Augmenting RT-Linux GPL Capabilities with TCP/IP *

Sergio Pérez, Joan Vila
Department of Computer Engineering
Universitat Politècnica de Valencia, Spain
{serpeal, jvila}@disca.upv.es

Abstract

This paper describes RTL-lwIP, which is a TCP/IP stack for embedded systems based on lwIP (Lightweight TCP/IP stack) that runs on RT-Linux and can be used by real-time tasks. RTL-lwIP allows real-time tasks to communicate directly with remote real-time tasks or even with Linux user processes. The importance of introducing TCP/IP on RT-Linux is that it enables the possibility of developing real-time distributed embedded systems based on CORBA, thus allowing interoperability with other platforms and web-integration.

This paper also gives some guidelines in order to implement RT-Linux drivers for Ethernet cards using as an example the implementation of a RT-Linux driver for the Ethernet card 3Com905C-X.

1. Introduction

Linux is becoming more and more an option for developing embedded systems, but it is not appropriate for real-time systems since Linux doesn't comply with the restrictions that that kind of systems demand. That Linux's lack got fulfilled with the public appearance of RT-Linux [1]. What RT-Linux does is to build a small microkernel or software layer, called RT-Linux, directly over the bare hardware, running Linux over this layer as the lowest priority task. RT-Linux allows creating tasks with hard real-time constraints at kernel level. Although real-time tasks are executed at kernel level, no invocations to any Linux kernel function can be performed because it could lead to deadlocks – due to violations of critical sections-. This is one of the main disadvantages of the RT-Linux architecture. If it were not so, real-time tasks could access all the TCP/IP stack functionalities (in Linux, the TCP/IP stack is embedded in the kernel) and what we would get would be a real-time distributed system.

The idea of achieving real-time distributed tasks using architectures like CORBA [2] (Common Object Request Broker Architecture) becomes really appealing for many reasons, but CORBA needs a communication protocol called IIOP (Internet Inter-ORB Protocol) that uses TCP/IP (with both UDP and TCP connections).

There's been a successful attempt to give to real-time tasks access to the Linux TCP/IP stack in a project called RTSOCK, developed by Robert Kavaler [3]. That approach has both advantages and disadvantages. The main advantage is that all of the standard layer 2 and layer 3 protocols already implemented in the kernel are available to the real-time task. One of the main disadvantages is that, since the Linux kernel is executed as the lowest priority tasks (that means that only when no real-time task need to be executed, the "Linux task" is executed), RTSOCK would cause a lot of unpredictable delays when receiving and sending through the Linux kernel. Other disadvantage is that TCP sockets are not supported (only UDP sockets).

This paper describes RTL-lwIP (RTL-lwIP is the porting of the lwIP TCP/IP stack [4] to RT-Linux; lwIP is described in subsection 3.1) which is a full TCP/IP stack suitable for embedded systems – due to its reduced code size, memory usage and processing improvements - that gives to real-time tasks the chance of communicating via TCP/IP directly with other real-time tasks or even with Linux user processes. This paper also present a guide to develop drivers for Ethernet cards taking advantage of the new RT-Linux capabilities (such as POSIX signals or dynamic memory managing) using as example the implementation of a driver for the Ethernet card 3Com905C-X.

Finally, source code of the RTL-lwIP project and related information can be found in the RTL-lwIP project home page [5].

The reminder of the paper is organized as follows: section 2 presents the RT-Linux architecture; section 3 describes the porting of the lwIP (Lightweight) TCP/IP stack to RT-Linux; section 4 describes how to write Ethernet card drivers showing the example of a driver for the Ethernet card 3Com905C-X; section 5 describes conclusions and future work and finally section 6 describes references and bibliography.

2. The RT-Linux GPL Architecture

RT-Linux is a small, deterministic, real-time operating system developed by Victor Yodaiken and Michael Barabanov in 1996 [6]. RT-Linux builds a small

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

microkernel or software layer, called RT-Linux, directly over the bare hardware. Linux runs over this layer as the lowest priority task, so it only runs when no other RT-tasks are running. The RT-Linux architecture is shown in the figure 1.

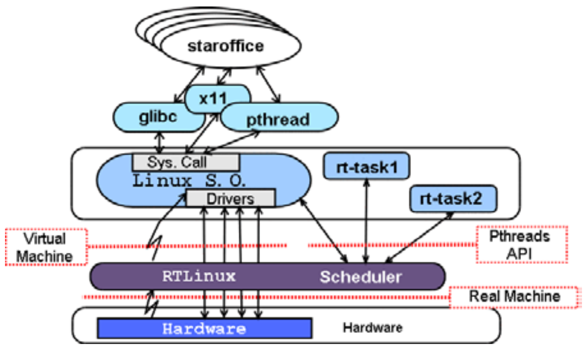

Figure 1. The RT-Linux architecture.

RT-Linux shares the same address space than the Linux kernel so, in theory, any Linux kernel function can be invoked from a real-time task, however this is not advisable and can lead to deadlocks (because can violate critical sections).

Developing real-time applications in RT-Linux usually requires splitting the application into two parts:

- **RT-tasks or RT-threads**: tasks with hard real-time constraints programmed and executed at kernel level.
- **Linux user processes**: tasks with soft or no real-time constraints. Unlike RT-tasks, the graphical system and some networking facilities are available for these tasks, so those parts of an application that need these resources have to be executed as Linux processes.

Both parts of an application can communicate using a special device called RTFIFOs. The execution of RT-tasks is done using the Linux facility to dynamically load new kernel modules. RT-tasks are dynamic kernel modules. Even RT-Linux is a set of kernel modules (scheduler, fifos…) that need to be dynamically loaded.

Yodaiken and Barabanov tried to give a POSIX Threads API to RT-Linux, but that API was not fully implemented. The OCERA IST project [7] (Open Components for Embedded Real-time Applications) is completing the POSIX Threads API (and some other things) of RT-Linux adding components to RT-Linux such as POSIX signals and POSIX timers, so we can say more and more that RT-Linux is becoming a POSIX-compliant operating system.

3. TCP/IP Over RT-Linux GPL

This section presents RTL-lwIP which is the porting of the lwIP TCP/IP stack [4] to RT-Linux.

RTL-lwIP includes IP, IPv6, ICMP, UDP and TCP protocols. It offers to RT-Tasks a socket API to communicate with other RT-Tasks or Linux processes over a network. RTL-lwIP also includes RT-Linux drivers for the Ethernet cards 3Com905C-X and Realtek8139 and a set of examples showing how to use RTL-lwIP.

3.1. The lwIP (Lightweight) TCP/IP Stack

lwIP is an implementation of the TCP/IP stack developed by Adam Dunkels [8]. In author words: "The focus of the lwIP stack is to reduce memory usage and code size, making lwIP suitable for use in small clients with very limited resources such as embedded systems".

Improvements achieved by lwIP in terms of processing speed and memory usage have been performed by means of violating the TCP/IP layers. As the barrier between the kernel and the application processes is not a strict protection, a more relaxed scheme for communication between the application and the lower layer protocols can be performed by means of shared memory. In particular, the application layer can be made aware of the buffer handling mechanisms used by the lower layers. Therefore, the application more efficiently reuse buffers. Also, since the application process can use the same memory as the networking code the application can read and write directly to the internal buffers, thus saving the expense of performing a copy. All of this is internal design stuff, but for the end user lwIP provides a tailor made API that does not require any data copying. This kind of scheme makes lwIP suitable for its use in embedded systems.

RTL-lwIP inherits all lwIP's benefits, besides providing new capabilities such as real-time capabilities and the characteristic of having an almost POSIX-compliant real-time operating system running under it.

In conclusion, many benefits can be obtained using RTL-lwIP. Besides providing IPV6 and TCP protocols, RTL-lwIP is more suitable for embedded systems, not only for its code size (RTL-lwIP TCP/IP stack, the driver for the 3Com905C-X card and some tools needed, all of them are less than 88kb sized) but for its memory usage improvements. Furthermore, there is currently a project which tries to do a standalone RT-Linux and, since no Linux benefits can be used, RTL-lwIP becomes an option for networking. Finally, other benefit inherited from lwIP is that all improvements done in lwIP can be easily ported to RT-Linux. For example: the RTL-lwIP distribution has a module which is a HTTP server for RT-Linux which has been ported directly from lwIP.

3.2. Porting lwIP To RT-Linux

lwIP consists of several modules. Apart from the modules implementing the TCP/IP protocols, a number of support modules are implemented. The support modules consist of the operating system emulation layer (*sys_arch* module), the buffer and memory management subsystems, network interface functions and functions for computing the internet checksum. Except the *sys_arch* module, the rest of modules are independent of the operating system, so minimal modifications have been done. Portability of lwIP stack has been achieved by gathering together all the operating system specific function calls and data structures into the operating *sys_arch* module, so when any of the rest of modules need such functions the *sys_arch* module is used.

3.3. lwIP Requirements

Requirements to port lwIP to any operating system are perfectly gathered together into the *sys_arch* module. That module provides a uniform interface to operating system services such as threads management, time management, synchronization features, dynamic memory management and interrupt management. RTL-lwIP implements that interface to access RT-Linux architectural-dependent function calls.

Basically, *sys_arch* provides thread management, semaphores and mailboxes to lwIP.

RT-Linux by itself provides an API to manage threads and semaphores. The *sys_arch* API not only interfaces that API but provides new functionalities such as creating not expulsive semaphores (the operation *signal* doesn't call the scheduler), creating semaphores with timeouts and the possibility of creating RT-tasks from other RT-tasks (which is not default in RT-Linux).

Mailboxes are a communication mechanism for messaging through the TCP/IP stack, but is only used inside the stack. They are implemented with buffers and semaphores, so no other RT-Linux special needs are required here.

Time management is achieved using the new RT-Linux capabilities, POSIX signals and POSIX timers, developed by Josep Vidal [9].

Porting lwIP to RT-Linux also requires a dynamic memory manager. Although lwIP includes a memory management subsystem, this is only used by the operating system independent modules, so in order to reserve memory for data structures inside *sys_arch* other memory manager must be used. This need has been solved with DIDMA (Doubly Indexed Dynamic Memory Allocator) which is a dynamic memory manager -developed by Miguel Masmano- for RT-Linux.

Finally, the interrupt management API provided by RTL-lwIP just interfaces the API offered by RT-Linux.

3.4. Network Drivers

Besides the *sys_arch* module, network drivers must be written in order complete the lwIP stack functionality. In lwIP, device drivers for physical network hardware are represented by a network interface structure which gathers up IP addresses and functions to send and receive packets. Those network interfaces are managed by lwIP drivers which interface the TCP/IP stack with the device drivers. lwIP includes a set of lwIP drivers which are only useful in Linux (except a loopback driver which is used to manage local host destination packets); RTL-lwIP extends that set of lwIP drivers with a lwIP driver for the Ethernet card 3Com905C-X and other for the Realtek8139 card. The implementation of the device driver for that NIC (Network Interface Card) is explained in the next section.

4. Developing a RT-Linux GPL driver for an Ethernet card

The proprietary distribution of RT-Linux provides a proprietary driver for the NIC 3Com905C-X. RTL-lwIP provides an open source driver (GPL license) for the same NIC and, as it is the first free RT-Linux driver for an Ethernet card, this section will try to establish the steps to develop other Ethernet drivers.

4.1. Interfacing the RT-Linux driver

The driver has a POSIX device structure, it means that can be accessed by means of the normal *open*, *close*, *write*, *read* and *ioctl* functions. With this structure, the normal operation mode would be that those threads that want to receive packets would be pooling over the NIC in order to get packets, but this operation mode is not efficient at all when working with network devices since packets arrive at high bursts (with this scheme internal buffers would probably overflow since threads wouldn't get packets with enough speed). The adopted scheme that is much more efficient is using POSIX signals as a way to notify threads of the arrival of packets. The idea is that threads that want to be notified of the arrival of a packet register themselves in the driver, so when a packet arrives the driver would signal that thread. One of the things that a signalled thread must do inside the signal handler is the read call over the driver. One problem derived from the POSIX signals is that when executing a signal handler no more signals are received (the signal being handled is masked inside the handler), so just signalling the thread when a packet arrives is not enough. That problem is solved by the thread by keeping a global variable *pendent_signals* meaning the number of pendent signals. That variable will be incremented by the driver - when signalling a thread - using an *atomic_increment* function registered in the driver during the thread's registering phase. The code of the thread's signal handler would be quite simple:

```
do{
   read(fd,(void *) &receive_buffer, 1536);
}while(dec_pendent_signals());
```

Where dec_pendent_signals is:

```
stop_interrupts(…);

if(pendent_signals==1) retval=0; else retval=1;

--pendent_signals;
allow_interrupts(…);

return retval;
```

4.2. Handling interrupts

A problem similar to that described in the previous subsection occurs in a lower level with the handler of the Ethernet card interrupts. When the driver is handling an interrupt, that interrupt is disabled what means that although the Ethernet card is generating an interrupt, the driver won't receive it. To solve this problem the scheme described above is not possible as the Ethernet card cannot register nor execute any function. The solution is given by the native operation mode of the card. 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 closed list of UPDs (Upload Packet Descriptors) and the pointer passed to the card is called *UpListPtr* (Upload List Pointer) as shown in the figure 2.

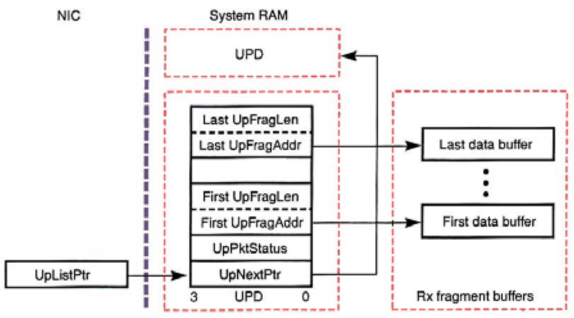

Figure 2. The UPD data structure.

As the list is closed, the last UPD will point to the UPD structure that in the figure is pointed by *UpListPtr*. The UPD structure has a bit in the *UpPktStatus* field that is changed when the card successfully uploads a packet, so being inside the interrupt handler the driver can check if new packets have been uploaded since the execution handler started. The code of the interrupt handler would also keep quite simple:

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

where UPLOADED is a mask for the bit that indicates that a packet upload has been completed. Obviously, when receiving a packet by means of *receive_packet* the field *UpPktStatus* is cleared.

This kind of scheme is used in most of Ethernet cards and solves the problem described above.

4.3. Buffering levels

Another discussion issue to take into account is how many buffering levels are required. As said before, the driver must reserve memory for a closed list of UPDs and it must give the card a pointer to the first element of that list in order to upload to main memory the incoming packets. That buffering level is not enough, since all broadcasts and multicast packets must be uploaded to main memory in order to check if they are addressed to the NIC or not. With one buffering level, if the read call of an incoming packet is not performed immediately, then the memory associated with that packet could be overwritten with a newer packet. Besides that, the NIC must be stalled every time that it is necessary to modify or to check any element of the list described above, and if stalling is done frequently incoming packets would get lost. In conclusion, other buffering level is needed. The copy from the NIC's buffer to the new buffering level is performed by means of the *receive_packet* function.

4.4. Interfacing the driver from RTL-lwIP

Finally, it is interesting to say that, as lwIP (and consequently RTL-lwIP) is designed to take advantage of sharing memory, i.e. that application processes can use the same memory as the networking code, the application is intended to read and write directly to the internal buffers, thus saving the expense of performing a copy. The driver has been designed to comply with that idea and that has been achieved by slightly modifying the *read* call. A normal *read* call would need the user to pass a pointer to a previously reserved memory area where the driver would fill it with data. The *read* call of the 3Com905C-X driver modifies the pointer passed in order to point to the next downloadable packet (driver's internal buffer). This "special" use must be taken into account when using the driver.

5. Conclusions and future work

This paper has presented RTL-LwIP which is a TCP/IP stack aimed at reducing memory usage and code size, making RTL-lwIP suitable for use in small clients with very limited resources such as embedded systems. It also tries to define how to write a RT-Linux driver for an Ethernet card using some new RT-Linux capabilities such as POSIX signals and a dynamic memory allocator called DIDMA. It presents the example of a RT-Linux driver for the Ethernet card 3Com905C-X.

As future work, the minimum CORBA specification [10] will be implemented in RT-Linux basing the communication protocol in RTL-lwIP. Finally, the RTL-lwIP distribution with source code and documentation can be found in the RTL-lwIP home site [5].

6. References and bibliography

[1] The RT-Linux project. http://www.rtlinux.at/
[2] CORBA home site. http://www.corba.org/
[3] Kavaler R., "RTSOCK". ftp://ftp.fsmlabs.at/pub/rtlinux/contrib/kavaler/rtsock-1.0.pre1
[4] lwIP project home site. http://www.sics.se/~adam/lwip/
[5] RTL-lwIP project home site. http://canals.disca.upv.es/~serpeal/RTL-lwIP/htmlFiles/index.html
[6] Barabanov M., Yodaiken V., "Real-Time Linux.", Linux Journal, March. 1996.
[7] OCERA project home site. http://www.ocera.org/
[8] Dunkels A., "Design and Implementation of the lwIP TCP/IP Stack.", Swedish Institute of Computer Science, 2001. http://www.sics.se/~adam/lwip/doc/lwip.pdf
[9] RT-Linux POSIX signals and timers, by Josep Vidal. http://bernia.disca.upv.es/rtportal/apps/rtl-signals/ http://bernia.disca.upv.es/rtportal/apps/rtl-timers/
[10] Specialized CORBA Specifications http://www.omg.org/technology/documents/specialized_corba.htm