Large-scale Timer Hardware Analysis and Low-level API Design Towards a Clean-slate Timer Subsystem for RIOT-OS

Niels Gandraß
Niels.Gandrass@haw-hamburg.de
Hamburg University of Applied Sciences
Hamburg, Germany

ABSTRACT

Hardware timers are peripherals found in every embedded system. While being required by nearly all applications running on MCUs, current timer drivers often leave potential for efficiency optimizations, especially when used in low-power scenarios. With the goal of developing an optimized timer-API for RIOT-OS, an open-source embedded OS for resource constrained devices in the Internet of Things, our contribution is threefold. First, we illustrate various use cases of hardware timer as well as related research in this field. Second, we conduct a multi-manufacturer and inter-MCU-family analysis of timer peripherals and present our results in a comparative fashion. We give detailed insight into similarities and differences of various timer types that are supported by RIOT-OS. Existing low-level timer peripheral drivers are furthermore analyzed and generic demands upon such are deduced. Third, we propose a low-level timer-API design that is based on insights from our conducted analysis, the timer driver comparison as well as our review of related work. This contribution shall provide a solid baseline to later derive requirements and deduce applicable implementation techniques for a clean-slate timer-API from.

KEYWORDS

energy efficiency, hardware timers, Internet of Things (IoT), low-power, operating systems, resource constrained devices

1 INTRODUCTION

Hardware timer peripherals are an essential component of all embedded devices [12]. Manufacturers of microcontroller units (MCUs) today offer a large variety of timer modules ranging from general-purpose to highly specialized and application specific peripherals. As the *Internet of Things* (IoT) emerges into our daily lives embedded devices adapt to the new requirements; they become smaller and more energy efficient. To support this rapid growth multipurpose embedded operating systems (OSes) are becoming increasingly popular among developers. These OSes usually provide a high-level API to timer functionalities, though, often only use simple multiplexing of virtual software timers onto a single hardware timer. This leaves potential for optimization. Furthermore, some implementations do not make use of the advanced power saving features provided by specialized low-power timer modules that manufacturers nowadays include in most of their MCUs.

RIOT-OS[1] is such an open-source operating system, explicitly targeted at low-power and resource constrained embedded IoT devices [3]. Its current timer subsystem, namely *xtimer*, multiplexes all virtual software timers onto a single hardware module [2]. The long-term goal of our ongoing research is to develop a new clean-slate timer-API for RIOT-OS. It shall be both able to utilize a wide range of the available timer hardware and to make use of the various power-saving features, including MCU-platform and -family specific ones. This work shall provide a baseline from which requirements for such a new high-level timer subsystem can later be derived. It furthermore shall highlight different implementation techniques and software concepts that are potentially relevant for the aspired timer module.

The remainder of this report is structured as follows. First, we present related work regarding both timer hardware peripherals and software modules for interfacing such in Section 2. Findings from both categories are then summarized and concepts that are potentially relevant for our work are highlighted. Second, we conduct an in-depth analysis of various hardware timer peripherals, described in Section 3. We give detailed insight into similarities and differences of hardware timers currently supported by RIOT-OS, taking all maintained manufacturers and a broad selection of their respective MCU-families into account. Third, low-level timer peripheral driver modules are analyzed and compared in Section 4. Based on insights gathered from both the conducted timer hardware and driver analyses we then propose a low-level timer-API design in Section 5. Last, an outlook on future work with respect to the aspired long term goal of a new timer subsystem for RIOT-OS is given in Section 6 before we finish with concluding words in Section 7. Moreover, the entire results of our conducted timer hardware analysis, including a detailed description of the applied criteria, are provided in the appendix Section A.

2 RELATED WORK

Scientific research highlighted in this section is split into two primary categories addressing a) characteristics of timer peripherals from a hardware point-of-view (Section 2.1), and b) design aspects, algorithms, and implementation techniques used in timer driver software (Section 2.2). Publications were selected according to the relevance for this work, as estimated to the best of our knowledge.

2.1 Timer Hardware

We start by taking a look at research that focuses on timer peripherals from a hardware point-of-view. We further split it into two subcategories as follows. First, publications describing generic concepts of timer peripherals as well as second, comparisons of different MCU-platforms, like we contribute and present in Section 3.

2.1.1 Description of Generic Timer Functions. Operation principles of general-purpose timers as well as their basic set of generic features and characteristics are described by Kamal [12, pp. 152-159].

[1] RIOT-OS project website: https://riot-os.org/ (Accessed 01.12.2019)

The author elaborates on frequently available operation and counting modes, different peripheral states, and various timer properties. Possible applications as well as usage scenarios for general-purpose timers are further depicted. Moreover, other types of timing hardware, here namely real-time-clocks (RTCs) and watchdogs (WDGs), are shortly discussed. For our conducted hardware analysis, common timer characteristics can be inferred from this work. These include among others: counter register width, prescaler availability, and auto-reload capability.

While less detailed than Kamal [12], Susnea and Mitescu [17, pp. 67-68, pp. 87-89] also give insight into general-purpose timer peripherals. The book extends the above publication by describing functions and operation principles of timer hardware which is capable of generating pulse-width-modulation (PWM) output, a feature that is also part of our analysis scope.

2.1.2 Comparison of Embedded Timer Peripherals. A major focus of this work lies on the comparison of different timer modules. Our long-term goal implies utilization of advanced timer features, hence solely superficial analyses of timer properties are insufficient for our purpose. Unfortunately, we found only one documented hardware analysis that elaborates on timer peripherals in-depth.

The timer hardware comparison that was conducted by Susnea and Mitescu [17, pp. 67-91] covers the Motorola HC11, Atmel AVR, and Intel 8051 MCU families. For each MCU platform, timer configuration and usage is outlined and moreover accompanied by detailed examples of different application scenarios. The authors emphasize that each platform offers a distinct set of features, still they all share many common operation principles. These include generation of precise time intervals, measurement of duration, and counting of events. Our aspired timer subsystem must in particular be capable of the first two. The found common operation principles furthermore entail that a timer-API can be platform independent regarding basic timer functions. Nonetheless, it should also be able to expose advanced platform specific features, even though portability might be degraded or lost completely whenever these specific features are used by the application. A generic timer peripheral block is further constructed from the outlined common operation principles. Besides mandatory components such as a counter register or a prescaler, the overall availability of capture channels is hereby identified across all platforms.

Further comparisons between MCUs from different manufacturers can be found in the literature, but they do not cover timer peripherals at the required level of detail. For example, Tsekoura et al. [23] analyzed various MCU families sharing four manufacturers that we also target. These chip manufacturers are STMicroelectronics, Atmel/Microchip, Silicon Labs, and Texas Instruments. However, the conducted research primarily focuses on general execution time and power consumption, while not discussing the impact of timer hardware properties in this context.

2.2 Software Modules

Software aspects that relate to timer drivers also need to be taken into account when designing a clean-slate timer subsystem. Publications depict in this subsection therefore include generic concepts and algorithms as well as application- or OS-specific driver implementations. Furthermore, techniques used in the context of real-time scheduling are highlighted as they are strongly dependent on efficient timer usage [14] and therefore also yield valuable insights.

2.2.1 Generic Design Aspects. Varghese and Lauck [24] describe several different approaches to implement a software timer module. These range from simple list based to prioritized tree-based implementation schemes. Each technique is discussed with respect to its applicability to different usage scenarios. Furthermore, three additional methods, which are based on the timing wheel mechanism [20], are proposed by the authors. These deliver a constant timer maintenance complexity through exploiting hashing and hierarchical relations. We propose to take this fundamental work into account when designing timer management software components.

The implementation proposed by Mincev and Milicev [14] also organizes software timers in a hierarchical tree, placing longer delays closer to the tree leaves. Here, minimizing the maintenance overhead is achieved by distributing timer ticks according to that structure while only propagating every n-th tick to the next layer nodes. Therefore, less servicing for long-running timers is observed.

If an application does not require meeting hard real-time deadlines, *Soft Timers* [1] can be used. They mitigate maintenance overhead during servicing of timer interrupts. More precisely, they reduce the cost that is introduced by saving and restoring CPU state during context switches upon timer interrupts. The key idea of the proposed solution is to only maintain timers if invoking the corresponding service routine is of low cost, as determined by the current system execution state. This approach also allows superseding strict periodic timer interrupts (i.e., system ticks), as advocated by Tsafrir et al. [22] and as in-depth analyzed in [21].

Lastly, Lindgren et al. [13] define a platform-independent timer-API including the design decisions for its implementation. With this approach, a set of virtual timers, each having an independent queue of pending tasks, is multiplexed onto hardware timers. Additionally, an evaluation with respect to computational complexity and correctness under concurrency is performed. Requirements for the underlying timer hardware are defined and evaluated for both the STM32 F4 and NXP LPC1789 MCUs. During this step, the authors identified generic characteristics of the analyzed timer hardware. These are counter width, interrupt capability, prescaler availability, auto-reload functionality, and compare channel count. The authors further emphasize the positive impact on timer maintenance performance from both a large counter width as well as a high number of available compare channels. These properties are therefore also incorporated into our hardware survey in Section 3.

2.2.2 Application-specific Timer Implementations. As the evolution of the Linux timer subsystem is well described, many of its design decisions and documented pitfalls can be taken into account when developing a clean-slate timer driver. Key design aspects of the *hrtimer* module, which uses multiplexing on one-shot hardware timers, are outlined by Gleixner and Niehaus [5]. Bellasi [4] further discusses it in the broader context of power management frameworks for Linux. Here, mainly power-saving optimization techniques, such as deferrable timers [19], are addressed. Patel et al. [15] moreover stress the timer interrupt inference problem, a special form of the priority inversion problem [18]. It arises when the

processing of a high-priority timer expiry gets delayed by running maintenance ISRs of low-priority timers. *TimerShield* is proposed, which introduces priority awareness to the timer subsystem and selectively delays low-priority timer maintenance tasks. We contend that in particular the proposed low-power timer handling techniques shall be considered when specifying a high-level timer module.

Requirements for the timer subsystem of RIOT-OS are defined by Baccelli et al. [2]. Furthermore, the hardware abstraction layers (HALs) and the power-management module are described in detail. For timekeeping tasks, a combination of platform-dependent low-level timer peripheral drivers which then are unified through a high-level timer API, is used to provide application developers with target platform agnostic timers. At the time of writing, *xtimer* implements such high-level functionalities through multiplexing software timers onto a single statically mapped hardware timer. Unfortunately, the current implementation suffers from problems, especially with regard to power-saving in dynamic scenarios. It must be noted that *ztimer*[2], a replacement for the *xtimer* module, is currently developed. While it tackles multiple problems of the *xtimer* module, we argue that it lacks analysis and utilization of many timer features, again particularly with respect to power-saving optimizations. In order to support current efforts by providing insight into the available timer peripherals we contribute our timer hardware analysis and the comparative overview of available low-level timer peripheral drivers, as found in Section 3 and Section 4.

Handziski et al. [7] propose another HAL design, separating the hardware abstraction architecture into three layers, each allowing a different granularity of peripheral access. Hardware-independent APIs are available while at the same time optional access to platform-specific features, at the cost of loosing application portability, is preserved. The authors cover abstraction of a wide range of common MCU-peripherals including timers and moreover illustrate power-saving techniques in the context of low-power wireless sensor networks. The proposed HAL architecture was successfully implemented for the TI MSP430 MCU-family in TinyOS[3]. For the aspired timer-API, providing the ability to optionally utilize platform specific features makes the development of highly optimized and task specific applications feasible. It therefore is considered an essential demand upon the API design proposed in Section 5.

A comparison between periodic and one-shot timers in the *Embedded Parallel Operating System*[4] (EPOS), running on an Atmel AVR MCU, was conducted by Gracioli et al. [6]. Results show that through using one-shot timers, context switches and ISR executions can be drastically reduced at the cost of an increased memory footprint. However, the authors outline that incorporating advanced techniques, such as the above described *Soft Timers*, can reduce some of the negative impacts introduced by one-shot timers.

In addition to the previously described generic timer-API, Lindgren et al. [13] also provide an implementation of the proposed interface for the ARM Cortex-M MCU-family. The authors utilize *Real Time For the Masses*[5] (RTFM), a concurrency framework for building real-time systems. Here, the Cortex-M SysTick- and debug-timers are used as base timers for multiplexing and counting of clock cycles. Maintenance complexity, dispatch latency, and hardware setup time are characterized by the authors.

2.2.3 Real-time Scheduling Based Approaches. Even though RIOT-OS only offers soft real-time capabilities [2] and scheduling algorithms are not within our primary focus, there still are lessons that can be learned from the following real-time-scheduling solutions.

Jupyung Lee and Kyu-Ho Park [11] utilize interrupt prediction while also distinguishing between urgent and non-urgent timer interrupts. When no interrupt from an urgent timer is expected, the system tick period is reduced, hence fewer wake-ups occur. This dynamic adjustment of wake-ups allows to meet real-time requirements, unlike the previously mentioned *Soft Timers* [1]. Dividing available timers into multiple classes, each suitable for specific application states, can in particular prove highly beneficial when developing a timer subsystem.

A different approach to real-time task scheduling optimizations is *SLOTH* [8], including its derivatives *SLEEPY SLOTH* [10] and *SLOTH ON TIME* [9]. Whereas the first two target event-driven real-time systems, the last is designed for time-triggered OSes. All the *SLOTH*-based approaches feature techniques which can also partly be applied to generic timer drivers. The key idea behind the event-driven solutions is to move scheduling completely into the interrupt service routine (ISR) context, thereby reducing task latencies. The time-driven concept instead targets timer peripheral management. Here, the common multiplexing of timeouts onto a single hardware timer (e.g., periodic system tick timer) is enhanced by utilizing additional timer devices. This dispersion of software timers across multiple peripherals reduces both maintenance overhead and scheduling latency. We expect especially the latter technique to prove important for a high-level timer-API.

2.3 Summary

Most of the above presented research is not directly related to the hardware analysis we contribute with this work. However, the outlined techniques, key aspects, and common pitfalls, e.g., the usage of a single periodic timer tick [2, 5, 6], are of great value and will later be taken into account when designing the aspired timer subsystem. Still, commonly addressed peripheral characteristics can be derived from the above publications. They yield a multitude of criteria we therefore incorporated into the scope of our broad MCU-platform analysis. These include basic properties of timer peripherals [12, 17] such as timer type, counter register width, and prescaler availability. Further features, seen as mandatory by e.g., Lindgren et al. [13], like interrupt capability and auto-reload functionality were also incorporated into the scope of our analysis.

In addition, many software modules utilize the concept of multiplexing. Here, multiple software timers are mapped onto a small set of hardware timers and their respective compare channels. It allows maintaining more virtual timers than compare channels are available, as well as the separation of short and long delays [14]. This yields further analysis criteria such as the number of available compare channels or maximum resolution. Showcased timer subsystems that are designed with respect to power-saving [1, 4, 7]

[2]RIOT-OS *ztimer* pull-request and discussion on GitHub: https://github.com/RIOT-OS/RIOT/pull/11874 (Accessed 03.12.2019)
[3]TinyOS project website: http://tinyos.net/ (Accessed 30.01.2020)
[4]EPOS project website: https://epos.lisha.ufsc.br/ (Accessed 30.01.2020)
[5]RTFM code repository: https://github.com/rtfm-rs/cortex-m-rtfm (Accessed 30.01.2020)

moreover demand the analysis of features like low-power clock support or the concepts of how interrupts are handled by the MCU.

With a view on high-level timer driver design, many of the concepts highlighted above can be applied to our use-case. We expect promising results especially from the usage of more than one hardware timer for multiplexing [9] combined with dividing timers into classes, based on their suitability in different usage scenarios [11]. Hierarchical timer chaining [14, 24] as well as a multi-layer timer-API architecture [7], keeping platform-independence while allowing to use platform-specific features, are further takeaways. We conclude that a low-level timer interface, as we propose it in Section 5, therefore must be capable of exposing available timer features appropriately to both the high-level driver module and the user application directly. This includes both basic features that are commonly found across all MCUs as well as advanced platform specific features.

3 HARDWARE-PLATFORM ANALYSIS

A major contribution of our work is the analysis of different timer peripherals, which are found in MCUs that are currently supported by RIOT-OS. In this section, we start by describing the scope of the analysis as well as the methodology we apply conducting it. Then, results, both specific to device families and also across all analyzed platforms, are presented and discussed with an outlook on the aspired timer subsystem. Furthermore, outstanding tasks and possible improvements are identified.

3.1 Scope

The following analysis covers all chip manufacturers that offer at least one microcontroller, supported by RIOT-OS at the time of writing. For each of those, all MCU families with RIOT-OS support were examined. This yields a total of 43 analyzed MCU families, produced by 8 different manufacturers. Some MCU families were combined during our analysis as they were found to share timer peripherals, such as with the STM32 device family. A detailed description of the analyzed criteria and properties as well as the applied methodology can be found in Section 3.2.

We analyzed the following MCU families:

- STMicroelectronics (ST)
  - STM32F0 / F1 / F2 / F3 / F4 / F7
  - STM32L0 / L1 / L2
- Microchip / Atmel
  - ATmega AVR
  - PIC32MX / PIC32MZ
  - SAMD21
  - SAM3A / N / S / U / X
- Espressif
  - ESP8266
  - ESP32
- Silicon Labs
  - EFM32 / EFR32
  - EZR32
- Texas Instruments (TI)
  - CC13x2 / CC26x2
  - CC2538
  - CC430
  - LM4F120
  - MSP430x1xx / MSP430x2xx
- NXP Semiconductors
  - Kinetis E / EA / K / L / M / V / W
  - LPC176x / LPC175x
  - LPC2387
- Nordic Semiconductor
  - nRF51x / nRF52x
- SiFive
  - FE310-Gx

3.2 Methodology

Each step of our hardware analysis is conceptually depicted in this section and appears in the order of execution.

3.2.1 Platform Selection and Information Acquisition. As a starting point, all CPUs that are currently supported by RIOT-OS, as listed in `/cpu`[6], were determined. Chip manufacturers and their respective MCU families were then prioritized and processed according to their estimated diversity in timer hardware. This prioritized approach was chosen to be able to compile a comprehensive list of analysis criteria early on. For each MCU family, documentation in the form of datasheets, reference manuals, application notes, and others were obtained from the respective manufacturers.

3.2.2 Definition of Analysis Criteria. After obtaining an initial overview, a first set of criteria and properties was defined. It includes all aspects we identified as mandatory and therefore are to be extracted from the gathered documents for every analyzed MCU platform. Selected characteristics derive from our review of related work, as summarized in Section 2.3, and were further extended according to their significance, as expected by us. They include basic properties of timer peripherals such as counter register width, prescaler configuration, compare match capabilities and auto-reload functionality. Furthermore, advanced aspects such as interrupt generation, timer chaining and low-power features were examined. A full list and the detailed definition of each criterion can be found in Section A.1.

3.2.3 Extraction of Timer Peripheral Details. All timer peripherals of each platform were analyzed and information found in the acquired documentation was transformed into a mind-map structure. Properties and implications beyond our defined criteria were recorded nonetheless in order to be used in future work (see Section 6). If the MCU documentation was unclear at some point, additional information sources were used. These included peripheral register descriptions as well as SDKs provided by the respective manufacturers. However, if a concrete property could not be determined with confidence, it was marked as currently unknown.

3.2.4 Consolidation of Results. Since our data acquisition was not limited to the defined criteria, it yielded more information than the initially selected properties. We therefore adopted and extended our set of analysis criteria once more before consolidating final results. For each MCU platform a *Timer Comparison Matrix (TCM)* was created. A TCM lists all available timer types and their respective properties. It allows to quickly determine various characteristics

[6] See: https://github.com/RIOT-OS/RIOT/tree/master/cpu (Accessed 01.12.2019)

and features of each of the available timers and allows comparison across different MCU families and manufacturers. The TCMs were subsequently used as a foundation for our inter-MCU-platform findings. All created TCMs can be found in the appendix Section A.

3.2.5 Inter-MCU-platform Findings. To obtain general insights we evaluated various properties across all platforms and timer types. These properties could either be found directly within the TCMs or could indirectly be derived from them. If one timer type is available in multiple versions (e.g., 16-bit and 32-bit general-purpose timers) each version was evaluated and counted separately as an independent timer type. Platforms were counted whenever any of their available timer types matched the respective property and criterion. Unresolved or unclear timer properties were excluded from the respective results. Exclusion of specific peripherals, such as watchdog timers, is furthermore possible and accordingly denoted in the criterion description if applied.

Each analyzed property features a unique identifier used for referencing, a short title, and a description asserting the respective property. Moreover, a criterion can be specified, allowing to split the property into multiple cases (e.g., separating counters by available register width). All evaluated inter-platform properties are depicted in Table 1 and discussed in Section 3.3.

3.3 Results

Cross-platform findings from our MCU timer hardware analysis are listed in Table 1 and discussed in the following. It should be noted that the total number of platforms and timer types may vary between properties. This can either be due to the respectively applied selection criterion or due to timer type exclusions, as stated in the property description. To cope with this we give the exact matched amounts as well as percentages for each depict result. A detailed description of our data collection and evaluation methodology can be found in Sections 3.2.4 and 3.2.5.

3.3.1 Counter Range. A common property of timer peripherals is the width of their internal counter register. It dictates the maximum number of cycles a timer is able to count before an over- or underflow happens. The less frequent such events happen, the less timer maintenance and wake-ups are required. Therefore, a large counter width is desirable. It was found that the available counter register size among all platforms is at least 16 bit. A total of 90 % even provide 32-bit timers while only 21 % offer timers featuring a width of at least 64 bit (see `R-01`). MCUs that solely offer small timers particularly benefit from the possibility of extending counter range via timer chaining. Otherwise, frequent timer maintenance is mandatory when working with long timeouts. We found that 71 % of all platforms that offer one or more 16-bit timers allow extending these small timers to a range of at least 32 bit (see `R-04`). As less wake-ups and reduced maintenance overhead are desired, we conclude that timer chaining shall be utilized, especially when exclusively working with small range timers. It however must be noted that the usage of timer chaining comes at the expense of sacrificing one additional timer module for every range extension. For some applications this can be undesirable and therefore usage of timer chaining must remain optional.

Prescalers dynamically reduce the clock frequency that is fed into a timer, thereby counting only every n-th pulse of its base clock. They hereby allow to achieve longer timeouts without requiring intermediate maintenance wake-ups. This advantage comes at the cost of a lowered timer resolution, due to the reduced base clock frequency. Prescalers are nonetheless useful, especially when dealing with small counter widths ($\leq$ 16 bit) and long timeout periods. This trade-off indicates that a separation of short high-precision delays and long-lasting timeouts is desirable for an optimized timer driver in order to perform well in diverse application scenarios.

Prescalers are available on all platforms in general as well as on 75 % of all analyzed individual timer peripherals (see `R-03`). The only platform that has non-prescalable general-purpose timers is the SiFive FE310-Gx (see Table 22), which instead features a 64-bit counter register and thereby eliminates the need for an additional prescaler.

3.3.2 Auto-reload. Timers often need to produce either periodic events or timeouts that are longer than the maximum time before a counter over- or underflow happens. Hence, restarting the timer is required. To prevent the missing of clock pulses and to reduce maintenance overhead, reloading is handled directly by the timer hardware via the auto-reload feature. Hereby the counter register is set to either a fixed or configurable value once a designated event happens. All applicable timers support a form of auto-reload, as indicated by `R-08`. It was further found that 17 % of all timer modules only allow auto-reloading at over- or underflow events while others allow to specify an arbitrary value at which the counter reloads. The latter is either achieved by sacrificing one compare channel (32 % of all timers) or through the usage of a designated auto-reload register (51 % of all timers). Using a separate auto-reload register benefits a timer subsystem by keeping all compare channels available for timeouts (see Section 3.3.3). We therefore argue that using a separate register is the preferred approach.

3.3.3 Compare Channels. Timers can trigger events at specific counter values based on the configuration of their compare channels. Each compare channel can be armed to a specific value that gets continuously compared to the current value of the counter register. This operation is performed directly by the timer hardware, thus no polling or active waiting is required. A match event is generated once the counter value reaches the configured threshold. Compare channels are used by timer subsystems to signal expiring timeouts. The more compare channels a hardware timer offers, the more freely different pending timeouts can be split across channels (e.g., separating short from long-running timers). This flexibility benefits the mapping of virtual timers to hardware peripherals, hereby reducing the overall timer maintenance overhead.

At a bare minimum, a hardware timer is required to provide at least one compare channel to be suitable for usage by an optimized timing subsystem. Otherwise, active polling of the current counter value would be required, thereby effectively occupying the CPU with timer maintenance tasks. Our analysis showed that all timer modules provide at least one compare channel, while most offer at least two (64 %) or even four (24 %) channels (see `R-02`). We therefore conclude that all MCUs within our scope meet this base requirement for the aspired timer subsystem.

ID	Title	Description	Criterion	Platforms [#]	Timer Types [#]	Platforms [%]	Timer Types [%]
R-01	Counter width	Usable size of the counter register in bits (Excluding watchdog timers)	≥ 16	19	83	100 %	87 %
			≥ 32	17	32	90 %	34 %
			≥ 64	4	4	21 %	4 %
R-02	Compare channels	Number of available compare channels (Excluding timers w/o compare channels)	≥ 1	19	80	100 %	100 %
			≥ 2	14	51	74 %	64 %
			≥ 4	10	19	53 %	24 %
R-03	Prescaler	Support for prescaling the timer clock	*yes*	19	87	100 %	74 %
R-04	Timer chaining	Support for timer module combination (Excluding watchdogs and RTCs)	R-01 ≤ 16†	10	15	71 %	38 %
			R-01 > 16‡	4	5	27 %	16 %
R-05	Compare interrupts	Unique INTs for each compare channel	*yes*	11	28	58 %	31 %
R-06	Overflow interrupts	Unique INTs for counter over-/underflow (Excluding watchdogs)	*yes*	8	13	42 %	19 %
R-07	Event flags	Availability of status bits for timer events	*yes*	16*	100	100 %	100 %
R-08	Auto-reload	Auto-reload at over/-underflow (OVF), at compare-channel match (CCM), or via auto-reload register (ARR) (Excluding watchdogs and RTCs)	OVF	3	14	16 %	17 %
			CCM	6	25	32 %	32 %
			ARR	10	40	53 %	51 %
			any	19	79	100 %	100 %
R-09	Clock sources	Number of available clock sources (Distinct external and internal clocks)	≥ 1	19	117	100 %	100 %
			≥ 2	16	59	84 %	50 %
			≥ 4	6	20	32 %	17 %
R-10	Internal clock sources	Number of available internal clock sources	≥ 1	18	110	95 %	92 %
			≥ 2	15	40	79 %	33 %
			≥ 4	1	2	5 %	2 %
R-11	External clock sources	Number of available external clock sources	≥ 1	19	114	100 %	95 %
			≥ 2	13	46	68 %	38 %
			≥ 4	3	7	16 %	6 %
R-12	Low-power clock	Low-power oscillator can be used by timer	*yes*	19	84	100 %	71 %
R-13	Deep-sleep active	Timer operational in lowest MCU power states	*yes*	19	68	100 %	57 %
R-14	GP-timers	Number of available general-purpose timers	= 1	1	-	5 %	-
			≥ 1	18	-	95 %	-
R-15	WDT interrupts	Watchdog generates interrupt prior to reset	*yes*	13	14	68 %	67 %
R-16	Unknown items	Timer has unresolved/unknown properties	*yes*	6	17	32 %	14 %

* Three platforms excluded due to unknown properties. See Section 3.3.4 for details.
† i.e.: Only counting timers that are chainable and have a maximum width of 16 bit. See Section 3.3.1 for details.
‡ i.e.: Only counting timers that are chainable and have a width greater than 16 bit. See Section 3.3.1 for details.

Table 1: Selected results across all 19 analyzed MCU platforms. Each result is evaluated for timer types as well as associated MCU platforms. It is characterized by the result description and may be split into multiple cases, as denoted by the specified criterion. If specific timer types were excluded it is indicated in the result description. See Sections 3.2.5 and 3.3 for details.

3.3.4 Interrupt Handling and Event Flags. Timer events such as over- or underflow and compare matches may generate interrupts upon occurrence. Corresponding interrupt service routines (ISRs) can then be used by timer drivers to detect expired timeouts and to execute maintenance tasks. Interrupt handling strongly depends on the actual MCU-platform, though we found it to be typically implemented uniform among chip families from the same manufacturer. For a timer subsystem it is important, whether an exclusive interrupt for every single event exists or if multiple events share one common interrupt vector. The former is the preferred method

as it does not require manually resolving the interrupt cause upon occurrence.

Our analysis revealed that out of all timers only 19 % provide fully independent overflow and only 31 % offer fully independent compare match interrupts (see R-05 and R-06). Additional peripheral status register reads are required in order to determine the exact cause of the fired interrupt if an interrupt vector is mapped to multiple events. This introduces a layer of indirection and therefore increases timer latency. Flags that indicate event occurrence are available throughout all platforms, as shown by R-07. Though, it is noteworthy that three platforms, namely Espressif ESP8266 (see Table 9), Espressif ESP32 (see Table 10), and Nordic Semiconductor nRF51x/52x (see Table 21), do not state the availability of event status bits in their documentation (see Section 3.4). These MCU families were therefore excluded from the total platform count. Nonetheless, we highly doubt that there is no way to extract such information but were not able to reliably confirm it either.

3.3.5 Clock sources. The selected clock source not only affects the frequency and resolution of a timer. It can also have an effect on specific timer features or limit the available operation modes. Clock sources can be categorized into *internal* and *external* clocks. The first are generated solely within the MCU and therefore are always available. The latter are either supplied via an external signal that is applied to a specific input pin or generated internally with the help of some external hardware that needs to be connected to the MCU, such as a crystal oscillator.

We observed that 95 % of all timers can be driven by at least one internal and 92 % of all timers by at least one external clock (see R-10 and R-11). It was furthermore found that on 84 % of all platforms multiple timer clock sources are configurable, as indicated by R-09. This yields numerous clock configuration options and therefore indicates that clock configuration should be considered an important aspect when designing a timer-API. Power-constrained devices benefit particularly from the run-time reconfiguration of timer clock sources since it supports dynamic power mode transitions and allows the utilization of special purpose low-power oscillators. As this applications entail a large set of specific requirements, implications for low-power operation and further energy saving considerations are discussed separately in Section 3.3.6.

3.3.6 Low-power Operation and Energy Saving. As with power-constrained devices low-power operation is crucial, the ability to operate timers of a low-power oscillator is highly important. As R-12 shows, 71 % of all analyzed timer types are able to run on such a low-power clock. This enables timer drivers to make use of the available MCU power-saving modes, for example powering down the CPU and main peripheral clock while keeping the required timers operational. Properly utilizing this feature is of utmost importance when designing a timer subsystem for OSes like RIOT-OS.

We found that all platforms provide at least one timer type that can run of a low-power clock. The analysis furthermore confirmed that all platforms offer at least one timer that is capable of both operating in even the lowest possible power states and waking the CPU upon event occurrences (see R-13). We refer to timers that meet these criteria as *always-on* peripherals. Among these, real-time-counters and -clocks are most commonly found. Five platforms also provide designated ultra low-power timer peripherals (see Tables 5, 6, 11, 14, and 18), as for example the Cryotimer on the Silicon Labs EFM32/EFR32 platform (see Table 11). Especially when dealing with long timeouts and deep-sleep periods, these timer types allow significant energy-saving optimizations and therefore must be made available via a well-designed timer subsystem.

3.3.7 Suitability of Timer Types. Even though we covered all types of timer peripherals within our analysis, we are convinced that not every type is applicable for the use in a generic timing system. We contend that especially watchdog timers fall into this category. Watchdogs are primarily designed to recover a system from a malfunction or error state. They achieve this through performing a full system reset if not periodically serviced by the application. Our analysis showed that 67 % of the analyzed watchdogs offer the ability to generate an interrupt before or even instead of performing a reset (see R-15), hereby making them theoretically capable of generating generic timeouts. These special purpose timers however usually are limited to a very basic feature set and often behave different across MCU platforms. They also are commonly already occupied by other components of the application. We therefore suggest to not repurpose them for the usage in a timer subsystem.

3.3.8 Peripheral Availability. Having a manifold range of timer peripherals to chose from opens up a wide spectrum of optimization opportunities. We found the feature sets of general-purpose timers to be largely uniform across all platforms, whereas special purpose timers differed greatly with respect to their offered functions and modes of operation. Taking a look at the first, there is only a single platform that guarantees the availability of just one general-purpose timer, namely the SiFive FE310-Gx (see Table 22). All other platforms provide more than one generic timer (see R-14). However, the availability of other timer types varies greatly between manufacturers and even among MCU families of the same manufacturer. We conclude that both, properly utilizing multiple available timer modules and the incorporation of platform specific peripherals, is a necessity for a well-designed generic timer subsystem.

We also encountered MCUs that only leave the possibility of multiplexing all virtual software timers onto one single general-purpose hardware timer (see R-14 and R-02). These namely are the SiFive FE310-Gx (see Table 22), with only a single general-purpose timer, and the Espressif ESP8266 (see Table 9), with only one compare channel while leaving the alarm functionality of the RTC undocumented. We therefore argue that a timer subsystem must be flexible enough to cope with situations in which only a single hardware timer is available.

3.3.9 Missing Information. Not always could every property of the analyzed timer types be determined with sufficient confidence, based on the available documentation or alternative information sources, as described in Section 3.2.1. A total of six platforms still suffers such unresolved properties, as indicated by R-16. Five of these platforms can more precisely be grouped into the two Espressif MCUs (see Table 9 and 10) and three of the Cortex-M based platforms, namely the Microchip / Atmel SAMD21 (see Table 8) and the Silicon Labs MCUs (see Table 11 and 12). The documentation of Espressif devices, especially the ESP8266, remains unclear about many of our analysis criteria, of which only some could be resolved

by inspecting the manufacturer provided SDKs. In contrast, the mentioned Cortex-M based MCUs only leave the SysTick timer, commonly found across Cortex-M devices, mostly or completely undocumented. We suspect them to be very similar to those found on other Cortex-M based devices but are unable to confirm it at the time of writing. Lastly the documentation of the Nordic Semiconductor nRF51x/52x MCUs (see Table 21) remains unclear about the availability of event flags.

3.4 Outstanding Tasks & Open Issues

Even though our timer hardware analysis already yielded insight into a broad range of properties and features, various outstanding tasks as well as some issues remain. These are, to the best of our knowledge, highlighted below.

3.4.1 Unresolved Properties. During our analysis we were able to gather information regarding the defined criteria for nearly all timer types within our scope, though some properties remained unknown, as described in Section 3.3.9. When resolving these properties the required effort must, however, be justifiable when compared to the estimated information gain and impact on the overall results.

3.4.2 Timer Resolution and Clock Tree Properties. A timers resolution, i.e., the shortest possible timeout, and the maximum timeout length it can achieve without requiring intermediate maintanance wake-ups both are base characteristics of every hardware timer. A timer subsystem can use these information to allocate requested timeouts to the most appropriate peripherals available. Determining these properties generically for a whole class of timers however is not feasible due to their strong linkage to MCU oscillator frequencies. As the system clocks depend on both the microcontroller and its configuration, their actual operation frequencies can vary largely, therefore preventing the calculation of a single appropriate value that can be used for comparisons across platforms.

Another aspect we do not have full insight into yet are the clocks each timer peripheral is able to run of. Since each MCU-platform provides different oscillators and methods of routing the generated clock signals to peripherals, a unified and comparable way of analyzing these has to be defined. As this clock tree analysis is an entire complex topic on its own, it was excluded from this hardware analysis for now. We nonetheless expect promising results from it, especially with respect to power-saving optimizations, and therefore suggest to conduct an in-depth analysis of the clock-trees for future work.

3.4.3 Peripheral Interconnect and Event Systems. Some MCUs are capable of routing various internal signals directly between components via a designated peripheral interconnect bus. Others allow to execute special hardware tasks based on specific events, generated by other peripherals. Both methods benefit the overall system performance by removing the need to execute a designated ISR on the CPU upon occurrence. Such peripheral interconnect and event systems might prove valuable for some applications (e.g., timer chaining). Furthermore, a reduction of maintenance tasks could be possible by exploiting event systems in order to execute simple maintenance tasks autonomously on the respective peripherals.

3.4.4 Configuration and Maintenance Costs. Every read, update, or reconfiguration requires resources, such as CPU time or battery power. Information about the resource needs of such operations, however, are only rarely available. Timer operations might also cause side effects that need to be taken into account. For example, it can be required to enable a high-power clock in order to read or write registers of a timer which is running of a low-power clock. As a result, frequent maintenance tasks drastically increase power-consumption as the high-power oscillator is started during every maintenance period. We therefore consider information on such costs beneficial to the timer peripheral selection process.

4 RIOT-OS LOW-LEVEL TIMER MODULES

In this section, low-level timer modules, as currently found in RIOT-OS, are depicted. Common design aspects are then outlined and potential API enhancements are identified.

4.1 Overview

Currently, several low-level timer abstractions exist in RIOT-OS, each aimed at a specific class of hardware timers. We looked into these modules and compared their respective properties. At the time of writing a total of five such modules exist, out of which the first three are designed to handle common timekeeping applications, while the latter two are used for more specialized tasks. Properties and use-cases of each module are outlined in this section and summarized in Table 2.

4.1.1 General-purpose Timer Module. The `periph_timer` module drives the various general-purpose timers of an MCU. It is primarily used as a generic interface to provide short timeouts with a high resolution. Multiple general-purpose timers, each coming with multiple compare channels, can be used via this module. The counter values are exposed as raw counter ticks and specification of a desired counting frequency is supported during initialization.

Basic common timer features are made available via the uniform function interface. These range from reading and writing the counter register value to arming timer channels in order to generate events at specific counter values. A user-defined ISR that is executed upon any compare match event can furthermore be attached during timer initialization. More advanced configurations such as counting mode, auto-reload, and, clock selection, however, are highly platform dependent and compile-time static.

4.1.2 Real-time Clock Driver. Real-time clock peripherals can be accessed by the `periph_rtc` module. Time is represented as both a wall-clock time struct (seconds, minutes, hours, days, ...) and an additional sub-second counter with microsecond resolution. Functions for convenient time conversion and comparison are furthermore part of the module.

Reading and writing the RTC time registers as well as explicitly powering up or down the hardware module is supported by the API. The chosen RTC peripheral is hard-coded, hereby eliminating the need to specify a peripheral identifier during API calls. At the same time this sacrifices the option to address multiple RTC modules, as available on some MCUs. This restriction also applies to alarm channels, out of which only one is exposed to the user and allows attachment of a callback function.

	`periph_timer`	`periph_rtc`	`periph_rtt`	`periph_pwm`	`periph_wdt`
Timer types	General-purpose	RTC, (RTT)	RTT, low-power	Various	Watchdog timers
Time unit	Counter ticks	Wall-clock time struct	Counter ticks, relative time struct	N/A	Counter ticks
Timeouts	short-running, high resolution	long-running, low resolution	long-running, low resolution	N/A	short-running, high resolution
Multiple timers usable (I)	✓	×	×	✓	×
Multiple channels / alarms usable (II)	✓	×	×	✓	×
INTs / Callbacks	One combined callback per timer for all compare match INTs. No overflow callback.	One alarm, additional left unusable. No overflow callback.	One alarm, additional left unusable. Overflow callback attachable.	×	WDT warning INT callback attachable if supported.
Low-power operation (III)	×	✓	✓	×	✓
Power up/down support (IV)	×	✓	✓	✓	×
Timer capabilities indicated (V)	×	×	×	×	✓
Peripheral allocation conflicts (VI)	✓	✓	✓	✓	×
Timer types heterogeneous (VII)	×	✓	✓	×	×

(I) Module is able to interface multiple timer instances, distinguished by an appropriate timer instance identifier.
(II) Module is able to interface multiple compare channels or RTC alarms per timer instance, distinguished by an appropriate channel identifier.
(III) Module is primarily used to drive timer types that offer low-power operation modes and features.
(IV) Explicitly powering up or down a timer instance is supported by the module. This is different from initialization and start or stop operations.
(V) Static properties and capabilities of the timer hardware (e.g., feature support, number of compare channels, …) are made available via the module.
(VI) Driven timer types are also used by other low-level modules, hereby creating a resource allocation conflict between involved modules. For example: Using TIM1 for PWM generation via `periph_pwm` must result in removal of TIM1 from the set of timers driven by `periph_timer`.
(VII) Module is used to drive other timer types than initially intended (e.g., on STM32: using a "Low-power Timer" instead of an "Real-time Timer" with `periph_rtt`).

Table 2: Comparison of current low-level timer modules in RIOT-OS

4.1.3 Real-time Timer Driver. The real-time timer driver `periph_rtt` is a mixture of the general-purpose and RTC modules. It typically is used to drive real-time timer peripherals but sometimes it is repurposed to expose low-power timers that provide long-running timeouts with a low resolution. Elapsed time can be represented as raw counter ticks or as relative time structs (conversion macros provided) and can be read and written. Instead of providing functions to start and stop the hardware counter, functions powering the whole timer module up and down are given.

Similar to the RTC driver, the RTT driver does not distinguish between timers. It likewise only supports a single hard-coded RTC peripheral. Equally, only a single alarm channel is exposed but an exclusive overflow callback can be attached during run-time.

4.1.4 PWM Driver. Generation of pulse-width modulation (PWM) signals often is closely linked to timer peripherals. For this task the `periph_pwm` module can both utilize hardware timers that feature PWM capabilities as well as designated PWM modules. Since this module is focused on output signal generation instead of sole counting tasks, raw values of the internal counter register are left unexposed. It therefore is not possible to determine the current position within the period of the generated signal.

Both the frequency and resolution of the generated waveform can be specified via the API, hereby impacting the pursued duty cycles. Multiple PWM channels, each featuring its own duty cycle, are supported and distinguished by a unique identifier. Powering up and down single PWM channels is not supported. Once the module is enabled, all channels are always active.

4.1.5 Watchdog Timer Driver. Watchdog timers are exposed via the `periph_wdt` module. As with PWM peripherals, these can neither be used for simple counting tasks nor to generate application timeouts. Nonetheless, they still are seen as timer peripherals on

most MCUs but provide only a very limited set of features that is fully exposed by the `periph_wdt` module.

Support for both normal and window operation mode is provided and a user-defined callback function can be executed upon a watchdog timeout. Availability of these features is indicated by respective preprocessor defines for each MCU platform. It is not possible to distinguish between multiple available watchdogs, as the peripheral mapping is hard-coded within the implementation.

4.2 Analysis

Examining the presented low-level timer modules, common design aspects can be identified. These are outlined in the following section and additionally depicted in Table 2.

With the current solution to peripheral handling in RIOT-OS, multiple drivers for timer hardware exist, each targeted at a specific set of timer types. These modules differ in the way counter values are represented and accessed, which events allow to attach a user-defined ISR, and how timer instances or channels are addressed. Handling of low-power specific tasks like switching the power state of a whole timer module is supported by some of the drivers, namely `periph_rtc` and `periph_rtt`. Provided functions are limited to the minimal set of features that is common to all timer peripherals of the respective class. Implementation of advanced features is hereby left to the individual application developer.

Although we found that the offered APIs differ in their usage, their functionality often overlaps. Having multiple low-level timer drivers allows to tailor the functional interfaces to specific timer types but comes at the cost of loosing interoperability between these. Another challenge with this approach is that either not all available timer types can be exposed due to missing driver modules or developers need to expose different timer types via the same, potentially unfavorable, API. This leads to features or even whole timer types being left unusable, as for example the ”Basic Timers” and the ”SysTick Timer” on the STM32 platform.

The ability to distinguish between timer instances was also found to differ between modules. Whereas `periph_timer` does support it, `periph_rtc` and `periph_rtt` do not, hereby leaving many timer instances unusable, as for example additional low-power timers on the STM32 platform. This problem also extends to the compare and alarm channels, hereby effectively limiting the number of usable timeouts to one per driver module.

Last we found that the management of timer configuration is highly platform dependent and differs largely across implementations. This includes, but is not limited to, the configuration of available peripherals and clock sources as well as module specific configurations like counting mode, auto-reload values, and counting directions. Programmatic determination of available timer features and capabilities is furthermore only supported by the `periph_wdt` driver. It is noteworthy that with RIOT-OS's current approach this is not yet problematic, due to the fact that only the minimal set of common timer features is exposed anyway.

5 LOW-LEVEL TIMER-API DESIGN

A primary objective of this work is to design and implement a clean-slate low-level timer-API for RIOT-OS. Within this section, we start by outlining generic demands on such a low-level timer subsystem. Subsequently, the design of an API that fulfills these demands is proposed and discussed in detail.

The contributed API design is based on insights from our timer hardware analysis (see Section 3), related work (see Section 2), as well as common pitfalls and problems that are present with current solutions (see Section 4). Our goal is not to fully replace existing modules but to provide a possibility to access timer hardware in a uniform and platform-independent way. The proposed design therefore exposes advanced timer features that might not be available on every single MCU, but are still frequently required by applications. Access to such features is kept lightweight and straightforward while preserving platform-independence whenever possible.

5.1 Generic API Requirements

RIOT-OS positions itself as a general purpose IoT operating system that features a wide-ranging hardware support while maintaining program portability whenever possible. Use-cases and application scenarios are vast and often bring their individual set of requirements. When it comes to timekeeping on resource restricted devices, however, ease of use, a small ROM and RAM footprint, as well as optimized low-power operation are essential for most use case scenarios. A low-level timer-API should therefore address these aspects and support the application developer. In this section, demands on such an API are outlined. They are presented contextually grouped, as indicated by the chapter titles.

5.1.1 Scope and Usability. Providing a simple but comprehensive feature set is quintessential. A low-level timer-API should therefore

(1) be intuitively and straightforwardly usable by the application developer.
(2) allow various different timer types such as general-purpose, low-power, RTT, or RTC to be used interchangeably via the same unified API.
(3) provide well-designed driver code for timer features, hereby relieving the application developer from the errorprone task of writing low-level driver code. This includes
    a) exposure of basic timer features to both the application developer and high-level RIOT-OS modules (e.g., `xtimer`).
    b) exposure of special timer features to meet advanced requirements such as optimized low-power operation. If available, such features should be implemented by the driver developer whenever possible with reasonable effort.
(4) expose identical features in a standardized way among all timers that support it. This allows interchangeably using timers independent of their exact type, as long as they offer the required functionality.
(5) provide access to multiple hardware timer instances to
    a) allow developers and applications themselves to dynamically choose from the full set of available peripherals.
    b) allow using them inside other drivers and modules, such as MAC layers, external peripherals, or signal generation.
    c) optionally allow managing peripheral allocation by some form of resource allocator, providing mutually exclusive timer usage or defining more complex application requirements like ”requiring at least *n* low-level timers”.
(6) allow representation of multiple hardware timers as a single timer instance. Specifically allowing the representation of

chained smaller timers as a single large timer that is usable as any other timer instance.
(7) encourage the driver developer to follow a common pattern for platform dependent code in order to obtain maintainability and guide integration of future MCUs. Similar functionality therefore should be implemented in the same code segments across drivers for different MCUs.

5.1.2 State and Capabilities. As applications require a variety of timer related information, a low-level timer-API should

(1) differentiate between the following types of information and properties:
   a) *static properties* of the underlying timer hardware, such as counter register width or available channels.
   b) *compile-time static properties* that can only be changed prior to compilation and therefore remain fixed during run-time, such as available timer drivers and modules.
   c) *dynamic run-time properties* that may change during run-time, such as counting mode or pending interrupts.
(2) provide static information about the capabilities of hardware timers, hereby allowing high-level modules to make platform independent use of advanced timer features.
(3) expose dynamic timer status information during run-time, such as the current mode of operation and pending interrupts. This eases timer operation in different contexts like an ISR, a normal thread, or any other context where interrupts are disabled. High-level modules that depend on this information are thereby relieved from using workarounds such as manually detecting counter register overflows.
(4) provide the ability to attach a callback function to both *compare match* and *overflow* interrupts, whenever available. The exact interrupt cause should be easily identifiable.

5.1.3 Platform Portability. To maintain cross-platform application portability a low-level timer-API should

(1) address timer peripherals in a standardized way across all platforms, i.e., a generic type for timer instances is defined. A timer instance on one platform shall therefore be presented the same way as a timer instance on another platform.
(2) maintain platform independence as far as possible and only be platform specific whenever inevitable. It should therefore aim to maintain cross-platform portability as long as required timer features are available on all targeted MCU platforms.
(3) give the application developer the ability to consciously sacrifice cross-platform portability whenever highly platform specific features are required for the application. In other words: Portability should never sacrifice features.

5.1.4 Configuration Management. Timer peripherals are managed by the timer subsystem. A low-level timer-API should therefore

(1) allow to select timer instances during compile-time that are then exposed to the application at run-time.
(2) allow dynamic run-time re-configuration of hardware timers whenever feasible. In particular run-time clock configuration to complement the usage of advanced low-power MCU operation modes is desirable.
(3) support cross-platform parameter configuration, similar to other OS and driver settings. If one timer hardware parameter applies to a *relevant* number of platforms, it should be configurable globally for all affected platforms.
(4) encourage a common pattern for peripheral configuration management. It should guide developers on where to store information and what should be configured on which layer.

5.1.5 System Impact and Resources. As timers are always embedded within a broader application context, efficient use of system resources is crucial. A low-level timer-API should therefore

(1) be modular in a way that functions and code not required for the current application can be excluded from the built binary, thereby reducing the overall memory footprint.
(2) not interfere with other system modules nor introduce side effects, whenever avoidable.
(3) use system resources efficiently by being both run-time and memory efficient (ROM and RAM). Maintenance overhead should be kept as low as possible.

5.2 Architectural Overview

While timer types differ in their specific set of features, many MCU platforms provide timers that are similar with respect to their basic operating principles. Even though implementations of their features might differ slightly, their underlying logic and semantics are alike. The following unified API design fosters a transparent and interchangeable usage of "all" timer peripherals, at least regarding their common timer operations.

The proposed low-level timer API is split into two: The hardware-facing API (hAPI), see Section 5.4, and the user-facing API (uAPI), see Section 5.5. The hAPI is made up of minimal function sets (i.e., drivers) for each timer type that are used to interact with the actual hardware peripherals. The hAPI is used by the uAPI, which provides convenient timer access to user applications and high-level modules. It consists of convenience and compound functions that are independent of the underlying timer type. The described separation of hardware-specific driver code and hardware-independent user functions is depict in Figure 1.

Each individual timer is represented by an associated struct. It identifies the exact hardware peripheral, specifies the timer type driver and provides static information about the timer instance (e.g., width and channel count). A detailed description can be found in the following sections.

5.3 Timer Types and Instances

One single hardware timer is represented by an instance of the `tim_periph_t` struct. For every exposed timer, as configured during compile-time, an instance of this struct needs to be created. A `tim_periph_t` struct (see Listing 1) consists of the following data:

- Hardware peripheral identifier `tim_t`
- Associated timer type driver `tim_driver_t`
- Static timer properties, such as counter width and number of available channels

Type specific sets of timer functions `tim_driver_t` are assigned to each `tim_periph_t`, depending on the respective timer type (e.g.,

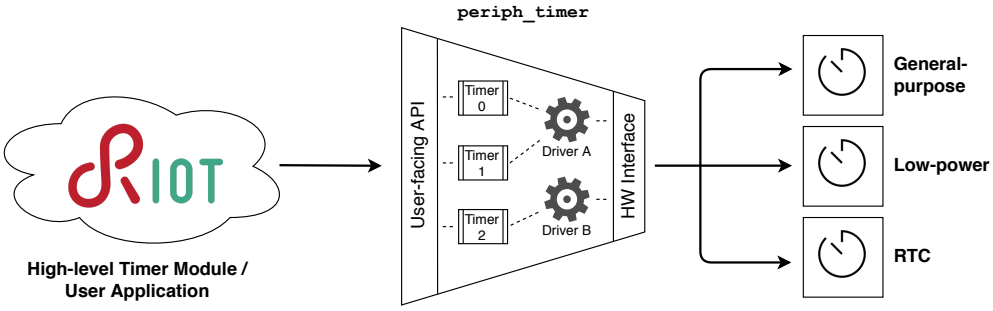


Figure 1: Overview of the proposed low-level timer-API design

```
typedef struct {
    const tim_t dev;
    const tim_driver_t *const driver;

    const uint16_t width    :8;
    const uint16_t channels :4;
    // ...
} tim_periph_t;
```

Listing 1: Definition of the `tim_periph_t` struct. It maps drivers to exposed hardware timers and provides information about static timer properties.

general-purpose, RTC, low-power, ...). Contents of the `tim_driver_t` struct are described in the following hAPI section. This flexible design allows the usage of different timer types through a unified user-facing API. Each type may be operated by (partly) individual driver code.

5.4 Hardware-facing API

The hardware-facing API (hAPI) is responsible for directly interacting with the registers of a hardware timer. It consists of minimal function sets, each represented as a group of function pointers within a designated `tim_driver_t` struct (see Listing 2).

```
typedef struct {
    int (*init)(/* ... */);
    tim_propval_t (*get_property)(/* ... */);
    int (*set_property)(/* ... */);
    int (*enable)(/* ... */);
    tim_cnt_t (*read)(/* ... */);
    void (*write)(/* ... */);
    int (*set_channel)(/* ... */);
    // ...
} tim_driver_t;
```

Listing 2: Definition of the `tim_driver_t` struct. One such struct exists for every timer type and contains pointers to its respectively implemented hAPI functions.

For every exposed timer type exactly one such driver is created and later mapped onto `tim_periph_t` instances of the corresponding peripheral type. Different driver code can hereby be provided for all timer types by assigning the desired function pointers within the driver struct.

5.4.1 Driver Code Reusability. MCUs commonly provide multiple instances of a single hardware timer type. Interfacing all timers of one class with the same generic driver not only reduces the memory footprint but also benefits code quality and maintainability. We refer to using one driver for multiple timer instances as *driver granular code reusability*. An example of this is shown in Figure 2a. Here, *Timer 0* and *Timer 1* are interfaced using *Driver A* while *Timer 2* uses a different *Driver B*. This behavior can be achieved by simply referencing the desired driver `tim_driver_t` within each timer instance `tim_periph_t`.

Our hardware analysis further revealed, that some timer types are available in two versions, e.g., a basic and an advanced general-purpose timer module. If these versions only differ slightly, it is worthwhile to reuse functions from one driver within another driver. We refer to using the exact same function within multiple drivers as *function granular code reusability*. The scenario depict in Figure 2b illustrates such a case. Here, *Driver A* and *Driver B* each possess exclusive functions (*fn_A* and *fn_B*) while also sharing common functions (*fn_1* and *fn_2*). To efficiently handle such scenarios function granular reusability of driver code is made possible. It can easily be achieved by assigning the same function pointer in both driver instances.

5.4.2 Timer Feature and Property Access. Every hardware timer offers a distinct set of properties. Each property can be put into either of the following groups:

(a) *Static attributes* such as the counter register width and the number of available channels.
(b) *Dynamic properties* such as the counting mode or pending events, e.g., compare match and overflow.

These properties are made available to the user application and high-level modules. Common static attributes, which apply to all timer types, are encoded into appropriate bit fields within the `tim_periph_t` timer instance structs (see Listing 1). This allows for easy determination of such static attributes as, e.g., the number of available channels. Properties that are either timer type specific or dynamic, i.e., can change during run-time, are exposed through a slim and straight forward interface (see Listing 3). All available properties are encoded within the `tim_prop_t` enum. They can be read using the `get_property(tim_prop_t)` and written using the `set_property(tim_prop_t, tim_propval_t)` function.

```
typedef enum {
    // ...
    TIM_PROP_MODE    = 0x01,
    TIM_PROP_CNT_DIR = 0x02,
    // ...
    TIM_PROP_OVF_PENDING        = 0xF0,
    TIM_PROP_CMP_MATCH_PENDING  = 0xF1,
    // ...
} tim_prop_t;

typedef struct {
    // ...
    tim_propval_t (*get_property)(tim_prop_t prop);
    int (*set_property)(tim_prop_t prop, tim_propval_t val);
    // ...
} tim_driver_t;
```

Listing 3: Dynamic property interface provided by the hAPI. Available properties are encoded within the `tim_prop_t` enum and can be accessed using the shown driver calls.

Advanced timer type specific features, which cannot commonly be found on all timers, are made accessible through the same flexible interface. These features can likewise be reconfigured using the `set_property()` function and their current configuration can be determined using the `get_property()` function. Examples are special counting modes, low-power features and hardware timer chaining. Such again are represented as separate entities within the `tim_prop_t` enum. If a given hardware timer does not support one of these specific features it simply can be left unimplemented. In this case, the platform developer may indicate the absence of this specific feature via the return values of the access functions.

5.4.3 Memory Footprint. Every available driver requires memory for storing the pointers to each function it provides. The fewer individual functions are provided, the less memory is consumed for associated function pointers. In order to keep the memory footprint low, the hAPI interface therefore is kept as compact as possible. This is accomplished by

(1) merging strongly coupled functionality into a single function. For example:
   a) `start()` and `stop()` are combined into a single `enable(run)` function.
   b) `set()`, `set_periodic()` and `clear()` are combined into a single `set_channel(tim_chan_mode_t)` function.
(2) moving all functions that can be implemented as a sole combination of other hAPI functions (i.e., compound functions) to the uAPI. Relative timer channel arming, for example, can be implemented by calling the hAPI functions `read()` and `set_channel()` and therefore is implemented solely within the uAPI as the `timer_set()` call (see Figure 3).
(3) exposing dynamic run-time properties and advanced timer functions through a getter-setter-based interface (see Section 5.4.2). A large amount of potentially unused function pointers can hereby be eliminated.
(4) grouping functions for specific features, such as PWM or timer chaining, into compile-time optional modules. Each module can be enabled by respective preprocessor directives.

A further reduction of memory consumption is achieved by strictly combining data into bit fields whenever appropriate, as for example done with static timer properties attached to `tim_periph_t` structs.

5.4.4 Virtual Drivers. In some use-cases it can be desirable to additionally provide *virtual timer drivers*. These drivers may not interact with a single hardware timer directly but instead can use other *base drivers* to interact with the required timers. They both can solely rely on base driver calls but also are able to provide additional implementations for specific functions. If required, base driver functions can further be overwritten by the virtual driver.

A common use-case is with hardware-supported timer chaining, as illustrated in Figure 2c. Here, *Timer 0* and *Timer 1* can be configured to operate as a combined timer module with an extended counter register size.

Both timers can be initialized and controlled using their base driver. But when chaining them into a combined module, additional configuration and a special behavior during reads and writes is mandatory. The virtual driver handles those two requirements by selectively extending the base driver functionality. It exposes the two small timers as a single `tim_periph_t` instance, featuring the enlarged counter value. User applications or high-level modules can now interact with the larger chained timer through the same uAPI as with any other `tim_periph_t` instance. It should, however, be noted that when exposing multiple hardware timers as a single timer instance it might be required to remove the base timers from the set of timer peripherals that is made available to the application. Otherwise accessing one of the base timers might cause unexpected behavior of the combined timer instance.

5.5 User-facing API

The user-facing API (uAPI) provides an interface to both the user application and high-level RIOT-OS modules that is independent of underlying timer types (see Table 3). It automatically delegates function calls to the respective hAPI driver, as specified within the `tim_periph_t` instance. It furthermore provides additional functions that are not required to be implemented within the hAPI, such as relative timeouts. In contrast to the hAPI, which provides multiple sets of functions that are specific to the underlying timer type, the uAPI only presents one single set of functions for all available timers due to its abstraction of the underlying timer type.

5.5.1 Basic Common Timer Functions. Basic functions that are commonly found on all timer types are exposed directly as separate functions by the uAPI. Feature access is provided by unbundling the compressed hAPI functions (see Section 5.4.3) into a user-friendly interface. An example is the hAPI `enable(run)` function that is split into the two more convenient `timer_start()` and `timer_stop()` uAPI functions.

In contrast to hAPI drivers, the uAPI is not required to minimize the amount of provided functions. As no drivers exist within the uAPI, no function pointers need to be stored and therefore no fixed amount of memory needs to be allocated for each of the provided uAPI functions. Unused functions are removed by the compiler during optimization and therefore do not negatively affect the final binary size. Besides fully delegatable function calls, other

uAPI Function	Description

`tim_periph_t timer_get_periph(`
`  tim_t dev`
`);`
	Returns the corresponding timer peripheral struct for the timer device identifier `dev`.

`int timer_init(`
`  tim_periph_t *const tim,`
`  unsigned long freq,`
`  tim_clk_t clk,`
`  bool ovf,`
`  tim_cb_t cb,`
`  void *arg`
`);`
	Initializes the timer module `tim`. The clock source `clk` is used to achieve the requested operating frequency `freq` (number of timer ticks per second)[(a)]. If supported, counter overflow event generation `ovf` can be enabled. A callback function `cb` to be executed at timer event occurrence is registered. A pointer for binding additional arguments `arg` may also be given.

`int timer_start(`
`  tim_periph_t *const tim`
`);`
	Enables the timer `tim`, i.e., starts incrementing the internal counter of the hardware peripheral.

`int timer_stop(`
`  tim_periph_t *const tim`
`);`
	Disables the timer `tim`, i.e., stops incrementing the internal counter of the hardware peripheral.

`tim_cnt_t timer_read(`
`  tim_periph_t *const tim`
`);`
	Reads the current raw counter value of the internal counter register of timer `tim`.

`void timer_write(`
`  tim_periph_t *const tim,`
`  tim_cnt_t cnt`
`);`
	Sets the current raw counter value of the internal counter register of timer `tim` to the absolute value `cnt`.

`int timer_set(`
`  tim_periph_t *const tim,`
`  unsigned int channel,`
`  unsigned int timeout`
`);`
	Arms the timer channel `channel` to generate a timeout event after the specified time `timeout`. The timeout period is relative to the current internal counter value of the timer `tim`.

`int timer_set_absolute(`
`  tim_periph_t *const tim,`
`  unsigned int channel,`
`  tim_cnt_t cnt`
`);`
	Arms the timer channel `channel` to generate a timeout event after the internal counter register of the timer `tim` has reached the absolute counter value `cnt`.

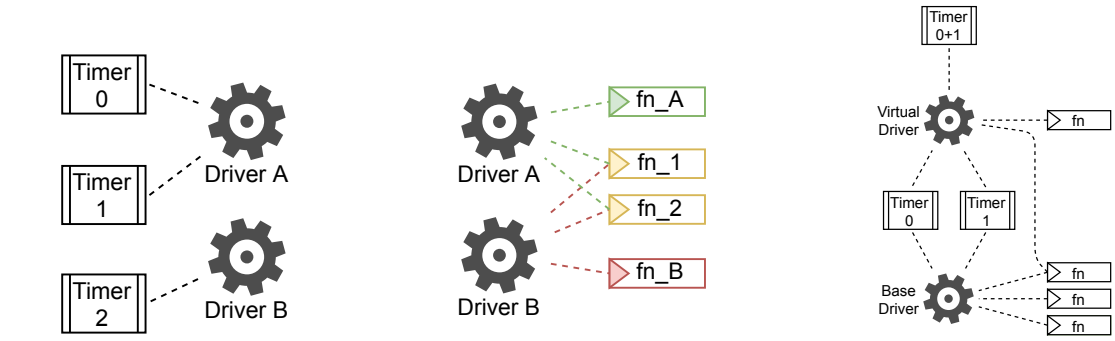
`int timer_clear(`
`  tim_periph_t *const tim,`
`  unsigned int channel`
`);`
	Disarms the timer channel `channel` of the timer `tim`. The channel is disarmed regardless of its current state.

`tim_propval_t timer_get_property(`
`  tim_periph_t *const tim,`
`  tim_prop_t prop`
`);`
	Reads the property `prop` of the timer `tim`.

`int timer_set_property(`
`  tim_periph_t *const tim,`
`  tim_prop_t prop,`
`  tim_propval_t val`
`);`
	Sets the property `prop` of the timer `tim` to the value `val`.

(a) If the specified clock source cannot safely be configured to generate the requested frequency an appropriate error code is returned and the timer is left uninitialized. A function that determines the nearest achievable frequency is optionally provided.

Table 3: Excerpt of the user-facing API. Functions are independent of underlying timer types and designed to be used by the user application or other high-level RIOT-OS modules. Additional functions for convenient feature access can be provided. Failed operations or incorrect usage is indicated by respective return values.



(a) Driver granular code reusability **(b) Function granular code reusability** **(c) A virtual timer driver**

Figure 2: Illustrations of hardware-facing API design concepts

basic convenience functions are provided to the user. As these can entirely be fulfilled by combining multiple hAPI function calls, they are implemented solely inside the uAPI. We refer to such functions as *compound functions*. An example of such is setting relative timeouts, as depicted in Figure 3.

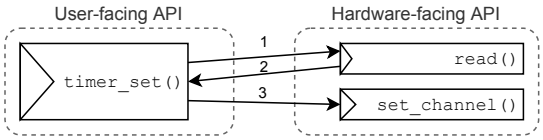


Figure 3: Exemplary uAPI compound function. Relative timer arming is here implemented solely within the uAPI as a combination of hAPI function calls.

5.5.2 Advanced Timer Functions and Status Information. Advanced timer functions, which are not broadly available and therefore timer type dependent, are made accessible via the hAPI (see Section 5.4.2). The provided interface is based on getter and setter functions that allow read and write access to the respective timer properties and features. They are exposed by the uAPI in the same way through the `timer_get_property(tim_prop_t)` and `timer_set_property(tim_prop_t, tim_propval_t)` functions. All available timer features and peripheral properties that are listed in the `tim_prop_t` enum can be addressed (see Listing 3). This includes both advanced timer functions and run-time dynamic timer properties.

To further aid usability, the uAPI can provide additional wrapper functions for conveniently accessing specific features and properties. Examples include, but are not limited to:

(a) `timer_is_enabled()`: Determines if a timer is running (i.e., counting)
(b) `timer_has_pending_ovf()`: Determines if a timer has an unhandled overflow pending
(c) `timer_set_count_dir()`: Changes a timers counting direction
(d) `timer_irq_ovf_enable()`: Enables generation of overflow interrupts

These convenience functions again do not end up in the final binary if unused, due to applied compile-time optimization.

5.6 Counter Value

Values of the internal counter register that each timer offers are represented by the `tim_cnt_t` type throughout both the hAPI and the uAPI. The exact size of the underlying datatype can be selected during compile-time. On MCU platforms that only offer small timers (e.g., 16-bit) the counter type width can hereby be adjusted accordingly to benefit performance and save memory. The selected counter width must, however, be at least the size of the largest timer exposed to the application, as configured compile-time static.

5.7 Interrupt Handling

Two separate callback functions can be attached to every timer instance during initialization. One that is executed after a compare match IRQ was generated (`tim_cmp_cb_t`) and one to be executed after a counter register overflow IRQ occurred (`tim_ovf_cb_t`). If one of both is provided, it is executed by the low-level timer module once the corresponding IRQ was generated and after all required interrupt maintenance tasks, such as updating the interrupt status registers, were performed. During invocation, the interrupt cause `tim_int_t` is determined by the executed callback and, if applicable, the triggered timer channel is passed as a function argument. An optional context can furthermore be bound to each of the callback functions via a void pointer. Interrupt generation (i.e., masking) can be run-time configured via the timer property interface.

The compare match and overflow callbacks are handled individually due to the disjoint use-cases of both events. Counter overflows are most often used solely for timer maintenance tasks whereas compare match events indicate elapsed timeouts, which the user application wants to be informed of. By removing the need to determine the interrupt cause within the attached callback function shorter maintenance periods can be achieved, hereby decreasing the overall resource consumption as well as reducing the latency of elapsed timeouts.

Channel specific callback functions however are not separated further, thus all compare match events are handled within the same function. Only 31% of all analyzed timer types where found to provide distinct interrupts for every compare channel. Storing a large number of callback function pointers, which will remain unused on many platforms, therefore is considered inapplicable. Although, a small number of platforms may experience a slightly lower timeout

latency, the majority of all MCU platforms only suffers the larger memory footprint, due to the additional function pointers that need to be stored, without gaining any benefits. Nonetheless, individual functions can still be dispatched within the attached compare match callback function, based on the provided channel argument.

5.8 Run-time Clock Configuration

System clock configuration is commonly implemented entirely compile-time static. In order to provide flexibility and allow dynamic power mode transitions, the low-level timer API supports changing timer clock sources during run-time. If not handled by a designated system-wide clock configuration module, it can be implemented within the timer-API itself. With this, however, two crucial restrictions apply:

(1) Selecting a platform specific clock source comes at the cost of loosing application portability.
(2) A clock source is only allowed to be changed during run-time, if its altering does not affect other timers or hardware peripherals in any way, i.e., it does not introduce side effects.

A clock source is uniquely identified by the `tim_clk_t` type. Its default definition might be used, though it is highly encouraged to redefine it for every MCU platform in order to reflect the actually available clocks. To cope with restriction (1), both platform specific clocks as well as platform independent abstract clock classes can be provided. The first allow for fine-grained platform specific optimizations while the latter provide a trade-off between optimization potential and application portability.

Restriction (2) entails, that a closer look at the clock tree of the respective MCU platform is mandatory during implementation. The driver developer must determine the impact domain, i.e., the set of affected peripherals, as well as possible side effects of a clock change. Given a timer clock *TIMx_CLK* can be selected among the three different clock sources *CLK_A*, *CLK_B*, and *CLK_C*. If *TIMx_CLK* can be selected among all clock sources individually for each timer module, available CLKs are: *CLK_A*, *CLK_B*, *CLK_C*. If however, using *CLK_A* as a clock for *TIMx_CLK* implies this choice for multiple peripherals, only *TIMx_CLK* is an allowed selection. In other words: When traversing the hierarchical clock tree, only clocks on the path originating from the timer peripheral up to the first branch that leads to other peripherals, are applicable.

Moreover, techniques for deriving abstract descriptions of MCU clock trees and dynamically altering clock configurations are being developed. One such novel approach is FlexClock [16], allowing run-time clock re-configuration in a generic platform-independent fashion. It combines a flexible scheme for light-weight modeling of MCU clock trees with a generic software implementation that can dynamically explore and reconfigure the modeled clock-trees during run-time. We suggest employing such techniques when implementing the proposed API design for a diverse range of MCU platforms from different manufacturers.

5.9 Issues

The proposed low-level timer-API brings many advantages when compared to the current RIOT-OS peripheral drivers, as depict in Section 4. These include the possibility to integrate currently unsupported timer types, all usable through a unified and MCU platform independent API. Advanced timer features and properties are exposed and timer configuration possibilities are widened. With the proposed API design, however, some implications and trade-offs still exist. These are outlined in this section.

As only one uniform set of functions should be provided by the uAPI, the same `tim_cnt_t` definition has to be used for all timer types. It must be able to store values of the widest counter register that is made available to the application. This implies that small timers (e.g., 16-bit) are required to use unnecessarily large data types when at least one larger timer (e.g., 32-bit) is exposed. This trade-off was made to avoid having separate bit-size dependent APIs or error prone void pointer casting.

When compared to existing implementations, the proposed API comes with an increased memory footprint. This is due to the function pointers within `tim_driver_t` structs. As such driver structs exists for every exposed timer type, the actual increase in memory consumption strongly depends on the used timers and how they are interfaced. If, for example, an application uses timers via three of the existing APIs, replacing them with one uniform low-level timer-API could potentially fully outweigh the described resource impact. Future work (see Section 6) shall determine the exact impact in different application scenarios. Though noteworthy, we expect the increase to be negligible in most of the use-cases.

Last, as with every module reorganization, implementations of the new API must be developed for all MCU platforms that are supported by RIOT-OS. Even though this is not considered to be an issue of the API design itself it must still be taken into account. Existing driver code segments for basic timer operation can of course be reused during this process. Solely additional timer functions that are currently left unimplemented require additional development effort. It can furthermore be desirable to provide appropriate compatibility layers for high-level modules.

6 FUTURE WORK

Completing outstanding tasks and resolving open issues related to the conducted hardware platform analysis, as described in Section 3.4, are next on our agenda. Nonetheless, additional work needs to be done in order to enhance the proposed API design and to finally implement the aspired clean-slate timer-API for RIOT-OS. A coarse overview of our next steps towards this goal is given in this section.

(1) *Abstract Timer Classes.* Our analysis revealed that, apart from general-purpose modules, timer peripherals vary greatly in function and availability between different manufacturers. To cope with this diversity we propose to define abstract timer classes, each describing a particular set of features a hardware timer of the respective category offers. Well-considered definition of appropriate categories has to be done. Conceivable types may include the following: general-purpose, low-power, high-resolution, and long-running timers. Introducing such would benefit a high-level timer module by allowing platform-agnostic and dynamic management of available timer resources, selecting the most appropriate ones for the current application.
(2) *Availability Analysis.* Strongly linked to the definition of abstract timer classes is the evaluation of their availability

across all hardware platforms. This step cannot only aid developers during selection of appropriate MCUs for their respective applications. It moreover allows us to estimate the total number of platforms that would potentially benefit from different aspects of the timer-API.

(3) *Micro-benchmarks.* Design decisions regarding the proposed low-level timer-API shall be evaluated with micro-benchmarks. This step is crucial in order to generate performance data that can then be used to compare different implementation variants as well as the impact of individual design decisions against each other. A respective set of use-cases and application scenarios needs to be carefully defined for this step. Conceivable analyses include the impact on memory footprint and maintenance overhead as well as experienced timeout latency. Further trade-off analyses, such as extending an existing function versus providing an additional API call shall also be part of the planned micro-benchmarks.

(4) *High-level Timekeeping-API Design.* Creating a new potential high-level timer-API for RIOT-OS is a long term goal of our work. During this step, all insights gathered from our conducted research may be taken into account. Advanced features that are made available by the proposed low-level timer-API, as depict in Section 5, shall of course also be incorporated during this process. Considering proposed methods and avoiding documented pitfalls from related work, as showcased in Section 2.2, is furthermore desired. Requirements for the high-level timer module shall be defined with respect to various application scenarios and different characteristics. The latter includes among others: maintenance complexity, power-saving and energy-efficiency, platform abstraction, API design, and maintainability. Based on the derived requirements a prototypical high-level timer subsystem can then be implemented and tested for a subset of the available platforms.

7 CONCLUSION

The contribution of this work was threefold. We first reviewed related work, depicting both timer hardware and software design considerations. Various implementation techniques and common pitfalls that are to be avoided were highlighted. Second, we conducted a large-scale timer hardware analysis covering all MCU-platforms that were supported by RIOT-OS at the time of writing. We provided detailed information about every timer peripheral type that is available on the targeted platforms and derived comparative inter-MCU-platform findings. Analysis results were then discussed with respect to the development of a clean-slate timer subsystem for RIOT-OS. Outstanding tasks as well as open issues of the conducted hardware platform analysis were subsequently outlined. We furthermore contributed an overview of all low-level timer peripheral drivers of RIOT-OS, highlighting common design aspects as well as their limitations. Third, we used insights gained from the review of related work, the conducted hardware analysis, and the low-level timer driver comparison to propose a comprehensive low-level timer-API design. It is split into a hardware-facing and a user-facing API component, thereby exposing the various available timer types through the same uniform API to both the user application and high-level modules. A platform independent set of basic features is offered while access to specialized timer features that are only available on a subset of peripherals is also made possible. Conclusively, outstanding tasks and future work were outlined.

ACKNOWLEDGMENTS

We would like to thank Michel Rottleuthner (Hamburg University of Applied Sciences) for his help during the collection of the data that was used in the hardware platform analysis as well as for sharing his expertise in this field of research in general.

REFERENCES

[1] Mohit Aron and Peter Druschel. 2000. Soft Timers: Efficient Microsecond Software Timer Support for Network Processing. *ACM Transactions on Computer Systems (TOCS)* 18, 3 (Aug. 2000), pages 197–228. https://doi.org/10.1145/354871.354872

[2] Emmanuel Baccelli, Cenk Gündogan, Oliver Hahm, Peter Kietzmann, Martine Lenders, Hauke Petersen, Kaspar Schleiser, Thomas C. Schmidt, and Matthias Wählisch. 2018. RIOT: an Open Source Operating System for Low-end Embedded Devices in the IoT. *IEEE Internet of Things Journal* 5 (Dec. 2018), pages 4428–4440. https://doi.org/10.1109/JIOT.2018.2815038

[3] Emmanuel Baccelli, Oliver Hahm, Mesut Günes, Matthias Wählisch, and Thomas C. Schmidt. 2013. RIOT OS: Towards an OS for the Internet of Things. In *Proceedings of the 2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE Press, Piscataway, NJ, USA, pages 79–80. https://doi.org/10.1109/INFCOMW.2013.6970748

[4] Patrick Bellasi. 2009. *Linux Power Management Architecture: A review on Linux PM frameworks.* Technical Report. Politecnico di Milano, Dipartimenti di Elettronica e Informazione.

[5] Thomas Gleixner and Douglas Niehaus. 2006. Hrtimers and Beyond: Transforming the Linux Time Subsystems. In *Proceedings of the 2006 Ottawa Linux Symposium (Volume One)*. pages 333–346.

[6] Giovani Gracioli, Danillo Santos, Roberto Matos, Lucas Wanner, and Antônio Fröhlich. 2008. One-shot time management analysis in EPOS. In *Proceedings of the International Conference of the Chilean Computer Science Society*. pages 92–99. https://doi.org/10.1109/SCCC.2008.13

[7] Vlado Handziski, Joseph Polastre, J.-H Hauer, Cory Sharp, Adam Wolisz, and David Culler. 2005. Flexible Hardware Abstraction for wireless sensor networks. In *Proceeedings of the Second European Workshop on Wireless Sensor Networks*. pages 145–157. https://doi.org/10.1109/EWSN.2005.1462006

[8] Wanja Hofer. 2014. *Sloth: The Virtue and Vice of Latency Hiding in Hardware-Centric Operating Systems.* Doctoral Thesis. Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU).

[9] Wanja Hofer, Daniel Danner, Rainer Muller, Fabian Scheler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2012. Sloth on Time: Efficient Hardware-Based Scheduling for Time-Triggered RTOS. In *Procceedingss of the 33rd IEEE Real-Time Systems Symposium*. pages 237–247. https://doi.org/10.1109/RTSS.2012.75

[10] Wanja Hofer, Daniel Lohmann, and Wolfgang Schröder-Preikschat. 2011. Sleepy Sloth: Threads as Interrupts as Threads. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium*. pages 67–77. https://doi.org/10.1109/RTSS.2011.14

[11] Jupyung Lee and Kyu-Ho Park. 2005. Delayed locking technique for improving real-time performance of embedded Linux by prediction of timer interrupt. In *Proceedings of the 11th IEEE Real Time and Embedded Technology and Applications Symposium*. pages 487– 496. https://doi.org/10.1109/RTAS.2005.16

[12] Raj Kamal. 2011. *Embedded Systems: Architecture, Programming and Design* (second ed.). Tata McGraw Hill Education.

[13] Per Lindgren, Emil Fresk, Marcus Lindner, Andreas Lindner, David Pereira, and Luís Miguel Pinho. 2016. Abstract Timers and Their Implementation onto the ARM Cortex-M Family of MCUs. *ACM SIGBED Review* 13 (Mar. 2016), pages 48–53. https://doi.org/10.1145/2907972.2907979

[14] Vesna Mincev and Dragan Milicev. 1998. A Tree-Driven Multiple-Rate Model of Time Measuring in Object-Oriented Real-Time Systems. In *Proceedings of the Conference on Parallel and Distributed Processing (IPPS)*. Springer Berlin Heidelberg, pages 1037–1046. https://doi.org/10.1007/3-540-64359-1_769

[15] Pratyush Patel, Manohar Vanga, and Bjorn Brandenburg. 2017. TimerShield: Protecting High-Priority Tasks from Low-Priority Timer Interference. In *Proceedings of the 2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. pages 3–12. https://doi.org/10.1109/RTAS.2017.40

[16] Michel Rottleuthner, Thomas C. Schmidt, and Matthias Wählisch. 2021. FlexClock: Generic Clock Reconfiguration for Low-end IoT Devices. arXiv:eess.SY/2102.10353

[17] Ioan Susnea and Marian Mitescu. 2005. *Microcontrollers in Practice (Springer Series in Advanced Microelectronics)* (first ed.). Springer-Verlag, Berlin, Heidelberg.

https://doi.org/10.1007/3-540-28308-0

[18] Andrew S. Tanenbaum. 2009. *Moderne Betriebssysteme* (3. aktualisierte auflage ed.). Pearson Studium.

[19] *corbet* (Pseudonym). 2007. Deferrable timers. News Article. Released in Linux Weekly News (LWN). https://lwn.net/Articles/228143/

[20] Edward W. Thompson and Stephen A. Szygenda. 1975. Three levels of accuracy for the simulation of different fault types in digital systems. In *Proceedings of the 12th Design Automation Conference (DAC)*. IEEE Press, pages 105–113.

[21] Dan Tsafrir. 2007. The Context-Switch Overhead Inflicted by Hardware Interrupts (and the Enigma of Do-Nothing Loops). In *Proceedings of the 2007 Workshop on Experimental Computer Science* (San Diego, California) *(ExpCS '07)*. Association for Computing Machinery, New York, NY, USA, pages 4–es. https://doi.org/10.1145/1281700.1281704

[22] Dan Tsafrir, Yoav Etsion, and Dror Feitelson. 2005. *General purpose timing: the failure of periodic timers.* Technical Report. School of Computer Science & Engineering, The Hebrew University.

[23] Ioanna Tsekoura, Gregor Rebel, Mladen Berekovic, and Peter Glösekötter. 2014. An evaluation of energy efficient microcontrollers. In *Proceedings of the 9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*. https://doi.org/10.1109/ReCoSoC.2014.6861368

[24] George Varghese and Anthony Lauck. 1997. Hashed and Hierarchical Timing Wheels: Efficient Data Structures for Implementing a Timer Facility. *IEEE/ACM Transactions on Networking* 5, 6 (Dec. 1997), pages 824–834. https://doi.org/10.1109/90.650142

A HARDWARE ANALYSIS RESULTS

Detailed results from the conducted timer hardware analysis are found in the following tables. Each table contains the analyzed timer module types and their respective properties for a set of MCUs as indicated by the table captions.

A.1 Column Key / Explanation of Criteria

A.1.1 Timer Type. Name of the respective timer type. Generic timer modules across various platforms are united under the type name "General-purpose" in order to be easily identifiable throughout the results. Names of special purpose timers are adopted from naming conventions in the corresponding datasheets.

A.1.2 Counter Width. Width of the internal counter register in bits. If multiple counter widths are available for a single timer type, these are listed below each other inside a single cell. Can be omitted if timer does not contain plain counter register (e.g., real-time-clocks).

A.1.3 Compare Channels. Number of compare channels available in a single timer module of the given type. Can be a single number, a range or multiple fixed values.

A.1.4 Prescaler Type. Availability of a prescaler that divides the timer clock. Can be one of the following:

- × No prescaler is available.
- E Prescaler can be continuously selected as exponentials of 2 (e.g., $1, 2, 4, 8, \ldots, 2^n$).
- F Prescaler can be selected from fixed values with varying intervals (e.g., 1, 16, 64, 512).
- R Prescaler can be continuously selected as discrete integer values (e.g., $1, 2, 3, 4, \ldots, 65536$).

A.1.5 Max Prescaler. Maximum value that can be selected as a prescaler (i.e., greatest clock divider resulting in longest time to over-/underflow) with respect to *Prescaler Type*. Can be omitted when *Prescaler Type* is ×.

A.1.6 Chaining Support. Indicates if chaining timers of the given type is possible. This feature can be used to combine small counters into a larger one (e.g., combining two 16-bit timers into a 32-bit timer). Can be one of the following:

- × No support for timer chaining available. Chaining by routing signals through additional peripherals is counted as not available.
- ✓ Combination of multiple timer modules is possible (e.g., configured in timer control registers).

A.1.7 Compare INT. Type of interrupts generated on a compare channel match event. Can be omitted if *Compare Channels* is 0. Can be one of the following:

- × Non-existing. Compare matches cannot generate any kind of interrupt.
- ○ Available but shared with other timer events. Applies if only a single interrupt per timer module is available.
- □ Available but shared with other compare channels. Applies if a single timer module has one interrupt that exclusively services all its compare matches.
- ✓ Available and offering unique interrupts for each compare channel (i.e., no status bit / event flag read is necessary to identify the compare channel that produced the match event).

A.1.8 Overflow INT. Type of the interrupt generated on a counter register over-/underflow. Can be one of the following:

- × Non-existing. A counter over-/underflow cannot generate any kind of interrupt.
- ○ Available but shared with other timer events. Applies if only a single interrupt per timer module is available.
- ✓ Available and offering a unique interrupt (i.e., no status bit / event flag read is necessary to distinguish from compare matches).

A.1.9 Event Flags. Determines the availability of status bits that indicate if an event (e.g., compare match or over-/underflow) was observed by the timer hardware. These flags need to be updated independently of the generated interrupts and must be available even if the corresponding interrupt is currently masked. Can be one of the following:

- × No event status bits / flags available.
- ✓ Event status bits / flags are available and updated even if the corresponding interrupt is masked.

A.1.10 Auto-reload. Availability and type of the auto-reload function. Can be one of the following:

- × Not available (i.e., one-shot mode).
- ○ Timer auto-reloads / warps only at counter over-/underflow (i.e., full width free-running mode).
- □ Auto-reload at arbitrary value is available but sacrifices one compare channel (i.e., limited width free-running mode).
- ✓ Auto-reload at arbitrary value is available. No compare channel is required, exclusive auto-reload match register available (i.e., limited width free-running mode).

A.1.11 PWM Generation. Indicates if a timer module can directly generate and output pulse-width-modulation (PWM) waveforms. Can be one of the following:

- × Not available. PWM generation through additional peripherals (e.g., exclusive PWM peripheral) counts as not available.
- ✓ PWM generation available.

A.1.12 Internal CLKs. Number of internal clocks the timer is able to run of.

A single clock is categorized as internal, if it can in at least one case be configured to be driven from an internal oscillator. If it is able to run from either an internal or external oscillator, it is categorized as both internal and external clock.

Listed clocks are based on their scope and potential side effects on other peripherals. Given a timer clock (e.g., `TIMx_CLK`) can be selected among three different clock sources (e.g., `CLK_A`,`CLK_B`, `CLK_C`). If `TIMx_CLK` can be selected among all clock sources individually for each timer module, available *Internal CLKs* are: `CLK_A`,`CLK_B`, `CLK_C`. If however, using `CLK_A` as a source clock for `TIMx_CLK` implies this choice for multiple peripherals, only `TIMx_CLK` is counted as *Internal CLK*. In other words: Within the hierarchical clock tree, only clocks starting from the timer peripheral up to the first branch that leads to other peripherals, are listed here.

A.1.13 External CLKs. Number of external clocks the timer is able to run of.

A single clock is categorized as external, if it can in at least one case be configured to be driven from an external oscillator. If it is able to run from either an external or internal oscillator, it is categorized as both external and internal clock.

Listed clocks are based on their scope and potential side effects on other peripherals. See Section A.1.12 for details.

A.1.14 Low-power CLK. Indicates if the timer module can be operated with a low-power clock source (internal or external). A low-power clock is defined as one that allows the CPU and high-frequency peripheral base clock to be turned off while the low-power clock is still operational (i.e., the timer can be operated in lower power-states). Can be one of the following:

× No low-power clock source available.
✓ Timer can be operated using a low-power clock source. Timer is operational in lower power states.

A.1.15 Deep-sleep Active. Indicates whether the timer is operational in the lowest power states of the MCU, as typically found with real-time-clocks. Very low power modes are characterized by the power-down of the CPU, nearly all peripherals, and oscillators. Modules of this category are often among the only wakeup-sources that can wake the device from deep sleep states. Can be one of the following:

× Timer is never active in the lowest power states.
✓ Timer can be operated in the lowest power states.

A.1.16 Unresolved or Not-applicable Items. In some cases one of the above described attributes does not apply to the timer module (e.g., counter width for some real-time-clocks), it is currently unknown or it is unclear and needs confirmation. In such cases one of the following values can be used for any of the above properties:

- Not applicable
? Unknown / Documentation unclear / Needs confirmation

Timer Type	Counter Width	Compare Channels	Prescaler Type	Max. Prescaler	Chaining Support	Compare INT	Overflow INT	Event Flags	Auto-reload	PWM Generation	Internal CLKs	External CLKs	Low-power CLK	Deep-sleep Active
General-purpose	16 bit 32 bit	1-4	R	2^{16}	✓	○	○	✓	✓	✓	2	3	×	×
Advanced-control	16 bit	4, 6	R	2^{16}	✓	○	○	✓	✓	✓	2	3	×	×
Basic	16 bit	0	R	2^{16}	✓	×	○	✓	✓	×	1	1	×	×
Low-power	16 bit	1	E	2^{7}	×	○	×	✓	✓	✓	3	3	✓	×
SysTick	24 bit	0	F	2^{3}	×	×	✓[b]	✓	✓	×	1	1	×	×
Real-time clock	-	1-2[c]	R[e]	2^{7+15}	×	○	○[d]	✓	-	×	1	2	✓	✓
Independent WDG	12 bit	0	E	2^{8}	×	×	×	-	×	×	1	0	✓[e]	✓
System window WDG	7 bit	0	E	2^{12+3}	×	×	×	-	×	×	1	1	×	×

Table 4: Timer Comparison Matrix: STMicroelectronics STM32

Timer Type	Counter Width	Compare Channels	Prescaler Type	Max. Prescaler	Chaining Support	Compare INT	Overflow INT	Event Flags	Auto-reload	PWM Generation	Internal CLKs	External CLKs	Low-power CLK	Deep-sleep Active
General-purpose	8 bit 16 bit	2 2-3	F	2^{10}	×	✓	✓	✓[a]	□	✓	1	2	✓	×
Asynchronous	8 bit	2	F	2^{10}	×	✓	✓	✓[a]	□	✓	1	3	✓	✓
High-speed	10 bit	3	E	2^{14}	×	✓	✓	✓[a]	□	✓	2	2	×	×
Watchdog	-	0	E	2^{10}	×	×	✓	✓	×	×	1	0	✓[e]	✓

Table 5: Timer Comparison Matrix: Microchip / Atmel megaAVR

Timer Type	Counter Width	Compare Channels	Prescaler Type	Max. Prescaler	Chaining Support	Compare INT	Overflow INT	Event Flags	Auto-reload	PWM Generation	Internal CLKs	External CLKs	Low-power CLK	Deep-sleep Active
General-purpose	16 bit 32 bit[g]	1	F	2^{8}	✓	✓	×	✓	✓	×	1	1	×	×
Asynchronous	16 bit	1	F	2^{8}	×	✓	×	✓	✓	×	1	2	✓	✓
Real-time clock	-	1	×	-	×	✓	×	✓	-	×	0	1	✓	✓
Watchdog	25 bit	0	E	2^{20}	×	×	✓	✓	×	×	1	0	✓	✓

Table 6: Timer Comparison Matrix: Microchip PIC32MX/MZ

Timer Type	Counter Width	Compare Channels	Prescaler Type	Max. Prescaler	Chaining Support	Compare INT	Overflow INT	Event Flags	Auto-reload	PWM Generation	Internal CLKs	External CLKs	Low-power CLK	Deep-sleep Active
Timer Counter (TC)[o]	16 bit	3	F	2^{7}	✓	○	○	✓	□	✓	2	5	✓	×
Pulse Width Modulation (PWM)[o]	16 bit	0	E	2^{10}	×	×	✓	✓	✓	✓	1	1	✓	×
SysTick	24 bit	0	F	2^{3}	×	×	✓	✓	✓	×	1	1	✓	×
Real-time timer (RTT)	32 bit	1	R	2^{16}	×	○	×	✓	○	×	1	1	✓	✓
Real-time clock (RTC)	-	1	F	2^{15}	×	○	×	✓	○	×	1	1	✓	✓
Watchdog (WDT)	12 bit	0	F	2^{7}	×	×	✓	✓	×	×	1	1	✓	×

Table 7: Timer Comparison Matrix: Microchip / Atmel SAM3

[a] When enabled
[b] System Tick Interrupt
[c] RTC Alarm(s)
[d] From periodic wakeup timer
[e] Independent oscillator
[f] For internal calibration only

Timer Type	Counter Width	Compare Channels	Prescaler Type	Max. Prescaler	Chaining Support	Compare INT	Overflow INT	Event Flags	Auto-reload	PWM Generation	Internal CLKs	External CLKs	Low-power CLK	Deep-sleep Active
General-purpose	8 bit 16 bit 32 bit[g]	2	F	2^{10}	✓	○	○	✓	✓	✓	1	1	✓	✓
General-purpose for Control	16 bit 24 bit	4	F	2^{10}	×[k]	○	○	✓	✓	✓	1	1	✓	✓
SysTick[j]	24 bit	0	?	?	×	×	✓[b]	?	✓	×	1	1	×	×
Real-time counter	32 bit	1	E	2^{10}	×	○	○	✓	✓	×	3	1	✓	✓
Watchdog	-	2	-	-	-	○	-	✓	-	×	2	1	✓	✓

Table 8: Timer Comparison Matrix: Microchip / Atmel SAMD21

Timer Type	Counter Width	Compare Channels	Prescaler Type	Max. Prescaler	Chaining Support	Compare INT	Overflow INT	Event Flags	Auto-reload	PWM Generation	Internal CLKs	External CLKs	Low-power CLK	Deep-sleep Active
General-purpose (FRC1)	23 bit	0	F	2^{8}	×	×	✓	?	○	✓	0	1	×	×
General-purpose (FRC2)	32 bit	1	F	2^{8}	×	✓	×	?	○	×	0	1	×	×
Real-time clock	32 bit	?	×	-	×	?	?	?	-	×	0	1	✓	✓
Watchdog	-	0	×	-	×	×	×	-	×	×	0	1	?	?

Table 9: Timer Comparison Matrix: Espressif ESP8266

Timer Type	Counter Width	Compare Channels	Prescaler Type	Max. Prescaler	Chaining Support	Compare INT	Overflow INT	Event Flags	Auto-reload	PWM Generation	Internal CLKs	External CLKs	Low-power CLK	Deep-sleep Active
General-purpose	64 bit	1	R	2^{16}	×	✓	×	?	□	×	1	1	×	×
Real-time clock	48 bit	1	×	-	×	✓	×	?	-	×	1	1	✓	✓
Main System Watchdog	32 bit	0	×	-	×	×	✓	?	×	×	1	1	×	×
RTC Watchdog	32 bit	0	×	-	×	×	✓	?	×	×	1	1	✓	✓

Table 10: Timer Comparison Matrix: Espressif ESP32

[g] Requires two hardware timer modules
[h] Incremented on every RTC count pulse
[i] Additional 8-bit repeat register
[j] Reference manual does not provide details
[k] Possible via events and another TCC utilized as event counter
[l] Clocked by SysClk which may use any available oscillator

Timer Type	Counter Width	Compare Channels	Prescaler Type	Max. Prescaler	Chaining Support	Compare INT	Overflow INT	Event Flags	Auto-reload	PWM Generation	Internal CLKs	External CLKs	Low-power CLK	Deep-sleep Active
General-purpose	16 bit 32 bit	3-4	E	2^{10}	✓	○	○	✓	✓	✓	2	2	×	×
Pulse counter	8 bit 16 bit	0	×	-	×	×	○	✓	✓	×	1	2	✓	✓
Low-energy	16 bit	2	E	2^{15}	×	○	○	✓	□	✓	1	1	✓	✓
Cryotimer	32 bit	1	E	2^{7}	×	✓	×	✓	○	×	2	1	✓	✓
SysTick[j]	24 bit	?	?	?	?	?	?	?	?	?	?	?	?	?
Real-time-counter	24 bit 32 bit	2	E	2^{15}	×	○	○	✓	□	×	1	1	✓	✓
Real-time clock	32 bit	3	E	2^{15}	×	○	○	✓	-	×	2	1	✓	✓
Watchdog	-	1	E	2^{17}	×	○	○	✓	×	×	3	2	✓	✓

Table 11: Timer Comparison Matrix: Silicon Labs EFM32/EFR32

Timer Type	Counter Width	Compare Channels	Prescaler Type	Max. Prescaler	Chaining Support	Compare INT	Overflow INT	Event Flags	Auto-reload	PWM Generation	Internal CLKs	External CLKs	Low-power CLK	Deep-sleep Active
General-purpose	16 bit	3	E	2^{10}	✓	○	○	✓	✓	✓	1	1	×	×
Low-energy	16 bit[i]	2	E	2^{15}	×	○	○	✓	□	✓	1	1	✓	×
SysTick[j]	24 bit	?	?	?	?	?	?	?	?	?	?	?	?	?
Real-time counter	24 bit	2	E	2^{15}	×	○	○	✓	□	×	1	1	✓	×
Backup Real-time counter	32 bit	1	E	2^{7}	×	○	×	✓	✓	×	2	1	✓	✓
Watchdog	-	0	F	2^{18}	×	○	×	✓	✓	×	2	1	✓	×

Table 12: Timer Comparison Matrix: Silicon Labs EZR32

[m] Only available in RTC-mode with external clock
[n] Supports masking of individual bits
[o] Module contains multiple timer peripherals. Values shown refer to a single counter
[p] Only pre-defined intervals are selectable

Timer Type	Counter Width	Compare Channels	Prescaler Type	Max. Prescaler	Chaining Support	Compare INT	Overflow INT	Event Flags	Auto-reload	PWM Generation	Internal CLKs	External CLKs	Low-power CLK	Deep-sleep Active
General-purpose (GPTM)	16 bit 32 bit[g]	2	R	2^8	✓	✓	○	✓	✓	✓	1	1	×	×
AUX Timer 0 AUX Timer 1	16 bit	0	E	2^{15}	✓	×	✓	✓	✓	×	1	1	✓	×
AUX Timer 2	16 bit	4	R	2^8	✓	✓	×	✓	✓	✓	3	2	✓	×
Radio Timer	32 bit	3	×	-	×	○	×	✓	○	×	1	0	×	×
SysTick	24 bit	0	×	-	×	-	✓	✓	✓	×	1	1	×	×
Real-time clock (RTC)	70 bit	3	×	-	×	✓	×	✓	○	×	1	1	✓	✓
Watchdog (WDT)	32 bit	0	F	2^5	×	×	✓	✓	○	×	1	1	×	×

Table 13: Timer Comparison Matrix: Texas Instruments CC13x2 / CC26x2

Timer Type	Counter Width	Compare Channels	Prescaler Type	Max. Prescaler	Chaining Support	Compare INT	Overflow INT	Event Flags	Auto-reload	PWM Generation	Internal CLKs	External CLKs	Low-power CLK	Deep-sleep Active
General-purpose	16 bit 32 bit[g]	2	R	2^8	✓	○	○	✓	✓	✓	1	1	×	×
MAC Timer	16 bit	2	×	-	×	○	○	✓	○	×	1	1	✓	✓
Sleep (SM) Timer	32 bit	1	×	-	×	✓	×	✓	○	×	1	1	✓	✓
SysTick	24 bit	0	×	-	×	-	✓	✓	✓	×	1	1	×	×
Watchdog	15 bit	1[p]	×	-	×	×	×	-	×	×	1	1	✓	✓

Table 14: Timer Comparison Matrix: Texas Instruments CC2538

Timer Type	Counter Width	Compare Channels	Prescaler Type	Max. Prescaler	Chaining Support	Compare INT	Overflow INT	Event Flags	Auto-reload	PWM Generation	Internal CLKs	External CLKs	Low-power CLK	Deep-sleep Active
Timer_A	16 bit	7	F	2^6	×	○	○	✓	□	✓	2	4	✓	✓
Real-time clock A (RTC_A)	8 bit 16 bit 24 bit 32 bit	1	E	2^8	×	○	○	✓	○	×	2	2	✓	✓
Real-time clock D (RTC_D)	8 bit 16 bit 24 bit 32 bit	1	E	2^8	×	○	○	✓	○	×	0	1	✓	✓
Watchdog (WDT_A)	32 bit	1^p	×	-	×	✓	×	✓	×	×	3	2	✓	✓

Table 15: Timer Comparison Matrix: Texas Instruments CC430

Timer Type	Counter Width	Compare Channels	Prescaler Type	Max. Prescaler	Chaining Support	Compare INT	Overflow INT	Event Flags	Auto-reload	PWM Generation	Internal CLKs	External CLKs	Low-power CLK	Deep-sleep Active
General-purpose / RTC	16 bit 32 bit	12	R ×	2^8 -	✓	○	○	✓	✓	✓	1	2	✓	$✓^m$
General-purpose / RTC	32 bit 64 bit	12	R ×	2^{16} -	✓	○	○	✓	✓	✓	1	2	✓	$✓^m$
SysTick	24 bit	0	×	-	×	×	$✓^b$	✓	✓	×	2	1	$✓^l$	×
Watchdog (SysClk) Watchdog (PIOSC)	32 bit	0	×	-	×	×	✓	✓	×	×	1 1	1 0	✓ ×	✓

Table 16: Timer Comparison Matrix: Texas Instruments LM4F120

Timer Type	Counter Width	Compare Channels	Prescaler Type	Max. Prescaler	Chaining Support	Compare INT	Overflow INT	Event Flags	Auto-reload	PWM Generation	Internal CLKs	External CLKs	Low-power CLK	Deep-sleep Active
General-purpose (Timer A)	16 bit	2-3	E	2^3	×	○	○	✓	□	✓	2	4	✓	✓
General-purpose (Timer B)	8 bit 10 bit 12 bit 16 bit	3, 7	E	2^3	×	○	○	✓	□	✓	2	4	✓	✓
Watchdog	16 bit	0	×	-	×	×	✓	✓	×	×	3	3	✓	✓

Table 17: Timer Comparison Matrix: Texas Instruments MSP430x1xx / MSP430x2xx

Timer Type	Counter Width	Compare Channels	Prescaler Type	Max. Prescaler	Chaining Support	Compare INT	Overflow INT	Event Flags	Auto-reload	PWM Generation	Internal CLKs	External CLKs	Low-power CLK	Deep-sleep Active
FlexTimer (FTM) Timer/PWM Module (TPM)	16 bit	2-8	E	2^7	×	○	○	✓	✓	✓	4 1	3 2	✓	×
Quad Timer (TMR)[o]	16 bit	2	E	2^7	✓	○	○	✓	✓	✓	1	1	✓	×
Periodic Interrupt Timer (PIT) Low-power PIT (LPIT)	32 bit	2-4	×	-	✓	✓	×	✓	□	×	1 3	1 1	× ✓	× ✓
Pulse Width Timer (PWT)	16 bit	0	E	2^7	×	-	○	✓	○	×	1	2	✓	×
Low-power Timer (LPTMR)	16 bit	1	E	2^{16}	×	✓	×	✓	□	×	4	2	✓	✓
Real-time counter	16 bit	1	E	2^{11}	×	✓	×	✓	□	×	3	3	✓	✓
Real-time clock	32 bit	1	×	-	×	○	○	✓	○	×	1	3	✓	✓
Watchdog	16 bit	1	F	2^8	×	✓	×	✓	×	×	3	2	✓	✓

Table 18: Timer Comparison Matrix: NXP Semiconductors Kinetis

Timer Type	Counter Width	Compare Channels	Prescaler Type	Max. Prescaler	Chaining Support	Compare INT	Overflow INT	Event Flags	Auto-reload	PWM Generation	Internal CLKs	External CLKs	Low-power CLK	Deep-sleep Active
General-purpose	32 bit	4	R	2^{16}	×	□	×	✓	□	×	1	1	×	×
Repetitive Interrupt	32 bit	1^n	×	-	×	✓	×	✓	□	×	1	1	×	×
SysTick	24 bit	1	×	-	×	✓	×	✓	□	×	1	1	×	×
Real-time clock	-	2	×	-	×	□	×	✓	-	×	0	1	✓	✓
Watchdog	32 bit	0	F	2^2	×	-	×	-	-	×	2	2	✓	✓

Table 19: Timer Comparison Matrix: NXP Semiconductors LPC176x/5x

Timer Type	Counter Width	Compare Channels	Prescaler Type	Max. Prescaler	Chaining Support	Compare INT	Overflow INT	Event Flags	Auto-reload	PWM Generation	Internal CLKs	External CLKs	Low-power CLK	Deep-sleep Active
General-purpose	32 bit	4	R	2^{16}	×	□	×	✓	□	×	1	1	×	×
Real-time clock	-	2	R	2^{13}	×	□	×	✓	-	×	2	1	✓	✓
Watchdog	32 bit	0	F	2^2	×	-	×	-	-	×	2	2	✓	✓

Table 20: Timer Comparison Matrix: NXP Semiconductors LPC2387

Timer Type	Counter Width	Compare Channels	Prescaler Type	Max. Prescaler	Chaining Support	Compare INT	Overflow INT	Event Flags	Auto-reload	PWM Generation	Internal CLKs	External CLKs	Low-power CLK	Deep-sleep Active
General-purpose	8 bit 16 bit 24 bit 32 bit	4	E	2^9	×	○	×	?	○	×	2	2	×	×
Real-time counter	24 bit	4	R	2^{12}	×	○	○	?	○	×	1	1	✓	✓
Watchdog	32 bit	0	×	-	×	×	×	?	✓	×	1	0	✓	✓

Table 21: Timer Comparison Matrix: Nordic Semiconductor nRF51x/52x

Timer Type	Counter Width	Compare Channels	Prescaler Type	Max. Prescaler	Chaining Support	Compare INT	Overflow INT	Event Flags	Auto-reload	PWM Generation	Internal CLKs	External CLKs	Low-power CLK	Deep-sleep Active
Machine timer	64 bit	1	$\times^h$	-	×	✓	×	✓	○	×	1	1	✓	✓
Real-time counter	≥48 bit	1	E	2^{15}	×	✓	×	✓	-	×	1	1	✓	✓
Watchdog	31 bit	1	E	2^{15}	×	✓	×	✓	□	×	1	1	✓	✓

Table 22: Timer Comparison Matrix: SiFive FE310-Gx