Inter-MCU-Platform Hardware Analysis Towards a Clean-slate Timer-API 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 twofold. First, we illustrate various approaches to hardware timer usage as well as related research in this field. Second, we conduct a multi-manufacturer and inter-MCU-family analysis of the available timer peripherals and present our results in a comparative fashion. We give detailed insight into similarities and differences of hardware timer peripherals currently supported by RIOT-OS. Both the reviewing of related work and the conducted hardware analysis shall provide a baseline to later derive requirements and deduce applicable implementation techniques for a clean-slate timer-API from.

KEYWORDS

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

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 adopt to the new requirements; they become smaller and more energy efficient. To support this rapid growth especially 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, which 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 being *xtimer*, multiplexes all virtual software timers onto a single hardware module [2]. The long-term goal of our on-going 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 while also making use of the various power-saving features, including partly MCU-platform and -family specific ones. This work shall provide a baseline from which requirements for such a new timer driver can later be derived. It furthermore shall highlight implementation techniques and software concepts, potentially relevant for the aspired timer subsystem.

The remainder of this report is structured as follows. First, we list major conferences and publication sources at which on-going research is published, further illustrating various approaches to hardware timer usage as well as related work around this topic in Sections 2 and 3. Second, we conduct an in-depth analysis of various hardware timer peripherals, described in Section 4. We give detailed insight into similarities and differences of many hardware timers currently supported by RIOT-OS, taking nearly all maintained manufacturers and a broad selection of their respective MCU-families into account. Lastly, an outlook on future work with respect to the aspired timer subsystem for RIOT-OS is given in Section 5 before we finish with concluding words (see Section 6). Moreover, the entire results of our hardware analysis, including a detailed description of our applied criteria, are provided in the appendix Section A.

2 CONFERENCES & JOURNALS

Even though the amount of related work in this specific and narrow field of research is not vast, as depict in Section 3, there still is a significant body of relevant conferences and publication sources available. However, widening the scope from only timekeeping specific topics to the entire subject of embedded systems and Internet of Things (IoT) research is a necessity.

This section depicts some of the major conferences as well as the primary sources for scientific publications in the broad field of embedded systems engineering, operating systems for resource constrained devices, and the IoT community.

2.1 Major Conferences

Plenty of recent as well as on-going research findings are presented at technical conferences and are published in the respective conference proceedings. These conferences include, but are not limited to, the following in no particular order:

- USENIX Annual Technical Conference[2] (USENIX ATC)
- USENIX Symposium on Operating Systems Design and Implementation[3] (USENIX OSDI)
- ACM Symposium on Operating Systems Principles[4] (SOSP)

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

[2] USENIX ATC website archive: https://www.usenix.org/conferences/byname/131 (Accessed 13.01.2020)

[3] USENIX OSDI website archive: https://www.usenix.org/conferences/byname/179 (Accessed 30.01.2020)

[4] ACM SOSP website: http://www.sosp.org/ (Accessed 30.01.2020)

- ACM Conference on Embedded Networked Sensor Systems[5] (SenSys)
- ACM/IEEE International Conference on Information Processing in Sensor Networks[6] (IPSN)
- International Conference on Embedded Wireless Systems and Networks[7] (EWSN)

While most of the above listed conferences serve a wide variety of topics there also exist small gatherings, specialized into more specific subjects, like the *RIOT Summit*[8]. The latter is explicitly targeted at developers and researchers contributing to RIOT-OS and therefore of great relevance in the context of this work.

2.2 Publication Sources

Besides presenting research at conferences, as illustrated in Section 2.1, online publishing has emerged as the primary method of publishing recent scientific results. The main online libraries that are widely used in this field of research, among others, are:

- IEEE Xplore[9]
- ACM Digital Library[10]
- ArXiv[11] *(Caution: unreviewed pre-print only)*

It is also noteworthy that, with increasing popularity of the open-access model and online libraries, traditional journal exclusive publications are becoming less popular in the recent years. Nonetheless and without claim of completeness, the following journals and letters are common in this field of research:

- ACM Transactions on Sensor Networks[12] (TOSN)
- ACM Transactions on Embedded Computing Systems[13] (TECS)
- ACM SIGOPS Operating Systems Review[14] (OSR)
- IEEE Internet of Things Journal[15] (IEEE IoT)

Furthermore, search engines specialized in scientific publications like Google Scholar[16], Microsoft Academic[17], or Semantic Scholar[18] can be utilized while looking for research results.

3 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 3.1), and B) design aspects, algorithms, and implementation techniques used in timer driver software (Section 3.2). Publications were selected according to the relevance for this work, as estimated to the best of our knowledge.

3.1 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 4.

3.1.1 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]. 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 depict. Moreover, other types of timing hardware, here namely real-time-clocks (RTCs) and watchdogs (WDGs), are shortly outlined. For our conducted hardware analysis, commonly found timer characteristics can be inferred from this work. These include among others: counter register width, prescaler availability, and auto-reload capability.

While not with such detail as Kamal [12] did, Susnea and Mitescu [17, pp. 67-68, pp. 87-89] also give insight into general-purpose timer peripherals. However, 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.

3.1.2 Comparison of Embedded Timer Peripherals. A major focus of this work is 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 conducted by Mitescu and Susnea [15, 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. A generic timer peripheral block is further constructed from these 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 literature, but they do not cover timer peripherals in the required level of detail. For example, Tsekoura et al. [22] analyzed various MCU families sharing four manufacturers we also target. These chip manufacturers namely 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.

[5] ACM SenSys website: https://sensys.acm.org/ (Accessed 13.01.2020)
[6] IPSN website: https://ipsn.acm.org/ (Accessed 13.01.2020)
[7] EWSN website: http://www.ewsn.org/ (Accessed 13.01.2020)
[8] RIOT Summit website: https://summit.riot-os.org/ (Accessed 13.01.2020)
[9] IEEE Xplore website: https://ieeexplore.ieee.org/ (Accessed 13.01.2020)
[10] ACM Digital Library website: https://dl.acm.org/ (Accessed 13.01.2020)
[11] ArXiv website: https://arxiv.org/ (Accessed 13.01.2020)
[12] TOSN in the ACM-DL: https://dl.acm.org/journal/tosn (Accessed 30.01.2020)
[13] TECS in the ACM-DL: https://dl.acm.org/journal/tecs (Accessed 30.01.2020)
[14] OSR in the ACM-DL: https://dl.acm.org/newsletter/sigops (Accessed 30.01.2020)
[15] IEEE IoT on IEEE Xplore: https://ieeexplore.ieee.org/xpl/RecentIssue.jsp?punumber=6488907 (Accessed 30.01.2020)
[16] Google Scholar website: https://scholar.google.com/ (Accessed 27.01.2020)
[17] Microsoft Academic website: https://academic.microsoft.com/ (Accessed 27.01.2020)
[18] Semantic Scholar website: https://www.semanticscholar.org/ (Accessed 27.01.2020)

3.2 Software Modules

Since our long-term goal is to develop a new timer API for RIOT-OS, software aspects that relate to timer drivers also need to be taken into account. Publications depict in this subsection 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].

3.2.1 Generic Design Aspects. Varghese and Lauck [23] 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 [19], are proposed by the authors. These deliver a constant timer maintenance complexity through exploiting hashing and hierarchical relations. We therefore suggest taking this 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 to meet hard real-time deadlines, *Soft Timers* [1] can be used. They mitigate maintenance overhead during servicing of timer interrupts. More precisely, the costs, introduced by saving and restoring CPU state during context switches upon timer interrupts, are reduced. The key idea of the proposed solution is to only maintain timers if invoking the corresponding service routine is at 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. [21] and as in-depth analyzed in [20].

Lastly, Lindgren et al. [13] define a platform independent timer-API including design decisions for the implementation of such. 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. These identified generic characteristics namely 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 part of our hardware survey, as depicted in Section 4.

3.2.2 Application-specific Timer Implementations. Since the evolution of the Linux timer subsystem is well described, many 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 depicts it in the broader context of power management frameworks for Linux. Here, especially power-saving optimization techniques, such as deferrable timers [18], are addressed. Patel et al. [16] moreover stress the timer interrupt inference problem, which arises when processing of a high-priority timer expiry is delayed by 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 timer software 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*[19], 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 the in Section 4 depict broad MCU timer hardware analysis.

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[20]. 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.

A comparison between periodic and one-shot timers in the *Embedded Parallel Operating System*[21] (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 utilizing the *Real Time For the Masses*[22] (RTFM) framework. The Cortex-M SysTick- and

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

debug-timers are used as base timers for multiplexing and counting of clock cycles. Maintenance complexity, dispatch latency, and hardware setup time get characterized by the authors.

3.2.3 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 an non-urgent timer interrupts. When no interrupt from an urgent timer is expected, the system tick period is reduced, hence less wakeups occur. This dynamic adjustment of wakeups allows to meet real-time requirements, unlike the previously depict *Soft Timers* [1]. Particularly dividing available timers into multiple classes, each suited for specific application states, can 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 of 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. Multiplexing software timers onto a single hardware timer (e.g. periodic system tick timer) is enhanced by utilizing multiple hardware timers, reducing both maintenance overhead and scheduling latency. We expect that especially the latter technique will prove important for the aspired timer-API.

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

4 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 characterizing our selected scope as well as the methodology we apply during our analysis. 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.

4.1 Scope

The following analysis covers almost all of the chip manufacturers that offer at least one microcontroller that is supported by RIOT-OS at the time of writing. For each of those, a set of MCU families with RIOT-OS support was selected and examined. A detailed description of the analyzed criteria and properties as well as the applied methodology can be found in Section 4.2.

The following MCU families were analyzed by us:

- STMicroelectronics (ST)
  - STM32F0 / F1 / F2 / F3 / F4 / F7
  - STM32L0 / L1 / L2
- Microchip / Atmel
  - ATmega AVR
  - PIC32MX / PIC32MZ
  - SAMD21
- Espressif
  - ESP8266
  - ESP32
- Silicon Labs
  - EFM32 / EFR32
  - EZR32
- Texas Instruments (TI)
  - LM4F120
  - MSP430x1xx / MSP430x2xx
- NXP Semiconductors
  - LPC176x / LPC175x
- Nordic Semiconductor
  - nRF51x / nRF52x
- SiFive
  - FE310-Gx

Since exhaustively analyzing all of the currently supported MCUs was not feasible for us, only the above described subset was picked. One criterion for platform selection was to achieve a broad manufacturer coverage. Since most manufacturers use similar peripheral blocks throughout their MCU families covering as many manufacturers as possible enables us to get a comprehensive overview of the different available types of timer hardware. For each of the selected chip manufacturers those devices were chosen which: A) appear to be popular, and B) offer different timer peripherals compared to already selected MCU families.

4.2 Methodology

Each step of our hardware analysis is conceptually depict in this section. They appear in the order they were executed.

4.2.1 Platform Selection and Information Acquisition. As a starting point we looked at all CPUs that are currently supported by RIOT-OS, as listed in `/cpu`[23]. The analyzed chip manufacturers and their respective MCU families were then prioritized and picked according to the criteria defined in Section 4.1. For each of those, documentation in form of datasheets, reference manuals, application notes, and others were obtained from the respective manufacturers.

4.2.2 Definition of Analysis Criteria. After obtaining an initial overview, a set of criteria and properties, to at least be extracted from the gathered documents, was defined. Selected characteristics derive from our review of related work, as summarized in Section 3.3, 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.

4.2.3 Extraction of Timer Peripheral Details. All timer peripherals, including real-time-clocks/-counters and watchdogs, of each platform were analyzed and information found in the acquired documentation was transformed into a mind-map structure. Properties and implications that are beyond our defined criteria were recorded nonetheless in order to be used in future work (see Section 5). If the MCU documentation was unclear at some point, additional information sources were used. These include 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.

4.2.4 Consolidation of Results. Since our data acquisition was not limited to the defined criteria, it yielded more information than the properties defined before. Therefore, we once adopted and extended our set of analysis criteria before consolidating final results. For each MCU platform a *Timer Comparison Matrix (TCM)* was created. A TCM lists all of the available types of timer peripherals including their respective properties. It allows to quickly determine various characteristics and features of each of the available timers and enables the comparison of peripherals across different MCU families and manufacturers. All created TCMs can be found in Section A.

4.2.5 Inter-MCU-platform Findings. To obtain inter-platform insight we evaluated various properties across all platforms and timer types, either available directly in the TCMs or being derivable from them. If one timer type is available in multiple versions (e.g. 16-bit and 32-bit general-purpose timers) each of them was evaluated and counted separately. Platforms were counted as matching, if any of their available timer types matched the respective property and criterion. Unresolved or unclear timer properties were excluded from the respective results. Furthermore, exclusion of specific peripherals (e.g. watchdog timers) is possible.

Each analyzed property features an identifier 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 bit-width). All evaluated inter-platform properties are depicted in Table 1 and discussed in Section 4.3.

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

4.3 Analysis Results

Cross-platform findings from our conducted MCU timer hardware analysis are listed in Table 1 and get discussed in the following. It should be noted that the total number of platforms and timer types can vary between properties due to the respectively applied selection criterion. To cope with this we give the exact matched amounts as well as percentages for each depict result. Moreover, a detailed description of our applied data collection and evaluation methodology can be found in the Sections 4.2.4 and 4.2.5.

4.3.1 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-/underflow happens. The less frequent such events happen, the less timer maintenance and wakeups are required. Therefore, a large counter width is desirable. `R-01` shows that the minimum register size among all platforms is 16 bit. A total of 85 % even provide 32-bit timers while only 34 % offer timers featuring a width of at least 64 bit. Especially beneficial to small 16-bit timers is the possibility of extending their counter range through the use of timer chaining if available. `R-04` shows that on 67 % of all platforms which offer one or more 16-bit timers, these small timers can be extended to a range of at least 32 bit. Though it is noteworthy, that the usage of timer chaining comes at the expense of sacrificing an additional timer module. We nonetheless conclude that timer chaining shall be utilized, especially when working with small range timers only.

Prescalers allow to dynamically reduce the clock frequency fed into the timer, thereby counting only every n-th pulse of the base clock. But, increasing the maximum timeout period using a prescaler comes at the cost of a reduced timer resolution. Nonetheless, this feature is useful, in particular when dealing with small counter widths ($\leq$ 16 bit) and long timeout periods. However, this trade-off dictates that the separation of short high-precision delays and long lasting timeouts is a necessity for a timer driver. Prescalers are available on all platforms in general as well as on 75 % of all analyzed peripherals, as indicated by `R-03`. The only platform that has non-prescalable general-purpose timers is the SiFive FE310-Gx (see Table 14), which instead features a 64-bit counter register and thereby eliminates the need for an additional prescaler.

4.3.2 Auto-reload. Timers often need to produce either periodic events or timeouts that are longer than the maximum time before a counter over-/underflow happens. Hence, restarting the timer is required. To prevent the missing of clock pulses and to reduce maintenance overhead, reloading is handled directly be 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-/underflow while others allow to specify an arbitrary value at which the counter reloads. The latter is either done by sacrificing one compare channel (32 %) but preferably achieved through the usage of a designated auto-reload

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	13	49	100 %	85 %
			≥ 32	11	21	85 %	36 %
			≥ 64	3	3	34 %	5 %
R-02	Compare channels	Number of available compare channels (Excluding timers w/o compare channels)	≥ 1	13	50	100 %	100 %
			≥ 2	9	32	69 %	64 %
			≥ 4	7	11	54 %	22 %
R-03	Prescaler	Support for prescaling the timer clock	*yes*	13	53	100 %	75 %
R-04	Timer chaining	Support for timer module combination (Excluding watchdogs and RTCs)	R-01 ≤ 16†	6	9	67 %	38 %
			R-01 > 16‡	3	5	23 %	24 %
R-05	Compare interrupts	Unique INTs for each compare channel	*yes*	6	17	46 %	30 %
R-06	Overflow interrupts	Unique INTs for counter over-/underflow (Excluding watchdogs)	*yes*	4	8	31 %	20 %
R-07	Event flags	Availability of status bits for timer events	*yes*	10*	60	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	9	23 %	17 %
			CCM	4	17	31 %	32 %
			ARR	6	27	46 %	51 %
			any	13	53	100 %	100 %
R-09	Low-power clock	Low-power oscillator can be used by timer	*yes*	13	51	100 %	70 %
R-10	Deep-sleep active	Timer operational in lowest MCU power states	*yes*	13	45	100 %	62 %
R-11	GP-timers	Number of available general-purpose timers	= 1	1	-	8 %	-
			≥ 1	12	-	92 %	-
R-12	WDT interrupts	Watchdog generates interrupt prior to reset	*yes*	9	10	69 %	67 %
R-13	Unknown items	Timer has unresolved/unknown properties	*yes*	7	14	54 %	19 %

Table 1: Selected overall results across all 13 analyzed MCU platforms

register (51 %). Having a separate register benefits a timer subsystem by leaving all compare channels usable (see Section 4.3.3).

4.3.3 Compare Channels. Triggering an event at a specific counter value of the timer is possible through compare channels. Each compare channel provides a register which is loaded with an arbitrary value that gets constantly compared to the current counter value by the timer hardware. A match event is generated once the counter value equals the configured threshold. Compare channels are used by timer subsystems to signal expiring timeouts. The more compare channels a hardware timer offers, the wider different pending timeouts can be split across channels (e.g. separating short from long running timers). This flexibility in the mapping of virtual timers to hardware peripherals can be utilized to reduce the overall timer maintenance overhead. At a bare minimum, a single hardware timer is required to provide at least one compare channel in order to be usable by a timing-API. Otherwise active polling of the current counter value would be required. As R-02 shows, all timer modules provide at least one compare channel, while most offer at least either two (64 %) or even four (22 %) channels.

4.3.4 Interrupt Handling and Event Flags. Timer events such as over-/underflow and compare matches can produce interrupts upon occurrence. Corresponding ISRs are used by timer drivers to detect expired timeouts and to execute maintenance tasks. Interrupt handling is strongly dependent on the actual MCU-platform, tough often uniform among chip families from the same manufacturer. For a timer subsystem it is important, whether an exclusive interrupt exists for every event or if multiple events share one common interrupt vector. As both R-05 and R-06 indicate, out of all timers only 20 % provide fully independent overflow and only 30 % offer fully independent compare match interrupts.

If an interrupt vector is mapped to multiple events, additional peripheral status register reads are required to determine the exact cause of the fired interrupt. This introduces an additional layer of indirection, therefore increasing timer latency. Event flags are

*Three platforms excluded due to unknown properties. See Section 4.3.4 for details.
†i.e.: Only counting timers that are chainable and have a maximum width of 16 bit. See Section 4.3.1 for details.
‡i.e.: Only counting timers that are chainable and have a width greater than 16 bit. See Section 4.3.1 for details.

available throughout all platforms, as shown by R-07. Though, it is noteworthy that three platforms, namely Espressif ESP8266 (see Table 6), Espressif ESP32 (see Table 7), and Nordic Semiconductor nRF51x/52x (see Table 13), remain unclear about event status bits in their documentation (see Section 4.4) and were therefore excluded from the total platform count of R-07. Nonetheless, we highly doubt that there is no way to extract such information but were not able to reliably confirm it either.

4.3.5 Low-power Operation and Energy Saving. As with resource constrained devices power-saving operation is crucial, the ability to operate timers of an low-power oscillator is highly important. As R-09 shows, 70 % of all analyzed timer types are able to run on 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.

While we found that all platforms provide at least one timer type which can run of a low-power clock, we further were able to confirm that all platforms offer at least one timer, which is capable of running in even the lowest possible power states and waking the CPU upon event occurrences (see R-10). Among these *always-on* peripherals, real-time-counters and -clocks are most commonly found. Three platforms also provide designated ultra low-power timer peripherals (see Tables 3, 4, and 8), as for example the Cryotimer on the Silicon Labs EFM32/EFR32 platform (see Table 8). Especially when dealing with long timeouts and sleep periods, these timer types allow a massive energy-consumption optimization and should therefore be utilized by a properly designed timer subsystem.

4.3.6 Suitability of Timer Types. Even though we included all types of timer peripherals into our analysis, we are convinced that not every type is applicable for the use in a generic timing system. We would like to stress that in our opinion 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. Even though 67 % of the analyzed watchdogs offer the ability to generate an interrupt before or even instead of performing a reset, as shown by R-12, we still suggest to not repurpose them for the usage in a timer subsystem.

4.3.7 Peripheral Availability. Having a range of timer peripherals to chose from opens up a wide spectrum of optimization possibilities and potential features. Even tough operation principles of general-purpose timers are mostly uniform across all platforms, the provided special purpose timers differ greatly in function. Taking a look at the first, there is only a single platform that guarantees solely one general-purpose timer, namely the SiFive FE310-Gx (see Table 14), while every other platform offers more than one, as indicated by R-11. However, the availability of other timer types varies greatly between manufacturers and even MCU families. Therefore, we conclude that allowing the usage of multiple timer modules as well as the incorporation of platform specific peripherals is a necessity for a well designed generic timer subsystem.

We also found that MCUs exist, that only leave the possibility of multiplexing all virtual software timers onto one single general-purpose hardware timer (see R-11 and R-02). Among the analyzed platforms these are namely the SiFive FE310-Gx (see Table 14), only offering a single general-purpose timer, and the Espressif ESP8266 (see Table 6), only providing one compare channel while leaving the alarm functionality of the RTC unclear. In addition to the above concluded requirement for simultaneous utilization of multiple timer peripherals, we further reason that a timer subsystem therefore also must be flexible enough to cope with situations in which only a single hardware timer is available.

4.3.8 Further Considerations. As already addressed in the above sections, not all properties of each analyzed timer type could be determined with sufficient confidence, based on the available documentation or alternative information sources, as described in Section 4.2.1. As shown by R-13, a total of seven platforms still suffer unresolved properties. Five of these platforms can more precisely be grouped into the two Espressif MCUs (see Table 6 and 7) and three of the Cortex-M based platforms, namely the Microchip / Atmel SAMD21 (see Table 5) and the Silicon Labs MCUs (see Table 8 and 9). Espressif devices, especially the ESP8266, leave a lot of open questions in regards to our analysis criteria of which only some could be resolved by inspecting the manufacturer provided SDKs. In contrast, the mentioned Cortex-M based MCUs solely leave the SysTick timer, commonly found across Cortex-M devices, mostly or completely undocumented. However, we suspect them to be very similar to what can be found in other Cortex-M based devices but are unable to confirm it at the time of writing.

4.4 Outstanding Tasks & Issues

Even though our conducted 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 awareness and knowledge, highlighted in this section.

4.4.1 Remaining MCU-platforms and Open Questions. While we already evaluated most of the MCUs that are supported by RIOT-OS at the time of writing, remaining platforms need to be included into our analysis. Since we carefully picked the analyzed MCUs we do not expect fundamentally different results from the remaining platforms. However, we still want to prevent ignoring any abnormalities or missing out important insights these platforms might bring. Furthermore, currently still unknown properties, as described in the previous section, shall be resolved.

4.4.2 Clock Properties and Implications. Despite being an important information for a timer subsystem, determining both the maximum resolution (i.e. shortest possible timeout) and the longest possible timeout is not feasible for a whole class of timers due to the strong dependence on the MCU's oscillator frequencies. As the system clocks depend both on the used 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 that has not yet been analyzed to the required extent are the different clocks each timer peripheral is able to run of. As 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. We expect promising results from a more detailed analysis of the clock trees, especially with respect to power-saving optimizations.

4.4.3 Peripheral Interconnect and Event Systems. Some MCUs offer the ability to route internal signals directly between components via a peripheral interconnect bus or automatically trigger specific actions based on events generated by other peripherals, hereby eliminating the need to execute a designated ISR on the CPU. Such systems might prove as 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 directly inside the respective peripherals.

4.4.4 Configuration and Maintenance Costs. Currently not part of our analysis scope, but not being less important, are the costs a read, update, or reconfiguration of a timer peripheral implies. To give an example: it might 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. Hereby, frequent maintenance tasks might drastically increase power-consumption since the high-power oscillator has to be started during every maintenance period. Having information on such costs could benefit the timer peripheral selection.

5 FUTURE WORK

Completing outstanding tasks and resolving current issues, as described in Section 4.4, are next on our agenda. Nonetheless, further work has to be done in order to develop a new clean-slate timer-API for RIOT-OS, that we can carefully design based on the findings of our work. A coarse overview of our next steps towards this goal is described in this section.

5.0.1 Abstract Timer Classes. As a result of our analysis, we found that, apart from general-purpose modules, timer peripherals vary greatly in function and availability between manufacturers. To cope with this diversity we propose to define abstract timer classes, each describing a distinctive set of features a hardware timer must offer to fall into the respective category. Well-considered definition of appropriate categories has to be done. However, 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.

5.0.2 Availability Analysis. Strongly coupled 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 application-appropriate MCUs, it moreover allows us to estimate the number of platforms that would potentially benefit from different design decisions of the timer-API.

5.0.3 Deriving Requirements. Incorporating all of the above described information, deriving requirements for the aspired timer subsystem is a next step. Here, considering proposed methods and avoiding documented pitfalls from related work, as showcased in Section 3.2, is 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 timer subsystem can then be implemented and tested with a subset of the available platforms.

6 CONCLUSION

With this work our contribution was twofold. We first reviewed related work, depicting both timer hardware and software design considerations. Among these, valuable implementation techniques and common pitfalls that are to be avoided were highlighted. Second, we conducted a large-scale timer hardware analysis covering many of the MCU-platforms that are currently supported by RIOT-OS. We hereby provided detailed information about every timer peripheral type that is available on the targeted platforms and further derived inter-MCU-platform findings from it. Analysis results were then discussed with respect to the development of a new clean-slate timer subsystem for RIOT-OS. Furthermore, currently still outstanding tasks as well as open issues of the conducted hardware platform analysis were outlined and future steps towards the aspired timer-API were defined.

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] Marian Mitescu and Ioan Susnea. 2005. *Using the MCU Timers*. Springer Berlin Heidelberg, Berlin, Heidelberg, pages 67–91. https://doi.org/10.1007/3-540-28308-0_6

[16] 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

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

[19] 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.

[20] 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 (ExpCS '07)*. Association for Computing Machinery, New York, NY, USA, pages 4–es. https://doi.org/10.1145/1281700.1281704

[21] 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.

[22] 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

[23] 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 External CLK. Possibility of clocking the timer module with an external oscillator. External clock sources that need to be routed through additional timer peripherals (e.g. external CLK usage through periodic RTC pulses) count as not available. Can be one of the following:

× Timer cannot be clocked by external oscillator.
✓ External clock source can be used for the timer.

A.1.13 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.1.14 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.1.15 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	External CLK	Low-power CLK	Deep-sleep Active
General-purpose	16 bit 32 bit	1-4	R	2^{16}	✓	○	○	✓	✓	✓	✓	×	×
Advanced-control	16 bit	4, 6	R	2^{16}	✓	○	○	✓	✓	✓	✓	×	×
Basic	16 bit	0	R	2^{16}	✓	×	○	✓	✓	×	×	×	×
Low-power	16 bit	1	E	2^{7}	×	○	×	✓	✓	✓	✓	✓	×
SysTick	24 bit	0	F	2^{3}	×	×	✓[b]	✓	✓	×	×	×	×
Real-time-clock	-	1-2[c]	R[e]	2^{7+15}	×	○	○[d]	✓	-	×	×	✓	✓
Independent WDG	12 bit	0	E	2^{8}	×	×	×	-	×	×	×	✓[e]	✓
System window WDG	7 bit	0	E	2^{12+3}	×	×	×	-	×	×	×	×	×

Table 2: Timer Comparison Matrix: STMicroelectronics STM32

[a] When enabled
[b] System Tick Interrupt
[c] RTC Alarm(s)
[d] From periodic wakeup timer
[e] Independent oscillator
[f] For internal calibration only
[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
[m] Only available in RTC-mode with external clock
[n] Supports masking of individual bits

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

Table 3: 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	External CLK	Low-power CLK	Deep-sleep Active
General-purpose	16 bit 32 bit[g]	1	F	2^{8}	✓	✓	×	✓	✓	×	✓	×	×
Asynchronous	16 bit	1	F	2^{8}	×	✓	×	✓	✓	×	✓	✓	✓
Real-time-clock	-	1	×	-	×	✓	×	✓	-	×	✓	✓	✓
Watchdog	25 bit	0	E	2^{20}	×	×	✓	✓	×	×	×	✓	✓

Table 4: 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	External CLK	Low-power CLK	Deep-sleep Active
General-purpose	8 bit 16 bit 32 bit[g]	2	F	2^{10}	✓	○	○	✓	✓	✓	✓	✓	✓
General-purpose for Control	16 bit 24 bit	4	F	2^{10}	×[k]	○	○	✓	✓	✓	✓	✓	✓
SysTick[j]	24 bit	0	?	?	×	×	✓[b]	?	✓	×	?	?	?
Real-time-counter	32 bit	1	E	2^{10}	×	○	○	✓	✓	×	✓	✓	✓
Watchdog	-	2	-	-	-	○	-	✓	-	×	✓	✓	✓

Table 5: 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	External CLK	Low-power CLK	Deep-sleep Active
General-purpose (FRC1)	23 bit	0	F	2^8	×	×	✓	?	○	✓	×	×	×
General-purpose (FRC2)	32 bit	1	F	2^8	×	✓	×	?	○	×	×	×	×
Real-time-clock	32 bit	?	×	-	×	?	?	?	-	×	×	✓	✓
Watchdog	-	0	×	-	×	×	×	-	×	×	×	?	?

Table 6: 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	External CLK	Low-power CLK	Deep-sleep Active
General-purpose	64 bit	1	R	2^{16}	×	✓	×	?	□	×	×	×	×
Real-time-clock	48 bit	1	×	-	×	✓	×	?	-	×	✓	✓	✓
Watchdog	32 bit	0	×	-	×	×	✓	?	×	×	×	✓	✓

Table 7: Timer Comparison Matrix: Espressif ESP32

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

Table 8: 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	External CLK	Low-power CLK	Deep-sleep Active
General-purpose	16 bit	3	E	2^{10}	✓	○	○	✓	✓	✓	✓	×	×
Low-energy	16 bit[i]	2	E	2^{15}	×	○	○	✓	□	✓	✓	✓	×
SysTick[j]	24 bit	?	?	?	?	?	?	?	?	?	?	?	?
Real-time-counter	24 bit	2	E	2^{15}	×	○	○	✓	□	×	✓	✓	×
Backup Real-time-counter	32 bit	1	E	2^{7}	×	○	×	✓	✓	×	✓	✓	✓
Watchdog	-	0	F	2^{18}	×	○	×	✓	✓	×	✓	✓	×

Table 9: Timer Comparison Matrix: Silicon Labs EZR32

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

Table 10: 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	External CLK	Low-power CLK	Deep-sleep Active
General-purpose (Timer A)	16 bit	2-3	E	2^{3}	×	○	○	✓	□	✓	✓	✓	✓
General-purpose (Timer B)	8 bit 10 bit 12 bit 16 bit	3, 7	E	2^{3}	×	○	○	✓	□	✓	✓	✓	✓
Watchdog	16 bit	0	×	-	×	×	✓	✓	×	×	✓	✓	✓

Table 11: 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	External CLK	Low-power CLK	Deep-sleep Active
General-purpose	32 bit	4	R	2^{16}	×	□	×	✓	□	×	×	×	×
Repetitive Interrupt	32 bit	1^{n}	×	-	×	✓	×	✓	□	×	×	×	×
SysTick	24 bit	1	×	-	×	✓	×	✓	□	×	✓	?	?
Real-time-clock	-	2	×	-	×	□	×	✓	-	×	✓	✓	✓
Watchdog	32 bit	0	F	2^{2}	×	-	×	-	-	×	×	✓	✓

Table 12: 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	External CLK	Low-power CLK	Deep-sleep Active
General-purpose	8 bit 16 bit 24 bit 32 bit	4	E	2^{9}	×	○	×	?	○	×	×	×	×
Real-time-counter	24 bit	4	R	2^{12}	×	○	○	?	○	×	✓	✓	✓
Watchdog	32 bit	0	×	-	×	×	×	?	✓	×	×	✓	✓

Table 13: 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	External CLK	Low-power CLK	Deep-sleep Active
Machine timer	64 bit	1	$\times^{h}$	-	×	✓	×	✓	○	×	×	✓	✓
Real-time-counter	≥48 bit	1	E	2^{15}	×	✓	×	✓	-	×	✓	✓	✓
Watchdog	31 bit	1	E	2^{15}	×	✓	×	✓	□	×	×	✓	✓

Table 14: Timer Comparison Matrix: SiFive FE310-Gx