

# Stand-Alone RTLinux-GPL\*

V. Esteve, I. Ripoll & A. Crespo  
Universidad Politecnica de Valencia  
Camino de Vera s/n, Valencia, Spain  
{vesteve, iripoll, alfons}@disca.upv.es

20th October 2003

## Abstract

This paper presents Stand-Alone RTLinux (SA-RTL) which is a porting of the RTLinux-GPL executive to a bare machine. Stand-Alone RTLinux provides the same API than the standard RTLinux-GPL but it do not need Linux to operate. On one hand SA-RTL lacks all the functionalities provided by Linux: lots of drivers, filesystem, graphic user interface, etc; but on the other hand SA-RTL is a faster, smaller, more robust RTOS. SA-RTL enables the use of RTLinux in very small hardware systems with reduced resources. It can be ported to systems with no MMU.

Besides the porting of the RTLinux API we also added two memory protection methods using the MMU (when available).

## 1 Introduction

One of the strongest points of RTLinux-GPL<sup>1</sup> is that it is a hard real-time operating systems that runs jointly with Linux. This combination forms a flexible system with all the functionality and features of a powerful desktop OS (graphic interface, complete network support, lots of hardware drivers, etc.), and a fast hard real-time system. Among the advantages of the standard RTLinux system we can mention: communication between the hard real-time tasks and the non-real-time application processes is fast; it is easy to port RTLinux to any architecture supported by Linux; fast application development, since it is not need to reboot the machine to test the application (if the system do not crash), just reload the application modules.

But the standard RTLinux has also several drawbacks:

- It has a large memory footprint. The Linux kernel code has to be included into the embedded system, and some intrinsic (not selectable) features are included but not used.
- RTLinux can only by ported to systems were Linux were ported previously. Due to design

constrains Linux can only be ported to architectures that support memory paging, therefore a large range of small embedded processors can not be used.

- Although the RTLinux patch has been designed to virtualise the interrupt management, it is possible that some drivers or even user processes disable interrupts or lock the system for long periods of time. The problem is that Linux is still executed in privileged mode (ring level zero in ia32 architecture) so it is possible to directly execute assembler code containing these privileged instructions.
- The more code is executed in the system, the more cache and TLB misses occur. If a low priority Linux application works with a large amount of data or if the user program have low spatial locality, then RTLinux tasks are throw out of the cache, which has a dramatic impact on the performance. This problem can be hardly solved because the cache replacement algorithm is transparently managed by the MMU processor.

---

\*This work has been supported by the European Union project IST-2001-35102

<sup>1</sup>In what follows, RTLinux will refer to the GPL version of RTLinux

- RTLinux threads<sup>2</sup> are executed as regular Linux modules. Therefore, all threads are executed in the same memory space with no protection among rlinux threads nor rlinux threads and rlinux executive or Linux kernel. Memory protection is a key mechanism to implement robust and fault tolerance systems.
- Long boot time. Linux boot and startup sequence may be long for some applications. With the Stand-alone booting is just to load the system image (which contains the application code), setup the memory and interrupt management, and jump to application code.

## 2 RTLinux Stand-alone

Stand-alone RTLinux (SA-RTL for short) is a porting to RTLinux to a bare machine. In SA-RTL there is only support of the hard real-time tasks. It is clear than the range of applications where SA-RTL can be used is quiet different than that of the standard RTLinux. The aim of SA-RTL is not to replace the standard RTLinux, but to open RTLinux to new target niches. SA-RTL is a RTOS suitable for tiny to medium size applications.

SA-RTL has almost no device driver support, only the hardware directly supported by RTLinux can be used in SA-RTL (the user has to implement, or port, the required drivers). On the other hand, SA-RTL provides several memory protection schemes, which can not be easily implemented in the standard RTLinux.

SA-RTL has been a challenging development, since a bare machine lacks may of the debugging and programming facilities commonly used in a Linux system. For this reason, jointly with the implementation of the SA-RTL, several debugging tools has been designed and used during the porting, these tools can now be used to debug user applications.

To build a SA-RTL system the user has to link into a single bootable file the SA-RTL executive and the application code to generate a single file system.

### 2.1 Implementation details

Although at the beginning we tried to port RTLinux starting from the original RTLinux tree, replacing and adding the required code to boot. But this approach showed useless due to the large amount of modifications required to detach the RTLinux code from Linux. It was not possible to run (boot) the system until all non-critical dependencies were removed. Therefore, we changed the porting strategy

to do incremental code porting, that is, generate a small and naive booting image and then continue moving code from the RTLinux tree to the SA-RTL (tasks, context switch, scheduling, synchronisation, etc.).

Part of the booting code (for the i32 architecture) was directly taken from the Linux code itself:

**bootsect.S** The boot sector that loads the remaining part of the kernel and then jumps to the startup code, file **setup.S**. The processor is running in Real Mode.

**setup.S** Calculates the amount of installed RAM and change the processor to Protected Mode. If SA-RTL has been compiled with no memory protection support then paging is disabled and flat memory (that is, all segments start at address 0 and limit 4Gb).

**head.S** This file contains the first function that runs in protected mode: **Startup\_32()**. Stack is initialized and **.bss** section is cleared (this is important o avoid difficult to track bugs). Then **start\_kernel()** is called which continues initialising the system.

Since the kernel size is quiet reduced it do not need to be compressed (most Linux kernels must be compressed, during boot time, due to the small memory space available in real mode. Loading an uncompressed image also speedup the booting time.

Standard RTLinux scheduler manages the Linux kernel as the lowest priority task, in other words Linux is the background task of the RTLinux scheduler. A new “Idle” task has been created to replace Linux as the background task when the system is idle. This idle task has been implemented as an infinite loop at the end of the **start\_kernel()** function.

Memory allocation is still alpha code. A pointer to the end of the used memory marks the start of free space. **kmalloc()** returns the block following the used memory (page aligned) and advances the pointer of used memory. Memory is not freed. Next SA-RTL version will use the TLSF, which is an improved version of the DIDMA allocator also developed as part of the OCERA project.

Although the low level synchronisation used in the code of SA-RTL is spin-locks, SA-RTL is a mono-processor system.

<sup>2</sup>The words “tasks” and “thread” are used interchangeable in this paper

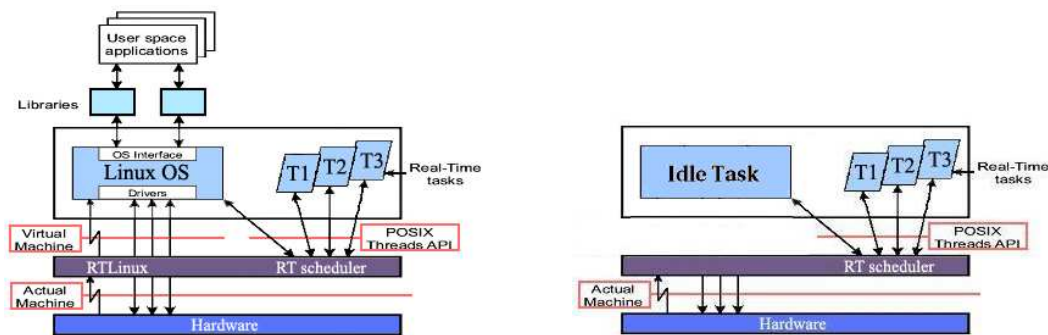


Figure 1: Standard RTL vs. Stand-Alone Structure

### 3 Memory protection

An important feature of real-time systems is “robustness”. Although good programming practices and code review and testing are necessary to produce robust code, it is almost possible to be completely sure to create bug-less code (only small toy code may be free of bugs). One method to improve robustness is by limiting the effect of a programming bug to the task that caused the bug, which can be achieved via memory protection.

The memory protection requirements of embedded systems do not need to be as complex and powerful as that required in a multiuser system. Memory protection in a multi-task multi-user systems has to solve two problems:

1. Programming bugs do not spread beyond the scope of the faulting process. This is achieved by allowing a process to write only on its own data space.
2. Protect data from being stoled by other users (processes). This requirement forces the operating system disable read access outside the each process space.

An embedded system, where the whole application is written by a small group of well intended programmers, is not a battlefield where information is stolen. In this scenario, memory protection is used to intercept and capture programming bugs that passed unnoticed the implementation and testing phases. In fact, system wide read access is a desirable feature in an embedded system because it speed up communication and simplifies the implementation of the RTOS (system calls that check the status like `pthread_self()`, etc.).

Out implementations will focus on write protection only, while allowing system wide read access to all threads.

### 3.1 Execution model

Before defining and implementing a memory protection mechanism in SA-RTL it is necessary to understand the limitations derived from the POSIX thread model. POSIX thread standard define a thread as:

A single flow of control within a process.  
 ...

Anything whose address may be determined by a thread, including but not limited to static variables, storage obtained via `malloc()`, directly addressable storage obtained through implementation-defined functions, and automatic variables, are accessible to all threads in the same process.

Standard RTLlinux do not support several “heavy processes”, just one single process where all the threads are executed, following the POSIX thread model. Therefore all threads are executed in the same memory space. Under this model the only protection that can be performed is to protect RTLlinux executive from user threads writes attempts. In order to provide memory protection between threads, we had to extend the concurrent model to support processes (or what we call contexts), where each heavy process have one or more threads. In this model it is possible to define memory protection between contexts, and so between threads in different contexts.

Three different memory mechanisms has been implemented in the SA-RTL. The user can select which method fits his/her needs in the RTLlinux configuration menu. As summary:

1. **Flat memory space with no protection.**  
 This is the original RTLlinux model, where all the code (RTLlinux executive and application) is compiled and linked as any conventional pro-

gram. There is no overhead when a system call is invoked.

2. **RTLinux executive protection.** This model protects the RTLinux executive against write access from application threads. Application threads are not protected among them.
3. **Context memory protection.** The execution model has been augmented to support several protected execution contexts. In each context lives one or more threads.

### 3.2 Hardware support

Memory protection can be achieved using several hardware mechanisms, among others: segmentation and pagination. Although segmentation seemed to be a better mechanism (lower fragmentation), it has been discarded due to portability issues, and lack of support by compilers (gcc). Segmentation is not supported in most modern processor and is “deprecated” in the i32 family. The underlying processor mechanism used to implement memory protection has been paging.

Currently, memory protection is only implemented in the i32 architecture.

The paging system is only used to protect memory pages, no address translation is performed. That is, the logical address generated by programs (kernel and applications) are translated to the a physical address equal to the logical address. The page size used in 4Kb.

We also use a special i32 feature to implement very fast but effective memory protection (RTLinux executive protection). The WP flag bit in the CR0 control register[Int97] controls whether the page protection bits are honoured (bit set to one) or ignored (bit set o zero) while in supervisor mode. Remember that in standard RTLinux, as well as SA-RTL, all code is executed in supervisor mode (ring level 0). If the WP flag is set, an attempt to access a page protected page triggers a page exception, which is handled by the kernel.

Using the WP bit is very fast, it can be changed by a simple instruction and do not invalidate TLB entries. Most Uni\*es (and Linux is not the exception) set this bit to one to implement the copy-on-write or to mmap files.

### 3.3 RTLinux executive protection

When this memory protection mechanism is selected, paging is activated at boot time. Only one single page directory is initialised so that logical pages are mapped into the same physical pages. Pages that

contain the SA-RTL executive are marked as read-only pages (write is not allowed), and the rest of the pages are marked as read and write. Code and data are page aligned.

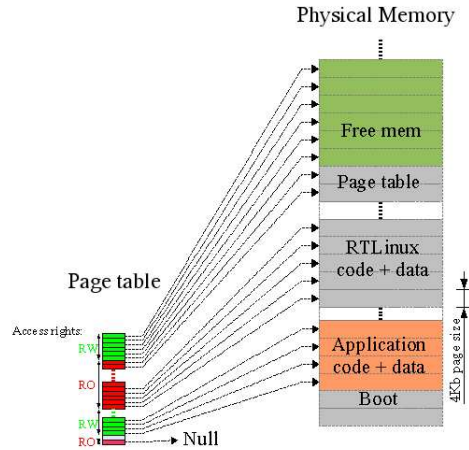


Figure 2: RTLinux executive protection layout

RTLinux to not have a single system entry point (like the 0x80 interrupt in Linux) but the RTLinux executive is linked with the application and the application directly calls the system function. Therefore, changing from protected mode (WP flag = 1) while in application code to non-protected mode (WP flag = 0) while in RTLinux code has to be done manually in each system call.

Two macros has been defined `STARTKERNELCODE` (which clears WP bit) and `ENDKERNELCODE` (which sets the WP bit) when memory protection is selected.

And system calls that require write access to kernel data are surrounded by these macros. The resulting new code is as follows:

```
int pthread_wakeup_np (pthread_t thread) {
#ifdef CONFIG_KERNEL_MEMORYPROT
    mprot_t mprot;
#endif

    STARTKERNELCODE(mprot);
    pthread_kill(thread, RTL_SIGNAL_WAKEUP);
    rtl_schedule();
    ENDKERNELCODE(mprot);

    return 0;
}
```

It is interesting to note that the linker do not need to arrange the executive and application in a special disposition, just keep the objects page aligned. It is also important to note that there is no overhead added to the context switch, all code reside in the same ring level and use the same page table directory. The only overhead introduced is the use of the TLB to “translate” addresses and the `START` and `END` kernel macros.

### 3.4 Context memory protection

This memory protection model works as the previous one but with the addition of a page directory per context and the corresponding page directory change on the context switch between threads of different contexts.

At boot time paging is enabled. A page directory per context is initialised. Page entries are initialised so that logical address generate the same physical address in all page directories, and all are marked as readable. In each context, the pages belonging to that context are marked as writable.

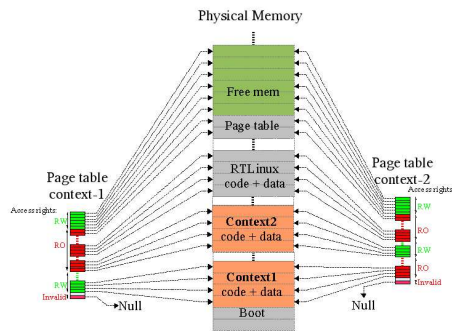


Figure 3: Context Memory Protection

SA-RTL executive is protected using the method described in the previous schema (using the special CR0.WP flag of the processor). And contexts are protected using the different page tables for each context. It has been added just two single assembler instruction to the scheduler: “SET\_CR3” macro. Following is the code added to do the `rtl_scheduler` function.

```
schedulers/rtl_sched.c:
...
    if (s->rtl_current != new_task)
    {
#if CONFIG_CONTEXT_MEMORYPROT
        SET_CR3(context[new_task->contextid].cr3);
#endif
        rtl_switch_to(&s->rtl_current,new_task);
        RTL_TRACE(TASKIN,new_task->tracer_id);
    };

arch/page.h:
#define SET_CR3(v) \
asm volatile ("movl %0,%%cr3 \n":"r" (v));
```

To reduce the number of TLB misses, the pages belonging to the SA-RTL executive are marked as global page. Pages marked as global are not invalidated in the TLB cache when CR3 control register is modified (CR3 is the physical address to the directory page table).

The user can group the threads in contexts just adding a different prefixes to the object filenames.

Object files with the same prefix are linked together in the same context. It is important to note that the use of memory protection is completely transparent to the application code.

## 4 Status

Next is the list of facilities already ported from RTLinux to SA-RTL, and also some facilities not included in the RTLinux distribution but developed in the OCERA project that eventually will be included in the RTLinux-GPL version:

- Fixed priority scheduler. The same scheduler code than RTLinux.
- Timer management. Both periodic and one-shot timer mode.
- Interprocess Communication mechanisms (IPC):
  - Semaphores.
  - Mutexes.
  - Condition variables.
  - Barriers. As defined in the POSIX Standard. OCERA contribution.
  - Signals. Original RTLinux signal code plus the user signals developed by OCERA.
- Debugging tools:
  - GDB Agent
  - Tracer. A non-POSIX tracing facility.
- Memory protection:
  - SA-RTL executive memory protection.
  - Context memory protection.
- Posix I/O Device (RT\_TERMINAL). A console driver for RTLinux developed in OCERA.
- `rtl_printf` support.

Some preliminary performance tests (Ted Baker test) showed a system overhead smaller than the 0.8%.

Also the porting of the SA-RTL to the StrongArm SA1100 processor is in alpha state.

## 5 Conclusions

The paper presents Stand-Alone RTLinux which is a porting to RTLinux-GPL to a bare machine. The porting is almost completed. A novel memory protection mechanism has been proposed and implemented which provides a very low temporal over-

head.

## References

- [Int97] Intel Corporation. *Intel Architecture Software Developers's Manual*, 1997.