

Building Ethernet Drivers on RTLinux-GPL¹

Sergio Pérez, Joan Vila, Ismael Ripoll

Department of Computer Engineering

Universitat Politècnica de Valencia

Camino de Vera, s/n. 46022 Valencia, SPAIN

{serpeal,jvila,iripoll}@disca.upv.es

Abstract

This paper describes how to port Linux Ethernet drivers to RTLinux-GPL (hereafter RTLinux). It presents an architecture that will let Linux and RTLinux to share the same driver while accessing an Ethernet card. This architecture will let real-time applications to access the network as a character device (i.e. using the standard `open()`, `read()`, `close()`, `write()` and `ioctl()` calls) and also would let Linux to use the driver as it always did (`ifconfig`, `route`, etc.). This paper also discusses some scheduling policies that will give to RTLinux the highest priority while accessing the Ethernet card.

1 Introduction

Nowadays, communication aspects in real-time systems are a very important issue. Achieving interoperability and web-integration is becoming more and more a need in the actual real-time systems. CAN, PROFIBUS and other networks have been the solution for the industrial and automation world during years, but now, that is changing. In its 30-year anniversary, Ethernet is the world's most dominant networking technology (around the 85% of the world's LANs today are Ethernet-based). It is a reliable, fast and cheap medium (when writing this lines an Ethernet card of 100 Mbps may cost less than 20 EURO). All of these sound features made the industry to take Ethernet into account, so now Ethernet is more and more an adopted solution in the industrial and automation world.

Ethernet itself is not real-time at all. The problem is that the time spent by Ethernet - also known as CSMA/CD (Carrier Sense Multiple Access with Collision Detection or IEEE 802.3 - to access the medium is not deterministic. Despite of that, there are other protocols used in the same technology that try to bound the time spent trying to get access to the medium and there are cards that implement those protocols, so achieving real-time with Ethernet is possible.

Obviously, if the protocol layers upside Ethernet don't implement a real-time transport protocol, com-

munications won't be real-time. There are many real-time protocols based on Ethernet, such as RT-EP [1] (developed in the University of Cantabria), and others that are not real-time such as the well-known TCP. But the focus of this paper is Ethernet itself, and particularly, the paper is focused in writing device drivers for Ethernet cards using the RTLinux platform.

This paper attempts to establish the steps for developing RTLinux drivers for Ethernet cards. The paper presents an architecture that let real-time applications to send and receive packets through a RTLinux driver that is shared with Linux, which means that Linux can use the same device driver transparently. The paper also discusses how to give more priority to RTLinux against Linux when sending and receiving packets.

The reminder of the paper is organized as follows: section 2 gives a quick overview of the RTLinux architecture and those critical aspects of RTLinux needed to build RTLinux Ethernet drivers; section 3 describes how to move Linux Ethernet drivers to RTLinux; section 4 discusses how to prioritise the access to the Ethernet card when Linux and RTLinux are sharing it; section 5 gives some conclusions and describes future work and finally section 6 shows the bibliography and some references.

¹This work has been supported by the Spanish Government Research Office (CICYT) under grant TIC2002-04123-C03-03.

2 The RTLinux Architecture

RTLinux is a small, deterministic, real-time operating system developed by Victor Yodaiken and Michael Barabanov in 1996 [2]. RTLinux does not modify the Linux kernel or provides additional system calls in order to gain access to real-time features. Instead of trying to make the Linux kernel predictable, it builds a small microkernel or software layer, called RTLinux, directly over the bare hardware. Linux runs on this layer. RT-Tasks (RTLinux tasks) are executed by the RTLinux's executive. Linux is executed as the lowest priority task of RTLinux.

An important concept that is going to appear through the paper is the concept of context. There are two main execution contexts, the RTLinux context and the Linux context. Although both codes are mapped in the same address space and linked together, each one is executed independently; therefore trying to access data located in one context from the other may cause race conditions or even lock the system.

While writing code, the programmer has to keep in mind the context where the code will be executed. And only use the facilities designed the that context. RTLinux takes full control over interrupts, virtualizing the Linux's interrupt handling system. It allows either capturing and handling interrupts or just bypassing them to Linux. All the interrupts not handled by RTLinux are dispatched to Linux. The RTLinux architecture is shown in the Figure 1.

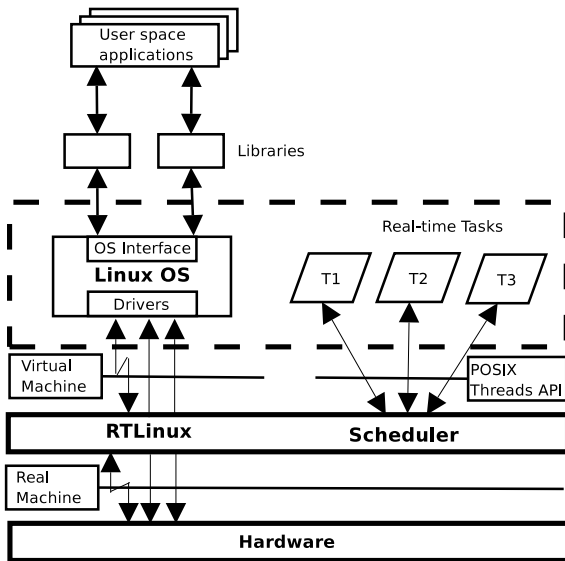


FIGURE 1: *The RTLinux architecture*

Although RTLinux tries to follow the POSIX standard, POSIX don't provide an API to manage interrupts so RTLinux implements its own API. RTLinux

allows to enable and disable interrupts, mask interrupts, and handle hardware and software interrupts (interrupts generated by RTLinux that are dispatched to the Linux kernel). The interrupt management global scheme including some API calls is shown in the Figure 2.

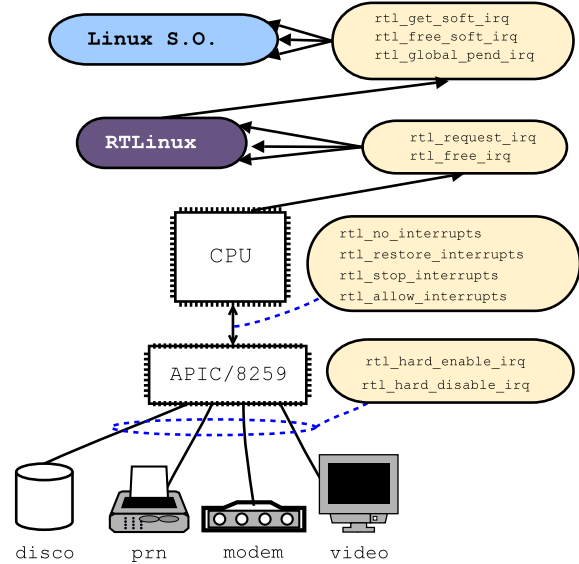


FIGURE 2: *Interrupt management global scheme*

The remainder of this section is devoted to review the required facilities of RTLinux used to write device drivers. First, the RTLinux interrupt handling system is presented and finally, a particular dynamic memory manager is described.

2.1 Hardware Interrupts

The particular API for managing interrupts is not really a critical issue for this paper, but it will help to understand how RTLinux manages interrupts and will clarify some concepts needed along this paper (for a deeper knowledge of the RTLinux API read [3]). The next functions deal with hardware interrupts. They are executed in the RTLinux context. There can be only one interrupt handler for each interrupt.

```
int rtl_request_irq(unsigned int irq, unsigned
    int (*handler)(unsigned int irq,
    struct pt_regs *regs));
```

This function registers `handler` as the interrupt handler of the interrupt `irq` and unmask the interrupt.

```
int rtl_free_irq(unsigned int irq);
```

This function frees the interrupt handler of interrupt `irq`.

2.2 Masking Interrupts

Masking an interrupt means that the processor won't receive the interrupt until unmasked. When the interrupt is triggered, the hardware automatically masks that interrupt line, so the handler should unmask the interrupt by calling the function `rtl_hard_enable_irq()` (which is described next). Masking an interrupt when triggered prevents from receiving more interrupts while the handler is not ready to attend them.

```
int rtl_hard_enable_irq(unsigned int irq);
```

This function unmarks the interrupt `irq`. Unmasking the interrupt means that it will be received when the interrupts are enabled

```
int rtl_hard_disable_irq(unsigned int irq);
```

This function masks the interrupt `irq`. No interrupts of this kind will be received until the interrupt becomes unmasked.

2.3 Software Interrupts

As said before, all interrupts not handled by RTLinux are redirected to Linux. RTLinux also implements a fictitious interrupt handling system allowing RTLinux to trigger virtual interrupts that will be handled in the Linux context in a safe way (avoiding race conditions). This powerful mechanism is called software interrupts and it allows to execute functions in the Linux context. Software interrupts will be really useful to achieve transparency with Linux while sharing an Ethernet driver with RTLinux. Next, a brief description of the API provided by RTLinux to manage software interrupts:

```
int rtl_get_soft_irq(void (*handler)(int,
    void *, struct pt_regs *),
    const char * devname);
```

This function registers a Linux's interrupt handler. The handler will be executed in the Linux context. `devname` is a string of characters that will identify the interrupt. This name will appear in the system's interrupts list: `/proc/interrupts`. It is interesting to note that this function doesn't allow to pass the interrupt as a parameter, instead it returns the number of a free interrupt that will be assigned to the function `handler`.

```
void rtl_free_soft_irq(unsigned int irq);
```

This function deregisters the interrupt handler of the interrupt `irq` (`irq` should be a value returned from a previous call to `rtl_get_soft_irq()`).

```
void rtl_global_pend_irq(int irq);
```

This function throws an interrupt that will be dispatched to Linux when it becomes executed. This function implements the fictitious interrupt triggering method which allows to execute functions in the Linux context.

2.4 Dynamic Memory Allocation (TLSF)

Real-time programmers never liked dynamic memory allocation because of the unbounded times of its operations (`malloc()` and `free()`). In the initial versions of RTLinux, real-time tasks had to manage their own reserved memory.

Recently, a dynamic memory manager called TLSF (Two Level Segregated Fit) has been implemented in RTLinux (although it could be used in other platforms). TLSF guarantees absolutely bounded response times while performing the `malloc()` and `free()` functions, besides providing quite low response times. TLSF provides the ANSI/ISO standard 'C' `malloc()`, `free()` and `realloc()` functions.

3 Porting a Linux Ethernet Driver to RTLinux

This paper is not dedicated to those Ethernet cards belonging to a specific bus such as PCI or ISA. Moreover, since there is not a standard defining how Ethernet card drivers should be written, this paper won't try to define step by step those lines in the source code of the driver that must be moved to RTLinux. Instead of that, this paper will try to provide the guidelines to move Linux Ethernet drivers to RTLinux. Most of the changes needed to port a Linux Ethernet driver to RTLinux deal with adapting the driver to the particular architecture of RTLinux.

First of all, let's know how do the Linux's Ethernet drivers support work.

3.1 Linux's Ethernet Device Drivers Support

In Linux, network devices (as other devices) must be registered in order to make itself known to the system.

Ethernet drivers insert a interface's data structure in a global list of network drivers (for a deeper knowledge on Linux device drivers read [4]). This structure is called `net_device` and has many fields pointing to functions that will be used to manage the Ethernet card. The main functions provided by the `net_device` structure are described next:

```
int open(struct net_device *dev)
```

This function turns the card up which means to start sending and receiving packets. This function is used by Linux by means of the `ifconfig` command executed with the option '`up`'. `ifconfig` also configures the interface: IP address, network mask etc...

```
int stop(struct net_device *dev)
```

This function turns the card down, which means freeing all the allocated resources and stop receiving and sending packets. This function is used by Linux by means of the `ifconfig` command executed with the option '`down`'.

```
int hard_start_xmit(struct sk_buff *skb,
struct net_device *dev)
```

This function queues a packet (contained in the `sk_buff` structure) in the internal buffers of the Ethernet card. The card will send the packets of its internal buffers in the order they were inserted. This function is used by the Linux kernel to send packets through the interface. Besides the functions provided by the `net_device` structure, there are other functions which are bus dependent (PCI, ISA, etc...) that an Ethernet driver must provide. These are:

```
int init(...)
```

This function is used to initialise the driver, which basically means registering the driver and allocating resources. This function is called when the driver module is inserted into the kernel by means of the Linux command `insmod`.

```
int remove(...)
```

This function is used to remove the driver, which basically means unregistering the driver and freeing bus specific resources. This function is called when the driver module is removed by means of the Linux command `rmmmod`.

Linux packet reception scheme is performed asynchronously. When a packet is uploaded, the driver passes the packet to the Linux kernel by means of the function `netif_rx()` (more details about this function are described in the subsection 3.5).

So this is how the Linux's Ethernet drivers support works in short.

3.2 Problems Sharing the Same Device

There are some problems derived from sharing the Ethernet card by Linux and RTLinux. These are:

- how to distinguish the incoming packets' destiny (Linux or RTLinux);
- Linux and RTLinux could get the card closed by the other when trying to access the card;
- how to prioritise RTLinux when requesting to send against Linux;

The first problem is easily solved by assigning a different IP address to RTLinux, so the interrupt handler of the driver should be modified to correctly manage this.

The second problem is solved by means of flags controlling who is accessing what function in each moment and maybe prioritising RTLinux in that access (for example, never allowing Linux to close the card if RTLinux is still using it).

The third problem suggests many solutions that are explained in section 4.

3.3 Building a RTLinux POSIX device driver

In this subsection, the paper will stand how to build a RTLinux Ethernet driver (by mapping the functions described in the subsection 3.1 to the POSIX standard `open()`, `close()`, `read()` and `write()` calls) taking advantage of the RTLinux capability of developing POSIX device drivers.

The first problem found when porting a Linux driver to RTLinux is that RTLinux has no network support. Linux has a network subsystem that allows to register network device drivers in a particular way (as explained in subsection 3.1). Instead of implementing that network subsystem in RTLinux (which would mean modifying the RTLinux sources), what has been done is to take advantage of the RTLinux capability of developing POSIX device drivers. This allows to access the I/O through virtual files located in a virtual file system with only one directory: `/dev`. The RTLinux Ethernet driver would be one of those files located in `/dev` (usually `/dev/eth0`), file that would be accessed by means of the ANSI 'C' standard `open()`, `close()`, `write()`, `read()` and `ioctl()` calls. In order to register a POSIX device driver in RTLinux, the driver has to declare a structure called `rtl_file_operations` that will be used to access the functions implemented by the driver (`open()`, etc.) The steps to register new devices are:

1. Declare a `rtl_file_operations` structure, that will keep pointers to the `open()`, `read()`, `close()` and `write()` functions of the driver.
2. Implement the `open()`, `close()`, `write()`, `read()` and `ioctl()` functions of the `rtl_file_operations` structure. The `rtl_file_operations` structure has other block specific functions such as `mmap()`, `poll()` or `llseek()`, that doesn't need to be implemented in Ethernet device drivers.
3. Fill the `rtl_file_operations` structure with the functions implemented.
4. Finally, register the driver by calling function `rtl_register_rtldev()`, whose arguments are the mayor number assigned to the driver, the device name (for example, `/dev/eth0`) and a pointer to the structure `rtl_file_operations`. Figure 3 presents the scheme described above.

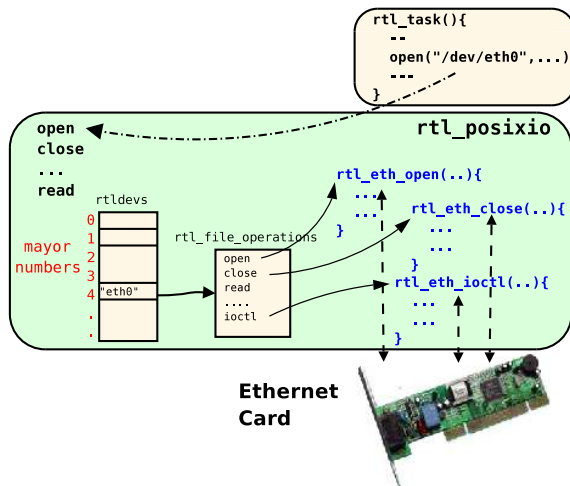


FIGURE 3: *Interrupt management global scheme*

Once the device driver has been registered, a RT-Task that wants to use the device driver has to open it (by means of the `open()` call) and then send and receive packets by means of the `write()` and `read()` calls, respectively. The `ioctl()` call would let the user to configure some parameters of the driver and also to ask the driver for some characteristics, such as the hardware address of the Ethernet card (also known as MAC address).

Now, let's see how to map the Linux calls `open()`, `stop()` and `hard_start_xmit()` into the RTLinux's ones (`open()`, `close()` and `write()`). Fortunately, except the `hard_start_xmit()` call, all the other functions are directly mapped to RTLinux functions (`open()` into `open()` and `stop()` into `close()`). The `hard_start_xmit()` function cannot be directly mapped into a `write()` function because the driver must prioritise the RTLinux write requests, so new code must be added in order to correctly buffer the requests. Anyway, the code of the original `hard_start_xmit()` will be used by both Linux and RTLinux to send through the card, so the original function could be moved to another one (for example `hard_write()`) that will be used in the new Linux `hard_start_xmit()` function and the RTLinux `write()` function.

Since the packet reception scheme is different in Linux than in RTLinux, two `read()` functions must be written, one to deliver packets to Linux and other for RTLinux. The subsection 3.5 describes how to implement the `read()` function in Linux. Implementing the `read()` function in RTLinux is much more easy: once the card has uploaded a packet to RAM, the driver has to copy the packet in the buffer provided in the `read()` call, taking into account that the `read()` call must be blocking (defined by POSIX). In RTLinux, a blocking read function can be implemented with semaphores or mutexes.

Despite of the implementation, the `read()` call has only one function: it returns a packet to the caller. One interesting issue is how to return that packet. A normal `read()` call would need the user to pass a pointer to a

previously reserved memory area and the driver would fill that area with data. That means that the driver would have to perform a copy for each packet. In the RTLinux implementation of the `read()` call, the driver could save the expense of that copy just by returning a pointer to the packet being hold in the internal buffers of the driver (the Linux implementation must perform a copy, as shown in the subsection 3.5); this technique is usually named zero copy. Using the zero copy technique forces to change the way of interfacing the `read()` call since the API it provides can't be modified and the way it is interfaced is different using the zero copy technique or not. The `read()` call API is the next one:

```
ssize_t read(int fd, void *buf, size_t count)
```

Using the zero copy technique, in order to get a pointer to the driver's internal buffer, an indirection level is needed. To achieve this, a structure encapsulating a memory pointer is needed. The next structure is an example of how that structure should be built:

```
struct memory{
    void *mem;
};
```

The next code shows how to interface the read call using the previous structure:

```
struct memory receive_buffer;
u16_t len;

// Obtain the size of the packet and put
// it into the "len" variable.
len=read(fd,(void *) &receive_buffer,1536);
```

The code of the read call should do the next:

```
((struct memory *)buf)->mem =
    internal_buffer_pointer;
```

Programming this way the driver's interface, after performing the `read()` call, `receive_buffer.mem` will contain a pointer to the driver's internal buffer. However, the driver could really work using both techniques. The RT-Task could dynamically change the mode of performing the `read()` call by means of the `ioctl()` function. This has to be taken into account when implementing the `ioctl()` function.

3.4 Handling Interrupts

To guarantee full control over the card the interrupt management of the driver must be moved to RTLinux. Basically this consists on replacing the Linux functions that manage interrupts (such as `request_irq()`, `free_irq()`, `disable_irq()` and `enable_irq()`) for those equivalent in RTLinux (explained in the subsections 2.1 and 2.2).

One problem related with handling interrupts is that, when executing the interrupt handler the card can continue triggering interrupts, and those interrupts will be unhandled (because the interrupt is masked inside the

handler) which may cause losing packets. This problem is solved by taking advantage of the native operation mode of most Ethernet cards (the 3Com905C-X driver's internal management is going to be used as an example). During the driver initialisation the driver must give to the card a pointer to a memory structure that the card will fill with the incoming packets. The structure is a circular list of UPDs (Upload Packet Descriptors) and the pointer passed to the card is called `UpListPtr` (Upload List Pointer) as shown in the Figure 4.

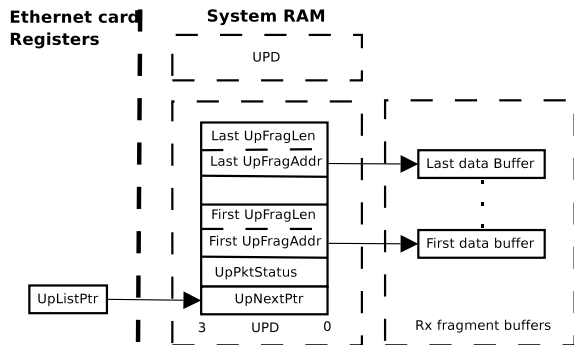


FIGURE 4: The UPD data structure

The UPD structure has a bit in the `UpPktStatus` field that is changed when the card uploads a packet, so being inside the interrupt handler the driver can check if new packets have been uploaded since the execution handler started, avoiding missing packets. The code of the interrupt handler would keep quite simple:

```
while(next_UPD->UpPktStatus & UPLOADED){
    receive_packet();
}
```

The card is writing packets constantly in the circular list, so it could overwrite a packet that has not still been read. In order to avoid this, the `receive_packet()` function must copy the incoming packets to another buffer, assuring that those packets won't be removed from the second level buffer until read. This second level buffer can be implemented in two ways:

1. one unique protected buffer or
2. two independent buffers: one for Linux and other for RTLinux.

The second implementation is simpler and faster than the first one, although it may imply using more resources. Both solutions avoid race conditions while accessing the buffer, but the second one also guarantees that RTLinux won't be interfered when reading, so this should be the adopted option.

3.5 The Linux Packet Reception Scheme in the RTLinux Driver

Subsection 3.1 stood that the way in which Ethernet drivers pass packets to Linux is by calling a function: `netif_rx`. This function cannot be executed in the

RTLinux context, so to pass packets to Linux in a safe way software interrupts are needed (read subsection 2.3). Triggering a software interrupt in the hardware interrupt's handler when an incoming packet is addressed to Linux is the perfect way of executing `netif_rx` in a safe way. This scheme is shown in the Figure 5.

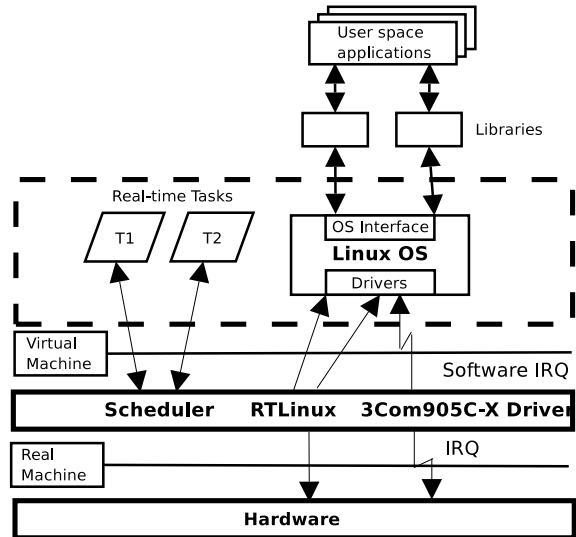


FIGURE 5: Interfacing the Driver from Linux

The Linux interrupt handler has to read incoming packets until the buffer is empty:

```
while(pkt_len=extract_frame_of_buffer(&buf)){
    linux_rx(&buf, pkt_len);
}
```

Where `linux_rx()` is a function that prepares the packet to be sent to the Linux kernel and sends it by means of `netif_rx()`. The `linux_rx()` code is shown next and should be the same for all the Ethernet drivers:

```
static int linux_rx(void *buf, size_t pkt_len){
    struct sk_buff *skb;

    if((skb=dev_alloc_skb(pkt_len+2)) != 0){

        memcpy(skb_put(skb,pkt_len),buf,pkt_len);
        skb->dev = net_device_structure;
        skb->protocol = eth_type_trans(skb,
            net_device_structure);
        skb->ip_summed = CHECKSUM_UNNECESSARY;

        netif_rx(skb);

        return 0;
    }

    return -1;
}
```

4 Sending Packets

Packet transmission works in a similar way than reception. The RTLlinux driver builds a linked list of DPD structures (Download Packet Descriptor) to be sent and a pointer to the head of the list is given to the Ethernet card.

Once the packet transmission has been explained, the paper will describe how to manage that list of DPDs taking into account that there are two different sources of packets (Linux and RTLlinux) and the driver must prioritise one against the other.

The first and more simple solution is to send the packets in the same order than requested by the application, with no distinction whether the packet comes from Linux or RTLlinux. This solution do not provide suitable real-time performance. Real-time performance can be achieved using two separate outgoing lists: a high priority list feed with RTLlinux packets and a low priority list with the Linux packets. Packets of the second list are only sent if the high priority list is empty.

There are several scheduling algorithms in the real-time communication theory that try to give a fair use of the network to the communicating processes. Several bandwidth reservation algorithms can be used: CBQ (Class-Based Queueing discipline), CSZ (Clark-Shenker-Zhang scheduler), etc ...

Current implementation is based on the two priority queues. This scheme has a worst case scenario, when the Ethernet card is beginning to send a Linux's packet and RTLlinux initiates a write call. Since aborting is not a normal operation in Ethernet cards, the high priority packet is delayed until the current packet is transmitted. The scheme described above would cause an interference to RTLlinux of only one Linux' packet with a maximum size of 1536 bytes.

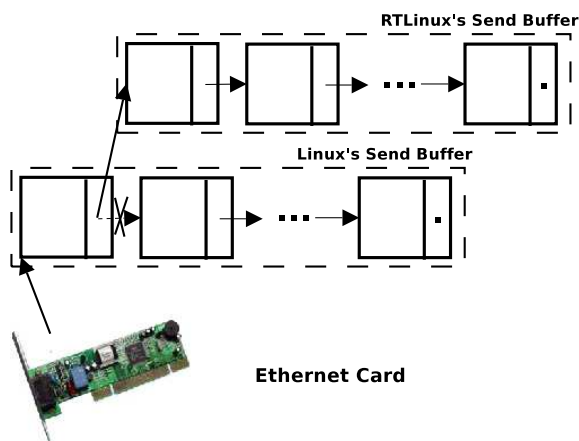


FIGURE 6: *Linux Interference*

5 Conclusions and Future Work

This paper presents some guidelines showing how to move Linux Ethernet drivers to RTLlinux. The paper has also presented an architecture that allows to share the same driver to access an Ethernet card from both Linux and RTLlinux. When writing this lines, two drivers have been written (3Com905C-X, Realtek8139) and there is other in progress (3Com509B). This drivers have been used in the project RTL-lwIP [5], which is a porting of the LwIP (Lightweight IP) stack to RTLlinux that allow RT-Tasks to send and receive packets through the network. The drivers can be found in the RTL-lwIP project home page [6].

6 Bibliography and References

References

- [1] Martínez J.M., González Harbour M., Gutiérrez J.J., July 2003, *RT-EP: Real-Time Ethernet Protocol for Analyzable Distributed Applications on a Minimum Real-Time POSIX Kernel*, PROCEEDINGS OF THE 2ND INTL WORKSHOP ON REAL-TIME LANS IN THE INTERNET AGE, PORTO, PORTUGAL.
- [2] Barabanov M., Yodaiken V., March 1996, *Real-Time Linux*, Linux Journal.
- [3] Ripoll I., April 2000, *RTLlinux API Tutorial*, <http://rtportal.upv.es/rtportal/tutorial/rtllinux-tutorial.pdf>
- [4] Rubini A., Corbet J., 2nd Edition June 2001, *Linux Device Drivers*, O'Reilly, 0-59600-008-1.
- [5] Pérez S., Vila J., September 2003, *Building Distributed Embedded Systems with RTLlinux-GPL*, PROCEEDINGS OF THE 9TH IEEE INTERNATIONAL CONFERENCE ON EMERGING TECHNOLOGIES AND FACTORY AUTOMATION, LISBON, PORTUGAL.
- [6] The RTL-lwIP project web site, <http://canals.disca.upv.es/serpeal/RTL-lwIP/htmlFiles/index.html>