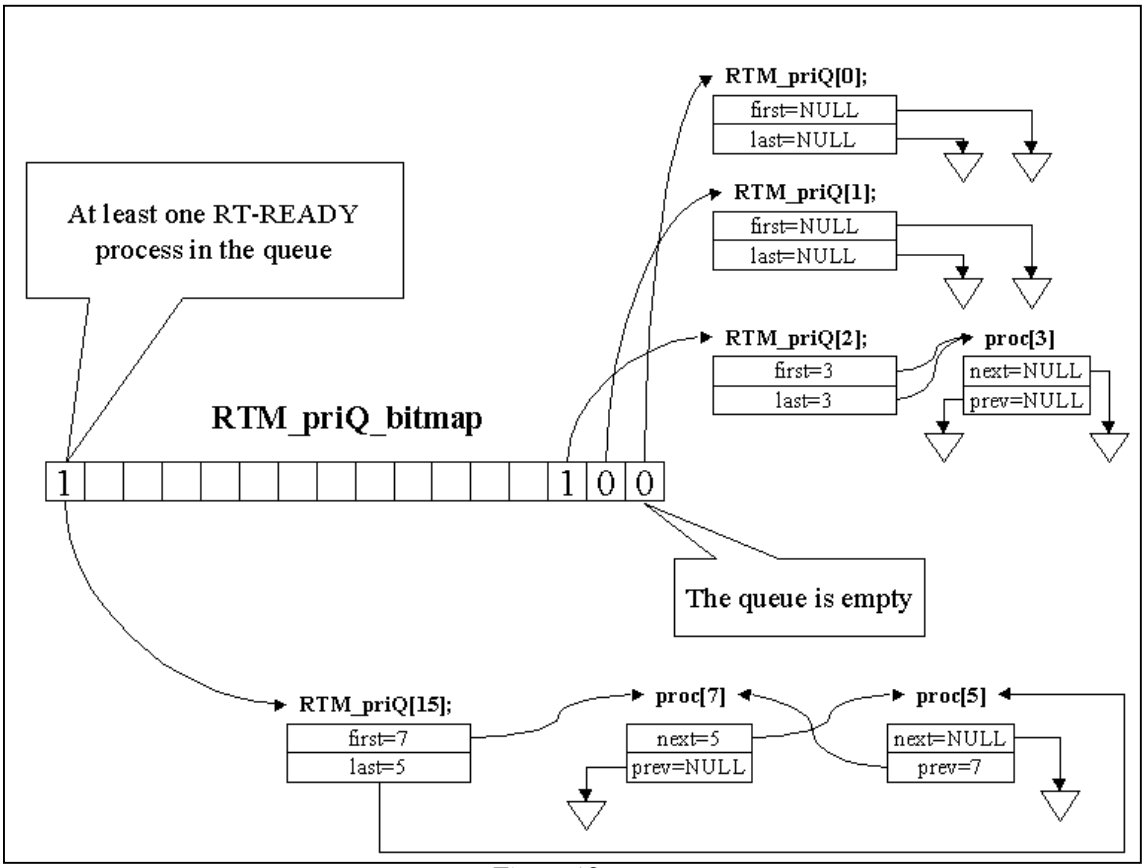

Figure 10

Several kernel functions operate on priority queues:

- ***RTM_pick_proc():*** It is the RT-scheduler. It selects the highest priority ready-to-run process.
- ***RTM_ready():*** It changes the state of a process to the RT-READY, appending the descriptor at the end of the priority queue using *RTM_priQ_fifo()*.
- ***RTM_unready():*** It changes the state of a RT-READY process to a RT-BLOCKED state and removes it from the priority queue using *RTM_priQ_rmv()*.
- ***RTM_pri_inh() function:*** RT-IPC could change the priority of the destination process of a message. It uses *RTM_priQ_rmv()* and *RTM_priQ_lifo()*.
- ***The RTM_pri_reset() function:*** change the priority of a process when it has done a reply. It uses *RTM_priQ_rmv()* and *RTM_priQ_lifo()*.
- ***The RTM_setpri():*** set the process priority using *RTM_priQ_rmv()* and *RTM_priQ_fifo()*.

3.7 RT-process Termination

Para terminar un proceso RT es conveniente retornarlo a NRT usando *rtm_setproc()*.

Si un proceso NRT hace un kill de un proceso RT debería interceptarse la llamada

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                              src/mm/forkexit.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
.....
16924   /*===========================================================================*
16925    *                                mm_exit                                    *
16926    *===========================================================================*/
........
16948     /* Tell the kernel and FS that the process is no longer runnable. */
Aqui habria que informar a la tarea RTMTASK que convierta al proceso nuevamente en NRT.
16949     tell_fs(EXIT, proc_nr, 0, 0);  /* file system can free the proc slot */
16950     sys_xit(rmp->mp_parent, proc_nr, &base, &size);



++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                              src/mm/signal.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
18262   /*===========================================================================*
18263    *                                check_sig                                  *
18264    *===========================================================================*/
18280     /* Return EINVAL for attempts to send SIGKILL to INIT alone. */
18281     if (proc_id == INIT_PID && signo == SIGKILL) return(EINVAL);
Aqui tambien se podría retornar EINVAL si se intenta enviar cualquier señal a un proceso RT
```

4. TIME MANAGEMENT

4.1 Timer Management Design Patterns [31] [32]

Timer strategies play an integral role in Real-Time systems.
The following timer management design patterns are used very frequently in Real-time systems:

- A *pause()* function
- Recovering from Message Loss
- Recovering from Software Faults
- Sequencing Operations
- Polling
- Periodic Operations
- Failure Detection
- Collecting Messages
- Inactivity Detection

A *pause()* function
A function *pause()* is used to suspend the active task for a specified number of milliseconds. As is explained in [32], an inaccuracy could occur because the pause function use the timer interrupt as is time base and it depends on the timer resolution.

Recovering from Message Loss
Usually a timer is kept while awaiting for a message. If the message is received, timer is stopped. If the timer expires, message loss is registered. In such a case, a retry logic is implemented by restarting the timer and awaiting for the message again. If the number of retries reaches a threshold, the activity is aborted and appropriate recovery action is initiated.

Recovering from Software Faults
Whenever a feature is initiated, a feature wide timer is kept to ensure feature success. If some software or hardware module involved in the feature hits recovery, the feature will fail and the timer expiry will be the only method to detect the feature failure. On expiry of the feature wide timer, the feature may be reinitiated or recovery action might be taken.
It is a good design practice to keep at least one timer active during the execution of a feature. The timer guards the system against unforeseen failure conditions.

Sequencing Operations
Timers are used for sequencing time based state transitions. Consider the traffic light controller where after each light operation, the controller has to wait for some defined amount of time. For example, timers can manage the transition between green, yellow and red. When a timer expires, the traffic light is moved to its next logical condition and a timer is started for the next transition.

Polling
In most systems, events are reported via messages. But sometimes it may not be practical to report the occurrence of an event via a message. So a timer is kept and the system polls for that condition on every timeout.
Consider the implementation of a congestion control system. The software runs a periodic congestion polling timer that polls for the CPU utilization and other congestion triggers. When the CPU utilizations reaches a threshold, the congestion control action is taken.

Periodic Operations
For implementing audits, periodic timers are kept. On each timer expiry, software audit is initiated. Another example where periodic timers are used could be implementing periodic billing of calls in telecom systems. When the call enters conversation, the periodic timer is started. On every timeout, the subscriber meter is incremented.

Failure Detection

For monitoring the health of other modules, a module runs a timer. It expects a sanity punch message periodically from all the other modules before the expiry of the timer. If certain number of sanity punches are missed in succession from a module, module failure is declared as failed.

Collecting Messages

Many times timers are used to collect messages. For example, in the digit collection procedure for a call, the system restarts a timer each time a digit message is received. The end of dialing is either received in the message or the timer expiry indicates it. No timer is started after the end of dialing is detected. Partial dialing is reported if the end of dialing is detected before the receipt of minimum number of digits required to route the call

Inactivity Detection

Timers are also used for detecting the inactivity in a session. Consider the case of a user session on the internet. Each time there is any kind of action by the user, the inactivity timer is restarted by the ISP. If the timer expires, the ISP terminates the user session and the internet connection is lost.

4.2 MINIX virtual timer interrupts

PIT interrupts are **real** interrupts. RT-MINIXv2 kernel emulates timer interrupts calling MINIX handler with a lower (or equal) rate than the PIT. These are virtual timer interrupts.

The timer interrupt rate of standard MINIX is defined as a constant in HZ as follow:

```
#define HZ           60    /* clock freq (software settable on IBM-PC) */
```

RT-MINIXv2 redefines HZ as follow:

```
#define HZ           50    /* clock freq (software settable on IBM-PC) */
```

The timer interrupt rate of standard RT-MINIXv2 is *RTM_tickrate* defined as a variable.
HZ and *RTM_tickrate* must be armonics frequencies to virtualize a timer interrupt for the MINIX kernel only when PIT interrupts occur, therefore they must related by an integer value:

*RTM_tickrate = RTM_timercoef * HZ;* where *RTM_timercoef = 1,2,3,.....N*

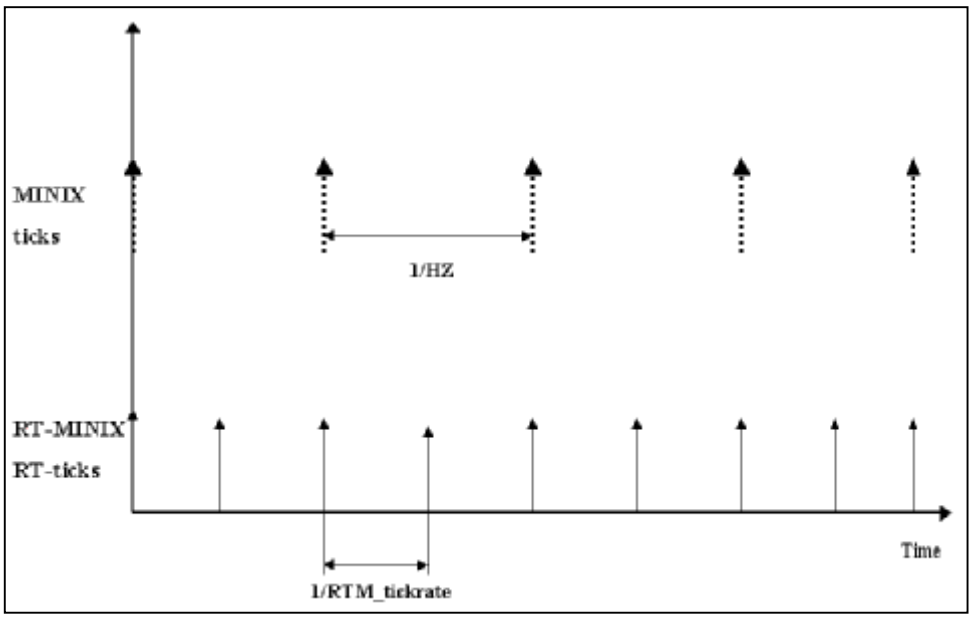

Figure 11

To preserve the illusion of the standard MINIX tick rate (HZ), the standard MINIX interrupt handler *clock_handler()* is called after *RTM_timercoef* timer interrupts. This lower interrupt rate is emulated using a virtual clock (explained later) with a period = *RTM_timercoef*. (See Figure 11)

To avoid that the MINIX timer interrupt handler could be executing with the interrupt priority of the RT-timer handler, a software IRQ is used to defer its processing with priority equals to (*RTM_PRILOWEST* – 1).

4.3 Timer Resolution

IBM-compatible PCs include a device called 8253/4 Programmable Interval Timer (PIT). This device issues a special interrupt on IRQ0 called timer interrupt, which notifies the kernel that one more time interval has elapsed.
The 8253/4 PIT uses an internal oscillator frequency at 1,193,180 Hz.
The PIT has a 16 bits LATCH register to set the ratio between the oscillator frequency and the the number of interrupts per second.

$$tick\_rate = 1193180/LATCH$$

Only some values of LATCH issues integer values of *tick_rate*. For other values of LATCH an accuracy error occurs.

LATCH	tick_rate
1	1193180
2	596590
4	298295
5	238636
10	119318
20	59659
59659	20

The PIT LATCH is a variable named *RTM_timercount* and it is initilized:

RTM_timercount = *TIMER_FREQ/RTM_tickrate* = 1193180/100 = 11931

with a reminder of 80 Hz

This reminder represents:

- an additional timer interrupt every (11931*100/80) = 15000 timer ticks
- a RT-timer interrupt frequency of 100.0067052217 [Hz]
- a RT-timer period of 0.009999329522788 [s]

resulting in an error in timer accuracy of 0.0000670478.

The next table shows some Real timer interrupt frequencies, timer periods and resulting errors.

Specified Tick Rate [int/s]	Specified Period [s]	Latch	Reminder [Hz]	Real Tick Rate [int/s]	Real Period [s]	Period Error
100	0.010000000	11931	80	100.006705	0.009999330	0.000067048
200	0.005000000	5965	180	200.030176	0.004999246	0.000150857
500	0.002000000	2386	180	500.075440	0.001999698	0.000150857
1000	0.001000000	1193	180	1000.15088	0.000999849	0.000150857
1500	0.000666667	795	680	1500.85535	0.000666287	0.000569906
2000	0.000500000	596	1180	2001.97987	0.000499506	0.000988954
3000	0.000333333	397	2180	3005.49118	0.000332724	0.001827050
4000	0.000250000	298	1180	4003.95973	0.000249753	0.000988954
5000	0.000200000	238	3180	5013.36134	0.000199467	0.002665147
7500	0.000133333	159	680	7504.27673	0.000133257	0.000569906
10000	0.000100000	119	3180	10026.7227	0.000099733	0.002665147

As RTM_tickrate is a multiple of HZ, it could be changed setting *RTM_timercoef* using *rtm_restart()* system call.
rtm_restart() can only be used before running any RT-process o when no one RT-process/handler is running, otherwise all it's system time reference would be erroneous.

MINIX has a global system variable *realtime* that stores the number of elapsed timer ticks since the system was started. It is set to 0 during kernel initialization
In RT-MINIXv2 the *RTM_counter.ticks* variable counts the number of elapsed timer ticks since the system was started. o it coud be reset by *rtm_restart()* system call.

RTM_counter.ticks is an unsigned 32 bits integer, and its returns to 0 time depends on the tick rate.
The next table shows some *return_to_zero* time of *RTM_tickcount* vs PIT interrupt frequency.

Tick rate [int/s]	Days for return_to_zero of *RTM_counter.ticks*
50	994
100	497
200	249
500	99
1000	50
1500	33
2000	25
3000	17
4000	12
5000	10
7500	7
10000	5

When *RTM_counter.ticks* returns to zero, anothe kernel variable named *RTM_counter.highticks* is increased.

4.4 8253/4 Programmable Interval Timer Programming

The 8253/4 Programmable Timer provides three independent 16-bit counters called timer channels that can count in binary or BCD. It can run in one of the six programmable modes:

- Mode 0 : Interrupt on Terminal Count
- Mode 1 : Programmable One-shot
- Mode 2 : Rate Generator
- Mode 3 : Square Wave Rate Generator
- Mode 4 : Software Triggered Strobe
- Mode 5 : Hardware Trigger Strobe

The programming of a timer channel is initiated by writing a control word into the control register port at 43H.

The control word has the following format:

D7	D6	D5	D4	D3	D2	D1	D0
SC1	SC0	RL1	RL0	M2	M1	M0	BCD

SC1	SC0	Comment
0	0	Select Counter 0
0	1	Select Counter 1
1	0	Select Counter 2
1	1	Illegal

RL1	RL0	Comment
0	0	Counter latching operation
0	1	Read/load MSB only
1	0	Read/load LSB only
1	1	Read/load LSB first, then MSB

M2	M1	M0	Mode
0	0	0	0
0	0	1	1
x	1	0	2
x	1	1	3
1	0	0	4
1	0	1	5

The BCD bit specifies whether to count in binary or BCD.
Since the counters are 16 bits long but the I/O port through which data is transferred is only 8 bits long, two data transfer operations will be needed if we want to fill the whole counter.
Setting the bits D5 and D4 to 11 in order to load first the LSB then the MSB as the preset 16-bit word count for the corresponding counter.

Here are some relevant I/O port addresses:

40H Timer Channel 0 Counter
41H Timer Channel 1 Counter
42H Timer Channel 2 Counter
43H Timer Control Register

Two kernel functions operates on the PIT Timer Channel 0.

- *RTM_set_timer()*: Used to change the timer interrupt rate. The function parameter is the latch counter.
- *RTM_read_timer()*: Used to read the current value of the latch counter.

4.5 Latency Measurement

One of the most common measurements requested of real-time kernel and realtime operating system is the interrupt latency [35]. This metric, while useful, is a very limited indication of the performance capabilities of a RT-kernel.
Interrupt latency numbers are most useful when used to measure the effectiveness of an RT-kernel at dealing with extremely high-priority interrupts or emergency interrupts.

Understanding the relative size of delays is important to the design of the real-time system. Most sources of delay in an RT-kernel are due to either code execution or context switches. Virtually all of these delays are fixed in length and repeatable. Bounded and repeatable is the fundamental characteristic desired of an RT-kernel.

Interrupt latencies are not fixed in length. Because an interrupt is, by definition, an asynchronous event, a system's interrupt latency is dependent on the state of the machine when the interrupt occurred. This state is a function of both the hardware and the software used in the system.

Interrupt latency is measured as the time that elapses between the instant when a hardware interrupt request is made and when that request is honored by the CPU and software. In other words, it is the delay encountered before execution of the interrupt service routine begins, following assertion of the CPU's interrupt input.

RT-MINIXv2 kernel uses the PIT timer counter 0 for timer interrupt latency measurement.
When the latch counter reachs zero, the PIT triggers an interrupt, and then the counter is reset to it's initial value *RTM_timercount*. As the PIT remains decrementing the counter, during the timer handler is servicing the interrupt, the value of the counter lets compute the latency of the handler.

Timer_Handler_Latency = RTM_timercount - RTM_read_timer()

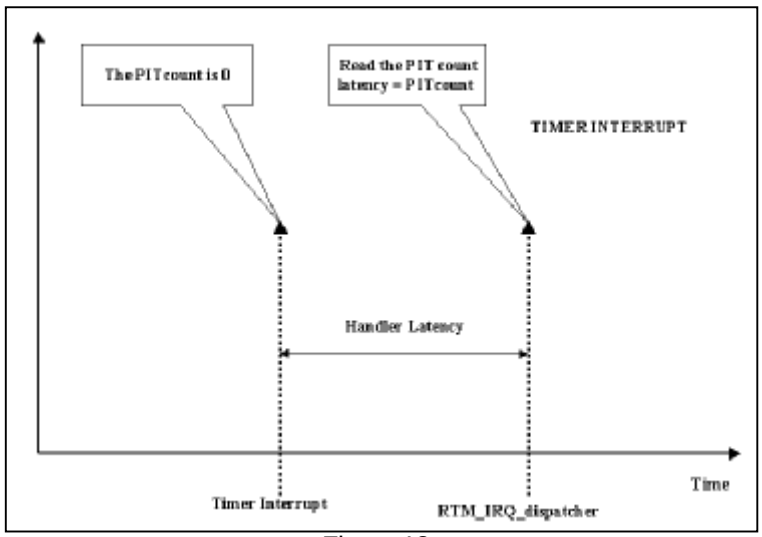

Figure 12

To consider the overhead of executing *RTM_read_timer()*, the kernel computes it in initialization time and store the number of timer Hz elapsed in the global variable *RTM_PIT_latency*.

When *RTM_flush_int()* is invoked to flush interrupts, it stores the current PIT latch counter in one local variable named *before*, and the current value of *RTM_counter.ticks* in another local variable named *befRT-ticks* before running the handler.
After the handler was run, the interrupt processing time (in timer Hz) can be computed as:

$$IRQ\_proc\_time\ [Hz] = before + [RTM\_timercount - RTM\_read\_timer()\ ] + [(RTM\_counter.ticks - befRT\text{-}ticks - 1\ ) * RTM\_timercount]$$

Note: This interrupt processing time includes interrupt and process interferences

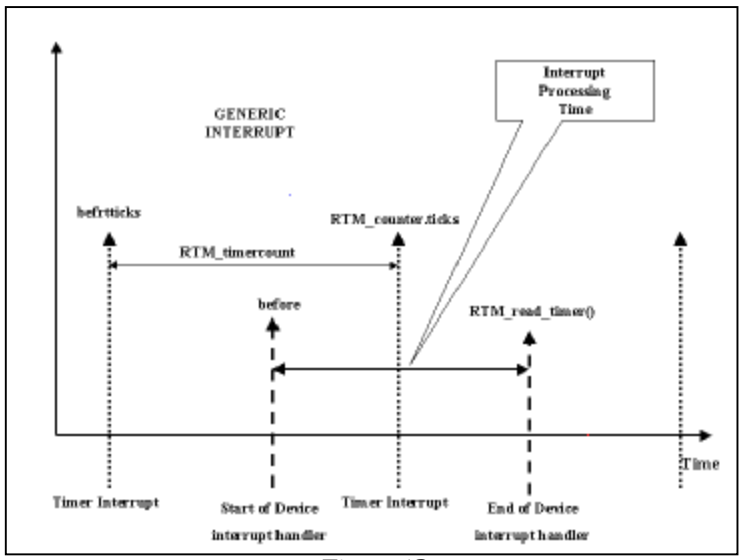

Figure 13

To compute the Interrupt Processing time in seconts

IRQ_proc_time [s] = IRQ_proc_time [Hz]/ TIMER_FREQ = IRQ_proc_time [Hz]/ 1193180[Hz/s]

SE PUEDE AGREGAR UN CAMPO MAS A LOS DESCRIPTORES DE INTERRUPCION HRtstamp (High Res Timestamp)

QUE GUARDEN EL TIMESTAMP DEL LATCH DEL PIT ADEMAS DEL CONTADOR DE TICKS.

AL FINALIZAR LA INTERRUPCION, LA LATENCIA SERÁ:

LATENCY = (RTM_timercount – RTM_read_timer())

+ RTM_timercount (RTM_counter.ticks – irqd->HRtstamp)

4.6 Starting Real-Time processing

At startup, the system is not ready for Real-Time processing. All interrupt handlers pointed by the *RTM_irq_dispatch()* are standard MINIX handlers, and all processes types are *RTM_P_STANDARD*. To start Real-Time processing the *rtm_RTstart()* system call must be used.

Sometimes, it is necesary to reconfigure the system to change some parameters without system recompilation and restarting. *rtm_restart()* could be used to change Real-Time processing parameters and reset system statistics. One of this parameters may be the timer interrupt frequency changing the kernel variable *RTM_timercoef*. But if the timer period is changed, all previous timing related statistics will be erroneous.
The kernel variables that are affected by the timer interrupt frequency change are:

- *RTM_counter.ticks*: counts the number of elapsed timer ticks since the system startup or since the last call to *RTM_restart()*. If the timer tick rate is changed, the value in this variable will be without of sense.
- *RTM_IRQ_latency*: it uses *RTM_counter.ticks* to obtain the *RTM_IRQ_dispatch()* latency.
- *RTM_idlemax*: it is the maximun count of idle loops in a RT-tick. As the period of the RT-tick is changed the value in this variable will be without of sense
- *RTM_desc_table[irq].timestamp*: for each interrupt descriptor, it is the timestamp in RT-ticks of the last interrupt. If the timer tick rate is changed, the value in this variable will be without of sense.
- *RTM_desc_table[irq].latency*: for each interrupt descriptor, it is the last interrupt latency in PIT Hz, but as it uses *RTM_counter.ticks* to obtain the value.

To return to Non Real-Time processing a user process could use ***rtm_RTstop()*** system call that can only be used when any RT-process is running.
Using ***rtm_RTstop()*** all interrupt will be serviced by Non Real-Time handlers.

4.7 Virtual Clocks

Some systems use software timers to handle periodic processing. Once the period of the timer has elapsed, the periodic process is scheduled, the timer is removed from a queue and inserted in other position of the queue depending on the period. These approach produce significant overhead to the periodic process and particulary in those that have small periods.

RT-MINIXv2 uses virtual clocks to manage periodic processing. It have *NR_VCLOCKS* virtual clocks descriptors RTM_vclock_desc_t in kernel space.

4.7.1. Virtual Clocks Management

The kernel have a wheel of active virtual clocks that are checked on each timer interrupt. (See Figure 14).

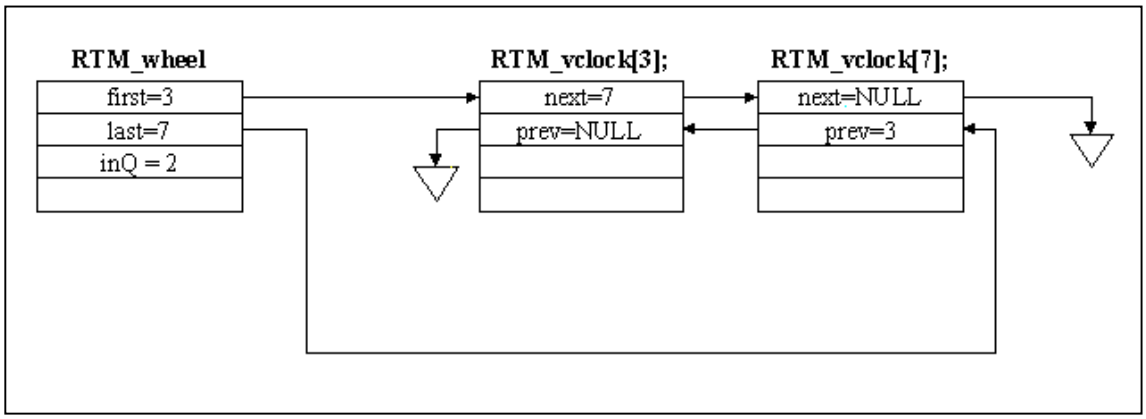

Figure 14

To reduce latency delay during the interrupt handling, some tasks are delayed to be processed by *RTM_flush_int()* out of interrupt time.
When *RTM_IRQ_dispatch()* is called by a timer interrupt (CLOCK_IRQ) , it enqueue the RT-timer interrupt handler to be processed by *RTM_flush_int()*.
As the timer IRQ descriptor have the highest priority (RTM_PRIHIGHEST), it will be the first handler called by *RTM_flush_int()*.

To check for virtual clocks that need to be triggered, the timer RT-handler does the following:

```
RTM_vclock_desc_t *pvc;                    /* virtual clock pointer */
for( i = 0, pvc = RTM_wheel.first; i < RTM_wheel.inQ; i++, pvc = pvc->next)
        if( ((RTM_counter.ticks - pvc->offset) % pvc->period)  == 0 )
                RTM_vclock_trigger(pvc->index);
```

pvc->offset: the timestamp of *RTM_counter.ticks* when the virtual clock was activated.

As the overhead of virtual clock checking is high because it is done on every timer tick, it is important to set the timer resolution to the lowest virtual clock period needed.

The available virtual clock actions are:

- *RTM_ACT_NONE:* do nothing. Used by software timers to keep the virtual clock data structure assigned to a process and active, therefore avoiding the abuse of create and delete operations.
- *RTM_ACT_SCHED:* clears the *RTM_P_SLEEP* flag of the process status flags, and if the process is ready to run, it is inserted in it's priority queue.
- *RTM_ACT_MESS:* send a message to the owner process mailbox

- *RTM_ACT_EXPIRED:* do nothing. Once a virtual clock has reached the limit of expirations, the kernel change the action type to *RTM_ACT_EXPIRED* and all virtual clock statistics and counts are freezed.
- *RTM_ACT_IPC: a RT-IPC function has reach its timeout. Se debe invalidad el estado de send o receive del proceso rapidamente de tal modo de que el proceso receptor o emisor no pueda recibir o enviar un mensaje una vez que el proceso vencio su timeout. Luego, se utiliza una software* IRQ *previamente asignada para que remueva al proceso de la cola de espera correspondiente.*
- *RTM_ACT_SOFTIRQ:* activates the software IRQ specified in the virtual clock *param*. The software IRQ must be previously assigned using *RTM_set_softirq()* kernel function.

The kernel functions that operate on virtual clocks are:

- *RTM_vclock_create()*: create a new virtual clock..
- *RTM_vclock_delete():* removes the virtual clock from the wheel and free the resource.
- *RTM_vclock_change():* change the parameters of an active virtual clock.

4.7.2. Periodic Process Scheduling

RT-MINIXv2 use virtual clocks to schedule periodic RT-USER processes.
A virtual clock is assigned to each periodic process. The virtual clock action type is *RTM_ACT_SCHED*.
If a periodic process is scheduled by the virtual clock in a RT-READY state implies that it has missed its deadline, therefore the kernel sends a MT_WATCHDOG message to a watchdog process specified in the process descriptor.

4.7.3. Alarms

Alarms are a special kind of a virtual clocks. The have the following behaviour.

- They are one shot.
- They have no period.
- They have a deadline.

To create an alarm using a virtual clock:

- vc.period = alarm.duration;
- vc.limit = 1;
- vc.action = alarm.action;
- vc.param = alarm.param;

Once the alarm was triggered, the virtual clock could be remove with *RTM_vclock_delete()*.

4.7.4. Timeouts Management

Timeouts could be created using virtual clocks. They have the following behaviour.

- They are one shot.
- They have no period.
- It is often that it is not reach their deadlines.

To create a timeout using a virtual clock:

- vc.period = timeout.duration;
- vc.limit = 1;
- vc.action = timeout.action;
- vc.param = timeout.param;

If the timeout has no reached it deadline, the virtual clock could be removed using *RTM_vclock_delete()* or its action could be set *RTM_ACT_NONE* using *RTM_vclock_change()* to keep the virtual clock assigned.
If the timeout has reached it deadline, the virtual clock could be remove with *RTM_vclock_delete()*.

4.7.5. Send/Receive Timeouts

RT-IPC primitives can include timeouts that return *E_TIMEOUT* error code to the calling process when a message can not be sent/received in a specified time period.

RT-MINIXv2 kernel uses virtual clocks to implement RT-IPC timeouts.
More about send/receive timeouts is detailed in the RT-IPC chapter.

4.8 Deadline Handler using Software IRQs

A deadline handler permits to take corrective actions when a process deadline have been missed[27].
Another use of virtual clocks is the implementation of deadline handler using software IRQs.
The process must:

A. sets a software IRQ handler to be run on each missed deadline
B. sets a one-shot virtual clock with period equal to its deadline with the *action* = RTM_ACT_SOFTIRQ and the *param* = software IRQ number.
C. runs its specific code.
D. removes the virtual clock and the software IRQ before exit..

If during the process execution the deadline is missed, the software IRQ handler is called and corrective actions can be taken.

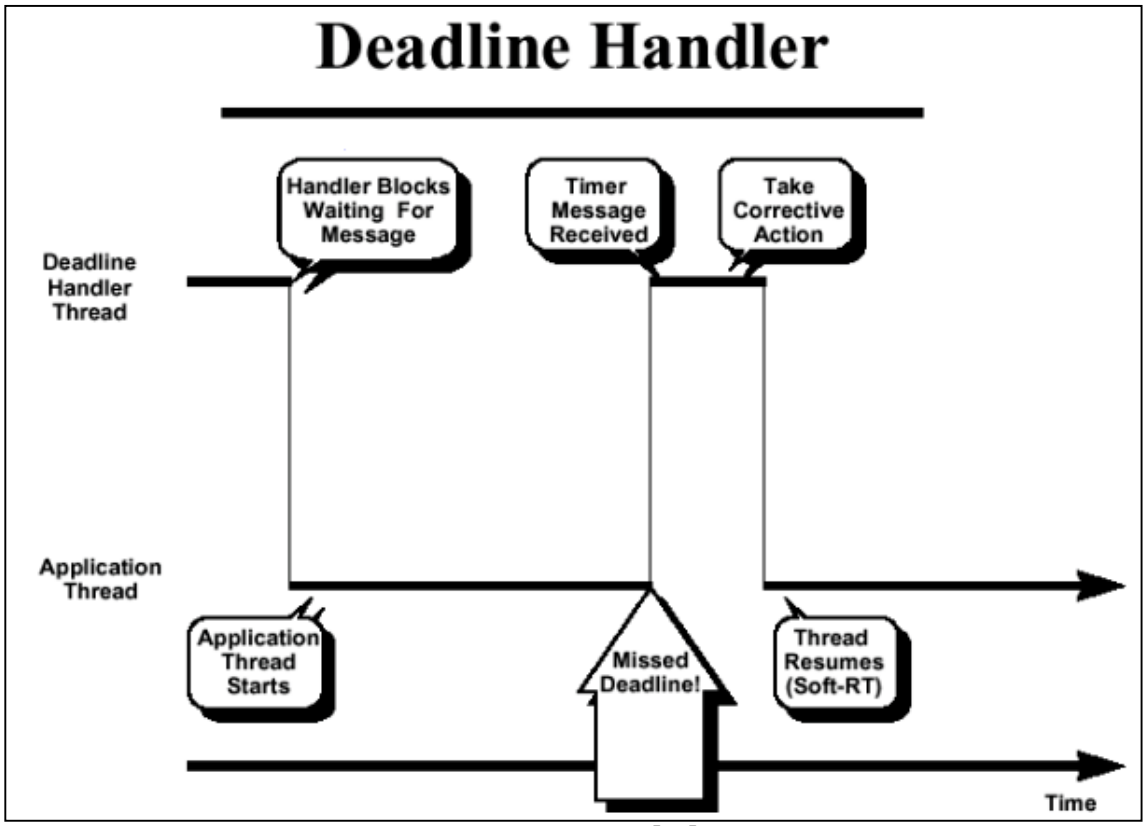

Figure 15 – From [27].