Enabling Linux for Real-Time on Embedded Multicore Devices

Patrik Strömblad
Principal Engineer
CTO Office

Linux has a strong momentum in the embedded software industry and has in the past years become the prevalent choice as operating system for new platforms. However, standard Linux is designed for overall throughput rather than for real-time, and consequently needs to be modified or extended to meet high demands on latency and determinism. In parallel, advancements and trends within the semiconductor industry are driving an evolution towards manycore processors. By utilizing more flexible processor architectures, new possibilities emerge to enable real-time in Linux.

The real-time application domain has historically been dominated by classic real-time operating systems (RTOS). This is now changing as Linux evolves in a direction towards meeting real-time requirements. In this paper we describe some fundamentals of a real-time system and why real-time capabilities are hard to achieve with the standard Linux kernel, followed by an analysis of different approaches for improving real-time behavior with Linux.

Introduction
Trends in semiconductor technology evolution are constantly enabling processors with an increasing number of cores. The workload on a multicore processor will most likely become even more multiprogrammed to support a mixture of best effort, high throughput O&M and control applications, together with low-level protocol processing applications with potentially high real-time requirements. Such systems must support applications with different requirements for throughput and determinism at the same time on the same processor device, which is hard to achieve since those characteristics are usually contradictory.

Linux is today the preferred operating system also for embedded systems, when possible from a characteristics perspective. Since standard Linux usually does not meet real-time requirements, the Linux kernel must be modified and/or further developed to gain real-time capabilities.

Adding real-time capabilities to Linux has until recent years been a question of how to improve the deterministic behavior on single-core processors. Now, when the multicore technology starts to dominate also the embedded Linux market, new opportunities emerge to make use of multicore technology to simply avoid the resource conflicts that can occur in the existing Linux kernel design. Given the fact that most new processors will have multiple cores, the Linux real-time research should focus on the best ways to enable scalable multicore solutions for real-time applications.

So far, the only options to mitigate effects of potential resource conflicts in the Linux kernel have been the kernel preemption patch approach with numerous patches in the kernel, or the thin kernel approach, which adds a virtualization RTOS layer between the hardware and the Linux kernel. New solutions are under study or already available, for example the vertical partitioning approach that benefits from the availability of multiple cores and adds real-time capabilities to selected cores while preserving throughput performance on other cores. The first two approaches affect all cores of a processor, and are therefore regarded as “horizontal” approaches, while the third one is regarded as a “vertical” approach since it divides the processor into real-time and non-real-time partitions, which is easily illustrated as a vertical line between the partitions.
This white paper describes and compares the three mentioned approaches and provides a brief evaluation of functionality and grade of real-time capability for each approach.

**Definitions and Fundamentals**

Before looking into the real-time enabling Linux solutions, let us start with some background defining the key characteristics of a real-time system and which role the operating system has in such a system.

**What is Real-Time?**

“Real-time” is often and incorrectly taken as a synonym to “high performance”. Real-time systems may be high performing, but strictly speaking, real-time characteristics are expressed in other terms than performance. The performance of a system depends on the system design and obvious factors such as CPU clock frequency, cache and memory bandwidth, and the OS API overhead.

**Response time** is the time elapsed between an external event and the system response to the event. Providing a real-time response is the same as having a deterministic response time.

Real-time systems have well-defined operational deadlines from event to system response, with strict requirements on maximum and average response time. Usually a real-time system requires a response time of microseconds to milliseconds. A system may though still be a real-time one also when the allowed response time is in the range of seconds or higher, as long as the maximum response time can be guaranteed.

**Non-real-time systems** cannot guarantee the response time in any situation and are often optimized for high throughput performance with best effort.

**Hard, Firm, or Soft Real-Time**

Real-time systems are often classified as hard, firm or soft. Missing a deadline in a **hard real-time** system means total system failure, and the impact can be critical. **Firm real-time** systems can tolerate infrequent misses although QoS deteriorates quickly when misses occur. More forgiving are **soft real-time** systems where misses slowly degrade QoS, but the system is still possible to use.

Typical **hard** real-time applications are pacemakers, process and engine controllers, robot control, and anti-lock brakes. Baseband processing and signaling in mobile network equipment and wireless modems are examples of **firm** real-time applications. Less time-critical processing like IP network control signaling, network servers, and live audio-video systems belong to the family of **soft** real-time applications.

The weakest part of the system determines the classification. In a hard real-time system, the response time for all possible combinations of code paths in the OS and application must be verified to prove that the system meets all worst-case deadlines. If not all cases can be verified due to the size and complexity of the OS kernel code or the number of possible combinations being too many, the system cannot be classified as a hard real-time system. Due to the clear definition of hard and soft real-time requirements, it is most common to distinguish between hard and soft real-time.

**Latency and Jitter**

When discussing real-time requirements in an operating system, the term latency is frequently used. There are however several definitions of latency, therefore we need to define how latency is used in this paper.

---

**Figure 1. Task response latency from interrupt to user space**
Interrupt latency – The time elapsed from the signaling of an interrupt request on hardware level until the kernel-level interrupt service routine (ISR) starts to execute. In Linux, the interrupt handling of an external event is usually split into one initial part (the kernel-level ISR triggered by the interrupt), and a second part deferred for later execution (using the tasklet or work queue concept, or as a thread in kernel or user space). The interrupt latency used in this paper consists of the first part only.

Scheduling latency – The time it takes from when a piece of driver code via some kernel API signals a thread (usually a user-space thread) to be scheduled, until the kernel scheduler resumes the actual thread, for example the returning of a blocking call on a device file. A number of sources can trigger the scheduling of a thread, for example an ISR routine in the kernel or another application thread.

Task response latency – The interrupt latency plus the scheduling latency.

Jitter - The difference between minimum and maximum latency.

Real-Time in Operating Systems
In a real-time system, the characteristic behavior of the operating system is very important. To start with, a deterministic response time from an external event until the application is invoked is what we normally refer to when talking about real-time characteristics of an operating system. However, since a chain is not stronger than its weakest link, it is also important to provide a deterministic runtime environment for the entire application execution flow so that it can respond within the specified operational deadline on a system level. This also implies the task scheduler and the resource handling API in the OS must behave in a deterministic way.

When designing a system for high throughput performance, the goal is to keep down the average latency, while designing a real-time system aims to keep the worst-case latency under a specified limit. As a consequence, the design of a both high-performing and real-time capable system must take both average and maximum latency into account.

Historically, the interrupt latency without scheduling, as defined above, is often what people refer to when talking about “latency” since leaving out the scheduling part presents the best figures. Those simplified figures are however becoming obsolete. Drivers are today usually implemented as POSIX applications running in user-space threads as they are easier to debug, maintain, and isolate from licensing issues compared to code in kernel space. The most important measure is thus the task response latency which includes the scheduling latency. Note that it is becoming common to refer to task response latency when talking about interrupt latency, so make sure that any latency discussion considers also the scheduling.

One of the most important requirements for a real-time capable operating system is deterministic task response latency, where the scheduling delay is included in the latency.

Latency and Jitter in the Linux Kernel
To understand where a Linux system may suffer from non-deterministic task response latency, we must first understand the kernel activities involved in the task response sequence.

The sequence above illustrates a number
of places where latency jitter (variance) due to resource conflicts may be added to the total amount of time spent inside the kernel. Some examples:

1. The external event triggers the generation of an asynchronous hardware exception in the CPU.
   - The exception can be delayed as interrupts may be disabled.

2. The kernel invokes the “top half” ISR whose task is to make either the user-space thread or a kernel “bottom half” thread ready.
   - The ISR can be delayed due to low-level kernel locks in the execution sequence from the ISR prologue to its epilogue when exiting.

3. After exit from interrupt level, the kernel enters the scheduler.
   - After returning from IRQ, there are several potential sources of jitter, e.g. pending interrupt or kernel threads on higher priority, or other threads holding resources needed for de-blocking/signaling. Most of the jitter usually derives from this step.

   Most of the potential jitter in the sequence usually derives from the scheduling. The research for making the Linux kernel real-time capable aims to either mitigate or completely avoid the effects from the resource conflicts in the kernel.

   With this knowledge on how the Linux kernel can degrade the real-time response, we will look into three conceptual ways to add real-time capabilities to Linux.

Three Ways to Enable Linux for Real-Time

There are basically two design methods to enable a Linux system for real-time:

Minimize the impact of possible resource conflicts in the Linux kernel by adding different levels of virtualization support on a “horizontal” level, or Avoid resource conflicts that would otherwise make Linux non-deterministic by embracing the fact that the Linux SMP kernel can utilize multiple cores or hardware threads, and design the system “vertically”.

This chapter elaborates the three real-time enabling design approaches mentioned in the Introduction chapter; two of them horizontal, one vertical:

A. The kernel preemption patch approach: Standard Linux can be modified with the PREEMPT_RT kernel patch. This can be considered as a “horizontal layering” design approach since it implements a kind of para-virtualization of the interrupt management and the kernel locking mechanisms.

B. The thin-kernel approach: Use a thin RTOS kernel layer between the hardware and Linux kernel in order to create a truly isolated virtual machine for Linux. Compared to PREEMPT_RT, this approach is an even more obvious “horizontal layering” design approach since it provides virtualization on hardware level using hypervisor technology.

C. The vertical partitioning approach: A Linux SMP kernel running on a multicore processor can be configured and modified to provide resource-isolation between two sets of cores; one dedicated for real-time applications and the other one for non-real-time applications.

Figure 3. Three approaches to enable real-time in Linux systems
The transition to multicore - or actually many-core- technology gradually changes the main task of the operating system from being responsible for time-multiplexing the CPU between tasks, to instead schedule the tasks in space, on different cores. The per-core scheduling becomes less important while inter-core communication needs to be more efficient and deterministic, which will be further elaborated on in this paper.

The different design approaches make different trade-offs in the areas of performance, latency, and functionality. An overview of each approach is given in the figure below, and in subsequent sections we will describe and compare the different approaches.

A: The Kernel Preemption Patch Approach- PREEMPT_RT
The PREEMPT_RT patch provides several modifications to get real-time support in the Linux kernel by mitigating the effects of resource conflicts. This is the well-known and “standard” approach to introduce real-time capabilities to the Linux kernel. The PREEMPT_RT patch has been developed by Ingo Molnar and his team, and is maintained today also by Thomas Gleixner. Most of the patches in this set have been contributed to the main line, but the kernel must be explicitly built for PREEMPT_RT.

The changes in this patch set include re-implementing some kernel locking mechanisms to enable full preemption where otherwise the Linux design states a non-preemptible region. For example, regular spinlocks are replaced with mutexes with priority inheritance, thus many interrupt handlers need to be migrated into kernel threads to become fully preemptible. In a way, PREEMPT_RT could also be seen as a kind of horizontal partitioning, since it is a patch that “para-virtualizes” and redefines the kernel locking design of the standard Linux kernel.

The patches add overhead to the locking kernel mechanisms and will decrease the throughput performance to some degree, depending on the applications. The PREEMPT_RT patch also requires adoptions in driver code and other kernel code that is not originating from kernel.org.

As long as Linux systems used single-core processors, the kernel preemption approach was more or less the only reasonable choice for the embedded market. Since it has also gained acceptance as a multicore alternative, Enea provides a Linux solution including the PREEMPT_RT patch.

B: The Thin Kernel Approach- A Hard Real-Time RTOS Kernel Layer
This design alternative, which offers truly hard real-time and the lowest figures for latency, is based on a thin RTOS kernel layer between the Linux kernel and the actual hardware platform.

The primary use of the thin kernel is to provide a hardware virtualization layer for the Linux kernel. If the actual CPU architecture offers true hardware virtualization support, the overhead introduced by the thin kernel can be kept down to as low as 2-3%.

In a thin kernel solution a Linux kernel image includes the thin kernel startup in the initial phase, hiding the fact that the RTOS kernel initially boots right before the Linux kernel. From a user’s perspective it looks like a regular Linux kernel image is booted, but in addition to the Linux user space there are also RTOS real-time partition(s) – with hard
From Linux user space one can load and manage the applications in the real-time partition and communicate with them using an inter-process communication (IPC) message mechanism.

Although the thin kernel approach has its advantages, mainly hard real-time support coexisting with a standard Linux kernel, the approach does have drawbacks. The real-time and non-real-time tasks are independent, which can make debugging more difficult since it adds another proprietary debug and execution environment. Also, the real-time tasks do not have full Linux platform support (the thin kernel execution is called thin for a reason), but they may provide a limited POSIX API that for example provides access to the Linux file system.

An RTOS kernel can provide a guest environment for Linux in different ways. One way is to provide an entire CPU virtualization environment as described above, which is a so-called “full virtualization”. Another way is to just provide an interrupt virtualization environment, a so-called “para-virtualization”.

Examples of the latter are Xenomai and the Real-Time Application Interface (RTAI).

The thin kernel solution offers excellent real-time characteristics with an average latency below a microsecond, and a worst-case latency limited to one or a few microseconds, and should be an option if figures in this area are required. The thin kernel solution is potentially the only alternative if true hard real-time is required.

C: The Vertical Partitioning Approach with CPU Resource Shielding

The strategy with the vertical partitioning approach is to utilize the fact that SMP Linux runs on multiple cores. We create kernel resource isolation “barriers” between sets of cores, “core clusters”, so that different applications can run on the same processor simultaneously without affecting the characteristics of each other. We can call this “CPU resource shielding”. The goal is to isolate a shielded set of cores into a real-time domain such that its local per-CPU schedulers are not in any way affected by the potential massive load that the applications outside this real-time domain generate in terms of kernel resource locking scenarios or thread preemptions.

The figure below exemplifies a Linux SMP kernel that is vertically divided into two types of execution partitions or domains. Non-critical applications for which to maximize throughput, e.g. O&M and control applications, are located to the non-shielded cores in the “normal”, non-real-time partition. Applications that require real-time task response latency and a deterministic behavior of the execution environment are allocated to the shielded cores in one or more real-time partitions.

Irrespective of whether a core is shielded or non-shielded, the Linux kernel shall be the one and only resource management layer for e.g. memory management and debug support, and the functionality level in these parts of the POSIX execution model should be preserved.

Some changes in the behavior of the applications in the real-time partition need to be introduced, for example:

- Disable the regular load balancing

![Figure 5. Vertical partitioning with CPU resource shielding](Image)
in the Linux kernel for shielded cores, i.e. threads are not automatically moved to shielded cores.

- Explicitly bind IRQs that belong to the real-time application to a shielded core.
- Move the IRQs and kernel threads not belonging to real-time applications to the non-shielded cores.
- Disable the local timer interrupts (removes the ability for the local scheduler to enforce time sharing and to do some book-keeping of resources).

In summary, the end goal is to remove all sources of potential resource conflicts that may add latency to the response from shielded cores.

It is important to note that the full feature level of the POSIX programming API will still be present on the shielded cores, since it is likely needed in startup and configuration scenarios. However, extensive use of the POSIX API will affect both soft and firm real-time characteristics, i.e. the ability to meet deadlines in time and with minimal jitter.

On the set of shielded cores in the real-time partition, the design must explicitly avoid an extensive and uncontrolled use of the normal POSIX API that would otherwise potentially suffer from resource conflicts in the kernel call implementations. Omitting the API is not an option; it would lead to a so called “bare-metal” environment, much discussed in the industry, where existing solutions often lack any kind of OS API for services such as task management, timeout management, and communication. Such a poor environment is undesirable, so a user-space real-time (RT) runtime environment that offers those services in a deterministic, low-intrusive and multicore scalable way needs to be defined.

The execution environment in the real-time partition, illustrated as the RT runtime environment in figure 5, provides a low-overhead, deterministic, and OS-agnostic API to services like interprocess-communication (IPC), timeout management, memory/buffer management, and thread management in user space.

The RT runtime environment runs in the context of a Linux process and should be implemented to avoid kernel calls on the shielded cores as much as possible. Avoiding the kernel is necessary to achieve low overhead and not suffer from side effects from kernel resource conflicts that could add to the latency when the application is in a running state. The RT runtime environment provides necessary support to implement different kinds of scheduling environments. A light-weight task scheduler may be implemented using a restricted set of native threads in Linux, or as a user-space light-weight thread package.

What is often forgotten when discussing so called “bare-metal” design, where applications run entirely in user space with run-to-completion, is the need for a high-speed inter-process communication (IPC) channel between these applications and the code “outside” the bare-metal environment. Such communication can be accomplished either by control messages and/or pure data, but in any case the IPC mechanism needs to be very fast and scalable.

![Figure 6. Vertical partitioning with CPU resource shielding](image)

**Figure 6. Vertical partitioning with CPU resource shielding**
The RT runtime environment should implement such a high-speed IPC “backplane” between the real-time partition and the “outside” by using lock-less techniques and user-space queues. This “backplane” does not only provide a channel to or from the real-time domain, it also provides a high-speed IPC network between applications spread over the cores of the entire SMP Linux processor. It is a challenge to design such an IPC backplane for high speed as well as scalability, yet not affecting the real-time characteristics in the real-time domain.

Research and preliminary benchmarks indicate that it is possible to reach equal or better latency figures using the vertical partitioning approach compared to using PREEMPT_RT. The main advantage of the vertical approach is however not the absolute latency gain itself, but that the Linux kernel on the non-shielded cores can preserve their throughput performance and standard behavior. As a comparison, applying PREEMPT_RT to the Linux kernel would decrease the throughput performance on all cores.

It should be noted that this solution does not introduce any restrictions in the debug capabilities. The full range of open source tools for standard Linux will be available for debugging the real-time partition and its applications, although of course the timing and latency will be affected.

Another area of interest that should be mentioned in this context is power management and power efficiency. Since the vertical partitioning solution uses native Linux and runs within a regular pthread, it will also be able to make use of the power management features that are developed for Linux SMP as long as the RT runtime layer allows yielding when there is nothing to do. We must however be aware of the fact that the characteristics of real-time responsiveness and power efficiency may be contradictory, and trade-offs might be needed depending on the individual configuration. For example, when Linux decides to halt a CPU because it is idle, the algorithm may allow for several steps towards deeper sleep and power saving modes. This also means longer and longer time to start up after a deep sleep, for example the layer 2 cache may have been shut down. So either the worst case latency must accept and include this startup time, or the system should have the ability to “warm up” the cores in time before starting traffic that requires tighter deadlines.

Enea provides a Linux solution for embedded systems on multicore devices that extends Linux with real-time capabilities using the vertical partitioning approach. This solution, Enea Linux/LWRT (LightWeight RunTime), also provides a user-space thread API.

Summary and Conclusions
To conclude, there are a number of ways to introduce real-time capabilities to Linux, and three of them have been outlined in this paper.

Two horizontal partitioning approaches:
The commonly accepted kernel preemption patch PREEMPT_RT which improves the real-time responsiveness to POSIX applications, and the hard real-time thin RTOS kernel which acts as a hardware abstraction layer for Linux and provides an OS API for real-time applications outside Linux.

One vertical partitioning approach:
CPU resource shielding with IPC and RT runtime environment in user space, which is based on standard SMP Linux, and equipped with a high-performance and scalable IPC mechanism. This alternative strives to improve the overall design in order to avoid the kernel resource conflicts instead of making workarounds to minimize the effects from them.

In systems with SMP Linux on multiple cores, the vertical partitioning method will most likely provide the ability to give both high general-purpose throughput capabilities and good real-time capabilities, with good real-time capabilities, with good real-time capabilities. The fact that the standard Linux kernel design continuously improves with regard to the kernel resource separation on core basis, i.e. the “space-partitioning capabilities”, is a promising foundation for the vertical partitioning approach as being a good solution for next-generation Linux-based embedded multicore platforms.

Enea as a Linux Partner
Enea has a long and proven track record as a provider of reliable, distributed real-time solutions for the telecom market and real-time embedded systems.

Enea develops and provides multicore real-time operating systems and Linux solutions with real-time capabilities. Open source software has through the years been a natural part of the Enea product portfolio. Recently Enea has consolidated its presence as an open source provider by releasing the Enea Linux distribution that is Yocto Project compatible, and by adding solutions to improve the real-time capabilities in Linux systems. Consequently, Enea provides solutions for all design approaches described in this paper:

- Enea Linux with PREEMPT_RT
- Enea Linux with a thin kernel solution based on OSE micro-kernel with virtualization support
- Enea Linux/LWRT providing vertical partitioning with a runtime threading package

The extensive know-how of real-time, IPC, Linux, and multicore techniques at Enea brings valuable experiences for supplying the market with future efficient real-time enablers for Linux. On www.enea.com you can read more about Enea Linux and other solutions from Enea.
Enea is a global software and services company focused on solutions for communication-driven products. With 40 years of experience Enea is a world leader in the development of software platforms with extreme demands on high-availability and performance. Enea's expertise in real-time operating systems and high availability middleware shortens development cycles, brings down product costs and increases system reliability. Enea's vertical solutions cover telecom handsets and infrastructure, medtech, industrial automation, automotive and mil/aero. Enea has 750 employees and is listed on Nasdaq OMX Nordic Exchange Stockholm AB. For more information please visit enea.com or contact us at info@enea.com.