A MINIMAL RT-LINUX EMBEDDED SYSTEM FOR CONTROL APPLICATIONS.

J. Vidal, P. Mendoza, J. Vila, A. Crespo, S. Sáez [*,1]

[*] {*jvidal, pabmench, jvila, alfons, ssaez*}*@disca.upv.es*

Abstract: The use of embedded PC in control systems is increasing. There exist a significant number of real-time operating systems but RT-Linux provides several advantages over most of them. RT-Linux shares the processor with standard Linux allowing to run accurately timed applications to perform data acquisition and control systems with Linux applications. However, the possibility of tailoring the Linux to build small versions with specific services permits to build embedded systems. The development of drivers to handle specific control devices and the availabity of high level services at the Linux level make the advantages overcome the drawbacks of use Linux for many control applications. Anyway, customising Linux for this kind of small hardware is not a straightforward work. In this paper we describe a tool RT-LEAST (Real Time Linux Embedded & Autonomous System Tailoring Tool) to automatically tailoring and installing a great variety of embedded systems minimising their requirements. Also, RT-LEAST is applied to design and develop a control application showing the benefits of the process development and the low hardware requirements obtained.

Keywords: control applications, embedded systems, real-time operating systems, Linux based systems.

1. INTRODUCTION

Linux has proved to be a competitive operating system for a wide range of systems. One of the newest fields where it is becoming more appealing is embedded systems. Some of the main reasons why Linux is being preferred against proprietary based solutions is easiness of tailoring, portability and recently, with the introduction of RT-Linux (Barabanov and Yodaiken, 1996), the possibility of building real-time applications. However, customizing Linux to small hardware platforms is not straightforward. There have been several efforts to minimize Linux (Guire, 2000) or customized distributions for proprietary embedded systems. The advantages of building small embedded control systems based on standard real-time operating systems can be summarized in: portability, low cost applications and reusability. Moreover, most of the services included in Unix based operating systems can be incorporated in the embedded system. In a previous work , a tool called RT-LEAST (Vidal *et al.*, 2001) for customize embedded systems based on RT-Linux has been described.

In this paper, we describe the main steps and tools to develop embedded control systems with RT-Linux using RT-LEAST. To show the possibilities of RT-Linux in control applications, a water level control of two tanks has been developed. In this experience an embedded PC based in i386 architecture and the RT-LEAST to setting up the system has been used .

Embedded applications used to develop the control applications at two levels: a real time control and a user interface. While real time control is developed using tasks of RT-Linux with high priority and time restrictions, the user interface runs over Linux, as the idle task of RT-Linux, with all the features offered at this level: Xwin, network, files, etc...

The set-up of an embedded system is not a trivial work. All the configuration issues involved with the system set-up has been solved using the RT-LEAST

[1] This work has been supported by the Spanish Government Research Office (CICYT) under grant TIC99-1043-C03-02.

tool. This tool allows automatically tailoring a great variety of small embedded devices: since biscuits PC to single RAM & processor computers.
This papers presents the main considerations, tools and system requirements to develop embedded control systems using RT-Linux. The paper is structured as follow: section 2 describes the control system: exposing the problem to solve and describing the system used to control it. Section 3 is an introduction of how to develop a control system using RT-Linux, explaining advantages, tools, limitations and requirements of RT-Linux. Section 4 describes the installation of an application using RT-LEAST. Finally, last section presents the conclusions of our work.

2. SYSTEM DESCRIPTION

In this section, the physical process to be controlled and the control system are described.

2.1 *Physical process*

The physical process is a common process in the laboratory with a low level complexity from the control point of view. However, more than the control itself, we are interested on the hardware and software development. The process has two connected water tanks with sensors (pressure sensor), controlled pumps and manual valves (Pons, 2001). The tanks drain an undetermined water quantity throughout the manual valve. The pump pumps water in following a computer output signal. And the sensor gives us an electrical value equivalent to the water level.The position of the valve is ignored. So the water level of each tank is controlled only with the sensor electrical response information and changing the electrical signal to the pump.

2.2 *Application description.*

The problem presents dynamic variations in its environment, and the system must response in a short time. From the user point of view, it is possible to change the regulator parameters and the setpoints of each tank. This changes can be done through a front end with some dials, buttons and output graphical representations.

2.3 *Control system*

The water level control application it is formed by an embedded system with a Data Acquisition Card and network support, and a host system where the user interface is executed.
The embedded system is based in a HS-4500 motherboard, with a 166 MHz Intel's CPU, 256 Mbytes of RAM, and a 32 Mbytes Disc On Chip.
The Data acquisition Card used to interact with the tanks is an ADLINK NuDAQ PCI-6308 Series with 2 output analogic channels and 16 of input.
The developed tool RT-LEAST requires to be installed in the Host PC for setting up and configuring the embedded machine purposes. Additionally, the server has to provide a Xserver to display the user interface.

3. DEVELOPING THE CONTROL SYSTEM WITH RT-LINUX

RT-Linux is a small, deterministic, real-time operating system that handles time-critical tasks and runs Linux as its lowest priority execution thread. In RT-Linux, a small hard-realtime kernel like a single POSIX process sitting on a bare machine and shares the processor with standard Linux that runs as the lowest priority thread of the RT-Linux kernel (Yodaiken, 98).
Nowadays most embedded systems are proprietary, expensive and difficult to tailor and manage. The combination of Linux & RT-Linux offers us a free, open source, powerful, portable and tailor-able operating system in order to deal with embedded systems (Epplin, 97). Also provides the right kind of cross-development tools: from C++ compilers to integrated development environments and GUI's like: RTiC, LabView, Matlab. By running the Linux kernel as a task under a real-time OS, you get all the advantages of hard real-time along with the huge toolkit and features included with Linux (X-window,TCP/IP, file systems, etc..).
Additionally, RT-Linux presents an interface to the real time extensions of POSIX which adds portability at level code of the applications.
The main drawback to build embedded systems using RT-Linux is the necessity to load Linux on top of RT-Linux. Linux is huge but its modular structure makes it easy to strip down for embedded applications. It can be easy trimmed down to fit on a single floppy disk or 2 MB flash disk.

3.1 *RT-Linux in detail*

RT-Linux 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 kernel of Linux predictable, what it does is to build a small microkernel or software layer, called RT-Linux, directly over the bare hardware. Linux runs over this layer.
RT-Linux implements the concept of RT-task and uses its own scheduler for these tasks. The default scheduler that comes with RT-Linux is a pre-emptive, fixed priority scheduler, but it can be easily customized (Balbastre and Ripoll, 2001) . Linux kernel, so all its user processes, is regarded by this scheduler just as another RT-task and shares the processor with other

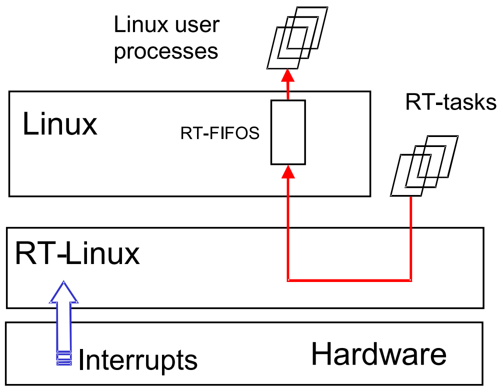


Figure 1. RT-Linux architecture.

RT-tasks (Fig. 1). More precisely, Linux is the lowest priority RT-task and thus, it is executed as a background task, so it only runs when no other RT-tasks are running. RT-Linux also provides full control over interrupts, and it implements a software interrupt manager. It allows either capturing and handling interrupts or just bypassing them to Linux.

3.2 *Developing the water level control application in RT-Linux*

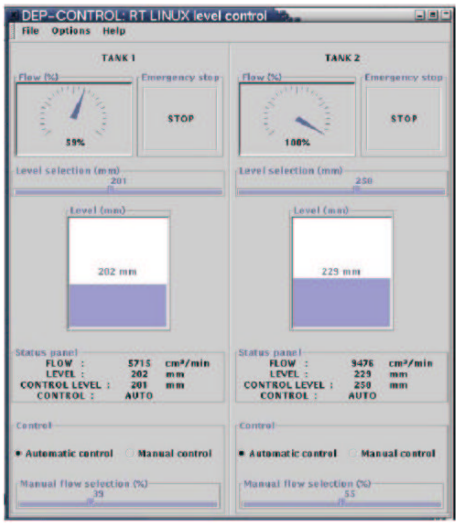
Developing control applications in RT-Linux usually requires splitting the application into two parts. Both parts can be communicated using a special device called RT-fifos.

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

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.

3.3 *Water level control application with Linux & RT-Linux*

Clearly, the application is divided in two main parts as it is shown in fig. 3: the user interface and the real time tasks. The develop of each part has its own characteristics, and each of them must interact with the other.
The ***user interface*** is executed like a Linux process; with the lowest priority, like the idle task of the system.
It allows the user interacting with the processes and shows the response of them, as it is shown in fig.2. It



```
Status panel                          Status panel
   FLOW :           5715  cm³/min        FLOW :           9476  cm³/min
   LEVEL :          202   mm             LEVEL :          229   mm
 CONTROL LEVEL :    201   mm           CONTROL LEVEL :    250   mm
   CONTROL :        AUTO                 CONTROL :        AUTO
```

Figure 2. User Interface.

could operate in two modes for each tank: Manual and automatic mode. In the first the control only stops the pump to avoid water overflowing. In automatic mode, the user select a level and the application must keep the water level close to this level. The interface has an emergency stop button that stops the pump.

The RT-Linux scheduler schedules the ***real time tasks***. The dynamic of the processes to control is slow, so the period of each task could be relatively great (over a half of one second). There are two kind of real-time tasks implemented over RT-Linux:

- The first kind are two tasks, one per tank, that periodically reads the water tank level. The lecture are made through a PC-Lab device.
- The second kind are two tasks, one per tank, that implements the control algorithm of each tank. This task are launched at a bigger period (half of one second) than the first one in order to have stable lectures of the water tank level.

The ***data acquisition card*** driver is implemented over RT-Linux using the COMEDI libraries.
Problems with turbulence effects over the sensor are solved with an average filter. The PC-Lab driver must be modified with this purpose or an additional real time task must be created to do this filtering. Cause these problems the driver must be a periodic task with a sorter period and higher priority than the control tasks.
The communication between user interface and the regulator tasks are made across rt-fifos. And the tasks interact with the PC-Lab driver across the system calls.

3.4 *Monitoring control applications in RT-Linux*

On some control applications is useful to know the stress of the system to make sure that the control algorithms deadlines are accomplished. In our work

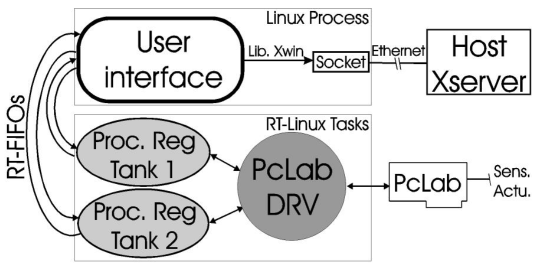


Figure 3. Application structure

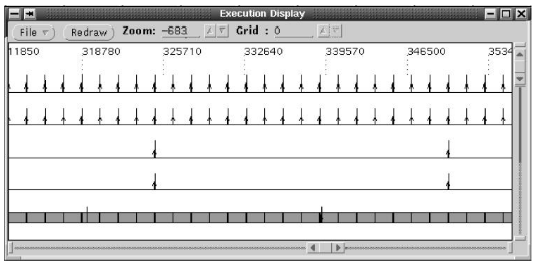


Figure 4. Chronogram of the control tasks.

it has been developed a scheduler version that allows displaying a chronogram (fig. 4) of the task in execution (Mendoza *et al.*, 2001).

As it is shown in the fig. 4 the system is not stressed, and the deadlines are accomplished widely. The first two tasks in fig. 4 correspond with the driver task reading each tank level. The next two correspond to the control algorithm of each tank. And the last one is the idle task, which corresponds to Linux. It is drawing and redirecting the control user interface and the execution tasks chronogram to a HOST via the network.

3.5 *Measurements of some timing parameters*

Some measurements about context switching times in RT-Linux have been made in the execution of the previous program (Mendoza *et al.*, 2001). It has been observed that the context switching time is different when it is caused by an interrupt or by a system call (Fig. 5).
In the case of interrupts, this time is very variable, but it can be up to 20 microseconds in the worst case. From these 20 microseconds, the scheduler uses about 12 to execute the scheduling algorithm. The variability is imputed to the effect of the Pentium cache. Times are always higher when switching from Linux to a RT-task than when switching from a RT-task to Linux. This is because the cache invalidation effect is much more important in the first case. In the case of system calls, the context switching time is much lower: 8 ms. The scheduler requires about 4 microseconds.

Another aspect to take in count is the minimum period of a RT-Linux task. In our embedded PC is about 33 microseconds.

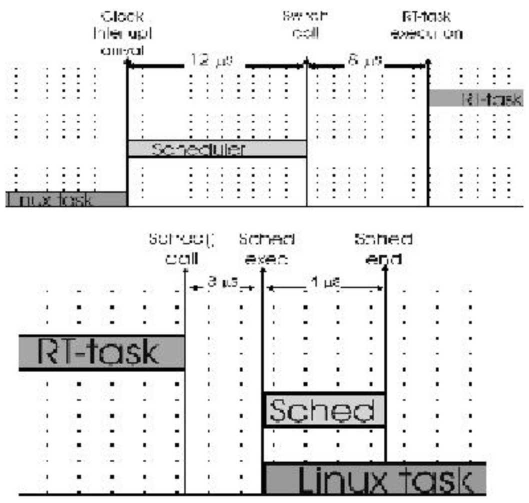


Figure 5. Context switching times.

4. INSTALLING THE CONTROL SYSTEM USING RT-LEAST

The RT-LEAST (Real-Time Linux Embedded & Autonomous System Tailoring) tool (Vidal *et al.*, 2001) has been developed in order to solve all the issues involved in setting up of an embedded system. RT-LEAST is a graphical tool able to tailoring a large set of embedded systems in an easy manner. There are three basic steps involved in setting up an embedded system:

- Building the execution support.
  - Customizing a kernel.
  - Building the root file system.
- Configuring the root file system and boot devices.
- Set-up of the embedded PC.

4.1 *Building a run-time environment*

Nowadays a typical Linux installation can easily consume over 1Gb of space if you take all the options. When you load up extra language tools, office suites, graphics programs and trial software, you need a sizeable disk partition to hold it all. However embedded systems usually have few storage memory and don't need the major part of the programs that we can find in a current Linux distribution. In order to fit Linux into embedded system it must be reduced the file system and kernel size. In the next sections we comment how to built a custom runt-time environment and the way of inserting it in our embedded system using the RT-LEAST Tool.

4.1.1. *Customizing a kernel* Customizing a kernel that supports all the hardware requirements of the embedded system is not a trivial task. The RT-LEAST tool provides a set of precompiled kernels. Each precompiled kernel supports different kernel features, like file systems,network, block devices like DOC, etc,

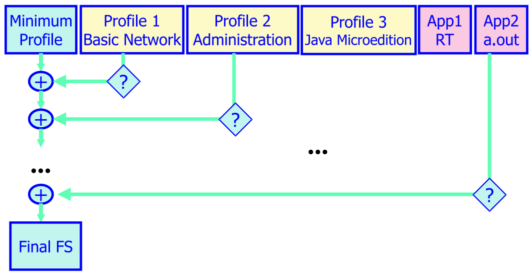


Figure 6. Combining profiles incrementally.

depending of the kernel size. The precompiled kernels have the network cards as precompiled modules, so the user only needs to choose the particular network card of the target system. If the set of precompiled kernels does not cover all the requirements of an embedded system, the tool always gives the option of compiling a new kernel. The RT-LEAST precompiled kernel used in the water tank control application is about 460k and has this main features: several file systems (Minix, ext2, NFS, Disk On Chip), network access (ip autoconfiguration, bootp protocol),block devices (Disk On Chip, RAM disk), into others.

4.1.2. *Building a file system* One of the most important sections of customizing Linux for an embedded system is minimizing the required file system (Cass, 2001). However this is not an easy task, since different applications may require different libraries and utilities. The idea for configuring the required file system of a given application is based on the concept of ”profiles”. A profile is basically a package containing a set of libraries, application programs, and configuration files, for allowing a given programming environment. In a profile are isolated all the libraries and binaries needed by the system to achieve a specific feature. The idea behind profiles is to combine them incrementally as shown in Fig. 6. A file system configuration tool generates a file system that contains the minimum profile, other selected profiles (as zipped tar modules) and the application files themselves, that are configured yet as another profile. The idea of profiles allows developing the system in a modular and flexible way. The main goal is to create a file system made up of modules. Depending on the services needed, different packages can be installed. In order to run the water tank control applications are needed the minimum profile for booting and basic execution support and the network profile to allow redirecting the user interface to another HOST and provide telnet access to the embedded PC. In the actual state of the project four profiles have been identified and implemented: the minimum profile, the network profile, the administration profile, and the Java 2 Micro edition profile. The GNAT profile (GNU Ada compiler) is now being developed. In table 1 there are shown the required profiles according to the application needs.

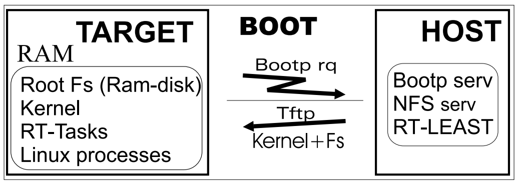


Figure 7. Network booting.

TABLE 1: *Embedded system requirements.*

Processor	386 or up *
RAM	4 Mb or up
Boot device (one of the next)	
Network card with bootp	
temporal connection floppy disk	
No volatile memory	
(Serial port)	
File system capacity	
Basic real time execution support	1 Mb
Network	(+) 130 Kb
On site admin-utilities	(+) 600 Kb
On site debugging	(+) 3.3 Mb
Java 2 Micro-edition	(+) 2 Mb
Java standard	(+) 25 Mb

4.2 *Configuring the root file system and boot devices*

The main goal of an embedded system configuration is to install a run-time environment according to the applications needs and hardware limitations. An autonomous system must be configured by selecting the boot and run-time environment physical device’s. During the design of an embedded system there are several configurations for the root and boot devices and run time execution features with different hardware requirements. In the previous step, the runtime environment has been built. Now, it has to be decided where to store it and how to boot the target systems, taking into account the available devices. The embedded PC that controls the water tank can be configured so it boots using the bootp protocol and places the run-time environment into the RAM memory. To do that, the RT-LEAST tool will configure the host machine so it acts as a boot server for the embedded machine. Next, it will adapt the root file system so it works in RAM disk and will patch the kernel so it uses the RAM disk as the root file system. Finally, it will construct a file that contains the runtime environment, i.e.: the kernel plus the root file system. This file will be transferred through the network when the embedded machine boots, thus providing a quickly and hardware independent way of installing the root-time environment to the embedded system, as shown in fig. 7. Another possibility is to select nonvolatile storage like the embedded PC Disk On Chip as root file system or boot device. For example, this solid memory can be selected for booting and storing the root file system. Alternatively it can be stored a compressed image of the root file system that will be decompressed on RAM disk at boot time. More or less the same combinations are allowed if the system has a floppy disk.

4.3 *Set-up of the embedded PC.*

Once the water tank control application has been developed it is compressed into a tar+gzip file. This compressed file will contain a script that will make the necessary actions to configure and load the application. Then, it will be customized the embedded PC operating system image according to our example using the RT-LEAST tool. First it will be chosen the precompiled kernel offered by RT-LEAST that supports RT-Linux, the bootp protocol and ram-disk file system. Second the file system will be tailored using the minimum and network's profiles plus the control application. Next the RT-LEAST tool will configure the embedded operating system image so it boots from network with the bootp protocol and for allocating the file system on RAM. Also RT-LEAST will configure the HOST so it acts as a bootp server for our embedded machine (fig. 8). Finally the embedded machine is powered on and the operating system image will be transferred from the HOST to the embedded machine by the tftp protocol. When the operating system image is loaded the control application is launch and will begin to control the water tanks. The operator will receive information of what it happens on the HOST by redirecting the user interface and the monitoring application.

5. CONCLUSIONS

In this paper, it has been shown RT-Linux as a free, portable, powerful, tailor-able, open source platform in order to implement control applications on embedded systems. It has been tailored Linux to fit it in embedded system, developing useful tools and techniques for control applications. The water level control of two tanks has been implemented to expose with an example the possibilities of RT-Linux in control systems. Finally, RT-LEAST tool has been developed in order to deal with all the installation issues. This tool makes easy configuring and setting up an embedded system, automatically tailoring Linux to a great variety of embedded devices and making an efficient use of the available resources

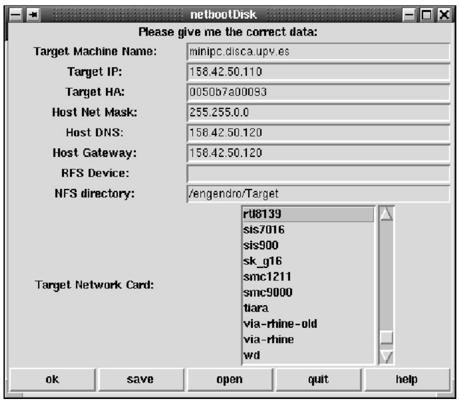
6. REFERENCES

Balbastre, P. and I. Ripoll (2001). Integrated dynamic priority scheduler for rtlinux. *Real time Workshop 2001*.

Barabanov, M. and V. Yodaiken (1996). Real-time linux.. *Linux Journal*.

Cass, S. (2001). Little linuxes.. *IEEE Spectrum*.

Epplin, J. (97). Linux as an embedded operating system. *Embedded.com*.



Please give me the correct data:

Target Machine Name:	minipc.disca.upv.es
Target IP:	158.42.50.110
Target HA:	0050b7a00093
Host Net Mask:	255.255.0.0
Host DNS:	158.42.50.120
Host Gateway:	158.42.50.120
RFS Device:	
NFS directory:	/engendro/Target
Target Network Card:	rtl8139 sis7016 sis900 sk_g16 smc1211 smc9000 tiara via-rhine-old via-rhine wd

Figure 8. Set-up of the embedded PC with RT-LEAST.

Guire, N. Mc (2000). Minirtl - hard real time linux for embedded systems. *4th Annual Linux Showcase & Conference 2000*.

Mendoza, P., J. Vila, S. Terrasa, P. Balbastre and I. Ripoll (2001). Using rt-linux for developing real-time embedded systems. *IFAC Conference on New Technologies for Computer Control*.

Pons, M. Á. (2001). Control del nivel de dos depósitos con el sistema operativo de tiempo real rt-linux.

Vidal, J., P. Mendoza, I. Ripoll and J. Vila (2001). A tool for customising rt-linux to embedded systems. *Real time Workshop 2001*.

Wright, C. and J. Walsh (1999). Hurricane hunting. *Linux Journal*.

Yodaiken, V. (98). The rtlinux manifesto. *The 5th Linux Expo*.