Efficient Design of High-Resolution Timekeeping in Real-Time Operating Systems

Federico Terraneo
DEIB – Politecnico di Milano, Italy

Daniele Cattaneo
DEIB – Politecnico di Milano, Italy

Abstract

High-resolution timekeeping is a desirable feature in real-time operating systems targeting microcontrollers, which traditionally has been held back due to its impact on context switch overhead. In this paper we present the design of a timing subsystem that decouples preemption from the timekeeping operation. This design, making use of 1+N hardware timers, significantly speeds up the context switch code while scaling effectively to multi-core microcontroller architectures with N cores. Preliminary experimental results on the Miosix fluid kernel show the effectiveness of the proposed design.

2012 ACM Subject Classification Computer systems organization → Real-time operating systems; Software and its engineering → Scheduling; Software and its engineering → Multiprocessing / multiprogramming / multitasking

Keywords and phrases RTOS, Task Scheduling, Multiprocessing

Digital Object Identifier 10.4230/OASIcs.NG-RES.2026.4

Supplementary Material *Software*: `https://github.com/fedetft/miosix-kernel.git` [16]
archived at `swh:1:dir:8104ca885b935dab2efb9b52c0eb0339f022d668`

Funding We acknowledge financial support under the National Recovery and Resilience Plan (NRPP), call for tender No. 104 published on 02/02/2022 by the Italian Ministry of University and Research (MUR), funded by the European Union – Next Generation EU, Mission 4, Component 1, CUP D53D23008590001 and D53D23008600006, project title LibreRT.

1 Introduction

Many computer applications – and in particular real-time applications – rely on the operating system to provide time information and the scheduling of specific routines at specified time intervals. Time management in an operating system thus plays a central role in a wide range of applications spanning the entire computing continuum, from the general-purpose to the embedded domain. To serve these needs, operating systems include a specific component, that in this paper we refer to as the timing subsystem, to expose time-related APIs to applications and to parts of the operating system itself, such as the task scheduler. This timing subsystem component operates by interacting with hardware timers, acting as the underlying source of time-awareness, and its design greatly affects the capabilities and performance of time measurement. While general-purpose operating systems switched long ago to high-resolution timing subsystems providing sub-microsecond timing resolution [15], many real-time operating systems reamin to this day stuck to more ancient designs only capable of providing timing resolutions in the order of a millisecond [6].

In this paper, we discuss the two main types of timing subsystems, tick-based and high-resolution, and propose an improved high-resolution timing subsystem that is well suited to microcontrollers. Our timing subsystem, called 1+N, can provide high-resolution time while limiting the overhead on the operating system scheduler, and can be easily scaled to multi-core

© Federico Terraneo and Daniele Cattaneo;
licensed under Creative Commons License CC-BY 4.0
7th Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2026).
Editors: Hazem Ismail Ali and Harrison Kurunathan; Article No. 4; pp. 4:1–4:15
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

microcontroller architectures. Preliminary experimental results on the Miosix fluid kernel [17] show it can be a promising solution to improve timing resolution in microcontroller-based embedded systems.

2 Related Works

Most literature about timekeeping in operating systems focuses on the resolution of the timers and their implementation in single core architectures. Relatively few works focus on using hardware timers efficiently in the context of a Real-Time OS. High resolution timekeeping in RTOSes has been proposed as far back as 2003, for instance by Etsion et al. [10]. This work explores the behavior of various interactive applications when increasing the timer tick frequency under then-current Linux versions, and finds much improved latencies in these applications, while incurring only a minor amount of additional overhead. A follow-up technical report from the same authors [19] makes the case for not having a periodic timer tick at all, and scheduling timer interrupts individually. This *tickless* approach in fact had already been proposed, separately from high resolution timers, in a Master's thesis by Barbanov [7], which presents the first attempt to real-time task support in the Linux operating system. Barbanov's work, continued by others, culminated in the *hrtimers* Linux extension [12], which was finally integrated in the mainline source code tree of the kernel, where it remains to this day. The *hrtimers* architecture features a tickless architecture using (if available) a global high-resolution timebase, and per-CPU timers to schedule context switches or events when necessary. Linux's approach to timekeeping has many similarities to the work we present in this paper, and similar implementations have been adopted in other desktop operating system kernels such as Apple's XNU [14].

While desktop operating systems have moved to tickless approaches and high resolution timers, embedded RTOSes are still bound by the tradition of the periodic tick. The most notable example is FreeRTOS, which according to the 2024 Eclipse IoT and Embedded Developer Survey [1] is the predominant real time OS targeted primarily at microcontrollers without MMUs. The FreeRTOS kernel assumes that architecture-specific ports expose a periodic tick which calls back into the scheduler to perform preemption. The specific mechanism for timekeeping is also delegated to the specific port, but the APIs for getting the current time return values in ticks, and the platform-independent configuration file has the frequency of these ticks as one of the primary options. Note that although FreeRTOS advertises tickless support, in their design it only means that the periodic tick is temporarly disabled during deep sleep states. Timekeeping and task wakeup is still bound to the limitations of a tick-based design. For this reason, to avoid ambiguities, in this paper we use the term high-resolution timing subsystem instead of tickless timing subsystem. For what concerns other RTOSes, Zephyr has a timing subsystem that – in tickless mode – is similar to Linux *hrtimers* [5], but still based on a traditional design using a single per-core hardware timer both for timekeeping and for use by the scheduler. Interstingly, on ARM Cortex-M CPUs hardware limitations prevent the per-CPU timers from being used directly for high-resolution timekeeping – so, one more hardware timer is used. Although our proposal also uses 1+N hardware timers, Zephyr's solution is very different as it uses the secondary timer to measure the clock skew generated by the reprogramming of the per-core timer, without decoupling timekeeping and context switches. Moreover, even with this compensation, tickless mode is still unavailable if the required clock resolution is configured too high.

Finally, there is still research that targets non-tickless kernels. For instance, Heider et al. [13] propose an approach where instead of using a single system timer, multiple ones are

used, partitioned over the running tasks. The experimental evaluation is performed on a modified FreeRTOS kernel. Another work, by Fröhlich et al. [11], argues that tickless kernels are inferior to conventional kernels using periodic ticks. Their findings are in the context of 8-bit microcontrollers, where the overhead of dynamically reprogramming the timer hardware at each context switch may be significant. However most RTOSes with tickless support are aimed at 32 bit microcontrollers, where reprogramming the timer requires much less of the CPU's resources. In this context, the high CPU clock frequencies result in considerable benefits from high-resolution timekeeping.

3 Background and problem statement

In this section, we will detail the traditional single-timer tick-based timing subystem – currently the workhorse of many real-time operating systems targeting microcontrollers – and outline its advantages and disadvantages. Then we will explain how this solution has been extended to a single-timer high-resolution timing subsystem, showing its merits are complementary to the tick-based solution. This will give the necessary context for the introduction of our optimized timing subsystem making use of 1+N hardware timers, which will be discussed in Section 4.

3.1 Tick-based timing subsystems

A tick-based timing subsystem derives its name by the use of a single hardware timer to trigger a periodic interrupt, whose Interrupt Service Routine (ISR) is used to perform the timekeeping operation. In this paper, we refer to this periodic timer ISR as the *tick ISR*. In a tick-based timing subsystem, time is kept by means of a software variable in the kernel that is incremented every time the tick ISR is executed. This variable can have different names in different kernels, here we generically call it the *tick counter*. A typical tick ISR period is 1 kHz, resulting in the time kept by the operating system having a 1 millisecond resolution. Operating systems expose time to applications by means of an API call allowing them to query the current value of the tick counter.

In addition to incrementing the tick counter, the tick ISR is also used to handle the wakeup of sleeping tasks. The sleeping operation is fundamental especially for real-time systems which often involve *periodic tasks* of different priority levels, which are implemented by alternatively performing computation and sleeping till the next period. Indeed, an important part of the timing subsystem in an operating system involves efficiently handling tasks that need to sleep either *for* a given amount of time, or *until* a specified absolute time is reached. The sleeping status of a task is typically recorded by the kernel in a flag in each Task Control Block (TCB) so that sleeping tasks become ineligible for scheduling, freeing CPU time for other tasks. The wakeup time of all sleeping tasks is recorded in a separate data structure, such as a list of sleeping tasks ordered by absolute wakeup time, and the tick ISR is also used to check whether some tasks need awakening. Awakened tasks are removed from the sleeping data structure, and their sleeping flag in the TCB is reset, making them again eligible for scheduling. Finally, current real-time operating systems are predominantly *preemptive*. Every time a task is scheduled, a time limit is given, after which the scheduler reconsiders its past decision, possibly running a different ready task of the same priority instead. Of course, should a higher priority task become ready at any point in time, preemption would occur immediately without waiting the time limit. This time limit is given different names in the literature, in this paper we will call it *time quantum*. It is the task of the scheduler to set this timeout after each scheduling decision. In tick-based timing subsystems, the expiration

of the time quantum is also taken care of by the tick ISR. This is done by using a counter variable, or by having the time quantum be fixed and equal to the tick period – in this latter case the tick ISR will call the scheduler every time. A simplified implementation of a tick-based timing subsystem is reported in Appendix A, Listing 1.

The tick-based timing subsystem described so far is currently a popular choice in real-time operating systems for microcontrollers due to the combined advantages of simplicity and low overhead. However, it suffers from several shortcomings that are becoming progressively more significant due to the recent trend towards the increase in computational power of microcontrollers.

Inherent tradeoff between timing resoution and tick ISR overhead. In a tick-based timing subsystem, incrementing the tick counter is done in software as part of the tick ISR. As such, increasing the timing resolution also increases the overhead, due to the tick ISR interrupting applications and the rest of the operating system more frequently. Indeed, to keep time at 1 millisecond resolution, the tick ISR already needs to be called 1000 times per second. Should the time resolution be increased to microsecond-level or better, one million tick ISR per second or more would be required. It is clear that regardless of the amount of code optimization poured into the tick ISR code, the overhead would be unacceptably high at best, or even overload the entire CPU leaving no time to run the rest of the operating system and applications. As such, a tick-based timing subsystem is incapable of meeting high-resolution time measurement requirements. Examples of such requirements include profiling code execution [8], and timestamping network packets to implement clock synchronization algorithms in distributed systems [18].

Poor compatibility to fast periodic tasks. The time resolution of the timing subsystem also sets the minimum achievable sleep interval a task can perform, and thus the minimum period of real-time periodic tasks. While millisecond-resolution timekeeping was acceptable in the past, modern microcontrollers are increasingly high-performance. Pushed by the increasing complexity of embedded software, microcontollers with CPU clock speeds from 500MHz to 1GHz are now common [2]. It is thus no longer inconceivable to have periodic tasks with sub-millisecond periods when it benefits the application requirements. As such, tick-based timing subsystems are becoming a limitation that prevents fully unleashing the capabilities of modern microcontrollers.

Poor code portability as the tick resolution is changed. The reason why 1 millisecond is currently the *de facto* standard for timekeeping resolution in tick-based timing subsystems, is that it is achievable with an acceptable overhead, and keeping time in milliseconds is reasonable at the application level. While a tick-based timing subsystem could be adapted to have a different tick period, such as $200\mu s$ to support periodic tasks up to 5kHz, application code would need to be adapted to the new tick resolution, introducing a cognitive load in programmers and possibly bugs due to a `sleepFor(10);` no longer meaning to sleep for 10 milliseconds, but for 2 milliseconds. Adding code to the operating system to perform the time conversion from an application-meaningful time resolution – such as microseconds or nanoseconds – to the internal tick resolution would already result in the loss of the simplicity advantages of tick-based timing subsystems. Additionally it would basically already constitute the biggest change needed to upgrade to a high-resolution timing subsystem, which we'll discuss shortly.

Limitations on how long interrupts can be left disabled. An often overlooked problem of tick-based timing subsystem is that the tick period introduces a limitation on how long interrupts can be left disabled. Disabling interrupts is an important operation for operating system code because it constitutes the simplest way to achieve mutual exclusion

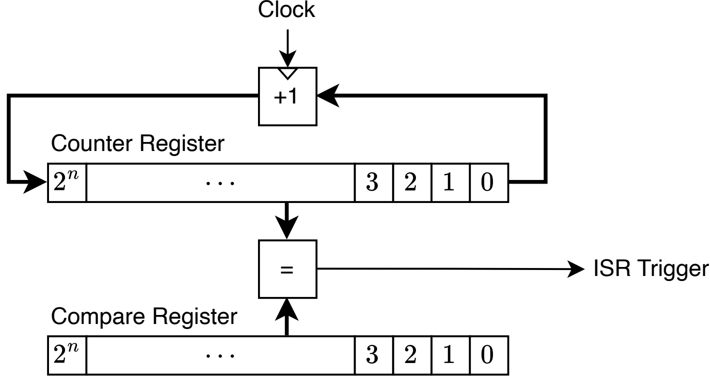

Figure 1 Simplified block diagram of the typical hardware timer found in microcontrollers.

between one or more code sections. However, disabling interrupts for longer than twice the tick resolution causes one or more tick ISRs to not be executed, resulting in a permament accumulation of clock offset in the notion of time for the entire operating system. Of course, attempts to lower the tick period to increase time resolution can only further exacerbate this issue.

Quantization issues approaching the tick resolution. While so far we considered that a given tick period such as 1 millisecond allows timekeeping and sleeping up to that resolution, as time intervals approach the tick resolution, quantization effects come into play resulting in time measurement errors and sleep time errors. This consideration further limits the *usable* time resolution to a multiple of the tick period. Interestingly, this issue also comes into play for the time quantum in scheduling, especially in implementations where the time quantum is equal to the tick period. In this case, should a preemption occur towards the end of the tick period, the newly scheduled task will only be able to use the CPU for a fraction of the expected time quantum (note that the tick ISR cannot be delayed to account for the the preemption as it must be kept strictly periodic not to let its timekeeping function accumulate clock offset). Should a code pattern arise where a task is subject to this shortened time quantum repeatedly, scheduling fairness will be impaired.

3.2 High-resolution timing subsystems

To explain the operation of a high-resolution timing subsystem, a deeper understanding of the architecture of hardware timers is required. Multiple timer architectures exist, not all of them suitable to be used in high-resolution timing subsystems. In this paper we will focus on the most common suitable architecture, that is found – often in multiple instances – in microcontrollers since it is also used for other purposes such as Pulse Width Modulation (PWM). This architecture is composed of a hardware *free-running counter* connected to at least one *match register* capable of generating interrupts, as shown in Figure 1. The free running counter is a sequential logic circuit made of flip-flops and logic gates to implement a binary counter with a fixed number of bits n, most typically 32 or 16 in current microcontrollers. When clocked at a system-specific frequency, often a sub-multiple of the CPU clock, the counter counts from 0 to $2^n - 1$, and then rolls over and keeps counting. The timer counter is memory-mappped, meaning the kernel can read from (and in some architectures also write to) the timer counter. The match register is a memory-mapped register with a number of bits equal to the timer counter. A hardware comparator is present

between the timer counter and match register, constantly comparing the two binary values. Whenever they are equal, an interrupt is fired to the CPU, resulting in the execution of the timer ISR. Additionally, the timer ISR is also called every time the timer counter rolls over to zero, and the software is given a way to differentiate between the two aforementioned events.

A high-resolution timing subsystem takes advantage of the memory-mapped timer counter to overcome the tradeoff between time resolution and ISR firing rate of tick-based systems. Indeed, at first glance, it looks like the operating system API call to query the time could simply be implemented by reading the timer counter and returning its value to applications. However, a slightly more complex implementation is usually required.

First of all, the 16 bit and even 32 bit counters found in current microcontrollers roll over too frequently to provide a monotonically increasing representation of time. Until 64 bit timers become common in microcontrollers, time needs to be kept partly in software and partly in hardware. This is usually done by having a software variable representing the *upper* bits of the time representation, and using the timer counter rollover interrupt to increment the software variable. The current time can then be obtained by combining the timer counter value with the software variable, using simple and efficient bitwise or bit-shift operations. Compared to a tick-based implementation, the ISR overhead for handling timer rollovers is minimal. A 16 bit timer clocked at 1MHz rolls over every 65.536 ms, enabling microsecond-level timekeeping with just ≈ 15 interrupts per second. A 32 bit timer even allows nanosecond-resolution timestamping.

The second issue to account for, is that the frequency at which timers can be clocked is often restricted by hardware, resulting in timekeeping at inconvenient resolutions. For example, a 180MHz CPU may have a hardware timer clocked at 90MHz, resulting in a timekeeping resolution in units of 11.1 ns. To prevent these awkward resolutions from affecting application portability, a time conversion algorithm is necessary to convert the actual timer resolution to either microseconds or nanoseconds, by performing a multiplication by an appropriate scaling factor. Optimized fixed-point time conversion implemenations exist, taking around 30 clock cycles in modern microcontrollers [4].

A complete high-resolution timing subsystem shall also handle the wakeup of sleeping tasks. While a tick-based system relied on the periodic nature of the tick ISR to check for tasks to wakeup, the solution for high-resolution timers makes use of the match register. The solution is as follows: every time a task goes to sleep and it is added to the sleeping data structure, the earliest task that needs to wake up needs to be found, and its wakeup time is written to the match register. As a consequence, the hardware timer will call the ISR when the specified time is reached, and the ISR code will need to perform two operations: first, it shall wake the task, and then it shall update the timer match register to the wakeup time of the next task that needs to wake up. As a result, instead of a periodic tick ISR, the high-resolution timer ISR will be triggered *aperiodically*, and only when a task needs to wakeup. Although for brevity we omitted some details, such as how to handle sleep times longer than the timer rollover period, the number of timer ISR executions in a high-resolution timing subsystem is in general lower than in a tick-based one. However, to set the match register a *reverse* time conversion operation is needed, to convert the wakeup time originally in the application-meaningful time resolution (microseconds or nanoseconds) back into the hardware timer resolution. This operation is usually more expensive, usually taking longer than 100 clock cycles [4]. The last operation that needs discussing to complete a high-resolution timing subsystem is how to handle the preemption time quantum. The most straightforward solution is to also consider this timeout when setting the timer match

register, which should thus be set to the minimimum value between the earliest task wakeup and the time quantum expiration.

Summarizing, the high-resolution timing subsystem described so far uses a single hardware timer with a single match register to serve all timing needs in an operating system. A simplified implementation of such a high-resolution timing subsystem is reported in Appendix A, Listing 2. The advantages and disadvantages of this solution are essentially complementary to the ones of a tick-based system, as in return for its higher complexity and – as we will see shortly – increased context switch overhead, it solves all the shortcomings of the tick-based solution. It clearly achieves high-resolution timing with low timer ISR overhead, can support fast periodic tasks, code portability is ensured by time conversion mapping any hardware timer resolution to a convenient and portable application-level resolution, and quantization is no longer a concern on account of the increased resolution. Finally, for what concerns the maximum timespan in which interrupts can be left disabled, it is now equal to one timer rollover period, thus a much longer timespan than two ticks. With all these positive features, it stands as a research question why are tick-based timing subsystems still prevalent in microcontroller operating systems? In this paper we posit one of the reasons being that high-resolution timing subsystems – as the one just described – slow down the scheduler code and thus the time needed to perform a context switch. The need to handle the time quantum expiration requires the scheduler to interact with the timing subsystem, and in an high-resolution timing subsystem this also requires to perform time conversion operations in the scheduler, which in such a timing-critical component can add a sizable amount of overhead. As such, solutions to reduce the context switch overhead of using a high-resolution timing subsystem are expected to increase their adoption in embedded real-time operating systems.

4 The 1+N timing subsystem

To significantly reduce the impact of a high-resolution timing subsystem on the context switch overhead, we start from noticing that the scheduler only needs to interact with the timing subsystem to make sure the scheduler itself will be called again at the end of the time quantum. Additionally, it should be considered that preemption time points do not need to be synchronized with the rest of the timekeeping functions in the operating system, as they are an implementation detail of the scheduler, and application code should not rely on the exact time preemptions occur. Finally, it should be noted that current microcontrollers feature an abundance of hardware timers. For example, the STM32F469 microcontroller we use for the evaluation has 17 hardware timers [3]. Additionally, some of these timers have more than one match register, making it possible to set separate interrupts independently.

As such, we propose to completely decouple the time quantum operation from the rest of the timing subsystem by serving it through either a separate, dedicated hardware timer or a separate match register. A real-time operating system running on a single core architecture will thus need either two hardware timers or a single timer with two match registers to implement our proposed timing subsystem. For reasons that will become clear when discussing the extension to the multi-core case, from now on we will mostly consider the case of using separate hardware timers.

The proposed timing subsystem – illustrated in Figure 2 – is thus similar in nature to the already described high-resolution one. The time measurement primitive and sleeping operation remain the same, and operate on the timer used for timekeeping that in this paper we will call *timer1*. However, whenever the timer match register needs to be set, only

(a) Single core.

(b) Multicore.

Figure 2 Diagram illustrating the operation of the 1+N timing subsystem and its relation with the hardware timers 1 and 2, both for single core (left) and multicore (right) implementation.

the earliest task wakeup needs to be considered as the time quantum expiration is handled separately. Likewise, the ISR for the timer1 only needs to call the scheduler if the wakeup of a higher priority task than the currently running one occurred.

The preemption time quantum is instead handled by a separate timer, that in this paper we will call *timer2*. The match register for timer2 is set at the end of the the scheduler code to set the time point when the scheduler will need to be called again at the end of the time quantum. The need for a time conversion operation is removed by the fact that the time quantum duration is a constant, allowing either the operating system code to perform precomputation, or relying on the compiler to perform a constant folding optimization. As such, the overhead to the scheduler is significantly reduced. The ISR for timer2 only needs to call the scheduler, as this timer is reserved for handling time quantum expiration.

As will be shown in the experimental results, this timing subsystem architecture can greatly reduce the context switch overhead compared to a single timer solution. A simplified implementation of the proposed optimized high-resolution timing subsystem is reported in Appendix A, Listing 3.

4.1 Extension to multicore architectures

Another significant advantage of the proposed 1+N timer architecture is its scalability towards multicore architectures. Indeed, as multicore microcontrollers become popular, extending real-time operating systems to make efficient use of the available cores is an important research topic.

It should be noted that the application-facing knowledge of time must be kept consistent across software running on multiple cores. Indeed, querying the current time should return the same value regardless of which core the software is running on, and the absolute time point when wakeups from sleep occur must be independent of which core the woken task will be run on. As such, using a single hardware timer for implementing these operations is the simplest possible way to satisfy this requirement. On the other hand, the scheduler in SMP (Symmetric Multi-Processor) operating systems is in most cases implemented as a distributed algorithm, accessing a global thread list to maximize CPU utilization. Using separate lists per-core instead would create a task partitioning problem, which is not easy to solve [9]. Each instance of the scheduler, one per core, is responsible of choosing a new thread to run only on its assigned core, although if the scheduler running on one core finds out that it is necessary to change the running thread on some other core, it may trigger a rescheduling on that core through an inter-processor interrupt. As a consequence, different tasks can begin running on separate cores at different time points, each having a different

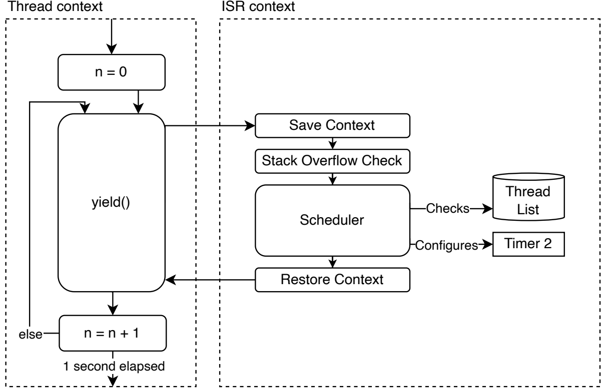

Figure 3 Flowchart of the benchmark used to compare the performance of the 1+N timer implementation compared to the standard high-resolution one.

time point for their time quantum expiration. Moreover, interrupt controllers for multicore architectures let the operating system decide on which core each interrupt request should be serviced and thus on which core each ISR should run on. Finally, some architectures such as ARM provide a dedicated per-core timer called the SysTick. Although this timer hardware is unsuitable for use as a high-resolution timer (it completely lacks a match register), it can be used for handling the preemption time quantum locally on each core.

These considerations led to the design of the 1+N high-resolution timing subsystem. This approach uses a single hardware timer for timekeeping in the operating system, and N additional hardware timers (one per core) to locally call the scheduler on each core at the end of the current task time quantum. From an implementation perspective, the implementation is a straightforward extension of the one in reported in Listing 3. The operation is summarized in the right side of Figure 2. The code of timer2 is replicated N times, and the scheduler is modified to set the time quantum expiration on the timer corresponding to the core it is currently running on. Each timer dedicated to the time quantum expiration will be configured so that the timer ISR will be called on the corresponding core, thereby balancing the ISR load among cores, while the global timer for handling timekeeping and sleeping will be configured to run its ISR on a core chosen by convention, such as the first core.

We implemented the proposed timer architecture in the Miosix operating system, that was previously using a traditional single-timer high-resolution timing subsystem. The source code of the Miosix kernel is freely available under a modified GPL license at the following address: `https://github.com/fedetft/miosix-kernel/tree/unstable`.

5 Experimental Evaluation

We evaluate the 1+N timing subsystem architecture as implemented in the Miosix kernel using a synthetic benchmark that measures context switch overhead by counting the number of context switches per second attainable with the system in an otherwise idle state. Indeed, an important benchmark for a real-time operating system is the performance of its scheduler, to ensure the task activation delay is as low as feasible. An additional goal of this benchmark is to demonstrate the low overhead of the 1+N timing subsystem on multi-core microcontrollers. Figure 3 shows a simplified flow-chart of the benchmark. We perform this benchmark with a single ready thread in the system as we focus on the inherent overhead caused by calling the scheduler and its need to interact with the timing subsystem, rather than on

the performance of the scheduling algorithm data structures for task selection. Note that, despite that consideration, the scheduler still has to examine the thread lists to find if there is any other task that in the meantime has become ready.

We ran the benchmark on two architectures to demonstrate both single-core and multi-core performance. The single core architecture chosen is the STM32F469 microcontroller providing a Cortex-M4 CPU running at 180MHz, while the multi-core one is the RP2040 from Raspberry Pi, which features dual Cortex-M0+ cores running at 200MHz and which is the only architecture that Miosix currently supports in SMP mode. The current development branch of the Miosix kernel provides both a standard high-resolution timing subsystem and our improved 1+N one, switchable through a compile-time option (`OS_TIMER_MODEL_UNIFIED` to disable 1+N). The measurements for 1+N and high-resolution tickless are performed by comparing two builds of the kernel, disabling and enabling this option respectively. Miosix did also support a tick-based timing subsystem, but it was removed in 2023. The comparison with a tick-based timing subsystem was thus done using the last release of Miosix supporting it, which is version 2.22. Since this version does not support SMP or the RP2040, we only provide tick-based data for the STM32F469.

Table 1 Context switches per seconds attainable on Miosix for different hardware configurations, with and without the 1+N timing subsystem (higher is better).

	RP2040	STM32F469
Tick	–	579718
High-resolution	232284	326086
1+N	407326	491801

Preliminary experimental data is shown in Table 1. Looking at the STM32F469 performance, we see that although the data is not fully comparable due to the different kernel versions used, switching from a tick-based timing subsystem to a high-resolution one results in an increase in context switch overhead, visible as a 1.7× decrease in the rate of context switches. No doubt the scheduler is a tightly optimized component of the operating system, so even a small addition such as the need to perform a time conversion algorithm greatly impacts the overall performance. Moving to the 1+N timing subsystem from a high-resolution one affords a 1.5× increase in the maximum context switch rate due to the removal of time conversions in the kernel, nearly regaining all of the previous tick-based performance while benefiting from high-resolution timekeeping.

The speedup of using the 1+N timing subsystem is higher for the dual-core RP2040 architecture, reaching 1.75×. This is both due to the fact that using a core-local timer in the scheduler is faster, and because the time conversion, which requires 64 bit integer operations, is slower on a simple core such as the Cortex-M0+.

6 Conclusion

In this work we provided an overview of how timing subsystems are designed in Real-Time Operating Systems (RTOSes), describing the two main methods for timekeeping (tick-based and high-resolution) and how they relate to the scheduling and preemption functions. Tick-based kernels are the traditional choice, and as such they are simple and fast. At the same time they are severely limited, especially when periodic tasks need to be scheduled to run at a precise time in the future, which is a common need for real-time and safety-critical

applications. High-resolution timing subsystems provide a solution to these problems, by more deeply relying on the hardware to keep a time counter directly. The operating system kernel still performs preemptions and wakeups as before but by directly setting the *compare* register of the hardware timer. This comes at the price of having to perform more complex computations for handling the timer: multiplexing scheduled wakeups and preemption quantum expiration, and conversions from an abstract time unit (such as nanoseconds) to the actual timebase frequency used by the hardware.

Given this premise, we propose the 1+N timing subsystem. This design leverages the abundance of hardware timers in current microcontroller architectures to handle preemption quantum expiration using per-CPU timers (which do not even have to meet the requirements for high-resolution timekeeping, such as the ARM SysTick timers) and on the other hand handles timekeeping and programmed wakeups through a single high-resolution timer providing microsecond to nanosecond resolution. In this way we avoid the overhead of high-resolution timing completely where it is not needed, achieving a boost in the context switch performance up to 1.75× compared to a traditional high-resolution timing subsystem implementation. This brings the overhead of high-resolution timekeeping much closer to the performance of a tick-based kernel. The results we present are preliminary, and we are currently further optimizing our implementation in the Miosix kernel. Moreover we expect this new timer architecture to be easily implemented in other existing real-time operating systems. Future works will be directed towards benchmarking the impact of different timing subsystems on real applications, as well as porting Miosix to microcontrollers with more than two cores to further verify the scalability of the proposed timing subsystem.

References

1. 2024 IoT & embedded developer survey report. `https://outreach.eclipse.foundation/iot-embedded-developer-survey-2024`, Accessed: 2025-03-03.
2. i.MX RT1170: 1 GHz Crossover MCU with Arm Cortex Cores. `https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/i-mx-rt-crossover-mcus/i-mx-rt1170-1-ghz-crossover-mcu-with-arm-cortex-cores:i.MX-RT1170`, Accessed: 2025-11-28.
3. STM32F469xx datasheet. `https://www.st.com/resource/en/datasheet/stm32f469ae.pdf`, Accessed: 2025-11-28.
4. timeconversion.cpp. `https://github.com/fedetft/miosix-kernel/blob/master/miosix/kernel/timeconversion.cpp`, Accessed: 2025-11-28.
5. Zephyr kernel services: Kernel timing. `https://docs.zephyrproject.org/latest/kernel/services/timing/clocks.html`, Accessed: 2025-11-28.
6. The rtos tick. `http://www.openrtos.net/implementation/a00011.html`, Accessed: 2025-12-01.
7. Michael Barabanov. A linux-based real-time operating system, 1997. URL: `https://www.usenix.org/legacy/publications/library/proceedings/ana97/program.html`.
8. Peter B. Danzig and Stephen Melvin. High resolution timing with low resolution clocks and microsecond resolution timer for sun workstations. *SIGOPS Oper. Syst. Rev.*, 24(1):23–26, 1990. `doi:10.1145/90994.91003`.
9. Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4), 2011. `doi:10.1145/1978802.1978814`.
10. Yoav Etsion, Dan Tsafrir, and Dror G. Feitelson. Effects of clock resolution on the scheduling of interactive and soft real-time processes. *SIGMETRICS Perform. Eval. Rev.*, 31(1):172–183, 2003. `doi:10.1145/885651.781049`.

11 Antônio Augusto Fröhlich, Giovani Gracioli, and João Felipe Santos. Periodic timers revisited: The real-time embedded system perspective. *Computers & Electrical Engineering*, 37(3):365–375, 2011. `doi:10.1016/j.compeleceng.2011.03.003`.

12 Thomas Gleixner and Douglas Niehaus. Hrtimers and beyond: Transforming the linux time subsystems. In *Proceedings of the 2006 Linux Symposium*, volume 1, pages 333–346, 2006.

13 Kay Heider, Christian Hakert, Kuan-Hsun Chen, and Jian-Jia Chen. Lazytick: Lazy and efficient management of job release in real-time operating systems. *ACM Trans. Embed. Comput. Syst.*, 24(5s), 2025. `doi:10.1145/3762651`.

14 Jonathan Levin. *Mac OS X and iOS internals: to the apple's core.* John Wiley & Sons, 2012.

15 Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, and Rebecca Isaacs. 30 seconds is not enough! a study of operating system timer usage. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 205–218, New York, NY, USA, 2008. Association for Computing Machinery. `doi:10.1145/1352592.1352614`.

16 Federico Terraneo. Miosix. Software, swhId: `swh:1:dir:8104ca885b935dab2efb9b52c0eb0339f022d668` (visited on 2026-02-25). URL: `https://github.com/fedetft/miosix-kernel.git`, `doi:10.4230/artifacts.25582`.

17 Federico Terraneo and Daniele Cattaneo. Fluid kernels: Seamlessly conquering the embedded computing continuum. *IEEE Transactions on Computers*, 74(12):4050–4064, 2025. `doi:10.1109/TC.2025.3605745`.

18 Federico Terraneo, Luigi Rinaldi, Martina Maggio, Alessandro Vittorio Papadopoulos, and Alberto Leva. Flopsync-2: Efficient monotonic clock synchronisation. In *2014 IEEE Real-Time Systems Symposium*, pages 11–20, 2014. `doi:10.1109/RTSS.2014.14`.

19 Dan Tsafrir, Yoav Etsion, and Dror G Feitelson. General-purpose timing: The failure of periodic timers, 2005. URL: `https://cris.technion.ac.il/en/publications/general-purpose-timing-the-failure-of-periodic-timers`.

A Simplified implementations of timing subsystems

In this appendix we show code listings that exemplify the principle of operation of the timing subsystems discussed in this paper: the classic tick-based one, the high-resolution one, and the proposed 1+N one.

```
1
2 long long tickCounter = 0;
3 Queue<Task *> sleepingTasks;  //Ordered by wakeup time
4
5 void ISRtick()
6 {
7     // Update current time
8     tickCounter++;
9
10    // Wakeup sleeping tasks
11    while(sleepingTasks.front().wakeup_time() >= tickCounter)
12    {
13        Task *task = sleepingTasks.remove();
14        clearSleepFlag(task);
15    }
16
17    // Handle preemption time quantum
18    callScheduler();
19 }
20
21 void scheduler()
22 {
23    runTask(selectTask());
24    // No time related API calls needed, preemption time quantum
25    // expiration tied to tick period
26 }
27
28 // Application-facing time API exposed by the OS
29
30 long long getTime()
31 {
32    return tickCounter;
33 }
34
35 void sleepUntil(long long timePoint)
36 {
37    Task *thisTask = getCurrentTask();
38    sleepingTasks.add(thisTask, timePoint);
39    setSleepFlag(thisTask);
40    yield();
41 }
42
43 void sleepFor(long long timeInterval)
44 {
45    sleepUntil(getTime() + timeInterval);
46 }
```

Listing 1 Tick based timing subsystem.

```
1
2 const int timerBits = 32; //Number of bits of hardware timer counter
3 const long long timeQuantumDuration = ...;
4 long long upperTimeBits = 0;
5 long long nextPreemptionTime;
6 Queue<Task *> sleepingTasks; //Ordered by wakeup time
7
8 void updateMatchRegister()
9 {
10     long long nextISR = min(sleepingTasks.front().wakeup_time(), nextPreemptionTime);
11     hardwareMatchRegister = reverseTimeConversion(nextISR);
12 }
13
14 void ISRtimer()
15 {
16     if(hardwareTimerRollover) {
17         // Update upper bits of current time
18         upperTimeBits++;
19     } else {
20         // Wakeup sleeping tasks
21         bool higherPrioTaskAwakened = false;
22         long long now = getTime();
23         while(sleepingTasks.front().wakeup_time() >= now)
24         {
25             Task *task = sleepingTasks.remove();
26             clearSleepFlag(task);
27             if(higherPriorityThanRunningTask(task))
28                higherPrioTaskAwakened = true;
29         }
30
31         // Handle preemption time quantum
32         if(now > nextPreemptionTime || higherPrioTaskAwakened) callScheduler();
33
34         updateMatchRegister();
35     }
36 }
37
38 void scheduler()
39 {
40     runTask(selectTask());
41
42     nextPreemptionTime = getTime() + timeQuantumDuration;
43     updateMatchRegister();
44 }
45
46 // Application-facing time API exposed by the OS
47
48 long long getTime()
49 {
50     return timeconversion(hardwareTimerCounter | upperTimeBits<<timerBits);
51 }
52
53 void sleepUntil(long long timePoint)
54 {
55     Task *thisTask = getCurrentTask();
56     sleepingTasks.add(thisTask, timePoint);
57     updateMatchRegister();
58     setSleepFlag(thisTask);
59     yield();
60 }
61
62 void sleepFor(long long timeInterval)
63 {
64     sleepUntil(getTime() + timeInterval);
65 }
```

Listing 2 High-resolution timing subsystem.

```
1
2 const int timerBits = 32; //Number of bits of hardware timer counter
3 const long long timeQuantumDuration = ...;
4 long long upperTimeBits = 0;
5 Queue<Task *> sleepingTasks; //Ordered by wakeup time
6
7 void updateMatchRegister()
8 {
9     long long nextISRtimer1 = sleepingTasks.front().wakeup_time());
10    hardwareMatchRegister1 = reverseTimeConversion(nextISRtimer1);
11 }
12
13 void ISRtimer1()
14 {
15     if(hardwareTimerRollover1) {
16         // Update upper bits of current time
17         upperTimeBits++;
18     } else {
19         // Wakeup sleeping tasks
20         bool higherPrioTaskAwakened = false;
21         long long now = getTime();
22         while(sleepingTasks.front().wakeup_time() >= now)
23         {
24             Task *task = sleepingTasks.remove();
25             clearSleepFlag(task);
26             if(higherPriorityThanRunningTask(task))
27                higherPrioTaskAwakened = true;
28         }
29
30         // Only call the scheduler if a higher priority task wakes
31         if(higherPrioTaskAwakened) callScheduler();
32
33         updateMatchRegister();
34     }
35 }
36
37 void ISRtimer2()
38 {
39     callScheduler();
40 }
41
42 void scheduler()
43 {
44     runTask(selectTask());
45     hardwareMatchRegister2 = hardwareTimerCounter2 + timeQuantumDuration;
46 }
47
48 // Application-facing time API exposed by the OS
49
50 long long getTime()
51 {
52     return timeconversion(hardwareTimerCounter1 | upperTimeBits<<timerBits);
53 }
54
55 void sleepUntil(long long timePoint)
56 {
57     Task *thisTask = getCurrentTask();
58     sleepingTasks.add(thisTask, timePoint);
59     updateMatchRegister();
60     setSleepFlag(thisTask);
61     yield();
62 }
63
64 void sleepFor(long long timeInterval)
65 {
66     sleepUntil(getTime() + timeInterval);
67 }
```

Listing 3 A single-core implementation of the 1+N high-resolution timing subsystem.