# MINIX4RT: Real-Time Interprocess Communications Facilities

**Pablo A. Pessolani**
Departamento de Sistemas de Información
Facultad Regional Santa Fe - Universidad Tecnológica Nacional
ppessolani@hotmail.com

**Abstract**

MINIX4RT is an extension of the well-known MINIX Operating System that adds Hard Real-Time services in a new microkernel but keeping backward compatibility with standard MINIX versions.
Interprocess Communications provides a mechanism to make Operating System extensible, but they must be Priority Inversion free for Real-Time applications. As MINIX Interprocess Communications primitives does not have this functionality, new primitives were added to the Real-Time microkernel. This article describes the Real-Time Interprocess Communications facilities available on MINIX4RT, its design, implementation, performance tests and their results.

**Keywords**: Real-Time, Interprocess Communications, Priority Inheritance, Priority Inversion.

**Resumen**

MINIX4RT es una extensión del conocido Sistema Operativo MINIX que incorpora servicios de Tiempo Real Estricto en un nuevo microkernel pero manteniendo compatibilidad con las versiones anteriores del MINIX estándar.
La Comunicación entre Procesos es un mecanismo que permite hacer extensible a un Sistema Operativo, pero debe estar libre de Inversión de Prioridades para ser utilizado en aplicaciones de Tiempo Real. Como las primitivas de MINIX no disponen de esta funcionalidad, se incorporaron nuevas primitivas de Comunicación entre Procesos al microkernel de Tiempo Real. El presente artículo describe las facilidades de Comunicaciones entre Procesos en Tiempo Real disponibles en MINIX4RT, su diseño, implementación, tests de desempeño y sus resultados.

**Keywords**: Tiempo Real, Comunicaciones Entre Procesos, Herencia de Prioridades, Inversión de Prioridades.

# 1 INTRODUCTION

MINIX4RT (previously named RT-MINIXv2) [1, 2] is a Real-Time (RT) version of the well known MINIX [3] Operating System (OS) designed to teach concepts on RT-programming, in particular, those related to RT-kernels; but it can be usable as a serious system on resource-limited computers. It can be used to experiment with novel OS policies and mechanisms, and evaluate the impact of architectural innovations.

The key difference between time sharing OS and RTOS is the need for deterministic timing behavior in the RTOS. Deterministic timing means those 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 microkernel-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 [4]. Therefore, IPC primitives of a RTOS need to have deterministic execution and blocking times.

The design constrains for MINIX4RT were:

- *Compatibility with MINIX*: All processes that run on MINIX must run on MINIX4RT without modifications and sensible performance impact.
- *Minimal MINIX source code changes*: As MINIX is often used in OS design courses, students have deep knowledge of its source code. Reducing the source code changes keep the student's experience to learn a MINIX based RTOS. Most new code must be added in separated functions with few changes in the original MINIX code. This also helps for easier system updates when new versions of MINIX are released.
- *Source Code readability*: As MINIX4RT is focused for academic uses, its source code must be easily understood, perhaps sacrificing performance.

As a microkernel based OS, MINIX4RT has a loosely layered structure with client-server communication between the layers using Message Passing. Two methods can be used to message transferences:

- *With buffering*: The message is sent to a data structure like a RT-Mach port [5] or a Message Queue that stores it until a process receives it.
- *Without buffering*: The message is sent to a process and the sender must wait to transfer the message until the message is received. This method is known as rendezvous.

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

- When a sender calls the *send()* primitive but the receiver is not waiting that message, the sender is blocked until the receiver calls the *receive()* primitive for that sender.
- 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 average queue length would be less than one [5] 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 Input/Output (I/O) device rises 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 too much for the implementation of communications protocols where messages can flow in bottom-up manner triggered by interrupts.

In a RT-environment, several messages can be sent to a queue waiting to be received. MINIX4RT IPC uses unidirectional communication channels called Message Queues (MQ) consisting of priority lists that holds messages in RT-kernel space. The messages have fixed sizes and strict copy to value semantics. MQs handle messages in priority order and guarantee message delivery in a timely fashion, as Kitayama, Nakajima, and Tokuda [5] propose for Real-Time Mach ports. The number of messages that a MQ can store must be specified for each RT-process at creation time.

The rest of this work is organized as follows. Section 2 introduces on MINIX4RT. Section 3 presents background information about MINIX IPC primitives. Section 4 presents the proposed RT-IPC model. Section 5 and Section 6 are about basic data structures used for RT-IPC named Message Queue Entries and Message Queues. Section 7 describes RT-IPC primitives. Performance evaluation and related works are detailed in Section 8. Finally, Section 9 presents conclusions and future works.

## 2 OVERVIEW OF MINIX4RT

MINIX4RT provides the capability of running Real-Time and Non Real-Time (NRT) processes on the same machine. The RT-processes are executed when necessary no matter what MINIX is doing.

The RT-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 RT-microkernel been executed only when there are not any RT-processes to run. When MINIX tells the hardware to disable interrupts, the RT-microkernel intercepts the request, records it, and returns to MINIX. If one of these “disabled” interrupts occurs, the RT-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.

The major features of MINIX4RT are summarized as follows:

***Layered Architecture***: MINIX4RT has a layered architecture that helps to change a component without affecting the others.

***Real-Time Sub-kernel***: A RT-microkernel 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 RT-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 RT-microkernel schedules it on period expiration.

***Timer Resolution Management Detached from MINIX Timer***: A Timer interrupt of 50 Hz is emulated for the MINIX kernel even though 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***: There are system facilities named Virtual Timers (VT) used for time-related purposes as alarms, timeouts, 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.

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. Therefore a RT-process is managed by the RT-kernel and blocked for the MINIX kernel and, a NRT-process is managed by the MINIX kernel and blocked for the RT-kernel. Before converting a process, several parameters (as priority, period, watchdog, etc.) must be passed to the RT-kernel using the *mrt_setpattr()* System Call.

# 3 OVERVIEW OF MINIX IPC

Message transfer is the basic mechanism that MINIX uses to communicate Tasks, Servers and Users' processes and to notify hardware interrupt occurrence.

The IPC primitives implemented as kernel functions are [3]:

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

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

# 4 MINIX4RT IPC MODEL

Sometimes there are programming needs to control the system behavior on message transfers. A different policy can be applied to 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, service replies, signal interrupts, etc. does not help for the application of policies for each kind of action.

A RT-process managed by the RT-kernel can not use MINIX IPC primitives because:

- *mini_send()* and *mini_rec()* kernel functions could change the RT-process to a ready state for the MINIX kernel, therefore would be selected to execute by the NRT-scheduler ignoring all its RT-execution attributes.

- As MINIX IPC does not support different behaviors for *mini_send()*, any priority inversion avoidance protocol can be applied on it.

- If a RT-process make a request to a NRT-process using *mini_send()*, the RT-process must wait for the reply until NRT-process will run at NRT-priority. This is a case of Unbounded Priority Inversion.

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

MINIX4RT offers a variety of new Kernel Calls that let apply different policies depending on the kind of message transfer. The term Kernel Calls is used to distinguish among OS RT-services against System Calls that provides standard services. RT-IPC Kernel Calls have the following features:

- Synchronous/Asynchronous message transfers using Message Queues.
- Configurable Message Queue size.
- Different behavior for requests, replies, signals and interrupt notifications.
- Synchronous primitives timeout support.
- Configurable dequeueing policy (Priority order or FIFO order).
- Basic Priority Inheritance Protocol [6] support to avoid Unbounded Priority Inversion.
- Sending timestamps and message IDs can be retrieved by the receiver. Sender's attributes are stored into the message header (priority, process type, process deadline, etc).

To eliminate the buffer allocation delay, the RT-kernel reserves a memory space (called the System Message Pool) where messages are stored.

A MQ of a specified size is allocated for the process when it is converted into a RT-process. A MQ has a bounded capacity that quantifies its ability to store messages. A field named *pmq* into the process descriptor is filled with a pointer to the MQ allocated for the process; therefore a RT-process can only have one MQ.

# 5 THE MESSAGE QUEUE ENTRY DESCRIPTOR

The message data structure defined in MINIX (see Section 5.1.1) does not include the fields necessary for Real-Time message operations. MINIX4RT defines a new data structure named Message Queue Entry Descriptor (MQE) that includes useful fields to conform MQs (see Figure 1). A MQE descriptor have the following structure and type definitions:

```
struct MRT_mqe_s {
        mrt_msgd_t      msgd;   /* A Message Descriptor                          */
        int             index;  /* A Message Queue Entry ID                      */
        MRT_vtimer_t    *pvt;   /* Virtual Timer that handle the message timeout*/
        link_t          link;   /* link pointer to be used in MQ and Free Lists  */
        };
typedef struct MRT_mqe_s MRT_mqe_t;
```

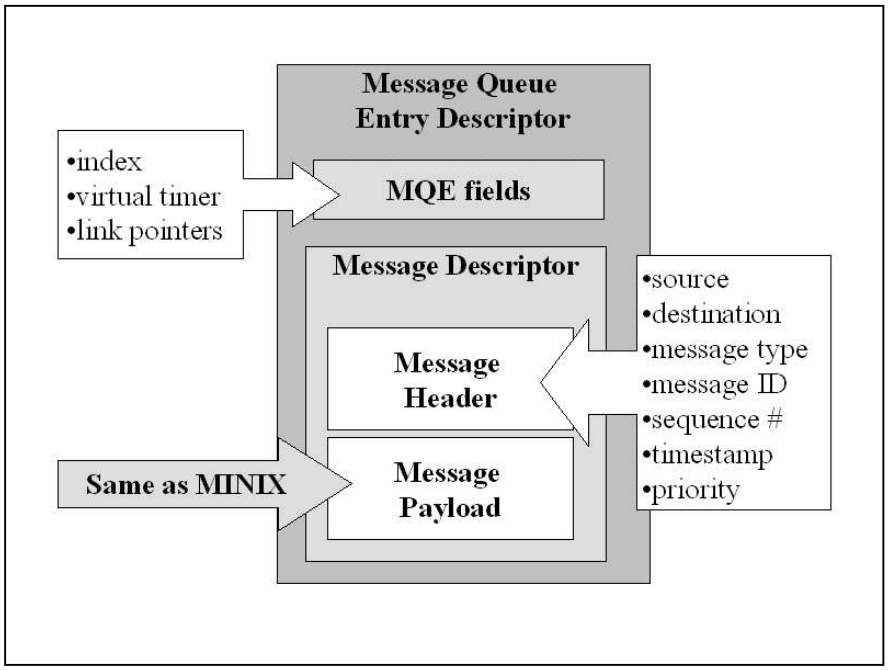


Figure 1: Message Queue Entry Descriptor.

The *index* field identifies the MQE. The *pvt* field is pointer to a Virtual Timer used to handle message timeouts. Linked lists management functions use the *link* field to insert/remove the MQE into/from a MQ or the System Message Pool implemented as a Free list.

## 5.1 Message Descriptor Data Structure

MINIX4RT defines a Message Descriptor Data Structure that is composed by a Message Payload Data Structure and a Message Header Data Structure described in the following sections.

### *5.1.1 Message Payload Data Structure*

MINIX has six messages payload types defined as:

```
typedef struct {int m1i1, m1i2, m1i3; char *m1p1, *m1p2, *m1p3;} mess_1;
typedef struct {int m2i1, m2i2, m2i3; long m2l1, m2l2; char *m2p1;} mess_2;
typedef struct {int m3i1, m3i2; char *m3p1; char m3ca1[M3_STRING];} mess_3;
typedef struct {long m4l1, m4l2, m4l3, m4l4, m4l5;} mess_4;
typedef struct {char m5c1, m5c2; int m5i1, m5i2; long m5l1, m5l2, m5l3;}mess_5;
typedef struct {int m6i1, m6i2, m6i3; long m6l1; sighandler_t m6f1;} mess_6;
```

MINIX4RT defines it Message Payload Data Structure equals to the MINIX message payload (see Figure 1).

The sizes of message elements will vary, depending upon the architecture of the machine.

### *5.1.2 Message Header Data Structure*

The RT-kernel needs more information to describe message attributes as the priority, the sender's deadline and the message type, but other fields are useful for RT-applications. The Message Header Data Structure includes those fields and is defined as follows:

```
struct mrt_mhdr_s{
                        mrtpid_t        src;        /* source process                */
                        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             */
                        };
typedef struct mrt_mhdr_s mrt_mhdr_t;
```

The *src* and *dst* fields reference to the message source and destination processes. The *mtype* is a code of what kind of message is it. The RT-kernel sets the *mid* field with the current number of messages sent by it. The *seqno* is the sequence number of the message sent by the source process. The *tstamp* field is the message timestamp in Timer ticks. The *priority* field is the sender's priority, and the *deadline* field is the sender's deadline.

# 6 THE MESSAGE QUEUES

A Message Queue Descriptor let the RT-kernel organize MQEs. A MQ descriptor is defined as follows:

```
struct MRT_msgQ_s    {
                int             index;      /* message queue ID                  */
                int             size;       /* message queue size                */
                unsigned int    flags;      /* message queue policy flags        */
                int             owner;      /* msg queue owner                   */
                long            delivered;  /* total #  of msgs delivered        */
                long            enqueued;   /* total # of msgs enqueued          */
                int             inQ;        /* current messages enqueued         */
                int             maxinQ;     /* message queue size                */
                link_t          link;       /* links to other MQs                */
                plist_t         plist;      /* Priority lists of MQEs including bitmap*/
};
typedef struct MRT_msgQ_s MRT_msgQ_t;
```

The *index* field identifies the MQ. The *size* is the number of MQEs that the MQ can store. The *flags* field specifies the message queue policy and status flags as priority or FIFO order. The *owner* field is the MQ owner process. The *delivered* and *enqueued* fields are statistical fields. The *inQ* field counts the number of MQEs enqueued, and *maxinQ* counts the maximum number of message enqueued. Linked lists management functions use the *link* field to insert/remove the MQs into/from process descriptor and MQ Free list. The *plist* field is a Priority List explained in Section 6.1.

## 6.1 Message Queue Management

To manage the MQs the RT-kernel uses several data structures (see Figure 2):

- *A Priority List data structure*: It is composed by a set of MQE lists (one list assigned for each priority level)

- *A bitmap with one bit assigned for each priority*: A bit set means that the related list has at least one message. Initially, all the bits are cleared indicating that all lists are empty.

- *An attribute flag*: Used to control the behavior of queue management algorithms according to an established policy.

On MQE enqueuing operations, the *priority*-th bit in the bitmap is set and the MQE is appended to the Priority List 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 the 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 Priority List.

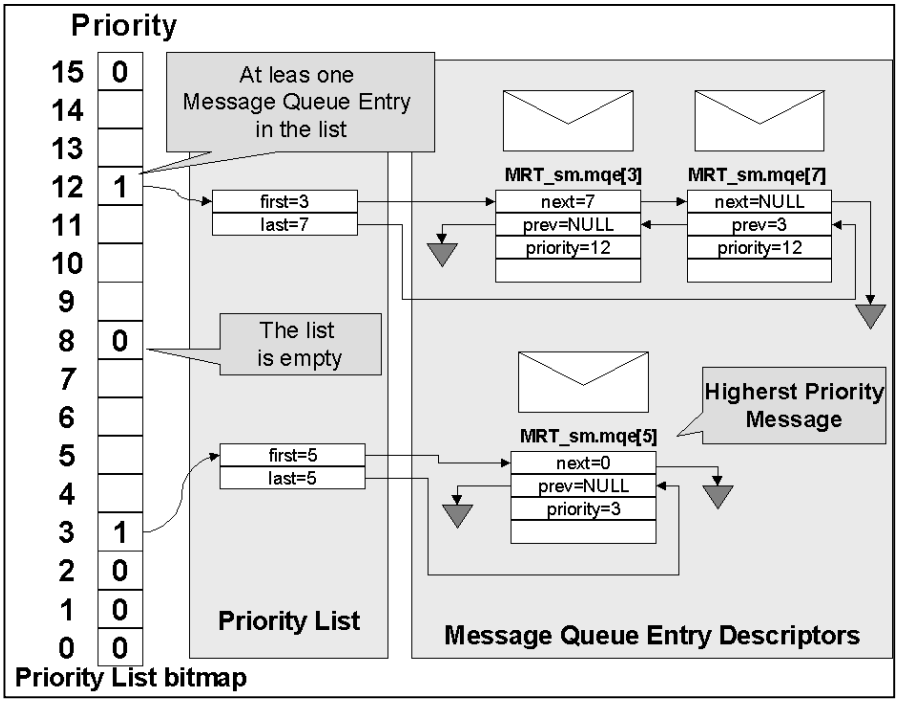


Figure 2: Message Queue Management Data Structures.

# 7 REAL-TIME IPC KERNEL CALLS

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 the same time (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.

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.

Sha, Rajkumar and Lehosky [7] suggest two protocols to avoid the priority inversion problem. One is the Basic Priority Inheritance Protocol (BPIP); the other is the Priority Ceiling Protocol (PCP).

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.

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. MINIX4RT provides RT-IPC primitives that are compliance with the BPIP offering a deterministic timing behavior.

### 7.1 *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.

If the destination process is waiting for the message, it is copied from the caller's message buffer to the destination's process message buffer. The destination process inherits the caller's priority if it is higher than its owns.

If destination process is not waiting for the message, the request is enqueued in the MQ owned by the destination process, and the caller is blocked until the message is received. The destination process and all other processes requested directly and indirectly by the destination 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 will be sent. A special value of *MRT_NOWAIT* can be specified to return without waiting if the destination process is not blocked receiving the message. To wait until the destination process will receive the message, *MRT_FOREVER* must be specified as *timeout*. If the timeout expired the message is removed from the destination's MQ, the caller process is unblocked returning and *E_MRT_TIMEOUT* error code and the priority of the destination process is set to the highest priority message into its MQ or its base priority specified in the *MRT_setpattr()* System Call..

### 7.2 *mrt_arqst()* Kernel Call

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

If the destination process is waiting for the message, it is copied from the caller's message buffer to the destination's process message buffer. The destination process inherits the caller's priority if it is higher than its owns. If destination process is not waiting for the message, the request is enqueued into the destination's MQ, and the caller returns without waiting for the message will be received. The destination process and all other processes requested directly and indirectly by the destination process inherit the caller's priority if it is higher than they own.

### 7.3 *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 bottom-up way (i.e. Server to Client).

If the destination process is blocked waiting for the reply, the message is copied from the caller's memory the destination's message buffer and destination process is unblocked. If the destination process is not blocked waiting for the message, the reply is enqueued in the destination's MQ.

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

### 7.4 *mrt_uprqst()* Kernel Call

The *mrt_uprqst()* Kernel Call sends a message to a process through a MQ in an asynchronous manner. Proxy like tasks can use it to make requests coming from remote processes in a bottom-up way. Its behavior is like *mrt_arqst()* but the priority is passed as a function parameter.

### 7.5 *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 bottom-up way. It behavior is like *mrt_arqst()* but the destination process priority remains unchanged.

### 7.6 *mrt_rcv()* kernel Call

The *mrt_rcv()* Kernel Call is used to receive a message. The caller searches for a message from the specified source into its MQ with the retrieving policy of the MQ (Priority or FIFO order). If there are no message from that source, 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 any message is received from the source in the specified period it returns *E_MRT_TIMEOUT* error code.

### 7.7 *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 two context switches for the caller process.

## 8 PERFORMANCE EVALUATION AND RELATED WORKS

This section describes IPC tests performed on MINIX4RT and their results. The IPC performance was tested on 20000 message transfers and four kinds of system setups/policies (see Table 1).

**Table 1: Setups and Policies of IPC Performance Tests**

| Test Name | With Timeout | Priority Queue/FIFO | Priority Inheritance |
|---|---|---|---|
| **NoVT/FIFO/NoInH** | No | FIFO | No BPIP |
| **VT/FIFO/NoInH** | Yes | FIFO | No BPIP |
| **VT/Prio/NoInH** | Yes | Priority Queue | No BPIP |
| **VT/Prio/InH** | Yes | Priority Queue | BPIP |

The tests were performed under different kinds of loads on the tested system (see Figure 3):

1. *Without Load (NOLoad):* All unneeded processes are killed before the test.
2. *CPU Load(CPULoad)*: A script loads the CPU without any I/O operation.
3. *I/O Disk Load(HDLoad)*: A process access files on the hard disk.
4. *I/O RS232e Load (RSLoad)*: A file transfer over the serial port at 19200 Kbps.

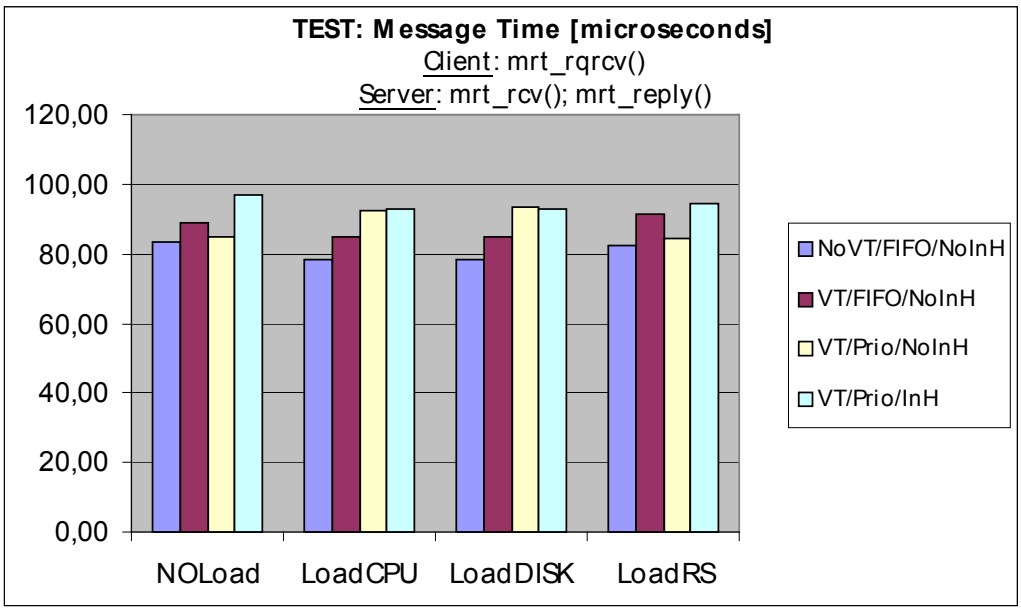


Figure 3: Message Transfer Time.

Table 2 presents Message Transfer Time values.

**Table 2: Message Transfer Time [microseconds]**

| | NoVT/FIFO/NoInH | VT/FIFO/NoInH | VT/Prio/NoInH | VT/Prio/InH |
|---|---|---|---|---|
| **NOLoad** | 83,10 | 88,85 | 84,70 | 97,00 |
| **LoadCPU** | 78,40 | 84,80 | 92,40 | 92,75 |
| **LoadDISK** | 78,40 | 84,75 | 93,20 | 92,65 |
| **LoadRS** | 82,40 | 91,25 | 84,45 | 94,45 |

The Programmable Interval Timer was set up at 1000[Hz]. This fact implies the execution of the Timer Interrupt Service Routing 1000 times by second adding an important overhead to the measurements but presents a more realistic scenario. Other tests performed on MINIX4RT showed an average Timer Interrupt Service Time of 32[μs].

The equipment used for the tests was an IBM Model 370C Notebook, Intel® DX4 75 MHz, AT Bus, Memory 8 MB, and MINIX4RT (Kernel 12052006). In spite of the equipment is quite old , it allows performance comparisons against reports of other systems [5, 8] with similar hardware. MINIX4RT presents better results than RT-MACH [5], but remarkable differences exist against QNX [8]. QNX provides priority based message queueing, but it only supports synchronous communications mode.

As Sacha's [8] reports, message transfer time on QNX consumes about 22 [microseconds] that is three times lower than on MINIX4RT and RT-MACH, but those values rise up to 205 [microseconds] when pipes are used.

## 9 CONCLUSIONS AND FUTURE WORKS

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. MINIX4RT combines Hard Real-Time with the standard MINIX platform so time sensitive control algorithms can operate together with background processing without worrying about interference.

Latest development version of MINIX4RT includes two new facilities for specific purposes with lower overhead than message transfer primitives:

- *RT-semaphores:* Used for mutual exclusion and synchronization of RT-processes
- *RT-FIFOs:* Used for variable-size data transfers among RT-Interrupt Service Routines and RT-Tasks.

MINIX4RT algorithms were developed to minimize priority inversion to meet applications with strict timing constrains. A sample of that is the use of Priority Lists on Message Queues and the implementation of the Basic Priority Inheritance Protocol.

The RT-microkernel has basic features as Interrupt Management, Process Management, Time Management, RT-IPC and Statistics gathering making it a good choice to conduct coding experiences in Real-Time Operating Systems courses.

## REFERENCES


[1] Pessolani, Pablo A, "*RT-MINIXv2: Architecture and Interrupt Handling",* 5th Argentine Symposium on Computing Technology, 2004.

[2] Pessolani, Pablo A., "*RT-MINIXv2: Real-Time Process Management and Scheduling*", 6th Argentine Symposium on Computing Technology, 2005.

[3] Tanenbaum Andrew S., Woodhull Albert S., "*Sistemas Operativos: Diseño e Implementación*" 2da Edición, ISBN 9701701658, Editorial Prentice-Hall , 1999.

[4] Jochen Liedtke, "*Improving IPC by Kernel Design*", 14th ACM Symposium on Operating System Principles (SOSP) 5th-8th December, Asheville, North Carolina, 1993.

[5] 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, 1993.

[6] 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. Software Engineering Institute Carnegie Mellon University, 1989.

[7] 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.

[8] Krzysztof M. Sacha, "*Measuring the Real-Time Operating System Performance*", Institute of Control and Computation Engineering, Warsaw University of Technology, Poland, 1995.