View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Servicio de Difusión de la Creación Intelectual

MINIX4RT: A Real-Time Operating System Based on MINIX

Pablo Andrés Pessolani

ABSTRACT

Tanenbaum's MINIX Operating System was extended with a Real-Time microkernel and services to conform MINIX4RT, a Real-Time Operating System for academic uses that includes more flexible Interprocess Communications facilities supporting basic priority inheritance protocol, a fixed priority scheduler, timer and event driven interrupt management, statistics and Real-Time metrics gathering keeping backward compatibility with standard MINIX versions.

Keywords: Operating Systems, Minix, Interrupt Handling, Real-Time.

Presented to the Universidad Nacional de La Plata of the Requirements for the Degree of Master on Computer Networks.

February 2006

Thesis Director: Ph.D. Silvio Gonnet

Thesis Co-Director: Eng. Armando De Giusti

This thesis is dedicated to my lovely daughters Flor and Vicky, to the "love of my life" Ana, and to my "iron willpower" parents Merce and Pety.

Acknowledgments

I would like to express my sicere gratitude to Dr. Silvio Gonnet for his suggestions, dedication and vast patience as thesis director.

I would like to thanks Nicolás Cesar and Felipe Ieder for their contributions with the system latency and timeliness tests and German Maglione for supplying the IBM Thinkpad 370 notebook used in the implementation and tests of MINIX4RT.

I acknowledge help received from Telecom Argentina S.A. for sponsoring my UNLP Master career, the Department of Information Systems of UTN-FRSF for sponsoring the presentation of articles about MINIX4RT in the Argentine Symposium of Technology 2004/5 and the Postgraduate Office of UNLP for its attention and patience.

This acknowledgment would be incomplete if I forgot to mention my whole family for their love, support, and encouragement throughout this course of study.

PREFACE

MINIX4RT is the result of fifteen years dedicated to study, to research and to teach about Operating Systems. It is a new open-source Real-Time Operating System intended as teaching tool, but it can be usable as a serious system on resource-limited computers.

MINIX4RT is a Real-Time branch of the popular MINIX used in grade level Operating Systems courses. This work discuss the modifications that have been made to MINIX that give it the ability to support the stringent timing requirements of Real-Time applications, while still giving Non Real-Time ones access to the full range of MINIX services without any changes.

MINIX4RT does not stops with this thesis. It is the authors intention to conform a team of developers around the world of people interesting in cooperate with the growth and enhancement of MINIX4RT.

CONTENTS

1. INTRODUCTION **12**

1.1. CONTRIBUTIONS 13

1.2. MOTIVATION 14

1.3. OTHER RTOS USED IN EDUCATION 15

1.4. MINIX TIME SHARING FEATURES 16

1.5. RT-MINIX FEATURES 17

1.6. MINIX4RT FEATURES 18

1.7. ORGANIZATION 22

1.8. TERMINOLOGY AND NOTATION 22

2. ARCHITECTURE AND INTERRUPT HANDLING **24**

2.1. MINIX SYSTEM ARCHITECTURE AND INTERRUPT PROCESSING 24

2.2. MINIX4RT SYSTEM ARCHITECTURE 26

2.3. INTERRUPT HANDLING 28

2.3.1. Interrupt Handling Virtualization *29*

2.3.2. Hardware Interrupts Emulation *30*

2.3.3. User and Kernel Stacks *31*

2.3.4. Interrupt Handler Types *32*

2.3.5. Interrupt Service Routines *33*

2.3.6. Interrupt Descriptor Data Structure *33*

2.3.7. Interrupt Handler Dispatching *35*

2.3.8. Interrupt Handler's Priority *38*

2.3.9. Real-Time Input/Output Tasks *40*

2.3.10. Software Interrupts *40*

2.3.11. Returning from System Calls and Interrupts Service Routines *41*

2.3.12. Flushing Deferred Interrupts *42*

2.3.13. Executing Interrupt Handlers *44*

2.3.14. Interrupt Descriptor Timestamp Field *46*

2.3.15. Kernel Functions for Interrupt Handling and Synchronization *46*

2.3.16. Estimating Interrupt Handler Processing Time *48*

2.3.17. Nested Interrupts *50*

2.3.18. Real-Time Interrupt Processing *51*

2.3.19. Standard MINIX Non Real-Time interrupts *52*

2.3.20. Real-Time Timer-Driven Interrupts *52*

2.3.21. Real-Time Event-Driven Interrupts *54*

2.4. PREVENTING INTERRUPT PRIORITY INVERSION 54

2.5. RT-PROCESS DISPATCH LATENCY ... 55

3. RT-PROCESS MANAGEMENT AND SCHEDULING ... 57

3.1. MINIX4RT EXECUTION MODES ... 58

3.2. REAL-TIME PROCESS CREATION ... 58

3.3. RT-PROCESS STATES AND TRANSITIONS ... 60

3.4. PROCESS DESCRIPTOR REAL-TIME FIELDS ... 62

3.5. THE RT-PROCESS SCHEDULER ... 64

3.6. PROCESS PRIORITY ... 65

3.6.1. NRT-Process Priorities ... 66

3.6.2. RT-Process Priorities ... 66

3.7. RT-READY QUEUES MANAGEMENT ... 67

3.8. RT-PROCESS TERMINATION ... 70

3.8.1. RT-process Termination Using the exit() System Call ... 70

3.8.2. RT-process Termination Using the signal() System Call ... 71

3.8.3. Releasing RT-process Resourses and Housecleaning ... 71

4. TIME MANAGEMENT ... 72

4.1. TIMING MECHANISMS ... 72

4.2. MINIX4RT TIMER INTERRUPTS ... 73

4.3. MINIX VIRTUAL TIMER INTERRUPTS ... 75

4.4. TIMER RESOLUTION ... 76

4.5. 8253/4 PROGRAMMABLE INTERVAL TIMER PROGRAMMING ... 78

4.6. ESTIMATING THE TIMER INTERRUPT LATENCY ... 80

4.7. REAL-TIME AND NON REAL-TIME TIMER HANDLERS ... 81

4.8. VIRTUAL TIMERS ... 81

4.8.1. Virtual Timers Handling Functions ... 82

4.8.2. Virtual Timers Queues ... 84

4.8.3. Executing Virtual Timers Actions ... 85

4.9. VIRTUAL TIMERS HANDLING: OTHER TESTED APPROACHES ... 87

5. REAL-TIME INTERPROCESS COMUNICATION (RT-IPC) ... 89

5.1. INTRODUCTION ... 89

5.2. MINIX IPC PRIMITIVES ... 90

5.3. MINIX4RT IPC PRIMITIVES FEATURES ... 91

5.4. MESSAGE DESCRIPTOR DATA STRUCTURE ... 93

5.4.1. Message Payload Data Structure ... 93

5.4.2. Message Header Data Structure ... 94

5.5. THE MESSAGE QUEUE ENTRY DESCRIPTOR 94
5.6. THE MESSAGE QUEUE DESCRIPTOR 95
5.7. THE RT-SYSTEM MESSAGE POOL 95
5.8. MESSAGE QUEUES MANAGEMENT 96
5.9. THE *MRT_RQST()* KERNEL CALL 98
5.10. THE *MRT_ARQST()* KERNEL CALL 99
5.11. THE *MRT_REPLY()* KERNEL CALL 99
5.12. THE *MRT_UPRQST()* KERNEL CALL 100
5.13. THE *MRT_SIGN()* KERNEL CALL 100
5.14. THE *MRT_SEND()* KERNEL FUNCTION 101
5.15. THE *MRT_RCV()* KERNEL CALL 102
5.16. THE *MRT_RQRCV()* KERNEL CALL 103
5.17. USING MINIX4RT IPC KERNEL CALLS 103
5.18. PRIORITY INVERTION 104
5.19. BASIC PRIORITY INHERITANCE PROTOCOL (BPIP) 106
5.20. THE PRIORITY CEILING PROTOCOL (PCP) 109
5.21. COMPLETE PRIORITY INHERITANCE 110

6. RT-SYSTEM CALLS, KERNEL CALLS AND FUNCTIONS 111

6.1. MINIX SYSTEM CALLS IMPLEMENTATION 111
6.2. MINIX4RT SYSTEM CALLS IMPLEMENTATION 112
6.2.1. RT-System Calls with Message Transfers 112
6.2.2. RT-Kernel Calls without Message Transfers 113
6.2.3. The RT-PID 113
6.3. ADDING NEW RT-SYSTEM CALLS USING MRTTASK 114
6.4. ADDING RT-KERNEL CALLS WITHOUT MESSAGE TRANSFERS 116
6.5. RT-KERNEL FUNCTIONS 117

7. REAL-TIME PROCESSING RELATED STATISTICS 120

7.1. SYSTEM-WIDE STATISTICS 120
7.2. INTERRUPTS SERVICE ROUTINES STATISTICS 121
7.3. PROCESS STATISTICS 121
7.4. THE IDLE PROCESS 122
7.5. THE FX KEYS 123
7.5.1. The Shift-F1 Hot-Key 123
7.5.2. The Ctrl-F1 Key 124
7.5.3. The Shift-F2 Hot-Key 125

7.5.4. The F4 Key *126*
7.5.5. The Shift-F4 Hot-Key *127*
7.5.6. The Ctrl-F4 Hot-Key *128*
7.5.7. The Shift-F5 Hot-Key *129*
7.5.8. The F6 Key *130*
7.5.9. The Shift-F6 Hot-Key *131*
7.5.10. The Ctrl-F6 Hot-Key *132*
7.5.11. The F8 Key *133*
7.5.12. The Shift-F8 Hot-Key *134*
7.6. THE MODIFIED *PS* COMMAND 136
7.6.1. The –A Option *136*
7.6.2. The –S Option *136*
7.7. THE *MRTSTATUS* COMMAND 137
7.7.1. The –s Option *138*
7.7.2. The –i Option *139*
7.7.3. The –t Option *139*
7.7.4. The –m Option *140*
7.7.5. The –c Option *141*
7.7.6. The –I Option *142*
7.7.7. The –T Option *143*
7.7.8. The –M Option *144*

8. CONCLUSIONS AND FUTURE WORKS 146

8.1. CONCLUSIONS 146
8.2. FUTURE WORKS 146

REFERENCES 148

APPENDIX A: RT-SYSTEM CALLS AND RT-KERNEL CALLS REFERENCE .. 151

A.1. SYSTEM CALLS REFERENCE 151
A.1.1. mrt_RTstart *151*
A.1.2. mrt_RTstop *152*
A.1.3. mrt_clrpstat *152*
A.1.4. mrt_getiattr *153*
A.1.5. mrt_getiint *154*
A.1.6. mrt_getistat *154*
A.1.7. mrt_getpattr *155*
A.1.8. mrt_getpint *156*

A.1.9. mrt_getpstat *156*
A.1.10. mrt_getsstat *157*
A.1.11. mrt_getsval *157*
A.1.12. mrt_restart *158*
A.1.13. mrt_setiattr *158*
A.1.14. mrt_setpattr *159*
A.2. KERNEL CALLS REFERENCE 161
A.2.1. mrt_rqst *161*
A.2.2. mrt_arqst *161*
A.2.3. mrt_uprqst *162*
A.2.4. mrt_sign *163*
A.2.5. mrt_reply *164*
A.2.6. mrt_rcv *164*
A.2.7. mrt_sleep *165*
A.2.8. mrt_wakeup *166*
A.2.9. prt_print *166*

APPENDIX B: SAMPLE PROGRAMS 168

B.1. MRTSTART.C 168
B.2. MRTSTOP.C 169
B.3. MRTTEST1.C 170
B.4. MRTTEST1B.C 171
B.5. MRTTEST1C.C 172
B.6. MRTTEST2.C 173
B.7. MRTTEST3.C 173
B.8. MRTTEST3B.C 174
B.9. MRTTEST3C.C 175
B.10. MRTTEST4.C 176
B.11. MRTTEST4B.C 177
B.12. MRTTEST5.C 178
B.13. MRTTEST5B.C 179
B.14. MRTTEST5C.C 179
B.15. MRTTEST6.C 180
B.16. MRTTEST6B.C 182
B.17. MRTTEST6C.C 183
B.18. MRTTEST6D.C 184
B.19. MRTTEST6E.C 188

B.20. MRTTEST7.C ... 191
B.21. MRTTEST7B.C ... 192
B.22. MRTTEST8.C ... 194
B.23. MRTTEST8B.C ... 197
B.24. MRTTEST8C.C ... 200
B.25. MRTTEST8D.C ... 203
B.26. MRTTEST8E.C ... 207
B.27. MRTTEST9.C ... 210

APPENDIX C: PERFORMANCE TESTS ... 215

C.1. INTERRUPT SERVICE TIME ... 215
C.1.1. Delay of RTLinux ... 216
C.1.2. Interrupt Service Time Tests of MINIX4RT ... 217
C.1.3. Interrupt Service Time of Software Interrupts ... 218
C.1.4. Interrupt Service Time of Event-Driven Interrupts ... 220
C.1.5. Interrupt Service Time of Timer-Driven Interrupts ... 222
C.2. VIRTUAL TIMER TIMELINESS ... 223
C.3. RT-IPC PERFORMANCE ... 226
C.3.1. Synchronous Message Transfers Tests without BPIP and without Timeouts ... 226
C.3.2. Asynchronous Message Transfers Tests without BPIP and without Timeouts ... 227
C.3.3. Synchronous Message Transfers Tests with BPIP and without Timeouts ... 228
C.3.4. Asynchronous Message Transfers Tests with BBIP and without Timeouts ... 228
C.3.5. Request/Receive Tests with BPIP and without Timeouts ... 228
C.3.6. Synchronous Message Transfers Tests with BPIP and with Timeouts ... 229
C.3.7. Asynchronous Message Transfers Tests with BBIP and with Timouts ... 230
C.3.8. Test Request/Receive Tests with BPIP and with Timeouts ... 231
C.3.9. RT-IPC Tests Results ... 232

APPENDIX D: SYSTEM DATA STRUCTURES ... 233

D.1. USER-LEVEL DATA STRUCTURES ... 233
D.1.1. System-wide Data Structures ... 233
D.1.2. Interrupt-related Data Structures ... 233
D.1.3. Process-related Data Structures ... 234
D.1.4. Kernel Calls-related Data Structures ... 235
D.1.5. Message-related Data Structures ... 235
D.2. KERNEL-LEVEL DATA STRUCTURES ... 236
D.2.1. System-wide Data Structures ... 236

D.2.2. Interrupt-related Data Structures *237*
D.2.3. Process-related Data Structures *238*
D.2.4. Message-related Data Structures *239*

GLOSARY 241

1. INTRODUCTION

A Real-Time Operating System (RTOS) supports Real-Time applications. Real-Time applications requirements from the Operating System (OS) are much different from those required by non-time constrained time-sharing applications. MINIX4RT satifies the constrains of many of the applications that impose stringent timing demands on their OS with disastrosous consequences resulting from temporal errors.

To assist Real-Time applications designers, RTOS must facilitate efficient interprocess communication and synchronization, a fast interrupt response time, asynchronous Input and Output (I/O) and timing related facilities.

Real-Time applications written to run on RTOS make use of and rely on the following system capabilities [1]:

- A preemptive kernel.
- Fixed-priority scheduling policies.
- Real-Time clocks and timers.
- Asynchronous I/O.
- Queued Real-Time signals.
- Process communication facilities.

Computer science students and professionals working on RTOS need a deep knowledge about every software component and the interactions with hardware devices considering timing constraints.

RTOS instructors can choose among commercial or free licence software to develop their laboratory practice. Commercially available RTOSs are too costly and proprietary to be used by academic institutions. Free licence and open source RTOSs have been designed with emphasis on predictability as a key design feature with complex source code readability.

The thesis of this work is an academic purpose RTOS designed to present an approach of adding Real-Time facilities to a standard time-sharing OS. The proposed architecture defines a way to schedule, manage, and execute Real-Time processes. The resulting system will be able to guarantee that Real-Time processes will meet their stated deadlines.

The thesis has been proved by building MINIX4RT, a research prototype implementation of that architecture based on MINIX [2], and then testing the implementation by executing periodic and non-periodic Real-Time processes using the added Real-Time facilities.

1.1. Contributions

This work makes several contributions to the existing body of knowledge of academic purpose Operating Systems.

- *Architecture*: Adds a Real-Time sub-kernel below a general-purpose OS.
- *Interrupt Management*: A detailed description of how interrupts are managed to support Real-Time processing with minimal priority inversion.
- *Process Management*: It presents a new viewpoint of two domains of process states, transitions and scheduling.
- *Time Management*: An original approach is proposed to deal with the execution of actions triggered by time that minimize the priority inversion problem.
- *System Calls*: A detailed description of how System Calls and Kernel Calls are implemented and how new ones can be added.
- *Statistics*: The prototype has several methods to collect system states, parameters and statistics that are useful for teaching, debugging and system verification.

1.2. Motivation

The goal of the MINIX4RT project is to provide an educational tool for RTOS courses as MINIX [3, 4, 5] and Linux [6] do for OS Design and Implementation courses.

The decision of adopting MINIX as foundation for this work is based on:

– *Documentation Availability*: Tanenbaum and Woodhull book [2] is the main MINIX reference, but plentyful documentation can be found on the Internet as on http://www.minix3.org, http://minix1.hampshire.edu/, etc.

– *Hardware Platform*: A PC x386 with 16 Mb of RAM and a hard disk of 100 Mb is enough to run MINIX. These modest hardware requirements allow old PCs to be recycled for Real-Time laboratories. Moreover, MINIX can run in emulated environments as BOSCH, VMWare, MS Virtual PC, QEMU etc. (more information about emulated environments in http://minix1.hampshire.edu/hints.html#emul-virt). This feature allows sharing laboratories among courses without additional manteinance and operational impact.

– *Modular and Elegant Design*: MINIX is a small UNIX-like operating system, originally developed by Andrew Tanenbaum as a teaching tool for operating systems classes. MINIX was designed with a more modular internal structure than the monolithic UNIX kernel, and this structure affects the way in which new features could be added to MINIX.

– *Existing Applications and Programming Tools*: The same project goal could be reached writing a RTOS from scratch but this strategy implies the construction of a new user interface to run applications like text editors, compilers, linkers, etc. that does not need Real-Time services. Using MINIX as the interface between the Real-Time Kernel and User-space applications simplifies the development of the system and allows the use of well-known tools.

– *Academic Experience*: The author teaches about Operating System in *Facultad Regional Santa Fe* of the *Universidad Tecnológica Nacional* (Argentine), where MINIX is used as an academic tool since 1993.

MINIX4RT implementation focus on source code readability (perhaps sacrificing performance) to allow instructors to easily do a multiplicity of grade courses assignments, laboratory tests and other academic uses with an open source RTOS. Some interesting projects could be:

– Adding new functionalities and System Calls to MINIX4RT.

– Coding and Testing Real-Time scheduling algorithms.

– Coding Servers to handle Real-Time Aperiodic tasks.

– Porting hard Real-Time network protocol stacks as RTNET [7] or RETHER [8].

– Building Remote Device Drivers to control Robots.

– Build embedded RTOS based on MINIX4RT.

– Add new features and System calls to MINIX4RT to be compliant with the IEEE POSIX 4 standard.

Plentyful statistics are gathered to make the RTOS more educational about its operation and helpful for debbuging applications.

1.3. Other RTOS Used in Education

Most RTOS used in education like RTLinux [9], RT-Mach [10], QNX [11], and RT-MINIX[12, 13] are suitable for Real-Time Systems courses but this fact must not be confused with teaching how a RTOS works. Those systems are focused on performance, schedulability, research, commecial market, etc. but not for academic purposes. Their source code readability (if they are available), complex algorithms and limited documentation do not help for students understanding.

Until the development of MINIX4RT, RTLinux was the system used in laboratory practice and assignments of the Advanced Operating System course in *Facultad Regional Santa Fe* of the *Universidad Tecnológica Nacional* (Argentine). Even RTLinux is used in education, its kernel source code is not well documented and requires a deep knowledge of the constant growing Linux kernel. As basic courses of Operating Systems in *Facultad Regional Santa Fe* are based on MINIX, MINIX4RT appears as the natural choice for practice in a RTOS course.

The major algorithms and data structures used by MINIX4RT were created or adapted trying to achieve a balance between efficiency and simplicity inspired in popular OS as:

- *RTLinux [9]*: The Virtual Machine (VM) concept limited to interrupt emulation.
- RT-MACH [*10*]: RT-Inter Process Communications.
- QNX [*11*]: Priority queues.
- Windows NT[*14*]: Interrupt queues.
- Linux [*15*]: Virtual timers.
- MACH [*16*]: Message queues.

1.4. MINIX Time Sharing Features

MINIX [2] is a complete, time-sharing, multitasking OS developed from scratch by Andrew S. Tanenbaum. It is a general-purpose OS broadly used in Computer Science degree courses.

Though it is copyrighted, the source has become widely available for universities for studying and research. 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*: Allows for modular design and clear implementation of new features.
- *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.
- *Interrupt hiding*: Interrupts are converted into message transfers.

1.5. RT-MINIX Features

Wainer and Rogina [12, 13] changed MINIX OS to support RT-processing and named it "RT-MINIX". Its main features are:

- *Scheduling Algorithms Selection*: Rate Monotonic (RM) and Earliest Deadline First (EDF) scheduling were included. These strategies were later combined with other traditional strategies, such as Least Laxity First, Least Slack First and Deadline Monotonic.

- *Joined Scheduling Queues*: Process execution priority was implemented using a multiqueue scheme to accommodate Real-Time processes along with interactive and CPU-bound tasks.

- *Real-Time Metrics collection:* Several variables about the whole operation are accessible to the user to provide data for benchmarking and testing new developments.

- *Timer Resolution Management:* The resolution of the Timer can be changed to get better accuracy while scheduling processes.

Several data structures in the OS were modified to consider processes period, execution time and criticality. But RT-MINIX does not have its own architecture, it is like a patch for MINIX in order to provide the user with a set of System Calls to create and manage periodic or aperiodic processes. That approach implies some academic and functional limitations because:

- *It does not have its own architecture*: The source code of kernel functions and fields of data structures that treat with Real-Time issues are merged with those that treats with non Real-Time ones.

- *It does not serve hardware interrupts in priority order*: This fact could produce unbounded priority inversion. While a higher priority interrupt handler is running, lower priority interrupts could be attended increasing the interference.

- *It has only one level of priority for all Real-Time processes*: This fact reduces the system schedulability. Even worst, MINIX Tasks and Servers have higher priorities than Real-Time processes. While a RT-process is

running, a standard MINIX Task or Server could preempt it. This is another case of unbounded priority inversion.

– *It uses standard MINIX message transfers as its IPC primitives*: MINIX use FIFO discipline to receive messages from several processes and this implies that a priority inversion problem is present.

– *It does not have any protocol against unbounded priority inversion:* Its utilization in projects that use cooperating processes is limited.

– *Increasing the Timer resolution also increases the system overhead:* To increase the Timer resolution, RT-MINIX increases the Timer frequency. This strategy executes MINIX Timer interrupt handler at higher frequency, thus increasing the system overhead.

– *A Real-Time process can use MINIX System Calls:* When MINIX Server receives a request from a Real-Time process it will be executed on behalf of it, but without any Real-Time attributes.

– *MINIX Timer handler was modified to support RT-alarms*: As MINIX scheduler treats Tasks in FIFO order, other MINIX Tasks could be executed before the Timer Task increasing the RT-alarms latency.

RT-MINIX defines a new set of signals to indicate special situations, such as missed deadlines, overload or uncertainty of the schedulability of the set of processes.

1.6. MINIX4RT Features

Existing RTOS can be divided in two categories:

– Systems implemented using somewhat stripped down and optimized (or specialized) versions of conventional time-sharing OS.

– Systems starting from scratch, focusing on predictability as a key design feature.

MINIX4RT design is based on the former category using MINIX as the conventional OS. It offers a predictable RT-computing environment at a lower cost than proprietary RTOS used for academic purposes. Furthermore, the same applications and tools used to edit text files, compile

programs, list directories contents, etc, which run on MINIX, run without any change on MINIX4RT. This is a important benefit because there is no need to migrate or cross-compile applications.

The following are desirable characteristics in a RTOS:

– *Small kernel size:* This it will enable the operating system to be used on embedded systems.

– *Low context-switching overheads:* This reduce the process activation latency and increase system schedulability.

– *Fast interrupt service:* Often, RTOS respond to external systems through an interrupt mechanism, therefore a fast interrupt service make the system more responsive.

– *User defined process scheduling:* A process could be schedulable periodically, or triggered by an event, or at specified time, etc. It is desirable that the user could select the process scheduling that best fit for his application.

– *Provision for user definable priorities*: The schedulability of a RTOS increase with higher numbers of priority levels that the user can define into his applications.

– *Provision for specification of deadlines:* The deadline is an important processing parameter of Real-Time applications. Some scheduling algorithms use processes' deadlines to assign priorities (i.e. Deadline Monotonic Scheduling). The deadline is used in critical situations where a watchdog monitoring process must be activated when the process could not meet it deadline.

Most of the characteristics described above are fulfilled by MINIX4RT considered in a academic environment.

The major features of MINIX4RT are summarized as follows:

– *Layered Architecture*: As it is explained in Chapter 2, MINIX4RT has a layered architecture that helps to change a component without affecting the others.

– *Real-Time Sub-kernel*: A Real-Time micro-kernel that deals with interrupts, IPC, time management and scheduling is installed *below* MINIX kernel. The advantage of using a microkernel for RTOS is that the preemptability is better, the size of the kernel becomes much smaller, and the addition/removal of services is easier.

– *Timer/Event Driven Interrupt Management*: Device Driver writers can choice among two strategies of Real-Time Interrupt management.

– *Fixed Priority Hardware Interrupt Processing*: A priority can be assigned to each hardware interrupt that let service then in priority order.

– *Two Stages Interrupt Handling*: Interrupt can be serviced in two stages. The hardware interrupt handler (inside interrupt time) performs the first part of the needed work and a software Interrupt handler (outside interrupt time) does the remaining work.

– *Fixed Priority Real-Time Scheduling*: Each process has an assigned priority. The RT-kernel schedules them in priority order with preemption.

– *Periodic and Non-Periodic RT-processing*: A period can be specified for a periodic process; the Real-Time kernel schedules it on period expiration.

– *Synchronous/Asynchronous Message Transfer using Message Queues*: The added RT-kernel offers a new set of Real-Time IPC primitives based on Message Queues.

– *Priority Based Message Queue Discipline*: A priority based discipline could be specified on each Message Queue for message dequeueing.

– *IPC with Basic Priority Inheritance Protocol support*: To avoid the unbounded priority inversion problem among communicating processes (explained in Chapter 5).

– *Receive and Synchronous Send Timeout* Support: To avoid deadlocks and detect dead processes

- *Timer Resolution Management Detached from MINIX Timer*: A Timer interrupt of 50 Hz is emulated for the MINIX kernel eventhough the hardware Timer interrupt has a higher frequency.

- *Process and Interrupt Handlers Deadline Expiration Watchdogs*: The use of watchdog processes is a common use strategy to deal with malfunctioning RT-processes. When a process does not perform its regular function in a specified time (*deadline*) another process (*watchdog*) is signaled to take corrective actions.

- *Software Timers*: They are system facilities used for time related purposes as alarms, timeouts and periodic processing, etc. One particular feature of MINIX4RT is that it handles software timer actions in priority order.

- *Statistics and Real-Time Metrics*: There are several facilities to gather information about the system status and performance (detailled in Chapter 7).

It is widely believed that microkernel based systems are inherently inefficient and a multilayer message transfer kernel has a performance disadvantage when compared with monolithic kernel. But [17] presents evidence that inefficiency is not inherited from the basic idea but from improper implementation.

MINIX4RT provides the capability of running Real-Time processes and MINIX processes on the same machine. These Real-Time processes are executed when necessary no matter what MINIX is doing.

The Real-Time microkernel works by treating the MINIX OS kernel as a task been executed under a small RTOS based on software emulation of interrupt control hardware. In fact, MINIX is like the *idle* process for the Real-Time microkernel been executed only when there are no Real-Time processes to run. When MINIX tells the hardware to disable interrupts, the Real-Time microkernel intercepts the request, records it, and returns to MINIX. If one of these "*disabled*" interrupts occurs, the Real-Time microkernel records its occurrence and returns without executing the MINIX interrupt handler. Later, when MINIX enables interrupts, all handlers of the recorded interrupts are executed.

MINIX4RT can handle devices in two ways:

– *Event Driven (ED)*: An ED-interrupt handler is executed when the Hardware Interrupt occurs or its execution is delayed until its priority will be greater than the priority of the current process.

– *Timer Driven (TD)*: A TD-interrupt handler is executed only on the expirations of its specified period.

The current version of MINIX4RT is based on version 2.0.2 for 32 bits INTEL [18] processors of MINIX; and thus it requires the same hardware platform.

1.7. Organization

The thesis is organized as follows. Chapter 2 describes *Architecture and Interrupt Handling* topics on MINIX, RT-MINIX, and MINIX4RT. Chapter 3 present details of Real-Time *Process Management and Scheduling*. Chapter 4 contains a discussion about *Time Management*. Chapter 5 is describes *Real-Time Interprocess Communication (RT-IPC)*, priority inversion and the priority inheritance protocol. Chapter 6 provides a detailled explanation of *Real-Time System Calls, Kernel Calls and Kernel Functions*. Chapter 7 is devoted to *Real-Time Processing Related Statistics*. Chapter 8 describes future works and summarizes the results of this work.

At the end of this document, a reference of MINIX4RT System Calls and Kernel Calls can be found in Appendix A. Appendix B presents several sample programs. Appendix C describes the set of tests carried out on the Real-Time system and shows its results and Appendix D shows main system data structures.

This thesis does not cover neither topics about Memory Management, File System Management nor Network Managment because MINIX services are used for these issues.

1.8. Terminology and Notation

In Real-Time terminology, a *Task* is the term often used for a *process*, but in MINIX terminology, a *Task* refers to a special *process* type used in the implementation of MINIX device drivers.

Other confusing term in computer science is the IBM-compatible Personal Computer (PC) device that can produce interrupts at regular periods (ticks). MINIX routines refer to it as the *Clock*, but *Timer* is the correct term for that device.

From here, Real-Time related words will be preceded by "RT-" prefix and Non Real-Time related words will be preceded by “NRT-“ prefix.

Additional terms used in this document are included in the *Glossary* to clarify terminology and notation.

2. ARCHITECTURE AND INTERRUPT HANDLING

This chapter reviews some MINIX background information, its architecture and interrupt processing needed to understand the MINIX4RT design approch. Also, it give the details of the MINIX4RT architecture, how interrupt virtualization is accomplished, the type of interrupt handling that systems programmers could choose to attend devices, and some considerations about priority inversion, a common problem presents in the design of RT-systems.

2.1. MINIX System Architecture and Interrupt Processing

MINIX is a collection of processes that communicate with each other and with User-level processes using message passing. This design results on a modular and flexible architecture, making it easy to replace one component without having even to recompile other modules[2].

MINIX is structured in four layers as it can see in Figure 2.1.

- *Layer 1:* The kernel that provides context switching, process scheduling, interrupt handling, basic memory management and IPC.
- *Layer 2:* Tasks that handle low level I/O operations.
- *Layer 3:* Server processes that handle System Call services with a Memory Manager server (MM) and a File System server (FS) (other servers could be added).
- *Layer 4:* The User-level processes such *init*, shells, compilers, editors, etc.

Each layer only communicates with the ones immediately above and below through the message passing primitives that scale very easily to distributed systems. Therefore, There is not

difficult to modify or to add new components without breaking what already works making it a good choice for teaching the design and implementation of an OS.

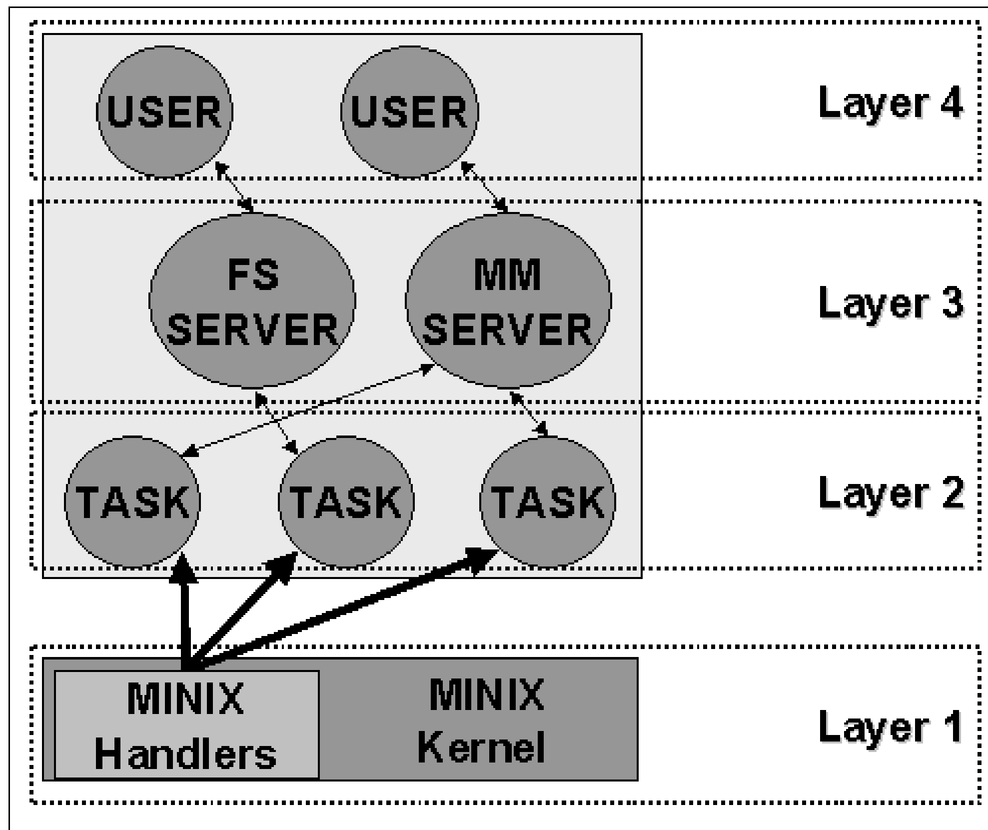


Figure 2.1: MINIX Architecture.

MINIX message passing primitives have some constraints:

- All messages have fixed and small sizes.
- The queue of waiting processes is in the process table.
- User processes can only communicate with Servers, which in turn communicate with I/O Tasks.
- Usually, an application does not construct messages by itself, this is accomplished by the System Calls library code.

Message passing is also used by the kernel to hide hardware interrupts. An interrupt is usually defined as an event that alters the sequence of instructions executed by a processor [15]. In MINIX, when a hardware device interrupts the CPU, an interrupt handler is called, but if more time is needed to complete the job, the handler sends a message to the device Task calling the scheduler on exit. As the scheduler gives I/O Tasks greater priority than User-level processes and Servers, the device Task

is executed to resume the interrupt service out of interrupt time. This approach is often called two stages interrupt handling.

An I/O Task is like a kernel thread that share kernel address space but it has its own processing attributes. The use of an I/O Task to complete the interrupt processing performs well enough in a time sharing environment but can introduce unbounded delay in RT-processing. Two factors affects the interrupt service response time:

1. MINIX scheduler uses three priority queues, one for I/O Tasks, one for Server processes and one for User-level processes. As each queue is arranged in FIFO order, it is not suitable to be used in time constrained systems where a priority order is needed.

2. MINIX hides interrupts using message transfers. It is very common that on each hardware interrupt the kernel sends a message to an I/O Task. This fact forces a context switch before running the Task increasing the system latency and reducing the schedulability of RT-processes.

A key component in the Intel x86 hardware architecture is the Interrupt Descriptor Table (IDT) [18]. The IDT associates each interrupt or exception vector with the address of the corresponding interrupt or exception handler. The IDT must be properly initialized before the kernel enables interrupts. 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 Non-Maskable Interrupts (NMI), the following 16 are assigned to maskable interrupts, that is, to interrupts caused by Interrupt Requests of hardware devices (IRQs). The remaining entries are available for processor traps, that INTEL designates as *"software interrupts"*. MINIX defines a table called *irq_table[]* that has function pointers to interrupt handlers codified using the C programming language.

2.2. MINIX4RT System Architecture

As MINIX4RT intends to be used in an academic environment, its design has been done to be as least intrusive as possible in the standard MINIX source code. Yodaiken and Barabanov [9] have proposed a separate, small, RT-kernel between the hardware and Linux (often called a sub-kernel) for RTLinux. The key idea is how interrupt management is done. As result, one RTOS hosts a standard time sharing OS . Those OSs have their own sets of System Calls.

MINIX4RT follows RTLinux approach where MINIX4RT hosts the standard MINIX time sharing OS. The left side of Figure 2.2 shows the MINIX OS framed by a dotted line, supported by the MINIX4RT kernel. At the right side of Figure 2.2 shows RT-handlers, RT-tasks and RT-processes also supported by the RT-kernel. At the center of Figure 2.2, there is a Task named *MRTTASK* that function as a glue among MINIX applications and the RT-kernel (explained in Chapter 6).

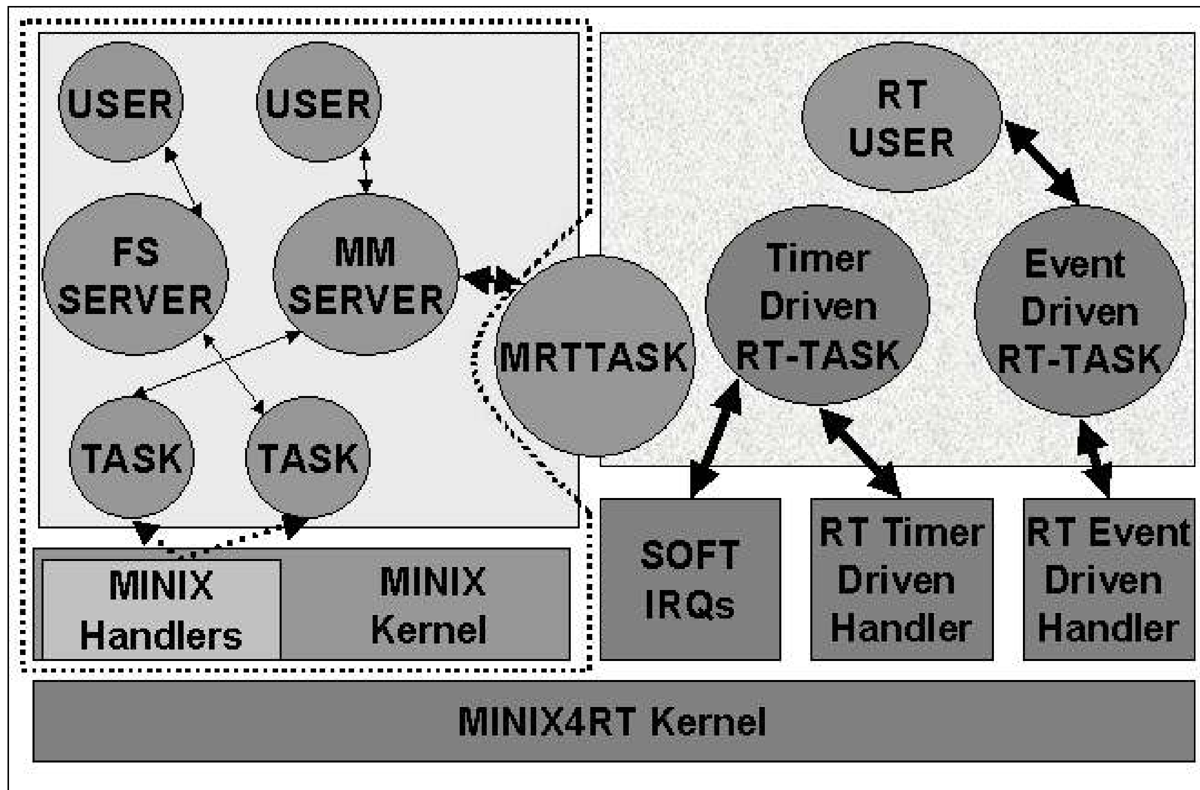


Figure 2.2: MINIX4RT Architecture.

It can be consider a RTOS to comprise two distinct classes of executable entities [19]:

1. Processes
2. Interrupt Handlers

Therefore, there is a need of two types of schedulers, one for processes and other for interrupt handlers. MINIX4RT effectively puts in place a new process scheduler that treats the MINIX kernel as the lowest priority process executing under the RT-kernel, and an interrupt handler scheduler to control the order of execution of interrupt handlers.

As NRT-interrupt handlers could block RT-processes or RT-interrupt handlers, the RT-kernel installs an interrupt dispatcher that executes the handler only if its priority is greater than the priority of the running RT-process or RT-handler. But, in spite of this fact that the handler could not be

executed, the interrupt is accepted and a RT-kernel code is executed consuming processor time that produce an interference to the running RT-process or RT-handler.

Under that design, MINIX only executes when there are no RT-process to run, and the RT-kernel is inactive. Thus, a MINIX process can never disable hardware interrupts or prevents 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.

To carry out with the functionalities described in the previous paragraph, the following issues must be solved:

- Interrupts must be captured by the RT-kernel.
- RT-scheduler and RT-services must be implemented.
- RT-applications need an interface layer to interact with the RT-kernel.
- RT-applications may need transfer data and synchronize with NRT-applications.
- Full process and interrupt handler preemptability is needed.

2.3. Interrupt Handling

As RTLinux does, MINIX4RT uses the Virtual Machine (VM) concept limited to interrupt emulation or virtualization. Its microkernel is underneath of MINIX and it runs NRT-processes only when there are not any RT-process ready to run.

Since interrupts can come at any time, the kernel might be handling one of them while another one (of a different type) occurs. This should be allowed as much as possible since it keeps the I/O devices busy. As a result, the interrupt handlers must be coded to run in a nested way.

When each interrupt handler terminates, the kernel must be able to resume execution of the interrupted process or switch to another process if the interrupt signal has caused a rescheduling activity or executes another lower priority interrupt handler.

Although the kernel may accept a new interrupt signal while handling a previous one, some critical regions exist inside the kernel code where interrupts must be disabled. Such critical regions

must be limited as much as possible since, the kernel, and in particular the interrupt handlers, should run most of the time with the interrupts enabled.

MINIX4RT avoids disabling interrupts for extended periods to improve the system response time. In spite of that, the RT-kernel disables interrupts (by intervals as short as possible) to protect data structures that are also accessed by interrupt handlers avoiding race condition. The coarse time granularity among disabling and enabling interrupts could inflict unpredictable interrupt dispatch latency.

RT-interrupt handlers can easily be replaced with NRT-handlers without recompiling the kernel. This feature is especially useful in certain debugging situations.

MINIX4RT operates in two processing modes, that will be explained in Chapter 3:

- *Non Real-Time Mode*: In this mode, only standard MINIX interrupt handlers and NRT-processes are executed and a limited number of RT-System Calls are allowed (i.e. a System Call that enables switching to Real-Time Mode). The RT-kernel functions are disabled.

- *Real-Time Mode*: In this mode, the system is controlled by the RT-kernel and all kind of interrupt handlers and processes can be executed. When an interrupt occurs, the RT-handler is invoked for a RT-defined interrupt, otherwise its NRT-handler is called.

The system starts in *Non Real-Time Mode*. To start the *Real-Time Mode* a NRT-process must invoke the *mrt_RTstart()* System Call.

2.3.1. Interrupt Handling Virtualization

The MINIX kernel disable interrupts for synchronization when it enters into critical sections avoiding that MINIX could be preempted when a RT-interrupt occurs.

MINIX4RT modifies some functions (and emulate its original behaviour) to avoid that the MINIX kernel could disable interrupts and could not be preempted by the RT-kernel. The trick is quite simple because MINIX uses the following functions for interrupt handling:

- *lock()*: Disables CPU maskable interrupts (CLI for Intel x86).

- *unlock()*: Enables CPU maskable interrupts (STI for Intel x86).

– *put_irq_handler()*: Registers an interrupt handler.

– *disable_irq()*: Disables a Programmable Interrupt Controller (PIC) Interrupt ReQuest (IRQ) line specified as a parameter.

– *enable_irq()*: Enables a PIC IRQ line specified as a parameter.

The exposed operation is guaranteed only is all device drivers and interrupt handler use those functions to handle interrupts without using assembler instructions or own code.

2.3.2. Hardware Interrupts Emulation

The RT-kernel installs a layer of emulation software between the MINIX kernel and the interrupt controller hardware. On RT-Mode, the emulator catches all hardware interrupts and redirects them to either standard MINIX handlers or to RT-kernel handlers. The RT-kernel provides a framework onto which MINIX4RT is mounted with the ability to fully preempt MINIX.

Whenever a NRT-interrupt happens during the execution of a higher priority RT-process, a bit in a bitmap is set by the *MRT_IRQ_dispatch()* function as it will be explained in Section 2.3.7. On returns of interrupts or on returns from System Calls the function *MRT_flush_int()* is called. This function is devoted to execute all pending interrupt handlers, but if MINIX has disabled interrupts using the emulated *lock()* function or the PIC IRQ line for this interrupt has been disabled with the emulated *disable_irq()*, the NRT-handler will not be executed. Later, when the MINIX kernel (virtually) re-enables interrupts, using the emulated *unlock()* function, all pending interrupts are executed.

A drawback of this approach is that the MINIX kernel suffers a slight performance loss when MINIX4RT VM is added due to processing time consumed by:

– The redirection of interrupt handlers to a common interrupt dispatcher.

– The interrupt mask/unmask functions.

– The search of pending interrupts in the interrupt descriptor queues (explained in Section 2.3.12)

– The deferred execution of interrupt handlers.

– The status and statistical information gathering as part of interrupt handling.

In consideration of both strengths and weaknesses, this strategy has shown itself to be flexible because it removes none of the capability of standard MINIX, yet it provides guaranteed scheduling and response time for critical processes.

The changes to standard MINIX are minimal with the VM approach. This low level of intrusion on the standard MINIX kernel improves the code maintainability to keep the RT-related code up-to-date with the latest release of the MINIX kernel.

2.3.3. User and Kernel Stacks

The stack is a LIFO list. Stacks are very useful for passing parameters between subprograms and for storage of variables or identifiers for recursive programs and languages with scope-limited variables, such as in "C". Stacks are ideal data structures for an OS process manager to track the status of processes in various states. In MINIX (and MINIX4RT) each process has two stacks:

– *User-Mode stack*: In User-Mode processing, only this stack can be used.

– *Kernel-Mode stack*: When entering the Kernel-Mode processing, the system switches to this stack.

On interrupts and system calls, the User-Mode stack is changed to the Kernel-Mode stack. If new interrupts occur during the service of other interrupts (nested interrupts), the stack remains in Kernel-Mode.

The variable *k_reenter* counts the level of reentrancy in the kernel:

– *k_reenter = (-1)*: When the system is in User-Mode.

– *k_reenter = 0*: When one kernel control path is running. That can be a system call, an exception/fault handler or an interrupt service routine.

– *k_reenter > 0*: When more than one kernel control path is running. This occurs on nested hardware interrupts.

To monitor the Kernel-Mode stack use, each interrupt descriptor (described in Section 2.3.6) has a field named *reenter* that keeps the maximum kernel reentrancy level (*k_reenter*) for each IRQ. It helps to size the Kernel-Mode stack for specific uses.

As it is expected that a RTOS will receive much more interrupts than a time-sharing OS, by default, the RT-kernel stack doubles in size MINIX's kernel stack.

2.3.4. Interrupt Handler Types

Not all interrupts have the same urgency. In fact, the interrupt handler itself is not a suitable place for all kind of actions. Long non-critical operations should be deferred, since while an interrupt handler is running, the signals on the corresponding IRQ line are ignored.

MINIX4RT defines the following kinds of hardware interrupt handlers:

- *Non RT-handler*: When the system is in NRT-mode, only NRT-handlers are executed. When the system is in RT-Mode, the NRT-handler is executed only if there are not any running RT-process or RT-handler; otherwise it is marked as triggered for later processing. The execution priority of NRT-handler is *MRT_PRILOWEST*.

- *RT Event-Driven (ED) handler*: When the system is in RT-Mode the RT ED-handler is executed only if its priority is greater than the priority of the interrupted RT-process or RT-handler, otherwise it is is marked as triggered for later processing.

- *RT Timer-Driven (TD) handler*: This type of handlers does not execute when the device interrupt occurs. They are executed on Timer Interrupts defined by a period. On each device interrupt, the handler is marked as triggered and it will be processed once it reaches its period following a Timer Interrupt.

The Timer interrupt handling differs from other IRQs; some actions are executed on interrupt time, but other are delayed to handle time related software facilities named Virtual Timers (VT) (explained in Chapter 4).

2.3.5. Interrupt Service Routines

At startup, the RT-kernel initializes the IDT (Interrupt Descriptor Table) pointing each entries of master PIC hardware interrupts to a code generated by the macro *hwint_master(irq)*. The entries for the slave PIC hardware interrupts are filled with the address of a code generated by the macro *hwint_slave(irq)*.

In RT-Mode, all interrupt service routines perform the same basic actions:

- Save the registers contents in the Kernel-Mode stack.
- Increase the kernel variable *k_reenter* (initialized in -1).
- If *k_reenter* = 0, the state of the User-Mode process is saved, otherwise the system is already in Kernel-Mode.
- Send an acknowledgment to the PIC that is servicing the IRQ line, thus allowing it to issue further interrupts.
- Execute the interrupt handler dispatcher *MRT_IRQ_dispatch()*.
- Terminate by jumping to the *restart* label if the *k_reenter = 0* or to *restart1* label for *k_reenter > 0*. More details in Section 2.3.11.

2.3.6. Interrupt Descriptor Data Structure

This section explains the data structures that supports interrupt handling and how they are laid out in various descriptors used to store information on the state, statistics and behavior of interrupt handlers.

The RT-kernel has its own interrupt descriptor table (other than INTEL's IDT) named *MRT_si.irqtab[]*. It is an array of *MRT_irqd_t* data structures that has one descriptor for each hardware and software interrupt (explained in Section 2.3.10.).

The *MRT_irqd_t* data structure has the following functional fields (see Appendix D):

- *nrthandler*: A pointer to a function that is the NRT-handler.
- *rthandler:* A pointer to a function that is the RT-handler.

– *period*: The processing period of a TD-interrupt handler in RT-ticks units.

– *task*: The RT-handler will not execute, instead a *MT_INTERRUPT* message will be sent to the speciried *task* when an interrupt occurs.

– *watchdog*: The RT-PID (RT process ID defined by *mrtpid_t* data type) of a RT-process that will take corrective actions against RT-handler missed deadlines. When a handler does not complete it's work before it's deadline, the RT-kernel sends a *MT_DEADLINE* message to the specified *watchdog* process.

– *priority*: Specifies the handler priority.

– *irqtype*: Specifies type of handler. It is a logical-OR of the following attributes:

- *MRT_RTIRQ*: Real-Time handler (otherwise it will be NRT-IRQ).
- *MRT_TDIRQ*: Timer-Driven handler (otherwise it will be ED-IRQ).
- *MRT_SOFTIRQ*: Software interrupt handler, explained in Section 2.3.10. (Otherwise it will be a hardware interrupt handler).

The *MRT_irqd_t* data structure has the following fields for kernel internal use:

– *irq*: The IRQ number.

– *harmonic*: It is the harmonic frequency of the MINIX Timer interrupt frequency (stored in a system variable named *MRT_sv.harmonic)* when RT-processing Mode starts. It is only used for Timer-Driven Interrupt descriptors to convert the period of a TD-interrupt handler when the user has changed the Timer interrupt frequency. More details in Chapter 4.

– *pvt*: A pointer to an assigned VT for TD-interrupt descriptors.

– *flags*: Some interrupt descriptor status and configuration flags that determines its behavior.

– *shower*: A counter for TD-interrupts in the last period (explained in Section 2.3.20).

- *latency*: The estimated interrupt handler latency in Timer Hz.
- *before*: An auxiliary field that stores the Timer-2 latch counter of the Programmable Interrupt Timer (PIT) in Hz on *MRT_IRQ_dispatch()* entry.
- *next*: A pointer to the next interrupt descriptor in the queue.
- *prev*: A pointer to the previous interrupt descriptor in the queue.
- *name*: The name of the handler.

The *MRT_irqd_t* data structure has the following statistical use fields:

- *count*: An interrupt counter for statistics.
- *mdl*: A missed deadlines counter.
- *maxshower*: Stores the maximum value of *shower*.
- *timestamp*: The last interrupt timestamp.
- *maxlat*: The maximum value of *latency*.
- *reenter*: Stores the maximum value of *k_reenter*. (Explained in Section 2.3.3).

2.3.7. Interrupt Handler Dispatching

MINIX4RT VM sets all Hardware Interrupt Service Routines (ISR) labeled as *HWINTxx* to call an interrupt dispatcher function named *MRT_IRQ_dispatch()*. This function base its decisions to process interrupts based on the interrupt descriptor table *MRT_si.irqtab[]*(see Figure 2.3).

A RT-kernel variable called *MRT_sv.prtylvl* stores the current system priority level of execution. This variable is set to the current process priority or to the running interrupt handler's priority. It is used to reduce the unbounded priority inversion problem deciding on the execution of an interrupt handler or to defer it.

If MINIX4RT is running in NRT-mode, the standard interrupt handlers (*irq_table[irq](irq)* in Figure 2.3) are executed without any interception, deferring and statistics gathering.

In RT-Mode, the function *MRT_IRQ_dispatch()* performs the following actions (see Figure 2.3 and Figure 2.4):

– Some statistics-related variables are updated as:

  - The system wide interrupt counter named *MRT_sv.counter.interrupts.*
  - The descriptor interrupt counter named *MRT_si.irqtab[irq].count*.
  - The kernel reentrancy level named *MRT_si.irqtab[irq].reenter*.

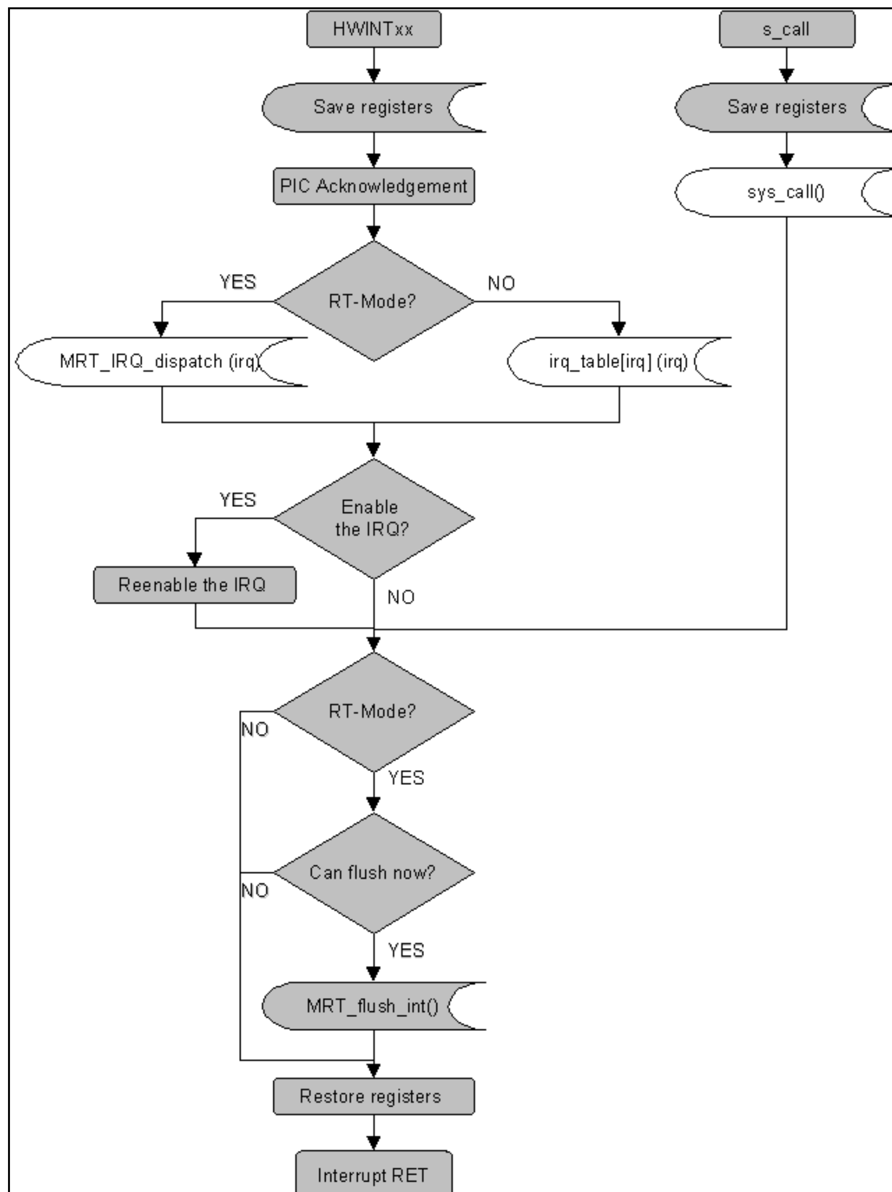


Figure 2.3: Hardware Interrupt Handling.

– If the bit *MRT_LATENCY* for latency computation has been set in the descriptor, the timer-2 of the PIT is read before running the handler and its value is stored in the before descriptor field.

– If a Timer interrupt has occurred:

- The system tick counter *MRT_sv.counter.ticks* is increased.
- The interrupt descriptor *timestamp* field is set.
- The Timer interrupt descriptor is marked for deferred processing only if it has some additional work to do controlled by Virtual Timers (more in Chapter 4).
- Returns from Interrupt and reenable the IRQ0 of the PIC.

– For interrupts other than the Timer, the interrupt descriptor timestamp field *MRT_si.irqtab[irq].timestamp* is set to *MRT_sv.counter.ticks*.

– If an Event-Driven (ED) interrupt has occurred with its priority greater than or equal to *MRT_sv.prtylvl*, the handler is called (**Note**: higher priority means a lower value in the *priority* field). The handler can signal the kernel to run the RT-scheduler on exit setting a bit (*MRT_SCHEDULE*) in the kernel variable *MRT_sv.flags*.

– If an ED-interrupt has occurred with its priority lower than *MRT_sv.prtylvl*, the descriptor is marked for deferred processing.

– If a NRT-interrupt has occurred with its priority greater or equal than *MRT_sv.prtylvl*, the handler is called and then exits.

– If a NRT-interrupt has occurred with its priority lower than *MRT_sv.prtylvl*, its descriptor is marked for deferred processing.

– If a TD-interrupt has occurred, the bit *MRT_TDTRIGGER* in *flag* field of the interrupt descriptor is set. This bit signals the interrupt associated VT that the TD-handler must be executed on the next period. The operation of TD-interrupts using VTs is explained in Chapter 4.

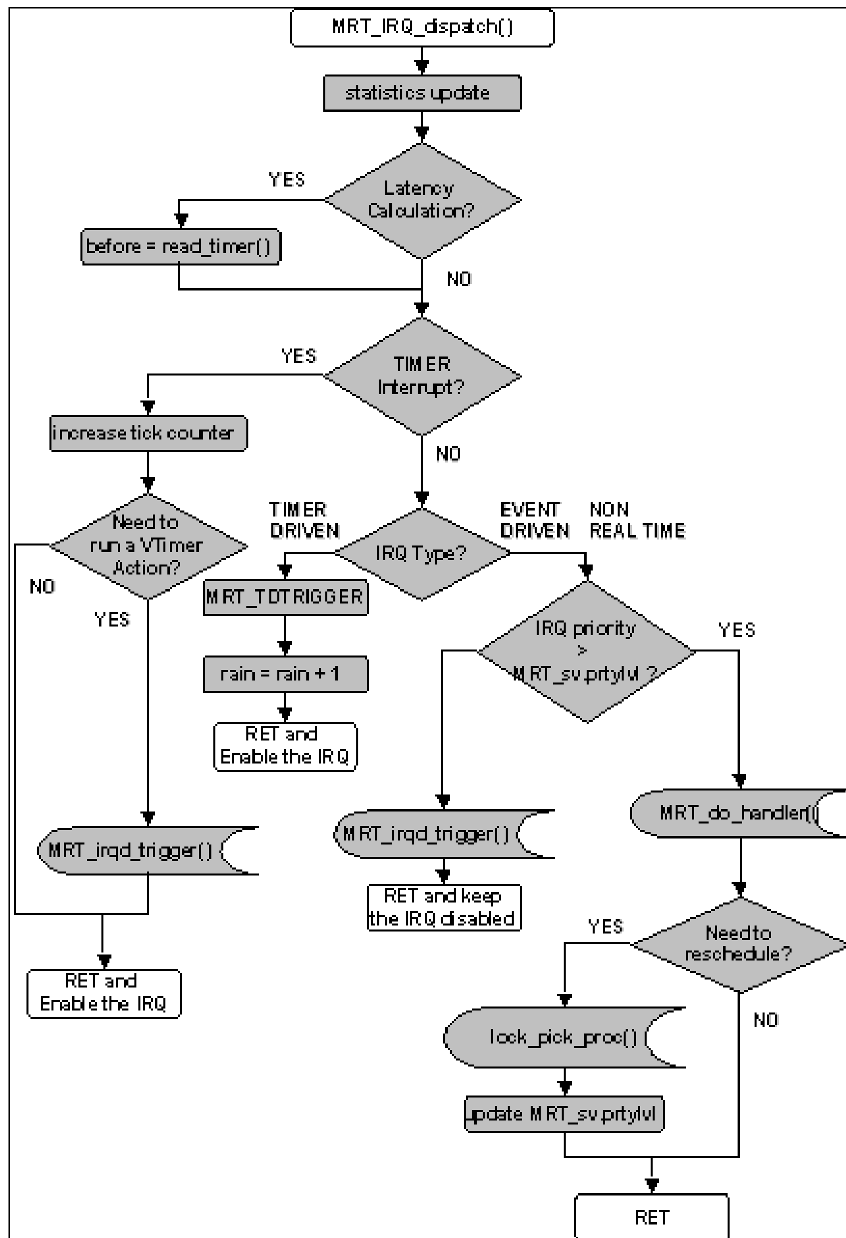


Figure 2.4: Interrupt Handler Dispatching.

2.3.8. Interrupt Handler's Priority

The PIC treats interrupts according to their priority level that are directly tied to the IRQ number. Higher-priority IRQs may improve the performance of the devices that use them.

The standard PC hardware has assigned priorities for standard interrupts related to IRQ number as shown in Figure 2.5. A lower IRQ number implies higher priority. Newer Advanced Programmable Interrupt Controllers (APICs) allow programmers to change the

priority of each IRQ line. The RT-kernel attends IRQs based on the *priority* field of their descriptors.

If a lower priority interrupt occurs during the execution of a running process or handler, its interrupt descriptor is marked as triggered and its processing is delayed. Later, after a context switch, a System Call or a Return from Interrupt, the RT-kernel calls *MRT_flush_int()* function that scans the queues for interrupt descriptors that has been triggered and runs their handlers.

MRT_flush_int() alternatively calls *MRT_irqd_flush()* that scans for triggered interrupt descriptors and *MRT_vtimer_flush()* that runs actions of expired VTs (see Section 4.8.3) in priority order.

The following pending interrupt handlers are not executed by *MRT_irqd_flush()*:

– NRT-interrupts once MINIX has (virtually) disabled processor interrupts using the emulated *lock()* function.

– All pending interrupt handlers with lower priority than the current system priority level of execution *MRT_sv.prtylvl*.

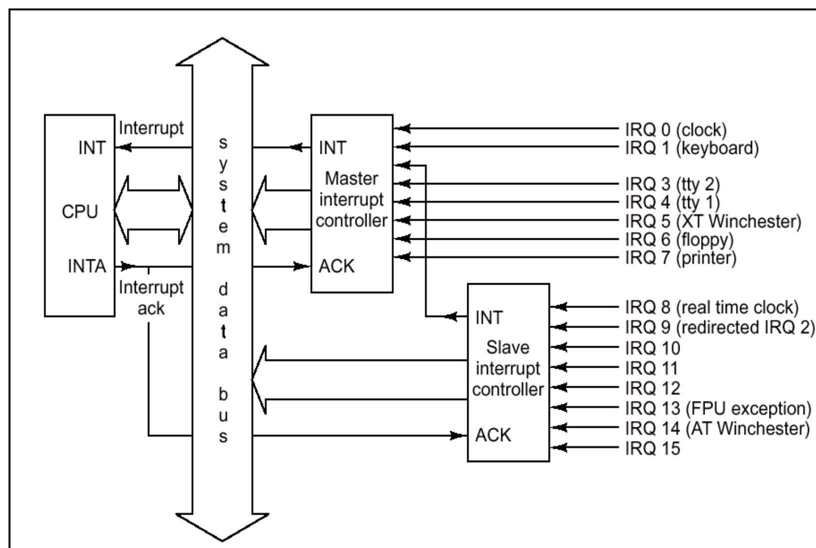


Figure 2.5: IRQ Priorities (from [2]).

Each RT-handler must notify the RT-kernel setting the bit *MRT_SCHEDULE* in *MRT_sv.flags* if it needs to run the RT-scheduler on exit

2.3.9. Real-Time Input/Output Tasks

In MINIX, as in other OSs, each kind of input/output device has a related device driver. MINIX device drivers have two components:

- An *Interrupt Service Routine* (ISR) that catchs device interrupts.
- An *I/O Task* for delayed interrupt processing and to accept requests from server processes.

A was explained in Chapter 1, *Task* is a special kind of MINIX process that has its own state and stack, but shares the kernel address space with the kernel and other *Tasks* (explained in Section 2.1). *Server* processes requests services to *Tasks* using standard message transfers. When a Hardware interrupt occurs, the kernel signals the I/O Task sending it a message, but this fact usually involves a context switch because the Task preempts the running process when receives the message causing a performance penalty.

MINIX4RT allows device driver writers to adopt the use of RT-Tasks for their design. As any RT-process, a *RT-Task* has an assigned priority and any RT-processes could request services to it using RT-Interprocess Communications (explained in Chapter 5).

The interrupt descriptor associated with that RT-Task can be a defined as a TD-interrupt or an ED-interrupt setting or clearing the *MRT_TDINT* bit of the *irqtype* field of the interrupt descriptor.

2.3.10. Software Interrupts

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 do not block the system for too long.

As it is explained above, MINIX uses a two stages interrupt management where an interrupt handler partially process the interrupt and then sends a message to an I/O Task to resume the interrupt processing. This approach implies at least a context switch to restore the state of the task.

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.

Programmers have at least three approaches to design Device Drivers:

A. Complete interrupt processing into the handler.

B. Two stages interrupt management like MINIX using RT-Tasks and message transfers.

C. Two stages interrupt management using Software interrupts as Linux does with *bottom halves*.

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

Software interrupt descriptors have the same data structure *MRT_irqd_t* than Hardware interrupts descriptors, therefore they have priority, counters, timestamps and other fields.

The motivation for introducing software interrupt is to allow a limited number of functions related to interrupt handling to be executed in a deferred manner. This approach increases the system responsiveness because some work is executed out of interrupt time. Moreover, the processing overhead is lower than using the I/O Task model because it avoids the context switch among the interrupted process and the I/O Task. Software interrupts can be used as kernel threads triggered by time using Virtual Timers.

2.3.11. Returning from System Calls and Interrupts Service Routines

The main purpose of the termination phase of interrupt/exception handlers and System Calls is to execute the highest priority process, but several issues must be considered before doing it. The system needs to execute:

- Pending Event Driven Interrupt handlers.
- Pending Software interrupt handlers.
- Virtual Timer Actions (explained in Chapter 4).

– Pending Standard MINIX interrupt handlers

The RT-kernel will execute only those pending issues that have greater priorities (lower *priority* field) than the *MRT_sv.prtylvl*.

Before returning the CPU to User-level Mode those interrupt handlers that has been deferred must be executed. The RT-kernel scans the interrupt descriptor queue for hardware and software pending interrupt handlers and invokes only those handlers with priorities greater than the current process priority. The kernel function that accomplishes with these issues is *MRT_flush_int()* (see Figure 2.3).

2.3.12. Flushing Deferred Interrupts

MINIX4RT uses a system of interrupt descriptors queues and a bitmap named *MRT_si.iQ.bitmap* to process deferred interrupts. A bit set in the i-th position of the bitmap implies that at least one descriptor in the i-th interrupt descriptor queue has been triggered for deferred processing (see Figure 2.6).

An interrupt descriptor is inserted into the queue using the kernel function *MRT_irqQ_ins()* (explained in Section 2.3.15), and it remains in the queue even once the interrupt has been serviced. Those enqueued interrupt descriptors that need service have a bit (named *MRT_TRIGGERED)* of the *flag* field set.

Some test performed by the author during the implementation, proved that inserting a descriptor when the interrupt occurs and removing it once the interrupt has been serviced, has less perfomance that keep the descriptors enqueued. Moreover, the processing time required with the CPU interrupts disabled is reduced improving kernel preemptability.

MRT_flush_int() scans for bits set in two bitmaps, the *MRT_si.iQ.bitmap* for deferred interrupts and the *MRT_st.tQ.bitmap* for expired VTs actions to execute. The scan stops when the priority of the represented by the bit position is lower than *MRT_sv.prtylvl* (see Figure 2.7).

For each bit set in *MRT_st.tQ.bitmap, MRT_flush_int()* calls *MRT_vtimer_flush()* that executes expired VTs actions. This operation is explainded in Section 4.8.2.

For each bit set in *MRT_si.iQ.bitmap, MRT_flush_int()* calls *RMT_irqd_flush()* that scans the interrupt descriptor queue for hardware and software interrupt handlers triggered to run. A handler is executed only if its descriptor status *flags* has the *MRT_TRIGGERED*

bit set and has not been disabled (*MRT_DISABLED*) by software. Once all triggered interrupt descriptors in the same queue has been serviced, the bit in *MRT_si.iQ.bitmap* is cleared.

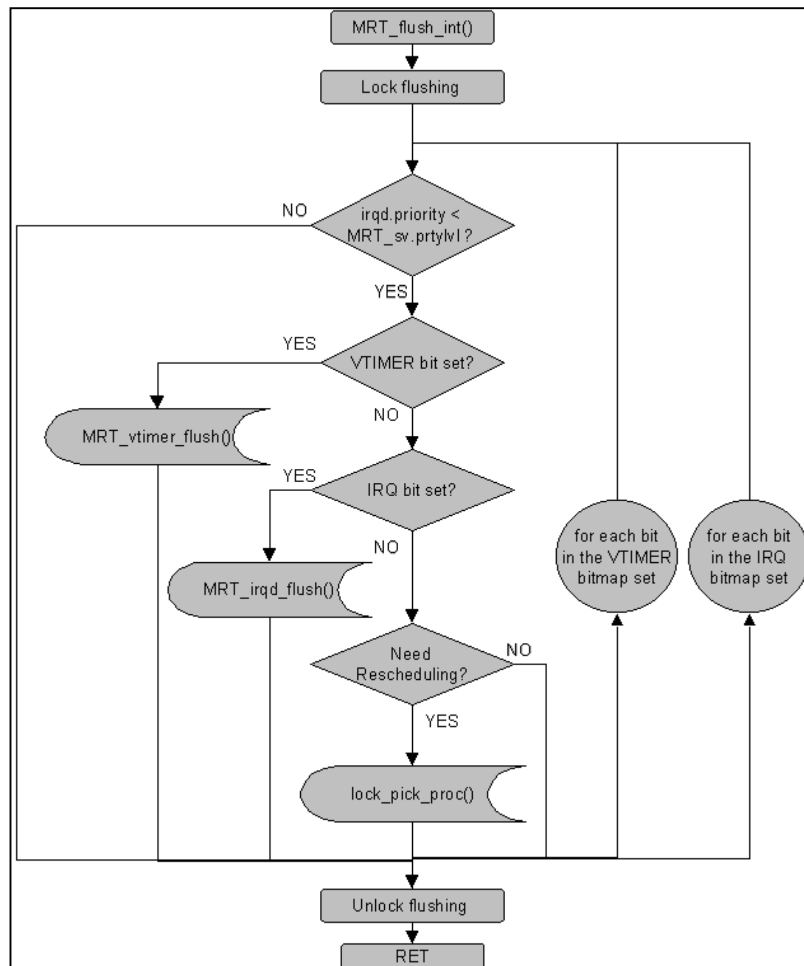


Figure 2.6: Interrupt Bitmap and Interrupt Queues.

If one of the handlers has set the *MRT_SCHEDULE* bit of the system variable *MRT_sv.flags,* the scheduler (*lock_pick_proc()* function) is invoked.

Each interrupt queue descriptor has two counter fields:

– *inQ*: Records the number of descriptors enqueued. It is increased by the *MRT_irqQ_ins()* and decreased by *MRT_irqQ_rmv()* kernel functions.

– *pending*: Records the number of triggered interrupts in the queue. It is increased when some descriptor is triggered by the *MRT_irqd_trigger()* kernel function and is decreased by the *MRT_irqd_serviced()*.

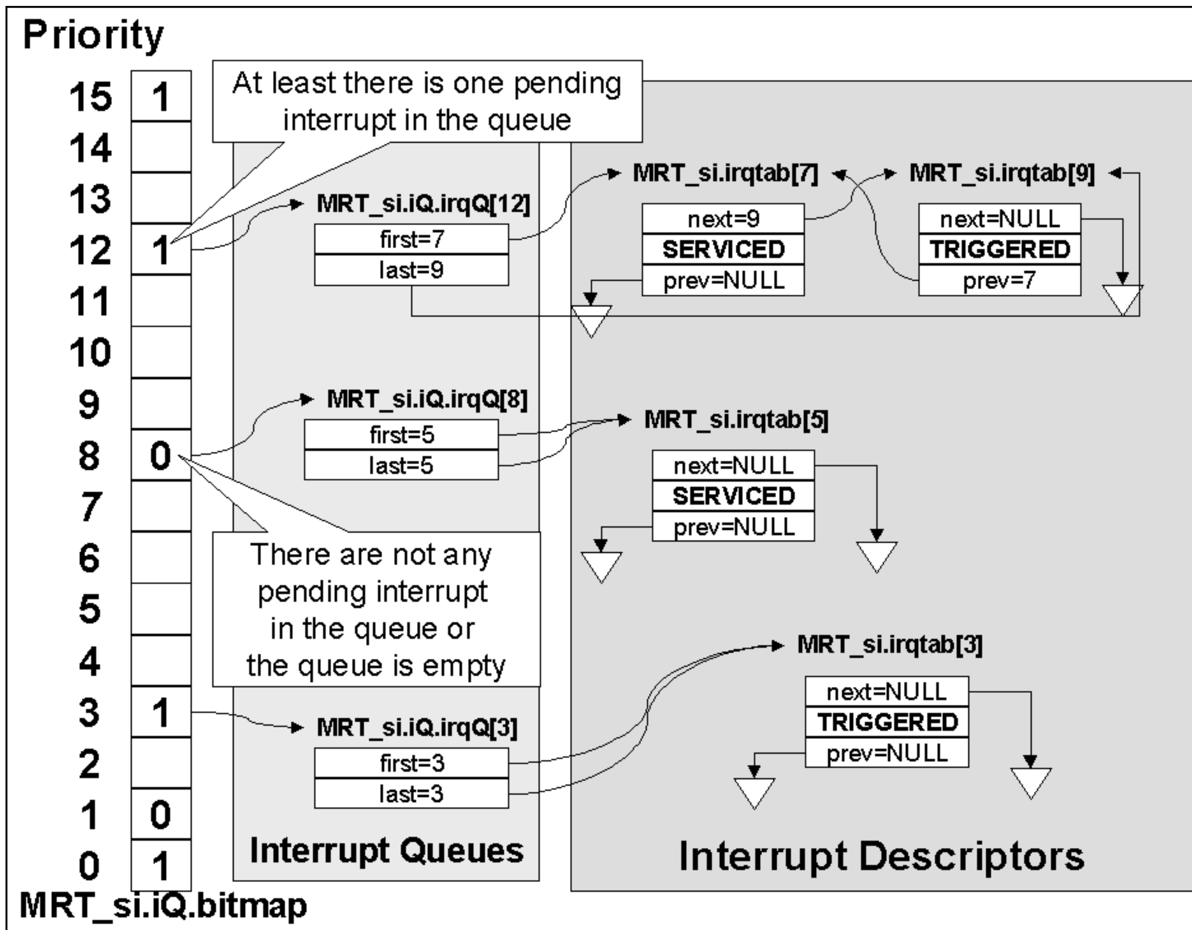


Figure 2.7: Flushing Interrupts and Virtual Timers.

Only that queues with (*inQ* > *1*) will be scanned to find the triggered descriptors. The scanning will stop when (*pending* = *0*).

As during the execution *MRT_flush_int()*, new interrupts can occur, a lock bit (*MRT_FLUSHLCK*) in the kernel variable *MRT_sv.flags* is set to avoid that *MRT_flush_int()* could be called again returning from a nested hardware interrupt. If the nested interrupt has greater priority than the interrupted handler, another bit named *MRT_NEWINT* in *MRT_sv.flags* is set to notify *MRT_flush_int()* that it needs to restart the scan again from the highest priority interrupt queue to find the new triggered interrupt descriptor.

2.3.13. Executing Interrupt Handlers

Before running interrupt handlers, the RT-kernel must (see Fig 2.8):

– Save the current system priority level of execution *MRT_sv.prtylvl*.

– Set *MRT_sv.prtylvl* to the Interrupt handler's priority.

– Restore CPU flags to its previous state.

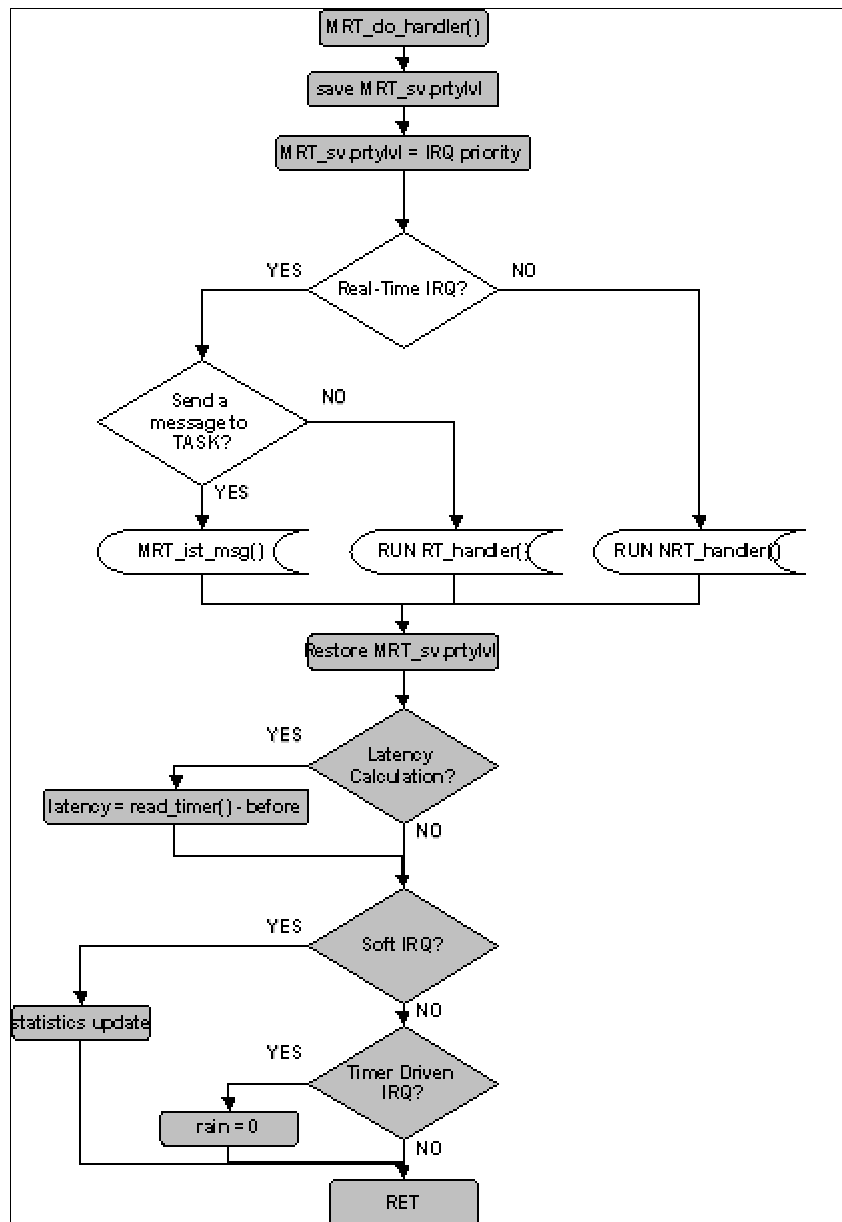


Figure 2.8: Running Interrupt Handlers.

If the interrupt descriptor *flag* field has the bit *MRT_ISTMSG* set, instead of executing a handler, the RT-kernel sends a *MT_INTERRUPT* message to the RT-Task specified in the *task* field of the interrupt descriptor.

Once the handler exits the CPU flags are saved in the stack and the interrupts are disabled and an estimation of the handler processing time could be computed (see Section 2.3.16).

The value returned by a hardware interrupt handler notifies the kernel if the PIC IRQ line must be enabled or not.

2.3.14. Interrupt Descriptor Timestamp Field

MINIX4RT includes a *timestamp* field in the interrupt descriptor data structure *MRT_irqd_t.* For hardware interrupts this field is set by *MRT_IRQ_dispatch()* to the value stored in *MRT_sv.counter.ticks*. For software interrupts *timestamp* field is filled after the handler was executed into the *MRT_do_handler()* function.

The units of the *timestamp* field are RT-ticks since the last execution of the *mrt_RTstart()* or *mrt_restart()* System Calls.

2.3.15. Kernel Functions for Interrupt Handling and Synchronization

Interrupt disabling is one of the key mechanisms used to ensure that a sequence of kernel statements is treated as a critical section. It allows a kernel control path to continue executing even when hardware devices issue IRQ signals, thus providing an effective way to protect data structures that are also accessed by interrupt handlers [15].

On Intel x86 architecture, when the kernel enters a critical section, it clears the *IF* flag of the *EFLAGS* register to disable interrupts. But, at the end of the critical section, often the kernel can not simply set the flag again. As interrupts can execute in nested fashion, the kernel does not necessarily know what the *IF* flag was before the current control path executed. In these cases, the control path must save the old setting of the flag and restore that setting at the end [15].

The RT-microkernel has the following functions to control interrupts:

- *MRT_lock()*: Disables interrupts at the CPU (*CLI* assembler instruction on INTEL).
- *MRT_unlock()*: Enables interrupts at the CPU (*STI* assembler instruction on INTEL).
- *MRT_saveFlock(&CPUflags)*: Saves the current CPU Flag Register into the stack and then clears the *IF* (interrupt flag) using the *CLI* assembler instruction to disable interrupts.

- *MRT_restoreF(&CPUflags)*: Restores the CPU *EFLAGS* Register from the stack.

Each IRQ line can be selectively disabled. Thus, the PIC can be programmed to disable IRQ lines. That is, the PIC can be told to stop issuing interrupts that refer to a given IRQ line, or to enable them. Disabled interrupts are not lost; the PIC sends them to the CPU as soon as they are enabled again. This feature is used by most interrupt handlers since it allows them to process IRQs of the same type serially.

Selective enabling/disabling of IRQs is more efficient than global masking/unmasking of maskable interrupts but they are more time consuming.

The RT-microkernel has the following functions to handle IRQs at PIC level:

- *MRT_disable_irq()*: Disables an IRQ line at PIC level.

- *MRT_enable_irq()*: Enables an IRQ line at PIC level.

- *MRT_get_PIC()*: Reads the PIC mask.

- *MRT_set_PIC()*: Sets the PIC with a specified mask.

Interrupt descriptors are used to store information on the state, statistics and behavior of interrupt handlers. The RT-kernel has the following functions to handle interrupt descriptors:

- *MRT_irqd_set()*: Sets all fields of an interrupt descriptor.

- *MRT_irqd_free()*: Removes an interrupt descriptor from its interrupt descriptor queue. It resets all fields of an interrupt descriptor. The RT-handler and NRT-handler points to the *spurious_irq()* handler.

- *MRT_softirq_set()*: Assigns a new interrupt descriptor from the software interrupt descriptor pool, then sets the descriptor using *MRT_set_irq()*.

- *MRT_softirq_free()*: Removes the descriptor from the interrupt descriptor queue using *MRT_irqd_free()* and frees the descriptor to the software interrupt descriptor pool.

- *MRT_irqQ_ins()*: Inserts an interrupt descriptor into an interrupt descriptor queue.

– *MRT_irqQ_rmv()*: Removes an interrupt descriptor from an interrupt descriptor queue.

Two kernel macros can be used to disable/enable descriptors:

– *MRT_disable_irqd()*: disables an interrupt descriptor in such a way that it will not be executed by the kernel even if triggered.

– *MRT_enable_irqd()*: enables an interrupt descriptor to be executed by the kernel if triggered.

Those macros set/clear the bit MRT_DISABLED in the flag field of the interrupt descriptor that is checked before running the handler.

2.3.16. Estimating Interrupt Handler Processing Time

A vital characteristic of a RTOS is how responsive is in servicing internal and external events. These events include external hardware interrupts, internal software signals, and internal Timer interrupts. One measure of responsiveness is latency, the time between the occurrence of an event and the execution of the first instruction in the interrupt code. A second measure is jitter, the variation in the period of nominally constant-period events.

One of the most common measurements requested to RTOSs is the *Interrupt Service Time*[20]. It is used to measure the effectiveness of a RT-kernel at dealing with extremely high-priority interrupts or emergency interrupts. Often a peripheral must be serviced within a certain time limit after an event. For example, a packet must be read from a network port before the next on arrives.

The *Interrupt Service Time* (T_{ist}) is the maximum time taken to respond to an interrupt request. It includes the time it takes for the current instruction to complete, the time for the CPU to respond to the interrupt (T_{il}) and the interrupt handler processing time (T_{int}) (see Figure 2.9).

$$T_{ist} = T_{il} + T_{int}$$

Understanding the relative size of delays is important to the design of the RT-system. Most sources of delay in a 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 a RT-kernel.

Interrupt Service Time are not fixed in length. Because an interrupt is, by definition, an asynchronous event, an *Interrupt Service Time* depends 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.

MINIX4RT uses the Timer-2 of the Programmable Interrupt Timer (PIT) to estimates the interrupt handler processing time (T_{int}). The PIT is programmed in *SQUARE_WAVE* mode and the divisor latch is set to 0 (to get a 65536 latch value) and its frequency is 1193182 [Hz], therefore its period is 65536/1193182 = 0.0549 [s].

At the start of *MRT_IRQ_dispatch()* kernel function, the value of the counter of the Timer-2 latch of the PIT is stored in the *before* field of the interrupt descriptor. On handler return, the counter is read again and the handler processing time could be estimated as the difference between those two values. The time units of the estimated value are PIT Hz or $(1193182)^{-1}$ [s] = $8{,}38\text{x}10^{-7}$ [s] = 0.838 [µs].

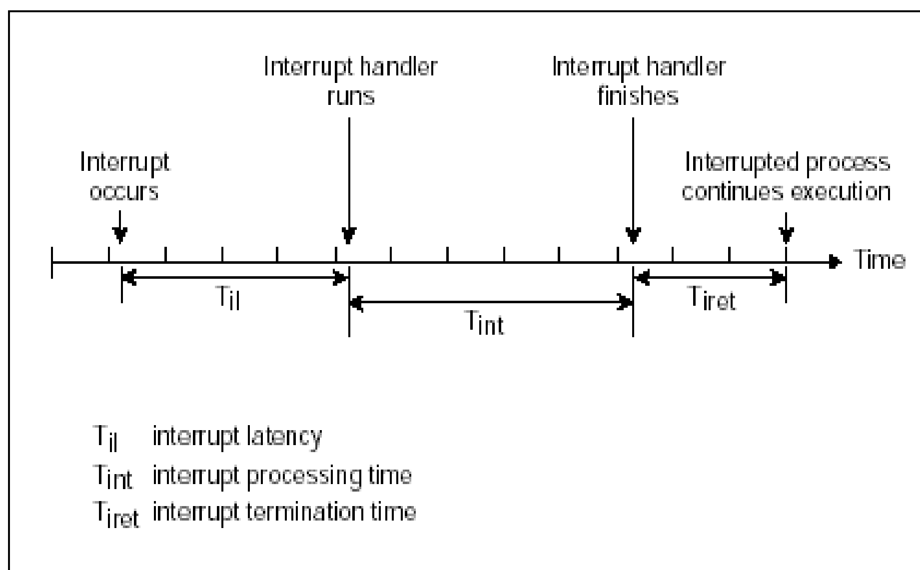


Figure 2.9: Interrupt Service Time (from [11]).

While the estimated interrupt ***handler*** processing time is not so useful as the Interrupt Service time (T_{ist}), it can be used to compare performance among several handler algorithms.

The PIT Timer-0 is programmed in *SQUARE_WAVE* mode as Timer-2. The PIT signals the Timer Interrupt once the Timer-0 counter reaches zero. As the Timer-0 remains decreasing the counters, during the execution of the Timer handler, the value of the Timer-0 counter lets compute the latency of the Timer handler. The estimated value by this method represents the *Interrupt Latency* (T_{il}) of the Timer interrupt.

2.3.17. Nested Interrupts

As has been mentioned above, MINIX4RT supports nested interrupts. Worst-case timing considerations for unmasked interrupts must be included in the computation of the service time for all interrupts currently being processed.

The following criteria is established for nested interrupts:

- A higher or equal priority interrupt will preempt a running handler. The computation time of the higher priority handler must be added to the *Service Time* as a *Blocking Time*.

- A lower priority interrupt will not preempt a running handler. The execution of the new interrupt handler will be deferred for later processing and the function *MRT_flush_int()* will not be invoked on exit to reduce the *Blocking Time*. This blocking time is a priority inversion that must be added to the running handler service time. It must be consider only once because the lower priority interrupt will be disable at PIC level until the handler will re-enable it.

Once the handler exits, the RT-scheduler could be invoked and another process preempts the former as *Process B* and *Process C* preempts *Process A* in Figure 2.10.

Next MINIX4RT releases will include an interrupt mask array named *MRT_sv.mask[]* used to change the PIC mask before running a handler/process to avoid that could be interrupted by a lower priority interrupt. The array items are filled by the *MRT_irqd_set()* kernel function based on the interrupt descriptor's priority. When a new interrupt descriptor is set, all items of the *MRT_sv.mask[]* array are recomputed with PIC masks that consider the new descriptor's priority and the IRQ lines that must be disabled when the process/handler with this priority will run.

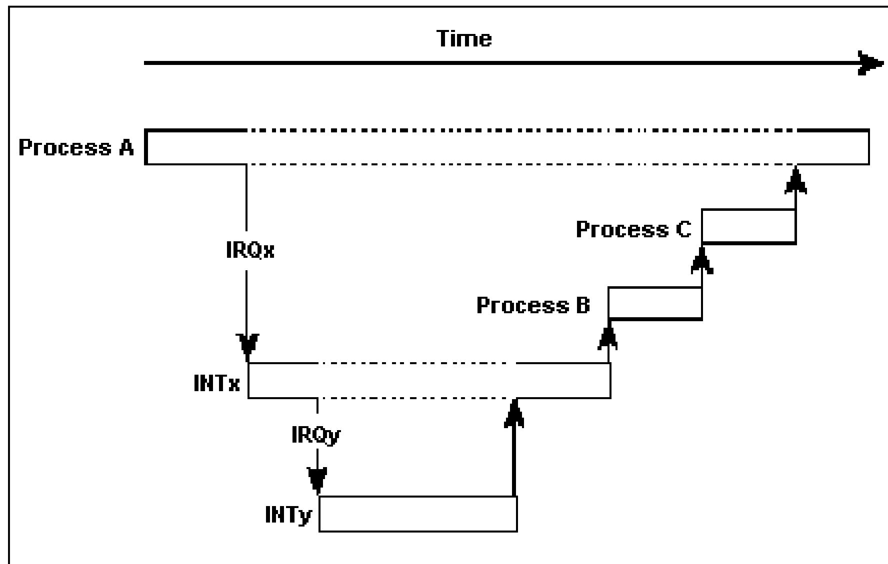


Figure 2.10: Nested Interrupts (from [11]).

2.3.18. Real-Time Interrupt Processing

The following categories of real-time implementations are proposed in [21]:

- 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 processes 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, and is thus non-integrated.

- 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 processes 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.

MINIX4RT does not match strictly in any of these categories but it depends on the set of processes running on the system and the type of hardware where it runs.

MINIX4RT could be considered as an *Event Driven with Non-Integrated Interrupts Scheduling and Timer Driven with counter* system.

– *Event Driven*: On ED-interrupts the handlers are called without delay.

– *Non-Integrated Interrupts*: Hardware interrupt priorities could no match with process priorities.

– *Timer Driven with counter*: The RT-Timer interrupt handler (VT really) invokes the scheduler only when a TD-process must be scheduled or when a TD-interrupt has occured in the last timer period.

2.3.19. Standard MINIX Non Real-Time interrupts

In RT-mode, NRT-handler are executed only if there are not a RT-process or RT-handler running when the NRT-interrupt occurs. The handler execution is delayed until *MRT_flush_int()* invokes it.

2.3.20. Real-Time Timer-Driven Interrupts

Some devices will raise interrupts at high rates but the interrupt processing can be delayed to be managed by a periodic process in the next schedule (Timer Driven).

Other devices do not raise interrupts, and a periodic process can be created to poll the devices checking their status and taking appropriated action.

This kind of interrupt processing lets that more than one interrupt occur in a time period without running the handler. The handler only is executed at a specified period reducing the system overload. MINIX4RT assigns a VT for the TD-interrupt descriptor.

The processing of a TD-interrupt handler has the following stages (see Figure 2.11):

– The TD-interrupt occurs and *MRT_IRQ_dispatch()* sets the *MRT_TDTRIGGER* bit in the *flag* interrupt descriptor field to signal the descriptor for later processing when its assigned VT reaches its period. The

shower field (counts the number of TD-interrupts since the last period) is increased.

– On each Timer interrupt, *MRT_IRQ_dispatch()* checks for expired VTs. If any VT has expired, the Timer handler is triggered and it will run in the next call to *MRT_flush_int()*.

– *MRT_flush_int()* runs the Timer handler that enqueues expired VTs for later processing of their actions. More details in Chapter 4.

– *MRT_flush_int()* calls *MRT_vtimer_flush()* that runs actions of those expired VTs. The action of a VT assigned to a TD-interrupt handler is *MRT_ACT_IRQTRIG*, that triggers the descriptor using *MRT_irqd_trigger()* function.

– *MRT_flush_int()* runs the handler of the TD-handler, afterwards resets the *shower* field and the *MRT_TDTRIGGER* bit in the *flag* fields of the TD-descriptor.

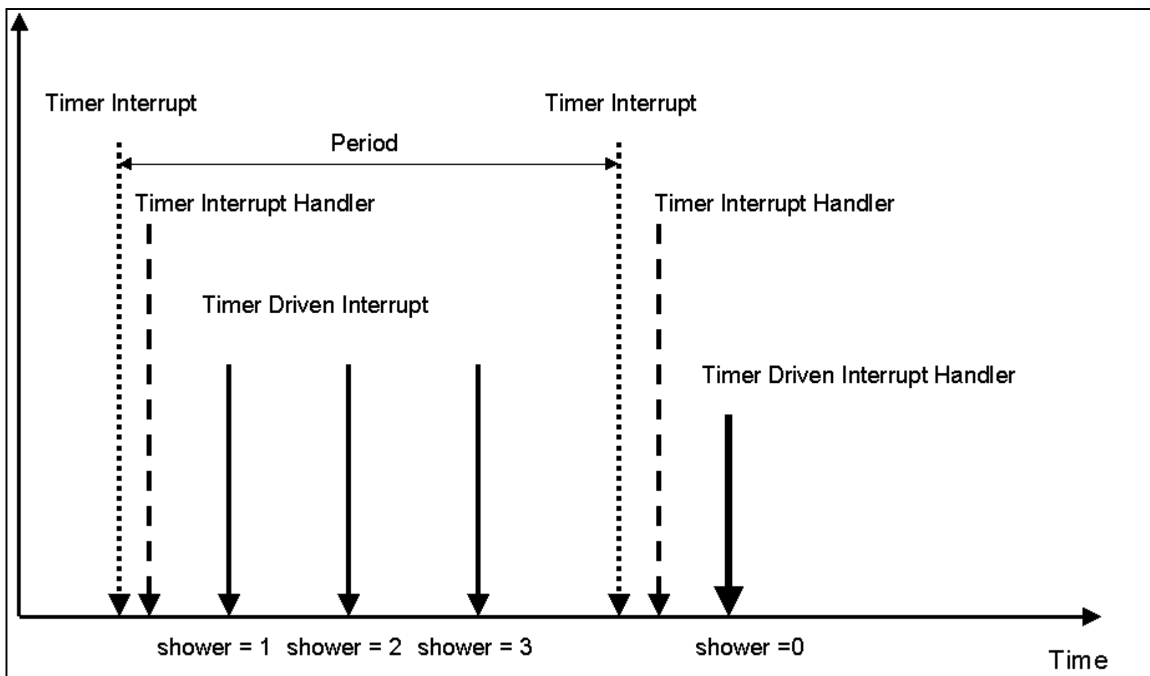


Figure 2.11: Timer Driven Interrupts.

Some devices must be serviced by polling. A software TD-interrupt descriptor can be set associated to a VT with a *MRT_ACT_IRQTRIG* action type and the period that the device needs.

The following is a stage of the operations executed:

– On each Timer Interrupt, *MRT_IRQ_dispatch()* checks for expired VTs. If a VT has expired, the Timer handler is triggered and it will run in the next call to *MRT_flush_int()*.

– *MRT_flush_int()* runs the Timer handler that enqueues expired VTs for later processing of their actions. More details in Chapter 4.

– *MRT_flush_int()* calls *MRT_vtimer_flush()* that runs actions of those expired VTs. The action of a VT assigned to a TD-interrupt handler is *MRT_ACT_IRQTRIG*, that triggers the descriptor using *MRT_irqd_trigger()* function.

– MRT_flush_int() runs the TD-handler that polls the device to check if it needs attention.

2.3.21. Real-Time Event-Driven Interrupts

When *MRT_IRQ_dispatch()* is invoked by a ED-interrupt, it checks if the priority of the interrupted process/handler (*MRT_sv.prtylvl*) is greater than the priority of the interrupt. If it is not, the *MRT_IRQ_dispatch()* triggers the interrupt descriptor for later processing using the kernel function *MRT_irqd_trigger()*. Next, when in the next running of *MRT_flush_int()*, all triggered interrupt descriptors are flushed in priority order.

If the interrupt's priority is greater than *MRT_sv.prtylvl*, the interrupt descriptor RT-handler is invoked with minimal delay.

2.4. Preventing Interrupt Priority Inversion

In RTOS unpredictability is introduced by interrupts from some devices. A type of unbounded priority invertion is produced when a higher priority process is executing and lower priority hardware devices produce an interrupt shower. That showers could only be generated by asynchronous devices like network, USB, parallel or serial port interfaces. Timer interrupts are periodic and the period is set by the kernel. Disks, diskettes and CD-ROMs generate interrupts once they finish with commands commended by the kernel, when the user inserts or removes a removable media, or on error conditions, but never cause a shower.

MINIX4RT handles interrupts on a preemptive basis; when an interrupt occurs, every execution at lower interrupt levels is suspended and execution begins immediately on the highest-level request. Processing continues until the highest-level interrupt processing has been completed. This places a responsibility on device drivers writers in that system responsiveness is directly related to how quickly a device driver exits its interrupt routine [22].

Future realeases of MINIX4RT will offer Prioritized Interrupt Disabling. With this feature the kernel changes the PIC mask raising the interrupt level to avoid that equal and lower level interrupts could use system resources delaying higher priority processing [23].

When a device triggers an interrupt of a given priority, the PIC masks from the CPU all interrupts of priority less than or equal to the device interrupt's priority. They become pending. When the interrupt level on the PIC drops below an interrupt's priority, the PIC lets the interrupt proceed to the CPU.

Priorized Interrupt Disable will be implemented using the *MRT_sv.mask[]* array as was described in Section 2.3.17. Before running interrupt handlers or processes, the RT-kernel checks their priorities and will change the PIC mask to the value stored in *MRT_sv.mask[priority]*. The *MRT_sv.mask[]* array is filled by the *MRT_irqd_set()* kernel function.

2.5. RT-Process Dispatch Latency

It is very often that a RT-process is scheduled by an IRQ related with an external event. The *RT-Process Dispatch Latency* is an important measurement in RTOS and in MINIX4RT is composed by (see Figure 2.12):

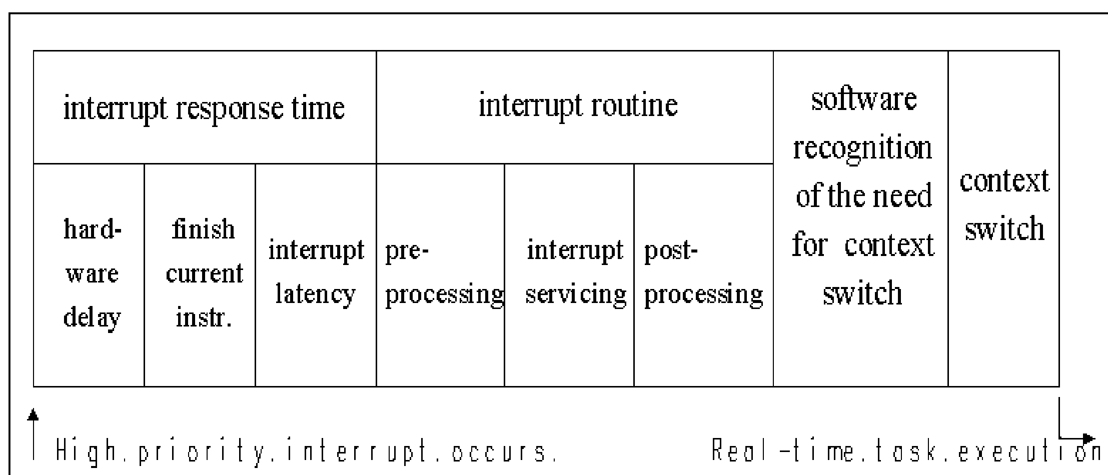


Figure 2.12: RT-Process Dispatch Latency (from [24]).

– *Hardware Delay*: Not controlled by the RTOS.

– Finish Current CPU Instruction: Not controlled by the RTOS.

– *Interrupt Latency*: It is affected by the granularity of the Interrupt Disabling-Interrupt Enabling periods that the RT-kernel use as a mutual exclusion mechanism.

– *Preprocessing Time*: Includes the processing costs of *MRT_IRQ_dispatch()* and the top half of *HWINTxx* functions.

– *Interrupt Servicing Time*: It is the time consumed by the *MRT_do_handler()* function that includes the handler itself.

– *Post Processing Time*: It includes the time consumed by the RT-scheduler, and the processing costs of the bottom half of *HWINTxx* function that makes the context switch.

3. RT-PROCESS MANAGEMENT AND SCHEDULING

There are several identifiable approaches to marrying RT and NRT technologies such as the ones listed below [25]:

1. A general purpose OS with added RT-features such as periodic processing and priority-inversion-free IPC mechanisms.
2. A general purpose OS with a dynamically configurable kernel that is capable of accommodating user or application specific process, scheduling, and memory management modules.
3. A subkernel that support coresident OSs partitioning the CPU into two virtual machines running an unmodified general purpose OS and a RT-kernel.

Yodaiken [26] proposes a variant of the last approch, where a RT-microkernel threats a time-sharing OS as the lower priority task that executes only if there are not any RT-process ready to run. This proposal was adopted for the MINIX4RT kernel and it has the following advantages:

- A clean separation exists between NRT and RT-services.
- The RT-kernel executes in a predictable manner, so it is possible to analyze the conditions under which RT-processes will be guaranteed to be feasible.
- The time sharing OS can function correctly with few modifications.

This chapter describes the composition of RT-processes, their states and transitions, introduces RT-process creation and termination, and gives details about RT-process scheduling.

3.1. MINIX4RT Execution Modes

MINIX4RT starts with the same functionalities as MINIX, therefore it can not run RT-processes. There must be a NRT-process that invokes a System Call that switches the behavior of the RT-kernel to allow the execution of RT-processes.

As it was introduced in Chapter 2, MINIX4RT has two execution modes:

– *NRT-mode*: Any RT-processes neither RT-interrupt handlers can not run in this mode, therefore the system runs as MINIX does.

– *RT-mode*: The system runs under the RT-kernel control and RT-processes and RT-interrupt handlers can be executed sharing the system with NRT-processes and NRT-interrupt handlers.

To switch between those execution modes the *mrt_RTstart()* and *mrt_RTstop()* System Calls are provided. They use the services of a new Task named *MRTTASK* (presented in Chapter 2 and detailled in Chapter 6) that is RT-kernel agent that function as glue among processes and the RT-kernel.

3.2. Real-Time Process Creation

Only NRT-process can be created and terminated under MINIX4RT. The RT-kernel does not add new System Calls to create RT-processes. On the other hand a NRT-process is converted into a RT-process using the *mrt_set2rt()* System Call. Before converting a process, several parameters (as priority, period, watchdog, etc.) must be passed to the RT-kernel using the *mrt_setproc()* System Call.

A RT-process must be converted back into a NRT-process (explained in Section 3.8) ,using the *mrt_set2nrt()* System Call, before it can be terminated. When the system runs in NRT-mode, any process that calls *mrt_set2rt()* trying to convert itself into a RT-process, receives an error return code.

As there are two kernels with a shared set of processes with their own process states and transitions each, the bit named *MRT_P_REALTIME* in the process descriptor status flags (*proc[].p_flags*) is set for RT-processes.

The MINIX *ready()* function enqueues a process descriptor into one of the MINIX Ready queues only if all bits of *proc[].p_flags* are cleared (0x00). With the *MRT_P_REALTIME* bit set, a RT-process can not be in the *READY* state, therefore it will be ineligible for the MINIX scheduler.

As is described by Tanenbaum [2], a MINIX process have 3 basic states (see Figure 3.1):

– *READY:* The process is ready to run and waiting to be selected by the MINIX process scheduler.

– *BLOCKED:* The process is blocked because it has done a MINIX System Call using *sendrec()* kernel function.

– *RUNNING:* The process is running under the MINIX kernel control.

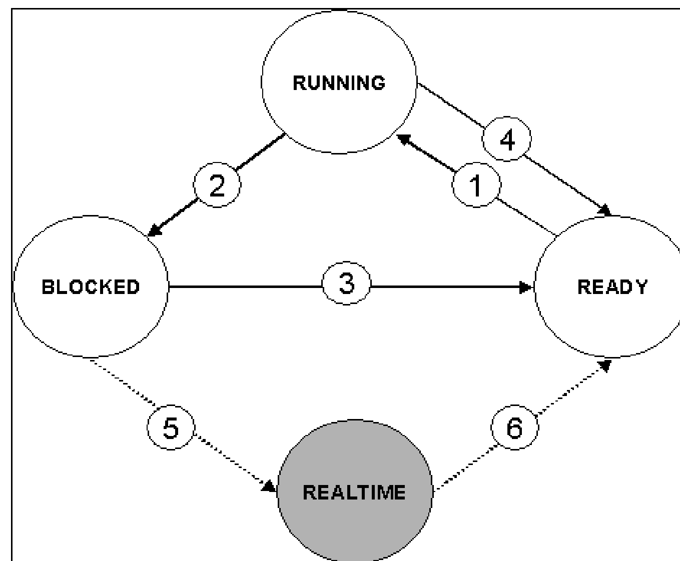


Figure 3.1: NRT-Process States and Transitions.

The following are the process states transitions under MINIX:

1. *READY to RUNNING*: The process has been selected to run by the scheduler.

2. *RUNNING to BLOCKED*: The process has done a blocking System Call.

3. *BLOCKED to READY*: The process has returned from a System Call.

4. *RUNNING to READY*: The running process has run for its entire alloted timeslice or it has been preempted by other process with higher priority.

When MINIX runs under the RT-kernel control, a fourth process state named *REALTIME* is added. This state is reached when the NRT-process is converted into a RT-process. The *ready()* MINIX function is inhibit of inserting the process into the Ready queue, therefore it will be ineligible for the MINIX sheduler.

Consecuently, the following NRT-process states transitions are the added under MINIX4RT:

5. *BLOCKED to REALTIME*: The NRT-process has done a *mrt_set2rt()* System Call converting itself into a RT-process.

6. *REALTIME to READY*: The RT-process has done a *mrt_set2nrt()* System Call converting itself into a NRT-process. This transition also occurs when the RT-process calls *exit()* or when it receives a NRT-signal sent by a NRT-process.

3.3. RT-Process States and Transitions

After a NRT-process is converted into a RT-process, it changes from one state to another. The states recognized by the RT-kernel are (Figure 3.2):

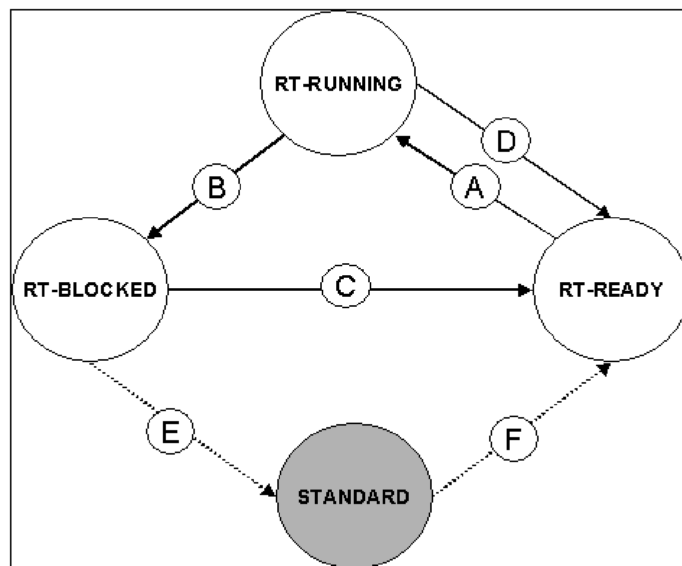


Figure 3.2: RT-Process States and Transitions.

– *RT-READY*: The RT-process is ready to run and waiting to be selected by the RT-process scheduler.

– *RT-BLOCKED*: The RT-process is suspended because it has done a blocking RT-System Call to the RT-kernel.

– *RT-RUNNING*: The RT-process is running under RT-kernel control.

– *STANDARD*: The RT-process has been converted into a NRT-process and must be ignored by the RT-kernel.

The RT-process state transitions are:

A. *RT-READY to RT-RUNNING*: The RT-process has been selected to run by the RT-scheduler.

B. *RT-RUNNING* to *RT-BLOCKED*: The RT-process has done a blocking RT-System Call.

C. *RT-BLOCKED* to *RT-READY*: The process has returned from a RT-System Call.

D. *RT-RUNNING* to *RT-READY*: The running RT-process has been preempted by other RT-process with higher priority.

E. *RT-BLOCKED* to *STANDARD*: The RT-process has done a *mrt_set2nrt()* System Call to convert itself into a NRT-process. This transition also occurs when the RT-process calls *exit()* or when it receives a NRT-signal from another NRT-process.

F. *STANDARD to RT-READY*: The NRT-process has done a *mrt_set2rt()* System Call to convert it into a RT-process.

Really, *STANDARD* and *REALTIME* are compound states. *REALTIME* is the set of MINIX4RT process states and *STANDARD* is the set of MINIX process states as it shows in Figure 3.3.

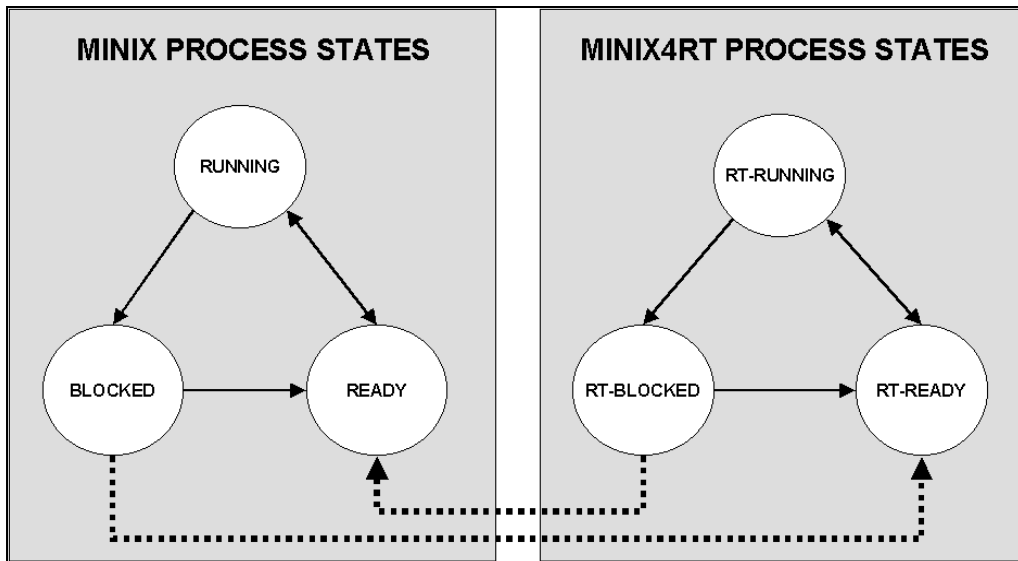


Figure 3.3: RT and NRT Process States and Transitions.

3.4. Process Descriptor Real-Time fields

MINIX uses a process descriptor table to keep the description and status information of every process in the system. Each process descriptor has a field named *p_flags* to indicate the reason why a process is blocked. If *p_flags = 0*, the process can be scheduled by the MINIX process scheduler.

New fields were added to the process data structure for RT-process management and statistics collection (see Figure 3.4). The data structure of a RT-process is presented in Appendix D.

Non Real-Time Fields
rt
st
*pmq
*pvt
getfrom
sendto
*pmsg
*pmhdr
*pnextrdy
*pprevrdy

flags
priority
baseprty
period
limit
deadline
watchdog

scheds
mdl
timestamp
msgsent
msgrcvd

Figure 3.4: Process Descriptor Real-Time Fields.

The RT-process attributes fields are (as part of the *rt* data structure):

- *flags*: To keep the RT-process status flags. It has a bit named *MRT_P_REALTIME* to distinguish among RT and NRT-processes. The RT-kernel accepts RT-System Calls from processes with this bit set (with the exception of *mrt_set2rt()* System Call). It has another bit named *MRT_P_PERIODIC* to distinguish among *periodic* and *non-periodic* RT-processes. A *periodic* RT-process performs computation at a regular time interval (period).

- *priority*: The effective scheduling priority used by the RT-scheduler to select the next RT-process to run. It usage is explained in Section 3.6.

- *baseprty*: The priority assigned to the process when it is converted to RT. It is used by the Basic Priority Inheritance Protocol (BPIP) to restore the efective priority (explained in Chapter 5).

- *period*: The scheduling period of a RT-periodic process. It is specified in RT-timer ticks (explained in Chapter 4).

- *limit*: A limit for the number of RT-schedulings for the process (explained in Section 3.5).

- *deadline*: The RT-process deadline. It is specified in RT-timer ticks.

- *watchdog*: The RT-PID (RT process ID defined by *mrtpid_t* data type) of a RT-process that provides services to protect the RT-process against deadline expiration. The watchdog process can be programmed to perform several actions on the occurrence of a RT-process overrun. When a RT-process does not complete its work before its deadline expiration, the RT-kernel sends a *MT_DEADLINE* message to the watchdog RT-process specified in the process descriptor.

Other RT-process resource management fields are:

- *pmq*: A pointer to a message queue assigned to the RT-process (detailled in Chapter 5).

- *pvt*: A pointer to a VT assigned to the RT-process (only for *periodic* processes).

– *getfrom*: The RT-PID of a RT-process from which the process is waiting to receive a RT-message.

– *sendto*: The RT-PID of a RT-process to which the process is waiting to send a synchronous message.

– *pmsg*: A pointer to the message received.

– *pmhdr*: A pointer to the header of the message received.

– *pnextrdy*: A pointer to the next ready RT-process descriptor in the RT-ready queue (explained in Section 3.7).

– *pprevrdy*: A pointer to the previous ready process descriptor in the RT-ready queue (explained in Section 3.7).

The RT-process statistical fields (*st* in Figure 3.4) are explained in Chapter 7.

3.5. The RT-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 OS component that divides, using a defined policy, the finite resource of CPU time between the runnable processes on a system.

The set of rules used to determine when and how to select which process to run next is called scheduling policy[15]. A scheduler's policy often determines the overall feel of a system and is responsible for optimally utilizing CPU time. The policy behind a RT-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." [27].

The first and most common scheduling method in RTOS is preemptive priority-based scheduling, where a lower priority process is preempted by a higher priority process when it becomes ready to run. The RT-scheduler always selects the highest priority runnable RT-process for execution. All unvoluntary context switches are triggered by interrupts. Timer interrupts can cause preemption due to Timer-Driven RT-process activation. If the priority of the activated RT-process is higher than the priority of the currently running process, the execution of current is interrupted and the RT-scheduler is invoked to select another RT-process to run.

The MINIX scheduler is implemented in the *pick_proc()* kernel function. MINIX4RT modifies its code calling the RT-scheduler (*MRT_pick_proc()* function) at first. As consecuence, when the MINIX scheduler is invoked, the RT-schedulers runs first. *MRT_pick_proc()* tries to find the *RT-READY* process with the highest priority, returning the pointer to the RT-process descriptor. If there are not such RT-process, the *pick_proc()* stills running its original code trying to find the highest priority NRT-process.

MRT_pick_proc() also updates system and RT-process scheduling statistics and controls that the number of schedulings of a RT-process does not reach the specified *limit*. Once the *limit* is reached, the RT-process is removed from its RT-READY queue. Also, the bit into the *flag* field named *MRT_STOP* is set to avoid that the process could run allowing gathering of RT-process statistical information.

The RT-scheduler uses an optimized process-selection algorithm, based on a set of ready queues and a bitmap [28]. Each bit in the bitmap represents a RT-READY queue. If a bit is set, it means that at least one process is RT-READY in that queue. Typically, the bitmap 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 RT-READY processes.

In MINIX (and other time-sharing OSs), the timeslice is the numeric value that represents how long a process can run until it is preempted. MINIX4RT does not use a timeslice for preempt a RT-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 RT-kernel calls the RT-scheduler to find the highest priority RT-READY process to execute (presumably the process that just became runnable).

3.6. 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).

3.6.1. NRT-Process Priorities

MINIX uses three Ready queues to schedule processes as it is shown in Figure 3.5:

– *TASK_Q*: Assigned for I/O Tasks.

– *SERVER_Q*: Assigned for Servers like Memory Manager (*MM*) and File System Manager (*FS*).

– *USER_Q*: Assigned for User-level processes.

The MINIX scheduling algorithm is simple. It looks up for a process into the READY queues starting with the *TASK_Q*, next continues with the *SERVER_Q*, and finally with the *USER_Q*. The scheduler selects the first process it finds in the non-empty highest priority queue. If all queues are empty, the *IDLE* process (detailled in Chapter 7) is scheduled.

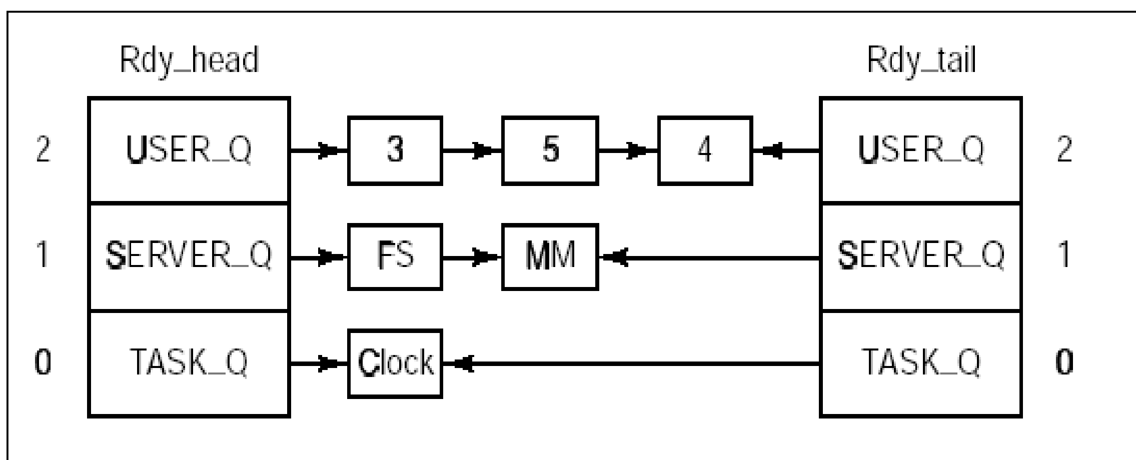


Figure 3.5: MINIX READY Queues (from [2]).

3.6.2. RT-Process Priorities

Each RT-process has a base priority (the *baseprty* field) and an effective priority (the *priority* field), and is scheduled in accordance with the latter. The base priority of a RT-process is stablished using the *mrt_setproc()* System Call before calling *mrt_set2rt()* System Call that convert the NRT-process into a RT-process. Both, the baseprty and the *priority* field can have a value ranging from *MRT_PRIHIGHEST* (0x00) to *MRT_PRILOWEST* (0x0F).

Normally, the effective priority of a RT-process equals its base priority but, it may be changed by the BPIP (more details in Chapter 5) or the scheduling policy.

The set of RT-ready queues is actually implemented as 16 (*MRT_NR_PRTY)* separate queues, assigning one queue for each priority. A ready RT-process could be inserted into the RT-queue corresponding to its effective *priority* field in FIFO or LIFO order. The first process descriptor in the highest RT-ready queue will be selected to run by the RT-scheduler.

The *priority* field is also used by the RT-kernel to minimize the Interrupt Blocking time. Only those RT-interrupt handlers with higher priorities are executed while the current RT-process is running.

3.7. RT-Ready Queues Management

To manage RT-ready queues the RT-kernel have the following data structures:

A set of RT-ready queues, one queue for each priority level (*RTM_sp.rdyQ.procL[]*).

A bitmap that have one bit assigned for each priority. Initially, all the bits are cleared indicating that all queues are empty (*RTM_sp.rdyQ.bitmap*).

When a RT-process becomes runnable (that is, its state becomes *RT-READY*), the corresponding bit to the process *priority* is set in *RTM_sp.rdyQ.bitmap*, and the process descriptor is appended to the RT-ready queue in accordance with its *priority* field.

Finding the highest priority RT-process on the system is therefore only a matter of finding the first bit set in *RTM_sp.rdyQ.bitmap*. Because the number of priorities is fixed, the time to complete a search is constant and unaffected by the number of running processes on the system.

Each ready queue descriptor (*MRT_procL_t*) have one pointer to the *first* process descriptor and other pointer to the *last* process descriptor in the queue (see Figure 3.6).

A process descriptor can be inserted into a queue in FIFO or LIFO order. Processes of the same priority will be managed under a FIFO policy, but sometimes a process that inherits its priority by the BPIP must be inserted into a queue in LIFO order (explained in Chapter 5).

Each queue descriptor also contains a field named *inQ* that counts the current number of runnable RT-processes in the queue and a field named *maxinQ* that keeps the highest value of the *inQ* field for statistics.

The following kernel functions help to manage RT-ready queues:

– *MRT_rdyQ_app()*: Appends a process descriptor at the tail of a RT-ready queue.

– *MRT_rdyQ_ins()*: Inserts a process descriptor at the head of a RT-ready queue.

– *MRT_rdyQ_rmv()*: Removes a process descriptor from a RT-ready queue.

Several kernel functions operate on RT-ready queues:

– *MRT_pick_proc()*: The RT-scheduler searchs for the highest priority ready-to-run RT-process.

– *MRT_inherit()*: RT-IPC could change the priority of the destination process of a message. This function removes a process descriptor from its current RT-ready queue, inserting it into its inherited RT-ready queue using *MRT_rdyQ_rmv()* and *MRT_rdyQ_ins()* RT-kernel functions.

– *MRT_disinherit()*: Used to change the priority of a RT-process on *mrt_reply()* System Call. This function removes the processs descriptor from the its current RT-ready queue (*MRT_rdyQ_rmv()*), and inserts the descriptor into its inherited or base priority RT-ready queue in accordance to the BPIP (*MRT_rdyQ_ins()*).

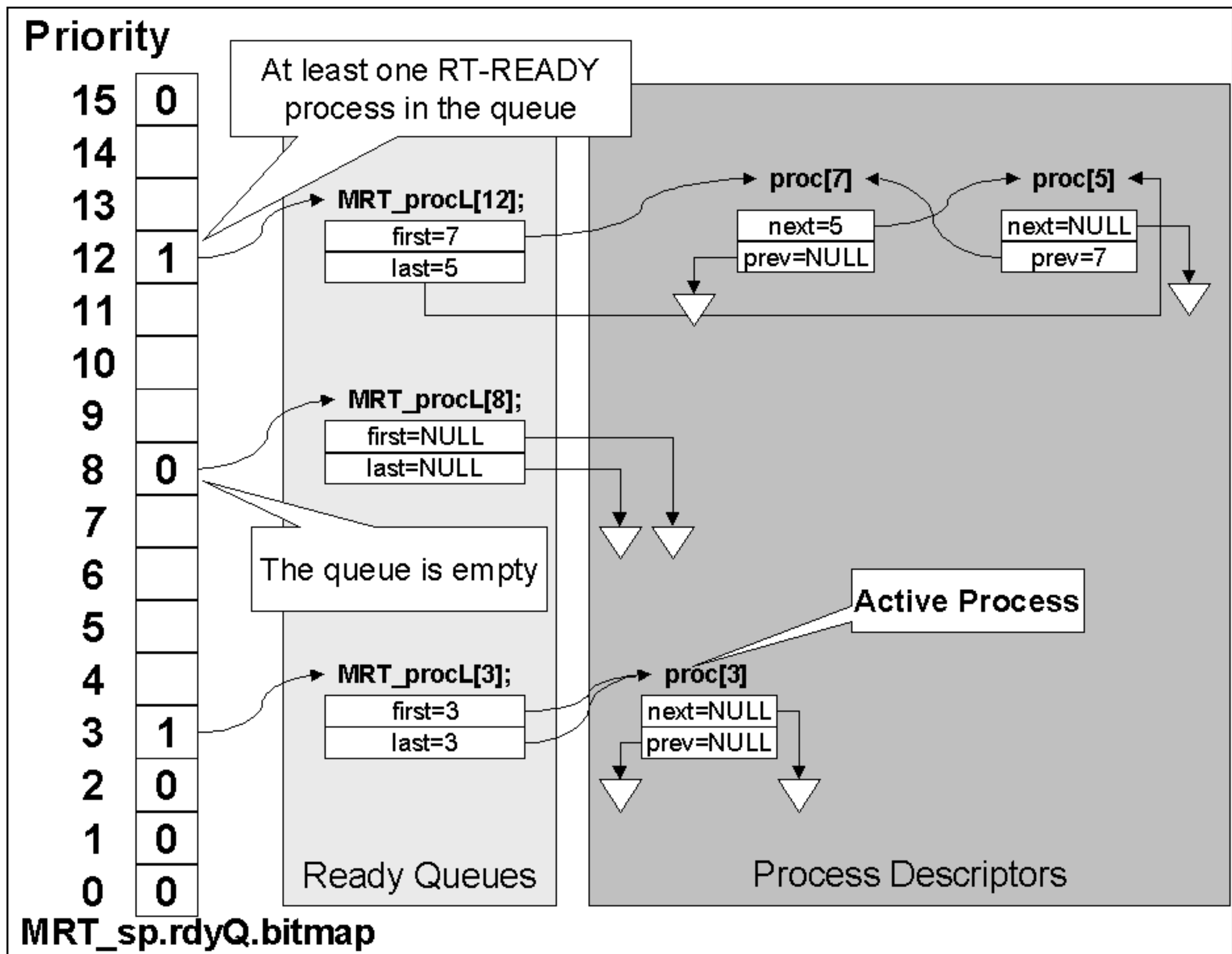


Figure 3.6: RT-kernel Priority Queues.

As it can be seen, there are similarities between interrupt descriptors queues and process descriptors RT-ready queues, but there are considerable differences in the queue usage. As it was described in Chapter 2, once an interrupt descriptor is set, it is inserted into an interrupt descriptor queue. The queue keeps the descriptor even though it has been triggered and its has been serviced. The interrupt descriptor can only be removed by the programmer using the *MRT_free_irqd()* RT-kernel function. RT-ready queues keep only process descriptors in *RT-READY* and *RT-RUNNING* states. Process descriptors in other states are removed. As the BPIP could change the process priority, a descriptor could be removed from one RT-ready queue and inserted into other.

MINIX4RT does not have a policy to assign process priorities, but it is easy to create a System Call that set a Rate Monotonic[29] priority schema using the *period* field of process descriptors or a Deadline Monotonic [30] using the *deadline* field of process descriptors. Dynamic priority algorithms as Earliest Deadline First (EDF) [31] or Least Laxity First (LLF) [32] requires more complex changes.

3.8. RT-process Termination

MINIX has four ways to terminate a running process:

– *Normal Exit*: Invoking the *exit()* System Call by the process itself (voluntary).

– *Error Exit*: The process discovers a fatal error and invokes the *exit()* System Call (voluntary).

– *Fatal Error*: The OS discovers a process fatal error (often a program bug) and terminates the process using the *do_kill()* kernel function (unvoluntary).

– *Killed*: Other process sends an uncached signal to the process (unvoluntary).

When MINIX4RT runs in RT-mode, a RT-process must be converted into a NRT-process before it can be terminated.

3.8.1. RT-process Termination Using the *exit()* System Call

As it will be explained in Chapter 5, RT-processes can not send/receive messages using standard *send()*/*receive()* MINIX primitives. If a RT-process calls *send()* or *receive()* funcions, it receives an error code on function return.

The *exit()* System Call uses *sendrec(),*a single function for doing a *send()* followed by a *receive()*. If a RT-process invokes *exit()*, it should return with an error, but the *exit()* System Call must not return to the calling process. To overcome this issue, two functions were modified in MINIX4RT as it described below:

– The *_exit()* Library function invokes *mrt_set2nrt()* before doing its standard work. The RT-kernel converts the RT-process into a NRT-process, and releases the resources owned by the terminanting process (VTs and a message queue (MQ) (explained in Chapter 5)).

– The *exit()* function of the *MM* Server that requests the *MRTTASK* to release the VTs and the MQ owned by the RT-process.

3.8.2. RT-process Termination Using the *signal()* System Call

If a NRT-process sends a signal to a RT-process, the target is converted into a NRT-process before it can receive the signal. That conversion is accomplished by the modified *sig_proc()* function of the *MM*. The *MM* sends a *MRT_STDSIGNAL* message to *MRTTASK* returning without any action if the target is a NRT-process or converting the target into a NRT-process before return.

If the signal sent is uncached by the process, the process will exit releasing RT-resources before.

3.8.3. Releasing RT-process Resourses and Housecleaning

To keep the system concistency, the RT-kernel (*MRT_proc_stop()* function) carry out some housecleaning tasks during the the conversion of a RT-process into a NRT one. It releases RT-resources owned by the converted process as is described below:

- Any RT-process with pending requests to the converted process will return with *E_MRT_EXITING* error code.

- A *MT_SIGNAL* message is sent to the watchdog RT-process of the converted process.

- A *MT_SIGNAL* message is sent to all RT-process with their *watchdog* field equals to the converted process. The *watchdog* fields of that RT-processes are reset.

- All *watchdog* fields of RT-interrupt descriptors that equals to the converted process are reset.

- All pending asynchronous messages sent by the converted process to other RT-processes are removed from their MQs.

- The VT owned by the process related to a converted periodic process is stopped (but not released).

- The VT owned by the process related to IPC timeout management is stopped (but not released).

4. TIME MANAGEMENT

4.1. Timing Mechanisms

Timers are mechanisms that are able to notify the kernel or user programs that a certain interval of time has elapsed [15], they play an integral role in RT-systems [33, 34].

RT-applications must be able to operate on data within strict timing constraints in order to schedule application or system events. Timing requirements can be in response to the need for either high system throughput or fast response time. Applications requiring high throughput may process large amounts of data and use a continuous stream of data points equally spaced in time.

The following types of timing mechanisms are often used:

– *A pause() function*: A function *pause()* is used to suspend the active process for a specified time. As is explained in [33], an inaccuracy could occur because the pause function use the Timer interrupt as its 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 timer, the feature may be reinitiated or recovery action might be taken.

– *Sequencing Operations*: Timers are used for sequencing time based state transitions.

– *Polling*: A timer is kept and the system polls for a condition on every timeout.

– *Periodic Operations*: For implementing audits, periodic timers are kept. On each timer expiry, software audit is initiated.

– *Failure Detection*: For monitoring the health of other modules, a module runs a timer. It expects a sanity message periodically from all the other modules before the expire of the timer. If certain number of sanity messages are missed in succession from a module, module failure is declared as failed.

– *Inactivity Detection*: Timers are also used for detecting the inactivity in a session.

4.2. MINIX4RT Timer Interrupts

MINIX use three software components to handle time related tasks.

– *The Clock ISR*: increments de real time counter, decrements the quantum of the running process and checks it for zero, makes CPU accounting, and decrements an alarm counter.

– *The Clock Task*: It is a Task that is scheduled when a Timer interrupt occurs and there is work to do, such as when an alarm must be sent or a process has run too long.

– *The Synchronous Alarm Task*: It is a Task to send messages (synchronous events) directly to the server that requested the synchronous alarm, which must be waiting for the message. Synchronous alarms can only be requested by servers, for example, the network server wanting to time out if an acknowledgement packet does not arrive in a certain amount of time.

MINIX4RT need to enhance Timer operations accuracy, resolution and predictability needed by a hard Real-Time OS, therefore it does not use any of the described components because they implies several context switches. This approach also facilitates software updates when new versions of MINIX will be realeased because the added source code is less intrusive.

Several time-keeping activities are triggered by interrupts raised by the Programmable Interval Timer (PIT) on IRQ line 0. Some of these activities need to be executed as soon as possible after the interrupt is raised, while the other are delayed out of interrupt time [15].

The RT-kernel carry on the following actions related to time:

- Updates the tick count since the RT-mode startup.
- Checks whether the interval of time associated with each timer has elapsed.
- Executes actions related to expired timers.
- Emulates a Timer interrupt for the MINIX kernel.
- Keeps updated an ordered timer list.
- Inserts and removes timers from free and expired lists.

The following RT-kernel functions deal with important Timer interrupts related activities:

- *MRT_irq_dispatch()*: As was explained in Chapter 2, this function is executed on each hardware interrupt. On each Timer interrupt it takes the following actions:
  - It updates the counter *MRT_sv.counter.ticks* (explained in Section 4.4).
  - It copies the *MRT_sv.counter.ticks* in the Timer interrupt descriptor *timestamp* field.
  - It checks for VT expirations.
  - If the Active VT queue is empty or any VT has expired, it reenables the Timer IRQ line before returns.

- If at least one VT has expired, it triggers the Timer interrupt descriptor (its handler is *MRT_clock_BH()*) for delayed processing, and reenables the Timer IRQ before returns.

– *MRT_clock_BH()*: It is the Timer interrupt (Clock in MINIX terminology) Botton Half. It is executed only if a VT has expired *outside* interrupt time called by *MRT_flush_int()* and it has the *PRI_HIGHEST* priority. Its purpose is to change expired VTs from the Active Queue to the Expired Queue (explained in Section 4.8.2).

– *MRT_vtimer_flush()*: This function, also called by *MRT_flush_int(),* executes Virtual Timer asociated actions (explained in Section 4.8.3).

It is important to note that the Timer ISR merely indicates that at least a VT has expired. Before the RT-kernel returns from interrupt, the Active VT queue will be checked for expired VTs and the their actions will be executed.

4.3. MINIX Virtual Timer Interrupts

One important use of VTs is the emulation of Timer interrupts for the MINIX kernel. After executing *mrt_RTstart()* System Call, a VT and a software interrupt are assigned to emulate Timer interrupts for MINIX. The RT-handler field of the software interrupt descriptor points to the original *clock_handler()* function of MINIX kernel.

IBM-compatible PCs include a device called 8253/4 Programmable Interval Timer (PIT). The PIT is programmed by the kernel so that they issue interrupts at a fixed, predefined frequency. These periodic interrupts are colled Timer ticks and are crucial for implementing the timers used by the kernel and user´s programs. This device issues a special interrupt on IRQ0 called Timer interrupt, which notifies the kernel that a time interval has elapsed. Modern CPUs have timers in the local APICs.

As the frequency of Timer interrupts of MINIX4RT could be greater than in MINIX, the RT-kernel emulates MINIX Timer interrupts invoking the standard timer handler (the *clock_handler()* function) at lower or equal frequency than the PIT.

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) */
```

MINIX4RT redefines *HZ* as follow to make more easy the use of integer decimal periods:

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

The Timer interrupt rate of MINIX4RT is established by the *MRT_sv.tickrate* system global variable. *HZ* and *MRT_sv.tickrate* must be harmonic frequencies to virtualize Timer interrupts for the MINIX kernel only when Timer interrupts occur, therefore they are related by an integer value:

```
MRT_sv.tickrate = MRT_sv.harmonic * HZ;    /* MRT_sv.harmonic = 1,2,3,.....N */
```

To preserve the illusion of the standard MINIX tick rate *HZ*, the MINIX interrupt handler *clock_handler()* is called after *MRT_sv.harmonic* timer interrupts. This interrupt rate is emulated using a VT with *period = MRT_sv.harmonic* and *MRT_ACT_IRQTRIG* action type (See Figure 4.1).

A software interrupt related to the VT is used to defer the processing of the MINIX Timer interrupt handler with a *MRT_PRILOWEST* priority. On each VT expiration, the software interrupt is triggered and its handler is executed by *RTM_flush_int()* function.

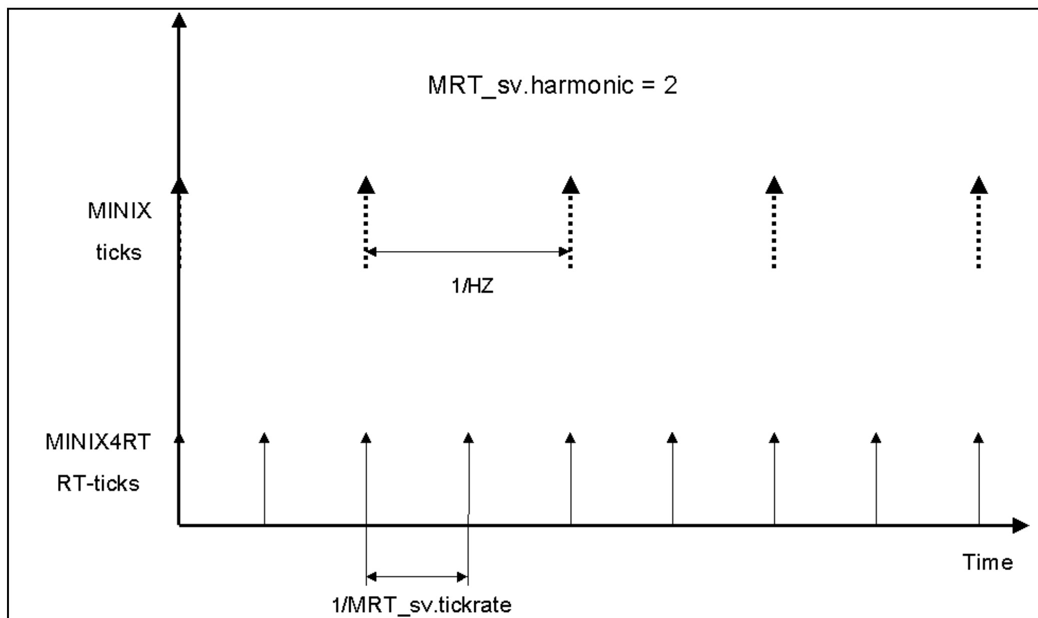


Figure 4.1: MINIX Virtual Timer Interrupt.

4.4. Timer Resolution

The 8253/4 PIT uses an internal oscillator frequency (*TIMER_FREQ)* at 1,193,180 Hz. It has a 16 bits *LATCH* register to set the ratio between the oscillator frequency and the number of interrupts per second (*tick_rate = 1193180/LATCH*).

Only some values of *LATCH* issues integer values of *tick_rate* as it is show in Table 4.1. Therefore, other values of *LATCH* result in accuracy error.

Table 4.1: Integer Values of *tick_rate.*

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

The value of the PIT *LATCH* is stored in a system global variable named *MRT_sv.PIT_latch* and it is initilized as:

MRT_sv.PIT_latch = TIMER_FREQ/MRT_sv.tickrate.

An example illustrates the accuracy error. For a tick rate of 100 [interrupts/s]:

MRT_sv.PIT_latch = 1193180/100 =11931 and a a reminder of 80 Hz

The reminder represents an additional Timer interrupt every (11931*100/80) = 15000 Timer ticks or 150[s]. The RT-Timer interrupt frequency is 100.0067052217 [Hz] and the RT-Timer period is 0.009999329522788 [s] resulting in an error in time accuracy of 0.0000670478. Table 4.2 shows some RT-interrupt frequencies, RT-Timer periods and resulting errors.

As *MRT_sv.tickrate* is a multiple of *HZ*, it could be changed setting *MRT_sv.harmonic* using *mrt_restart()*. This System Call can only be used before running any RT-process or when there are not any RT-process running, otherwise all system time reference (in Timer ticks units) would be erroneous (i.e. periods, timestamps, tick counters, etc.).

MINIX kernel keeps tracks of the number of elapsed (virtual) Timer ticks since the system was started in the global system variable named *realtime*. It is set to 0 during kernel initialization.

MINIX4RT kernel keeps tracks of elapsed (real) Timer ticks since the last *mrt_RTstart()* or *mrt_restart()* System Call invocation in the global system variable named *MRT_sv.counter.ticks*. It is set to 0 during RT-system initialization and incremented by one unit when a Timer interrupt occurs, that is on every Timer tick. Since *MRT_sv.counter.ticks* is an unsigned 32 bits integer, the time that makes it overflow depends on the tick rate. However, the RT-kernel handles the overflow using another kernel variable named *MRT_sv.counter.highticks*.

Note that since the system updates *MRT_sv.counter.ticks* and *MRT_sv.counter.highticks*, only the former is loaded in the *timestamp* field of descriptors (interrupt descriptors, process descriptors, message descriptors, etc). The next versions of MINIX4RT will include the extension of the *timestamp* field.

Table 4.2: Period Errors.

Configured TickRate [int/s]	Configured 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

Table 4.3 shows some wrap around time (in days) of *MRT_sv.counter.ticks* for some Timer interrupt frequencies.

Table 4.3: MRT_sv.counter.ticks Overflow Time.

MRT_sv.tickrate [interrupt/s]	MRT_sv.counter.ticks Days to Overflow
50	994
100	497
200	249
500	99
1000	50
1500	33
2000	25
3000	17
4000	12
5000	10
7500	7
10000	5

4.5. 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.

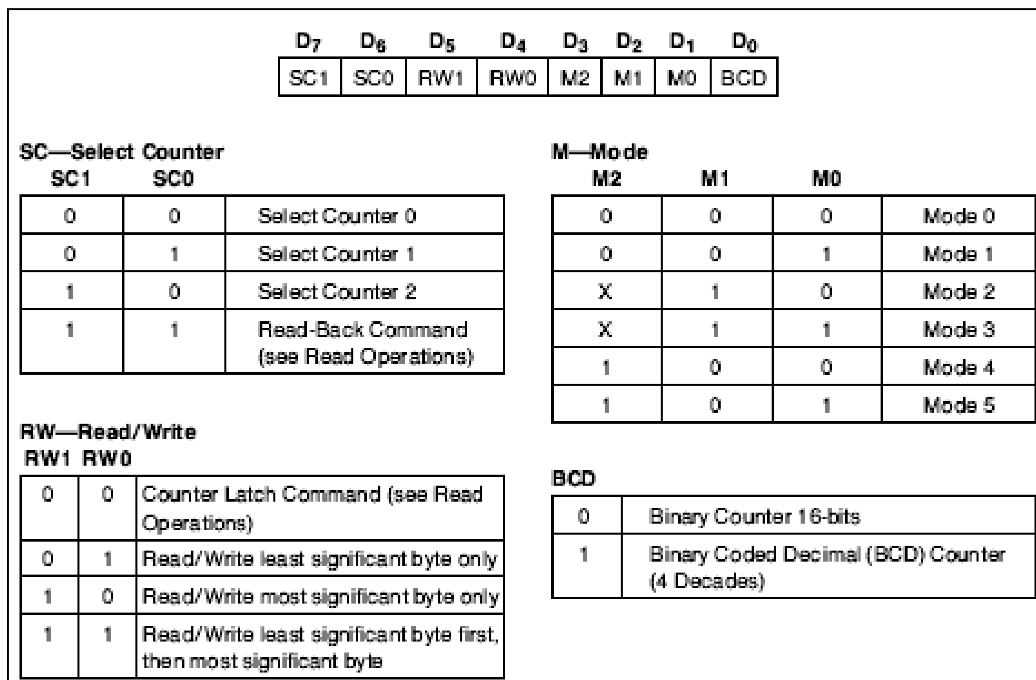
– *Mode 2*: Rate Generator.

– *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 format shown in Figure 4.2:



D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
SC1	SC0	RW1	RW0	M2	M1	M0	BCD

SC—Select Counter

SC1	SC0	
0	0	Select Counter 0
0	1	Select Counter 1
1	0	Select Counter 2
1	1	Read-Back Command (see Read Operations)

RW—Read/Write

RW1	RW0	
0	0	Counter Latch Command (see Read Operations)
0	1	Read/Write least significant byte only
1	0	Read/Write most significant byte only
1	1	Read/Write least significant byte first, then most significant byte

M—Mode

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

BCD

0	Binary Counter 16-bits
1	Binary Coded Decimal (BCD) Counter (4 Decades)

Figure 4.2: PIT Control Word Format.

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 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 RT-kernel functions operates on the PIT Channel 0:

– *MRT_set_timer()*: Sets the Timer interrupt rate with a frequency of (*HZ *harmonic)*. The function argument is the *harmonic* frequency of the MINIX Timer frequency.

– *MRT_read_timer()*: Reads the current value of the PIT counter.

4.6. Estimating the Timer Interrupt Latency

When the PIT counter reachs zero, the PIT raise IRQ 0 and resets the counter to the value in the LATCH. As the PIT remains decrementing the counter, during the execution of the Timer interrupt handler, the value of the counter lets compute the latency of the handler (See Figure 4.3).

Timer_Handler_Latency = MRT_sv.PIT_latch - MRT_read_timer()

To consider the overhead of executing *MRT_read_timer()*, the kernel estimates its value at initialization time and stores the result (in PIT Hz units) into the system variable *MRT_sv.PIT_latency*.

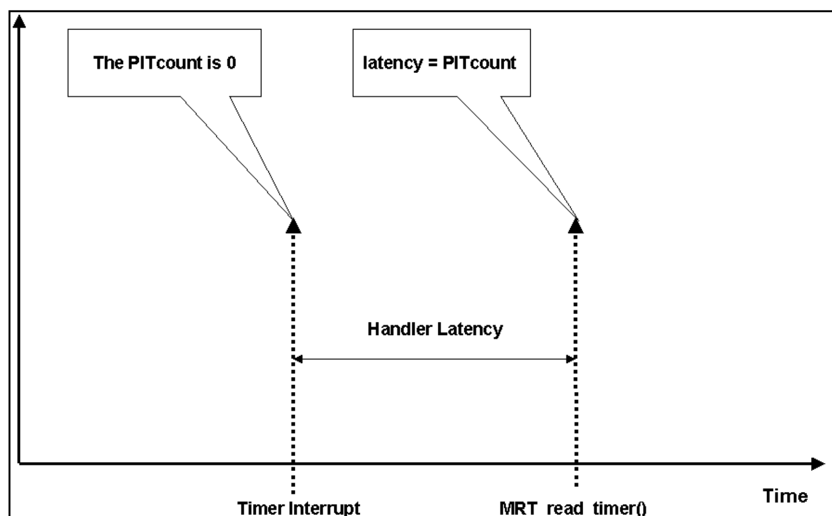


Figure 4.3: Estimating Timer Interrupt Latency.

Applications can get the estimated Timer Interrupt Latency using the *mrt_getistat()* System Call for IRQ 0, and the value of *MRT_sv.PIT_latency* using *mrt_getsval()* System Call (more details in Appendix A).

4.7. Real-Time and Non Real-Time Timer Handlers

At startup, the system is not ready for RT-processing (described in Chapter 3). All interrupt handlers used by the kernel are MINIX NRT-handlers, including the Timer handler that is the function *clock_handler()*. When the RT-mode starts, the system enables the use of RT-handlers for those interrupt descriptors defined as Real-Time (the bit *MRT_RTIRQ* set in the *flag* field), including the RT-Timer handler named *MRT_clock_BH()*.

Sometimes, it is necesary to reconfigure the system to change some parameters without recompilation and restarting it. The *mrt_restart()* System Call can be used to change RT-processing parameters and reset system statistics. One of this parameters is the Timer interrupt frequency, and it could be changed using the *harmonic* argument of *mrt_restart()*. All statistical fields of data structures are reset because changing the Timer period causes that they will be erroneous as it was explained in Section 4.4.

4.8. Virtual Timers

A Virtual Timer (VT) is a software facility that allows to take an action at some future moment, after a given time interval has elapsed; a time-out denotes a moment at which the time interval associated with a timer has elapsed [15].

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

MINIX4RT uses VTs to manage periodic processing and other related activities. It has *NR_VTIMERS* Virtual Timers descriptors data structures defined in kernel space.

A VT descriptor has the *MRT_vtimer_t* data structure (see Appendix D) with the following fields:

- *period*: The period of the VT in RT-ticks.

– *nextexp*: The number of RT-ticks for the next VT expiration in the queue (explained in Section 4.8.2).

– *limit*: The number of expirations until free the VT. A limit of 0 means that the VT has no expiration limit.

– *action*: The code of the action to execute on expiration (explained in Section 4.8.1).

– *param*: A generic integer used as a parameter for the action on VT expirations. The *param* field enables to define a single general-purpose function that handles the time-outs of several device drivers; the *param* field could store the device ID or other meaningful data that could be used by the function to differentiate the device.

– *index*: The VT identification number.

– *owner*: The VT RT-process owner.

– *priority*: The VT owner's priority. This field is used on VT expirations to run actions in priority order.

– *timestamp*: The last expiration timestamp.

– *expired*: Counts the number of VT expirations.

– *next*: A pointer to the next VT in the queue.

– *prev*: A pointer to the previous VT in the queue.

4.8.1. Virtual Timers Handling Functions

The RT-kernel functions that handle VTs are:

– *MRT_vtimer_flush():* Searches for expired VTs with equal or greater priorities than the *MRT_sv.prtylvl* and runs their actions as was explained in Chapter 2.

– *MRT_vtimer_alloc():* Allocates a VT with the parameters passed as arguments.

– *MRT_vtimer_free():* Reset all parameters of a VT and inserts it into the VT Free queue.

– *MRT_vtimer_ins():* Inserts an VT into the Active/Expired/Free queue.

– *MRT_vtimer_rmv():* Removes a VT from the Active/Expired/Free queue.

– *MRT_vtimer_search():* Searches for a VT in a queue.

– *MRT_vtimer_run():* Executes a VT action. Once the action has been executed, the VT is reescheduled for the next period (if *expired* >= *limit*) or it is released to the Free VT queue. The VT actions could be:

- *MRT_ACT_NONE:* No action is executed.
- *MRT_ACT_PERIODIC*: Used for periodic processes. This action wakes up the VT owner process.
- *MRT_ACT_MSGOWN:* Sends a *MT_TIMEOUT* message to the VT owner process.
- *MRT_ACT_MSGWDOG:* Sends a *MT_TIMEOUT* message to the VT owner's watchdog process.
- *MRT_ACT_IRQTRIG:* Trigger an IRQ descriptor specified in the *param* field.
- *MRT_ACT_SNDTO:* A *send* type timeout has expired (see Chapter 5).
- *MRT_ACT_RCVTO:* A *receive* timeout has expired (see Chapter 5).
- *MRT_ACT_WAKEUP:* This action wakes up the VT owner process if it has called the *mrt_sleep()* System Call that put the process into the *RT-BLOCKED* state.
- *MRT_ACT_SCHED:* This action wakes up the process specified in the *param* field.
- *MRT_ACT_DEBUG:* Used for debugging purposes. Its prints the *param* field in console.

4.8.2. Virtual Timers Queues

The RT-kernel uses several queues for handling VTs. The queues are:

- *MRT_st.timerQ:* It is the queue for Active VTs ordered by expiration time (see Figure 4.4). The *firstexp* field of the queue counts the number of RT-ticks for the expiration of the first VT. Each VT has a field named *nextexp* that counts the number of RT-timer ticks for the expiration of the next VT in the queue. Once a VT has expired, it is removed from this queue.

- *MRT_st.freeQ*: It is the queue for Free VTs.

- *MRT_st.exp.expiredQ[]*: It is an array of queues of expired VTs with pending actions. There is one queue for each system priority. Once the action of a VT is executed, the VT is removed from this queue (see Figure 4.5).

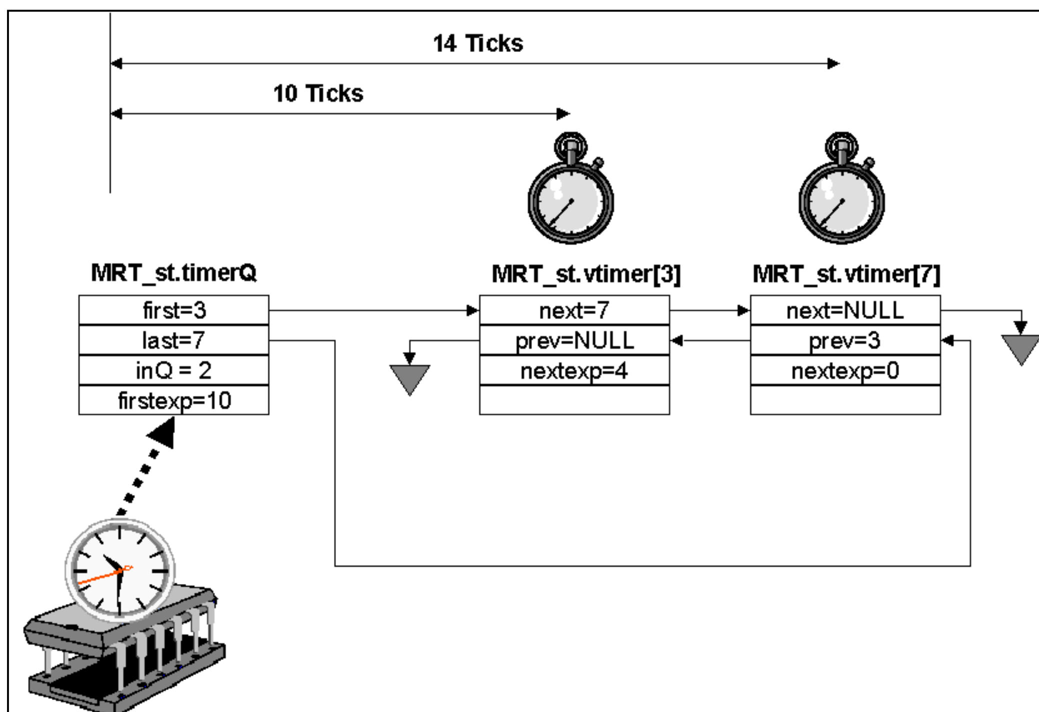


Figure 4.4: Active Virtual Timers Queue.

The data structure of a VT queue descriptors (*MRT_timerQ_t*) has the following fields:

- *first*: A pointer to the first VT in the queue.

- *last*: A pointer to the last VT in the queue.

– *firstexp*: It counts the number of RT-ticks to the first VT expiration.

– *maxper*: It is the total amount of *nextexp* fields of all VTs in the queue. (only used in the Active queue).

– *inQ*: The current number of VTs enqueued.

– *maxinQ:* The maximun number of VTs enqueued.

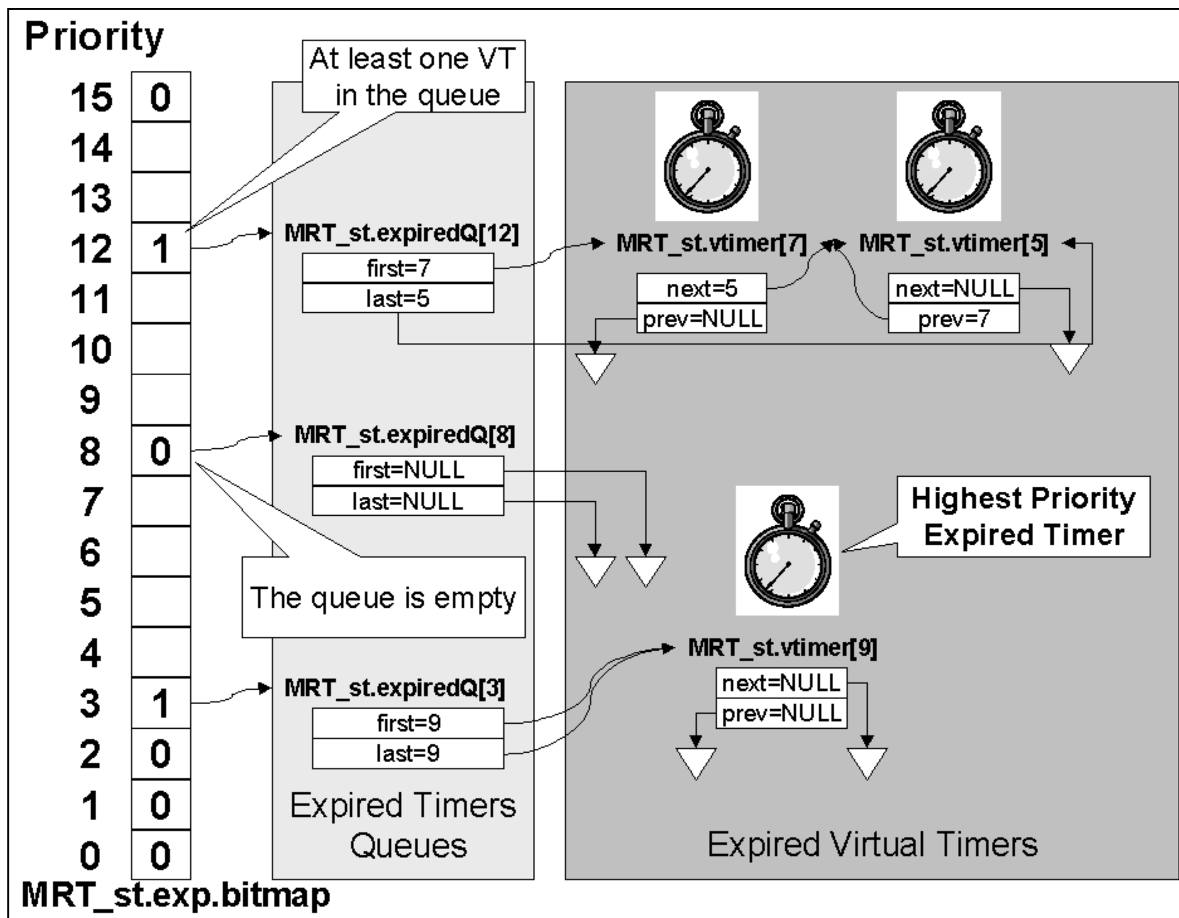


Figure 4.5: Virtual Timer Expired Queues.

4.8.3. Executing Virtual Timers Actions

Executing VT actions in the Timer handler could produce unpredictable latencies on high priority processes. As the Timer interrupt descriptor has the highest priority, all VT actions execute at the highest system priority including those actions of VTs that belongs to processes with lower priorities. This is another type of unbounded priority inversion.

Executing VT actions in *Bottom Halves* routines does not help much, because the VT actions are executed before returning the system to user-mode, therefore the priority inversion persists.

As it is illustrated in Figure 4.6, the execution of actions of all expired VTs preempt the execution of a higher priority process.

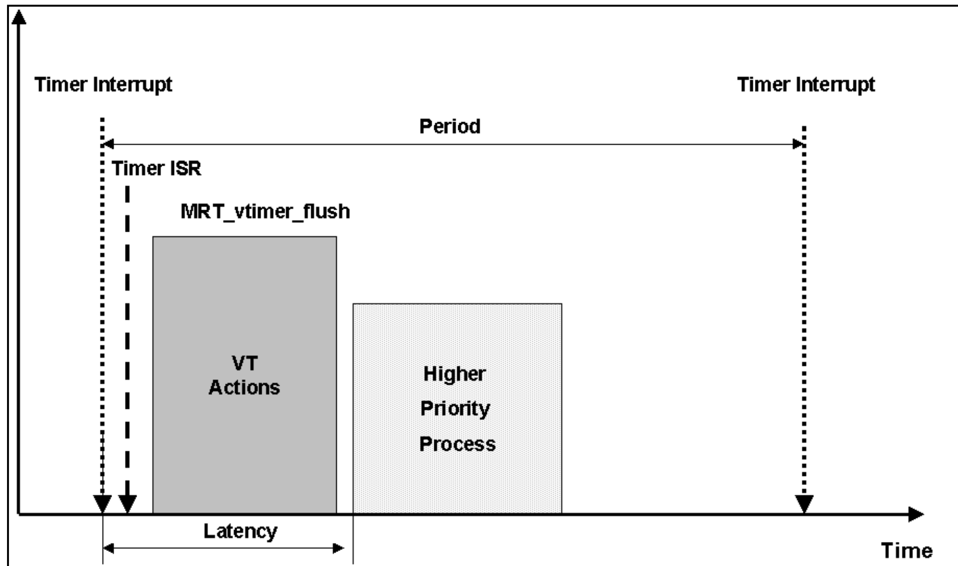


Figure 4.6: VT Actions with Unbounded Priority Inversion.

To avoid this problem, the RT-kernel runs expired VT actions in priority order only if they have higher or equal priorities than *MRT_sv.prtylvl* (see Figure 4.7).

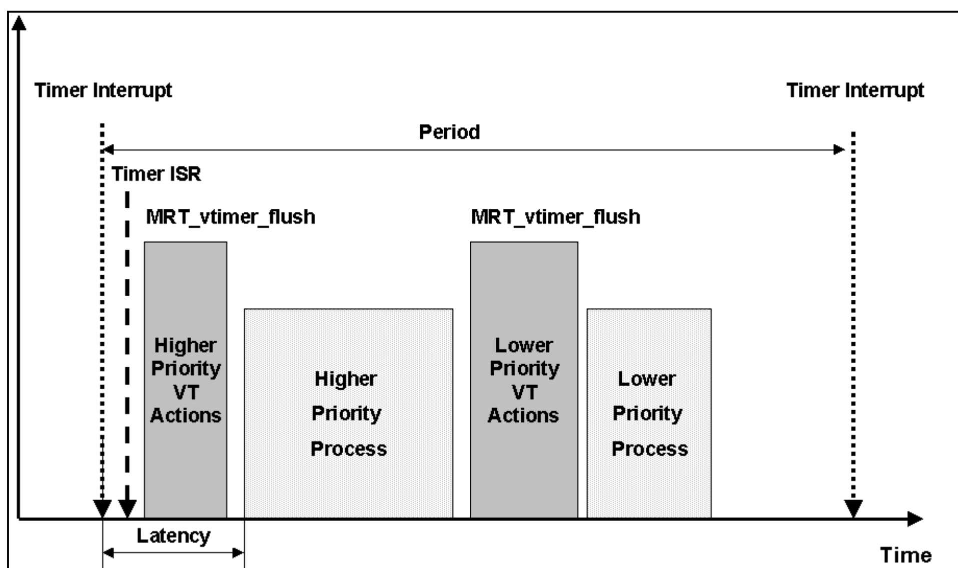


Figure 4.7: VT Actions Priority Ordered Execution.

Only the execution of actions of all expired VTs with higher or equal priority than *MRT_sv.prtylvl* could preempt the execution of a running process or IRQ handler. VT actions with lower priorities will be executed later when the *MRT_sv.prtylvl* decline. The flow diagram is shown in Figure 4.8.

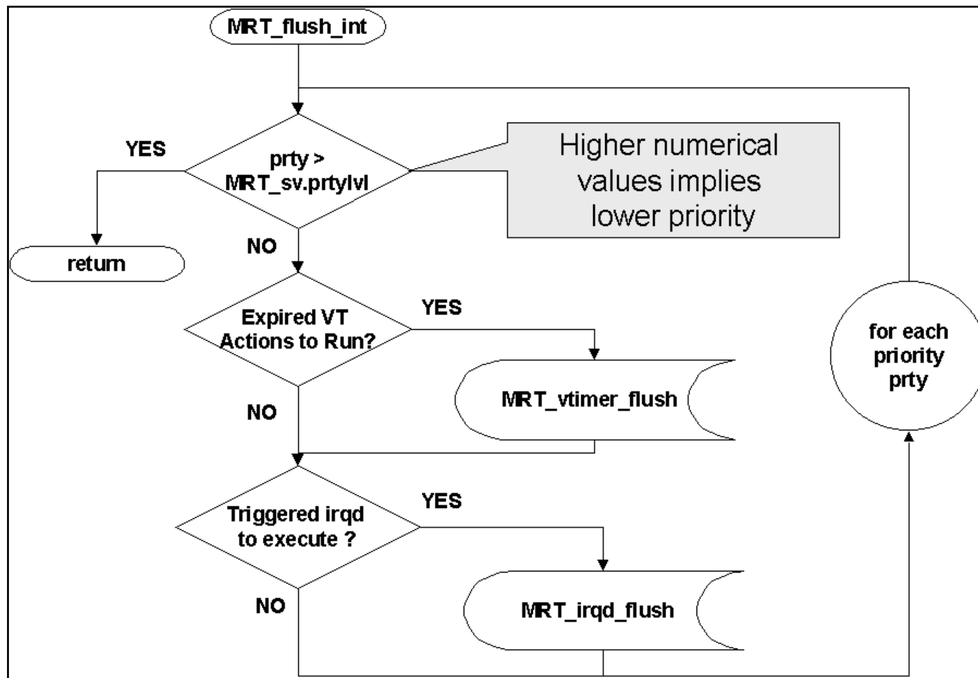


Figure 4.8: Flushing Expired VTs and Triggerd IRQ Descriptors.

4.9. Virtual Timers Handling: Other Tested Approaches

This section dicusses other approaches tested during the development and implementation of VTs on MINIX4RT.

Varghese and Lauck [34] suggest the use of *Timer Wheels* to handle a lots of VTs in an efficient way. This approach has been tested but the coding complexity, larger memory requirements and its minimal performance improvement on a reduced VT environment move further away the educational aims of the MINIX4RT project. In spite of that, that approach could be considered for custom versions or for coding laboratory practice.

When the PIT is used in *SQUARE_WAVE* mode, increasing Timer resolution implies increasing the PIT frequency rising the system overhead because the Timer ISR is executed more frequently.

A better approach [35] to increase the Timer resolution without increasing the PIT frequency is to execute the Timer handler only in those moments that is needed. Instead of producing interrupts at higher rates (*SQUARE_WAVE* mode), the PIT is programmed in *ONE-SHOT* mode. This means that on every Timer interrupt, the PIT is programmed to generate an interrupt as soon as the earliest scheduled VT action. In spite of this feature is not available in the current version of MINIX4RT, it will be included in the next as an optional operation mode.

5. REAL-TIME INTERPROCESS COMUNICATION (RT-IPC)

5.1. Introduction

The key difference between time sharing OS and RTOS is the need for *deterministic* timing behavior in the RTOS. Deterministic timing means that OS services consume only known and expected amounts of time. Inter-process communication (IPC) by message passing is one of the central paradigms of most kernel-based and other Client/Server architectures. It helps to increase modularity, extendibility, security and scalability, and it is the key feature for distributed systems and applications [36]. Therefore, IPC primitives of a RTOS need to have deterministic execution and blocking times.

Messages are units of information that pass from the sender to the receiver. Two methods can be used to message transferences:

- *With buffering*: The message is sent to a data structure like RT-Mach ports[10] or Message Queues that stores it until a process receives it.

- *Without buffering*: The message is sent to a process and the sender must wait to tranfer the message. This method is known as rendezvous.

Tanenbaum selects the rendezvous approach for MINIX. It has the following semantics:

- When a sender calls *send()* but the receiver is not waiting that message, the sender is blocked until the receiver calls *receive()* for that message.

- When a receiver wants to receive a message but it has not been sent, the receiver is blocked until the sender calls *send()* for that message.

Rendezvous approach is fine in a time-sharing environment because it is very rare that two or more messages are queued into a Message Queue and the avarage queue length would be less than one but it can not be used for asynchronous communications among processes.

MINIX's kernel hides interrupts turning them into messages, but interrupts are asynchronous events. When a I/O device raice an interrupt, its handler traps it, and will try to *send()* a message to an I/O Task. If the I/O Task is not waiting for that message, the handler must register this fact and will try to send the message latter because the kernel can not be blocked. This approach does not help to much for the implementation of communications protocols where messages can flow from down to top triggered by interrupts.

In a RT-environment, several messages can be sent to a queue waiting to be received. They must be treated according the senders´ priorities and must guarantee message delivery in a timely fashion[10].

MINIX4RT IPC uses unidirectional communication channels called *Message Queues* (MQ) consisting of a list that holds messages in kernel space. The number of messages that a MQ can store can be specified for each RT-process at creation time. Messages have fixed sizes and strict copy to value semantics.

5.2. MINIX IPC Primitives

MINIX has the following IPC kernel functions:

- *mini_send(caller, dest, m_ptr)*: A message is copied from the *caller*'s message buffer pointed by *m_ptr* to the *dest*'s message buffer if *dest* process it is blocked waiting for that message, otherwise the *caller* process is blocked.

- *mini_rec(caller, src, m_ptr)*: If the sender process *src* is blocked trying to send a message to the *caller* process, the message is copied from the *src*'s buffer to the buffer pointed by *m_ptr* and the *src* process is unblocked, otherwise the *caller* process is blocked.

MINIX offers the following primitives in the *libc.a* library for higher layers:

- *send(dest, m_ptr)*: A message is copied from the sender's message buffer pointed by *m_ptr* to the *dest*'s message buffer if *dest* process is blocked waiting for that message, otherwise the caller process is blocked.
- *receive(src, m_ptr)*: If the *src* process is blocked trying to send a message to the receiver, the message is received and *src* process is unblocked, otherwise the receiver process is blocked.
- *sendrec (dest_src, m_ptr)*: A *send()* followed by a *receive()* to/from the same process in a single function. It is used for system calls avoiding to make two transitions from User-mode to Kernel-mode.

5.3. MINIX4RT IPC Primitives Features

Sometimes there are needs to use some type of policy to control the system behavior on message transfers. That policy may differ for messages sent to *request* services from those messages used by Servers to *reply* that requests. MINIX uses the same *mini_send()* primitive for both operations without distinguish among them.

The use of the same function for service *requests*, for service *replies*, to *signal* interrupts, etc. does not help for the apply a policy for each type of action.

A RT-process can not use MINIX IPC primitives because:

- *mini_send()* and *mini_rec()* kernel funcions could change the RT-process status to *READY*, and therefore the RT-process would be selected to execute by MINIX scheduler loosing all its RT-execution attributes.
- As MINIX IPC does not support different behaviors for *mini_send()*, any priority inversion avoidance protocols can be applied.
- If a RT-process make a request to a a NRT-process using *mini_send()*, the RT-process must wait for the reply until NRT-process will run at NRT-priority. This is another case of Unbounded Priority Invesion (detailled in Section 5.17)

As a RT-process can not use MINIX IPC primitives, it is inhibited of making any MINIX System Calls (except *exit()* as was explained in Chapter 3).

MINIX4RT offers a variety of new *Kernel Calls* that let apply different policies depending on the message transfer use. The term *Kernel Calls* is used to distinguish the way that they operate against *System Calls*. A more detailed explanation will be find in Chapter 6.

RT-IPC Kernel Calls have the following features:

- Synchronous/Asynchronous message transfer using Message Queues.
- Configurable Message Queue size.
- Different beahavior of send operations for requests, replies, signals and interrupts.
- Timeout support for synchronous primitives.
- Configurable dequeuing policy (Priority order or FIFO order).
- Priority Hand-Off to avoid unbounded priority inversion.
- Basic Priority Inheritance Protocol [37] support to avoid unbounded priority inversion.
- Sending timestamps and message IDs can be retrieved by the receiver.
- Senders process's attributes are stored with the message header (priority, process type, process deadline, etc).

The RT-kernel has a message pool named *MRT_sm.pool* that has *NR_MESSAGES* message queue entries in kernel space. Each message queue entry includes space for the message itself and its header.

When a NRT-process is prepared to be converted into a RT-process using the *mrt_setproc()* System Call a MQ is assigned to it. A MQ has a bounded capacity that quantifies its ability to store messages. The size requested of the MQ is a field of the *mrt_pattr_t* data structure passed as function parameter of *mrt_setproc()*. The *pmq* field of the process descriptor is filled with a pointer to the message queue assigned to the process.

5.4. Message Descriptor Data Structure

A Message Descriptor Data Structure (*mrt_msgd_t*) is composed by the Message Payload Data Structure and the Message Header Data Structure described in the next sections (see Appendix D).

5.4.1. Message Payload Data Structure

MINIX kernel defines six messages types (see Figure 5.1). The sizes of message elements will vary, depending upon the architecture of the machine; this diagram illustrates sizes on a machine with 32-bit pointers.

MINIX4RT keeps the message formats but without the first two fields (*m_source* and *m_type*) in the Message Payload Data Structure (*MRT_msg_t*).

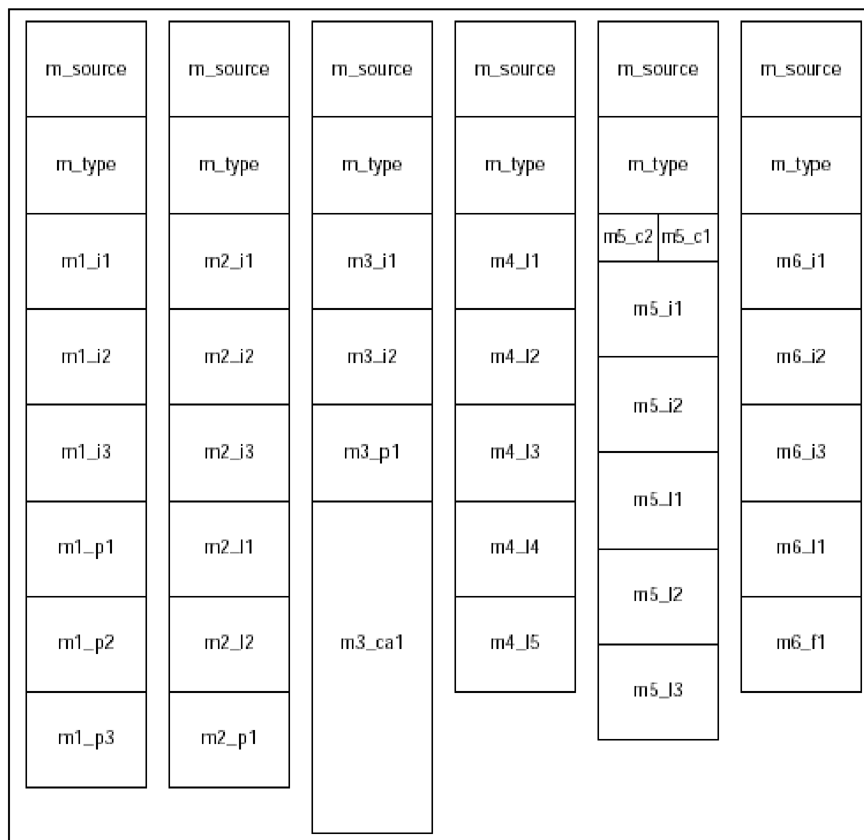


Figure 5.1: MINIX Message Types (From [2]).

5.4.2. Message Header Data Structure

As the RT-kernel needs more information to describe a message, a Message Header Data Structure (*MRT_msg_hdr_t*) with the following fields is defined:

- *source*: RT-PID of the process who sent the message.
- *dest*: RT-PID of the destination process.
- *mtype*: What kind of message is it.
- *mid*: A message ID set by the kernel.
- *seqno*: A message sequence number. It is the number of message sent by the sender RT-process.
- *tstamp*: The value of the system counter *RTM_sv.counter.ticks* when the message was inserted into the MQ.
- *priority*: The message priority that equals the sender's priority (see Section 5.18).

5.5. The Message Queue Entry Descriptor

A Message Queue Entry Descriptor (MQE) can store one Message Descriptor (*mrt_msgd_t*) and other fields that are needed to conform MQs and handling message timeouts. The fields of the *MRT_mqe_t* data structures are (see Appendix D):

- *msgd*: A Message Descriptor.
- *index*: A Message Queue Entry ID.
- *pvt*: A pointer to a VT that handle the message timeouts.
- *next*: A forward pointer.
- *prev*: A backward pointer.

5.6. The Message Queue Descriptor

A Message Queue Descriptor (*MRT_msgq_t*) let the RT-kernel organize MQEs. Each MQ descriptor have the following fields:

- *index*: The message queue ID used for quick searches.
- *size*: The message queue size.
- *flags*: The message queue policy flags:
- *MRT_PRTYORDER*: Priority Order Policy
- *MRT_FIFOORDER*: First In First Out Order Policy
- *inQ*: It counts the number of messages enqueued.
- *maxinQ*: It counts the highest number of message enqueued.
- *owner*: The message queue owner.
- *delivered*: The total number of messages delivered.
- *enqueued*: The total number of messages enqueued.
- *pvt*: A pointer for the VT related with the MQ. It is used to handle the timeout of the *mrt_rcv()* Kernel Call.
- *mQ*: An array of queues to handle the messages in priority or FIFO order.

5.7. The RT-System Message Pool

The memory space where messages are stored is called the *System Message Pool*. It is allocated in kernel memory space before communications to eliminate the buffer allocation delay (see Figure 5.2).

The list that keeps the free Message Queue Entries is *RTM_sm.mfreeQ*. It has the same data structure type *MRT_msgq_t* than other MQs.

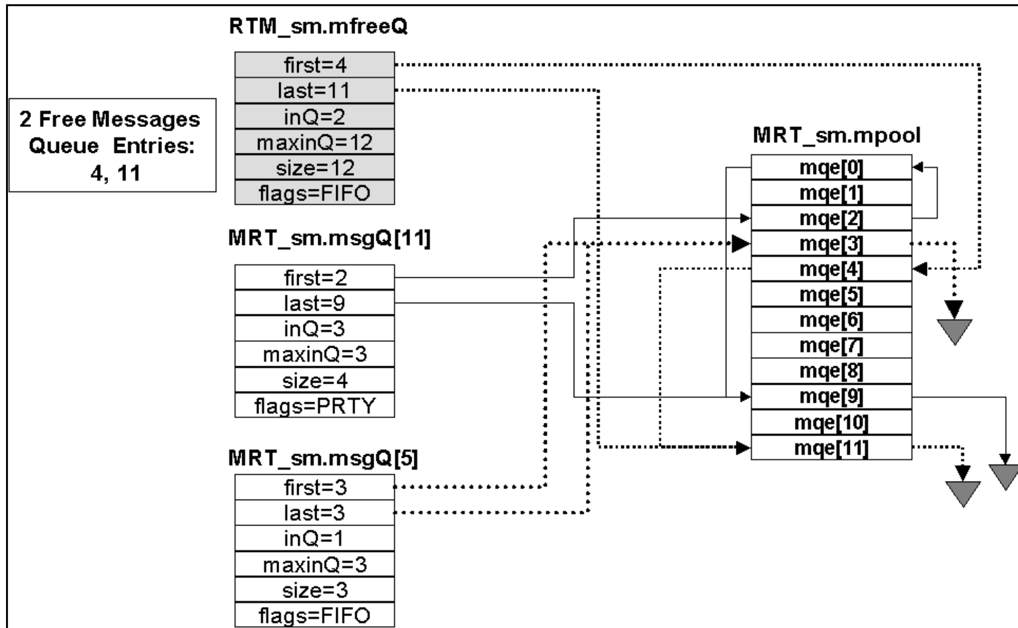


Figure 5.2: System Message Pool.

5.8. Message Queues Management

To manage the MQs the RT-kernel uses the *MRT_mQ_t* data structure that is composed by:

A set of MQE queues, one queue assigned for each priority level.

A bitmap with one bit assigned for each priority. A bit set means that the related queue has at least one message. Initially, all the bits are cleared indicating that all queues are empty.

When a MQE is enqueued, the related bit to its *priority* is set in bitmap, and the MQE is appended to the queue in accordance with its *priority* field.

Finding the highest priority MQE in the MQ is therefore only a matter of finding the first bit set in bitmap. Because the number of priorities is fixed, the time to complete a search is constant and unaffected by the number of enqueued MQE in the queues.

Each queue have two pointers, one for the *first* message descriptor and one for the *last* message descriptor enqueued (see Figure 5.3). The insertions in the queue can be in FIFO or LIFO order. MQE of the same priority will be managed under a FIFO policy.

The following kernel functions help to manage MQs and MQEs:

– *MRT_msgQ_rst()*: Resets all MQ fields.

– *MRT_msgQ_alloc()*: Allocs a MQ for a RT-process.

– *MRT_mqe_rst()*: Resets all MQE fields.

– *MRT_mqe_alloc()*: Allocs a MQE from the Free queue for a MQ.

– *MRT_mqe_free()*: Releases a MQE returning it to the system Free queue.

– *MRT_mqe_app()*: Appends a MQE at the tail of a MQ.

– *MRT_mqe_rmv()*: Removes a MQE from a MQ.

– *MRT_mqe_pick()*: Selects the highest priority (Priority Ordering) or the oldest (FIFO Ordering) MQE sent by a specified source from a MQ.

As it can be seen, that the same design pattern was employed in the management of interrupt descriptor queues, ready process queues, expired timer queues and message queues.

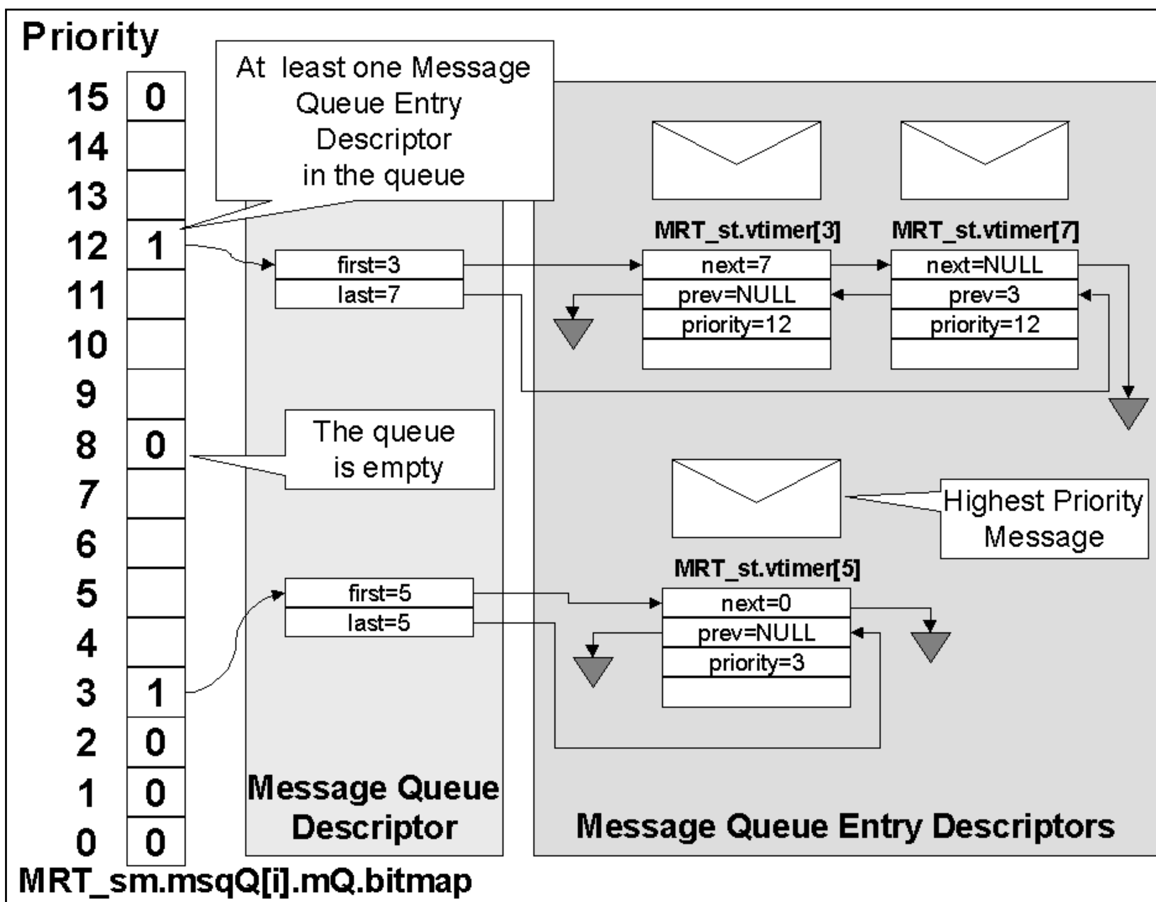


Figure 5.3: Message Queue Management.

5.9. The *mrt_rqst()* Kernel Call

The *mrt_rqst()* Kernel Call sends a request message to a RT-process through a MQ in a synchronous manner specifying a timeout.

The Kernel Call prototype is:

int mrt_rqst(dest, m_ptr, timeout)

Where:

- ***dest***: It is the RT-PID of the destination process.
- ***m_ptr***: It is a pointer to the message buffer.
- ***timeout***: It is the time to wait that the request message could be sent.

If the *dest* proccess is waiting for the message, it is copied from the caller's message buffer pointed by *m_ptr* to the *dest's* process message buffer. The *dest* process inherits the caller's priority if it is higher than its owns.

If *dest* process is not waiting for the message, the request is enqueued in the MQ owned by the *dest* process, and the caller is blocked until the message is received. The *dest* process and all other processes requested directly and indirectly by the *dest* process inherit the caller's priority if it is higher than they own.

A *timeout* in Timer ticks can be specified to wait for the request message could be sent. A special value of *MRT_NOWAIT* can be specified to return without waiting if the *dest* process is not blocked receiving the message. To wait until the *dest* process will receive the message, *MRT_FOREVER* must be specified as a *timeout*. If the *timeout* expired:

- The message is removed from the *dest's* MQ.
- The caller process is unblocked returning and *E_MRT_TIMEOUT* error code.
- The *dest's* process priority is set to the highest priority message in it's MQ.

5.10. The *mrt_arqst()* Kernel Call

The *mrt_arqst()* Kernel Call sends a request message to a process through a MQ in an asynchronous manner.

The Kernel Call prototype is:

int mrt_arqst(dest, m_ptr)

Where:

- ***dest***: is the RT-PID of the destination process.
- ***m_ptr***: is a pointer to the message buffer.

If the *dest* proccess is waiting for the message, it is copied from the caller's message buffer pointed by *m_ptr* to the *dest's* process message buffer. The *dest* process inherits the caller's priority if it is higher than its owns.

If *dest* process is not waiting for the message, the request is enqueued in the *dest*'s MQ, and the caller returns without waiting for the message will be received. The *dest* process and all other processes requested directly and indirectly by the *dest* process inherit the caller's priority if it is higher than they own.

5.11. The *mrt_reply()* Kernel Call

The *mrt_reply()* Kernel Call sends a message to a process through a MQ in an asynchronous manner. It can be used for replies in a botton-up way (i.e. Server to Client).

The Kernel Call prototype is:

int MRT_reply(dest, m_ptr)

Where:

- ***dest***: is the RT-PID of the destination process.
- ***m_ptr***: is a pointer to the message buffer.

If the *dest* process is blocked waiting for the reply, the message is copied from the caller's memory *m_ptr* to the *dest* process message buffer and *dest* is unblocked.

If the *dest* process is not blocked waiting for the message, the reply is enqueued in the *dest* process MQ.

At last, the caller's priority is set to the highest priority message in it's MQ or to its base priority *baseprty* if its MQ is empty.

5.12. The *mrt_uprqst()* Kernel Call

The *mrt_uprqst()* Kernel Call sends a message to a process through a MQ in an asynchronous manner. It can be used by the Tasks to make requests coming from remote processes in a botton-up way.

The Kernel Call prototype is:

int mrt_uprqst(dest, m_ptr, priority)

Where:

- ***dest***: is the RT-PID of the destination process.
- ***m_ptr***: is a pointer to the message buffer.
- ***priority***: is the priority of the message.

If the *dest* proccess is waiting for the message, it is copied from the caller's message buffer *m_ptr* to *dest*'s process message buffer.

If the *dest* process is not waiting the message, it is enqueued into the *dest* process MQ.

In both cases, the priority of the *dest* process will be changed if and only if its current priority is lower than the specified *priority* function argument.

5.13. The *mrt_sign()* Kernel Call

The *mrt_sign()* Kernel Call sends a message to a process through a MQ in an asynchronous manner. It can be used by the Tasks to sign processes in a botton-up way.

The Kernel Call prototype is:

int mrt_sign(dest, m_ptr)

Where:

- ***dest***: is the RT-PID of the destination process
- ***m_ptr***: is a pointer to the message buffer

If the *dest* process is blocked waiting for the message, it is copied from the caller's memory *m_ptr* to the *dest*'s process message buffer.

If the *dest* process is not waiting the message, the message is enqueued into the *dest*'s process MQ.

5.14. The *MRT_send()* Kernel Function

The *MRT_send()* Kernel Function sends a message to a process through a MQ in asynchronous manner. It can be used by ISRs to send messages to Tasks in a botton-up way.

The Kernel Call prototype is:

int MRT_send(dest, prty, mtype)

Where:

- ***dest***: The RT-PID of the destination process.
- ***prty***: The message priority.
- ***mtype***: The message type.

If the *dest* process is blocked waiting for the message, the message header is copied into the *dest*'s process message buffer.

If the *dest* process is not blocked waiting the message, the message header is copied into the *dest*'s process MQ, the field *st.mdl* (Missed DeadLines) of the *dest* process's descriptor is increased and a *MT_SIGNAL* message is sent to the watchdog process of the *dest* process.

5.15. The *mrt_rcv()* kernel Call

The *mrt_rcv()* Kernel Call is used to receive a message.

The Kernel Call prototype is:

int mrt_rcv(source, hdr_ptr, m_ptr, timeout)

Where:

- ***source***: specify the RT-PID of the message sender from which the caller wants to receive a message. A special value of *MRT_ANYPROC* can be specified to receive a message from any source.
- ***hdr_ptr***: specify the caller's buffer for the message header.
- ***m_ptr***: specify the caller's buffer for the message payload.
- ***timeout***: specify a timeout to wait for the message. Two special values can be specified:
  - *MRT_NOWAIT*: The process returns with o without receiving the message. If the message has been received, the function return code is *OK* else returns *E_TRY_AGAIN*.
  - *MRT_FOREVER*: The process waits until the message is receive.

The caller searches for a message from the specified *source* into it's MQ with the retrieving policy of the MQ (Priority or FIFO order).

If there are no message from the *source* process, the caller is blocked.

If the *source* process is blocked trying to send the message in a synchronous manner it is unblocked.

The caller process can specify a *timeout* to unblock itself. If no message from the specified *source* is received in the specified period it returns *E_MRT_TIMEOUT* error code.

If a message is received, the *hdr_ptr* buffer is filled with the header fields of the message received.

5.16. The *mrt_rqrcv()* kernel Call

The *mrt_rqrcv()* Kernel Call optimize the performance of the common operations of send a request message to a server process and waits for the reply message. It saves one context switch for the caller process.

The Kernel Call prototype is:

int mrt_rqrcv(dest, rqst, rply, hdr, timeout)

Where:

- ***dest***: Specifies the RT-PID of the destination processs.
- ***rqst***: Specifies the caller's buffer for the request message.
- ***rply***: Specifies the caller's buffer for the reply message payload.
- ***hdr***: Specifies the caller's buffer for the message header.
- ***timeout***: specify a timeout to wait for the message. A special value of *MRT_FOREVER* can be specified to wait until the reply message is received.

5.17. Using MINIX4RT IPC Kernel Calls

Table 5.1 resumes the uses of *send-type* RT-Kernel Calls. Table 5.2 resumes the uses of the *mrt_rcv()* RT-Kernel Call.

Table 5.1: Send-type Kernel Calls Uses.

Use	Kernel Call	Caller's Process Priority	Destination's Process Priority	Caller waits until
Synchronous Request	*mrt_rqst()*	It is not changed	It is set to the caller's priority if it is higher than it's current priority	The message is received or a timeout expire.
Asynchronous Request (used by non blocking Servers)	*mrt_arqst()*	It is not changed	It is set to the caller's priority if it is higher than it's current priority	It does not wait

Table 5.1: Send-type Kernel Calls Uses (cont.)

Use	Kernel Call	Caller's Process Priority	Destination's Process Priority	Caller waits until
Reply	*mrt_reply()*	It is set to the value of the highest priority message in its MQ or to its base priority	It is not changed	It does not wait
Up request message	*mrt_uprqst()*	It is not changed	It is set to speficied priority if it is higher than its current priority	It does not wait
Signal message	*mrt_sign()*	It is not changed	It is not changed	It does not wait
Request & Receive	*mrt_rqrcv()*	It is not changed	It is set to the caller's priority if it is higher than it's current priority	The reply message is received or a timeout expire.
Send a message (used by ISR)	*MRT_send()*	It is not changed	It is set to speficied priority if it is higher than its current priority	It does not wait if the destination process is waiting the message or enqueue the message and increase de missed deadline of the destination process

Table 5.2: *mrt_rcv()* Kernel Calls Uses.

mrt_rcv use	source	timeout
To receive a reply.	RT-PID of the requested process	MRT_NOWAIT < *timeout* < MRT_FOREVER
To receive requests.	MRT_ANYSRC	MRT_FOREVER

5.18. Priority Invertion

In many RT-applications, there are resources that must be shared among processes in a way that prevents more than one process from using the resource at any moment (mutual exclusion). *Priority inversion* is the term used to describe a situation where a process is waiting for a lower priority process to free a shared, exclusive use resource. Clearly, a priority based RT-system cannot tolerate significant periods of priority inversion. A sample helps to illustrate the problem. See Figure 5.4, where:

T_H: Highest Priority Process.

T_M: Medium Priority Process.

T_L: Lowest Priority Process.

S: Server Process.

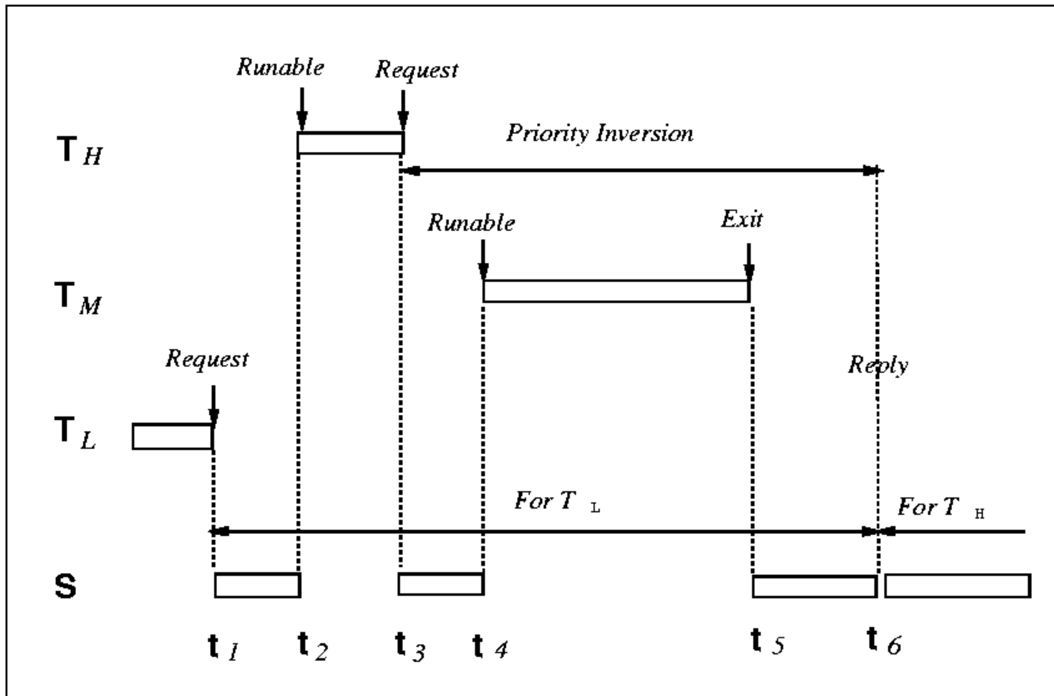


Figure 5.4: Priority Inversion.

The sequence of events are:

t_1: T_L requests a service to S.

t_2: S is preempted by T_H.

t_3: T_H requests a service to S.

t_4: S stills working on T_L request.

t_5: T_M preempts S.

t_6: T_M exits and S stills working on T_L request.

t_7: S replies to T_L and starts working on T_H request.

The time interval between t_3 and t_6 is a *Priority Inversion*, but the interval between t_4 and t_5 may be unbounded.

The unbounded priority inversion is one serious problem in RT-systems. There has been developed many mechanisms to avoid it. The priority inversion problem in Client/Server communication is more serious one, since the length of priority inversion tends to be much longer than that of synchronization.

5.19. Basic Priority Inheritance Protocol (BPIP)

Sha, Rajkumar and Lehosky [38] have proposed two protocols to avoid the priority inversion problem. One is the Basic Priority Inheritance Protocol (BPIP), the other is the Priority Ceiling Protocol (PCP).

Priority inheritance mechanisms are intended to prevent unbounded priority inversion by adjusting the effective priority of a lower priority process T_L whenever a higher priority process T_H is suspended waiting to be served.

A Server process ***S*** is said to be executing on *behalf of* a Client process ***P*** if ***S*** is executing a ***P'***s request or another process request that is executing a ***P'***s request.

A process ***S*** is said to be *blocking* process ***P*** if ***P*** (or a Server executing on *behalf of* ***P***) has made a request to ***S***, but ***S*** is executing other process request.

The BPIP consists of the following rules [37]:

- A Server process ***S*** executes at its runtime assigned priority when it does not block any process or when it is not executing on behalf of a Client.
- If a Server process ***S*** is executing on behalf of a Client process ***P*** or is blocking one or more process, ***S*** executes at either the priority of ***P*** or the priority of the highest priority process (if any) that ***S*** blocks, whichever is higher.
- Process requesting services of other process are served in priority order.

Under the BPIP, a Client process can be blocked in two ways:

– *Directly*: when the called process either has a queued request or is executing other process request.

– *Indirectly or push-through*: when a process inherits a higher priority.

The BPIP potentially requires priorities to be modified when processes make requests. A Server process may inherit the priority of a higher priority Client process even though the Server is not executing that Client request. Thus, each Server has an assigned base priority (the *baseprty* field) and an effective priority (the *priority field*).

The behavior of BPIP in a Client/Server application is illustrated in Figure 5.5.

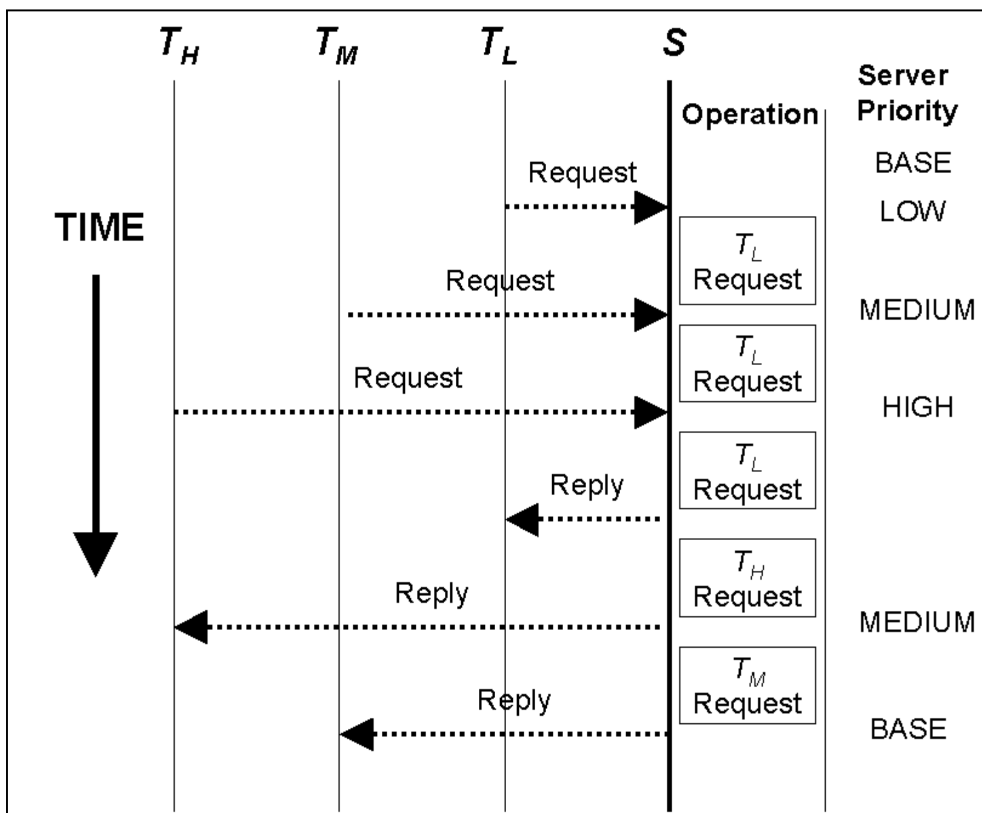


Figure 5.5: Basic Priority Inheritance Protocol Sample.

Asumptions:

- *HIGH*, *MEDIUM*, *LOW*, *BASE* are process priorities.

- *HIGH > MEDIUM > LOW > BASE.*

- *S's priority = BASE.*

- T_L*'s priority = LOW.*

- T_M*'s priority = MEDIUM.*
- T_H*'s priority = HIGH.*
- T_L *priority = LOW.*

The sequence of events is described below:

- T_L sends a request message to ***S***.
- ***S*** executes on behalf of T_L with T_L's priority.
- While ***S*** is processing T_L's request, T_M preempts ***S*** and sends a new request message to it. As ***S*** has not finished with T_L's request, it is still running but at T_M's priority because it is blocking T_M.
- While ***S*** is processing T_L's request, T_H preempts ***S*** and sends a new request message to it. As ***S*** has not finished with T_L's request, it is still running but at T_H's priority because it is blocking T_H.
- Afterward ***S*** finishs with T_L's request and sends a reply message to T_L's. Then, it gets the highest priority message from it MQ (T_H's request), and starts to process it.
- When ***S*** finishs with T_H's request, it sends a reply message to T_H and it's priority is changed to the highest priority message from it MQ (T_M's request) waiting to be received.
- When ***S*** finishs with T_M's request, it sends a reply message to T_M and it's priority is changed to the highest priority message waiting to be received but as the MQ is empty, it's efective priority is reset it's base priority.

In the previous sample, the higher priority process T_H must wait that ***S*** finishs with the lower priority process (T_L) request. It is a bounded priority inversion limited to the time consumed by ***S*** when it was executing on behalf of T_L.

To achieve the correct behavior and be compliance with BPIP, priority inheritance needs to be a transitive operation. Therefore, the RT-kernel must search across the chain of requested processes to apply the priority inheritance until it finds the process that has no pending requests. To search across the chain of a requested process, the RT-kernel uses *mrt_sendto* and *mrt_getfrom* fields of process descriptors.

5.20. The Priority Ceiling Protocol (PCP)

One drawback of the BPIP is that it not prevents deadlocks and chained blocking. Dutertre [39] shows this behaviour on a BPIP implementation based on semaphores. MINIX4RT is a systen based on message transfer and its IPC primitives are based on the proposal of Borger and Rajkumar [37].

The kinds of deadlocks exposed by Dutertre can not occur in MINIX4RT top-down Client/Server programming model because all request goes down and replies goes up. It has the same behavior that imposing a total ordering on resource use. System programmers must consider this programming model to avoid deadlocks.

A necessary conditions for deadlocks is the use of blocking primitives, but MINIX4RT offers blocking and non-blocking Kernel calls with and without timeouts.

Chained blocking occurs when the system uses semaphores and a higher priority process T_H is blocked by several processes of lower priorities in succession. For example, a process T_M locks a semaphore S_1 and a process T_L locks a semaphore S_2. When T_H tries to lock S_1 it fails and T_M inherits T_H´s priority. When T_M tries to lock S_2 it fails and T_L inherits T_M´s priority. T_H must wait until T_L release S_2 and T_M releases S_1 although T_H does not use S_2 directly.

Chained blocking could occurs in a system with message transfers as MINIX4RT. For example, a process T_M sends a request message to Server S_1 and a process T_L sends a request message to Server S_2. If T_H sends a request message to Server S_1, it must wait and S_1 inherits T_H´s priority. If S_1 sends a request message to Server S_2 to process T_M´s request, it must wait and S_2 inherit T_H´s priority (transitive). T_H is blocked waiting for:

- The reply of S_2 to T_L.
- The reply of S_2 to and S_1.
- The reply of S_1 to T_M.
- The reply of S_1 to T_H.

If PCP would be developed for MINIX4RT, it will need to set a priority ceiling for each process. As Server and Task processes could be requested by any User-level processes, the priority ceiling of all Servers and Tasks must be set to the highest User-level process priority. If a process request a service and the Server is blocked waiting a synchronous I/O, any other User-level process that requests another non related service to other Server will be blocked, because the Priority Ceiling

is equal to the highest priority, disabling that any other request be treated until the former will finish. Therefore, the systems has the same behaviour than a monolitic non reentrant kernel.

If Server's Priority Ceiling is greater than or equal to any other User-level processes's priority, the use of PCP slows down the system performance because only one process could be attended by the RTOS at once.

5.21. Complete Priority Inheritance

Both, the BPIP and the PCP use the priority inheritance as a method to avoid the unbounded priority inversion problem. Those protocols require process priorities to be modified, but changing the priority of a process if one of the operations needed to implement priority inheritance. A RT-process has resources as VTs and messages sent to other processes that are handled according to their priorities, therefore they must change their priorities too.

To achieve a correct behavior of IPC primitives a Complete Priority Inheritance needs the following actions on processes and resources:

- If the process is in the RT-ready state, it must be removed from its current RT-ready Queue and inserted it into other RT-ready Queue according to its inherit priority.

- All process's expired VTs must be removed from their current Expired VT Queues and inserted into other Expired VT Queues according to the inherit priority of their owner.

- All messages sent to other processes must be removed from their current MQE lists and inserted into other MQE list according to the inherit priority of the sender.

If a request message sent by a lower priority process T_L does not change its priority when T_L inherits a higher priority, T_L request may be (unbounded) delayed by other medium priority requests sent to the same destination process. Therefore a Client with higher priority that request a service from T_L will be delayed too.

As it can be seen, the Complete Priority Inheritance could implies a performance penalty to IPC and increases the inversion delays produced by the OS itself [27]. The current version of MINIX4RT only changes the priority of the first process in the inheritance chain.

6. RT-SYSTEM CALLS, KERNEL CALLS AND FUNCTIONS

6.1. MINIX System Calls Implementation

MINIX offers *mini_send()* and *mini_rec()* basic kernel primitives as was explained in Chapter 5. At higher levels *send()*, *receive()* and *sendrec()* are used. All System Calls are implemented using *sendrec()* to send the request to a server process and to wait the reply from it. The standard servers are Memory Manager (*MM*) and File Sytem manager (*FS*). These interactions are deeply explained by Ashton [3].

An example illustrates the implementation of a MINIX System Call (see Figure6.1).

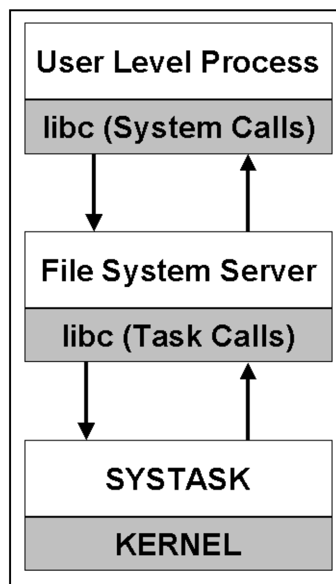


Figure 6.1: MINIX System Calls Implementation.

When a User-level process wants to get its user time consumed it calls *utime()*, the call is converted by the system library (*libc.a*) into a *sendrec()* Call to the *FS* server. As the *FS* server does not have the information requested, it must transfer the request to a Task named *SYSTASK* using a

Task Call (with *_taskcall()* function). A Task Call is like a System Call used by servers to request services to Tasks. Task Calls use *send()/receive()* primitives too.

As Tasks share the kernel memory space and data structures, the *SYSTASK* is like a kernel representative that can get process accounting times from kernel tables to reply to the *FS* server that can reply to the User-level process.

6.2. MINIX4RT System Calls Implementation

MINIX4RT offers a set of System Calls that operates in a similar way as MINIX System Calls does. Unlike MINIX, it does not use the *SYSTASK* for source code readability.

For those time constrained services MINIX4RT offers Kernel Calls that are not inplemented using message transfers. They call RT-kernel services through a processor Trap.

The following sections describe how MINIX4RT implements System Calls and Kernel Calls and Appendix D is the reference of data structures used by them.

6.2.1. RT-System Calls with Message Transfers

Some RT-services do not need be requested by RT-processes. Examples of these services are the start of the Real-Time Processing Mode, stops it, gets system statistics, etc. They are not used for time constrained functions and they use MINIX *send()/receive()* primitives. Therefore, the RT-kernel could preempt the requesting process during the Sytem Call if a RT-process is ready to run. These type of RT-related System Calls are implemented using the *MM* server that interacts with a new task called *MRTTASK* (see Figure 6.2)

MRTTASK is like a RT-kernel representative that can access to RT-kernel data structures to reply to the *MM* server that can reply to the User-level process.

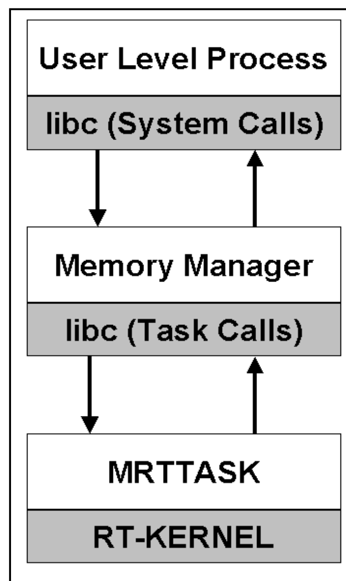


Figure 6.2: RT-System Calls with Message Transfers.

6.2.2. RT-Kernel Calls without Message Transfers

For time constrained services the MINIX4RT offers RT-kernel Calls. They interact with the RT-kernel directly without message transfers as in a monolitic OS using a processor Trap. (see Figure 6.3).

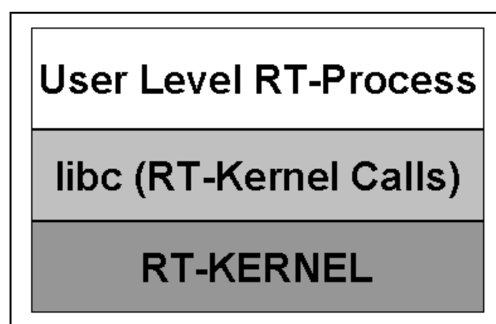


Figure 6.3: RT-Kernel Calls without Message Transfers.

This approach does not generate a sequence of context switches but implies a change from User-mode to Kernel-mode and viceversa.

6.2.3. The RT-PID

Every process in the system has an assigned unique identifier named the process identifier (PID). PIDs are the common mechanism used by applications to reference

processes. Inside the MINIX kernel the *p_nr* field of the process descriptor is used for its IPC primitives instead of the PID.

As is was described in Chapter 5, the IPC Kernel Calls use the RT-PID to identify senders and receivers RT-processes. The RT-PID is a data structure of type *mrtpid_t* with the following fields:

- *pid*: The process ID assigned by MINIX when the process was created.
- *p_nr*: The process number assigned by the MINIX kernel when the process was created. It is related with the slot number of the kernel process table.

The IPC RT-Kernel Calls could be implemented using only the process PID to identify the RT-process. This strategy implies that the RT-kernel must scan all the process table until it finds the process that matches the specified PID. Once it find the process, the RT-kernel could get its *p_nr* field of the process descriptor needed for carry out the Kernel Call. This approach impose an unacceptable performance penalty for a RTOS.

The IPC Kernel Calls could be implemented using only the process *p_nr* to identify a RT-process, but if the RT-process terminates and other process gets the same slot number in the kernel process table, all related processes that reference to the ended process (*p_nr)* will refer to the new process with the same *p_nr*. Therefore, both related field are needed to identify univocally a RT-process and to avoid an unacceptable delay.

6.3. Adding New RT-System Calls using MRTTASK

As MINIX4RT was designed for an academic environment, it is useful to know how to enhance its functionalities to make laboratory and coding practice. The following sections guide instructors into the steps of adding new System Calls.

To add a new RT-System Call it is appropriate to follow some system naming conventions. For a new System Call that will be named *mrt_**newcall**:*

- *mrt_**newcall**():* RT-system call to be used by User-level programs.
- *rtt_**newcall**()*: The name of the function used by de *MM* Server to invoke *MRTTASK*.

– *MRT_**newcall**()* : The name of the function executed by *MRTTASK* that makes the work.

– *MRT_**NEWCALL***: RT-System Call number.

To link each RT-System Call number with its service routine, the *MM* server uses a System Call dispatch table (defined in */src/mm/table.c*), which is stored in the *call_vec[]* array and has *NCALLS* entries. The *n*th entry contains the service routine address of the System Call number *n*.

To associate each function number with its corresponding service routine, the *MRTTASK* uses a dispatch table which is stored in the *MRT_vector[]* array. The *n*th entry contains the service routine address of the System Call having number (*MRT_FIRSTCALL* + *n*).

The Table 6.1 shows some steps to add a new RT-System Call (i.e. ***newcall***) using *MRTTASK*.

Table 6.1: Steps to Add a New RT-System Call.

Directory	Action	File	Description
/usr/src/lib/syscall/	create	mrt*newcall*.s	It is an Assembler source code file. Create a label named *_mrt_newcall* to jump to a C function named *__mrt_newcall.*
	modify	Makefile	Add the *mrtnewcall.s* compilation
/usr/src/lib/posix/	create	*_mrtnewcall.c*	Create a function named *_mrt_newcall()* that sends a message to the *MM* server using *_syscall(MM, MRTCALL, &m)* The message *m* must have the *MRTTASK* operation code (*MRT_NEWCALL*) in the field *m.m1_i1, m.m2_i1 or m.m3_i1*. The other message fields can be used for the *MRTTASK* function parameters.
	modify	Makefile	Add the *mrtnewcall.c* compilation.
/usr/src/lib/syslib/	create	*rtt_newcall.c*	Create a function named *rtt_newcall()* that sends a message to the *MRTTASK* using *_taskcall(MRTTASK, MRT_NEWCALL, &m)*. The message *m* have the *MRTTASK* function parameters.
	modify	Makefile	Add the *rtt_newcall.c* compilation.
/usr /include/minix/	modify	syslib.h	Change *MRT_NCALLS* and define *MRT_NEWCALL* Declare the function *rtt_newcall()* that the *MM* server invokes to request a service from *MRTTASK*.
		com.h	Add some *#define* of message fields used in *mrttask.c*
/usr /include/	modify	unistd.h	Add the function *mrt_newcall()* prototype and define data structures if needed.
	modify	errno.h	Add new error types i.e. *E_MRT_NEWERROR*

Table 6.1: Steps to Add a New RT-System Call (cont).

Directory	Action	File	Description
/usr/src/mm	modify	getset.c	Change the function *do_getset()* To add the case *MRT_NEWCALL* and the invocation to *MRTTASK*
	modify	param.h	Add message fields aliases
/usr /src/kernel/	modify	mrttask.c	Add the new functions of *MRTTASK*. The array of function pointer *MRT_vector[]* must be changed to point to the new function *MRT_newcall()*.
	# make clean; make install		Library compilation.
/usr/src/tools	*# make hdboot*		Kernel compilation.

6.4. Adding RT-Kernel Calls without Message Transfers

Only RT-processes can make RT-System Calls direct to the RT-kernel (RT-kernel Calls) with the exception of *mrt_set2rt()*.

The Table 6.2 shows some steps to add a new RT-Kernel Call (i.e. ***newkrn***).

Table 6.2: Steps to Add a New RT-Kernel Call.

Directory	Action	File	Description
/usr/src/lib/syscall/	create	*mrtnewkrn.s*	It is an Assembler source code file. Create a label named *_mrt_newkrn* to jump to a C function named *__mrt_newkrn*
	modify	Makefile	Add the *mrtnewkrn.s* compilation
/usr/src/lib/posix/	create	*_mrtnewkrn.c*	Create a function named *_mrt_newkrn()* to make a direct RT-Kernel Call using *rtkrncall(MRTNEWKRN, parm_ptr);* *MRTNEWKRN* is the number of the RT-Kernel Call The *parm_ptr* is a pointer to the RT-Kernel Call parameters.
	modify	Makefile	Add the *mrtnewkrn.c* compilation.
/usr /include/minix/		com.h	Update the *NRTKCALLS* constant and define *MRTNEWKRN* constant.
/usr /include/	modify	unistd.h	Add the function *mrt_newkrn()* prototype and define data structures if needed.
	modify	errno.h	Add new error types *E_MRT_NEWERROR*.

Table 6.2: Steps to Add a New RT-Kernel Call (cont).

Directory	Action	File	Description
/usr /src/kernel/	modify	mrtproc.c	Add the new function *MRT_newkrn()*.
	modify	mrtipc.c	Add the new function *MRT_newkrn()*.
	modify	proto.h	Add the new function *MRT_newkrn()* declaration.
/usr /src/lib	*# make clean; make install*		Library compilation.
/usr/src/tools	*# make hdboot*		Kernel compilation.

As it was explained in the previous section, it is appropriate to follow some system naming conventions to add a new RT-Kernel Call that will be named *mrt_**newkrn***:

– *mrt_**newkrn**()*: The RT-Kernel Call to be used by user space programs

– *MRT_**newkrn**()*: The name of the function executed by RT-kernel that do the work.

– *MRT**NEWKRN***: RT-kernel call number

6.5. RT-Kernel Functions

RT-kernel functions can only be used from programs linked with the RT-kernel code. They can be used in RT-Device Drivers, RT-ISRs and RT-tasks. Table 6.3 shows a summary of RT-kernel functions.

Table 6.3: RT-Kernel Functions.

Synchronization Primitives	
void MRT_lock(void)	Disables maskable interrupts.
void MRT_unlock(void)	Enables maskable interrupts.
*void MRT_saveFunlock(long *flags)*	Saves CPU Flags Register into a local static variable flags and enable interrupts.
*void MRT_saveFlock(long *flags)*	Saves CPU Flags Register into a local static variable flags and disables interrupts.
*void MRT_restoreF(long *flags)*	Restores CPU Flags Register from a local static variable flags.

Table 6.3: RT-Kernel Functions (cont.).

Synchronization Primitives	
void MRT_enable_irq(unsigned irq)	Enables an IRQ line at the PIC 8259 controller.
void MRT_disable_irq(unsigned irq)	Disables an IRQ line at the PIC 8259 controller.
unsigned MRT_get_PIC(void)	Gets the PIC 8259 mask.
void MRT_disable_irq(unsigned mask)	Sets the PIC 8259 to a specified mask.
Interrupt Management	
*void MRT_irqd_trigger(MRT_irqd_t *irqd)*	Sets the interrupt descriptor for running in the next returning from an IRQ or System Call.
*MRT_irqd_t *MRT_irqd_serviced(MRT_irqd_t *irqd)*	Unsets the interrupt descriptor for running in the next returning from an IRQ or System Call. It must not be used after running the interrupt handler because the RT-kernel do that.
*void MRT_irqQ_ins(MRT_irqd_t *irqd)*	Inserts an interrupt descriptor into an interrupt queue.
*void MRT_irqQ_rmv(MRT_irqd_t *irqd)*	Removes an interrupt descriptor from an interrupt queue.
int MRT_softirq_alloc(void)	Allocates a software interrupt descriptor. Returns the descriptor number (the index of the *MRT_si.irqtab[]* array).
void MRT_softirq_free(int irq_nbr)	Frees a software interrupt descriptor.
*void MRT_irqd_set(int irq, MRT_irqd_t *irqd)*	Sets the parameters of an interrupt descriptor (hardware and software interrupts).
*void MRT_irqd_free(MRT_irqd_t *irqd)*	Removes a interrupt descriptor from the interrupt queue and resets the descriptor fields.
*void MRT_irqd_rst(MRT_irqd_t *irqd)*	Reset all fields of an interrupt descriptor.
Process Management	
*void MRT_rdyQ_ins(MRT_proc_t *rp)*	Inserts a process descriptor at the head of a RT-ready Queue.
*void MRT_rdyQ_app(MRT_proc_t *rp)*	Appends a Process Descriptor at the tail of a RT-Ready Queue. There is a macro defined in proc.h named *MRT_ready()* that is assigned to *MRT_rdyQ_app()*.
*void MRT_rdyQ_rmv (MRT_proc_t *rp)*	Removes a process descriptor from a RT-ready Queue. There is a macro defined in proc.h named *MRT_unready()* that is assigned to *MRT_rdyQ_rmv()*.
void lock_pick_proc(void)	It is the scheduler. Its searches the RT-ready Queues for the highest priority RT-process. If the queues are empty, it runs the MINIX scheduler algorithm. The global variable proc_ptr is set to selected process descriptor. It resets the *MRT_NEEDSCHED* in the global system variable *MRT_sv.flags*.
void MRT_proc_rst (int p_nr)	Resets all RT parameters and statistics of a process specified in the p_nr argument.

Table 6.3: RT-Kernel Functions (cont.).

Process Management	
*int MRT_set2rt(MRT_proc_t *rp)*	Converts a NRT-process into a RT-process. Before call this function, fill all process descriptors fields.
*int MRT_set2nrt(MRT_proc_t *rp)*	Converts a RT-process into a NRT-process.
*int MRT_sleep(struct proc *rp,long int timeout)*	Sets the *MRT_P_SLEEP* state flag of a process and calls *MRT_unready()*. A timeout could be specified to wakeup the process at its expiration.
*int MRT_wakeup(struct proc *rp)*	Clears the MRT_P_SLEEP state flag of a process and calls *MRT_ready()*.
Time Management	
void MRT_set_timer(unsigned int Harmonic)	Initializes channel 0 of the 8253A timer to generate *Harmonic* * *HZ* interrupts by second.
unsigned int MRT_read_timer2(void)	Reads the channel 2 counter of the PIT. Used for latency calculation.
*void MRT_vtimer_free(MRT_vtimer_t *vt)*	Resets all fields of a VT descriptor.
*MRT_vtimer_t *MRT_vtimer_alloc (MRT_vtimer_t *vt)*	Allocates a VT with the parameters passed by the function argument.
*void MRT_vtimer_ins(MRT_vtimer_t *vt,MRT_vtQ_t *ptq)*	Inserts a VT into a VT queue.
*void MRT_vtimer_rmv(MRT_vtimer_t *vt,MRT_vtQ_t *ptq)*	Removes a VT from a VT queue.
Message Transfer	
int MRT_send(int dest,int prty,int mtype)	Sends an *mtype* message to a process identified by its process number *dest*. The message priority will be *prty*.

7. REAL-TIME PROCESSING RELATED STATISTICS

Tools for monitoring the behavior of an operating system are invaluable in performance tuning and debugging. MINIX4RT, as a RTOS for academic uses, provides several interfaces to system and process statistics which might be used for performance analysis.

This chapter describe multiple ways of gather statistics, Appendix A is a reference that includes statistics related System Call and Kernel Call, and Appendix B gives short examples of using statistics related programming interfaces. In all cases, however, the man pages are the definitive references.

7.1. System-wide Statistics

The *mrt_getstat()* System Call let gather system-wide statistics. The address of a data structure named *mrt_sysstat_s* must be passed as a parameter. On return, the RT-kernel fills the following fields with statistics since the last call to *mrt_RTstart()* or *mrt_RTrestart()* System Calls:

- *scheds*: It counts the number of RT-schedulings.
- *messages:* It counts the number of messages sent (received an not).
- *interrupts*: It counts the number of Hardware interrupts.
- *ticks*: It counts the number of Timer interrupts.
- *highticks*: It counts the number of *ticks* overruns.
- *idlemax*: It is the highest value counted by kernel *idlecount* variable. (explained in Section 7.4).

- *idlelast*: It is the last value counted by kernel *idlecount* variable. (explained in Section 7.4).

7.2. Interrupts Service Routines Statistics

The *mrt_getistat()* System Call let gather interrupt statistics. The interrupt number and the address of a data structure named *mrt_istat_s* must be passed as parameters. On return, the RT-kernel fills the following fields with statistics since the last call to *mrt_RTstart()* or *mrt_RTrestart()* for the specified interrupt descriptor number:

- *count*: It counts the the number of interrupts.
- *maxshower*: It is the highest number of TD-interrupts in a period.
- *mdl*: It counts the the number of Missed Deadlines.
- *timestamp*: It is the timestamp of the last interrupt.
- *maxlat*: It is the highest interrupt latency in PIT Hz units.
- *reenter*: It is the highest kernel reentrancy level.

7.3. Process Statistics

The *mrt_getpstat()* System Call let gather process statistics. The process RT-PID and the address of a data structure named *mrt_pstat_s* must be passed as parameters. On return, the RT-kernel fills the following fields with statistics for the specified process since it was converted into a RT-process:

- *scheds*: The number times that the RT-process was scheduled.
- *mdl*: The number of Missed Deadlines.
- *timestamp*: The last schedule timestamp in PIT ticks.
- *maxlat*: The maximun latency to dispatch the process (not implemented yet).
- *minlax*: The minimun laxity for the process (not implemented yet).

- *msgsent*: The number of RT-messages sent by the process.

- *msgrcvd*: The number of RT-messages received by the process.

Those fields match with the *rtstats_t* data structure of the *proc* data structure.

7.4. The IDLE Process

MINIX4RT CPU load estimation is based on the *IDLE* process. The *IDLE* process is executed when there are not any process ready to run. As more time is used by IDLE in a specified period, lower is the CPU used by the system and the other processes. *IDLE* is a funcion called *idle_task()* that jumps to *MRT_idle()*.

IDLE process uses five global variables to do its works. They are initialized by *MRT_idle_init()* Kernel Function and by the *mrt_RTstart()* System Call. The variables used by *IDLE* are:

- *MRT_sv.idlecount*: When the *IDLE* process is running, increments the value of this variable in an infinite loop. (function *MRT_idle()*)

- *MRT_sv.counter.idlelast*: The last value of *MRT_sv.idlecount* before the RT-kernel resets it.

- *MRT_sv.counter.idlemax*: The highest value of *MRT_sv.idlelast* since the last execution of *mrt_RTstart()* or *mrt_RTrestart()* System Calls.

- *MRT_sv.idlerefresh*: A counter that is decreased on each PIT tick to control the reset action of the *MRT_sv.idlecount* variable. Once it is reaches zero, the RT-kernel copies *MRT_sv.idlecount* to *MRT_sv.counter.idlelast*, compares its value against *MRT_sv.counter.idlemax* and replaces it with the highest. Then, the RT-kernel resets *MRT_sv.idlecount* and copies *MRT_sv.refresh* to *MRT_sv.idlerefresh*.

- *MRT_sv.refresh*: The period to refresh *IDLE* counters in PIT ticks. It is specified as an argument of the *mrt_RTstart()* System Call.

7.5. The Fx Keys

MINIX4RT as MINIX, use de *Fx Keys* to show statistics, attributes and status information about the system behavior. Several variables of interrupts, processes, MQs, and VTs are accessible to the user to provide data for benchmarking and testing new developments. When the user press an *Fx key*, the information is shown in the system console.

The *F1*, *F2, F3* and *F5* keys are used by MINIX. *F1* shows MINIX processes attributes, *F2* shows MINIX processes memory map, *F3* Toggle scrolling mode and *F5* shows Ethernet statistics.

7.5.1. The Shift-F1 Hot-Key

The Shift-F1 Hot-Key displays the RT-attributes of all running processes. In the last line of Figure 7.1 the process named *mrttest8c* that has the PID 27 is a RT-process (FLAGS=4002) waiting to receive a message from process 6 (RCVF=6). Its efective priority is 4 (PRTY=4) and its base priority is 4 too (BASE=4).

PROC	PID	FLAGS	PRTY	BASE	PERIOD	LIMIT	DEAD	WDOG	RCVF	SNDT	NAME
-11	0	0	15	15	0	0	0	-1	-111	-111	TTY
-10	0	0	15	15	0	0	0	-1	-111	-111	WINCH
-9	0	0	15	15	0	0	0	-1	-111	-111	MRTTASK
-8	0	0	15	15	0	0	0	-1	-111	-111	SYN_AL
-7	0	0	15	15	0	0	0	-1	-111	-111	IDLE
-6	0	0	15	15	0	0	0	-1	-111	-111	PRINTER
-5	0	0	15	15	0	0	0	-1	-111	-111	FLOPPY
-4	0	0	15	15	0	0	0	-1	-111	-111	MEMORY
-3	0	0	15	15	0	0	0	-1	-111	-111	CLOCK
-2	0	0	15	15	0	0	0	-1	-111	-111	SYS
-1	0	0	15	15	0	0	0	-1	-111	-111	HARDWAR
0	0	0	15	15	0	0	0	-1	-111	-111	MM
1	0	0	15	15	0	0	0	-1	-111	-111	FS
2	1	0	15	15	0	0	0	-1	-111	-111	INIT
3	26	0	15	15	0	0	0	-1	-111	-111	sh
4	17	0	15	15	0	0	0	-1	-111	-111	update
5	27	4002	4	4	0	0	0	-1	6	-111	mrttest8c

Figure 7.1: RT-Process Attributes.

The columns displayed have the following meanings:

- *PROC*: The process number.

- *PID*: The Process Identifier.
- *FLAGS*: The process' RT-status flags.
- *PRTY*: The process' effective priority.
- *BASE*: The process' base priority.
- *PERIOD*: The process' period specified in PIT ticks (only for *MRT_P_PERIODIC*).
- *LIMIT*: The limit for the number of RT-schedulings.
- *DEAD*: The process deadline specified in PIT ticks.
- *WDOG*: The process' watchdog process.
- *RCVF*: The process from which the process wants to receive a RT-message.
- *SNDT*: The process to which the process wants to send a RT-message.
- *NAME*: The name of the process.

7.5.2. The Ctrl-F1 Key

The Ctrl-F1 Hot-Key displays the RT-statistics of all running processes. In the last line of Figure 7.2 a process named *mrttest8c* was scheduled 10 times (SCHEDS=10), the last was at time 32753 ticks (TSTAMP=32753) since the last *mrt_RTstart()/mrt_restart()* system call invocation. The columns displayed have the following meanings:

- *PROC*: The process number. The first process is (-*NR_TASKS*).
- *PID*: The Process Identifier.
- *SCHED*: Counts how many times the process has been selected by the RT-scheduler.
- *MDL*: The number of Missed Deadlines.
- *TSTAMP*: The last schedule timestamp in PIT ticks.

– *MAXLAT*: The maximun latency to dispatch the process (not implemented yet).

– *MINLAX*: The minimun laxity for the process (not implemented yet).

– *SENT*: The number of RT-messages sent by the process.

– *RCVD*: The number of RT-messages received by the process.

– *VT*: The VT assigned to a Periodic process.

– *MQ*: The MQ assigned to a RT-processes.

PROC	-PID-	SCHEDS	MDL	TSTAMP	MAXLAT	MINLAX	SENT	RCVD	VT	MQ	COMMAND
-11	0	0	0	0	0	0	0	0	-1	-1	TTY
-10	0	0	0	0	0	0	0	0	-1	-1	WINCH
-9	0	0	0	0	0	0	0	0	-1	-1	MRTTASK
-8	0	0	0	0	0	0	0	0	-1	-1	SYN_AL
-7	0	0	0	0	0	0	0	0	-1	-1	IDLE
-6	0	0	0	0	0	0	0	0	-1	-1	PRINTER
-5	0	0	0	0	0	0	0	0	-1	-1	FLOPPY
-4	0	0	0	0	0	0	0	0	-1	-1	MEMORY
-3	0	0	0	0	0	0	0	0	-1	-1	CLOCK
-2	0	0	0	0	0	0	0	0	-1	-1	SYS
-1	0	0	0	0	0	0	0	0	-1	-1	HARDWAR
0	0	0	0	0	0	0	0	0	-1	-1	MM
1	0	0	0	0	0	0	0	0	-1	-1	FS
2	1	0	0	0	0	0	0	0	-1	-1	INIT
3	26	0	0	0	0	0	0	0	-1	-1	sh
4	17	0	0	0	0	0	0	0	-1	-1	update
5	27	10	0	32753	0	0	6	5	-1	1	mrttest8c

Figure7.2: RT-Processes Statistics.

7.5.3. The Shift-F2 Hot-Key

PIT Latency between 2 reads: is the time in PIT Hz, between two sequencial reads of the PIT LATCH counter. It is useful to estimate the time consumed by a read operation of the PIT *LATCH* used as was explained in Chapter 4.

The other system status, counters and statistics are simple to understand that not need further explanation (see Figure7.3).

```
Total number of interrupts = 117579
Total number of RT scheds  = 23
Total number of RT messages= 10
Free messages desc. in Pool= 32/32
Total number of RT ticks   = 109588
RT ticks/s                 = 200
MINIX ticks/s              = 50
PIT Latency between 2 reads= 14
Processing Mode Flags      = 0000000000000001
Bitmap of hard ints in use = 0100000011011011
Bitmap of soft ints in use = 0000000000000001
Bitmap of MINIX virtual PIC= 1011111100100000
Bitmap of the REAL PIC     = 1011111100100000
MRT_sv.counter.idlelast    = 8059642
MRT_sv.counter.idlemax     = 8087867
CPU Load                   = 1%
Sizeof mrt_msg_t           = 28
```

Figure 7.3: RT-System Wide Attributes and Statistics.

7.5.4. The F4 Key

The F4 Key displays the RT-attributes, status and statistics of all RT-Hardware interrupt descriptors as is shown in Figure 7.4. The columns displayed have the following meanings:

- *IRQ*: The interrupt descriptor number.
- *TSK*: The RT-Task number related with the interrupt.
- *WDG*: The watchdog process number.
- *COUNT*: It counts the number of interrupts.
- *MLAT*: Maximun Latency of the interrupt handler.
- *LAT*: The last latency of the interrupt handler.
- *RE*: The interrupt descriptor *reenter* field.
- *PY*: The interrupt priority.
- *TY*: The interrupt type.

– *VT*: The VT assigned to the interrupt descriptor (only for TD-interrupts).

– *TSTAMP*: The timestamp of the last handler execution.

IRQ	TSK	WDG	-COUNT-	MLAT	-LAT	RE	PY	TY	VT	-TSTAMP-	MS	MDL	PER	FLAG	NAME
0	-3	-1	121848	0	0	2	0	1	-1	121848	0	0	0	1	RT-CLOCK
1	-1	-1	2229	0	0	2	15	0	-1	121847	0	0	1	1	KEYBOARD
2	-1	-1	0	0	0	0	15	0	-1	0	0	0	0	0	PIC-CASCADE
3	-1	-1	0	0	0	0	15	0	-1	0	0	0	1	1	SERIAL-PORT-2
4	-11	-1	26	0	0	1	7	1	-1	49006	0	0	1	1	RT-RS232
5	-1	-1	0	0	0	0	15	0	-1	0	0	0	0	0	FREE
6	-1	-1	0	0	0	0	15	0	-1	0	0	0	1	1	DISKETTE
7	-1	-1	4	0	0	0	15	0	-1	102144	0	0	1	1	PARALLEL-PORT
8	-1	-1	0	0	0	0	15	0	-1	0	0	0	0	0	FREE
9	-1	-1	0	0	0	0	15	0	-1	0	0	0	0	0	FREE
10	-1	-1	0	0	0	0	15	0	-1	0	0	0	0	0	FREE
11	-1	-1	0	0	0	0	15	0	-1	0	0	0	0	0	FREE
12	-1	-1	0	0	0	0	15	0	-1	0	0	0	0	0	FREE
13	-1	-1	0	0	0	0	15	0	-1	0	0	0	0	0	FREE
14	-1	-1	6038	0	0	2	15	0	-1	118903	0	0	1	1	HARD-DISK
15	-1	-1	0	0	0	0	15	0	-1	0	0	0	0	0	FREE

Figure 7.4: Hardware Interrupt Descriptors.

– *MS*: The maximun value of *shower* field of the interrupt descriptor.

– *MDL*: The number of missed deadlines.

– *PER*: The period of the interrupt (only for TD-interrupts).

– *FLAG*: The status flags of the interrupt descriptor.

– *NAME*: The name of the interrupt descriptor.

Figure 7.4 shows that IRQ 4 is a Timer Driven RT-interrupt descriptor (TY=1) named RT-RS232 with a priority of 7 (PY=7) and a period of 1 (PER=1) tick. The handler was executed 26 times, the last at 49006 ticks since the last *mrt_RTstart()/mrt_restart()* system call invocation.

7.5.5. The Shift-F4 Hot-Key

The Shift-F4 Hot-Key displays the RT-attributes, status and statistics of all Software interrupt descriptors. The attributes, status and statistics displayed are the same as for the F4

key. Figure 7.5 shows the sofware IRQ 16 dedicated to manage the MINIX clock (timer) virtualization. The handler was executed 31215 times since the last *mrt_RTstart()/mrt_restart()* system call invocation.

IRQ	TSK	WDG	-COUNT-	MLAT	-LAT	RE	PY	TY	VT	-TSTAMP-	MS	MDL	PER	FLAG	NAME
16	-3	-1	31215	0	0	0	15	7	0	124860	0	0	4	1	MINIX-CLOCK
17	-1	-1	0	0	0	0	15	4	-1	0	0	0	0	0	SOFTIRQ
18	-1	-1	0	0	0	0	15	4	-1	0	0	0	0	0	SOFTIRQ
19	-1	-1	0	0	0	0	15	4	-1	0	0	0	0	0	SOFTIRQ
20	-1	-1	0	0	0	0	15	4	-1	0	0	0	0	0	SOFTIRQ
21	-1	-1	0	0	0	0	15	4	-1	0	0	0	0	0	SOFTIRQ
22	-1	-1	0	0	0	0	15	4	-1	0	0	0	0	0	SOFTIRQ
23	-1	-1	0	0	0	0	15	4	-1	0	0	0	0	0	SOFTIRQ
24	-1	-1	0	0	0	0	15	4	-1	0	0	0	0	0	SOFTIRQ
25	-1	-1	0	0	0	0	15	4	-1	0	0	0	0	0	SOFTIRQ
26	-1	-1	0	0	0	0	15	4	-1	0	0	0	0	0	SOFTIRQ
27	-1	-1	0	0	0	0	15	4	-1	0	0	0	0	0	SOFTIRQ
28	-1	-1	0	0	0	0	15	4	-1	0	0	0	0	0	SOFTIRQ
29	-1	-1	0	0	0	0	15	4	-1	0	0	0	0	0	SOFTIRQ
30	-1	-1	0	0	0	0	15	4	-1	0	0	0	0	0	SOFTIRQ
31	-1	-1	0	0	0	0	15	4	-1	0	0	0	0	0	SOFTIRQ

Figure 7.5: Software Interrupt Descriptors.

7.5.6. The Ctrl-F4 Hot-Key

The Ctrl-F4 Hot-Key displays the interrupt descriptors queues and the bitmap of Interrupt Queues as is shown in Figure 7.6. The columns displayed have the following meanings:

- *PRI*: The priority of the interrupt descriptor.
- *INQ*: The current number of descriptors enqueued.
- *PEND*: The number of pending interrupt descriptors.
- *Enqueued Interrupt Descriptors*: The list of Interrupt descriptors for each queue.

```
Interrupt Pending Queues bitmap =  0000000000000000
PRI INQ PEND Enqueued Interrupt Descriptors
  0   1    0 =>[ 0]
  1   0    0
  2   0    0
  3   0    0
  4   0    0
  5   0    0
  6   0    0
  7   1    0 =>[ 4]
  8   0    0
  9   0    0
 10   0    0
 11   0    0
 12   0    0
 13   0    0
 14   0    0
 15   6    0 =>[ 1]=>[ 3]=>[ 6]=>[ 7]=>[14]=>[16]
```

Figure 7.6: Interrupt Descriptors Queues.

7.5.7. The Shift-F5 Hot-Key

The Shift-F5 Hot-Key displays the RT-ready processes queues and MINIX ready processes queues and the RT-ready queue bitmap.

```
Priority Queues bitmap =  0000000000010000
PRI MAXINQ INQ PROC-LIST
  0      0   0
  1      0   0
  2      0   0
  3      0   0
  4      2   1 =>[ 7]
  5      0   0
  6      0   0
  7      0   0
  8      0   0
  9      0   0
 10      0   0
 11      0   0
 12      0   0
 13      0   0
 14      0   0
 15      0   0
PRI MINIX-PROC-LIST
 0 ->[0]
 1
 2
```

Figure 7.7: RT-Ready Queues and MINIX Ready Queues.

Figure 7.7 shows process number 7 as the only process in the RT-ready queue of priority 4. This queue has got 2 process enqueued.

The columns displayed have the following meanings:

- *PRI*: The priority of RT-ready processes queue.
- *MAXINQ*: The maximun number of descriptors enqueued.
- *INQ*: The current number of descriptors enqueued.
- *PROC-LIST*: The current list of RT-ready processes in each queue.
- *MINIX-PROC-LIST*: The current list of MINIX ready processes in each queue.

7.5.8. The F6 Key

The F6 Key displays the RT-message pool with the messages attributtes. The columns displayed have the following meanings:

MSG	SRCE	DEST	TYPE	-MID-	SEQNO	TIMESTAMP	TOUT	PRI	DLINE	LAXTY
0	7	6	2	5	9	3233	-1	4	0	0
1	-1	-1	0	0	0	0	-1	0	0	0
2	-1	-1	0	0	0	0	-1	0	0	0
3	-1	-1	0	0	0	0	-1	0	0	0
4	-1	-1	0	0	0	0	-1	0	0	0
5	-1	-1	0	0	0	0	-1	0	0	0
6	-1	-1	0	0	0	0	-1	0	0	0
7	-1	-1	0	0	0	0	-1	0	0	0
8	-1	-1	0	0	0	0	-1	0	0	0
9	-1	-1	0	0	0	0	-1	0	0	0
10	-1	-1	0	0	0	0	-1	0	0	0
11	-1	-1	0	0	0	0	-1	0	0	0
12	-1	-1	0	0	0	0	-1	0	0	0
13	-1	-1	0	0	0	0	-1	0	0	0
14	-1	-1	0	0	0	0	-1	0	0	0
15	-1	-1	0	0	0	0	-1	0	0	0
16	-1	-1	0	0	0	0	-1	0	0	0
17	-1	-1	0	0	0	0	-1	0	0	0
18	-1	-1	0	0	0	0	-1	0	0	0
19	-1	-1	0	0	0	0	-1	0	0	0

Figure 7.8: RT-message Pool (Message Attributtes).

- *MSG*: The message queue entry number in the array *MRT_sm.mpool.mqe[]*.

- *SRCE*: The message source process.
- *DEST*: The message destination process.
- *TYPE*: The message type described in *include/minix/const.h*.
- *MID*: The system wide message ID.
- *SEQNO*: The source process message sequence number.
- *TIMESTAMP*: The message sent timestamp.
- *TOUT*: The message timeout.
- *PRI*: The message sender priority.
- *DLINE*: The message sender deadline.
- *LAXTY*: The message sender laxity.

Figure 7.8 shows message number 0 with a timestamp of 3233 from source 7 and destination 6 with a message ID of 2, a sequence number of 9 and a priority of 4.

7.5.9. The Shift-F6 Hot-Key

The Shift-F6 Hot-Key displays RT-message pool with messages contents. The first line of Figure 7.9 shows message number 1 from source 7 and destination 6 with a contents of "HELLO FATHER".

The columns displayed have the following meanings:

- *MSG*: The message queue entry number in the *MRT_sm.mpool.mqe[]* array.
- *SRCE*: The message source process.
- *DEST*: The message destination process.
- *TYPE*: The message type described in *include/minix/const.h*.
- *MID*: The system wide message ID.

– *VT*: The VT assigned to the message.

– *01234567890123456789*: The message content.

```
MSG SRCE DEST TYPE -MID- VT 01234567890123456789
  1    7    6    2     5 -1 HELLO FATHER--|----|
  2   -1   -1    0     0 -1 ----|----|----|----|
  3   -1   -1    0     0 -1 ----|----|----|----|
  4   -1   -1    0     0 -1 ----|----|----|----|
  5   -1   -1    0     0 -1 ----|----|----|----|
  6   -1   -1    0     0 -1 ----|----|----|----|
  7   -1   -1    0     0 -1 ----|----|----|----|
  8   -1   -1    0     0 -1 ----|----|----|----|
  9   -1   -1    0     0 -1 ----|----|----|----|
 10   -1   -1    0     0 -1 ----|----|----|----|
 11   -1   -1    0     0 -1 ----|----|----|----|
 12   -1   -1    0     0 -1 ----|----|----|----|
 13   -1   -1    0     0 -1 ----|----|----|----|
 14   -1   -1    0     0 -1 ----|----|----|----|
 15   -1   -1    0     0 -1 ----|----|----|----|
 16   -1   -1    0     0 -1 ----|----|----|----|
 17   -1   -1    0     0 -1 ----|----|----|----|
 18   -1   -1    0     0 -1 ----|----|----|----|
 19   -1   -1    0     0 -1 ----|----|----|----|
```

Figure 7.9: RT-Message Pool (Message Contents).

7.5.10. The Ctrl-F6 Hot-Key

The Ctrl-F6 Hot-Key displays MQ Status an Statistics. The columns displayed have the following meanings:

– *ID*: The MQ number in the *MRT_sm.msgQ[]* array.

– *SZ*: The MQ size (in MQEs).

– *FLAG*: The MQ status and policy flags.

– *BITM*: The MQ priority bitmap.

– *INQ*: The number of message enqueued.

– *MAX*: The maximun number of message enqueued.

– *OWN*: The MQ owner.

– *VT*: The assigned VT for the MQ.

– *DLVD*: The total number of messages delivered.

– *ENQD*: The total number of messages enqueued.

MESSAGE QUEUES STATUS AND STATISTICS

ID	SZ	FLAG	BITM	INQ	MAX	OWN	VT	DLVD	ENQD
0	4	1	0	1	2	6	-1	3	4
1	3	1	0	0	1	7	-1	2	2
2	0	0	0	0	0	-1	-1	0	0
3	0	0	0	0	0	-1	-1	0	0
4	0	0	0	0	0	-1	-1	0	0
5	0	0	0	0	0	-1	-1	0	0
6	0	0	0	0	0	-1	-1	0	0
7	0	0	0	0	0	-1	-1	0	0
8	0	0	0	0	0	-1	-1	0	0
9	0	0	0	0	0	-1	-1	0	0
10	0	0	0	0	0	-1	-1	0	0
11	0	0	0	0	0	-1	-1	0	0
12	0	0	0	0	0	-1	-1	0	0
13	0	0	0	0	0	-1	-1	0	0
14	0	0	0	0	0	-1	-1	0	0
15	0	0	0	0	0	-1	-1	0	0

Figure 7.10: Message Queue Status and Statistics.

Figure 7.10 shows that Message Queue with ID 0 have a size of 4 messages with one message enqueued. Its owner is process number 6 and it has enqueued 4 messages and delivered 3 messages.

7.5.11. The F8 Key

The F8 Key displays VT attributes, status and statistics.Figure 7.11 shows VT number 0 that is the emulated MINIX CLOCK interrupt with a period of 4 and action type 4 (MRT_ACT_IRQTRIG) with an action parameter 16 is owned by process -3 (the CLOCK Task) with priority 15. It has expired 37461 times and the last expiration was at tick 149844.

The columns displayed have the following meanings:

– *VT*: VT number.

– *PERIOD*: VT period in PIT ticks.

- *NEXTEXP*: Next VT expiration.
- *LIMIT*: A limit for VT expirations.
- *ACT*: The action of the VT.
- *PAR*: The VT parameter field.
- *NR*: Same as the VT number.
- *OWN*: The owner process of the VT.
- *PRTY*: The VT priority.
- *EXPIRE*: The number of VT expirations.
- *TIMESTAMP*: The last VT expiration timestamp.

-VT	--PERIOD--	-NEXTEXP--	LIMIT	ACT	PAR	NR	OWN	PRTY	EXPIRE	TIMESTAMP
0	4	0	0	4	16	0	-3	15	37461	149844
1	0	0	0	0	0	1	-1	15	0	0
2	0	0	0	0	0	2	-1	15	0	0
3	0	0	0	0	0	3	-1	15	0	0
4	0	0	0	0	0	4	-1	15	0	0
5	0	0	0	0	0	5	-1	15	0	0
6	0	0	0	0	0	6	-1	15	0	0
7	0	0	0	0	0	7	-1	15	0	0
8	0	0	0	0	0	8	-1	15	0	0
9	0	0	0	0	0	9	-1	15	0	0
10	0	0	0	0	0	10	-1	15	0	0
11	0	0	0	0	0	11	-1	15	0	0
12	0	0	0	0	0	12	-1	15	0	0
13	0	0	0	0	0	13	-1	15	0	0
14	0	0	0	0	0	14	-1	15	0	0
15	0	0	0	0	0	15	-1	15	0	0

Bitmap of Expired Virtual Timers Priorities= 0000000000000000

Figure 7.11: Virtual Timer Attributes, Status and Statistics.

7.5.12. The Shift-F8 Hot-Key

The Ctrl-F8 Hot-Key displays RT-Virtual timer queues status. Figure 7.12 shows the Active Queue that have VT number 0 enqueued and 15 Expired Queues, one queue for each system priority. The last line shows the Free VT Queue that have 15 free VTs.

The columns displayed have the following meanings:

- *TYPE*: The VT queue type
- *PRTY*: The Priority of the VT queue
- *INQ*: The number of VTs enqueued
- *MAXINQ*: The maximun number of VTs enqueued.
- *LIST*: The lists of VTs.
- *ACTV*: It is the Active VT queue.
- *EXPD*: It is an Expired VT queue.
- *FREE*: It is the Free VT queue.

```
TYPE PRTY INQ MAXINQ LIST
ACTV  ALL   1      3 --->[0]->[NULL]
EXPD    0   0      0 --->[NULL]
EXPD    1   0      0 --->[NULL]
EXPD    2   0      0 --->[NULL]
EXPD    3   0      0 --->[NULL]
EXPD    4   0      1 --->[NULL]
EXPD    5   0      0 --->[NULL]
EXPD    6   0      0 --->[NULL]
EXPD    7   0      0 --->[NULL]
EXPD    8   0      0 --->[NULL]
EXPD    9   0      0 --->[NULL]
EXPD   10   0      0 --->[NULL]
EXPD   11   0      0 --->[NULL]
EXPD   12   0      0 --->[NULL]
EXPD   13   0      0 --->[NULL]
EXPD   14   0      0 --->[NULL]
EXPD   15   0      1 --->[NULL]
FREE NONE  15     16 --->[12]->[13]->[14]->[15]->[1]->[2]->[3]->[4]->[5]->[6]->[
7]->[8]->[9]->[10]->[11]->[NULL]
```

Figure 7.12: RT-Virtual Timer Queues.

7.6. The Modified *ps* Command

The *ps* command reports information about active processes. It was modified to support new options to show processes statistics and attributes related to RT-processing. The columns have the same meanings of fields explained in previous sections.

7.6.1. The –A Option

This option shows process attributes related to RT-processing (see Figure 7.13).

```
# ps -A
PROC -PID- FLAGS PRTY BASE PERIOD LIMIT DEAD WDOG RCVF SNDT CMD
 -11     0     0   15   15      0     0    0   -1 -111 -111 TTY
 -10     0     0   15   15      0     0    0   -1 -111 -111 WINCH
  -9     0     0   15   15      0     0    0   -1 -111 -111 MRTTASK
  -8     0     0   15   15      0     0    0   -1 -111 -111 SYN_AL
  -7     0     0   15   15      0     0    0   -1 -111 -111 IDLE
  -6     0     0   15   15      0     0    0   -1 -111 -111 PRINTER
  -5     0     0   15   15      0     0    0   -1 -111 -111 FLOPPY
  -4     0     0   15   15      0     0    0   -1 -111 -111 MEMORY
  -3     0     0   15   15      0     0    0   -1 -111 -111 CLOCK
  -2     0     0   15   15      0     0    0   -1 -111 -111 SYS
  -1     0     0   15   15      0     0    0   -1 -111 -111 HARDWAR
   0     0     0   15   15      0     0    0   -1 -111 -111 MM
   1     0     0   15   15      0     0    0   -1 -111 -111 FS
   2     1     0   15   15      0     0    0   -1 -111 -111 INIT
   3    26     0   15   15      0     0    0   -1 -111 -111 -sh
   4    17     0   15   15      0     0    0   -1 -111 -111 update
   5    27     0   15   15      0     0    0   -1 -111 -111 getty
   6    36     0    4    4      0     0    0   -1 -111 -111 ./mrttest8c
   7    37  4002    4    4      0     0    0   -1    6 -111 ./mrttest8c
   8    30     0   15   15      0     0    0   -1 -111 -111 -sh
   9    42     0   15   15      0     0    0   -1 -111 -111 ps -A
```

Figure 7.13: *ps* Command with –A Option.

7.6.2. The –S Option

This option shows process statistics related to RT-processing. Figure 7.14 shows process number 6 named *mrttest8c* with PID 36 has been scheduled 12 times, the last schedule at time 22314 ticks since the last *mrt_RTstart()/mrt_restart()* System Call invocation.

```
# ps -S
PROC -PID- SCHEDS MDL TSTAMP MAXLAT MINLAX SENT RCVD CMD
 -11     0      0   0      0      0      0    0    0 TTY
 -10     0      0   0      0      0      0    0    0 WINCH
  -9     0      0   0      0      0      0    0    0 MRTTASK
  -8     0      0   0      0      0      0    0    0 SYN_AL
  -7     0      0   0      0      0      0    0    0 IDLE
  -6     0      0   0      0      0      0    0    0 PRINTER
  -5     0      0   0      0      0      0    0    0 FLOPPY
  -4     0      0   0      0      0      0    0    0 MEMORY
  -3     0      0   0      0      0      0    0    0 CLOCK
  -2     0      0   0      0      0      0    0    0 SYS
  -1     0      0   0      0      0      0    0    0 HARDWAR
   0     0      0   0      0      0      0    0    0 MM
   1     0      0   0      0      0      0    0    0 FS
   2     1      0   0      0      0      0    0    0 INIT
   3    26      0   0      0      0      0    0    0 -sh
   4    17      0   0      0      0      0    0    0 update
   5    27      0   0      0      0      0    0    0 getty
   6    36     12   0  22314      0      0   10    0 ./mrttest8c
   7    37     11   0  22314      0      0    0   10 ./mrttest8c
   8    30      0   0      0      0      0    0    0 -sh
   9    44      0   0      0      0      0    0    0 ps -S
```

Figure 7.14: *ps* Command with –S Option.

7.7. The *mrtstatus* Command

The *mrtstatus* command provides additional information related to RT-processing. As the *ps* command, it uses the character special files */dev/mem* and */dev/kmem* MINIX devices that map system and kernel memory to files. Both devices, the */dev/null,* and RAM disks are supported by the *MEMORY* Task.

The MEMORY Task offers several IOCTL operations on memory devices. The *MIOCGPSINFO* is used by the MM and the FS to gather information about the addresses of their process tables needed by the *ps* command.

MINIX4RT adds a new IOCTL operation to the MEMORY Task named *MIOCGMRTINFO* to gather information about the addresses of main RT-kernel tables and data structures needed by the *mrtstatus* command. The *mrtstatus* command reports:

- RT-kernel constants.
- System-wide statistics.
- Interrupt descriptor statistics.
- Virtual Timer statistics.

– Messages statistics.

– Interrupt Queue statistics.

– Virtual Timer statistics.

– Message Queues statistics.

The several columns of the reports have the same meanings of fields explained in previous sections, other will be described in the following subsections.

7.7.1. The –s Option

This option (default) shows System-wide statistics related to RT-processing (see Figure 7.15).

```
# mrtstatus
flags          = 1
virtual_PIC    = BF20
PIT_latency    = 16
PIT_latch      = 5965
tickrate       = 200
harmonic       = 4
refresh        = 200
scheds         = 23
messages       = 10
interrupts     = 34935
ticks          = 31485
highticks      = 0
idlemax        = 8087325
idlelast       = 8080760
#
```

Figure 7.15: ***mrtstatus*** **Command with –s Option.**

7.7.2. The –i Option

This option shows interrupt descriptors status and statistics (see Figure 7.16).

```
# ./mrtstatus -i
IRQ PER TSK WDOG PTY TYPE COUNT MSHOWER MSDL TIMESTAMP MAXL REENT NAME
  0   0  -3   -1   0    1 34740       0    0     34740    0     1 RT-CLOCK
  1   1  -1   -1   F    0    24       0    0      2301    0     1 KEYBOARD
  2   0  -1   -1   F    0     0       0    0         0    0     0 PIC-CASCADE
  3   1  -1   -1   F    0     0       0    0         0    0     0 SERIAL-PORT-2
  4   1 -11   -1   7    1  3642       0    0     34736    0     1 RT-RS232
  5   0  -1   -1   F    0     0       0    0         0    0     0 FREE
  6   1  -1   -1   F    0     0       0    0         0    0     0 DISKETTE
  7   1  -1   -1   F    0     0       0    0         0    0     0 PARALLEL-PORT
  8   0  -1   -1   F    0     0       0    0         0    0     0 FREE
  9   0  -1   -1   F    0     0       0    0         0    0     0 FREE
 10   0  -1   -1   F    0     0       0    0         0    0     0 FREE
 11   0  -1   -1   F    0     0       0    0         0    0     0 FREE
 12   0  -1   -1   F    0     0       0    0         0    0     0 FREE
 13   0  -1   -1   F    0     0       0    0         0    0     0 FREE
 14   1  -1   -1   F    0   222       0    0     33976    0     1 HARD-DISK
 15   0  -1   -1   F    0     0       0    0         0    0     0 FREE
 16   4  -3   -1   F    7  8685       0    0     34740    0     0 MINIX-CLOCK
#
```

Figure 7.16: *mrtstatus* Command with –i Option.

7.7.3. The –t Option

This option shows VTs status and statistics (see Figure 7.17). The columns displayed have the following meanings:

- *VT*: VT number.
- *PERIOD*: VT period in PIT ticks.
- *NEXTEXP*: Next VT expiration.
- *LIMIT*: A limit for VT expirations.
- *ACT*: The action of the VT.
- *PAR*: The VT parameter field.
- *NR*: Same as the VT number.

– *OWN*: The owner process of the VT.

– *PRTY*: The VT priority.

– *EXPIRE*: The number of VT expirations.

– *TIMESTAMP*: The last VT expiration timestamp.

```
# ./mrtstatus -t
-VT --PERIOD-- -NEXTEXP-- LIMIT ACT PAR NR OWN PRTY EXPIRE TIMESTAMP
  0         4          0     0   4  16  0  -3   15   9603     38412
  1         0          0     0   0   0  1  -1   15      0         0
  2         0          0     0   0   0  2  -1   15      0         0
  3         0          0     0   0   0  3  -1   15      0         0
  4         0          0     0   0   0  4  -1   15      0         0
  5         0          0     0   0   0  5  -1   15      0         0
  6         0          0     0   0   0  6  -1   15      0         0
  7         0          0     0   0   0  7  -1   15      0         0
  8         0          0     0   0   0  8  -1   15      0         0
  9         0          0     0   0   0  9  -1   15      0         0
 10         0          0     0   0   0 10  -1   15      0         0
 11         0          0     0   0   0 11  -1   15      0         0
 12         0          0     0   0   0 12  -1   15      0         0
 13         0          0     0   0   0 13  -1   15      0         0
 14         0          0     0   0   0 14  -1   15      0         0
 15         0          0     0   0   0 15  -1   15      0         0
#
```

Figure 7.17: *mrtstatus* Command with –t Option.

7.7.4. The –m Option

This option shows messages status and statistics (see Figure 7.18). The columns displayed have the following meanings:

– *MSG*: Message number.

– *SPID/SNBR*: Source PID and Source Number (RT-PID).

– *DPID/DNBR*: Destination PID and Destination Number (RT-PID).

– *TYPE*: Message Type.

– *MID*: Message ID.

– *SEQNO*: Sequence Number.

– *TIMESTAMP*: Message timestamp.

– *PRI*: Message Priority.

– *DLINE*: Message Sender Deadline.

– *LAXTY*: Message Sender Laxity.

```
# ./mrtstatus -M
MSG SPID/SNBR DPID/DNBR TYPE -MID- SEQNO TIMESTAMP PRI DLINE LAXTY
  0   -1/  -1   -1/  -1    0     0     0         0   0     0     0
  1   -1/  -1   -1/  -1    0     0     0         0   0     0     0
  2   -1/  -1   -1/  -1    0     0     0         0   0     0     0
  3   -1/  -1   -1/  -1    0     0     0         0   0     0     0
  4   -1/  -1   -1/  -1    0     0     0         0   0     0     0
  5   -1/  -1   -1/  -1    0     0     0         0   0     0     0
  6   -1/  -1   -1/  -1    0     0     0         0   0     0     0
  7   -1/  -1   -1/  -1    0     0     0         0   0     0     0
  8   -1/  -1   -1/  -1    0     0     0         0   0     0     0
  9   -1/  -1   -1/  -1    0     0     0         0   0     0     0
 10   -1/  -1   -1/  -1    0     0     0         0   0     0     0
 11   -1/  -1   -1/  -1    0     0     0         0   0     0     0
 12   -1/  -1   -1/  -1    0     0     0         0   0     0     0
 13   -1/  -1   -1/  -1    0     0     0         0   0     0     0
 14   -1/  -1   -1/  -1    0     0     0         0   0     0     0
 25   10/  60   12/  61    3    26     1     83788   4     0     0
 16   -1/  -1   -1/  -1    0     0     0         0   0     0     0
 17   -1/  -1   -1/  -1    0     0     0         0   0     0     0
 18   -1/  -1   -1/  -1    0     0     0         0   0     0     0
 19   -1/  -1   -1/  -1    0     0     0         0   0     0     0
 20   -1/  -1   -1/  -1    0     0     0         0   0     0     0
```

Figure 7.18: *mrtstatus* Command with –m Option.

7.7.5. The –c Option

This option shows the values of RT-kernel constant (see Figure 7.19).

The constants showed have the following meanings:

– *NR_VTIMERS*: Number of system Virtual Timers.

– *NR_PRTY*: Number of priority levels.

– *NR_IRQ_VECTORS*: Number of IRQ vectors (and Hardware interrupt descriptors).

– *NR_IRQ_SOFT*: Number of Software interrupt descriptors.

– *NR_MSGQ*: Number of Message Queues (It limits the number of RT-processes)

– *NR_MESSAGES*: Number of system messages.

```
# ./mrtstatus -c
NR_VTIMERS      = 16
NR_PRTY         = 16
NR_IRQ_VECTORS  = 16
NR_IRQ_SOFT     = 16
NR_MSGQ         = 16
NR_MESSAGES     = 32
#
```

Figure 7.19: *mrtstatus* Command with –c Option.

7.7.6. The –I Option

This option shows interrupt queues status and statistics (see Figure 7.20).

The columns displayed have the following meanings:

– *PRI*: The priority of the interrupt descriptor.

– *INQ*: The current number of descriptors enqueued.

– *PEND*: The number of pending interrupt descriptors.

Each bit set in the bitmap displayed represents an interrupt queue with at least one descriptor triggered (service pending).

```
# ./mrtstatus -I
Interrupt Pending Queues bitmap =  0000000000000000
PRI INQ PEND
  0   1    0
  1   0    0
  2   0    0
  3   0    0
  4   0    0
  5   0    0
  6   0    0
  7   1    0
  8   0    0
  9   0    0
 10   0    0
 11   0    0
 12   0    0
 13   0    0
 14   0    0
 15   6    0
#
```

Figure 7.20: mrtstatus Command with –I Option.

7.7.7. The –T Option

This option shows VTs queues status and statistics (see Figure 7.21).

The columns displayed have the following meanings:

- *TYPE*: The VT queue type
- *PRTY*: The Priority of the VT queue
- *INQ*: The number of VTs enqueued
- *MAXINQ*: The maximun number of VTs enqueued.
- *ACTV*: It is the Active VT queue.
- *EXPD*: It is an Expired VT queue.
- *FREE*: It is the Free VT queue.

```
# ./mrtstatus -M
ID  SZ FLAG INQ MAX OWN DLVD ENQD
 0  1     0   0   0   6    0    0
 1 10     0   0   1   7    3    3
 2  0     0   0   0  -1    0    0
 3  0     0   0   0  -1    0    0
 4  0     0   0   0  -1    0    0
 5  0     0   0   0  -1    0    0
 6  0     0   0   0  -1    0    0
 7  0     0   0   0  -1    0    0
 8  0     0   0   0  -1    0    0
 9  0     0   0   0  -1    0    0
10  0     0   0   0  -1    0    0
11  0     0   0   0  -1    0    0
12  0     0   0   0  -1    0    0
13  0     0   0   0  -1    0    0
14  0     0   0   0  -1    0    0
15  0     0   0   0  -1    0    0
#
```

Figure 7.21: *mrtstatus* Command with –T Option.

7.7.8. The –M Option

This option shows MQ status and statistics (see Figure 7.22).

```
# ./mrtstatus -T
TYPE PRTY INQ MAXINQ
ACTV  ALL   2      3
EXPD    0   0      0
EXPD    1   0      0
EXPD    2   0      0
EXPD    3   0      0
EXPD    4   0      1
EXPD    5   0      0
EXPD    6   0      0
EXPD    7   0      0
EXPD    8   0      0
EXPD    9   0      0
EXPD   10   0      0
EXPD   11   0      0
EXPD   12   0      0
EXPD   13   0      0
EXPD   14   0      0
EXPD   15   0      1
FREE NONE  14     16
#
```

Figure 7.22: *mrtstatus* Command with –M Option.

The columns displayed have the following meanings:

- *ID*: The MQ number in the *MRT_sm.msgQ[]* array.
- *SZ*: The MQ size (in MQEs).
- *FLAG*: The MQ status and policy flags.
- *INQ*: The number of message enqueued.
- *MAX*: The maximun number of message enqueued.
- *OWN*: The MQ owner.
- *DLVD*: The total number of messages delivered.
- *ENQD*: The total number of messages enqueued.

8. CONCLUSIONS AND FUTURE WORKS

8.1. Conclusions

MINIX has proved to be a feasible testbed for OS development and extensions that could be easily added to it. In a similar way, MINIX4RT has an architecture that can be used as a starting point for adding RT-services.

In spite of it was designed for an academic environment, it can be optimized for production systems even in embedded systems. Its Virtual Machine architecture, code readability, MINIX compatibility and the similarities of several of its algorithms and data structures helps to minimize the understanding time of its source code. Those characteristics make it suitable for course assignments and RT-project developments as the support of Rate Monotonic/Deadline Monotonic scheduling algorithms, Sporadic/Deferable Servers implementation and performance evaluation tests.

MINIX4RT microkernel has basic features as Interrupt Management, Process Management, Time Management, Real-Time IPC and Statistics gathering making it a good choice to conduct coding experiences. Device-drivers writers have at their disposal several flavors of interrupt handling as Event-Driven, Timer-Driven, Software and Non-Real-Time Interrupt Service Routines execution.

8.2. Future Works

MINIX4RT development does not finish with this thesis. There are other planed projects for its improvement as:

- *RT-FIFOs*: They are mechanisms equivalent to RTLinux RT-FIFOs that permit RT-processes communicate with NRT-processes.

– *RT-Semaphores*: They are known mechanisms in OSs used for synchronization and mutual exclusion among RT-processes.

– *Non Periodic Time Management*: Often, OSs increase the Timer frequency when they need better time resolution. This approach increase the timer interrupt overhead because it is executed more frequently. Sometimes, a more efficient strategy for better timer resolution is to use the ONE_SHOT mode for programming the PIT instead the SQUARE_WAVE mode. In this mode, on each Timer interrupt the PIT is programmed to generate an interrupt at the time of the the next VT expiration time.

– *Update MINIX4RT as a branch of MINIX3*: MINIX Version 3 offers a lot of improvements over previous versions and MINIX4RT will benefit of it's new features.

– *POSIX 1003.1b compatibility*: To support standard functions that RT-applications need, such as enhanced IPC, scheduling and memory management control, asynchronous I/O operations and file synchronization.

References

[1] Compaq Computer Corporation, "*Tru64 UNIX:Guide to Realtime Programming*", August 2000, http://www.helsinki.fi/atk/unix/dec_manuals/DOC_51A/HTML/ARH9TBTE/TITLE.HTM.

[2] Tanenbaum Andrew S., Woodhull Albert S., "*Sistemas Operativos: Diseño e Implementación 2da Edición*", ISBN 9701701658, Editorial Prentice-Hall , 1999.

[3] Paul Ashton, Carl Cerecke,Craig McGeachie, Stuart Yeates, "*Use of interaction networks in teaching Minix*" Technical Remailbox . COSC 08/95, Dept. of Computer Science . University of Canterbury,1995.

[4] Paul Ashton, Daniel Ayers, Peter Smith. "*SunOS MINIX: A tool for use in Operating System laboratories*", Technical Remailbox, Australian Computer Science Communications, 16(1): 259-269, 1994.

[5] Stephen J Hartley, "*More Experience with MINIX in Operating System lab*", available online at ftp://ftp.mcs.drexel.edu/pub/shartley/minix.PO.gz.

[6] Victor Yodaiken, "*Cheap Operating System Research and Teaching with Linux*", available online at http://citeseer.ist.psu.edu/75556.html.

[7] RTnet-Hard Real-Time Networking for Linux/RTAI, available online at http://www.rts.uni-hannover.de/rtnet/index.html.

[8] Rether: A Real-Time Ethernet Protocol, http://www.ecsl.cs.sunysb.edu/rether/.

[9] Victor Yodaiken, Michael Barabanov, "*A Real-Time Linux*", ", Proceedings of Linux Applications Development and Deployment Conference (USELINUX), January, 1997, available online at http://rtlinux.cs.nmt.edu/.

[10] Takuro Kitayama, Tatsuo Nakajima, and Hideyuki Tokuda, "*RT-IPC: An IPC Extension for Real-Time Mach*", School of Computer Science, Carnegie Mellon University, Japan Advanced Institute of Science and Technology.

[11] QNX Software Systems Ltd. 2002 – "*QNX Neutrino Realtime Operating System – System Architecture*", http://www.mikecramer.com/Qnx/momentics_nc_docs/neutrino/sys_arch/kernel.html.

[12] Pablo J. Rogina - Gabriel Wainer., "*New Real-Time Extensions to the MINIX operating system*", Proc. of 5 th Int. Conference on Information Systems Analysis and Synthesis (ISAS'99),August, 1999.

[13] Gabriel A. Wainer, "*Implementing Real-Time services in MINIX*", ACM Operating Systems Review, July 1995.

[14] Mark Russinovich ,"*Inside NT's Interrupt Handling*", Windows & .NET Magazine, November 1997, available on line at http://www.winntmag.com/Articles/Print.cfm?ArticleID=298.

[15] Daniel P. Bovet, Marco Cesati, "*Understanding the Linux Kernel Second Edition, 2003*" - O'Reilly – 2003.

[16] Keith Loepere, "*Mach 3 Kernel Principles*", Open Software Foundation and Carnegie Mellon University, July 15, 1992.

[17] Jochen Liedtke, "*On µ-Kernel Construction*", In Proceedings of the Fifteenth ACM Symposium on Operating System Principles (Copper Mountain Resort, CO., Dec. 3-6). ACM Press, New York, NY, 1995, pp. 237-250.

[18] Intel Corporation, Intel Architecture Software Developer's Manual Volume 3: System Programming Guide, 1997.

[19] Kevin Jeffay, Donald L. Stone , "*Accounting for Interrupt Handling Costs in Dynamic Priority Task Systems*", University of North Carolina at Chapel Hill, Department of Computer Science.

[20] INtime Interrupt Latency Report, "*Measured Interrupt Response Times*", November, 1998, Technical Paper available on line at http://www.profimatics.de/products/intime/manuals/intime.interrupt.latency.report.pdf.

[21] Daniel I. Katcher, Hiroshi Arakawa, Jay K. Strosnider, "*Engineering and Analisys of Fixed Priority Schedulers*", Software Engineering, 1993, volume 19, number 9, pages 920-934.

[22] Matthew B. Ragen, "*Real-Time Systems With Microsoft Windows NT*", Microsoft Corporation -April 1995, http://www.theragens.com/misc/MR%20-%20Windows%20NT%20Real-Time.htm.

[23] Herman Bruyninckx , "*Real-Time and Embedded Guide*", K.U.Leuven, Mechanical Engineering, Leuven, Belgium.

[24] Real-Time Systems: (Shin) http://cslab.snu.ac.kr/course/rts03/notes/rt_c2mod.ppt.

[25] Gregory Bollella, Kevin Jeffay ,"*Support For Real-Time Computing Within General Purpose Operating Systems*", 1995.

[26] Victor Yodaiken, "*The RTLinux Manifesto*", Department of Computer Science New Mexico Institute of Technology.

[27] Victor Yodaiken, "*Against Priority Inheritance*", Fsmlabs Technical Report, available on line at http://www.fsmlabs.com/articles/inherit/inherit.pdf, June 25, 2002.

[28] "*About UNIX and Real-Time Scheduling*", http://www.pcengines.ch/schedule.htm., 1989.

[29] C. L. Liu and J. Layland. "*Scheduling algorithms for multiprogramming in a hard real-time environment*", Journal of the ACM, 10(1), 1973.

[30] N. C. Audsley A. Burns M. F. Richardson A. J. Wellings "*Hard Real-Time Scheduling: The Deadline-Monotonic Approach*", Department of Computer Science, University of York, York, YO1 5DD, England.

[31] J. Stankovic, M. Spuri, K. Ramamritham, and G. Buttazzo, "*Deadline Scheduling For Real-Time Systems: EDF and Related Algorithms*". Norwell, MA: Kluwer, 1998.

[32] Aloysius Ka-Lau Mok. "*Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment.*" MIT LCS Technical Report 297, Department of Computer Science, MIT, May, 1983.

[33] TICS Realtime, "*Different Timing Mechanisms and How They are Used*" , available on line at http://www.concentric.net/~Tics/tics1196a.htm.

[34] George Varghese, Tony Lauck, "*Hashed and Hierarchical Timing Wheels: Efficient Data Structures for Implementing a Timer Facility*", IEEE\slash ACM Transactions on Networking. (1996), available on line at http://citeseer.nj.nec.com/cache/papers/cs/1218/http:zSzzSzwww.ccrc.wustl.eduzSz~varghesezSzPAPERSzSzt wheel.pdf/varghese96hashed.pdf.

[35] Robert Hill, Balaji Srinivasan, Shyam Pather, Douglas Niehaus, "*Temporal Resolution and Real-Time Extensions to Linux*.", ITTC Technical Report ITTC-FY98-11510-03, University of Kansas, 1998.

[36] Jochen Liedtke , "*Improving IPC by Kernel Design*", 14th ACM Symposium on Operating System Principles (SOSP) 5th-8th December, 1993, Asheville, North Carolina.

[37] Mark W. Borger, Ragunathan Rajkumar. "*Implementing Priority Inheritance Algorithms in an Ada Runtime System*", Technical Remailbox . CMU/SEI-89-TR-15. ESD-TR-89-23. April 1989. Software Engineering Institute Carnegie Mellon University.

[38] Sha, L., Lehoczky, J.P., and Rajkumar, R. "*Priority Inheritance Protocols: An Approach to Real-Time Synchronization*". Tech. Rept. CMU-CS-87-181, Carnegie Mellon University, Computer Science Department, 1987.

[39] Bruno Dutertre, "*The Priority Ceiling Protocol: Formalization and Analysis Using PVS*", System Design Laboratory SRI International, Menlo Park, CA 94025, November 8, 1999.

Appendix A: RT-SYSTEM CALLS AND RT-KERNEL CALLS REFERENCE

A.1. System Calls Reference

A.1.1. mrt_RTstart

```
NAME
     mrt_RTstart - Starts the Real-Time Processing Mode.

SYNOPSIS
     #include <unistd.h>

     int mrt_RTstart(int Harmonic,int Refresh)

ARGUMENTS

     Harmonic:The Harmonic Number of the MINIX timer frequency (HZ = 50 Hz).
     Refresh: The idle refresh counter in timer ticks.

DESCRIPTION
    mrt_RTstart starts the Real-Time Processing Mode and configure the Timer to
generate  Harmonic  *  HZ  interrupts  by  second.  The  idle  process statistics
are renewed each Refresh ticks.

     This call is restricted to the super-user and must be executed by  a  Non
Realtime process.

RETURN VALUE
     Upon successful completion, a value of 0 (OK) is returned.  Otherwise,  a
negative value is returned to indicate an error.
```

```
ERRORS
     mrt_RTstart will fail and processing mode will be  unchanged  if  one  or more
of the following are true:

     [E_MRT_BADHARM]A bad value has been specified for the Harmonic argument.

     [E_MRT_BADRFSH]A bad value has been specified for the Refresh argument.

     [E_MRT_RTACTIVE]The system is already in Real-Time Processing Mode.

SEE ALSO
     mrt_RTstop(2).
```

A.1.2. mrt_RTstop

```
NAME
     mrt_RTstop - Stops the Real-Time processing mode.

SYNOPSIS
     #include <unistd.h>

     int mrt_RTstop( void )

ARGUMENTS
     None No Arguments.

DESCRIPTION
     mrt_RTstop Stops the Real-Time Processing Mode.

     This call is restricted to the super-user and must be executed by  a  Non
Realtime process.

RETURN VALUE
     Upon successful completion, a value of 0 (OK) is returned.  Otherwise,  a
negative value is returned to indicate an error.

ERRORS
     mrt_RTstop will fail  and  processing  mode  will  be  unchanged  if  the
following is true:

     [E_MRT_RTACTIVE] A Real-Time process is still running.
SEE ALSO
     mrt_RTstart(2).

NOTES
     Kill all RT-process before call mrt_RTstop.
```

A.1.3. mrt_clrpstat

```
NAME
     mrt_clrpstat - Clears all RT-processing statistics of a specified process.

SYNOPSIS
     #include <unistd.h>

     int mrt_clrpstat(pid_t Pid)

ARGUMENTS

     Pid: The PID number of the RT-process.
```

```
DESCRIPTION
     mrt_clrpstat Clears all RT-processing statistics of a specified process. This
call is restricted to the super-user.  It must be executed by a NRT-process.

RETURN VALUE
     Upon successful completion, a value of 0 (OK) is returned.  Otherwise,  a
negative value is returned to indicate an error.

ERRORS
     mrt_clrpstat will fail if one or more of the following are true:

     [E_BAD_PROC]The RT-process of the specified Pid does not exist.

SEE ALSO
     mrt_getpstat(2).
```

A.1.4. mrt_getiattr

```
NAME
     mrt_getiattr - Gets the RT-processing Attributes of an IRQ descriptor.

SYNOPSIS
     #include <unistd.h>

     int  mrt_getiattr(int irq, mrt_irqattr_t *attr)

ARGUMENTS

     irq: The IRQ number

     attr: An IRQ Attributes descriptor with the RT-processing for the IRQ that
will be filled by the system call.

DESCRIPTION
     mrt_getiattr Gets the RT-processing Attributes of an IRQ descriptor.   On
successful  return of the system call, the following mrt_irqattr_t fields will be
filled by the system:
     period:  The processing period for a TD- interrupt handler in  RT-ticks units.
     task: The number  of  the  RT-task  to  send  a  message  for  deferred
processing.
     watchdog: The ID number of a watchdog RT-process.
     priority: The priority of the handler
     irqtype: The type of handler.
     name: The reference name for the handler.
     This call is restricted to the super-user.  It must be executed by a NRT-
process.

RETURN VALUE
     Upon successful completion, a value of 0 (OK) is returned.  Otherwise,  a
negative value is returned to indicate an error.

ERRORS
     mrt_getiattr will fail if one or more of the following are true:

     [EINVAL]Invalid mrt_irqattr_t pointer.

     [E_MRT_BADIRQ] Invalid value specified for irq number in irq.

SEE ALSO
     mrt_setiattr(2).
```

A.1.5. mrt_getiint

```
NAME
     mrt_getiint - Gets the RT-processing Internal use fields values of an IRQ
descriptor.

SYNOPSIS
     #include <unistd.h>

     int  mrt_getiint(int irq, mrt_irq_int_t *stat)

ARGUMENTS

     irq: The IRQ number.

     iint: An IRQ internal fields descriptor with  the  RT-processing  internal
fields  values of the specified IRQ descriptor will be filled by the system.

DESCRIPTION
     mrt_getiint Gets the RT-processing Internal use fields values of  an  IRQ
descriptor. On successful  return  of  the  system call, the following
     mrt_irq_int_t fields will be filled by the system:
       irq:  The number of the irq.
       harmonic:  The value of the RT_sv.harmonic system value when the  systems
change to Real Time mode. (only for Real Time Timer Driven descriptors).
       vimer: The number of the Virtual Timer used for Real Time Timer Driven
descriptors.
       flags:  Some status flags.
     This call is restricted to the super-user.  It must be executed by a NRT-
process.

RETURN VALUE
     Upon successful completion, a value of 0 (OK) is returned.  Otherwise,  a
negative value is returned to indicate an error.

ERRORS
     mrt_getiint will fail if one or more of the following are true:

     [EINVAL] Invalid mrt_irq_int_t pointer.

     [E_MRT_BADIRQ] Invalid value specified for irq number in irq.

SEE ALSO
     mrt_getiarg(2).
```

A.1.6. mrt_getistat

```
NAME
     mrt_getistat - Gets the RT-processing Statistics of an IRQ descriptor.

SYNOPSIS
     #include <unistd.h>

     int  mrt_getistat(int irq, mrt_irqstat_t *stat)

ARGUMENTS

     irq: The IRQ number.

     stat: An IRQ statistics descriptor with the  RT-processing  statistics  of the
specified IRQ descriptor will be filled by the system.

DESCRIPTION
```

```
     mrt_getistat Gets the RT-processing Statistics of an IRQ descriptor.   On
successful  return of the system call, the following mrt_irqstat_t fields will be
filled by the system:

       count:  An interrupt counter.
       maxrain:  The maximun number of  interrupts  into  a  period  of  Timer
Driven handler.
       maxrain:  The number of Missed DeadLines of the handler.
       timestamp: The last interrupt timestamp.
       maxlat: The maximun (approximate) latency of the handler.

     This call is restricted to the super-user. It must be executed by a NRT-
process.

RETURN VALUE
     Upon successful completion, a value of 0 (OK) is returned.  Otherwise,  a
negative value is returned to indicate an error.

ERRORS
     mrt_getistat will fail if one or more of the following are true:

     [EINVAL] Invalid mrt_irqstat_t pointer.

     [E_MRT_BADIRQ] Invalid specified irq number.

SEE ALSO
     mrt_getiattr(2).
```

A.1.7. mrt_getpattr

```
NAME
     mrt_getpattr - Gets the RT-processing attributtes  of  the  calling  NRT-
process

SYNOPSIS
     #include <unistd.h>

     int mrt_getpattr(pid_t Pid, mrt_pattr_t *P_attrib)

ARGUMENTS

     Pid: The PID number of the RT-process.

     P_attrib: A pointer to a mrt_pattr_t data structure to store the process's RT-
rocessing attributes.

DESCRIPTION
     mrt_getpattr Gets the RT-Processing Attributes of the specified process. This
call is restricted to the super-user. It must be executed by a NRT-process.

RETURN VALUE
     Upon successful completion, a value of 0 (OK) is returned.  Otherwise,  a
negative value is returned to indicate an error.

ERRORS
     mrt_getpattr will fail if one or more of the following are true:

     [EINVAL]Invalid mrt_pattr_t pointer.

SEE ALSO
     mrt_setpattr(2).
```

A.1.8. mrt_getpint

```
NAME
     mrt_getpint - Gets the RT-processing internal variables  of  a  specified
process.

SYNOPSIS
     #include <unistd.h>

     int mrt_getpint(pid_t Pid, mrt_pint_t *p_int)

ARGUMENTS

     Pid: The PID number of the RT-process.

     p_int: A pointer  to  a  mrt_pint_t  data  structure  to  store  RT-process
internal variables.

DESCRIPTION
     mrt_getpint Gets the RT-processing  internal  variables  of  a  specified
process. This call is restricted to the super-user.  It must be executed by a NRT-
process.

RETURN VALUE
     Upon successful completion, a value of 0 (OK) is returned.  Otherwise,  a
negative value is returned to indicate an error.

ERRORS
     mrt_getpint will fail if one or more of the following are true:

     [EINVAL] Invalid mrt_pint_t pointer.

     [E_BAD_PROC] The RT-process of the specified Pid does not exist.

SEE ALSO
     mrt_getpstat(2).
```

A.1.9. mrt_getpstat

```
NAME
     mrt_getpstat - Gets the RT-processing statistics of a specified process.

SYNOPSIS
     #include <unistd.h>

     int mrt_getpstat(pid_t Pid, mrt_pstat_t *p_stats)

ARGUMENTS

     Pid: The PID number of the RT-process.

     p_stats: A pointer to  a  mrt_pstat_t  data  structure  to  store  RT-process
statistics.

DESCRIPTION
     mrt_getpstat Gets the RT-processing statistics of a specified RT-process. This
call is restricted to the super-user.  It must be executed by a NRT-process.

RETURN VALUE
     Upon successful completion, a value of 0 (OK) is returned.  Otherwise,  a
negative value is returned to indicate an error.

ERRORS
     mrt_getpstat will fail if one or more of the following are true:
```

```
     [EINVAL] Invalid mrt_pstat_t pointer.

     [E_BAD_PROC] The RT-process of the specified Pid does not exist.

SEE ALSO
     mrt_getpint(2).  mrt_clrpstat(2).
```

A.1.10.mrt_getsstat

```
NAME
     mrt_getsstat - Gets the System Wide RT-processing statistics.

SYNOPSIS
     #include <unistd.h>

     int mrt_getsstat(mrt_sysstat_t *stat)

ARGUMENTS

     stat A pointer to a mrt_sysstat_t data structure to store the system  RT-
processing statistics.

DESCRIPTION
     mrt_getsstat Gets the System Wide RT-processing statistics.

     This call is restricted to the super-user. It must be executed by a NRT-
process.

RETURN VALUE
     Upon successful completion, a value of 0 (OK) is returned.  Otherwise,  a
negative value is returned to indicate an error.

ERRORS
     mrt_getsstat will fail if one or more of the following are true:

     [EINVAL] Invalid mrt_sysstat_t pointer.

SEE ALSO
     mrt_getsval(2).
```

A.1.11.mrt_getsval

```
NAME
     mrt_getsval - Gets the System Wide RT-processing values.

SYNOPSIS
     #include <unistd.h>

     int mrt_getsval(mrt_sysval_t *val)

ARGUMENTS

     val  A pointer to a mrt_sysval_t data structure to store the  system  RT-
processing statistics.

DESCRIPTION
     mrt_getsval Gets the System Wide RT-processing values.

     This call is restricted to the super-user.  It must be executed by a NRT-
process.
```

```
RETURN VALUE
     Upon successful completion, a value of 0 (OK) is returned.  Otherwise,  a
negative value is returned to indicate an error.

ERRORS
     mrt_getsval will fail if one or more of the following are true:

     [EINVAL] Invalid mrt_sysval_t pointer.


SEE ALSO
     mrt_getsstat(2).
```

A.1.12.mrt_restart

```
NAME
     mrt_restart - Restart the Real-Time Processing Mode.

SYNOPSIS
     #include <unistd.h>

     int mrt_restart(int Harmonic,int Refresh)

ARGUMENTS

     Armonic: The Harmonic Number of the MINIX timer frequency (HZ=50 Hz).

     Refresh: The idle refresh counter in timer ticks.

DESCRIPTION
     mrt_restart Restarts the Real-Time  Processing  Mode  and  configure  the
Timer  to  generate  Harmonic  *  HZ interrupts by second.The idle process
statistics are renewed each Refresh ticks.

     This call is restricted to the super-user and must be executed by  a  Non
Realtime process.

RETURN VALUE
     Upon successful completion, a value of 0 (OK) is returned.  Otherwise,  a
negative value is returned to indicate an error.

ERRORS
     mrt_restart will fail and processing mode will be  unchanged  if  one  or more
of the following are true:

     [E_MRT_BADHARM] A bad value has been specified for the Harmonic argument.

     [E_MRT_BADRFSH] A bad value has been specified for the Refresh argument.

     [E_MRT_RTACTIVE] At least a RT-process is running in the system.

SEE ALSO
     mrt_RTstart(2).  mrt_RTstop(2).
```

A.1.13.mrt_setiattr

```
NAME
     mrt_setiattr - Sets the RT-processing Attributes of an IRQ descriptor.

SYNOPSIS
     #include <unistd.h>
```

```
     int  mrt_setiattr(int irq, mrt_irqattr_t *attr)

ARGUMENTS

     irq: The IRQ number.

     attr An IRQ Attributes descriptor with the RT-processing  Attributes  for the
IRQ.

DESCRIPTION
     mrt_setiattr Sets the RT-processing Attributes of an IRQ descriptor.  The
mrt_irqattr_t  data  structure  has  the  following  fields  that must be filled:
       period: The processing period for a TD- interrupt handler in  RT-ticks
units.
       task: The ID number of the RT-task  to  send  a  message  for  deferred
processing.
       watchdog: The ID number of a watchdog RT-process.
       priority: The priority of the handler.
       irqtype: The type of handler. It must an OR of the following bits:
             MRT_RTIRQ: for Real-Time handlers  (Otherwise  it  will  be  NRT-
handler).
             MRT_TDIRQ: for Timer-Driven handlers (Otherwise it will be  Event
Driven-IRQ handler).
             MRT_SOFTIRQ: for Software IRQ handlers (Otherwise it  will  be  a
Hardware IRQ handler).
       name: A reference name for the handler.

     This call is restricted to the super-user.  It must be executed by a NRT-
process.


RETURN VALUE
     Upon successful completion, a value of 0 (OK) is returned.  Otherwise,  a
negative value is returned to indicate an error.


ERRORS
     mrt_setiattr will fail if one or more of the following are true:

     [EINVAL] Invalid mrt_irqattr_t pointer

     [E_MRT_BADIRQ] Invalid value specified for irq number.

     [E_MRT_BADTASK] Invalid value specified for the TASK number in attr->task

     [E_MRT_BADWDOG] Invalid value specified for the watchdog PID in attr->watchdog

     [E_MRT_BADPRTY] Invalid value specified for the IRQ descriptor priority in
attr->priority

     [E_MRT_BADIRQT] Invalid value specified for the IRQ type in attr->irqtype

SEE ALSO
     mrt_getiattr(2).
```

A.1.14.mrt_setpattr

```
NAME
     mrt_setpattr - Sets the RT-processing attributtes  of  the  calling  NRT-
process.

SYNOPSIS
     #include <unistd.h>
```

```
     int mrt_setpattr(mrt_pattr_t *p_attrib)
```

ARGUMENTS

p_attrib: A pointer to a mrt_pattr_t data structure with the RT-processing attributes of the calling NRT-process. The field of the data structure are:

- flags: Real Time Flags.
- baseprty: Real Time Base priority.
- period: period in RT-ticks for Periodic Processes.
- limit: maximun number of process schedulings.
- deadline: process deadline.
- watchdog: Watchdog process.
- mq_size: Message Queue Size.
- mq_flags: Message Queue Policy Flags.

The Real Time Flags flags can be an OR of the following flags:

- MRT_P_REALTIME: to set the process as Real-Time.
- MRT_P_PERIODIC: to set the process as Real-Time Periodic.

The Message Queue Policy Flags mq_flags can be an OR of the following flags:

- MRT_PRTYORDER: Priority Order Policy (otherwise FIFO policy).
- MRT_PRTYINHERIT: Priority Inheritance policy.

DESCRIPTION

mrt_setpattr Sets the RT-Processing Attributes of the calling NRT-process. This call is restricted to the super-user. It must be executed by a NRT-process in System Real Time Processing Mode.

RETURN VALUE

Upon successful completion, a value of 0 (OK) is returned. Otherwise, a negative value is returned to indicate an error.

ERRORS

mrt_setpattr will fail if one or more of the following are true:

[EINVAL] Invalid mrt_pattr_t pointer.

[E_MRT_BADPTYPE] Invalid value specified for process type in p_attrib->ptype.

[E_MRT_BADPRTY]Invalid value specified for process priority in p_attrib->priority.

[E_MRT_BADWDOG] Non existing or NRT-watchdog process specified in p_attrib ->watchdog.

[E_MRT_NOMSGQ] The system cannot assign a Message Queue of the specified size in p_attrib->mq_size.

[E_MRT_NOVTIMER] The system cannot assign a Virtual Timer for a Periodic RT-process.

SEE ALSO

mrt_getpattr(2).

A.2. Kernel Calls Reference

A.2.1. mrt_rqst

```
NAME
     mrt_rqst - sends a synchronous request message.

SYNOPSIS
     #include <unistd.h>
     int mrt_rqst(mrtpid_t mrtpid, mrt_msg_t *m_ptr, lcounter_t timeout);

ARGUMENTS

     mrtpid: The RT-PID of the destination RT-process.

     m_ptr: A pointer to the message buffer.

     timeout: The number of RT-ticks for waiting to send de request. A  MRT_NOWAIT
value  can be specified to return without waiting if the destination process is not
waiting for this message.  A MRT_FOREVER value can be specified to wait until the
destination process receive the message.

DESCRIPTION
     mrt_rqst sends a request message to a process through a message queue  in a
synchronous way with or without specifying a timeout. It must be executed by a RT-
process.

RETURN VALUE
     Upon successful completion, a value of 0 (OK) is returned.  Otherwise,  a
negative value is returned to indicate an error.

ERRORS
     mrt_rqst will fail if one or more of the following are true:

     [E_MRT_NORTMODE] The system is in Non Real Time processing Mode.

     [E_MRT_BADPROC] The process PID does not much with the process number  use by
the kernel.

     [E_MRT_BADPTYPE] The Destination process is not a RT-process.

     [E_BAD_DEST]   The Destination process is does not exist.

     [E_TRY_AGAIN]  The Destination process message queue is full.

     [E_MRT_NOMQENT] The system message queue entry free pool is empty.

SEE ALSO
     mrt_arqst(2).  mrt_uprqst(2).
```

A.2.2. mrt_arqst

```
NAME
     mrt_arqst - sends an Asynchronous request message.

SYNOPSIS
     #include <unistd.h>
```

```
     int mrt_arqst(mrtpid_t mrtpid, mrt_msg_t *m_ptr);

ARGUMENTS

     mrtpid: The RT-PID of the destination RT-process.

     m_ptr: A pointer to the message buffer.

DESCRIPTION
     mrt_arqst sends a request message to a process through a message queue in an
Asynchronous way. It must be executed by a RT-process.

RETURN VALUE
     Upon successful completion, a value of 0 (OK) is returned.  Otherwise,  a
negative value is returned to indicate an error.

ERRORS
     mrt_arqst will fail if one or more of the following are true:

     [E_MRT_NORTMODE] The system is in Non Real Time processing Mode.

     [E_MRT_BADPTYPE] The Source process is not a RT-process.

     [E_MRT_BADPTYPE] The Destination process is not a RT-process.

     [E_BAD_DEST]  The Destination process is does not exist.

     [E_TRY_AGAIN]  The Destination process message queue is full.

     [E_MRT_NOMQENT] The system message queue entry free pool is empty.

SEE ALSO
     mrt_rqst(2).  mrt_uprqst(2).
```

A.2.3. mrt_uprqst

```
NAME
     mrt_uprqst - sends an Asynchronous request message in a botton-up way.

SYNOPSIS
     #include <unistd.h>

     int mrt_uprqst(mrtpid_t mrtpid, mrt_msg_t *m_ptr, int priority);

ARGUMENTS

     mrtpid: The RT-PID of the destination RT-process.

     m_ptr: A pointer to the message buffer.

     priority: message priority.

DESCRIPTION
     mrt_uprqst sends a request message to a process through a  message  queue in
an  Asynchronous  way  in a botton-up manner. The destination process increase its
priority if the priority argument is higher that the one it owns. It must be
executed by a RT-process.

RETURN VALUE
     Upon successful completion, a value of 0 (OK) is returned.  Otherwise,  a
negative value is returned to indicate an error.

ERRORS
     mrt_uprqst will fail if one or more of the following are true:
```

```
     [E_MRT_NORTMODE] The system is in Non Real Time processing Mode.

     [E_MRT_BADPTYPE] The Source process is not a RT-process.

     [E_MRT_BADPTYPE] The Destination process is not a RT-process.

     [E_BAD_DEST]   The Destination process is does not exist.

     [E_TRY_AGAIN]  The Destination process message queue is full.

     [E_MRT_NOMQENT] The system message queue entry free pool is empty.

SEE ALSO
     mrt_rqst(2).  mrt_arqst(2).
```

A.2.4. mrt_sign

```
NAME
     mrt_sign - sends a message to a process through a MQ in  an  asynchronous way.

SYNOPSIS
     #include <unistd.h>

     int mrt_sign(mrtpid_t mrtpid, mrt_msg_t *m_ptr);

ARGUMENTS

     mrtpid: The RT-PID of the destination RT-process.

     m_ptr: A pointer to the message buffer.

     priority: Message priority.

DESCRIPTION
     mrt_sign sends a request message to a process through a message queue  in an
Asynchronous way in a botton-up manner. It must be executed by a RT-process.


RETURN VALUE
     Upon successful completion, a value of 0 (OK) is returned. Otherwise, a
negative value is returned to indicate an error.


ERRORS
     mrt_sign will fail if one or more of the following are true:

     [E_MRT_NORTMODE] The system is in Non Real Time processing Mode.

     [E_MRT_BADPTYPE] The Source process is not a RT-process.

     [E_MRT_BADPTYPE] The Destination process is not a RT-process.

     [E_BAD_DEST]   The Destination process is does not exist.

     [E_TRY_AGAIN]  The Destination process message queue is full.

     [E_MRT_NOMQENT] The system message queue entry free pool is empty.

SEE ALSO
     mrt_rqst(2).  mrt_arqst(2).
```

A.2.5. mrt_reply

```
NAME
     mrt_reply - sends a message to a process through a MQ in an  asynchronous way.

SYNOPSIS
     #include <unistd.h>

     int mrt_reply(mrtpid_t mrtpid, Imrt_msg_t *m_ptr);

ARGUMENTS

     mrtpid: The RT-PID of the destination RT-process.

     m_ptr: A pointer to the message buffer.

DESCRIPTION
     mrt_reply sends a message to a process through a MQ  in  an  asynchronous way.
It must be executed by a RT-process.

RETURN VALUE
     Upon successful completion, a value of 0 (OK) is returned.  Otherwise,  a
negative value is returned to indicate an error.

ERRORS
     mrt_reply will fail if one or more of the following are true:

     [E_MRT_NORTMODE] The system is in Non Real Time processing Mode.

     [E_MRT_BADPTYPE] The Source process is not a RT-process.

     [E_MRT_BADPTYPE] The Destination process is not a RT-process.

     [E_BAD_DEST]   The Destination process is does not exist.

     [E_TRY_AGAIN]  The Destination process message queue is full.

     [E_MRT_NOMQENT] The system message queue entry free pool is empty.

SEE ALSO
     mrt_arqst(2).  mrt_uprqst(2).
```

A.2.6. mrt_rcv

```
NAME
     mrt_rcv - Gets a message from the RT-process MQ.

SYNOPSIS
     #include <unistd.h>

     int mrt_rcv(mrtpid_t mrtpid, mrt_hdr_t *hdr_ptr,
                     mrt_msg_t *m_ptr, lcounter_t timeout);

ARGUMENTS

     mrtpid
          The RT-PID of the message sender from  which  the  caller  wants  to
receive  a  message. A special value of MRT_ANYPROC can be specified to receive a
message from any source.

     dr_ptr: The caller´s buffer for the message header.

     m_ptr: The caller^Rs buffer for the message payload.
```

```
     timeout: The  number  of  RT-ticks  for  waiting  to  receive  a  message. A
MRT_NOWAIT value can be specified to return with o without receiving the message.
If the message has been received, the  function  return code  is  OK  else  returns
E_TRY_AGAIN.  A MRT_FOREVER value can be specified to waits until the message is
receive.

DESCRIPTION
     mrt_rcv gets a message from the RT-process MQ. It must be executed by a RT-
process.

RETURN VALUE
     Upon successful completion, a value of 0 (OK) is returned.  Otherwise,  a
negative value is returned to indicate an error.

ERRORS
     mrt_rcv will fail if one or more of the following are true:

     [E_MRT_NORTMODE] The system is in Non Real Time processing Mode.

     [E_MRT_BADPTYPE] The Source process is not a RT-process.

     [E_TRY_AGAIN] The process  has  specified a MRT_NOWAIT timeout and the MQ is
empty or there are not  any  message  from  the  specified source.

     [MRT_NOVTIMER] The  process  has  specified  a timeout to wait for  a  message
but  there  are  not  any Virtual Timer to allocate.


SEE ALSO
     mrt_rqst(2).  mrt_reply(2).
```

A.2.7. mrt_sleep

```
NAME
     mrt_sleep - Blocks a process for a specified time.

SYNOPSIS
     #include <unistd.h>

     int mrt_sleep(lcounter_t time);

ARGUMENTS

     time: The process is blocked  for  a  specified  time  (in  RT-ticks).   A
MRT_FOREVER  value  can  be specified for the time to wait until the process will
be unblocked by  other  process  using  the  mrt_wakeup System Call.

DESCRIPTION
     mrt_sleep blocks a process for a specified time. It must be executed by a RT-
process.

RETURN VALUE
     Upon successful completion, a value of 0 (OK) is returned.  Otherwise,  a
negative value is returned to indicate an error.

ERRORS
     mrt_sleep will fail if one or more of the following are true:

     [E_MRT_NORTMODE] The system is in Non Real Time processing Mode.

     [E_MRT_BADPER] The  process  has  specified  an  invalid  time  argument.

     [MRT_NOVTIMER]  The  process  has specified a time to wait but there are not
any Virtual Timer to allocate.
```

```
SEE ALSO
     mrt_wakeup(2).
```

A.2.8. mrt_wakeup

```
NAME
     mrt_wakeup - Wakes up a blocked (MRT_SLEEP flag set) process.

SYNOPSIS
     #include <unistd.h>

     int mrt_wakeup(mrtpid_t mrtpid);

ARGUMENTS

     mrtpid: The RT-PID of the RT-process to unblock.

DESCRIPTION
     mrt_wakeup wakes up a blocked (MRT_SLEEP flag set) process. It must be
executed by a RT-process.

RETURN VALUE
     Upon successful completion, a value of 0 (OK) is returned.  Otherwise,  a
negative value is returned to indicate an error.

ERRORS
     mrt_wakeup will fail if one or more of the following are true:

     [E_MRT_NORTMODE] The system is in Non Real Time processing Mode.

     [E_MRT_BADPROC] The process PID does not much with the process number  use by
the kernel.

     [E_MRT_BADPTYPE] The Destination process is not a RT-process.

SEE ALSO
     mrt_sleep(2).
```

A.2.9. prt_print

```
NAME
     mrt_print - Prints a specified text on the system console.

SYNOPSIS
     #include <unistd.h>

     int mrt_print(const char* string);

ARGUMENTS

     string: The string to print.

DESCRIPTION
     mrt_print Prints a specified text on  the  system  console.  The  maximum
string length accepted is MAXPRTSTR. It must be executed by a RT-process.

RETURN VALUE
     Upon successful completion, a value of 0 (OK) is returned.  Otherwise,  a
negative value is returned to indicate an error.

ERRORS
     mrt_print will fail if one or more of the following are true:
```

```
[E_MRT_NORTMODE] The system is in Non Real Time processing Mode.

[E_MRT_BADPTYPE] The caller process is not a RT-process.
```

Appendix B: SAMPLE PROGRAMS

This appendix show several sample programs that use MINIX4RT System Calls and Kernel Calls.

B.1. mrtstart.c

This program starts the Real-Time processing mode. The harmonic frequency of MINIX Timer can be specified as an argument and the number of Timer ticks to refresh IDLE process statistics. By default *harmonic=4* and *refresh=200* are assumed.

Listing B.1: mrtstart.c

```
/*************************************************************************/
/*                              mrtstart.c                               */
/* Starts the Real Time mode using mrt_RTstart() System Call             */
/* Usage:                                                                */
/*    mrtstart [harmonic [refresh]]                                      */
/* where:                                                                */
/*    harmonic: The Harmonic Number of the MINIX timer frequency (HZ = 50 Hz).*/
/*    refresh: The idle refresh counter in timer ticks.                  */
/*************************************************************************/


#include <minix/config.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <stdlib.h>
#include <minix/type.h>
#include <unistd.h>
#include <stdio.h>
#include <lib.h>
#include <minix/syslib.h>
#include <minix/const.h>

_PROTOTYPE(int main, (int argc, char *argv []));

int main(argc, argv)
int argc;
char *argv[];
{
```

```
	int rcode, harmonic, refresh;

	harmonic = 4;
	refresh = 4 * HZ;

	switch (argc)
		{
		case 1:
			break;
		case 2:
			harmonic = atoi(argv[1]);
			break;
		case 3:
			harmonic = atoi(argv[1]);
			refresh = atoi(argv[2]);
			break;
		default:
			printf("usage: \n\tmrtstart [harmonic [refresh]]\n");
			exit(1);
		}

	rcode = mrt_RTstart(harmonic,refresh);
	if( rcode != 0)
		{
		printf("mrt_RTstart: rcode=%5d.\n", rcode);
		exit(rcode);
		}
	else
		{
		printf("System is in Realtime Mode.");
		printf(" Harmonic=%d (%d ticks\/s), IDLE refresh=%d ticks\n"
		,harmonic,harmonic*HZ,refresh);
		}
	exit(0);
}
```

B.2. mrtstop.c

This program stops the Real-Time processing mode only if there are not any RT-process running in the system.

Listing B.2: mrtstop.c

```
/****************************************************************************/
/*                              mrtstop.c                                   */
/* Stops the Real Time processing mode using mrt_RTstop() System Call       */
/* Usage:                                                                   */
/*    mrtstop                                                               */
/****************************************************************************/

#include <minix/config.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <stdlib.h>
#include <minix/type.h>
#include <unistd.h>
#include <stdio.h>
#include <lib.h>
#include <minix/syslib.h>
```

```
_PROTOTYPE(int main, (int argc, char *argv []));

int main(argc, argv)
int argc;
char *argv[];
{
	int rcode;

	rcode = mrt_RTstop();
	if( rcode != 0)
		{
		printf("mrt_RTstop: rcode=%5d.\n", rcode);
		exit(1);
		}
	else
		printf("System is in Non Realtime Mode\n");

	exit(0);
}
```

B.3. mrttest1.c

This program gets and displays Interrrupt Descriptors of attribute fields. At the last line the test must display *irq=32 rcode=-2001.*

Listing B.3: mrttest1.c

```
/******************************************************************************/
/*                              mrttest1.c                                    */
/* Test the mrt_getiattr() System Call to get Interrupt Descriptor Attributes */
/* Usage:                                                                     */
/*    mrttest1                                                                */
/******************************************************************************/

#include <minix/config.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

#include <lib.h>
#include <minix/syslib.h>

_PROTOTYPE(int main, (int argc, char *argv []));

int main(argc, argv)
int argc;
char *argv[];
{

	int irq, rcode;
	mrt_irqattr_t iattrs;

	printf("IRQ PERIOD TASK WATCHDOG PRIORITY IRQTYPE NAME\n");

	for( irq = 0; irq < 33; irq++)
		{
		rcode = mrt_getiattr(irq, &iattrs);
		if( rcode != 0)
```

```
                    printf("irq=%2d rcode=%5d.The test is OK if irq=32 rcode=-2001
\n"
                           ,irq ,rcode);
             else
                    printf("%3d %6d %4d %8d %8d %7X %-16s\n",
                           irq,
                           iattrs.period,
                           iattrs.task,
                           iattrs.watchdog,
                           iattrs.priority,
                           iattrs.irqtype,
                           iattrs.name);
             }
}
```

B.4. mrttest1b.c

This program gets and displays Interrrupt Descriptors of statistical fields. At the last line the test must display *irq=32 rcode=-2001*.

Listing B.4: mrttest1b.c

```
/********************************************************************************/
/*                              mrttest1b.c                                     */
/* Test the mrt_getistat() System Call to get Interrupt Descriptor Statistics  */
/* Usage:                                                                       */
/*    mrttest1b                                                                 */
/********************************************************************************/

#include <minix/config.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <lib.h>
#include <minix/syslib.h>

_PROTOTYPE(int main, (int argc, char *argv []));

int main(argc, argv)
int argc;
char *argv[];
{

      int irq, rcode;
      mrt_irqstat_t istat;

      printf("IRQ COUNT MAXSHWR MDL TSTAMP MAXLAT REENTER\n");

      for( irq = 0; irq < 33; irq++)
             {
             rcode = mrt_getistat(irq, &istat);
             if( rcode != 0)
                    printf("irq=%2d rcode=%5d. Its OK if irq=32 rcode=-2001 \n"
                           ,irq ,rcode);
             else
                    printf("%3d %5d %7d %3d %6d %6d %7d\n",
                           irq,
                           istat.count,
```

```
                          istat.maxshower,
                          istat.mdl,
                          istat.timestamp,
                          istat.maxlat,
                          istat.reenter);
             }
}
```

B.5. mrttest1c.c

This program gets and displays Interrrupt Descriptors of internal use fields. At the last line the test must display *irq=32 rcode=-2001.*

Listing B.5: mrttest1c.c

```
/******************************************************************************/
/*                              mrttest1c.c                                   */
/* Test the mrt_getiint() System Call to get Interrupt Descrip. Internal Data */
/* Usage:                                                                     */
/*    mrttest1c                                                               */
/******************************************************************************/

#include <minix/config.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <lib.h>
#include <minix/syslib.h>

_PROTOTYPE(int main, (int argc, char *argv []));

int main(argc, argv)
int argc;
char *argv[];
{

      int irq, rcode;
      mrt_irqint_t iint;

      printf("IRQ NUMBER HARMONIC VTIMER FLAGS\n");

      for( irq = 0; irq < 33; irq++)
             {
             rcode = mrt_getiint(irq, &iint);
             if( rcode != 0)
                    printf("irq=%2d rcode=%5d.It is OK if irq=32 rcode=-2001 \n"
                           ,irq ,rcode);
             else
                    printf("%3d %7d %7d %7d %5X\n",
                           irq,
                           iint.irq,
                           iint.harmonic,
                           iint.vtimer,
                           iint.flags);
             }
}
```

B.6. mrttest2.c

This program sets Interrrupt Descriptors processing attribute fields.

Listing B.6: mrttest2.c

```
/*****************************************************************************/
/*                              mrttest2.c                                    */
/* Test the mrt_setiattr() System Call to set Interrupt Descriptor Attributes */
/* Usage:                                                                     */
/*    mrttest2                                                                */
/*****************************************************************************/

#include <minix/config.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <lib.h>
#include <minix/syslib.h>
#include <minix/com.h>
#include <minix/const.h>

#define RS232_IRQ   4
#define HARDWARE    -1
#define RS232_TASK  -11

_PROTOTYPE(int main, (int argc, char *argv []));


int main(argc, argv)
int argc;
char *argv[];
{

	int rcode;
	mrt_irqattr_t rs_attrs;

	rs_attrs.period    = 0;
	rs_attrs.task      = RS232_TASK;
	rs_attrs.watchdog  = HARDWARE;
	rs_attrs.priority  = MRT_PRI05;
	rs_attrs.irqtype   = MRT_RTIRQ;
	strncpy(rs_attrs.name,"RT-RS232NEW",15);

	rcode = mrt_setiattr(RS232_IRQ , &rs_attrs);
	printf("mrt_setiattr: \nirq=%2d rcode=%5d.\n", RS232_IRQ,rcode);
	if( rcode != 0) exit(1);
}
```

B.7. mrttest3.c

This program gets and displays System-wide statistics.

Listing B.7: mrttest3.c

```
/*****************************************************************************/
/*                              mrttest3.c                                   */
/* Test the mrt_getsstat() System Call to get System Wide Statistics         */
/* Usage:                                                                    */
/*    mrttest3                                                               */
/*****************************************************************************/


#include <minix/config.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <lib.h>
#include <minix/syslib.h>

_PROTOTYPE(int main, (int argc, char *argv []));

int main(argc, argv)
int argc;
char *argv[];
{
	mrt_sysstat_t stats;
	int rcode;

	rcode = mrt_getsstat(&stats);
	printf("mrt_getsstat: rcode=%d.\n\n", rcode);
	if( rcode != 0) exit(1);
	printf("schedulings = %-d\n",stats.scheds);
	printf("messages    = %-d\n",stats.messages);
	printf("interrupts  = %-d\n",stats.interrupts);
	printf("ticks       = %-d:%-d\n",stats.highticks,stats.ticks);
	printf("idle last/max     = %-d/%-d\n",stats.idlelast,stats.idlemax);
}
```

B.8. mrttest3b.c

This program displays a graph of CPU usage.

Listing B.8: mrttest3b.c

```
/*****************************************************************************/
/*                              mrttest3b.c                                  */
/* Test the mrt_getsstat() System Call displaying a graph of the CPU Usage   */
/* Usage:                                                                    */
/*    mrttest3b                                                              */
/*****************************************************************************/

#include <minix/config.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <lib.h>
```

```
#include <minix/syslib.h>

#define TOPSCALE          70

_PROTOTYPE(int main, (int argc, char *argv []));

int main(argc, argv)
int argc;
char *argv[];
{
      mrt_sysstat_t stats;
      int rcode, i, j;
      long cpu,idle;
printf("                              CPU USAGE\n");
printf("|------10-----20-----30-----40-----50-----60-----70-----80-----90----
100\n");
      for( i = 0; i <  20; i++)
            {
            rcode = mrt_getsstat(&stats);
            if( rcode != 0)
                  {
                  printf("mrt_getsstat: rcode=%d.\n\n", rcode);
                  exit(rcode);
                  }
            else
                  printf("|");
            idle = (stats.idlelast*TOPSCALE);
            idle /= stats.idlemax;
            cpu = (TOPSCALE - idle);
            for(j = 0;  j < cpu; j++)
                  printf("#");
            printf("\n");
            sleep(1);
            }
}
```

B.9. mrttest3c.c

This program gets and displays System processing values.

Listing B.9: mrttest3c.c

```
/****************************************************************************/
/*                              mrttest3c.c                                 */
/* Test the mrt_getsval() System Call to get System Values                  */
/* Usage:                                                                   */
/*    mrttest3c                                                             */
/****************************************************************************/

#include <minix/config.h>

#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <lib.h>
#include <minix/syslib.h>

_PROTOTYPE(int main, (int argc, char *argv []));
```

```
int main(argc, argv)
int argc;
char *argv[];
{
	mrt_sysval_t val;
	int rcode;

	rcode = mrt_getsval(&val);
	printf("mrt_getsval: rcode=%d.\n\n", rcode);
	if( rcode != 0) exit(1);

	printf("flags       = x%-X\n",val.flags);
	printf("virtual_PIC = x%-X\n",val.virtual_PIC);
	printf("PIT_latency = %-d\n",val.PIT_latency);
	printf("PIT_latch   = %-d\n",val.PIT_latch);
	printf("tickrate    = %-d\n",val.tickrate);
	printf("harmonic    = %-d\n",val.harmonic);
	printf("refresh     = %-d\n",val.refresh);
}
```

B.10. mrttest4.c

This program gets and displays Process Descriptor Attributes.

Listing B.10: mrttest4.c

```
/****************************************************************************/
/*                              mrttest4.c                                  */
/* Test the mrt_getpattr() System Call to get Process Descriptor Attributes */
/* Usage:                                                                   */
/*     mrttest4                                                             */
/****************************************************************************/

#include <minix/config.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>


#include <lib.h>
#include <minix/syslib.h>

_PROTOTYPE(int main, (int argc, char *argv []));

int main(argc, argv)
int argc;
char **argv;
{
	mrt_pattr_t pattr;
	int rcode;
	pid_t pid;

	if(argc != 2)
		{
		printf("format: mrttest4 <pid>\n");
		exit(1);
		}
	pid = atoi(argv[1]);
```

```
	rcode = mrt_getpattr(pid, &pattr);
	if( rcode != 0)
		{
		printf("mrt_getpattr: pid= %d rcode=%5d.\n", pid, rcode);
		exit(1);
		}

	printf("flags      = %X\n",pattr.flags);
	printf("baseprty   = %X\n",pattr.baseprty);
	printf("period     = %d\n",pattr.period);
	printf("limit      = %d\n",pattr.limit);
	printf("deadline   = %d\n",pattr.deadline);
	printf("watchdog   = %d\n",pattr.watchdog);
	printf("MQ size    = %d\n",pattr.mq_size);
	printf("MQ flags   = %X\n",pattr.mq_flags);
}
```

B.11. mrttest4b.c

This program sets Process Descriptor Attributes.

Listing B.11: mrttest4b.c

```
/******************************************************************************/
/*                              mrttest4b.c                                   */
/* Test the mrt_setpattr() System Call to set Process Descriptor Attributes   */
/* Usage:                                                                     */
/*    mrttest4b                                                               */
/******************************************************************************/

#include <minix/config.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <lib.h>
#include <minix/syslib.h>

_PROTOTYPE(int main, (int argc, char *argv []));

int main(argc, argv)
int argc;
char **argv;
{
	mrt_pattr_t pattr;
	int rcode;

	pattr.flags         = (MRT_P_REALTIME | MRT_P_PERIODIC);
	pattr.baseprty      = MRT_PRI03;
	pattr.period        = 10;
	pattr.limit         = 222;
	pattr.deadline      = 11;
	pattr.watchdog      = -1;
	pattr.mq_size       = 3;
	pattr.mq_flags      = 0;

	rcode = mrt_setpattr(&pattr);
	if( rcode != 0)
		{
```

```
            printf("mrt_setpattr: rcode=%5d.\n", rcode);
            exit(1);
            }
      else
            printf("Process Attributes set\n");
      sleep(60);
}
```

B.12. mrttest5.c

This program gets and displays Process Descriptor statistical fields.

Listing B.12: mrttest5.c

```
/****************************************************************************/
/*                              mrttest5.c                                  */
/* Test the mrt_getpstat() System Call to get Process Statistics            */
/* Usage:                                                                   */
/*    mrttest5                                                              */
/****************************************************************************/

#include <minix/config.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <lib.h>
#include <minix/syslib.h>

_PROTOTYPE(int main, (int argc, char *argv []));


int main(argc, argv)
int argc;
char *argv[];
{
      mrt_pstat_t pstats;
      int rcode;
      pid_t pid;

      if (argc == 2)
            pid = atoi(argv[1]);
      else
            pid = getpid();

      rcode = mrt_getpstat(pid, &pstats);
      printf("mrt_getpstat: pid = %d, rcode=%5d.\n", pid, rcode);
      if( rcode != 0) exit(1);

      printf("scheds     = %10d\n",pstats.scheds);
      printf("mdl        = %10d\n",pstats.mdl);
      printf("timestamp  = %10d\n",pstats.timestamp);
      printf("maxlat     = %10d\n",pstats.maxlat);
      printf("minlax     = %10d\n",pstats.minlax);
      printf("msgsent    = %10d\n",pstats.msgsent);
      printf("msgrcvd    = %10d\n",pstats.msgrcvd);
}
```

B.13. mrttest5b.c

This program gets and displays Process Descriptor internal use fields.

Listing B.13: mrttest5b.c

```
/****************************************************************************/
/*                              mrttest5b.c                                 */
/* Test the mrt_getpint() System Call to get Process Internal Data         */
/* Usage:                                                                   */
/*    mrttest5b                                                             */
/****************************************************************************/

#include <minix/config.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

#include <lib.h>
#include <minix/syslib.h>

_PROTOTYPE(int main, (int argc, char *argv []));

int main(argc, argv)
int argc;
char *argv[];
{
	mrt_pint_t pint;
	int rcode;
	pid_t pid;

	if (argc == 2)
		pid = atoi(argv[1]);
	else
		pid = getpid();

	rcode = mrt_getpint(pid, &pint);
	printf("mrt_getpint: pid = %d, rcode=%5d.\n", pid, rcode);
	if( rcode != 0) exit(1);
	printf("vt        = %d\n",pint.vt);
	printf("priority  = %d\n",pint.priority);
	printf("mqId      = %d\n",pint.mqID);
	printf("p_nr      = %d\n",pint.p_nr);
}
```

B.14. mrttest5c.c

This program clears Process Descriptor statistical fields.

Listing B.14: mrttest5c.c

```
/****************************************************************************/
/*                              mrttest5c.c                                 */
/* Test the mrt_getiattr() System Call to clear Process statistics         */
/* Usage:                                                                   */
```

```
/*     mrttest5C                                                               */
/*****************************************************************************/
/* MRT test 5c: mrt_clrpstat */

#include <minix/config.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <lib.h>
#include <minix/syslib.h>

_PROTOTYPE(int main, (int argc, char *argv []));

int main(argc, argv)
int argc;
char *argv[];
{
	int rcode;
	pid_t pid;

	if (argc == 2)
		pid = atoi(argv[1]);
	else
		pid = getpid();

	rcode = mrt_clrpstat(pid);
	printf("mrt_clrpstat: pid = %d, rcode=%5d.\n", pid, rcode);
	if( rcode != 0) exit(1);
	printf("Process %d statistics cleared\n",pid);
}
```

B.15. mrttest6.c

This program:

- Sets its process descriptor attributes to convert itself into a RT-process.
- Converts itself the process is converted into a RT-process.
- Converts itself back into a NRT-process.
- Displays its process descriptor statistics.

Listing B.15: mrttest6.c

```
/*****************************************************************************/
/*                              mrttest6.c                                     */
/* Test the mrt_set2rt(), mrt_print() and mrt_set2nrt() System Call          */
/* Usage:                                                                      */
/*     mrttest6                                                                */
/*****************************************************************************/


#include <minix/config.h>
```

```c
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <stdlib.h>
#include <minix/type.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <lib.h>
#include <minix/syslib.h>

_PROTOTYPE(int main, (int argc, char *argv []));

int main(argc, argv)
int argc;
char *argv[];
{
	mrt_pattr_t pattr;
	mrt_pstat_t pstats;
	int rcode, pid;

	pattr.flags         = MRT_P_REALTIME;
	pattr.baseprty      = MRT_PRI03;
	pattr.period        = 0;
	pattr.limit         = 0;
	pattr.deadline      = 0;
	pattr.watchdog      = -1;
	pattr.mq_size       = 3;
	pattr.mq_flags      = 0;

	rcode = mrt_setpattr(&pattr);
	if( rcode != 0)
		{
		printf("mrt_setpattr: rcode=%5d.\n", rcode);
		exit(1);
		}
	else
		printf("Process Attributes set\n");

	rcode = mrt_set2rt();
	if( rcode != 0 )
		{
		printf("mrt_set2rt rcode = %d\n",rcode);
		exit(1);
		}
	rcode = mrt_print("THAT'S ALL FOLKS!!");
	if( rcode != 0 )
		{
		printf("mrt_print rcode = %d\n",rcode);
		exit(1);
		}

	rcode = mrt_set2nrt();
	pid = getpid();

	rcode = mrt_getpstat(pid, &pstats);
	printf("mrt_getpstat: pid = %d, rcode=%5d.\n", pid, rcode);
	if( rcode != 0) exit(1);

	printf("scheds     = %10d\n",pstats.scheds);
	printf("mdl        = %10d\n",pstats.mdl);
	printf("timestamp  = %10d\n",pstats.timestamp);
	printf("maxlat     = %10d\n",pstats.maxlat);
	printf("minlax     = %10d\n",pstats.minlax);
	printf("msgsent    = %10d\n",pstats.msgsent);
	printf("msgrcvd    = %10d\n",pstats.msgrcvd);
}
```

B.16. mrttest6b.c

This program:

- Sets its process descriptor attributes to convert itself into a RT-process.
- Converts itself the process is converted into a RT-process.
- Prints a text on the system console using *mrt_print()* Kernel Call.
- Exits without calling *mrt_set2nrt()*Kernel Call.

Listing B.16: mrttest6b.c

```
/*****************************************************************************/
/*                              mrttest6b.c                                  */
/* Test the mrt_set2rt(), mrt_print() and _exit System Call                 */
/* Usage:                                                                    */
/*    mrttest6b                                                              */
/*****************************************************************************/

#include <minix/config.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <stdlib.h>
#include <minix/type.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <lib.h>
#include <minix/syslib.h>

_PROTOTYPE(int main, (int argc, char *argv []));

int main(argc, argv)
int argc;
char *argv[];
{
	mrt_pattr_t pattr;
	mrt_pstat_t pstats;
	int rcode, pid;

	pattr.flags  = MRT_P_REALTIME;
	pattr.baseprty      = MRT_PRI03;
	pattr.period = 0;
	pattr.limit  = 0;
	pattr.deadline      = 0;
	pattr.watchdog      = -1;
	pattr.mq_size       = 3;
	pattr.mq_flags      = 0;

	rcode = mrt_setpattr(&pattr);
	if( rcode != 0)
		{
		printf("mrt_setpattr: rcode=%5d.\n", rcode);
		exit(1);
		}
	else
```

```
            printf("Process Attributes set\n");


	rcode = mrt_set2rt();
	if( rcode != 0 )
		{
		printf("MRTPRINT mrt_set2rt rcode = %d\n",rcode);
		exit(1);
		}

	rcode = mrt_print("THAT'S ALL FOLKS!! Exit Without mrt_set2nrt");
}
```

B.17. mrttest6c.c

This program:

- Sets its process descriptor attributes to convert itself into a RT-process.
- Converts itself into a RT-process.
- Prints a text on the system console using *mrt_print()* Kernel Call.
- Blocks itself by 2000 Timer ticks
- Converts itself back into a NRT-process.

Listing B.17: mrttest6c.c

```
/*****************************************************************************/
/*                              mrttest6c.c                                  */
/* Test the mrt_set2rt(), mrt_print() and mrt_set2nrt(), mrt_sleep() Sys Calls */
/* Usage:                                                                    */
/*    mrttest6c                                                              */
/*****************************************************************************/


#include <minix/config.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <stdlib.h>
#include <minix/type.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <lib.h>
#include <minix/syslib.h>

_PROTOTYPE(int main, (int argc, char *argv []));
int main(argc, argv)
int argc;
char *argv[];
{
	mrt_pattr_t pattr;
	mrt_pstat_t pstats;
```

```
        int rcode, pid;

        pattr.flags         = MRT_P_REALTIME;
        pattr.baseprty      = MRT_PRI03;
        pattr.period        = 0;
        pattr.limit         = 0;
        pattr.deadline      = 0;
        pattr.watchdog      = -1;
        pattr.mq_size       = 3;
        pattr.mq_flags      = 0;

        rcode = mrt_setpattr(&pattr);
        if( rcode != 0)
                {
                printf("mrt_setpattr: rcode=%5d.\n", rcode);
                exit(1);
                }
        else
                printf("Process Attributes set\n");

        rcode = mrt_set2rt();
        if( rcode != 0 )
                {
                printf("mrt_set2rt rcode = %d\n",rcode);
                exit(1);
                }

        rcode = mrt_print("I am going to sleep...\n");
        if( rcode != 0 )
                {
                printf("mrt_print1 rcode = %d\n",rcode);
                exit(1);
                }

        rcode = mrt_sleep(200*10);
        if( rcode != 0 )
                {
                mrt_set2nrt();
                printf("mrt_sleep rcode = %d\n",rcode);
                exit(1);
                }

        rcode = mrt_print("I wake up...\n");
        if( rcode != 0 )
                {
                printf("mrt_print2 rcode = %d\n",rcode);
                exit(1);
                }
        rcode = mrt_set2nrt();
}
```

B.18. mrttest6d.c

This program forks into two processes, the *FATHER* and the *SON*:

The FATHER:

- Gets SON's process descriptor internal values to obtain its RT-PID.
- Sets its process descriptor attributes to convert itself into a RT-process.

– Converts itself into a RT-process.

– Prints a text on the system console using *mrt_print()* Kernel Call.

– Wakes up SON using *mrt_wakeup()* Kernel Call.

– Converts itself back into a NRT-process.

The *SON*:

– Sets its process descriptor attributes to convert itself into a RT-process.

– Converts itself into a RT-process.

– Prints a text on the system console using *mrt_print()* Kernel Call.

– Sleeps (using *mrt_sleep()* Kernel Call) until *FATHER* will wake up it

– Converts itself back into a NRT-process.

Listing B.18: mrttest6d.c

```
/******************************************************************************/
/*                                mrttest6d.c                                 */
/* Test the mrt_wakeup(), mrt_sleep() Sys Calls                               */
/* Usage:                                                                     */
/*     mrttest6d                                                              */
/******************************************************************************/

#include <minix/config.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <lib.h>
#include <minix/syslib.h>

_PROTOTYPE(int main, (int argc, char *argv []));

int main(argc, argv)
int argc;
char *argv[];
{
	mrt_pattr_t s_pattr, f_pattr;
	int rcode;
	pid_t pid, ppid;
	mrtpid_t mrtpid;
	mrt_pint_t pint;

/*----------------------------------------------------------------------------*/
/*                                     FATHER                                 */
/*----------------------------------------------------------------------------*/
```

```
	if( (pid = fork()) != 0)
		{
		ppid = getpid();
		printf("[FATHER]I am [%d],I will wakeup my son [%d] in 20 secs\n"
			,ppid,pid);
		sleep(20);

		rcode = mrt_getpint(pid, &pint);
		if( rcode != 0)
			{
			printf("[FATHER] mrt_getpattr: pid=%d rcode=%5d.\n"
				, pid, rcode);
			exit(1);
			}
		mrtpid.pid   = pid;
		mrtpid.p_nr = pint.p_nr;

		s_pattr.flags       = MRT_P_REALTIME;
		s_pattr.baseprty    = MRT_PRI04;
		s_pattr.period      = 0;
		s_pattr.limit       = 0;
		s_pattr.deadline    = 0;
		s_pattr.watchdog    = -1;
		s_pattr.mq_size     = 3;
		s_pattr.mq_flags    = 0;

		rcode = mrt_setpattr(&s_pattr);
		if( rcode != 0)
			{
			printf("[FATHER] mrt_setpattr: rcode=%5d.\n", rcode);
			exit(1);
			}
		else
			printf("[FATHER] Process Attributes set\n");

		rcode = mrt_set2rt();
		if( rcode != 0 )
			{
			printf("[FATHER] mrt_set2rt rcode = %d\n",rcode);
			exit(1);
			}

		rcode = mrt_print("[FATHER] I will wakeup my son...\n");
		if( rcode != 0 )
			{
			mrt_set2nrt();
			printf("[FATHER] mrt_print rcode = %d\n",rcode);
			exit(1);
			}

		rcode = mrt_wakeup(mrtpid);

		if( rcode != 0 )
			{
			mrt_set2nrt();
			printf("[FATHER] mrt_wakeup rcode = %d\n",rcode);
			exit(1);
			}

		rcode = mrt_print("[FATHER] mrt_wakeup OK!!\n");
		if( rcode != 0 )
			{
			mrt_set2nrt();
			printf("[FATHER] mrt_print rcode = %d\n",rcode);
			exit(1);
			}

		rcode = mrt_set2nrt();
```

```
            printf("[FATHER]Bye\n");
            }
/*---------------------------------------------------------------------------*/
/*                                     SON                                   */
/*---------------------------------------------------------------------------*/
      else
            {
            pid = getpid();
            printf("[SON] I am [%d], and I will change to RT and put on sleep\n"
                   ,pid);
            s_pattr.flags       = MRT_P_REALTIME;
            s_pattr.baseprty    = MRT_PRI03;
            s_pattr.period      = 0;
            s_pattr.limit       = 0;
            s_pattr.deadline    = 0;
            s_pattr.watchdog    = -1;
            s_pattr.mq_size     = 3;
            s_pattr.mq_flags    = 0;

            rcode = mrt_setpattr(&s_pattr);
            if( rcode != 0)
                  {
                  printf("[SON] mrt_setpattr: rcode=%5d.\n", rcode);
                  exit(1);
                  }
            else
                  printf("[SON] Process Attributes set\n");


            rcode = mrt_set2rt();
            if( rcode != 0 )
                  {
                  printf("[SON] mrt_set2rt rcode = %d\n",rcode);
                  exit(1);
                  }

            rcode = mrt_print("[SON] I am going to sleep...\n");
            if( rcode != 0 )
                  {
                  mrt_set2nrt();
                  printf("[SON] mrt_print1 rcode = %d\n",rcode);
                  exit(1);
                  }

            rcode = mrt_sleep(MRT_FOREVER);
            if( rcode != 0 )
                  {
                  mrt_set2nrt();
                  printf("[SON] mrt_sleep rcode = %d\n",rcode);
                  exit(1);
                  }

            rcode = mrt_print("[SON] I wake up...\n");
            if( rcode != 0 )
                  {
                  mrt_set2nrt();
                  printf("[SON] mrt_print2 rcode = %d\n",rcode);
                  exit(1);
                  }

            rcode = mrt_set2nrt();
            printf("[SON]Bye\n");
            }
}
```

B.19. mrttest6e.c

This program forks into eleven processes, the *FATHER* and ten *SONs*:

The FATHER:

- Gets SONs' process descriptor internal values to obtain their RT-PID.
- Sets its process descriptor attributes to convert itself into a RT-process.
- Converts itself into a RT-process.
- Prints a text on the system console using *mrt_print()* Kernel Call.
- Wakes up ten *SON*s using *mrt_wakeup()* Kernel Call.
- Converts itself back into a NRT-process.

The *SONs*:

- Set their process descriptor attributes to convert themselves into RT-processes.
- Convert themselves into RT-processes.
- Print texts on the system console using *mrt_print()* Kernel Calls.
- Sleep (using *mrt_sleep()* Kernel Call) until *FATHER* will wake up them.
- Convert themselves back into NRT-processes.

Listing B.19: mrttest6e.c

```
/******************************************************************************/
/*                              mrttest6e.c                                   */
/* Test the mrt_sleep(), mrt_wakeup() Sys Calls                               */
/* Usage:                                                                     */
/*    mrttest6e                                                               */
/******************************************************************************/

#include <minix/config.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
```

```
#include <lib.h>
#include <minix/syslib.h>

_PROTOTYPE(int main, (int argc, char *argv []));

int main(argc, argv)
int argc;
char *argv[];
{
	mrt_pattr_t s_pattr, f_pattr;
	int rcode,i,j;
	pid_t pid[10], ppid;
	mrtpid_t mrtpid[10];
	mrt_pint_t pint[10];
/*---------------------------------------------------------------------------------*/
/*                                    FATHER                                       */
/*---------------------------------------------------------------------------------*/
	for( i = 0; i < 10 ; i++)
		pid[i] = 1;

	for( i = 0; i < 10 && pid[i] != 0 ; i++)
		pid[i] = fork();

	if( i == 10)
		{
		ppid = getpid();
		printf("[FATHER]I am [%d],I will wakeup my sons \n",ppid,pid);

		for ( j = 0; j < 10; j++)
			{
			rcode = mrt_getpint(pid[j], &pint[j]);
			if( rcode != 0)
				{
				printf("[FATHER] mrt_getpattr: pid=%d rcode=%5d\n"
				     ,pid[j], rcode);
				exit(1);
				}
			mrtpid[j].pid  = pid[j];
			mrtpid[j].p_nr = pint[j].p_nr;
			}

		s_pattr.flags       = MRT_P_REALTIME;
		s_pattr.baseprty    = MRT_PRI04;
		s_pattr.period      = 0;
		s_pattr.limit       = 0;
		s_pattr.deadline    = 0;
		s_pattr.watchdog    = -1;
		s_pattr.mq_size     = 3;
		s_pattr.mq_flags    = 0;

		rcode = mrt_setpattr(&s_pattr);
		if( rcode != 0)
			{
			printf("[FATHER] mrt_setpattr: rcode=%5d.\n", rcode);
			exit(1);
			}
		else
			printf("[FATHER] Process Attributes set\n");

		rcode = mrt_set2rt();
		if( rcode != 0 )
			{
			printf("[FATHER] mrt_set2rt rcode = %d\n",rcode);
			exit(1);
			}

		mrt_print("[FATHER] I will wakeup my sons...\n");
```

```
            for ( j = 0; j < 10; j++);
                  mrt_wakeup(mrtpid[j]);

            rcode = mrt_set2nrt();

            printf("[FATHER]Bye\n");
            }
/*----------------------------------------------------------------------------*/
/*                                    SON                                     */
/*----------------------------------------------------------------------------*/
      else
            {
            pid[i] = getpid();
            printf("[SON]%d I am [%d], and I will change to RT and put on sleep\n"
                   ,i,pid[i]);
            s_pattr.flags        = MRT_P_REALTIME;
            s_pattr.baseprty     = MRT_PRI03;
            s_pattr.period       = 0;
            s_pattr.limit        = 0;
            s_pattr.deadline     = 0;
            s_pattr.watchdog     = -1;
            s_pattr.mq_size      = 3;
            s_pattr.mq_flags     = 0;

            rcode = mrt_setpattr(&s_pattr);
            if( rcode != 0)
                  {
                  printf("[SON] mrt_setpattr: rcode=%5d.\n", rcode);
                  exit(1);
                  }
            else
                  printf("[SON] Process Attributes set\n");

            rcode = mrt_set2rt();
            if( rcode != 0 )
                  {
                  printf("[SON] mrt_set2rt rcode = %d\n",rcode);
                  exit(1);
                  }

            rcode = mrt_print("[SON] I am going to sleep...\n");
            if( rcode != 0 )
                  {
                  mrt_set2nrt();
                  printf("[SON] mrt_print1 rcode = %d\n",rcode);
                  exit(1);
                  }

            rcode = mrt_sleep(MRT_FOREVER);
            if( rcode != 0 )
                  {
                  mrt_set2nrt();
                  printf("[SON] mrt_sleep rcode = %d\n",rcode);
                  exit(1);
                  }

            rcode = mrt_print("[SON] I wake up...\n");
            if( rcode != 0 )
                  {
                  mrt_set2nrt();
                  printf("[SON] mrt_print2 rcode = %d\n",rcode);
                  exit(1);
                  }

            rcode = mrt_set2nrt();
            printf("[SON]%d Bye\n",i);
            }
}
```

B.20. mrttest7.c

This program test periodic processing.

– Sets its process descriptor attributes to convert itself into a Periodic RT-process.

– Converts itself into a RT-process.

– Loops until it processing limit is reached

– Prints a text on each loop.

– When the processing limit is reached the process is set in stopped setting the *MRT_STOP* bit of its flags.

Listing B.20: mrttest7.c

```
/*****************************************************************************/
/*                                mrttest7.c                                 */
/* Test a periodic process                                                   */
/* Usage:                                                                    */
/*     mrttest7                                                              */
/*****************************************************************************/


#include <minix/config.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <stdlib.h>
#include <minix/type.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <lib.h>
#include <minix/syslib.h>

_PROTOTYPE(int main, (int argc, char *argv []));

int main(argc, argv)
int argc;
char *argv[];
{
	mrt_pattr_t pattr;
	int rcode, pid;

	pattr.flags		= MRT_P_REALTIME | MRT_P_PERIODIC;
	pattr.baseprty	= MRT_PRI03;
	pattr.period		= 50;
	pattr.limit		= 10;
	pattr.deadline	= 0;
	pattr.watchdog	= -1;
	pattr.mq_size		= 3;
	pattr.mq_flags	= 0;
```

```
        rcode = mrt_setpattr(&pattr);
        if( rcode != 0)
                {
                printf("mrt_setpattr: rcode=%5d.\n", rcode);
                exit(1);
                }
        else
                printf("Process Attributes set\n");


        rcode = mrt_set2rt();
        if( rcode != 0 )
                {
                printf("mrt_set2rt rcode = %d\n",rcode);
                exit(1);
                }

        rcode = mrt_print("Entering in a loop...\n");
        if( rcode != 0 )
                {
                printf("mrt_print1 rcode = %d\n",rcode);
                exit(1);
                }

        do      {
                mrt_print("loop\n");
                rcode = mrt_sleep(MRT_FOREVER);
                } while( rcode == 0 );

}
```

B.21. mrttest7b.c

This program test periodic processing.

- Sets its process descriptor attributes to convert itself into a Periodic RT-process.
- Converts itself into a RT-process.
- Loops five times.
- Prints a text on each loop.
- Converts itself back into a NRT-process.

Listing B.21: mrttest7b.c

```
/******************************************************************************/
/*                              mrttest7b.c                                   */
/* Test a periodic process                                                    */
/* Usage:                                                                     */
/*    mrttest7b                                                               */
/******************************************************************************/
#include <minix/config.h>
#include <sys/types.h>
```

```
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <stdlib.h>
#include <minix/type.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <lib.h>
#include <minix/syslib.h>

_PROTOTYPE(int main, (int argc, char *argv []));

int main(argc, argv)
int argc;
char *argv[];
{
	mrt_pattr_t pattr;
	int rcode, pid,i;

	pattr.flags		= MRT_P_REALTIME | MRT_P_PERIODIC;
	pattr.baseprty	= MRT_PRI03;
	pattr.period		= 500;
	pattr.limit		= 0;
	pattr.deadline	= 0;
	pattr.watchdog	= -1;
	pattr.mq_size		= 3;
	pattr.mq_flags	= 0;

	rcode = mrt_setpattr(&pattr);
	if( rcode != 0)
		{
		printf("mrt_setpattr: rcode=%5d.\n", rcode);
		exit(1);
		}
	else
		printf("Process Attributes set\n");


	rcode = mrt_set2rt();
	if( rcode != 0 )
		{
		printf("mrt_set2rt rcode = %d\n",rcode);
		exit(1);
		}

	rcode = mrt_print("Entering in a loop...\n");
	if( rcode != 0 )
		{
		printf("mrt_print1 rcode = %d\n",rcode);
		exit(1);
		}

	for( i = 0; i < 5; i++)
		{
		mrt_print("loop\n");
		rcode = mrt_sleep(MRT_FOREVER);
		}

	mrt_set2nrt();
	printf("exiting...\n");
}
```

B.22. mrttest8.c

This program forks into two processes, the *FATHER* and the *SON*:

The FATHER:

- Gets SON's process descriptor internal values to obtain its RT-PID.
- Sets its process descriptor attributes to convert itself into a RT-process.
- Converts itself into a RT-process.
- Prints a text on the system console using *mrt_print()* Kernel Call.
- Send a request message to SON using *mrt_rqst()* Kernel Call.
- Converts itself back into a NRT-process.

The *SON*:

- Sets its process descriptor attributes to convert itself into a RT-process.
- Converts itself into a RT-process.
- Prints a text on the system console using *mrt_print()* Kernel Call.
- Sleeps for 2000 Timer ticks.
- Converts itself back into a NRT-process.

Listing B.22: mrttest8.c

```
/******************************************************************************/
/*                              mrttest8.c                                    */
/* Test the mrt_rqst() Kernel Call                                            */
/* Usage:                                                                     */
/*    mrttest8                                                                */
/******************************************************************************/


#include <minix/config.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <lib.h>
```

```c
#include <minix/syslib.h>

_PROTOTYPE(int main, (int argc, char *argv []));
int main(argc, argv)
int argc;
char *argv[];
{
	mrt_pattr_t s_pattr, f_pattr;
	int rcode;
	pid_t pid, ppid;
	mrtpid_t mrtpid;
	mrt_pint_t pint;
	mrt_msg_t    msg;
/*----------------------------------------------------------------------------*/
/*                                     FATHER                                 */
/*----------------------------------------------------------------------------*/
	if( (pid = fork()) != 0)
		{
		ppid = getpid();
		printf("[FATHER]I am [%d],I will REQUEST my son [%d] in 10 secs\n"
		      ,ppid,pid);
		sleep(10);

		rcode = mrt_getpint(pid, &pint);
		if( rcode != 0)
			{
			printf("[FATHER] mrt_getpattr: pid=%d rcode=%5d.\n"
			     ,pid,rcode);
			exit(1);
			}
		mrtpid.pid   = pid;
		mrtpid.p_nr = pint.p_nr;

		s_pattr.flags       = MRT_P_REALTIME;
		s_pattr.baseprty    = MRT_PRI04;
		s_pattr.period      = 0;
		s_pattr.limit       = 0;
		s_pattr.deadline    = 0;
		s_pattr.watchdog    = -1;
		s_pattr.mq_size     = 3;
		s_pattr.mq_flags    = 0;

		rcode = mrt_setpattr(&s_pattr);
		if( rcode != 0)
			{
			printf("[FATHER] mrt_setpattr: rcode=%5d.\n", rcode);
			exit(1);
			}
		else
			printf("[FATHER] Process Attributes set\n");

		rcode = mrt_set2rt();
		if( rcode != 0 )
			{
			printf("[FATHER] mrt_set2rt rcode = %d\n",rcode);
			exit(1);
			}

		rcode = mrt_print("[FATHER] I will rqst my son...\n");
		if( rcode != 0 )
			{
			mrt_set2nrt();
			printf("[FATHER] mrt_print rcode = %d\n",rcode);
			exit(1);
			}

		rcode = mrt_rqst(mrtpid,&msg,10*200);
```

```
            mrt_set2nrt();
            printf("[FATHER] mrt_rqst rcode = %d\n",rcode);
            sleep(30);
            printf("[FATHER] bye\n");
            exit(0);
            }
/*----------------------------------------------------------------------------*/
/*                                    SON                                     */
/*----------------------------------------------------------------------------*/
      else
            {
            pid = getpid();
            printf("[SON] I am [%d] , and I will change to RT and put on sleep\n"
                       ,pid);
            s_pattr.flags       = MRT_P_REALTIME;
            s_pattr.baseprty    = MRT_PRI03;
            s_pattr.period      = 0;
            s_pattr.limit       = 0;
            s_pattr.deadline    = 0;
            s_pattr.watchdog    = -1;
            s_pattr.mq_size     = 3;
            s_pattr.mq_flags    = 0;

            rcode = mrt_setpattr(&s_pattr);
            if( rcode != 0)
                  {
                  printf("[SON] mrt_setpattr: rcode=%5d.\n", rcode);
                  exit(1);
                  }
            else
                  printf("[SON] Process Attributes set\n");

            rcode = mrt_set2rt();
            if( rcode != 0 )
                  {
                  printf("[SON] mrt_set2rt rcode = %d\n",rcode);
                  exit(1);
                  }

            rcode = mrt_print("[SON] I am going to sleep...\n");
            if( rcode != 0 )
                  {
                  mrt_set2nrt();
                  printf("[SON] mrt_print1 rcode = %d\n",rcode);
                  exit(1);
                  }

            rcode = mrt_sleep(60*200);
            if( rcode != 0 )
                  {
                  mrt_set2nrt();
                  printf("[SON] mrt_sleep rcode = %d\n",rcode);
                  exit(0);
                  }

            rcode = mrt_print("[SON] I wake up...\n");
            if( rcode != 0 )
                  {
                  mrt_set2nrt();
                  printf("[SON] mrt_print2 rcode = %d\n",rcode);
                  exit(1);
                  }

            rcode = mrt_set2nrt();
            printf("[SON]Bye\n");
            }
}
```

B.23. mrttest8b.c

This program forks into two processes, the *FATHER* and the *SON*:

The FATHER:

- Gets SON's process descriptor internal values to obtain its RT-PID.
- Sets its process descriptor attributes to convert itself into a RT-process.
- Converts itself into a RT-process.
- Prints a text on the system console using *mrt_print()* Kernel Call.
- Calls *mrt_rcv()* Kernel Call to receive a message from SON with a timeout of 2000 Timer ticks.
- Converts itself back into a NRT-process.

The *SON*:

- Sets its process descriptor attributes to convert itself into a RT-process.
- Converts itself into a RT-process.
- Prints a text on the system console using *mrt_print()* Kernel Call.
- Sleeps for 2000 Timer ticks.
- Converts itself back into a NRT-process.

Listing B.23: mrttest8b.c

```
/******************************************************************************/
/*                              mrttest8b.c                                   */
/* Test the mrt_rcv() Kernel Call                                             */
/* Usage:                                                                     */
/*    mrttest8b                                                               */
/******************************************************************************/

#include <minix/config.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
```

```
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <lib.h>
#include <minix/syslib.h>

_PROTOTYPE(int main, (int argc, char *argv []));

int main(argc, argv)
int argc;
char *argv[];
{
	mrt_pattr_t s_pattr, f_pattr;
	int rcode;
	pid_t pid, ppid;
	mrtpid_t mrtpid;
	mrt_pint_t pint;
	mrt_msg_t    msg;
	mrt_mhdr_t hmsg;

/*----------------------------------------------------------------------------*/
/*                                    FATHER                                  */
/*----------------------------------------------------------------------------*/
	if( (pid = fork()) != 0)
		{
		ppid = getpid();
		printf("[FATHER]I am [%d], I will RECEIVE msgs from [%d] in 10 secs\n"
			,ppid,pid);
		sleep(10);
		rcode = mrt_getpint(pid, &pint);
		if( rcode != 0)
			{
			printf("[FATHER] mrt_getpattr: pid= %d rcode=%5d.\n"
			     , pid,rcode);
			exit(1);
			}
		mrtpid.pid   = pid;
		mrtpid.p_nr = pint.p_nr;

		s_pattr.flags       = MRT_P_REALTIME;
		s_pattr.baseprty    = MRT_PRI04;
		s_pattr.period      = 0;
		s_pattr.limit       = 0;
		s_pattr.deadline    = 0;
		s_pattr.watchdog    = -1;
		s_pattr.mq_size     = 3;
		s_pattr.mq_flags    = 0;

		rcode = mrt_setpattr(&s_pattr);
		if( rcode != 0)
			{
			printf("[FATHER] mrt_setpattr: rcode=%5d.\n", rcode);
			exit(1);
			}
		else
			printf("[FATHER] Process Attributes set\n");

		rcode = mrt_set2rt();
		if( rcode != 0 )
			{
			printf("[FATHER] mrt_set2rt rcode = %d\n",rcode);
			exit(1);
			}

		rcode = mrt_print("[FATHER] I will RECEIVE a msg from my son...\n");
		if( rcode != 0 )
```

```
                {
                mrt_set2nrt();
                printf("[FATHER] mrt_print rcode = %d\n",rcode);
                exit(1);
                }

        rcode = mrt_rcv(mrtpid,&msg,&hmsg,10*200);
        mrt_set2nrt();
        printf("[FATHER] mrt_rcv rcode = %d\n",rcode);
        sleep(30);
        printf("[FATHER] bye\n");
        exit(0);
        }
/*--------------------------------------------------------------------------------*/
/*                                    SON                                         */
/*--------------------------------------------------------------------------------*/
    else
        {
        pid = getpid();
        printf("[SON] I am [%d], and I will change to RT and put on sleep\n"
                   ,pid);
        s_pattr.flags       = MRT_P_REALTIME;
        s_pattr.baseprty    = MRT_PRI03;
        s_pattr.period      = 0;
        s_pattr.limit       = 0;
        s_pattr.deadline    = 0;
        s_pattr.watchdog    = -1;
        s_pattr.mq_size     = 3;
        s_pattr.mq_flags    = 0;

        rcode = mrt_setpattr(&s_pattr);
        if( rcode != 0)
                {
                printf("[SON] mrt_setpattr: rcode=%5d.\n", rcode);
                exit(1);
                }
        else
                printf("[SON] Process Attributes set\n");


        rcode = mrt_set2rt();
        if( rcode != 0 )
                {
                printf("[SON] mrt_set2rt rcode = %d\n",rcode);
                exit(1);
                }

        rcode = mrt_print("[SON] I am going to sleep...\n");
        if( rcode != 0 )
                {
                mrt_set2nrt();
                printf("[SON] mrt_print1 rcode = %d\n",rcode);
                exit(1);
                }

        rcode = mrt_sleep(60*200);
        if( rcode != 0 )
                {
                mrt_set2nrt();
                printf("[SON] mrt_sleep rcode = %d\n",rcode);
                exit(0);
                }

        rcode = mrt_print("[SON] I wake up...\n");
        if( rcode != 0 )
                {
                mrt_set2nrt();
                printf("[SON] mrt_print2 rcode = %d\n",rcode);
```

```
                exit(1);
                }

        rcode = mrt_set2nrt();
        printf("[SON]Bye\n");
        }
}
```

B.24. mrttest8c.c

This program forks into two processes, the *FATHER* and the *SON*:

The FATHER:

- Gets SON's process descriptor internal values to obtain its RT-PID.
- Sets its process descriptor attributes to convert itself into a RT-process.
- Converts itself into a RT-process.
- Loops 10 Times
- Sleeps for 2000 Timer ticks on each loop.
- Send a request message to SON on each loop.
- Prints a text on the system console using *mrt_print()* Kernel Call.
- Converts itself back into a NRT-process.

The *SON*:

- Sets its process descriptor attributes to convert itself into a RT-process.
- Converts itself into a RT-process.
- Loops forever
- Calls *mrt_rcv()* Kernel Call to receive a message from FATHER.
- Prints the message content on the system console using *mrt_print()* Kernel Call.

Listing B.24: mrttest8c.c

```
/*****************************************************************************/
/*                               mrttest8c.c                                 */
/* Test the mrt_rqst() and mrt_rcv() Kernel Calls                            */
/* Usage:                                                                    */
/*    mrttest8c                                                              */
/*****************************************************************************/

#include <minix/config.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <lib.h>
#include <minix/syslib.h>

_PROTOTYPE(int main, (int argc, char *argv []));

int main(argc, argv)
int argc;
char *argv[];
{
      mrt_pattr_t s_pattr, f_pattr;
      int rcode,i;
      pid_t pid, ppid;
      mrtpid_t mrtpid;
      mrt_pint_t pint;
      mrt_msg_t    msg;
      mrt_mhdr_t hmsg;
      char text[50];

/*---------------------------------------------------------------------------*/
/*                                    FATHER                                 */
/*---------------------------------------------------------------------------*/
      if( (pid = fork()) != 0)
             {
             ppid = getpid();
             printf("[FATHER]I am [%d],I will REQUEST msgs to [%d]\n"
                    ,ppid,pid);
             rcode = mrt_getpint(pid, &pint);
             if( rcode != 0)
                    {
                    printf("[FATHER] mrt_getpattr: pid= %d rcode=%5d.\n"
                           ,pid,rcode);
                    exit(1);
                    }
             mrtpid.pid   = pid;
             mrtpid.p_nr = pint.p_nr;

             s_pattr.flags       = MRT_P_REALTIME;
             s_pattr.baseprty    = MRT_PRI04;
             s_pattr.period      = 0;
             s_pattr.limit       = 0;
             s_pattr.deadline    = 0;
             s_pattr.watchdog    = -1;
             s_pattr.mq_size     = 1;
             s_pattr.mq_flags    = 0;

             rcode = mrt_setpattr(&s_pattr);
             if( rcode != 0)
                    {
                    printf("[FATHER] mrt_setpattr: rcode=%5d.\n", rcode);
```

```
                exit(1);
                }
        else
                printf("[FATHER] Process Attributes set\n");

        rcode = mrt_set2rt();
        if( rcode != 0 )
                {
                printf("[FATHER] mrt_set2rt rcode = %d\n",rcode);
                exit(1);
                }

        mrt_print("[FATHER] I will REQUESTS 10 msgs to my son...\n");

        if( rcode != 0 )
                {
                mrt_set2nrt();
                printf("[FATHER] mrt_print rcode = %d\n",rcode);
                exit(1);
                }

        for( i = 0; i < 10; i++)
                {
                mrt_sleep(200*10);
                sprintf(msg.m_m7.m7ca1,"Hello son (%d)",i);
                rcode = mrt_rqst(mrtpid,&msg,MRT_FOREVER);
                if( rcode != 0)
                        sprintf(text,"[FATHER] msg %d NOT sent\n",i);
                else
                        sprintf(text,"[FATHER] msg %d sent\n",i);
                mrt_print(text);
                }

        mrt_set2nrt();
        sleep(30);
        printf("[FATHER] bye\n");
        exit(0);
        }
/*------------------------------------------------------------------------------*/
/*                                    SON                                       */
/*------------------------------------------------------------------------------*/
    else
        {
        pid = getpid();
        ppid = getppid();
        printf("[SON] I am [%d], and I will change to RT and put on sleep\n"
                ,pid);
        rcode = mrt_getpint(ppid, &pint);
        if( rcode != 0)
                {
                printf("[SON] mrt_getpattr: pid=%d rcode=%5d.\n"
                     ,ppid,rcode);
                exit(1);
                }
        mrtpid.pid  = ppid;
        mrtpid.p_nr = pint.p_nr;

        s_pattr.flags       = MRT_P_REALTIME;
        s_pattr.baseprty    = MRT_PRI04;
        s_pattr.period      = 0;
        s_pattr.limit       = 0;
        s_pattr.deadline    = 0;
        s_pattr.watchdog    = -1;
        s_pattr.mq_size     = 10;
        s_pattr.mq_flags    = 0;

        rcode = mrt_setpattr(&s_pattr);
        if( rcode != 0)
```

```
                        {
                        printf("[SON] mrt_setpattr: rcode=%5d.\n", rcode);
                        exit(1);
                        }
                else
                        printf("[SON] Process Attributes set\n");

                rcode = mrt_set2rt();
                if( rcode != 0 )
                        {
                        printf("[SON] mrt_set2rt rcode = %d\n",rcode);
                        exit(1);
                        }

                rcode = mrt_print("[SON] I am going to RECEIVE messages..\n");
                if( rcode != 0 )
                        {
                        mrt_set2nrt();
                        printf("[SON] mrt_print1 rcode = %d\n",rcode);
                        exit(1);
                        }

                i = 0;
                mrt_sleep(200*10);
                while(1)
                        {
                        rcode = mrt_rcv(mrtpid,&msg,&hmsg,MRT_FOREVER);
                        if( rcode != 0 )
                                {
                                mrt_set2nrt();
                                printf("[SON] mrt_rcv rcode = %d\n",rcode);
                                exit(1);
                                }
                        else
                                {
                                sprintf(text,"[SON]%d: %s",i,msg.m_m7.m7ca1);
                                mrt_print(text);
                                i++;
                                }
                        }

                }
}
```

B.25. mrttest8d.c

This program forks into two processes, the *FATHER* and the *SON*:

The FATHER:

- Gets SON's process descriptor internal values to obtain its RT-PID.
- Sets its process descriptor attributes to convert itself into a RT-process.
- Converts itself into a RT-process.
- Loops 10 Times

– Send an asynchronous request message to SON on each loop.

– Prints a text on the system console using *mrt_print()* Kernel Call.

– Converts itself back into a NRT-process.

The *SON*:

– Sets its process descriptor attributes to convert itself into a RT-process.

– Converts itself into a RT-process.

– Loops forever

– Calls *mrt_rcv()* Kernel Call to receive a message from FATHER.

– Prints the message content on the system console using *mrt_print()* Kernel Call.

Listing B.25: mrttest8d.c

```
/******************************************************************************/
/*                              mrttest8d.c                                   */
/* Test the mrt_arqst() and mrt_rcv() Kernel Calls                            */
/* Usage:                                                                     */
/*    mrttest8d                                                               */
/******************************************************************************/

#include <minix/config.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <lib.h>
#include <minix/syslib.h>

_PROTOTYPE(int main, (int argc, char *argv []));
int main(argc, argv)
int argc;
char *argv[];
{
      mrt_pattr_t s_pattr, f_pattr;
      int rcode,i;
      pid_t pid, ppid;
      mrtpid_t mrtpid;
      mrt_pint_t pint;
      mrt_msg_t    msg;
      mrt_mhdr_t hmsg;
      char text[50];
/*----------------------------------------------------------------------------*/
/*                                   FATHER                                   */
/*----------------------------------------------------------------------------*/
      if( (pid = fork()) != 0)
```

```
            {
            ppid = getpid();
            printf("[FATHER]I am [%d],I will REQUEST msgs to [%d]\n"
                    ,ppid,pid);
            rcode = mrt_getpint(pid, &pint);
            if( rcode != 0)
                  {
                  printf("[FATHER] mrt_getpattr: pid=%d rcode=%5d.\n"
                        ,pid,rcode);
                  exit(1);
                  }
            mrtpid.pid   = pid;
            mrtpid.p_nr  = pint.p_nr;

            s_pattr.flags       = MRT_P_REALTIME;
            s_pattr.baseprty    = MRT_PRI04;
            s_pattr.period      = 0;
            s_pattr.limit       = 0;
            s_pattr.deadline    = 0;
            s_pattr.watchdog    = -1;
            s_pattr.mq_size     = 1;
            s_pattr.mq_flags    = 0;

            rcode = mrt_setpattr(&s_pattr);
            if( rcode != 0)
                  {
                  printf("[FATHER] mrt_setpattr: rcode=%5d.\n", rcode);
                  exit(1);
                  }
            else
                  printf("[FATHER] Process Attributes set\n");

            rcode = mrt_set2rt();
            if( rcode != 0 )
                  {
                  printf("[FATHER] mrt_set2rt rcode = %d\n",rcode);
                  exit(1);
                  }

            mrt_print("[FATHER] I will REQUESTS 10 msgs to my son...\n");

            if( rcode != 0 )
                  {
                  mrt_set2nrt();
                  printf("[FATHER] mrt_print rcode = %d\n",rcode);
                  exit(1);
                  }

            mrt_sleep(200*10);

            for( i = 0; i < 10; i++)
                  {
                  sprintf(msg.m_m7.m7ca1,"Hello son (%d)",i);
                  rcode = mrt_arqst(mrtpid,&msg);
                  if( rcode != 0)
                        sprintf(text,"[FATHER] msg %d NOT sent\n",i);
                  else
                        sprintf(text,"[FATHER] msg %d sent\n",i);
                  mrt_print(text);
                  }

            mrt_set2nrt();
            sleep(30);

            printf("[FATHER] bye\n");
            exit(0);
            }
/*----------------------------------------------------------------------------*/
```

```
/*                                  SON                                         */
/*----------------------------------------------------------------------------*/
      else
            {
            pid = getpid();
            ppid = getppid();
            printf("[SON] I am [%d], and I will change to RT and put on sleep\n"
                   ,pid);
            rcode = mrt_getpint(ppid, &pint);
            if( rcode != 0)
                  {
                  printf("[SON] mrt_getpattr: pid= %d rcode=%5d.\n"
                      ,ppid,rcode);
                  exit(1);
                  }

            mrtpid.pid   = ppid;
            mrtpid.p_nr  = pint.p_nr;

            s_pattr.flags       = MRT_P_REALTIME;
            s_pattr.baseprty    = MRT_PRI04;
            s_pattr.period      = 0;
            s_pattr.limit       = 0;
            s_pattr.deadline    = 0;
            s_pattr.watchdog    = -1;
            s_pattr.mq_size     = 10;
            s_pattr.mq_flags    = 0;

            rcode = mrt_setpattr(&s_pattr);
            if( rcode != 0)
                  {
                  printf("[SON] mrt_setpattr: rcode=%5d.\n", rcode);
                  exit(1);
                  }
            else
                  printf("[SON] Process Attributes set\n");

            rcode = mrt_set2rt();
            if( rcode != 0 )
                  {
                  printf("[SON] mrt_set2rt rcode = %d\n",rcode);
                  exit(1);
                  }

            rcode = mrt_print("[SON] I am going to RECEIVE messages..\n");
            if( rcode != 0 )
                  {
                  mrt_set2nrt();
                  printf("[SON] mrt_print1 rcode = %d\n",rcode);
                  exit(1);
                  }

            i = 0;
            mrt_sleep(200*10);

            while(1)
                  {
                  rcode = mrt_rcv(mrtpid,&msg,&hmsg,MRT_FOREVER);
                  if( rcode != 0 )
                        {
                        mrt_set2nrt();
                        printf("[SON] mrt_rcv rcode = %d\n",rcode);
                        exit(1);
                        }
                  else
                        {
                        sprintf(text,"[SON]%d: %s",i,msg.m_m7.m7ca1);
                        mrt_print(text);
```

```
                    i++;
                    }
                }
            }
}
```

B.26. mrttest8e.c

This program forks into two processes, the *FATHER* and the *SON*:

The FATHER:

- Gets SON's process descriptor internal values to obtain its RT-PID.
- Sets its process descriptor attributes to convert itself into a RT-process.
- Converts itself into a RT-process.
- Sends a request message to SON.
- Calls *mrt_rcv()* Kernel Call to receive the reply from SON.
- Prints the reply on the system console using *mrt_print()* Kernel Call.
- Converts itself back into a NRT-process.

The *SON*:

- Sets its process descriptor attributes to convert itself into a RT-process.
- Converts itself into a RT-process.
- Calls *mrt_rcv()* Kernel Call to receive a message from FATHER.
- Sends a reply message to FATHER.
- Prints the request on the system console using *mrt_print()* Kernel Call.
- Converts itself back into a NRT-process.

Listing B.26: mrttest8e.c

```
/*************************************************************************/
/*                            mrttest8e.c                                 */
/* Test the mrt_rqst(), mrt_rcv(), mrt_rply() Kernel Calls                */
/* Usage:                                                                 */
```

```
/*     mrttest8e                                                              */
/******************************************************************************/

#include <minix/config.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <lib.h>
#include <minix/syslib.h>

_PROTOTYPE(int main, (int argc, char *argv []));

int main(argc, argv)
int argc;
char *argv[];
{
	mrt_pattr_t s_pattr, f_pattr;
	int rcode,i;
	pid_t pid, ppid;
	mrtpid_t mrtpid;
	mrt_pint_t pint;
	mrt_msg_t    msg;
	mrt_mhdr_t hmsg;
	char text[50];

/*----------------------------------------------------------------------------*/
/*                                   FATHER                                   */
/*----------------------------------------------------------------------------*/
	if( (pid = fork()) != 0)
		{
		ppid = getpid();
		printf("[FATHER]I am [%d],I will REQUEST to [%d]\n"
			 ,ppid,pid);

		rcode = mrt_getpint(pid, &pint);
		if( rcode != 0)
			{
			printf("[FATHER] mrt_getpattr: pid= %d rcode=%5d.\n"
				,pid,rcode);
			exit(1);
			}
		mrtpid.pid   = pid;
		mrtpid.p_nr  = pint.p_nr;

		s_pattr.flags       = MRT_P_REALTIME;
		s_pattr.baseprty    = MRT_PRI04;
		s_pattr.period      = 0;
		s_pattr.limit       = 0;
		s_pattr.deadline    = 0;
		s_pattr.watchdog    = -1;
		s_pattr.mq_size     = 1;
		s_pattr.mq_flags    = 0;

		rcode = mrt_setpattr(&s_pattr);
		if( rcode != 0)
			{
			printf("[FATHER] mrt_setpattr: rcode=%5d.\n", rcode);
			exit(1);
			}
		else
			printf("[FATHER] Process Attributes set\n");

		rcode = mrt_set2rt();
```

```
            if( rcode != 0 )
                   {
                   printf("[FATHER] mrt_set2rt rcode = %d\n",rcode);
                   exit(1);
                   }

            if( rcode != 0 )
                   {
                   mrt_set2nrt();
                   printf("[FATHER] mrt_print rcode = %d\n",rcode);
                   exit(1);
                   }

            mrt_sleep(200*10);

            sprintf(msg.m_m7.m7ca1,"Hello SON (%d)",i);
            rcode = mrt_rqst(mrtpid,&msg,MRT_FOREVER);
            if( rcode != 0)
                   sprintf(text,"[FATHER] mrt_rqst rcode %d\n",rcode);
            else
                   sprintf(text,"[FATHER] msg %d request\n",i);
            mrt_print(text);


            rcode = mrt_rcv(mrtpid,&msg,&hmsg,MRT_FOREVER);
            if( rcode != 0 )
                   sprintf(text,"[FATHER] mrt_rcv error %d\n",rcode);
            else
                   sprintf(text,"[FATHER] msg [%s] received\n",msg.m_m7.m7ca1);
            mrt_print(text);

            mrt_set2nrt();
            sleep(30);
            printf("[FATHER] bye\n");
            exit(0);
            }
/*---------------------------------------------------------------------------*/
/*                                     SON                                   */
/*---------------------------------------------------------------------------*/
      else
            {
            pid = getpid();
            ppid = getppid();
            printf("[SON] I am [%d] , and I will change to RT and put on sleep\n"
                   ,pid);
            rcode = mrt_getpint(ppid, &pint);
            if( rcode != 0)
                   {
                   printf("[SON] mrt_getpattr: pid= %d rcode=%5d.\n"
                         ,ppid,rcode);
                   exit(1);
                   }
            mrtpid.pid   = ppid;
            mrtpid.p_nr  = MRT_ANYPROC;

            s_pattr.flags       = MRT_P_REALTIME;
            s_pattr.baseprty    = MRT_PRI04;
            s_pattr.period      = 0;
            s_pattr.limit       = 0;
            s_pattr.deadline    = 0;
            s_pattr.watchdog    = -1;
            s_pattr.mq_size     = 10;
            s_pattr.mq_flags    = 0;

            rcode = mrt_setpattr(&s_pattr);
            if( rcode != 0)
                   {
                   printf("[SON] mrt_setpattr: rcode=%5d.\n", rcode);
```

```
                exit(1);
                }
        else
                printf("[SON] Process Attributes set\n");

        rcode = mrt_set2rt();
        if( rcode != 0 )
                {
                printf("[SON] mrt_set2rt rcode = %d\n",rcode);
                exit(1);
                }

        rcode = mrt_print("[SON] I am going to RECEIVE messages..\n");
        if( rcode != 0 )
                {
                mrt_set2nrt();
                printf("[SON] mrt_print1 rcode = %d\n",rcode);
                exit(1);
                }

        i = 0;
        rcode = mrt_rcv(mrtpid,&msg,&hmsg,MRT_FOREVER);
        if( rcode != 0 )
                sprintf(text,"[SON] mrt_rcv error %d\n",rcode);
        else
                sprintf(text,"[SON] msg [%s] received\n",msg.m_m7.m7ca1);
        mrt_print(text);

        sprintf(msg.m_m7.m7ca1,"Hello DAD (%d)",i);

        rcode = mrt_reply(hmsg.src,&msg);
        if( rcode != 0)
                sprintf(text,"[SON] mrt_reply rcode %d\n",rcode);
        else
                sprintf(text,"[SON] msg %d reply\n",i);
        mrt_print(text);

        mrt_set2nrt();
        printf("Message Header\n");
        printf("Source %d/%d \n",hmsg.src.pid,hmsg.src.p_nr);
        printf("Dest   %d/%d \n",hmsg.dst.pid,hmsg.dst.p_nr);
        printf("Mtype  %d\n",hmsg.mtype);
        printf("Mid\seqno %d/%d\n",hmsg.mid,hmsg.seqno);
        printf("Tstamp  %d\n",hmsg.tstamp);
        printf("Priority %d\n",hmsg.priority);
        printf("Deadline %d\n",hmsg.deadline);
        printf("Laxity   %d\n",hmsg.laxity);
        printf("Timeout  %d\n",hmsg.timeout);
        printf("[SON] bye\n");
        exit(0);
        }
}
```

B.27. mrttest9.c

This program forks into two processes, the *FATHER* and the *SON*:

The FATHER:

- Gets SON's process descriptor internal values to obtain its RT-PID.

– Sets its process descriptor attributes to convert itself into a RT-process.

– Converts itself into a RT-process.

– Loops 5 times

– Sends an up request message to SON on each loop.

– Converts itself back into a NRT-process.

The *SON*:

– Sets its process descriptor attributes to convert itself into a RT-process.

– Converts itself into a RT-process.

– Loops forever.

– Calls *mrt_rcv()* Kernel Call to receive a message from FATHER on each loop.

– Prints the request on the system console using *mrt_print()* Kernel Call on each loop.

Listing B.27: mrttest9.c

```
/****************************************************************************/
/*                              mrttest9.c                                  */
/* Test the mrt_uprqst() and mrt_rcv() Kernel Calls                         */
/* Usage:                                                                   */
/*      mrttest9                                                            */
/****************************************************************************/


#include <minix/config.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <lib.h>
#include <minix/syslib.h>

_PROTOTYPE(int main, (int argc, char *argv []));

int main(argc, argv)
int argc;
char *argv[];
{
	mrt_pattr_t s_pattr, f_pattr;
```

```
	int rcode,i;
	pid_t pid, ppid;
	mrtpid_t mrtpid;
	mrt_pint_t pint;
	mrt_msg_t    msg;
	mrt_mhdr_t hmsg;
	char text[50];

/*----------------------------------------------------------------------------*/
/*                                    FATHER                                  */
/*----------------------------------------------------------------------------*/
	if( (pid = fork()) != 0)
		{
		ppid = getpid();
		printf("[FATHER]I am [%d],I will UP REQUEST to my son [%d]\n"
			,ppid,pid);

		rcode = mrt_getpint(pid, &pint);
		if( rcode != 0)
			{
			printf("[FATHER] mrt_getpattr: pid= %d rcode=%5d.\n"
				, pid, rcode);
			exit(1);
			}

		mrtpid.pid   = pid;
		mrtpid.p_nr  = pint.p_nr;

		s_pattr.flags       = MRT_P_REALTIME;
		s_pattr.baseprty    = MRT_PRI04;
		s_pattr.period      = 0;
		s_pattr.limit       = 0;
		s_pattr.deadline    = 0;
		s_pattr.watchdog    = -1;
		s_pattr.mq_size     = 1;
		s_pattr.mq_flags    = 0;

		rcode = mrt_setpattr(&s_pattr);
		if( rcode != 0)
			{
			printf("[FATHER] mrt_setpattr: rcode=%5d.\n", rcode);
			exit(1);
			}
		else
			printf("[FATHER] Process Attributes set\n");

		rcode = mrt_set2rt();
		if( rcode != 0 )
			{
			printf("[FATHER] mrt_set2rt rcode = %d\n",rcode);
			exit(1);
			}

		if( rcode != 0 )
			{
			mrt_set2nrt();
			printf("[FATHER] mrt_print rcode = %d\n",rcode);
			exit(1);
			}

		mrt_sleep(200*10);

		for ( i = 0; i  < 5 ;  i++)
			{
			sprintf(msg.m_m7.m7ca1,"Hello SON (%d)",i);
			rcode = mrt_uprqst(mrtpid,&msg,MRT_PRI02);
			if( rcode != 0)
				sprintf(text,"[FATHER] mrt_uprqst rcode %d\n",rcode);
```

```
                else
                        sprintf(text,"[FATHER] msg %d UP request\n",i);
                mrt_print(text);
                }

        mrt_set2nrt();
        printf("[FATHER] bye\n");
        exit(0);
        }

/*----------------------------------------------------------------------------*/
/*                                    SON                                     */
/*----------------------------------------------------------------------------*/
    else
        {
        pid = getpid();
        ppid = getppid();
        printf("[SON] I am [%d] , and I will change to RT and put on sleep\n"
                ,pid);

        rcode = mrt_getpint(ppid, &pint);
        if( rcode != 0)
                {
                printf("[SON] mrt_getpattr: pid= %d rcode=%5d.\n"
                        ,ppid,rcode);
                exit(1);
                }
        mrtpid.pid   = ppid;
        mrtpid.p_nr  = MRT_ANYPROC;

        s_pattr.flags       = MRT_P_REALTIME;
        s_pattr.baseprty    = MRT_PRI04;
        s_pattr.period      = 0;
        s_pattr.limit       = 0;
        s_pattr.deadline    = 0;
        s_pattr.watchdog    = -1;
        s_pattr.mq_size     = 10;
        s_pattr.mq_flags    = 0;

        rcode = mrt_setpattr(&s_pattr);
        if( rcode != 0)
                {
                printf("[SON] mrt_setpattr: rcode=%5d.\n", rcode);
                exit(1);
                }
        else
                printf("[SON] Process Attributes set\n");

        rcode = mrt_set2rt();
        if( rcode != 0 )
                {
                printf("[SON] mrt_set2rt rcode = %d\n",rcode);
                exit(1);
                }

        rcode = mrt_print("[SON] I am going to RECEIVE messages..\n");
        if( rcode != 0 )
                {
                mrt_set2nrt();
                printf("[SON] mrt_print1 rcode = %d\n",rcode);
                exit(1);
                }

        while(1)
                {
                rcode = mrt_rcv(mrtpid,&msg,&hmsg,MRT_FOREVER);
                if( rcode != 0 )
                        sprintf(text,"[SON] mrt_rcv error %d\n",rcode);
```

```
                        else
                                sprintf(text,"[SON] msg [%s] received\n",msg.m_m7.m7ca1);
                        mrt_print(text);
                        }
                }
}
```

Appendix C: PERFORMANCE TESTS

This appendix describe the set of tests carried out on MINIX4RT. The first set that are focused on interrupt service times and Timer services timeliness were performed by Felipe Ieder and Nicolas Cesar as a laboratory practice of Real-Time Operating Systems at Facultad Regional Santa Fe of Universidad Tecnológica Nacional (FRSF-UTN) in May 2004. Because the FRSF-UTN does not have suitable equipment, the methods used for the tests are not rigurous, therefore the results are not accurated.

The set of test on RT-IPC perfomance were performed by the author in December 2005 using an up-to-date kernel.

C.1. Interrupt Service Time

The interrupt service time tests include different options for interrupt handlers.

- Event Driven Interrupt Handler
- Timer Driven Interrupt Handler
- Software Interrupt Handler

The tests were carried out using three different frequencies of interrupts.(1000, 5000 and 10000 [Hz]). The parallel port was used as the source of interrupts for the set of tests.

The equipment used for the test was:

- *LUNIX*: Intel Pentium III 800MHz, 256 MB RAM, Hard Disk 80GB,Chipset VIA Software: RT Linux with kernel 2.4.21 on Mandrake 9.1. This computer was used as the monitor machine.

– *PAP*: IBM Model 370C Notebook, Intel® DX4 75MHz on system board, AT Bus, Memory 8 MB, Diskette Drive 1.44MB, Hard Disk Drive 540MB 2.5-inch, PCMCIA One Type-III or two Type-II. Software: MINIX4RT (Kernel 28042004). This computer was used as tested machine.

The tests are performed under different kinds of load for the system:

– *No load (NOLoad)*: All background processes that are started at init are killed before the test.

– *I/O Diskette Load (DKTLoad): A* process access files on a diskette during the test. The diskette interrupt descriptor is Real-Time Timer Driven with a priority lower than the parallel port.

– *I/O Disk Load(HDLoad)*: A process access files on the hard disk during the test.

– *CPU Load(CPULoad)*: A script load the CPU without any I/O operation during the test.

C.1.1. Delay of RTLinux

Before testing MINIX4RT, a set of tests were carried out over the system that made the measurements. A parallel port loop was made using a DB25 male connector to connect D1 (pin 3) with ACK (pin 10) of the parallel port (see Figure C.1 and Figure C.2).

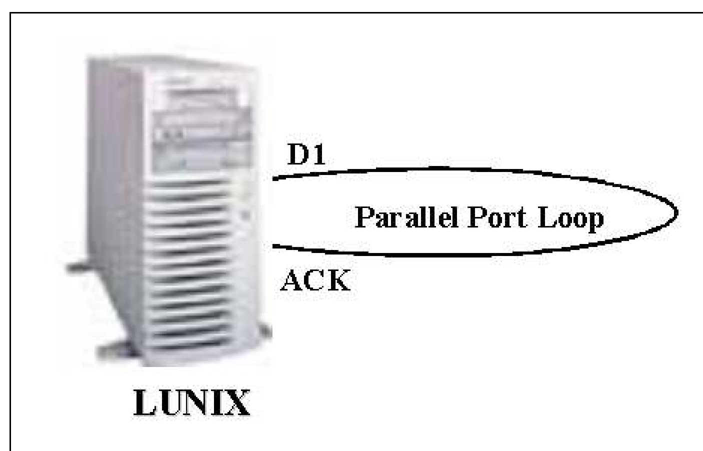


Figure C.1: Delay test of LUNIX (RTLinux)

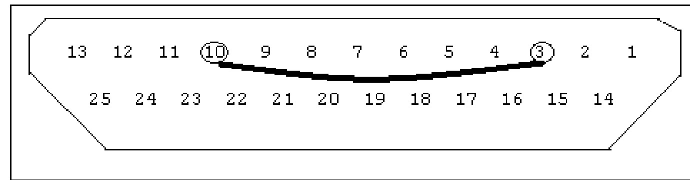


Figure C.2: DB25 Male Connector

A process in *LUNIX* machine generate a pulse on pin D1, that generates a parallel port interrupt. The time elapsed beetween the pulse generation and the service of the parallel port interrupt is considered as the delay of the *LUNIX* machine that must be subtracted from the measurements as is explained en Section C.1.2. The results of *LUNIX* tests are showed in Table C.1.

Table C.1: Delay of LUNIX (RTLinux).

	Frequecy [Hz]		
Delay [ns]	**10000**	**5000**	**1000**
Average	8379	8378	8340
Standard Deviation	670	606	212
Minimum	7872	7968	8032
Maximum	16576	17728	11648

C.1.2. Interrupt Service Time Tests of MINIX4RT

The tests were performed conecting the *LUNIX* machine with *PAP* machine (see Figure C.3) using a parallel port cable with the pinout detailled in Table C.2.

Table C.2: Testing Cable Pinout

DB25 Male 1 PIN		DB25 Male 2 PIN
2	<====>	15
3	<====>	13
4	<====>	12
5	<====>	10
6	<====>	11
15	<====>	2
13	<====>	3
12	<====>	4
10	<====>	5
11	<====>	6
25	<====>	25

The following is the sequence of events of the tests:

- *LUNIX* generates a pulse on a parallel port Data Bit (D3) that produces a parallel port interrupt on *PAP* (ACK).
- The parallel port interrupt handler of *PAP* generates a pulse pulse on a parallel port Data Bit (D3) that produces an interrupt on *LUNIX* (ACK).

The time elapsed beetween the pulse generation on *LUNIX* and the parallel port interrupt service in *LUNIX* is considered as the delay of the *LUNIX* machine plus the MINIX4RT interrupt service time.

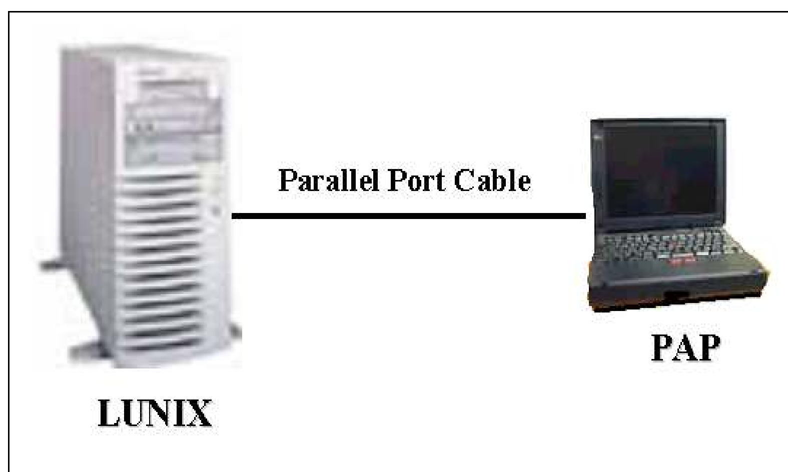


Figure C.3: Interrupt Service Time Tests of MINIX4RT

C.1.3. Interrupt Service Time of Software Interrupts

This section shows the tests results of Interrupt Service Time for Software interrupts. The RT-interrupt handler triggers a Software Interrupt that generates the response pulse on the parallel port.

Table C.4: Interrupt Service Time of Software Interrupts for 10000 [Hz]

Delay [ns]	DKTLoad	HDLoad	CPULoad	NOLoad
Average	41841	42777	43752	40974
Standard Deviation	6594	7948	8027	8104
Minimum	10112	30528	30368	30528
Maximum	100320	105920	117569	151456

Table C.5: Interrupt Service Time of Software Interrupts for 5000 [Hz]

Delay [ns]	DKTLoad	HDLoad	CPULoad	NOLoad
Average	33085	42916	38622	43732
Standard Deviation	3135	4030	2759	4216
Minimum	30592	31552	33152	30624
Maximum	69792	87200	58976	88288

Table C.6: Interrupt Service Time of Software Interrupts for 1000 [Hz]

Delay [ns]	DKTLoad	HDLoad	CPULoad	NOLoad
Average	38319	44443	39652	35986
Standard Deviation	6865	7523	2582	6534
Minimum	30688	30848	35712	30688
Maximum	62304	59584	51008	87264

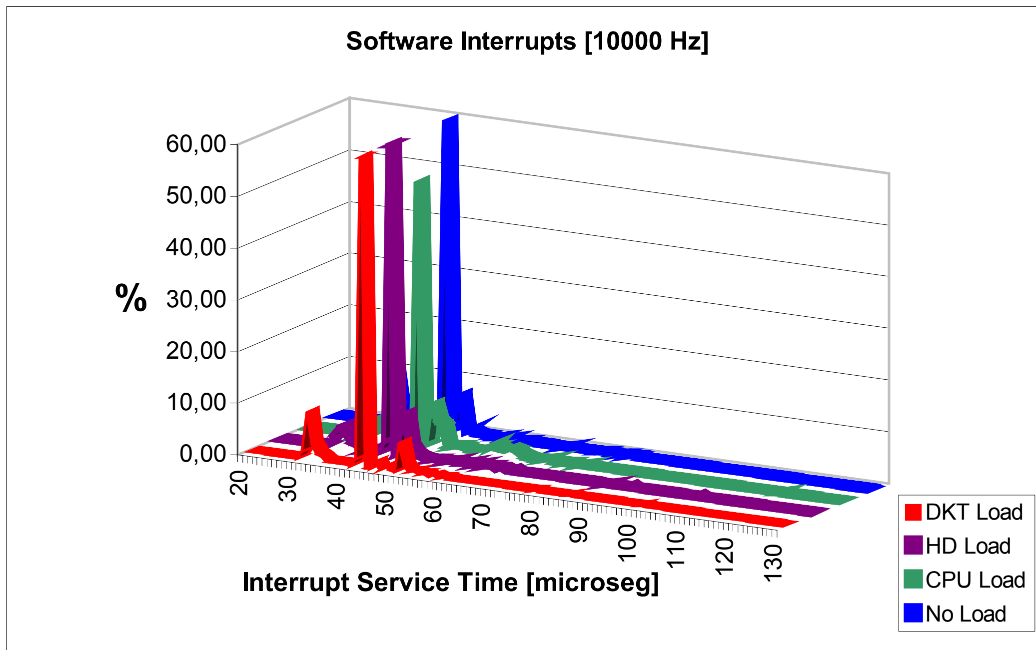


Figure C.4: Interrupt Service Time for Software Interrupts for 10000 [Hz]

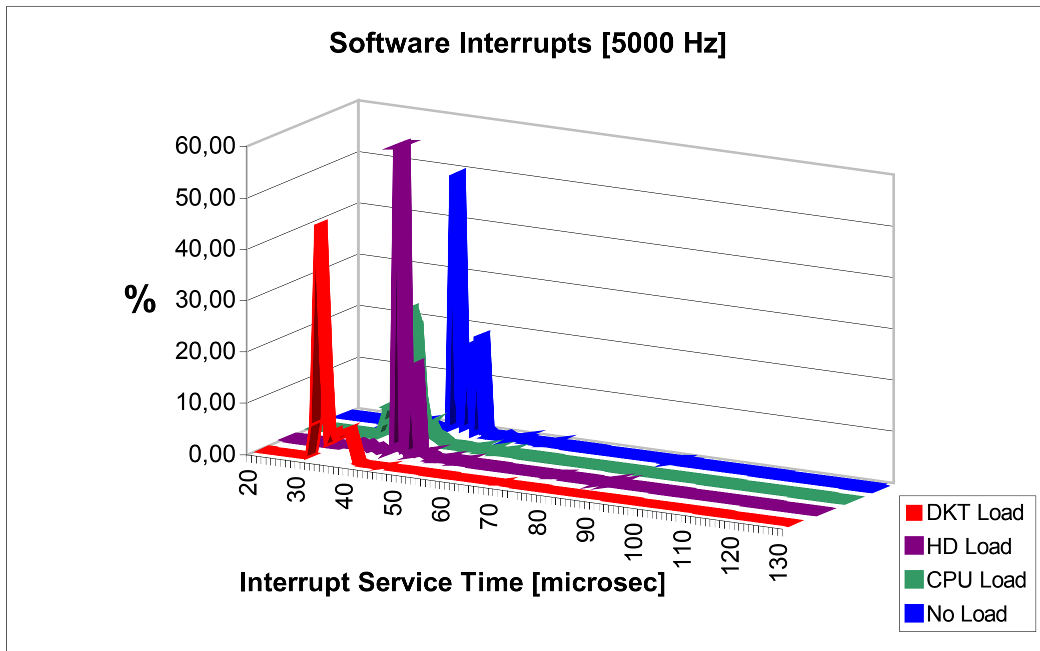


Figure C.5: Interrupt Service Time for Software Interrupts for 5000 [Hz]

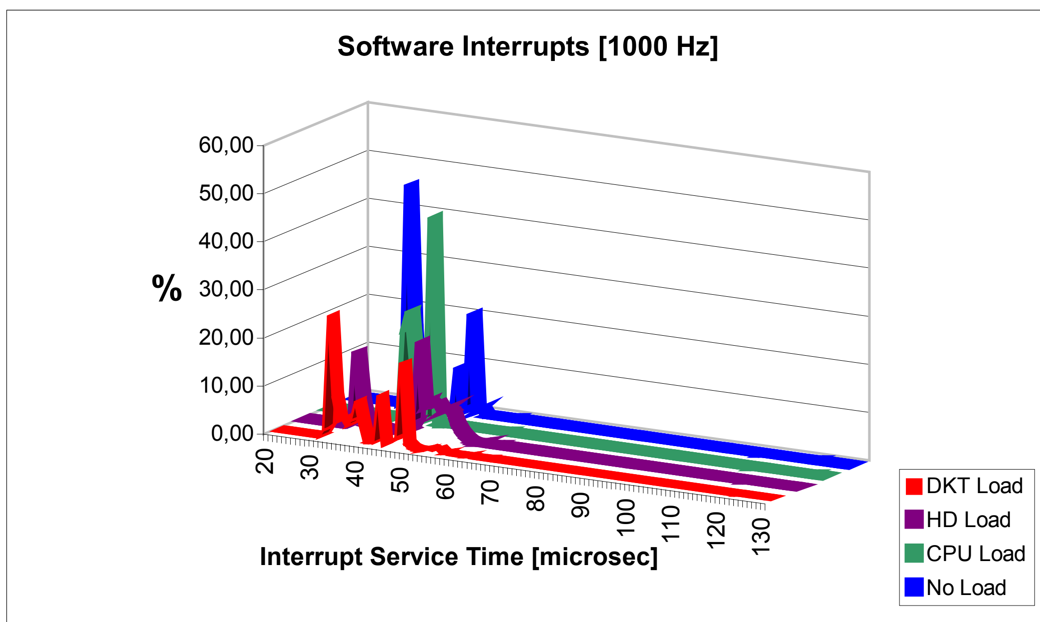


Figure C.6: Interrupt Service Time for Software Interrupts for 1000 [Hz]

C.1.4. Interrupt Service Time of Event-Driven Interrupts

This section describe the tests results of Interrupt Service Time for Event-Driven interrupts. The RT-interrupt handler generates the response pulse on the parallel port.

Table C.7: Interrupt Service Time of Event-Driven Interrupts for 10000 [Hz]

Delay [ns]	DKTLoad	HDLoad	CPULoad	NOLoad
Average	26395	38525	39909	37556
Standard Deviation	8709	9239	9309	9786
Minimum	21888	21888	21856	21376
Maximum	100352	89632	122816	55904

Table C.8: Interrupt Service Time of Event-Driven Interrupts for 5000 [Hz]

Delay [ns]	DKTLoad	HDLoad	CPULoad	NOLoad
Average	38685	41094	41706	40368
Standard Deviation	8303	6131	5355	6857
Minimum	21824	21888	23168	21536
Maximum	75392	82560	83424	52608

Table C.9: Interrupt Service Time of Event-Driven Interrupts for 1000 [Hz]

Delay [ns]	DKTLoad	HDLoad	CPULoad	NOLoad
Average	31092	33790	33506	29895
Standard Deviation	10294	10445	9100	10284
Minimum	21824	22144	23232	22144
Maximum	69664	90912	92896	86464

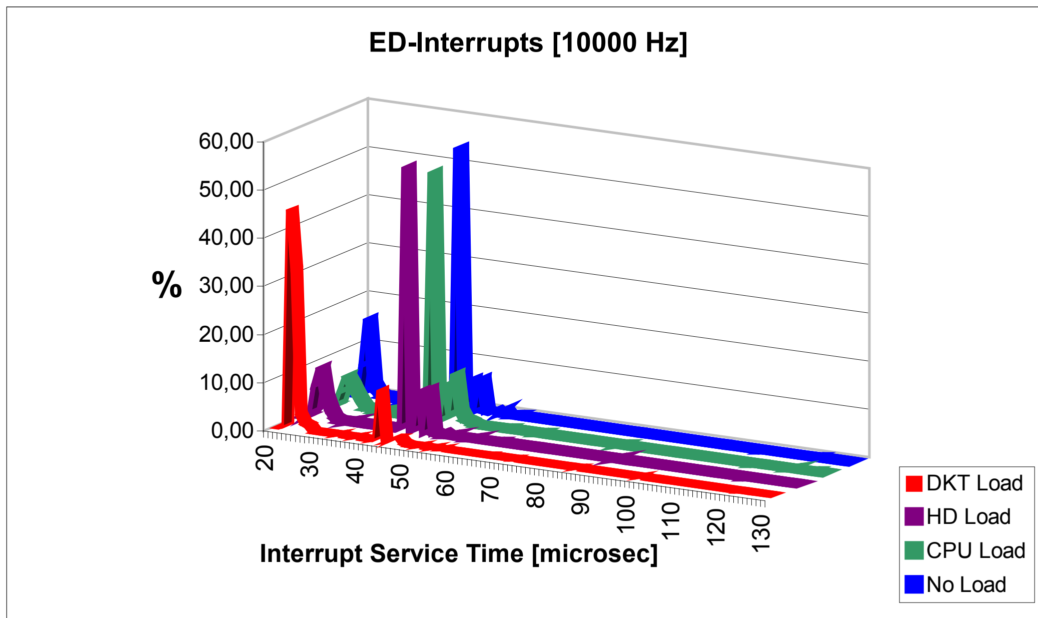


Figure C.7: Interrupt Service Time for ED-Interrupts [10000 Hz]

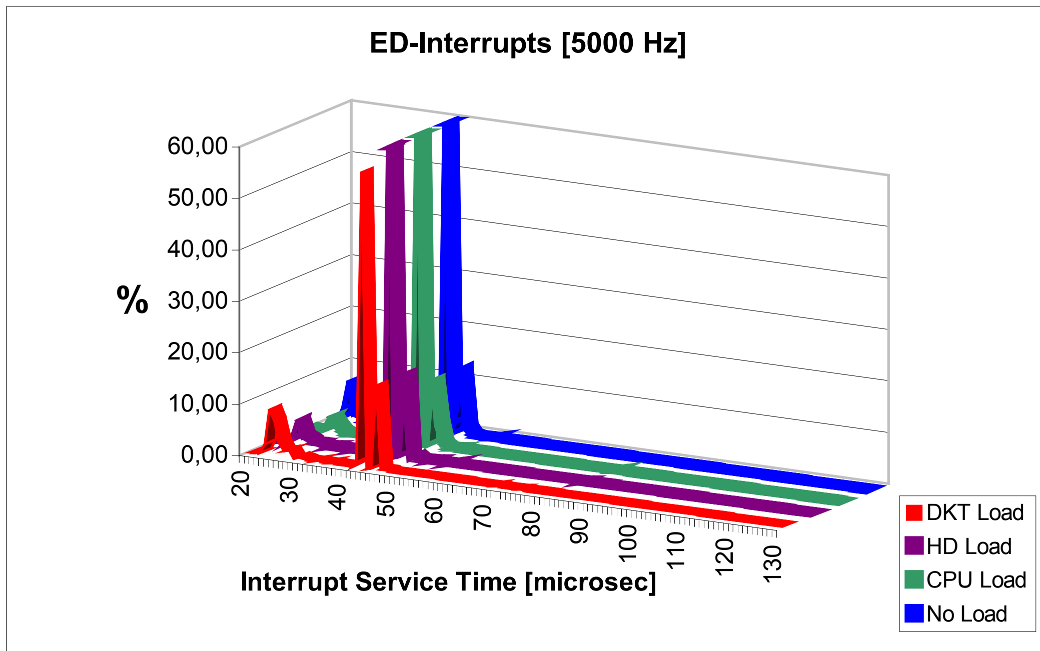


Figure C.8: Interrupt Service Time for ED-Interrupts [5000 Hz]

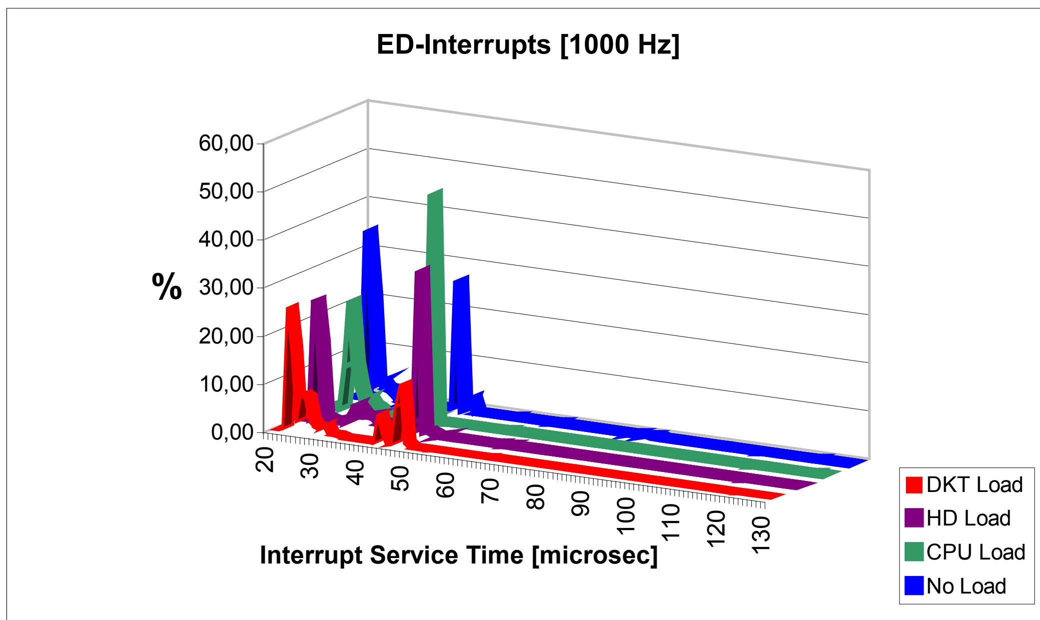


Figure C.9: Interrupt Service Time for ED-Interrupts [1000 Hz]

C.1.5. Interrupt Service Time of Timer-Driven Interrupts

This section describe the tests results Interrupt Service Time of Timer-Driven interrupts. In this tests, the RT-interrupt handler sets the interrupt descriptor to be executed

in the next Timer Interrupt. The TD-Interrupt descriptor generates the response pulse on the parallel port. The Timer interrupt frequency was 5000 [Hz].

Table C.10: Interrupt Service Time of Timer-Driven Interrupts for 1000 [Hz]

Delay [ns]	DKTLoad	HDLoad	CPULoad	NOLoad
Average	532139	462369	322101	256622
Standard Deviation	24242	27099	21207	23484
Minimum	488544	412416	288832	215616
Maximum	610688	540256	378624	314464

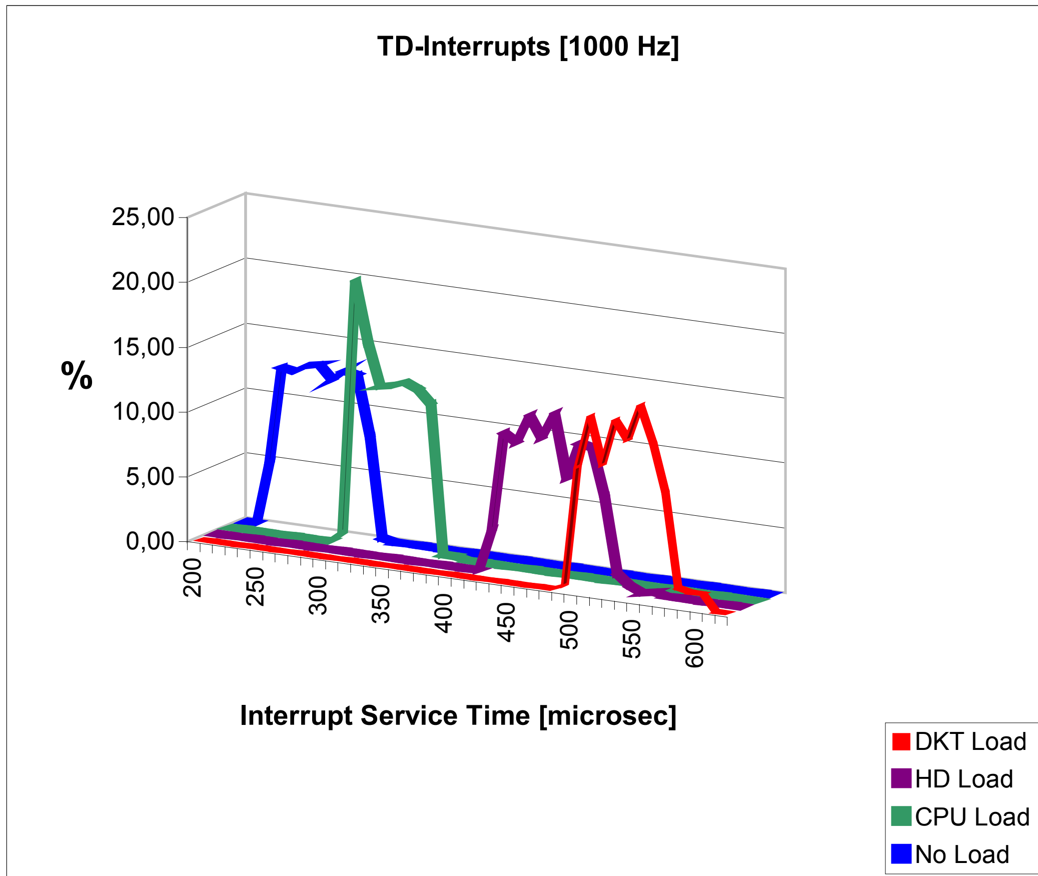


Figure C.10: Interrupt Service Time for TD-Interrupts [1000 Hz]

C.2. Virtual Timer Timeliness

A periodic RT-application correctness depends on the timeliness of its scheduling. MINIX4RT implements periodic processessing using Virtual Timers facilities. Therefore, the timeliness of Virtual Timers is an important measurement that evaluates performance of a RTOS.

These tests were carried out using the same equipment and loads as in Section C.1.

Table C.11: Virtual Timer Timeliness for 10000 [Hz]

Delay [ns]	DKTLoad	HDLoad	CPULoad	NOLoad
Average	99524	99524	99730	99732
Standard Deviation	6773	7371	7079	6528
Minimum	39968	53760	62880	60320
Maximum	123712	142848	127264	138688

Table C.12: Virtual Timer Timeliness for 5000 [Hz]

Delay [ns]	DKTLoad	HDLoad	CPULoad	NOLoad
Average	199456	199447	199465	199464
Standard Deviation	5228	9341	3335	1614
Minimum	167328	154592	155232	183904
Maximum	233632	246432	242880	213536

Table C.13: Virtual Timer Timeliness for 1000 [Hz]

Delay [ns]	DKTLoad	HDLoad	CPULoad	NOLoad
Average	999833	999772	999839	999799
Standard Deviation	7277	12511	3625	3896
Minimum	955424	936064	961024	975520
Maximum	1048064	1041728	1038496	1023712

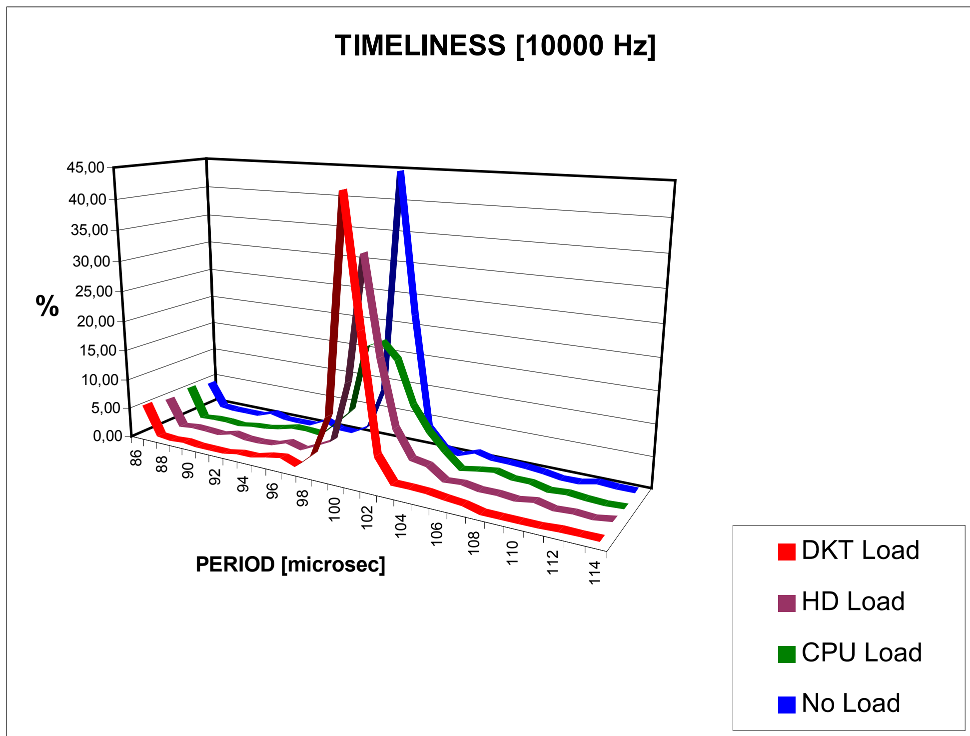


Figure C.11: Virtual Timer Timeliness for 10000 [Hz]

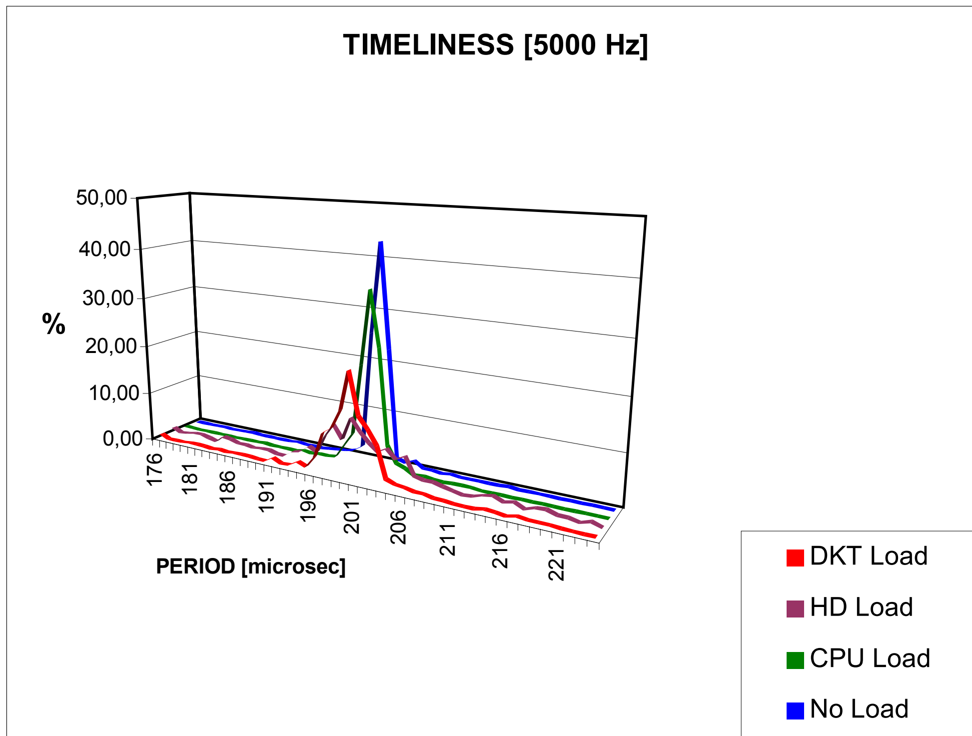


Figure C.12: Virtual Timer Timeliness for 5000 [Hz]

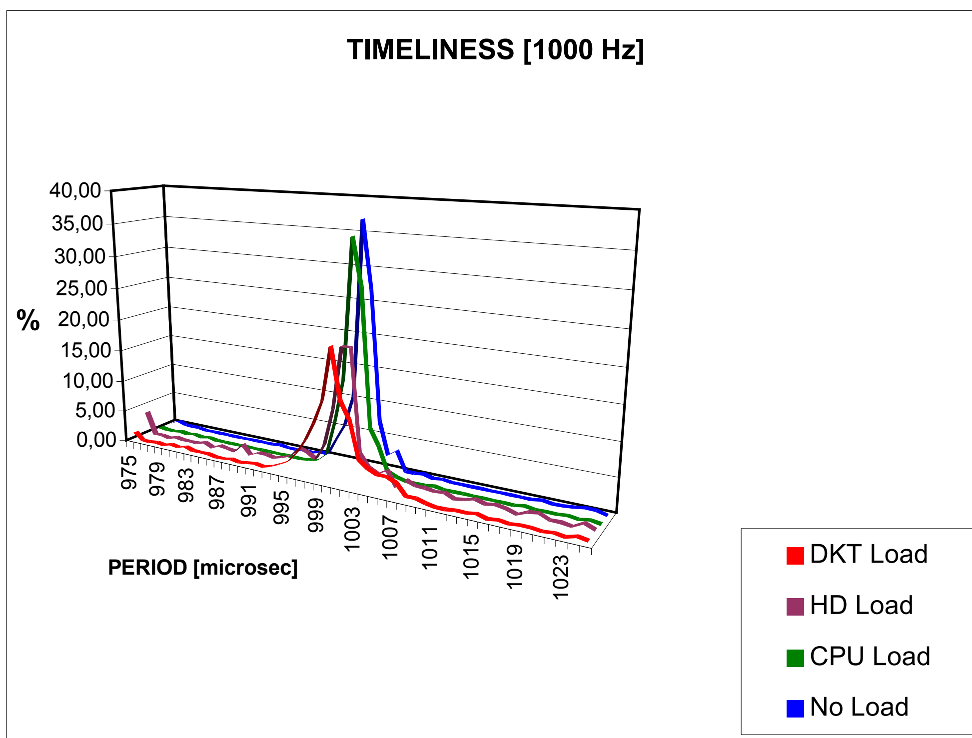


Figure C.13: Virtual Timer Timeliness for 1000 [Hz]

C.3. RT-IPC performance

The equipment used for the tests was:

- *PAP*: IBM Model 370C Notebook, Intel® DX4 75MHz on system board, AT Bus, Memory 8 MB, Diskette Drive 1.44MB, Hard Disk Drive 540MB 2.5-inch, PCMCIA One Type-III or two Type-II. Software: MINIX4RT (Kernel 20122005).

As in MINIX4RT messages can be transfered in two ways (synchronous or asynchronous) and with or without priority inheritance (BPIP), four sets of tests were carried out to evaluate the performance of each.

C.3.1. Synchronous Message Transfers Tests without BPIP and without Timeouts

Table C.14 shows the test results. Listing C.1 shows pseudocode of two programs:

- *FATHER:* Makes a loop sending request messages to SON and receiving the replies.

- *SON*: Makes a loop receiving request messages and sending replies.

Listing C.1**: Pseudocode of Programs to Test Synchronous Message Transfers without BPIP and without Timeouts**

```
father()
{
          ......
          before = get_sys_ticks();   /* pseudo code */
          for( i = 0; i < 10000; i++)
                      {
                      mrt_rqst(SON,&rqst,MRT_FOREVER);
                      mrt_rcv(SON,&rply,&hrply,MRT_FOREVER);
                      }
          testime = get_sys_ticks() - before;   /* pseudo code */
          ......
}
son()
{
          ......
              while(1)
                      {
                      mrt_rcv(FATHER,&rqst,&hrqst,MRT_FOREVER);
                      mrt_reply(FATHER,&rply);
                      }
          ......
}
```

Table C.14: Synchronous Message Transfers Test Results without BPIP and without Timeouts

Clock Freq	1000	ticks/s
Testing time	1925	ticks
mrt_arqst	10000	msgs
mrt_reply	10000	msgs
Total msg	20000	msgs
Throughput	10390	msgs/s
RT-schedulings	20000	scheds
Time by msg	96	microsec/msg

C.3.2. Asynchronous Message Transfers Tests without BPIP and without Timeouts

Table C.15 shows the test results. Listing C.2 shows pseudocode of two processes:

- *FATHER:* makes two loops. An internal loop sending 10 asynchronous request messages to SON. An an external loop receiving replies.
- *SON*: makes two loops. An internal loop receiving 10 messages from FATHER. An an external loop sending replies.

Listing C.2**: Pseudocode of Programs to Test Asynchronous Message Transfers without BPIP and without Timeouts**

```
father()
{
          ......
          before = get_sys_ticks();   /* pseudo code */
          for( i = 0; i < 1000; i++)
                      {
                      for( j = 0; j < 10; j++)
                              mrt_arqst(SON,&rqst);
                      mrt_rcv(SON,&rply,&hrply,MRT_FOREVER);
                      }
          testime = get_sys_ticks() - before;   /* pseudo code */
          ......
}
son()
{
          ......
              while(1)
                      {
                      for( j = 0; j < 10; j++)
                             mrt_rcv(mrtpid,&rqst,&hrqst,MRT_FOREVER);
                      mrt_reply(mrtpid,&rply);
                      }
          ......
}
```

Table C.15: Asynchronous Message Transfers Test Results without BPIP and without Timeouts

Clock Freq	1000	ticks/s
Testing time	993	ticks
mrt_arqst	10000	msgs
mrt_reply	1000	msgs
Total msg	11000	msgs
Throughput	11078	msgs/s
RT-schedulings	4000	scheds
Time by msg	90	microsec/msg

C.3.3. Synchronous Message Transfers Tests with BPIP and without Timeouts

This test was carried out using the programs of C.3.1. Table C.16 shows the test results.

Table C.16: Synchronous Message Transfers Test Results with BPIP and without Timeouts

Clock Freq	1000	ticks/s
Testing time	2164	ticks
mrt_arqst	10000	msgs
mrt_reply	10000	msgs
Total msg	20000	msgs
Throughput	9242	msgs/s
RT-schedulings	20000	scheds
Time by msg	108	microsec/msg

C.3.4. Asynchronous Message Transfers Tests with BBIP and without Timeouts

This test was carried out using the programs of C.3.2. Table C.17 shows the test results.

Table C.17: Asynchronous Message Transfers Test Results with BPIP and without Timeouts

Clock Freq	1000	ticks/s
Testing time	1008	ticks
mrt_arqst	10000	msgs
mrt_reply	1000	msgs
Total msg	11000	msgs
Throughput	10912	msgs/s
RT-schedulings	4000	scheds
Time by msg	91	microsec/msg

C.3.5. Request/Receive Tests with BPIP and without Timeouts

Table C.18 shows the test results. Listing C.3 shows pseudocode of two programs:

– *FATHER:* Makes a loop sending request messages to SON and receiving the replies using the *mrt_rqrcv()* kernel call.

– *SON*: Makes a loop receiving request messages and sending replies.

Listing C.3**: Pseudocode of Programs to Test Request/Receive with BPIP and without Timeouts**

```
father()
{
          ......
          before = get_sys_ticks();   /* pseudo code */
          for( i = 0; i < 10000; i++)
                      {
                      mrt_rqrcv(SON,&rqst,&rply,&hrply,MRT_FOREVER);
                      }
          testime = get_sys_ticks() - before;   /* pseudo code */
          ......
}
son()
{
          ......
              while(1)
                      {
                      mrt_rcv(FATHER,&rqst,&hrqst,MRT_FOREVER);
                      mrt_reply(FATHER,&rply);
                      }
          ......
}
```

Table C.18 Request/Receive Test Results with BPIP and without Timeouts

Clock Freq	1000	ticks/s
Testing time	1749	ticks
mrt_rqrcv	10000	msgs
mrt_reply	10000	msgs
Total msg	20000	msgs
Throughput	11435	msgs/s
RT-schedulings	20000	scheds
Time by msg	87	microsec/msg

C.3.6. Synchronous Message Transfers Tests with BPIP and with Timeouts

Table C.19 shows the test results. Listing C.4shows pseudocode of two processes:

Listing C.4**: Pseudocode of Programs to Test Synchronous Message Transfers with BPIP and with Timeouts**

```
father()
{
          ......
          before = get_sys_ticks();   /* pseudo code */
          for( i = 0; i < 10000; i++)
                      {
                      mrt_rqst(SON,&rqst, 60*1000);
```

```
                    mrt_rcv(SON,&rply,&hrply,60*1000);
                    }
         testime = get_sys_ticks() - before;   /* pseudo code */
         ......
}
son()
{
         ......
             while(1)
                    {
                    mrt_rcv(FATHER,&rqst,&hrqst, 60*1000);
                    mrt_reply(FATHER,&rply);
                    }
         ......
}
```

Table C.19: Synchronous Message Transfers Test Results with BPIP and with Timeouts

Clock Freq	1000	ticks/s
Testing time	2222	ticks
mrt_arqst	10000	msgs
mrt_reply	10000	msgs
Total msg	20000	msgs
Throughput	9000	msgs/s
RT-schedulings	20000	scheds
Time by msg	111	microsec/msg

C.3.7. Asynchronous Message Transfers Tests with BBIP and with Timouts

Table C.20 shows the test results. Listing C.5 shows pseudocode of two processes:

Listing C.5**: Pseudocode of Programs to Test Asynchronous Message Transfers with BPIP and with Timeouts**

```
father()
{
         ......
         before = get_sys_ticks();   /* pseudo code */
         for( i = 0; i < 1000; i++)
                    {
                    for( j = 0; j < 10; j++)
                           mrt_arqst(SON,&rqst);
                    mrt_rcv(SON,&rply,&hrply,60*1000);
                    }
         testime = get_sys_ticks() - before;   /* pseudo code */
         ......
}
son()
{
         ......
             while(1)
                    {
                    for( j = 0; j < 10; j++)
                          mrt_rcv(mrtpid,&rqst,&hrqst,60*1000);
                    mrt_reply(mrtpid,&rply);
                    }
}
```

Table C.20: Asynchronous Message Transfers Test Results with BPIP and without Timouts

Clock Freq	1000	ticks/s
Testing time	1071	ticks
mrt_arqst	10000	msgs
mrt_reply	1000	msgs
Total msg	11000	msgs
Throughput	10270	msgs/s
RT-schedulings	4000	scheds
Time by msg	97	microsec/msg

C.3.8. Test Request/Receive Tests with BPIP and with Timeouts

Table C.21 shows the test results. Listing C.6 shows pseudocode of two processes:

Listing C.6**: Pseudocode of Programs to Test Synchronous Message Transfers with BPIP and with Timeouts**

```
father()
{
         ......
         before = get_sys_ticks();   /* pseudo code */
         for( i = 0; i < 10000; i++)
                    {
                    mrt_rqrcv(SON,&rqst,.&rply,&hrply,60*1000);
                    }
         testime = get_sys_ticks() - before;   /* pseudo code */
         ......
}
son()
{
         ......
             while(1)
                    {
                    mrt_rcv(FATHER,&rqst,&hrqst, 60*1000);
                    mrt_reply(FATHER,&rply);
                    }
         ......
}
```

Table C.21: Synchronous Message Transfers Test Results with BPIP and with Timeouts

Clock Freq	1000	ticks/s
Testing time	1754	ticks
mrt_rqrcv	10000	msgs
mrt_reply	10000	msgs
Total msg	20000	msgs
Throughput	11402	msgs/s
RT-schedulings	20000	scheds
Time by msg	87	microsec/msg

C.3.9. RT-IPC Tests Results

Figure C.11 shows a comparison of Message Time for all tests.

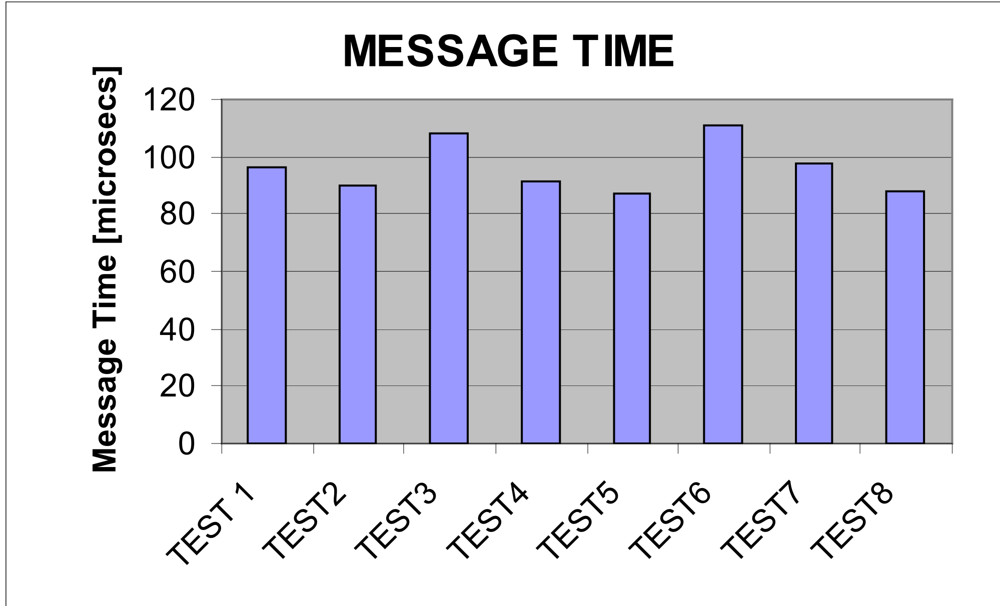


Figure C.11: Virtual Timer Timeliness for 10000 [Hz]

Figure C.12 shows a comparison of Message Thoughput for all tests.

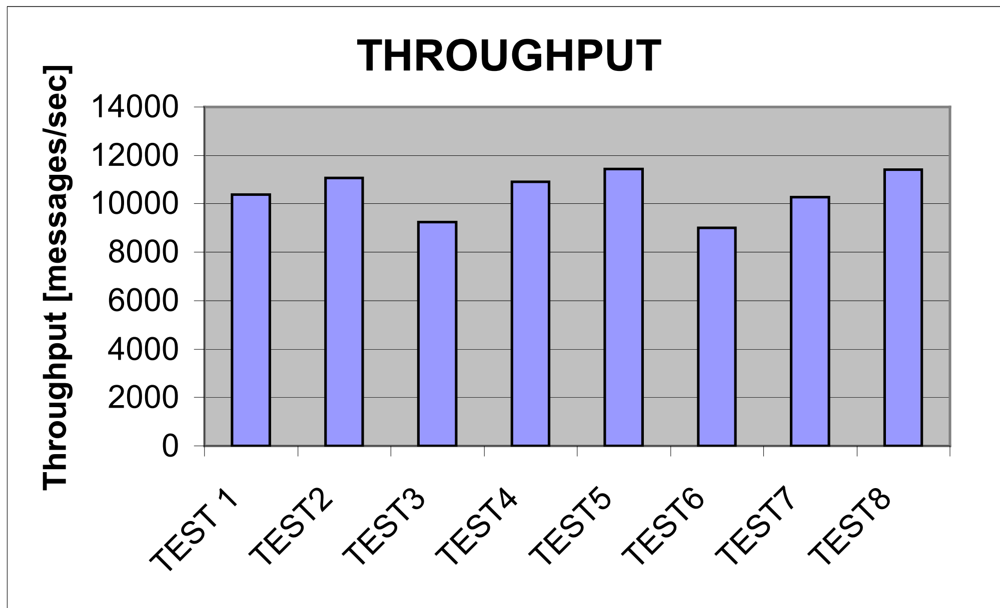


Figure C.12: Virtual Timer Timeliness for 10000 [Hz]

Appendix D: SYSTEM DATA STRUCTURES

D.1. User-Level Data Structures

The following data structures are used by User-Level programs.

D.1.1. System-wide Data Structures

Listing D.1: **System-wide Data Structures.**

```
/*--------------------------- SYSTEM WIDE STATISTICS -------------------------*/
struct mrt_sysstat_s {
	lcounter_t		scheds;		/* schedules counter				*/
	lcounter_t		messages;	/* message counter - ID count		*/
	lcounter_t		interrupts;	/* Interrupt counter				*/
	lcounter_t		ticks;		/* Less Significative tick counter	*/
	lcounter_t		highticks;	/* More Significative tick counter	*/
	lcounter_t		idlemax;	/* Maximun idle counter				*/
	lcounter_t		idlelast;	/* last idle counter				*/
	};
typedef struct mrt_sysstat_s mrt_sysstat_t;

/*--------------------------- SYSTEM OPERATIONAL VALUES ----------------------*/
struct mrt_sysval_s {
	unsigned int flags;
	bitmap16_t		virtual_PIC; /* Virtual PIC for MINIX			*/
	lcounter_t		PIT_latency; /* PIT latency in Hz between two reads */
	unsigned		PIT_latch;	/* TIMER_FREQ/MRT_tickrate			*/
	scounter_t		tickrate;	/* Real-Time ticks by second		*/
	scounter_t		harmonic;	/* MRT_tickrate = MRT_harmonic x HZ	*/
	scounter_t		refresh;	/* Idle refresh tick count			*/
	};
typedef struct mrt_sysval_s mrt_sysval_t;
```

D.1.2. Interrupt-related Data Structures

Listing D.2: **Interrupt-related Data Structures.**

```
/*--------------------------- IRQ SPECIFIED ARGUMENTS  ------------------------*/
struct mrt_irqattr_s {
	lcounter_t		period;		/* For Timer Driven period in ticks	*/
	proc_nbr_t		task;		/* Real Time task number			*/
```

```
      proc_nbr_t          watchdog;    /* Interrupt Watch dog process         */
      priority_t          priority;    /* RT or NRT priority                  */
      irq_type_t          irqtype;     /* IRQ Type                            */
      char                name[MAXPNAME];/* name of the RT driver             */
      };
typedef struct mrt_irqattr_s mrt_irqattr_t;

/*--------------------------- IRQ STATISTICS ----------------------------------*/
struct mrt_irqstat_s {
      lcounter_t          count;       /* Interrupt counter                   */
      scounter_t          maxshower;   /* Maximum shower value                */
      lcounter_t          mdl;         /* Missed Deadlines                    */
      lcounter_t          timestamp;   /* Last Interrupt timestamp            */
      lcounter_t          maxlat;      /* Maximun Interrupt latency PIT Hz    */
      int                 reenter;     /* Maximun reentrancy level            */
      };
typedef struct mrt_irqstat_s mrt_irqstat_t;

/*--------------------------- IRQ INTERNAL ------------------------------------*/
struct mrt_irqint_s  {
      int                 irq;         /* irq number                          */
      scounter_t          harmonic;    /* MRT_sv.harmonic when mrtode starts  */
      int                 vtimer;      /* VT assigned for Timer Driven IRQs   */
      int                 flags;       /* MRT_ENQUEUED, MRT_TRIGGEREG         */
      };
typedef struct mrt_irqint_s mrt_irqint_t;
```

D.1.3. Process-related Data Structures

Listing D.3**: Process-related Data Structures.**

```
struct mrtpid_s {
      pid_t               pid;               /* request destination PID      */
      int                 p_nr;              /* request destination number   */
      };
typedef struct mrtpid_s mrtpid_t;

struct mrt_pattr_s {
      int                 flags;             /* Real Time Flags              */
      priority_t          baseprty;          /* Real Time BASE priority      */
      lcounter_t          period;            /* period in RT-ticks           */
      scounter_t          limit;             /* max number of process
schedulings*/
      lcounter_t          deadline;          /* process deadline             */
      int                 watchdog;          /* Watchdog process             */
      scounter_t          mq_size;           /* Message Queue Size           */
unsigned int        mq_flags;          /* Message Queue Policy Flags    */
      };
typedef struct mrt_pattr_s  mrt_pattr_t;

struct mrt_pstat_s {
      lcounter_t          scheds;            /* number of schedules          */
      lcounter_t          mdl;               /* Missed DeadLines             */
      lcounter_t          timestamp;         /* Last schedule timestamp in ticks*/
      lcounter_t          maxlat;            /* Maximun latency in timer Hz  */
      scounter_t          minlax;            /* Minimun laxity in timer Hz   */
      lcounter_t          msgsent;           /* Messages sent by the process */
      lcounter_t          msgrcvd;           /* Messages received by the process*/
      };
typedef struct mrt_pstat_s mrt_pstat_t;

struct mrt_pint_s {
      int                 vt;                /* VT timer ID for periodic process*/
      priority_t          priority;          /* Real Time EFECTIVE priority  */
      scounter_t          mqID;              /* Message Queue ID             */
```

```
	int			p_nr;			/* process slot				*/
	};
typedef struct mrt_pint_s mrt_pint_t;
```

D.1.4. Kernel Calls-related Data Structures

Listing D.4: Kernel Calls-related Data Structures.

```
struct mrt_rqst_s {
	int			p_nr;			/* request destination number	*/
	pid_t			pid;			/* request destination PID	*/
	mrt_msg_t		*msg;			/* Message to be sent		*/
	lcounter_t		timeout;		/* rqst timeout in ticks	*/
	};
typedef struct mrt_rqst_s mrt_rqst_t;

struct mrt_rply_s {
	int			p_nr;			/* request destination number	*/
	pid_t			pid;			/* request destination PID	*/
		mrt_msg_t		*msg;			/* Message to be sent
	*/
	};
typedef struct mrt_rply_s mrt_rply_t;
typedef struct mrt_rply_s mrt_sign_t;

struct mrt_rcv_s {
	int			p_nr;			/* requested source proc number	*/
	pid_t			pid;			/* requested source proc PID	*/
		mrt_msg_t		*msg;			/* pointer to the msg buffer
	*/
	mrt_mhdr_t		*hdr;			/* pointer to the msg header  buf*/
	lcounter_t		timeout;		/* receive timeout in ticks	*/
	};
typedef struct mrt_rcv_s mrt_rcv_t;

struct mrt_uprq_s {
	int			p_nr;			/* request destination number	*/
	pid_t			pid;			/* request destination PID	*/
		mrt_msg_t		*msg;			/* pointer to the buffer for
the msg*/
	int			priority;		/* message RT priority		*/
	lcounter_t		deadline;		/* process deadline		*/
	lcounter_t		laxity;			/* process laxity in timer Hz	*/
	};
typedef struct mrt_uprq_s mrt_uprq_t;
```

D.1.5. Message-related Data Structures

Listing D.5: Message-related Data Structures.

```
struct mrt_mhdr_s{
	mrtpid_t		src;
	mrtpid_t		dst;		/* destination process			*/
	unsigned int		mtype;		/* what kind of message is it		*/
	lcounter_t		mid;		/* message ID				*/
	scounter_t		seqno;		/* msg sequence nbr			*/
	lcounter_t		tstamp;		/* sent timestamp			*/
	priority_t		priority;	/* sender's priority			*/
	lcounter_t		deadline;	/* sender's deadline			*/

	lcounter_t		laxity;		/* sender's laxity			*/
```

```
	};
typedef struct mrt_mhdr_s mrt_mhdr_t;


union mrt_msg_u {
	mess_1 m_m1;
	mess_2 m_m2;
	mess_3 m_m3;
	mess_4 m_m4;
	mess_5 m_m5;
	mess_6 m_m6;
	mess_7 m_m7;
	} ;

typedef union mrt_msg_u mrt_msg_t;

/* Real-Time Message Structure */
struct mrt_msgd_s {
	mrt_mhdr_t		hdr;
	mrt_msg_t		m_u;
} ;
typedef struct mrt_msgd_s mrt_msgd_t;
```

D.2. Kernel-Level Data Structures

The following data structures are used by Kernel-Level programs.

D.2.1. System-wide Data Structures

Listing D.6: System-wide Data Structures.

```
 struct MRT_sysstat_s {
	lcounter_t		scheds;		/* schedules counter			*/
	lcounter_t		messages;	/* message counter - ID count		*/
volatile lcounter_t		interrupts;	/* Interrupt counter			*/
volatile lcounter_t		ticks;		/* Less Significative tick counter	*/
volatile lcounter_t		highticks;	/* More Significative tick counter	*/
	lcounter_t		idlemax;	/* idle maximum value			*/
	lcounter_t		idlelast;	/* last counter				*/
	};
typedef struct MRT_sysstat_s MRT_sysstat_t;

typedef	struct {
	/* DON'T MOVE flags FROM THE FIRST POSITION IN THE STRUCT !! */
	unsigned int flags; /* MRT_RTMODE	0x0001 RT processing mode	*/
			    /* MRT_DBG232	0x0002 Flag to allow printf232	*/
			    /* MRT_LATENCY	0x0004 Flag to allow latency comp. */
			    /* MRT_MINIXCLI	0x0008 MINIX virtual IF 1=CLI	*/
			    /* MRT_NEWINT	0x0010 an int has occurs dur int
flush*/
			    /* MRT_FLUSHLCK	0x0020 set when MRT_flush_int runs*/
			    /* MRT_NOFLUSH	0x0040 to avoid MRT_irq_flush	*/
			    /* MRT_NEEDSCHED	0x0080 to invoke the scheduler	*/
			    /* MRT_RTSCHED	0x0100 The current proc has been */
			    /* selected by thge RT scheduler MRT_pick_proc	*/
	bitmap16_t		virtual_PIC; /* Virtual PIC for MINIX		*/
	lcounter_t		PIT_latency; /* PIT latency in Hz between two reads */
	unsigned		PIT_latch;	/* TIMER_FREQ/MRT_tickrate		*/
	scounter_t		tickrate;	/* Real-Time ticks by second		*/
	scounter_t		harmonic;	/* tickrate = MRT_sv.harmonic * HZ	*/
```

```
      scounter_t          refresh;      /* ticks to refresh idlerfsh         */
      int                 MINIX_soft;   /* MINIX CLOCK Software interrupt    */
      priority_t          prtylvl;      /* Current syetem priority level     */
      lcounter_t          idlecount;    /* idle loop counter                 */
      scounter_t          idlerfsh;     /* idle refresh loop counter         */
      MRT_sysstat_ t      counter;      /* system statistics                 */
      } MRT_sysval_t;
```

D.2.2. Interrupt-related Data Structures

Listing D.7: Interrupt-related Data Structures.

```
struct MRT_irqd_s {
      /*-------------------- STATISTICAL FIELDS   ----------------------------*/
      lcounter_t          count; /* Interrupt counter                        */
      scounter_t          maxshower;   /* Maximun shower value               */
      lcounter_t          mdl;         /* Missed DeadLines                   */
      lcounter_t          timestamp;   /* Last Interrupt timestamp           */
      lcounter_t          maxlat;      /* Maximun Interrupt latency in timer hz */
      int                 reenter;     /* maximun reentrancy level           */
      /*-------------------- INTERNAL USE  FIELDS   --------------------------*/
      int                 irq;         /* IRQ number                         */
      scounter_t          harmonic;    /* Harmonic when the period was set   */
      MRT_vtimer_t        *pvt;        /* VT assigned for Timer Driven IRQs  */
      int                 flags;       /* MRT_ENQUEUED, MRT_TRIGGERED        */
      lcounter_t          latency;     /* Interrupt latency in timer hz      */
      scounter_t          shower;      /* shower counter since last period   */
      scounter_t          before;      /* TIC counter in MRT_IRQ_dispatch    */
      struct MRT_irqd_s   *next;       /* next irq desc in the priority list */
      struct MRT_irqd_s   *prev;       /* prev irq desc in the priority list */
      /*-------------------- FUNCTIONAL FIELDS   ----------------------------*/
      irq_handler_t       nrthandler;  /* Non real time handler              */
      irq_handler_t       rthandler;   /* real time handler                  */
      lcounter_t          period;      /* For Timer Driven period in ticks   */
      proc_nbr_t          task;        /* Real Time task number              */
      proc_nbr_t          watchdog;    /* Interrupt watchdog process         */
      priority_t          priority;    /* RT or NRT priority                 */
      irq_type_t          irqtype;     /* MRT_RTIRQ | MRT_TDIRQ | MRT_SOFTIRQ */
      char                name[MAXPNAME];/* name of the RT driver            */
};
typedef struct MRT_irqd_s MRT_irqd_t;

typedef      struct {
      MRT_irqd_t          *first;      /* Queue head                         */
      MRT_irqd_t          *last;       /* Queue tail                         */
      int                 inQ;         /* nbr of irqs descriptors in queue   */
      int                 pending;     /* number of pend irq_d in queue      */
      } MRT_irqQ_t;

typedef      struct {
      /* DON'T MOVE bitmap FROM THE FIRST POSITION IN THE STRUCT !! */
      bitmap16_t          bitmap;      /* Priority IRQ bitmap                */
      MRT_irqQ_t          irqQ[MRT_NR_PRTY];/* IRQs Queues                   */
      } MRT_iQ_t;

typedef      struct {
      MRT_iQ_t            iQ;
      bitmap16_t          hard_use;    /* bitmap of in-use hard interrupts   */
      bitmap16_t          soft_use;    /* bitmap of in-use soft interrupts   */
      MRT_irqd_t          irqtab[NR_IRQ_VECTORS+NR_IRQ_SOFT];/*IRQ desc table */
      bitmap16_t          mask[NR_IRQ_VECTORS];/* PIC masks for each priority */
            } MRT_sysirq_t;
```

D.2.3. Process-related Data Structures

Listing D.8: **Process-related Data Structures.**

```
typedef struct {
      int               flags;      /* Real Time Flags                      */
      priority_t        priority;   /* EFECTIVE Real Time priority          */
      priority_t        baseprty;   /* BASE Real Time priority              */
      lcounter_t        period;     /* period in RT-ticks                   */
      scounter_t        limit;      /* maximun number of process scheds     */
      lcounter_t        deadline;   /* process deadline                     */
      pid_t             watchdog;   /* Watchdog process                     */
      } rtattrib_t;

typedef struct {
      lcounter_t        scheds;     /* number of RT schedules               */
      lcounter_t        mdl;        /* Missed DeadLines                     */
      lcounter_t        timestamp;  /* Last schedule timestamp in ticks     */
      lcounter_t        maxlat;     /* Maximun latency in timer Hz          */
      lcounter_t        minlax;     /* Minimun laxity in timer Hz           */
      lcounter_t        msgsent;    /* messages sent by the process         */
      lcounter_t        msgrcvd;    /* messages received by the process     */
      } rtstats_t;


struct proc {
  struct stackframe_s p_reg;     /* process' registers saved in stack frame    */

#if (CHIP == INTEL)
  reg_t p_ldt_sel;               /* selector in gdt giving ldt base and limit  */
  struct segdesc_s p_ldt[2];     /* local descriptors for code and data        */
                                 /* 2 is LDT_SIZE - avoid include protect.h    */
#endif /* (CHIP == INTEL) */

#if (CHIP == M68000)
  reg_t p_splow;                 /* lowest observed stack value                */
  int p_trap;                    /* trap type (only low byte)                  */
#if (SHADOWING == 0)
  char *p_crp;                   /* mmu table pointer (really struct _rpr *)   */
#else
  phys_clicks p_shadow;          /* set if shadowed process image              */
  int align;                     /* make the struct size a multiple of 4       */
#endif
  int p_nflips;                  /* statistics                                 */
  char p_physio;                 /* cannot be (un)shadowed now if set          */
#if defined(FPP)
  struct fsave p_fsave;          /* FPP state frame and registers              */
  int align2;                    /* make the struct size a multiple of 4       */
#endif
#endif /* (CHIP == M68000) */

  reg_t *p_stguard;              /* stack guard word                           */

  int p_nr;                      /* number of this process (for fast access)   */

  int p_int_blocked;             /* nonzero if int msg blocked by busy task    */
  int p_int_held;                /* nonzero if int msg held by busy syscall    */
  struct proc *p_nextheld;       /* next in chain of held-up int processes     */

  int p_flags;                   /* P_SLOT_FREE, SENDING, RECEIVING, etc.      */
  struct mem_map p_map[NR_SEGS]; /* memory map                                 */
  pid_t p_pid;                   /* process id passed in from MM               */

  clock_t user_time;             /* user time in ticks                         */
  clock_t sys_time;              /* sys time in ticks                          */
```

```
  clock_t child_utime;          /* cumulative user time of children          */
  clock_t child_stime;          /* cumulative sys time of children           */
  clock_t p_alarm;              /* time of next alarm in ticks, or 0         */

  struct proc *p_callerq;       /* head of list of procs wishing to send     */
  struct proc *p_sendlink;      /* link to next proc wishing to send         */
  message *p_messbuf;           /* pointer to message buffer                 */
  int p_getfrom;                /* from whom does process want to receive?   */
  int p_sendto;

  struct proc *p_nextready;     /* pointer to next ready process             */
  sigset_t p_pending;           /* bit map for pending signals               */
  unsigned p_pendcount;         /* count of pending and unfinished signals   */

  char p_name[16];              /* name of the process                       */

#ifdef MRT
  rtattrib_t        rt;         /* Real Time Attributes                      */
  rtstats_t         st;         /* Real Time Statistics                      */

  MRT_msgQ_t        *pmq;       /* Real Time Message Queue                   */
  MRT_vtimer_t      *pvt;       /* virtual timer ID for periodic process     */

  int               getfrom;    /* from whom does RT-process want to receive? */
  int               sendto;     /* to whom does RT-process want to send?     */

  mrt_msg_t         *pmsg;      /* pointer to message buffer                 */
  mrt_mhdr_t        *pmhdr;     /* pointer to message header buffer          */

  struct proc*pnextrdy;         /*  next ready process in the priority queue  */
  struct proc*pprevrdy;         /*  previous ready process in the priority queue
*/

#endif /* MRT */

};
typedef      struct proc MRT_proc_t;

typedef      struct {
      struct proc        *first;      /* Queue head                          */
      struct proc        *last;       /* Queue tail                          */
      int                inQ;         /* Current number of process in queue  */
      int                maxinQ;      /* Maximun number of process enqueued  */
      } MRT_procL_t;

typedef      struct {
      bitmap16_t         bitmap;      /* bitmap of priority queues           */
      MRT_procL_t        procL[MRT_NR_PRTY]; /* Array of priority queues     */
      } MRT_procQ_t;

typedef    struct         {
      MRT_procQ_t  rdyQ;
      }MRT_sysproc_t;
```

D.2.4. Message-related Data Structures

Listing D.9: Message-related Data Structures.

```
/* Real-Time Message Queue Entry */
struct MRT_mqe_s {
      mrt_msgd_t         msgd;
      int                index;
      MRT_vtimer_t       *pvt;        /* VT for timeouts                     */
      struct MRT_mqe_s   *next;
      struct MRT_mqe_s   *prev;
```

```
	};
typedef struct MRT_mqe_s MRT_mqe_t;

struct MRT_mpool_struct   {
	MRT_mqe_t          mqe[NR_MESSAGES];
	};
typedef struct MRT_mpool_struct MRT_mpool_t;

typedef      struct {
	bitmap16_t         bitmap;
	MRT_mqe_t          *first[MRT_NR_PRTY];/* pointer to the first msgd    */
	MRT_mqe_t          *last[MRT_NR_PRTY]; /* pointer to the last msgd     */
	} MRT_mQ_t;

/* Real-Time Message Queue */
struct MRT_msgq_s   {
	int                index;       /* message queue ID (for quick search) */
	int                size;        /* message queue size                  */
	int                flags;       /* message queue policy flags          */
	int                inQ;         /* # messages enqueued                 */
	int                maxinQ;      /* maximun # of messages enqueued      */
	int                owner;       /* msg queue owner                     */
	long               delivered;   /* total #  of msgs delivered          */
	long               enqueued;    /* total # of msgs enqueued            */
	MRT_vtimer_t       *pvt;        /* VT assigned for Timer Driven IRQs   */
	MRT_mQ_t           mQ;
};
typedef struct MRT_msgq_s MRT_msgQ_t;

typedef      struct {
	MRT_mpool_t  mpool;             /* message pool                        */
	MRT_msgQ_t   mfreeQ;            /* free message list                   */
	MRT_msgQ_t   msgQ[NR_MSGQ];     /* message queues                      */
	} MRT_sysmsg_t;
```

GLOSARY

Context	The minimum information that is needed in order to save a currently executing process so that it can be resumed.
Context switching	The process of saving and restoring sufficient information for a Real-Time process so that it can be resumed after being interrupted.
CPU	Central Processing Unit.
CPU utilization	A measure of the percentage of non Idle processing.
Critical region/section	Code that interacts with a serially reusable resource.
Deadline	A deadline is a point in time at which some operation must be completed.
Deadline Monotonic Scheduling	Scheduling policy in Real-Time systems that meets a periodic process's deadline that does not equal its period.
Deadlock	A catastrophic situation that can arise when processes are competing forthe same set of two or more serially reusable resources.
Default	The value or status that is assumed unless otherwise specified.
Device Driver	A program that translate Operating System mandated function calls into device specific calls

Direct Memory Access (DMA)	A scheme in which access to the computer's memory is afforded to other devices in the system without the intervention of the CPU.
Dynamic Real-time Scheduling Algorithm	Scheduling algorithm that uses deadlines to assign priorities to processes throughout execution
Earliest-Deadline-First (EDF)	Scheduling policy that gives a processor to the process with the closest deadline.
Embedded System	A computing machine contained in a device whose purposeis not to be a computer. For example, the computers in automobiles and household appliances are embedded computers. Embedded computers use embedded software, which integrates an operating system with specific drivers and application software. Their design often requires special software-hardware codesign methods for speed, low power, low cost, high testability, or other special requirements.
Event	Any occurrence that results in a change in the state of a system.
Exception	Error or other special condition that arises during program execution.
Exception handler	Code used to process exceptions.
First-In-First-Out (FIFO)	Nonpreemptive scheduling policy that dispatches processes according to their arrival time in the ready queue
Hard Real-Time Scheduling	Scheduling policy that ensures processes meet their deadlines.
Hard Real-Time System	A Real-Time system in which missing even one deadline results in system failure.
Host	A computer that is the one responsible for performing a certain computation or function.

Input and Output (I/O)	Input/output, or I/O, is the collection of interfaces that different functional units (sub-systems) of an information processing system use to communicate with each other, or the signals (information) sent through those interfaces.
Interprocess Communications (IPC)	IPC is a set of techniques for the exchange of data between two or more threads in one or more processes.
Interrupt	An input to a processor that signals the occurrence of an asynchronous event. The processor's response to an interrupt is to save the current machine state and execute a predefined subprogram. The subprogram restores the machine state on exit and the processor continues in the original program.
Interrupt Controller	A device that provides additional interrupt handling capability to a CPU.
Interrupt Handler	A predefined subprogram that is executed when an interrupt occurs. The handler can perform input or output, save data, update pointers, or notify other processes of the event. The handler must return to the interrupted program with the machine state unchanged.
Interrupt Latency	The delay between when an interrupt occurs and when theCPU begins reacting to it.
Interrupt Service Routine (ISR)	Piece of code that your processor executes when an external event, such as a timer, occurs.
Kernel	The lowest portion of the operating system that provides for task scheduling, dispatching, and interprocess communication.
Kernel Calls	A kernel call is the mechanism used by an application program to request service from the operating system kernel
Latency (process scheduling)	Time a Process spends in a system mode before it is serviced.

Laxity	Value determined by subtracting the sum of the current time and a process's remaining execution time from the process's deadline. This value decreases as a process nears its deadline.
Least Slack First	It is a scheduling algorithm. It assigns priority based on the slack time of a process.
Library	A set of precompiled routines that may be linked with a program at compile time or loaded at load time or dynamically at run time.
Link	The portion of the compilation process in which separate modules are placed together and cross-module references resolved.
Linker	A computer program that takes one or more object files, assembles them into blocks that are to fit into particular regions in memory, and resolves all external (and possibly internal) references to other segments of a program and to libraries of precompiled program units.
Make	Utility to maintain, update, and regenerate related programs and files. The commands to be executed are placed in a makefile.
Makefiles	Files that contain a collection of commands that allow groups of programs, object files, libraries, and so on, to interact. Makefiles are executed by your development system's make utility.
Message Queue	An interprocess communication facility consisting of a memory location and at least two basic operations send/receivethat can be performed on it.
Microkernel	A microkernel is a type of kernel which consists of defining a very simple abstraction over the hardware, with a set of primitives or system calls to implement minimal OS services such as thread management, address spaces and

	interprocess communication
Multitasking	The operation by which a microprocessor schedules the handling of multiple tasks or processes. In generated code, the number of tasks is equal to the number of sample times in your model.
Multitasking Operating System	An operating system that provides sufficient functionality to allow multiple programs to run on a single processor so that the illusion of simultaneity is created. Contrast with multiprocessing operating system.
Operating System	A set of programs that manages the operations of a computer.It oversees the interaction between the hardware and the software and providesa set of services to system users.
Policy	Policy is a describing or proscribing set of rules and actions that encompass an ideal goal
Predictability	A system whose timing behavior is always within an acceptable range
Preempt	A condition that occurs when a higher-priority process interrupts a lower priority priority.
Preemptive Priority System	A system that uses preemption schemes instead of round-robin or first-come, first-served scheduling.
Primitives	Primitives are functions provided by the Operating System/Kernel as basic building blocks
Priority	Measure of a process's or thread's importance used to determine the order and duration of execution.
Priority Inversion	A condition that occurs because a lower priority process executes when a higher priority process is ready to run.

Process	The context, consisting of allocated memory, open files, network connections,in which an operating system places a running program.
Process Control Block/Process Descriptor	An area of memory containing information about the context of an executing program. Although the process control block is primarily a software mechanism used by the operating system for the control of system resources, some computers use a fixed set of process control blocks as a mechanism to hold the context of an interrupted process.
Race Condition	A situation where multiple processes access and manipulates hared data with the outcome dependent on the relative timing of these processes.
Rate-Monotonic Scheduling	Is an optimal preemptive static-priority scheduling algorithm used in Real-Time operating systems
Ready State	In the process control block model, the state of those processes that are ready to run, but are not running.
Real-Time	Refers to systems whose correctness depends not only on outputs bu tthe timeliness of those outputs. Failure to meet one or more of the deadlines can result in system failure.
Real-Time Computing	Support for environments in which response time to an event must occur within a predetermined amount of time. Real-time systems may be categorized into hard, firm and, soft real time.
Real-Time System	Computer that processes real-world events as they happen, under the constraint of a real-time clock, and that can implement algorithms in dedicated hardware. Examples include mobile telephones, test and measurement devices, and avionic and automotive control systems.
Response Time	The time between the presentation of a set of inputs to a software system and the appearance of all the associated

outputs.

Round-Robin Scheduling	Scheduling policy that permits each ready process to execute for at most one quantum per round. After the last process in the queue has executed once, the scheduler begins a new round by scheduling the first process in the queue,
Schedualability Analysis	The compile-time prediction of execution-time performance.
Scheduler	The part of the kernel that determines which task will run.
Scheduling policy	The way that the scheduler uses to determine which task runs by following a scheduling algorithm
Semaphore	A special variable or object type used for protecting critical regions.
Server	A process used to manage multiple requests to a serially reusable resource.
Stack	A first-in, last-out data structure handle by some CPU instructions.
Synchronous	An operation or operations that synchronize processes.
System Calls	A system call is the mechanism used by an application program to request service from the operating system
System Implementation	A phase of the software development life cycle during which a software product is integrated into its operational environment.
Task	A special kind of process in MINIX that treats with devices
Throughput	A measure of the number of operation or transactions per second that can be processed.

Time-Sharing	Time-sharing refers to sharing a computing resource among many users by multitasking.
Timeslice	Amount of time that a process is allowed to run on a processor before the process is preempted.
Timing Constraint	Time period during which a process (or subset of a process's instructions) must complete
Trap	Internal interrupt caused by the execution of a certain instruction.
User Space	Memory not required by the operating system.
Watchdog	A process that is scheduled or signaled when something in the system behavior is wrong.