Life With Adeos

What is Adeos?

Adeos is a resource virtualization layer available as a Linux kernel patch, which general design has been proposed by Karim Yaghmour in a technical paper, back in 2001.

The current incarnation of this proposal makes it a simple, yet efficient real-time system enabler, providing a mean to run a regular GNU/Linux environment and a RTOS, side-by-side on the same hardware.

To this end, Adeos enables multiple entities - called *domains* - to exist simultaneously on the same machine. These domains do not necessarily see each other, but all of them see Adeos. A domain is most probably a complete OS, but there is no assumption being made regarding the sophistication of what is in a domain. However, all domains are likely to compete for processing external events (e.g. interrupts) or internal ones (e.g. traps, exceptions), according to the system-wide priority they have been given.

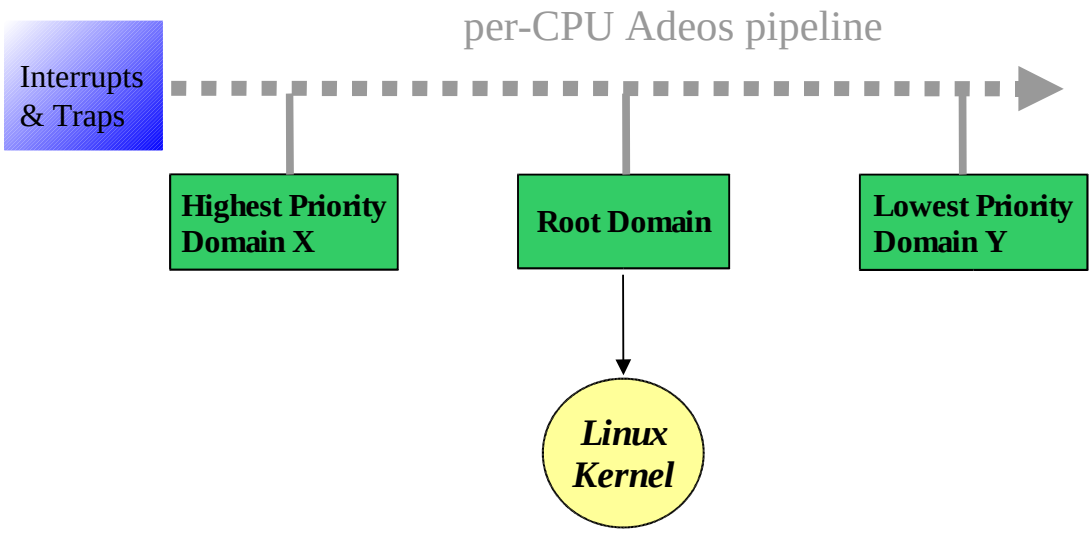
Additionally to its straightforward virtualization capabilities, another key advantage of Adeos relies in its ability to export a generic API to client domains, which does not depend on the CPU architecture. Therefore, much if not most of the porting effort for client domains occurs at the Adeos level.

The event pipeline

The fundamental Adeos structure one must keep in mind is the chain of client domains asking for event control. A domain is a kernel-based software component which can ask the Adeos layer to be notified of:

- Every incoming external interrupt, or auto-generated virtual interrupt;
- Every system call issued by Linux applications,
- Other system events triggered by the kernel code (e.g. Linux task switching, signal notification, Linux task exits etc.).

Adeos ensures that events are dispatched in an orderly manner to the various client domains according to their respective priority in the system, so it is possible to provide for timely and predictable delivery of such events. This is achieved by assigning each domain a static priority. This priority value strictly defines the delivery order of events to the domains. All active domains are queued according to their respective priority, forming the *pipeline* abstraction used by Adeos to make the events flow, from the highest to the lowest priority domain. Incoming events (including interrupts) are pushed to the head of the pipeline (i.e. to the highest priority domain) and progress down to its tail (i.e. to the lowest priority domain). The figure below gives a general view of some Adeos-based system, where multiple domains are sharing events through the pipeline abstraction:



In the figure above, the position of the Linux kernel with respect to the pipelining order might be anywhere; this said, the Linux kernel still has a special role since it stands for the *root domain,* because all other domains need Linux to install them, usually by mean of loading the kernel modules which embody them.

Optimistic interrupt protection

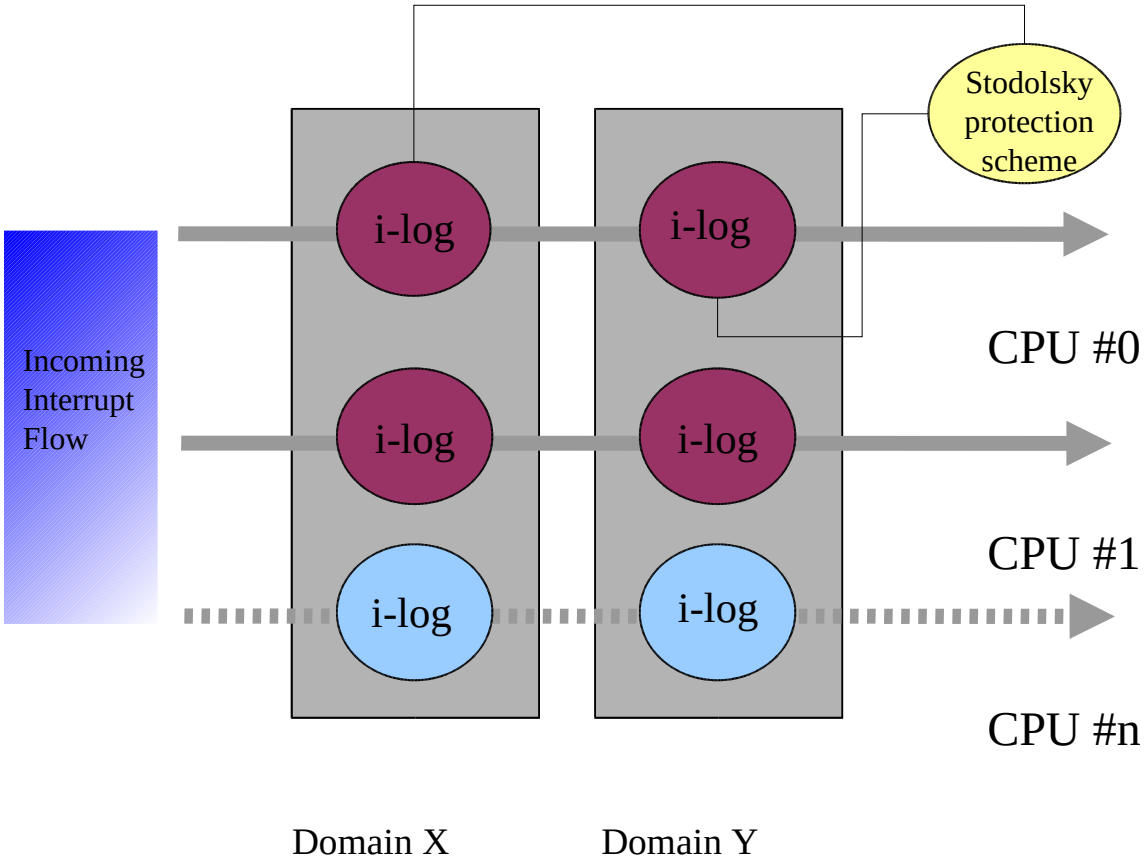
In order to dispatch interrupts in an prioritized manner, while still allowing domains to create interrupt-free sections, Adeos implements the so-called *Optimistic interrupt protection* scheme as described by Stodolsky, Chen, and Bershad, in the following paper: *http://citeseer.nj.nec.com/stodolsky93fast.html*.

The stage of the pipeline occupied by any given domain can be *stalled*, which means that the next incoming interrupt will not be delivered to the domain's handler, and will be prevented from flowing down to the lowest priority domain(s) in the same move. While a stage is stalled, pending interrupts accumulate in the domain's interrupt log, and eventually get played when the stage is *unstalled*, by an internal operation called *synchronization*. Domains use this feature to protect their own critical sections from unwanted preemption by their own interrupt handlers. However, thanks to the virtualization of the interrupt control Adeos brings, a higher priority domain can still receive interrupts, and eventually preempt any lower priority domain. Practically, this means that, albeit an Adeos-enabled Linux kernel regularly stalls its own stage to perform critical operations, a real-time system running ahead of it in the pipeline would still be able to receive interrupts any time, with no incurred delay.

When a domain has finished processing all the pending interrupts it has received, it then calls a special Adeos service which yields the CPU to the next domain down the pipeline, so the latter can process in turn the pending events it has been notified of, and this cycle continues down to the lowest priority domain of the pipeline.

The figure below illustrates how several domains running on mutiple CPUs can share the

incoming interrupts through the Adeos pipeline abstraction. Of course, a correct book-keeping of pending interrupts must be done while a stage is stalled: this is achieved by a per-domain, per-CPU interrupt log, as illustrated below:



System-event propagation

Interrupts are not the only kind of events which can flow through the pipeline abstraction; internal events triggered by the Linux kernel itself or its applications can generate what is called *system events*. Basically, system events are synchronous notifications of trap, exception, or some action performed by the Linux kernel, and which gets notified to any interested parties through the pipeline.

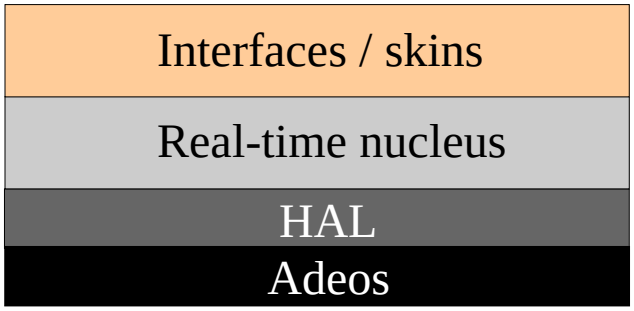
Since those events are synchronous by essence, there is no way to defer their notification through the stall operation, just because you cannot delay their handling. The rationale behind such design decision comes from the fact that a code triggering a system event might just not be able to continue without immediate intervention from the corresponding handler: e.g. the page fault handler must run immediately upon memory addressing exception, delaying it makes no sense. In other words, stall/unstall operations on any given domain only concern interrupts, either real or virtual.

What does Adeos provide to RTAI/fusion?

Conversely, this question could also be: *what basic guarantees does RTAI/fusion need to provide real-time services*? The answer is simple and straightforward: it must be allowed to handle *all* incoming interrupts *first*, before the Linux kernel has had the opportunity to notice them, and it must be able to handle them *immediately,* regardless of any current attempt from the Linux kernel to lock them out using the CPU interrupt mask.

These guarantees give RTAI/fusion predictable interrupt latencies in the lowest micro-second level range whatever activity Linux is undergoing, which once coupled with fusion's fast co-scheduling technique of Linux tasks (i.e. *shadow* threading), provide for deterministic scheduling latencies of real-time threads.

The figure below illustrates the position of the Adeos layer in the RTAI/fusion architecture:



You will notice that the Adeos interface is directly exposed to the Hardware Abstraction Layer which underlies the RTAI/fusion core. Therefore, most of the requests for Adeos services are issued from the HAL, which implementation can be found in the relevant *arch/<archname>/hal* directories, the generic bits being available under *arch/generic/hal*. Having a look at the latter is the best approach to understand how RTAI/fusion makes use of Adeos for its own purposes.

RTAI/fusion's primary and secondary domains

RTAI/fusion allows to run real-time threads either strictly in kernel space, or within the address space of a Linux process. In the rest of this article, we will refer to the latter as the *fusion threads*, not to be confused with regular Linux tasks (even when they belong to the SCHED_FIFO class). All threads managed by RTAI/fusion are known from the *real-time nucleus*.

Support for real-time threads exclusively running in kernel space is a reminiscence of the mere co-kernel era, before the advent of true real-time support in user-space (i.e. LXRT), when RTAI applications would only run embodied into kernel modules; this feature has been kept in RTAI/fusion mainly for the purpose of supporting legacy applications, and won't be discussed here.

What is more interesting is that RTAI/fusion has a symbiotic approach with respect to Linux; this is what makes it different from previous RTAI cores, aka RTAI/classic, like those from the 24.1.x and 3.x series. To this end, *fusion* threads are not only able to run

over the context of the highest priority domain in the pipeline (i.e. the *primary* domain) like kernel-based RTAI threads, but also in the regular Linux space (i.e. the *secondary* domain), while still being considered as real-time by RTAI/fusion, albeit suffering higher scheduling latencies. In RTAI/fusion's jargon, the former are said to run in *primary execution mode*, whilst the latter undergo the *secondary execution mode*.

In order to provide a complete real-time support to threads running in the secondary domain, RTAI/fusion needs the following to be achieved:

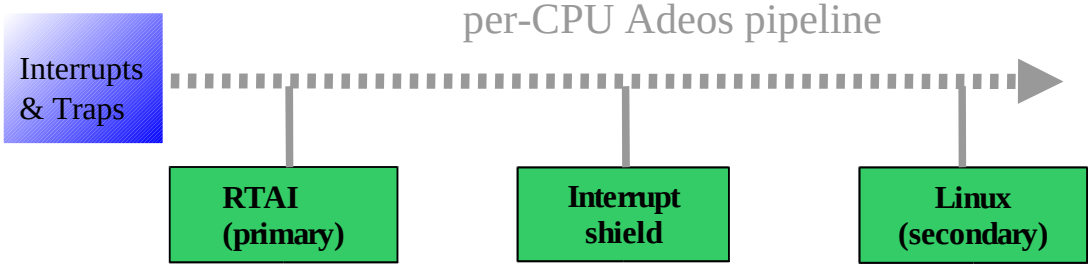
- *Common priority scheme.* Scheduling-wise, we need a way to have both the real-time nucleus and the Linux kernel share the same priority scheme with respect to the set of threads they share the control of; in other words, a *fusion* thread should have its priority properly enforced at any time, regardless of its current domain, among all existing *fusion* threads. RTAI/fusion applies what it calls the *root thread's mutable priority* technique, by which the Linux kernel automatically inherits the priority of the *fusion* thread controlled by the real-time nucleus which happens to enter the secondary domain. Practically, this means that *fusion* threads currently running in the primary domain won't necessarily preempt those running in the secondary one, unless their effective priority is actually higher. The above behaviour is to be opposed to what happens with previous RTAI cores, where threads migrating to the Linux space actually lose their real-time priority in the same move, by inheriting the lowest priority defined by the RTAI scheduler. This said, regular Linux tasks unknown to RTAI/fusion, and which only happen to belong to the SCHED_FIFO class, will always be preempted when competing for the CPU with *fusion* threads from the primary RTAI domain, albeit they will still compete priority-wise with *fusion* threads running in the secondary domain.
- *Predictability of program execution times*. When a *fusion* thread runs over the Linux (i.e. secondary) domain, either executing kernel or application code, its timing should not be perturbated by non real-time Linux interrupt activities, and generally speaking, by any, low priority, asynchronous activity occurring at kernel level. A simple way to prevent most opportunities for the latter to happen is to starve the Linux kernel from interrupts when a *fusion* thread is running in the Linux domain, so that no deferred post-processing could be triggered from top-halves interrupt handlers during this period of time. A simple way to starve the Linux kernel from interrupts is to block them when needed inside an intermediate Adeos domain, sitting between those occupied by the real-time nucleus and the Linux kernel, which is called the *interrupt shield* in RTAI/fusion's jargon. This shield is engaged whenever a *fusion* thread is scheduled in by the Linux kernel, and disengaged in all other cases. It should be noted that the shielding support can be enabled/disabled on a per-thread basis; by default, new *fusion* threads enable it.
- *Fine-grained Linux kernel.* We need the Linux kernel to exhibit the shortest possible non-preemptible section, so that rescheduling opportunities are taken as soon as possible after a *fusion* thread running in the secondary domain becomes ready-to-run. Additionally, this ensures that *fusion* threads can migrate from the

primary to the secondary domain within a short and time-bounded period of time, since this operation involves reaching a kernel rescheduling point. For this reason, RTAI/fusion benefits from the continuous trend of improvements regarding the overall preemptibility of the Linux kernel, including Ingo Molnar's PREEMPT_RT extension. Of course, *fusion* threads which only run in the primary domain are not impacted by the level of granularity of the Linux kernel, and always benefit from ultra-low and bounded latencies, since they do not need to synchonize in any way with the undergoing Linux operations, which they actually always preempt unconditionally.

- *Priority inversion management.* Both the real-time nucleus and the Linux kernel should handle the case where a high priority thread is kept from running because a low priority one holds a contended resource for a possibly unbounded amount of time. RTAI/fusion provides this support, but only the PREEMPT_RT variant does so for the Linux kernel. For this reason, RTAI/fusion keeps an eye and provides support for the current developments of PREEMPT_RT, albeit the mainline kernel still remains the system of reference for now.

As a consequence of the requirements above, when the RTAI/fusion core is loaded, the underlying Adeos pipeline contains three stages, through which all interrupts are flowing, by order of priority:

- RTAI's primary domain, which is the home of the real-time nucleus;
- The interrupt shield domain;
- The Linux domain.



System call interception

Since real-time APIs (i.e. *skins*) which are stackable over the RTAI/fusion nucleus, can export their own set of services to *fusion* threads in user-space, there must be a way to properly dispatch the corresponding system calls, and the regular Linux kernel system calls altogether, to the proper handlers. To this end, RTAI/fusion intercepts every system call trap/exception issued on behalf of any of the RTAI or Linux domains by the *fusion* threads. This is made possible by subscribing an event handler using the proper Adeos service (for more on this, see the *adeos_catch_event()* service, when specifying the ADEOS_SYSCALL_PROLOGUE event). RTAI/fusion uses this capability to:

- Dispatch the real-time services requests from applications to the proper system call handlers, which are implemented by the various APIs running over the real-time nucleus;
- Ensure that every system call is performed under the control of the proper domain, either RTAI or Linux, by migrating the caller seamlessly to the target domain as required. For instance, a Linux system call issued from a *fusion* thread running in the RTAI domain will cause the automatic migration of the caller to the Linux domain, before the request is relayed to the regular Linux system call handler. Conversely, a *fusion* thread which invokes a possibly blocking RTAI/fusion system call will be moved to the RTAI domain before the service is eventually performed, so that the caller may sleep under the control of the real-time nucleus.

The combination of both makes *fusion* threads particularly well integrated into the Linux realm. For instance, a common system call path for RTAI/fusion and the regular Linux applications makes the former appear as a natural extension of the latter. As an illustration of this, *fusion* threads both support the full Linux signals semantics and *ptracing* feature, which in turn enables the GDB support natively for them.

Interrupt propagation

Because it is ahead in the pipeline, the real-time nucleus which lives in the RTAI domain is first notified of any incoming interrupt of interest, processes it, then marks such interrupt to be passed down the pipeline, eventually to the Linux kernel domain, if needed.

When notified from an incoming interrupt, the real-time nucleus reschedules after the outer interrupt handler has returned (in case interrupts are piling up), and switches in the highest priority runnable thread it controls.

The RTAI domain yields the CPU to the interrupt shield domain when no real-time activity is pending, which in turn let them through whenever it is disengaged to the Linux kernel, or block them if engaged.

Adeos has two basic propagation modes for interrupts through the pipeline:

- In the implicit mode, any incoming interrupt is automatically marked as pending by Adeos into each and every receiving domain's log accepting the interrupt source.
- In the explicit mode, an interrupt must be propagated "manually" if needed by the interrupt handler to the neighbour domain down the pipeline.

This setting is defined on a per-domain, per-interrupt basis. RTAI/fusion always uses the explicit mode for all interrupts it intercepts. This means that each handler must call the explicit propagation service to pass an incoming interrupt down the pipeline. When no handler is defined for a given interrupt by RTAI/fusion, the interrupt is unconditionally propagated down to the Linux kernel: this keeps the system working when the real-time nucleus does not intercept such interrupt.

Tips and tricks

Enabling/Disabling interrupt sources

In addition to being able to *stall* a domain entirely so that no interrupt could flow through it anymore until it is explicitely *unstalled*, Adeos allows to selectively disable, and conversely re-enable, the actual source of interrupts, at hardware level.

After having taken over the box, Adeos handles the interrupt disabling requests for all domains, including for the Linux kernel and the real-time nucleus. This means disabling the interrupt source at the hardware PIC level, *and* locking out any interrupt delivery from this source to the current domain at the pipeline level. Conversely, enabling interrupts means reactivating the interrupt source at the PIC level, and allowing further delivery from this source to the current domain. Therefore, a domain enabling an interrupt source *must be the same* as the one which disabled it, because such operation is domain-dependent.

Practically, this means that, when used in pair, the *rthal_irq_disable()* and *rthal_irq_enable()* services which encapsulate the relevant Adeos calls inside the real-time HAL underlying RTAI/fusion, must be issued from the same Adeos domain. For instance, if a real-time interrupt handler connected to some interrupt source using the *rthal_irq_request()* service, disables the source using *rthal_irq_disable()*, then such source will be blocked for the RTAI domain until *rthal_irq_enable()* is called for the same interrupt, and from the same domain. Failing to deal with this requirement usually leads to the permanent loss of the affected interrupt channel.

Sharing interrupts between domains

A typical example of mis-using the Adeos pipeline when sharing hardware interrupts between domains is as follows:

```
void realtime_eth_handler (unsigned irq, void *cookie)
{
      /*
      This interrupt handler has been installed using
      rthal_irq_request(), so it will always be invoked on behalf of
      the RTAI (primary) domain.
      */
      rthal_irq_disable(irq);
      /* The RTAI domain won't receive this irq anymore */

      rthal_irq_host_pend(irq);
      /* This irq has been marked as pending for Linux */
}

void linux_eth_handler (int irq, void *dev_id, struct pt_regs *regs)
{
      /*
        This interrupt handler has been installed using
        rthal_irq_host_request(), so it will always be invoked on
```

```
         behalf of the Linux (secondary) domain, as a shared interrupt
         handler (Linux-wise).
       */
       rthal_irq_enable(irq);
       /*
          BUG: This won't work as expected: we are only unlocking the
         interrupt source for the Linux domain which is current here,
         not for the RTAI domain!
        */
}
```

In the non-working example above, since RTAI/fusion always uses the explicit propagation mode for all interrupts it intercepts, the next ethernet interrupt will be marked as pending in the RTAI log only, waiting for the RTAI handler to possibly propagate it manually down to Linux. But since the interrupt is still locked at pipeline level for RTAI (remember that nobody actually issued the expected *rthal_irq_enable()* from the RTAI domain, but only mistakenly from the Linux one), this won't happen, because the RTAI handler won't run until the lock is removed. Therefore, well, chicken-and-egg problem: we are toast.

Fortunately, there is a solution for sharing interrupts properly, between domains which need to keep the interrupt source disabled until the final processing is done (e.g. dealing with level-triggered interrupts is one of those issues): actually, you don't need to do anything, because Adeos already masks any incoming interrupts at PIC level before feeding the pipeline with it. Therefore, you only need to process the interrupt as you see fit in the relevant domain handler, and make sure to re-enable the interrupt source from the last one using *rthal_irq_enable()*. Whenever the Linux kernel is one of those recipients, the regular kernel handler will do this re-enabling automatically, so basically, you just need to bother calling *rthal_irq_enable()* in handlers which don't propagate the incoming interrupts downstream to the Linux kernel.

Specifically on the x86 architecture, it happens that the timer interrupt is not being masked upon receipt by Adeos, for performances reasons. This said, the timer source is not one you may want to disable in any way, so this is a non-issue.

Interrupt sharing and latency

However, keeping an interrupt source masked while the propagation takes place through the entire pipeline may increase the latency.

Since Adeos guarantees that no stack overflow can occur due to interrupts piling up over any given domain, and because it also stalls the current stage before firing an interrupt handler, there is no need to disable the interrupt source in the RTAI handler. Instead you may even want to re-enable it, so that further occurrences can be immediately logged, and will get played immediately after the current handler invocation returns.

So, the solution is to re-write the previous example this way, this time properly:

```
void realtime_eth_handler (unsigned irq, void *cookie)
{
```

```
	rthal_irq_enable(irq);
	rthal_irq_host_pend(irq);
	/* This irq has been marked as pending for Linux */
}

void linux_eth_handler (int irq, void *dev_id, struct pt_regs *regs)
{
	/* process the IRQ normally. */
}
```

Conclusion

Adeos is a rather simple piece of code, with very interesting properties when used properly. The backbone of the Adeos scheme is the event pipeline, and as such, it brings all the critical features we need in RTAI/fusion:

- Predictable interrupt latencies;
- Precise interrupt virtualization control (per-domain and per-interrupt handler registration, per-domain and per-cpu interrupt masking);
- Uniform, prioritized and domain-oriented propagation scheme for events;
- A generic and straightforward API to ease portability of client code.

RTAI/fusion uses these features to seek the best possible integration of the real-time services it brings with the Linux kernel.

RTAI/classic's (i.e. 24.1.x and 3.x) *hard real-time mode* and RTAI/fusion's *primary mode* both deliver true real-time performances in the lowest micro-second latency range. But when it comes to calling Linux kernel services, the former consider that RTAI tasks doing so are doomed to lose their deterministic behaviour and just don't bother trying to keep them real-time; on the other hand, RTAI/fusion bets on future evolutions of Linux, like PREEMPT_RT, to improve the kernel's overall granularity, so that the secondary mode will still be real-time in the deterministic sense, with bounded albeit higher worst-case latencies. This is the reason why RTAI/fusion is working hard since day one to reach a tight integration level with the Linux kernel. Think symbiotic, seek mutualism.

Useful links

- Karim Yaghmour's Adeos proposal: *http://www.opersys.com/ftp/pub/Adeos/adeos.pdf*
- Optimistic Interrupt Protection: *http://citeseer.nj.nec.com/stodolsky93fast.html*
- The Adeos project workspace on GNA: *https://gna.org/projects/adeos/*
- Where to download the latest Adeos patches: *http://download.gna.org/adeos/patches/*
- The Adeos API reference manual: *http://home.gna.org/adeos/doc/api/globals.html*

rpm@xenomai.org