HAW HAMBURG

MASTERTHESIS
Niels Gandraß

A Uniform API for Cross-Platform Timer Hardware Abstraction on Resource Constrained IoT Devices

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE
WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Niels Gandraß

A Uniform API for Cross-Platform Timer Hardware Abstraction on Resource Constrained IoT Devices

Masterarbeit eingereicht im Rahmen der Masterprüfung
im Studiengang *Master of Science Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Thomas C. Schmidt
Zweitgutachter: Prof. Dr. Franz Korf

Eingereicht am: 23. Februar 2022

Niels Gandraß

Thema der Arbeit

A Uniform API for Cross-Platform Timer Hardware Abstraction on Resource Constrained IoT Devices

Stichworte

Betriebssysteme, Eingebettete Systeme, Hardwareabstraktion, Hardware Timer, Ressourcenbeschränkte Geräte, Softwaredesign

Kurzzusammenfassung

Timer sind Bestandteil jedes Mikrocontrollers und beeinflussen zahlreiche Aspekte eingebetteter Systeme. Ihre wachsende Vielfalt lässt sich jedoch nur noch unzureichend über die historisch gewachsenen APIs eingebetteter Betriebssysteme abbilden. Ein solider und zukunftsorientierter Ersatz für die aktuelle Hardware-Abstraktion ist daher notwendig.

Diese Arbeit bietet einen umfassenden Einblick in heutige Timer-Peripherie. Wir analysieren die Timer von 43 Gerätefamilien und acht Herstellern, charakterisieren Timeouts für typische IoT Anwendungen, beleuchten die Timer APIs eingebetteter Betriebssysteme, identifizieren Mängel aktueller Lösungen und diskutieren Ansätze zur Hardware-Abstraktion. Auf dieser Wissensbasis entwickeln wir ein zweischichtiges Low-level Timer API Design und implementieren es für das RIOT Betriebssystem. *uTimer* ermöglicht die transparente Nutzung unterschiedlicher Timer-Typen über eine gemeinsame Schnittstelle. Diese ist plattformunabhängig nutzbar und stellt grundlegende sowie gerätespezifische Funktionen zur Verfügung. Timer und Kanäle werden im Durchschnitt verdreifacht und neue Timer-Typen, wie beispielsweise *ultra low-power timer*, nativ unterstützt. Die Vereinheitlichung aktueller APIs sorgt darüber hinaus für konsistenten und gut wartbaren Timer-Code innerhalb des gesamten RIOT-Ökosystems.

Wir entwickeln plattformunabhängige Unit-Tests sowie Benchmarks und führen diese auf insgesamt 14 Mikrocontrollern in einem HiL-Testbett mit CI Unterstützung aus. Die vorgestellte *uTimer* API benötigt für ihre zusätzliche Abstraktion lediglich zwischen 6 und 21 CPU-Zyklen. Dies entspricht einem Timeout-Fehler von weniger als 0.05 %. Diesem steht die Nutzbarkeit zahlreicher neuer Timer-Funktionen gegenüber. Verzichtet man für spezielle Anwendungsfälle auf die Abstraktion von Timer-Typen, so kann jeglicher API-Overhead vollständig vermieden werden. Die automatisierte Ausführung von Tests und Benchmarks auf der aktuellen Code-Basis sowie für alle bevorstehenden Änderungen erleichtert außerdem die Entwicklung nachhaltiger Peripherie-Treiber und ermöglicht deren fundierte Bewertung.

Niels Gandraß

Title of Thesis

A Uniform API for Cross-Platform Timer Hardware Abstraction on Resource Constrained IoT Devices

Keywords

embedded systems, hardware abstraction, hardware timers, operating systems, resource constrained devices, software design

Abstract

Timers are part of every microcontroller and influence numerous aspects of embedded applications. As the heterogeneous spectrum of timer peripherals continues to grow, embedded OSs increasingly struggle to expose novel timers and advanced features via their time-honoured APIs. The need for comprehensive and future-proof hardware abstraction therefore arises.

This work provides detailed insight into modern timer peripherals. We conduct a survey of timer hardware across 43 device families from eight manufacturers, characterize typical timeouts within IoT applications, examine timer APIs of popular embedded OSs, identify deficiencies of current solutions, and discuss different approaches to hardware abstraction. This forms a solid base, on which a two-layered low-level timer API design is proposed and implemented for the RIOT OS. Its uniform interface allows transparently interchangeable use of all hardware timers. Generic and peripheral-specific features are exposed, yet application portability is preserved. On average, available timers and channels are tripled, and previously unexposed advanced timer types such as ultra-low power timers become usable. Current interfaces are streamlined to form a single API, fostering uniformity and maintainability of timer code throughout the whole RIOT ecosystem.

We develop platform-independent timer unit tests and benchmark suites that are executed using a HiL testbed with CI support. The performance of our novel *uTimer* API is assessed on 14 MCUs. It only adds between six and 21 CPU cycles of abstraction overhead compared to existing solutions. This corresponds to timeout errors of less than 0.05 % while numerous modern timer features are made available to user applications and high-level OS modules. For high-performance use, deliberate renunciation of timer-type transparency eliminates any performance differences between APIs at the cost of narrowing application portability. Automated execution of the contributed unit tests and benchmarks on nightly builds and pull requests moreover facilitates the development of sustainable implementations and permits their sound evaluation.

Contents

List of Figures

List of Tables

List of Acronyms

API Application Programming Interface.

BLE Bluetooth Low Energy.

CAN Controller Area Network.

CI Continuous Integration.

CLK Clock.

CMP Compare Match.

CoAP Constrained Application Protocol.

CPU Central Processing Unit.

DHCP Dynamic Host Configuration Protocol.

DMA Direct Memory Access.

DUT Device Under Test.

GPIO General-purpose input/output.

HAL Hardware Abstraction Layer.

hAPI Hardware-facing API.

HiL Hardware in the Loop.

INT Interrupt.

IoT Internet of Things.

IPv6 Internet Protocol version 6.

IRQ Interrupt Request.

ISR Interrupt Service Routine.

LoRaWAN Long Range Wide Area Network.

LSB Least Significant Bit.

MCU Microcontroller Unit.

MQTT Message Queuing Telemetry Transport.

MSB Most Significant Bit.

NOP No Operation.

OS Operating System.

OVF Overflow.

PHiLIP Primitive Hardware in the Loop Integration Product.

PLL Phase-locked Loop.

PWM Pulse-width Modulation.

RAM Random-access Memory.

RF Robot Framework.

ROM Read-only Memory.

RPL Routing Protocol for Low-Power and Lossy Networks.

RTC Real-time clock.

RTOS Real-time Operating System.

SDK Software Development Kit.

SDLC Systems Development Life Cycle.

SoC System on a Chip.

TCM Timer Comparison Matrix.

TCP Transmission Control Protocol.

uAPI User-facing API.

UART Universal Asynchronous Receiver-Transmitter.

UDP User Datagram Protocol.

WDG Watchdog Timer.

WSN Wireless Sensor Network.

List of Symbols

Symbol	Notation	Description
D_{cb}	Callback Execution Delay	Time between timeout expiry and user-callback execution
J_{cb}	Callback Execution Jitter	Standard deviation of D_{cb}
f	Frequency	Number of timer cycles per second
O_{GPIO}	GPIO Overhead	Time required to signal start and stop of a measurement
p	Padding	Number of bytes for aligning data within memory
s	Size	Number of bytes required for storing a data block
σ	Standard Deviation	Statistical measure for the dispersion of data
τ	Timeout	Length of a timeout period
E_τ	Timeout Error	L_τ in proportion to the timeout length τ
J_τ	Timeout Jitter	Standard deviation of L_τ
L_τ	Timeout Latency	Difference between expected and actual end of a timeout

List of Listings

1 Introduction

With the ubiquitous Internet of Things (IoT), applications and deployment contexts of embedded systems vastly increased in number and heterogeneity. Devices are often severely constrained in terms of computational power, memory, and energy consumption. Hardware manufacturers respond to this by continuously improving their microcontroller units (MCUs), including CPUs and on-board peripherals. Timers in particular are required by almost every application and have great influence on the total energy consumption, wherefore an ever-increasing variety of them exists. Firmware developers must adapt to this frequent changes in hardware accordingly. Embedded operating systems conveniently abstract from the underlying hardware and therefore became the prevalent solution for developing sustainable applications in the IoT. They are expected to provide portable yet feature-rich hardware interfaces, carefully weighing generalization against its inevitable performance degradation. However, current operating systems struggle in doing so due to their inflexible APIs and the high heterogeneity of target devices. Their long unchanged function definitions are limited to basic timer features, hence prevent exposure of advanced timer features and innovative timer types. The need for sound and flexible low-level peripheral abstraction therefore arises to meet the requirements of embedded applications.

RIOT [3] is an increasingly popular open-source operating system that targets low-power and resource constrained devices in the IoT and Wireless Sensor Networks. It provides five distinct low-level timer APIs, distinguished by exposed timer types and features. We aspire to unify the existing modules into a single streamlined interface. It shall comprehensively present novel hardware timers including all their functions, while preserving application portability. Our goal is not to provide a fully-featured and ready-to-use implementation, but rather a sound and future-proof low-level timer API design. The major contributions of this work are therefore primarily of analytical and conceptual nature. We give profound insight into the variety of hardware timers and how they are used within embedded applications, to base a reliable API design on. Since RIOT targets more than 200 MCU boards, a solid interface must not only be able to expose this hardware diversity, but the numerous device-specific drivers must likewise perform correctly and resource efficiently. Automated cross-platform unit tests and performance benchmarks encourage the development of sustainable implementations and permit their sound evaluation. The contributions of our work are highlighted in the remainder of this section.

Chapter 2 introduces fundamental concepts and techniques that are encountered throughout this thesis. It explains the basic components that make up a hardware timer and how they interact with each other. Abstraction of on-board MCU peripherals and embedded operating systems are moreover highlighted. The manifold challenges that arise when designing platform-independent hardware interfaces for a heterogeneous range of devices are outlined in our problem statement in Chapter 3. We address the limits of platform-specific solutions and the difficulties that come with platform-agnostic ones. Hardware diversity, functional validation, and performance optimization are likewise discussed.

In Chapter 4, a total of four analyses are conducted. First, a review of related work on low-level timer properties, hardware abstraction techniques, and timer modules of embedded operating systems is performed. The demands of high-level timer subsystems for peripheral interfaces are furthermore highlighted. Second, we take a closer look at the current low-level timer APIs of RIOT, identifying similarities and potential shortcomings. Third, we contribute a large-scale analysis of all timer peripherals that are found on RIOT supported boards. Results from those 43 MCU families are compiled into a comprehensive overview and discussed in a comparative fashion. Fourth, timing requirements of IoT applications are determined by examining timer usage within the whole RIOT code base as well as prevalent network communication protocols. We use the insights from all four analyses within Chapter 5 to base the design of our low-level timer API on them. The proposed *uTimer* API consists of two distinct layers that allow transparently interchangeable use of all available timer peripherals, provide out-of-the-box support for both generic and device-specific functions, and preserve platform-independence whenever possible. Besides the design itself, we discuss its trade-offs and remaining issues. Within Chapter 6, uTimer is implemented for six distinct microcontroller platforms, covering a total of 129 boards. We describe both its generic and platform-specific components, as well as their integration into the RIOT system architecture. The correct behavior of implementations is subsequently verified by timer unit test suites across all platforms.

Our design is evaluated within Chapter 7. Extensive benchmark suites are developed to assess the manifold aspects of low-level timer drivers in an automated fashion. The performance of uTimer is compared to existing APIs on 14 different MCUs. We reveal the impact of individual design decisions and isolate the overhead that is inherent to the APIs themselves. This is followed by an evaluation of various qualitative aspects and the suitability for integration into high-level timer subsystems. The impact of the contributed benchmark suites on the operating systems development life cycle is furthermore discussed there. Chapter 8 then closes this thesis with a concluding summary of our contributions and gives an outlook on future work.

2 Background

This chapter introduces basic concepts and techniques that are used in this thesis. Section 2.1 starts by explaining what microcontrollers are and how their basic timer peripherals work. It is followed by a description of hardware abstraction techniques in Section 2.2 and a definition of embedded operating systems, including their purpose and goals,in Section 2.3. This chapter provides only an initial overview, as all aspects touched upon will be covered in detail within our extensive analysis Chapter 4.

2.1 Microcontrollers and Timer Peripherals

A microcontroller unit (MCU) is an integrated circuit often found in embedded systems. It combines a central processing unit (CPU), integrated memory, and peripheral devices within a single chip package. In addition to system memory, it typically likewise includes read-only memory (ROM) and random-access memory (RAM). Clock generators and internal oscillators are moreover found in most devices. Today, a broad spectrum of MCUs is offered to cover a wide range of possible applications. These differ in terms of computing power, energy consumption, and on-board peripherals.

Among the manifold peripherals that are available, hardware timers are one of the few that are essential to all embedded devices [29]. They are used to measure time and generate periodic signals independent of the main program flow. The CPU hereby can primarily execute the deployed firmware and is only rarely interrupted for timekeeping tasks. This is indispensable for most embedded applications. Hardware timers, also referred to as *low-level timers*, differ from software timers in that they are always physical hardware modules and are not generated by the device firmware based on a purely virtual clock source.

Components of a basic timer peripheral are illustrated in Figure 2.1. Hardware timers count pulses of an independent clock or external events within an internal counter register. It is either incremented or decremented at a fixed frequency until a threshold is reached. This can either be the maximum counter width or an arbitrary value, specified within a designated auto-reload register. Once reached, the counter value is reset, an overflow is signalled, and the timer may continue to count. Compare channels allow measuring time by triggering system events, whenever

the counter register reaches a certain value. They hereby request the CPU to interrupt the main program flow, enabling user applications to asynchronously react to elapsed timeouts. Counting speed is determined by the selected timebase, i.e., clock source, and can be altered by a prescaler or phase-locked loop (PLL). All functions of a timer module can be configured through its control registers. Peripheral status is likewise exposed and includes various information, such as pending interrupts or current counting mode.

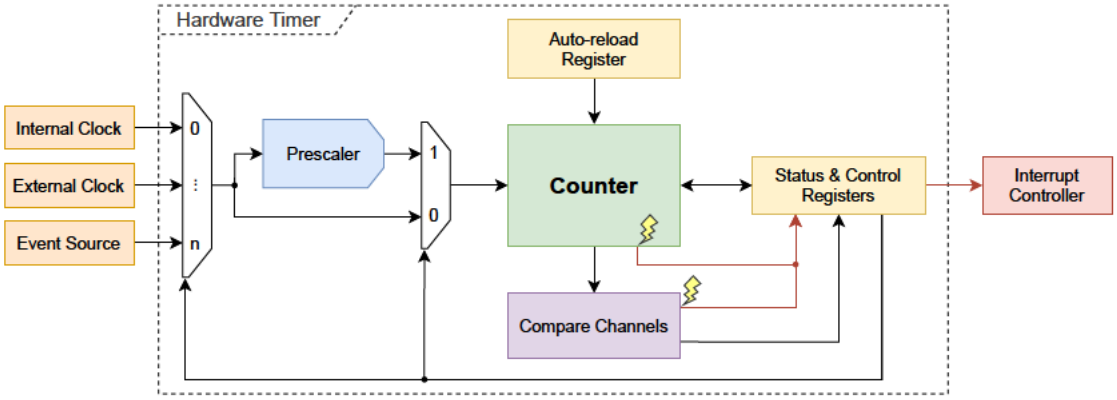


Figure 2.1: Core components and functional overview of a basic hardware timer. Interrupt event sources are indicated by flash symbols.

Today, various types of specialized hardware timers are offered alongside the ubiquitous general-purpose modules described above. These include among others: high-speed timers, low-power timers, and real-time counters. Although each type possesses a distinct feature set, they all share the same basic concepts of operation. Firmware developers can therefore select the most suitable peripherals from a wide range of timers and use either solely its basic functions or also advanced timer-type specific features, whenever required for the current use case. In addition to the huge variety of on-board peripherals, external timekeeping devices[1] exist, but are used far less frequently. They become an attractive energy-saving solution whenever very long-running timeouts ($> 10\,\mathrm{s}$) are employed. Such external devices allow to completely power down the MCU during timeout periods and only consume ultra low currents ($< 50\,\mathrm{nA}$) themselves.

2.2 Hardware Abstraction Techniques

All hardware peripherals within MCUs are usually controlled via designated registers. Those registers are read to determine device status or to receive application data. Writing them allows to control and configure on-board devices or triggers specific hardware functions. Whenever accessing peripheral registers, the CPU must know their exact memory address and the firmware

[1] For example, the Texas Instruments TPL5111 ultra low power system timer (35 nA) for power gating in duty cycled applications. See: `https://www.ti.com/product/TPL5111` (Accessed: 04.01.2022)

developer must interpret bit values correctly. This direct form of register access is not only highly error-prone and cumbersome, but it also ties an application firmware to the target hardware. To aid this problem, hardware is abstracted by so-called hardware abstraction layers (HALs). They enable the user application to access hardware through a well-defined application programming interface (API). Device manufacturers always provide at least a basic register mapping, while some vendors even offer a rich function-based HAL. Register mappings are usually made available in the form of header files that contain preprocessor macros. These often define custom data types that in turn are used to model peripheral registers as well as complete peripheral devices. Function-based APIs are built on top of these mappings and are commonly shipped within vendor-specific software development kits (SDKs).

Besides such basic forms of hardware abstraction, there may be additional, more complex layers. In general, lower layers are hardware-dependent up to a certain point and higher layers do get increasingly independent of the underlying hardware while hiding more and more details of low-level implementations. Well-designed APIs make up the interfaces between two adjacent abstraction layers.

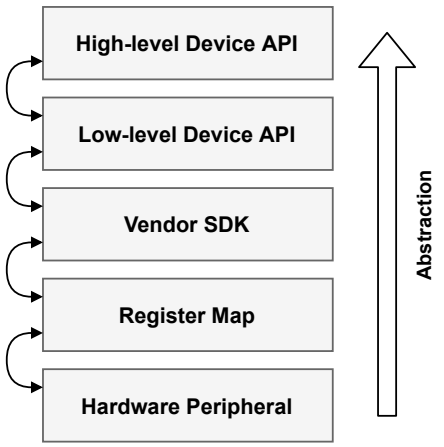


Figure 2.2: Example of a highly abstracted hardware peripheral. Each box represents one layer of abstraction.

An example of a multi-layered hardware abstraction is illustrated in Figure 2.2. Here, registers are first mapped to their memory addresses and peripherals are then exposed via a vendor-provided SDK. This SDK in turn is used by a low-level device API to provide the application with performant and user-friendly access to all peripheral features. Finally, a high-level device API eliminates hardware-dependence entirely and provides additional features that go beyond pure hardware presentation. User applications that exclusively use this high-level device API can be run on any target hardware that implements the low-level device API. However, this large amount of abstraction imposes a notable computational overhead and limits access to special hardware-specific features, which potentially makes the API unsuitable for some applications.

2.3 Embedded Operating Systems

Operating systems (OSs) are collections of reusable software components that interface device hardware, manage system resources, and provide basic services to user applications. Embedded OSs make up a subset of these and are designed to run on resource constrained hardware. The targeted devices can be limited in multiple aspects, including processing speed, system memory, and allowed power consumption [31]. Operating systems for embedded devices therefore place a strong emphasis on resource efficiency and offer only a very limited set of features compared to general-purpose operating systems. Developers are relieved from repeatedly implementing hardware-specific driver code and provided with extensively tested solutions to common software problems. These typically include memory management, task scheduling, or hardware abstraction and are complemented with implementations for various communication protocols. Application code can often be written in a portable and platform-agnostic fashion, allowing for quick changes in target hardware. Embedded OSs hereby greatly reduce time-to-market and can improve code quality. The application and the operating system are usually statically linked together into a single firmware binary, effectively limiting the system to run only a single application. As a consequence, the OS generally dictates the set of usable programming languages, as for example the nesC dialect used for writing TinyOS [56] application.

There is a wide variety of application-specific operating systems, of which real-time operating systems (RTOSs), such as QNX [8] or FreeRTOS [6], are by far the prevalent type. However, a sharp distinction between different OS types is not always possible, as nowadays most embedded OSs fall into several categories. RIOT [3], for example, is an open-source OS specifically designed for resource constrained devices in the Internet of Things. It offers real-time capabilities and implements full multi-threading. The microkernel architecture of RIOT allows out-of-the-box support for a comprehensive list of features while keeping the minimum memory requirements low. User applications can be written in the C, C++, or Rust programming languages. We will go into detail about various aspects of the RIOT operating system throughout the body of this work.

3 Problem Statement

The challenges with designing extensive yet flexible application programming interfaces for a broad and heterogeneous spectrum of target devices are manifold. In this chapter we describe the different aspects to this problem with a focus on timer peripherals.

Hardware timers are part of every microcontroller and essential to almost all applications. Due to their vast use cases, they affect numerous aspects of embedded systems [29]. These include energy consumption, responsiveness, data transmission, and system performance in general. With the tight resource constrains that are put upon devices within the Internet of Things or Wireless Sensor Networks, their comprehensive yet efficient use is of utmost importance [17]. Although developing hardware APIs may seem simple at first, it quickly becomes challenging when different types of peripherals shall be exposed via a uniform interface [22]. Because of the far-reaching influence of timer peripherals, we nevertheless consider it essential to address this problem in a sound manner. Updating nowadays outdated and insufficient APIs to support advanced timer features and optimizing low-level driver implementations brings numerous benefits for modern embedded applications.

3.1 Limits of Platform-specific Solutions

Hardware APIs present peripherals to user applications and other system modules in a convenient and user-friendly way. Whenever only a single peripheral needs to be exposed, the associated software interface can be easily tailored to it. Truly platform-specific solutions do not distinguish between different types within a single class of peripherals, such as basic and advanced timers. This approach is perfectly sufficient, as long as only a single microcontroller is targeted and future application portability is no concern. API design, however, quickly becomes challenging when multiple yet still mostly homogeneous devices, e.g., MCUs of the same family, are targeted. Even though their peripherals are almost identical in operation principles and features, some may require a slightly modified driver or do not support a specific feature. With a large and highly heterogeneous range of target devices, this problem intensifies and can no longer be solved by any platform-specific solution. In such cases, differentiating between target platforms within low-level peripheral layers becomes inevitable [3].

Well-designed hardware APIs nonetheless should bundle peripherals of the same class behind a single interface. Fully platform-specific solutions struggle at implementing this due to their inflexibility and lack of abstraction. Platform-agnostic APIs instead are capable of modeling these peripheral differences. They achieve this by providing multiple implementation variants that possess a common interface. Whenever the target device is changed, the respective driver for the new device is automatically loaded. Since all drivers share the same interface, user applications remain fully compatible and therefore do not need to be manually modified. This approach, however, tends to limit available functions to the basic set of features that is common to all target devices [30]. While this potentially results in simple and streamlined application programming interfaces, features that are only supported by a subset of the platforms are hereby often left unexposed. With growing peripheral diversity, this not only increasingly limits the power of the API, but also prevents most optimization opportunities that MCUs nowadays offer.

Embedded operating systems are designed to provide a sound base that user applications can build upon [3]. They support a large range of devices and allow for application portability to a certain extent. Fully platform-specific hardware APIs therefore are generally considered unsuitable for embedded OSs.

3.2 The Abstraction Trade-off

In the previous section we pointed out that an increase in application portability, i.e., platform compatibility, is usually linked to an increase in abstraction. However, each additional software layer entails further computational and memory overhead [22]. Containing the computational overhead is particularly important with timekeeping applications in order to maintain timeout accuracy and to ensure overall system responsiveness. The effects on memory consumption, on the other hand, are less important, but also not to be neglected [51].

This means that operating system developers must carefully weigh the benefits and drawbacks of different API designs. Forcing user applications and OS modules to directly interact with timer hardware registers may yield near-optimal performance but is highly error-prone, laborious, and prevents portability. Abstracting timer peripherals results in portable and user-friendly solutions but decreases performance. Hence, choosing an appropriate level of abstraction can be challenging and is referred to as the *abstraction trade-off* [55]. Sacrificing system memory to reduce computational complexity can alleviate the negative performance impact to a certain extent. Corresponding techniques include, for example, replacing resource-heavy calculations with lookup tables and loop unrolling. This idea was first described by Hellman [24] and is referred to as the *time-memory trade-off*[1].

[1] The *time-memory trade-off* is also referred to as the *space-time trade-off*.

Resource requirements of embedded systems are also important from an economic point of view. Here, the challenge lies in balancing production and development costs. An increased resource consumption may require to upgrade the target hardware. This can lead to significant increases in production costs, as embedded devices are often deployed in large quantities. Complete renunciation of abstraction, on the other hand, skyrockets firmware development costs, dramatically increases time-to-market, and is therefore also undesirable [31].

Next to pure performance aspects stand the capabilities of an API. The more abstraction is applied, on one hand, the more details of the underlying hardware are hidden [22]. With little to no abstraction, on the other hand, solutions fail to support heterogeneous target devices. The current peripheral APIs of embedded OSs therefore tend to be limited to generic features, leaving progressive but device-specific ones unexposed, as confirmed by our review of timer APIs in Section 4.1.4. In order to comprehensively support novel timers while sacrificing neither portability nor compatibility, a careful balance between feature support and generalization must be obtained. The need for modern low-level timer APIs that supersede the existing rigid and functionally limited interfaces therefore arises.

So the problem of abstracting timer hardware lies in carefully balancing the manifold aspects of the outlined trade-offs. A low-level timer API should offer feature-rich and convenient timer use while confining performance degradation and allowing for comprehensive application portability at the same time.

3.3 Hardware Heterogeneity

The great diversity of embedded systems makes the development of generic solutions extremely difficult [31]. With respect to hardware timers, characteristics and features of general-purpose peripherals are well-known and allow for sound generalization. Advanced timer types, on the other hand, differ largely between device families in both capabilities and features.

Hardware manufacturers continuously improve existing peripherals and present new timer types to satisfy the demanding requirements of modern embedded applications. The RIOT OS alone supports more than 200 different microcontroller boards, consisting of 43 device families from eight manufacturers, and new devices steadily expand this range as they become available [48]. Each additional board potentially introduces novel types of timer peripherals. A comprehensive overview of the properties and capabilities of the target hardware does not exist. However, profound knowledge about it is a necessity for designing extensive yet lasting APIs. Accordingly, it is becoming increasingly difficult for OS developers to design streamlined software interfaces that are also able to support the latest hardware features.

3.4 Validation and Optimization

Besides all above discussed aspects, long-term system operability and component availability are likewise important parts of the life cycle of embedded systems [31]. Sound application portability allows for effortless changes in target hardware whenever the current microcontroller is no longer manufactured or cheaper devices become available. This challenges embedded operating systems by expecting user applications to behave identical on all supported devices. In addition, extensive regression testing must be performed to ensure that neither upcoming nor recent changes to the OS codebase cause bugs or major performance degradation within existing implementations.

Unit tests support developers by verifying proper behavior of system modules and low-level code on both new and currently supported devices. With RIOT OS, for example, extensive test suites already assert the functionality of low-level timer drivers. Their periodic execution on a subset of all supported boards greatly reduces the risk of malfunctioning or even fully dysfunctional implementations [58]. Benchmarks that additionally assess the performance of low-level drivers, however, are not yet available [28]. This makes quantifying the impact of software design decisions and pending pull requests on non-functional requirements impossible. Whether or not a certain microcontroller fulfills application requirements therefore cannot easily be determined. The lack of benchmarks not only hinders the development of performant peripheral drivers, but potentially also leaves side effects of imminent code changes undetected. This problem again intensifies with the steadily increasing number of targeted boards. Automated and periodic execution of qualified benchmark suites therefore is almost inevitable for providing reliable and lasting embedded OSs for a constantly evolving spectrum of embedded devices [3].

4 Analysis

A well-founded overview of both hardware and software is essential for creating a knowledge base that ensures a viable and future-proof API design. We therefore conduct four different analyses prior to designing the aspired low-level timer interface, as describe in Chapter 5.

This chapter starts with a survey of related work and embedded operating systems, which are frequently used with resource-constrained IoT devices, in Section 4.1. The timer subsystem of RIOT OS is separately discussed in detail within Section 4.2. Comprehensive insight into the diverse spectrum of timer peripherals is given by our large-scale hardware analysis in Section 4.3. Timer usage within the whole RIOT codebase is examined and findings are extended by timing requirements of network communication protocols in Section 4.4. Each section is individually concluded with a summary of its most important aspects.

4.1 Related Work

Scientific research related to embedded timer peripherals and their associated hardware APIs is presented in this section. It is split into the following four categories:

a) characteristics of timer peripherals from a hardware point-of-view,
b) comparative analyses of timer hardware across different MCUs,
c) generic design aspects and implementation techniques for timer subsystems,
d) properties, similarities, and issues of timer APIs found in popular embedded OSs for resource-constrained devices.

The listed publications were selected according to their anticipated relevance for this work, as estimated to the best of our knowledge.

4.1.1 Properties of Hardware Timers

Common operation principles of general-purpose timers, their basic feature set, and their typical characteristics are described by Kamal [29, pp. 152 – 159]. The author elaborates on available operation and counting modes, different peripheral states, and various timer properties. These

include, among others, counter register width, clock prescaling capability, and auto-reload functionality. We likewise cover these aspects within our analysis of timer hardware, as described in Section 4.3. Common use cases and exemplary applications for general-purpose timers are further depicted. These include measurement of time periods, task scheduling, and baud rate control for communication protocols. Moreover, other types of timing hardware, here namely real-time clocks (RTCs) and watchdog timers (WDGs), are briefly illustrated.

While less detailed than Kamal [29] did, Susnea and Mitescu [54] also give insight into general-purpose timer peripherals. Their book extends the above publication by describing functions and operation principles of timer hardware that is capable of generating pulse-width modulation (PWM) output signals. The authors furthermore identify compare channels as another key component of timer peripherals that we therefore also incorporated into our hardware analysis.

Besides on-board peripherals, Dimitrievski *et al.* [17] highlight the huge energy-saving potentials of external ultra-low power timers. They harness a Texas Instruments TPL511, which consumes less than 35 nA and provides an accuracy of 1 %. Battery lifetime of an IoT sensor node, deployed within a Wireless Sensor Network (WSN), could hereby be extended by a factor of up to 80. This not only shows the importance of energy preservation with resource restricted devices in general, but also showcases the tremendous optimization potential timer peripherals offer.

Reverter [47] uses a hardware timer to autonomously trigger peripheral actions independent of the CPU. This reduces the total number of wake-ups, resulting in a halving of the systems power consumption. He moreover found wake-ups to entail a significant overhead power consumption[1]. Driver support for hardware features that allow to maintain CPU sleep, such as auto-reload or timer chaining, therefore is of utmost importance with resource constrained devices.

4.1.2 Comparative Analyses of Timer Peripherals

Susnea and Mitescu [54, pp. 67-68, pp. 87-89] conducted a comparison of timer peripherals from three MCU families, namely Motorola HC11, Atmel AVR, and Intel 8051. They found that even though specific timer features differ, the assessed peripherals still share many common operation principles. Asynchronously waiting for a specified amount of time, measuring elapsed time, and counting of events are pointed out as predominant use cases. A low-level timer API, as we aspire it, must in particular be capable of the first two.

Sharp *et al.*, members of the TinyOS [56] developer community, compiled an overview of timer hardware during the specification of their timer interfaces. Timer properties of the Atmega128, the MSP430, and the Intel PXA27x MCUs are depicted within their TinyOS Enhancement Proposal #102 Appendix A [52, Ln. 676ff]. The authors found all three platforms to provide several

[1] Overhead power consumption denotes the energy that is used just for transitioning between device power modes.

timer peripherals of at least two different types. Differences in counter width, clocking options, prescaler capabilities, and in the number of available compare channels were emphasized.

Further analyses of microcontroller hardware exist, but they do not cover timer peripherals at the required level of detail to derive interface design decisions from them. We therefore contribute a large-scale analysis of timer hardware ourselves in Section 4.3. It covers all the 43 device families that are supported by RIOT OS at the time of writing.

4.1.3 Hardware Abstraction and Timer Interfaces

The hardware abstraction layers (HALs) in RIOT are described by Baccelli *et al.* [3]. The OS provides a single peripheral layer that exposes hardware to system modules and user applications. This design keeps abstraction overhead low, but limits feature support.

The two-layered design by Kleeberger *et al.* [30] separates register access from peripheral driver code. This allows profound feature support and makes the abstract device drivers independent of the underlying hardware. The authors leverage this additional layer of abstraction to mock hardware peripherals during unit tests.

Handziski *et al.* [22] introduce another level of abstraction in their proposed HAL design. Each of its three layers allows for a different granularity of peripheral access. Hardware-independent APIs are provided, while access to platform-specific features is available at the cost of consciously sacrificing application portability. Developers are able to mix interfaces of different abstraction levels within the same firmware. The authors confirmed the feasibility of their multi-layered HAL by implementing it for a Wireless Sensor Network (WSN) application.

The *Common Microcontroller Software Interface Standard (CMSIS)* [2] is a widely used set of uniform APIs for Cortex-based devices, such as the extensive STM32 device family. It exposes core components, including the *SysTick* timer. Additional timer peripherals, however, remain unsupported by it, since these are not part of the generic Cortex processor family.

Another platform-independent but more extensive timer API is proposed by Lindgren *et al.* [35]. They assess similarities between timers of the STM32F4 and LPC1700 MCU families and discuss their impact on the proposed design. Common timer properties and features that can be exposed through a uniform API are identified. These include counter width, interrupt capability, prescalers, auto-reload, and compare channel count. The authors furthermore emphasize, that the exposure of multiple compare channels significantly benefits timer maintenance.

High-level timer modules heavily depend on low-level timer APIs and the underlying hardware abstraction layers. Even though these modules are beyond our scope, they provide important insight and allow the deviation of some implications for our aspired low-level timer API. A quick survey confirmed the statement of Lindgren *et al.* [35]: Access to multiple compare channels is

an indispensable requirement for many high-level timer modules [19], [25], [26], [43]. Most indeed do make beneficial use of timer features that go beyond the generation of a simple monotonic clock tick [13], [14], [19], [20], [25], [26]. With contemporary applications, driver support for low-power operation modes as well as dynamic clock and power management of peripherals are just as important [7], [41], [50].

4.1.4 Embedded Operating Systems

The timer subsystem of RIOT OS [4] consists of five low-level and two high-level timer modules. Due to its importance for our work, we present the provided features and discuss its different APIs separately and in-depth within Section 4.2.

Within this subsection, we inspect the timer APIs of four popular embedded operating systems that are likewise targeted at low-end embedded devices. More complex and sophisticated solutions, such as QNX [8] or Windows Embedded Compact [42], are not part of this survey. Each OS is discussed individually before a comparative summary concludes our findings.

Contiki / Contiki-NG

Contiki [11] is a cross-platform embedded OS for low-power IoT devices. It is free to use and distributed under the terms of the 3-clause BSD license. While the original Contiki-OS received no updates since 2018, its successor Contiki-NG [12] is still under active development. Both possess a simple `clock` module that provides system time as an unsigned integer, representing clock ticks since system boot. Its API only allows to read the counter value and to actively wait for a number of clock ticks, hereby blocking the CPU. Based on this clock, five different timer APIs exist. Their interfaces and features are similar, but timer behavior and resolution differs.

The `timer` module offers high-resolution short-running timeouts, while `stimer` provides long-running timeouts with one second resolution. Both require the application to actively poll for timer expiry but are safe to use within interrupt contexts. The `etimer` and `ctimer` modules can be used to either post an event to a process or to execute a callback function. Both, however, are not interrupt safe and operate with system tick resolution. Last, `rtimer` can use a separate `clock` instance to provide real-time task execution but might be unavailable when other OS modules are in use. None of the APIs allows direct access to available hardware timer peripherals, solely relying on software multiplexing timeouts onto the central system clock.

FreeRTOS

FreeRTOS [6] is a widely used real-time embedded OS that targets more than 40 MCU families and requires only 9 kBytes of ROM in its minimal configuration. It is actively developed by *Amazon Web Services* and released under the MIT license. The operating system itself only supports virtual software timers that are multiplexed onto a single system clock source. Application developers can extend it by manually writing platform-dependent code to make use of available hardware timers. This, however, closely couples the firmware to the current target hardware what impedes application portability. A single timer API is provided out-of-the-box, offering basic features, such as starting, stopping, deleting, and resetting timers. Each of its function calls exists in two variants that must be used according to the calling context. The API allows changing a timers period even after creation. A callback function can be attached to each timer and is executed by the timer service task upon timeout expiry. All virtual timers can furthermore be named and possess a unique identifier.

TinyOS

TinyOS [34], [56] targets low-power MCUs, lays its focus on wireless communication, and therefore is commonly deployed within WSNs. Applications are written in the nesC dialect and are made up of multiple components that are connected via predefined interfaces. The OS can be used free of charge and is distributed under the terms of the BSD license. Its timer subsystem is described in the TinyOS Enhancement Proposal 102 [52]. All of its five timer APIs directly interface hardware peripherals. They, however, "do not attempt to capture [timer] diversity in a platform-independent fashion, [besides] measuring time and triggering events at specific times." [52, ln. 41ff]. Each interface is parameterized by its precision (e.g., 1 ms) and three additionally by their width (e.g., 32 bit), providing static properties of the underlying peripheral to the application.

Platforms expose their timers via the `Counter` and `Alarm` interfaces. The first only allows to read the current system time and signals overflows by a respective event. The `Alarm` interface extends the `Counter` interface by compare channels, signaling timeout expiry via designated events. It is split into basic commands, such as `start()`, `stop()`, or `fired()`, and an extended interface with commands for advanced use, such as `startAt()`, `isRunning()`, or `getAlarmValue()`. Synchronous delays using active waiting are available via the `BusyWait` interface and `LocalTime` provides a fixed 32-bit counter without overflow event handling. Last, the `Timer` interface again is split into a basic and an extended command set. It supports periodic timeouts and can therefore be used as a base clock for software timer multiplexing.

Zephyr

Among all reviewed operating systems, Zephyr [39] is the newest project. This real-time open-source OS is targeted at generic IoT devices and is actively developed by the Linux Foundation. It supports a diverse range of boards and is distributed under the Apache 2.0 license. All available hardware is modeled using a DeviceTree [16] data structure and system components can interactively be configured using Kconfig [36].

Low-level hardware timers are exposed via a single `Counter` interface that operates on device level. It requires user applications to provide a platform-specific mapping for each MCU target individually. The API supports basic functions, including `start()`, `read()`, and `set_channel_alarm()`, but does not allow writing the counter value. It can be used to generate absolute and relative one-shot timeouts, but lacks support for periodic timeouts. Separate callback functions can be attached to compare match as well as overflow events. Both, static timer attributes, such as counter width or channel count, and run-time properties, such as counting mode or pending interrupts, are made available to the application. An optional guard period can be set to delay the start of absolute alarms whenever they would exceed counter range boundaries. RTCs should be exposed via the designated interface. However, "all RTC peripherals are implemented [platform-dependent] through `Counter`" [37] at the time of writing.

Platform-agnostic software timers are provided by the kernel timer module `k_timer`, which multiplexes all virtual timers onto the kernel system clock. It offers periodic timeouts and allows the attachment of callback functions to timeout expiry and early stop events. Each software timer is named and maintains a status value that indicates how often it expired since it was last read. Resolution is limited by the system clock tick rate, which usually is in the order of milliseconds. Requesting timeouts in the microsecond range is unlikely to work and strongly discouraged by the developers [38].

Comparison and Synopsis

Out of all four showcased embedded OSs, two provide APIs for peripheral-level access to hardware timers, while the other two solely offer virtual software timeouts. Three statically multiplex all virtual timeouts onto a single system clock, where TinyOS is the exception. However, TinyOS still requires the user to manually distribute virtual timeouts on the instantiated software modules. While three of the OSs do provide at least one form of fully platform-agnostic timeouts, none of them exposes advanced timer features via their provided APIs. Developers are left with the task of manually implementing low-level driver code for such, at all times. Furthermore, none of the systems allows to dynamically select application-appropriate peripherals based on hardware attributes, such as counter width or compare channel count. Possibilities to control timer counting modes or peripheral interconnect systems were present within none of the analyzed

software modules. All virtual timeouts were found to be in the range of milliseconds or more, whereas low-level timeouts are usually expressed as raw counter ticks.

4.1.5 Miscellaneous

Comparative benchmarks of the two high-level timer modules that RIOT OS offers are performed by Ismail [28]. Various characteristics of virtual software timeouts are assessed. These include system timekeeping, timeout accuracy, scaling behavior, and maintenance overhead. The author uses PHiLIP [58] to record GPIO event traces independent of the target hardware. Measurements assess eight different microcontroller boards and are conducted in an automated fashion. We base our evaluation setup on this work, as described in-depth within Section 7.3.

4.1.6 Summary

Hardware timers were found to share many common operation principles and properties. Besides basic general-purpose timers, a multitude of advanced timer types, each offering a distinct set of features, are often available. Even though they all still share certain properties, timer type specific features were found. This means that our API design can provide a generic platform-independent interface but also must be capable of exposing advanced features that might not be available on every device. We evaluate the addressed properties and capabilities in detail within our timer hardware analysis in Section 4.3. Other comparative analyses of embedded hardware were previously conducted, but only two discuss hardware timers at the required level of detail. The importance of energy consumption with resource-constrained devices was highlighted within multiple publications. As timer peripherals possess a huge power-saving potential, a well-design API should support the respective features.

There is a multitude of options for presenting hardware to applications and OS modules. Single-layered solutions tend to yield the best run-time performance, but are limited in the set of features they can expose in a platform-independent way. At least one additional layer of abstraction therefore was found necessary for exposing a heterogeneous range of peripherals. A simple timer API implementation confirmed the feasibility of this approach. Every OS module depends on the capabilities of the underlying abstraction layers. A brief look at high-level timer subsystems affirms the importance of profound yet performant low-level hardware interfaces. Access to all available timers, including their compare channels, and dynamic peripheral configuration at run-time were identified to be of utmost importance for high-level timer modules.

We identified a clear demand for efficient platform-independent timeouts, but at the same time found existing interfaces to lack the flexibility to adequately provide such. Popular embedded operating systems try to obviate this deficiency by multiplexing virtual software timeouts. This

not only limits timeout resolution to the millisecond range, it moreover wastes the enormous potential of advanced timer peripherals that nowadays are widely available. Especially with small IoT devices, developers therefore regularly are forced to bypass existing hardware interfaces along with manually replacing them with error-prone and platform-specific solutions. Generic mapping and configuration of peripherals, e.g., using Devicetree [16] and Kconfig [36], was furthermore found beneficial to application portability.

4.2 RIOT OS Timer Modules

All current low-level timer APIs of RIOT are highlighted in this section. We outline common design aspects, give insight into implementation details and discuss current shortcomings.

At the time of writing, a total of five low-level timer modules exist in RIOT OS. Each of these exposes a specific class of hardware timers or provides an advanced timer-related function to the user application. Three APIs, namely `periph_timer`, `periph_rtc`, and `periph_rtt`, are designed to handle common timekeeping applications while the two remaining modules, namely `periph_pwm` and `periph_wdt`, are used for more specialized tasks. An overview of all analyzed low-level APIs including their properties and functions is given in Table 4.1. Each module is individually discussed in the remainder of this section and a brief overview of available high-level timer subsystems is given before we finish with a summary of all analyzed low-level timer APIs.

4.2.1 General-purpose Timer Driver

The `periph_timer` module drives the various general-purpose timers of an MCU. It is designed to be used as a generic interface that provides short-running timeouts with a high resolution. Multiple general-purpose timers, including all their compare channels, can be addressed and simultaneously be used via this API. Their counter values are exposed as raw counter ticks and a desired counting frequency can be specified during peripheral initialization.

Basic common timer features are made available via the platform-independent function interface. These include reading the counter register value and arming timer channels to generate events at specified counter values. Writing the counter value is not supported by the API. A user-defined interrupt service routine (ISR), to be executed upon compare match events, can furthermore be attached during timer initialization. Only a single combined callback function can be attached, hereby leaving the distinction between source channels to the user application. More advanced configuration options such as counting mode, auto-reload, and clock selection, however, are not exposed. These are implemented in a highly platform-dependent fashion and handled directly by the respective peripheral driver, hence are unavailable at run-time.

	`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)	✓	×	×	✓	×
Interrupts / 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 alarms 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 a platform-specific timer instance identifier.
(II) Module is able to interface multiple compare channels or RTC alarms per timer instance, distinguished by a numeric 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 (i.e., power gating) 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., width, feature support, number of compare channels) are made available via the module.
(VI) Exposed timer peripherals are simultaneously used by other low-level modules, hereby creating a resource allocation conflict. For example: Using TIM1 for PWM generation via `periph_pwm` must result in removal of TIM1 from the set of timers that are driven by `periph_timer`. Otherwise, a peripheral allocation conflict arises.
(VII) Module is used to drive other timer types than initially intended (e.g., using a "low-power timer" instead of a "real-time timer" with `periph_rtt`).

Table 4.1: Comparison of current low-level timer modules in RIOT OS

4.2.2 Real-time Clock Driver

Real-time clock (RTC) peripherals can be accessed via the `periph_rtc` module. It provides user applications with long-running timeouts. Elapsed time is represented as both a wall-clock time struct[2] with second resolution and an additional sub-second counter with microsecond resolution. Functions for convenient time conversion and comparison are offered by 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 exposed RTC peripheral is hard-coded, hereby eliminating the need to specify a peripheral identifier during API calls. This, however, sacrifices the option to address multiple RTC modules, as provided by some MCUs. This restriction likewise applies to alarm channels. Only one single channel is exposed to the user application, hereby leaving additional alarms unusable. A user-defined callback function can be attached to the alarm channel, but overflow events remain unexposed.

4.2.3 Real-time Timer Driver

The real-time timer driver `periph_rtt` is a mixture of the general-purpose and RTC modules, offering long-running timeouts with a low resolution. It is designed to be used with designated real-time timer peripherals, but regularly is repurposed to expose other timer types, such as low-power timers. Elapsed time is represented as raw counter ticks, but can be converted to relative wall-clock time using the provided preprocessor macros. The current counter value can both be read and set to an arbitrary value. While not providing functions to start and stop the hardware counter, functions for powering the whole timer module up and down are given instead.

Similar to the RTC driver, the RTT driver does not distinguish between hardware timers. It therefore likewise only supports a single hard-coded timer peripheral. Equally, only a single alarm channel is exposed. However, in contrast to the RTC module, an exclusive overflow callback can be attached. Configuration of this callback is independent of timer initialization and can therefore be altered at any point during run-time.

4.2.4 Pulse-width Modulation Driver

The generation of pulse-width modulation (PWM) signals is a common task with embedded systems. Some timer peripherals therefore offer designated hardware features to support this. With RIOT OS, the `periph_pwm` interface can utilize both hardware timers that offer such PWM capabilities and designated PWM hardware. Since this API is focused on output signal

[2] The wall-clock time struct consists of one unsigned integer value for each of the following time units: second, minute, hour, day, month, year, day of week, day of year

generation instead of sole counting tasks, raw values of the internal counter register are left unexposed. It therefore is impossible for applications to determine the current position within the period of a generated signal and the module cannot be used to generate timeouts.

The frequency and resolution of the generated waveform can be specified during initialization, hereby impacting the duty cycle of the output signal. Multiple simultaneously operating PWM channels are supported and distinguished by a unique identifier. Each channel has its own configurable duty cycle but shares frequency and resolution with all other channels of the hardware module. Powering up or down single PWM channels is not supported. Once the hardware peripheral is enabled, all channels are always active.

4.2.5 Watchdog Timer Driver

Watchdog timers (WDGs) 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 are classified as timer peripherals on most MCUs while providing only a very limited feature set. The `periph_wdt` interface comprehensively exposes this set of features to other OS modules and user applications alike.

Two common operation modes, namely normal and windowed mode, are supported and a user-defined callback function can be executed prior to a watchdog timeout, whenever supported by the hardware. Availability of hardware capabilities and features is indicated by preprocessor definitions for each MCU platform. It is not possible to distinguish between multiple available watchdogs, as the peripheral mapping is hard-coded within the implementation.

4.2.6 High-level Timer Subsystems

RIOT provides user applications with high-level timeouts through the `xtimer`[3] module and its successor `ztimer`[4]. Both timer subsystems allow retrieving the current system time, provide wall-clock time based timeouts, and support active waiting, i.e., spinning. Ultra short timeouts must be used with caution, as solely surpassing the timeout duration is guaranteed, but a low timeout latency cannot be ensured.

Both modules multiplex virtual software timeouts onto one or more hardware timers. To achieve this, they use the generic low-level timer APIs that previously were discussed in this section. The `xtimer` module maps all timeouts onto a single channel of one hardware timer that is exposed via `periph_timer`. The `ztimer` module instead allows to define multiple clock tick

[3]RIOT docs on `xtimer`: `https://doc.riot-os.org/group__sys__xtimer.html` (Accessed: 12.01.2022)
[4]RIOT docs on `ztimer`: `https://doc.riot-os.org/group__sys__ztimer.html` (Accessed: 12.01.2022)

sources, which can use any of the three generic low-level timer APIs. An automated heuristic-based method for timeout distribution, such as grouping timeouts based on their durations or time units, is not provided. User applications therefore must split timeouts manually across the defined clock sources, which again can use only a single compare channel or RTC alarm.

4.2.7 Summary

RIOT offers three generic and two special purpose low-level timer interfaces. The generic ones, namely `periph_timer/rtt/rtc`, expose common timer types, such as general-purpose and low-power timers as well as RTCs. The special purpose drivers, namely `periph_pwm/wdg`, do not expose timers directly, but instead use the underlying hardware to provide higher-level features, such as signal generation. Separating these software modules allows optimizing their interfaces for specific timer types, but prevents their transparent and interchangeable use. With this approach, either not all timers can be used due to missing interface definitions or developers are forced to expose dissimilar timer types via the same, potentially unfavorable, API.

Functions provided by the low-level interfaces are limited to the feature set that is common to all MCU-platforms, leaving additional timer features unsupported. Though functionality of most modules overlaps, we found their APIs to differ in use and feature availability. This includes, for example, counter value representation, interrupt handling, and peripheral instance management. Underlying timer types moreover vary between MCU families, which is particularly common with the real-time timer driver `periph_rtt`. In some cases, the same peripheral is simultaneously exposed by multiple modules, leading to resource allocation conflicts and undefined behavior. Special timer types, such as high-speed counters or ultra low-power timers, frequently remain unsupported by any of the APIs. Three modules statically map to a single hardware timer and channel, hereby leaving a significant number of available peripherals and compare channels unusable, e.g., multiple low-power timers on the STM32 platform. This also limits the number of simultaneous timeouts to one per software module. Attachment of user-defined callbacks, to be executed upon timer overflow interrupts, is only supported by `periph_rtt`. Timer selection and configuration requires manual changes to system header files. The structure and location of those files is highly heterogeneous and differs largely across target platforms. Determination of timer hardware properties and capabilities during run-time is only supported by the watchdog driver `periph_wdt`. User applications and high-level timer subsystems therefore are unable to dynamically adjust their behavior based on the available peripherals. Considering the above, the application developer often has to expose hardware timers through insufficient or potentially unfavorable interfaces, encouraging bugs and narrowing the window for optimal hardware utilization. Whenever additional timer features or specific timer peripherals are needed, manual (re-)writing of low-level driver code is required by the application developer.

4.3 Hardware-platform Analysis

A major contribution of our work is the analysis of different timer peripherals, which are found in microcontrollers that are currently supported by RIOT OS. In this section, we start by defining the analysis scope as well as the methodology we apply conducting it. Results, both specific to device families and also across all analyzed platforms, then are presented and discussed with an outlook on the aspired low-level timer API. Outstanding tasks and possible improvements are identified before a quick summary concludes this section.

4.3.1 Scope

Our analysis covers all chip manufacturers that offer at least one microcontroller, that is targeted by RIOT OS at the time of writing. For each of those, all supported device families were examined. This yields a total of 43 MCU families, produced by eight different manufacturers. Device families that share timer types, such as the STM32 device families, were combined during our analysis. Table 4.2 gives a detailed overview of all device families that are part of our analysis. Combined families are represented within a single row.

Manufacturer	Device Family
STMicroelectronics	– STM32F0 / F1 / F2 / F3 / F4 / F7 STM32L0 / L1 / L2
Microchip / Atmel	– ATmega AVR – PIC32MX / MZ – SAMD21 – SAM3A / N / S / U / X
Espressif	– ESP8266 – ESP32
Silicon Labs	– EFM32 / EFR32 – EZR32
Texas Instruments	– 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

Table 4.2: Summary of device families we analyzed within our hardware-platform analysis

4.3.2 Methodology

Each step of our hardware analysis is conceptually depicted in this subsection. All steps appear in the order of their execution.

Platform Selection and Information Acquisition

At first, all CPUs that are currently supported by RIOT OS, as listed in `/cpu`[5], were determined. Chip manufacturers and their respective device families were then rated and processed according to their expected 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.

Definition of Analysis Criteria

After obtaining a brief hardware overview, the initial set of analysis criteria and properties was defined. It includes all aspects we rated as being of relevance for a timer API and therefore are to be extracted from the gathered documents for every assessed MCU platform. Selected characteristics derive from our surveys of both related work, as summarized in Section 4.1.6, and existing timer interfaces, as summarized in Section 4.4.3. These were further expanded by us to cover additional aspects, according to their anticipated significance. The initial set of criteria includes basic timer properties such as counter register width, prescaler capabilities, compare match channels, and auto-reload functionality as well as advanced aspects such as interrupt generation, timer chaining, and low-power features. A full list including the detailed definition of each criterion is given in Section A.1.

Extraction of Timer Hardware Details

All selected timer peripherals were analyzed and information found within the previously 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 8.1). If the device documentation was unclear at some point, additional information sources were used to resolve the respective property. These included peripheral register descriptions as well as SDKs provided by the respective manufacturers. However, if a property could not be determined with sufficient confidence, it was marked as unknown.

[5] See: `https://github.com/RIOT-OS/RIOT/tree/master/cpu` (Accessed 22.04.2020)

Consolidation of Results

Since our data acquisition was not strictly limited to the defined criteria, it yielded information beyond the initially chosen set of 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 for a group of devices. It allows to quickly determine various characteristics and features of each of the available timers and makes comparison across different MCU families and manufacturers possible. The TCMs were subsequently used as a data basis to derive our inter-MCU-platform findings from. All created TCMs can be found in Section A.2.

Inter-MCU-platform Findings

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

Each analyzed property possesses a unique identifier that is used for referencing, a short title, and a description that elucidates the respective property. Moreover, a selection criterion can be specified, allowing to split properties into multiple cases, e.g., to separate counters by available register width. All assessed inter-MCU-platform properties are listed in Table 4.3 and get discussed in Section 4.3.3.

4.3.3 Results

Cross-platform findings from our timer hardware analysis are listed in Table 4.3 and discussed in the remainder of this section. It should be noted that the total number of platforms and timer types may vary between properties. This can either be due to an applied selection criterion or due to the exclusion of specific timer types, as stated in the property description. To cope with this, we list both the exact number of timer types or platforms and their proportional share among all applicable types or platforms for each result. A detailed description of our data collection and evaluation methodology can be found in Section 4.3.2.

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 $\leq 16^{\dagger}$	10	15	71 %	38 %
			R-01 $> 16^{\ddagger}$	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 or unknown properties	*yes*	6	17	32 %	14 %

∗ Three platforms are excluded due to unknown properties. See "Interrupt Handling and Event Flags" within Section 4.3.3 for details.
† i.e.: Only counting timers that are chainable and have a counter width of 16 bit or less.
‡ i.e.: Only counting timers that are chainable and have a counter width greater than 16 bit.

Table 4.3: Hardware analysis results across all MCU platforms. Each result is evaluated for both timer types and associated MCU platforms. It is explained by its description and may be split into multiple cases, as denoted by the specified selection criterion. If specific timer types were excluded, it is indicated by the result description. See Sections 4.3.2 and 4.3.3 for details.

Counter Range

A ubiquitous property of timer peripherals is the width of their internal counter registers. It dictates the maximum number of cycles a timer is able to count before an over- or underflow happens. The less frequent such events occur, the less timer maintenance and CPU wake-ups are required. Greater widths allow for longer running low-level timeouts, i.e., timeouts that directly use a physical compare channel, without the need for complex high-level software timers. A large counter width therefore is desirable.

Our analysis revealed 16 bit to be the minimum counter register size among all platforms. 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 extending counter range through the use of timer chaining. This feature allows combining one or more hardware timers to extend their width. 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 lowering the number of wake-ups and a reduction of 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 use of timer chaining comes at the cost of sacrificing one timer peripheral for every range extension. This can be undesirable for some applications and the usage of timer chaining therefore shall remain optional.

Prescalers can reduce the frequency of a clock prior to feeding it into a timer, hereby making it only count every n-th pulse of the base clock. They allow to use a single clock source for multiple hardware timers with different counting frequencies and extend long-running timeouts. This advantage, however, comes at the cost of a reduced resolution, due to the lower base clock frequency. Prescalers are nonetheless indispensable, especially when dealing with small counter widths ($\leq$ 16 bit), high clock frequencies, and long time durations. This trade-off indicates, that a separation of short high-precision delays and long-running timeouts is desirable for an optimized timer driver in order to perform well in a diverse range of application scenarios.

Prescalers are generally available on all platforms, as well as on 75 % of all analyzed timer types individually (see R-03). The only platform that has non-prescalable general-purpose timers is the SiFive FE310-Gx (see Table A.16), which instead features a 64-bit counter register and thereby eliminates the need for an additional prescaler. Moreover, all other timer types, which are available on this MCU, do provide prescalers.

Auto-reload Capabilities

Timers frequently need to generate either periodic events or timeouts that are longer than the time before the next counter over- or underflow happens. Both use cases therefore require a

convenient way of resetting and restarting the timer. To prevent the missing of clock pulses and to reduce maintenance overhead, this task can directly be handled by the timer hardware via the auto-reload feature. It resets the counter register to either a fixed or configurable value, once a designated event occurs. This can be a counter over- and underflow, a compare channel match, or the reaching of a configurable threshold.

All applicable timers were found to support at least one 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 specification of 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 use of a designated auto-reload register (51 % of all timers). Having a separate auto-reload register benefits a timer subsystem by keeping all compare channels available to the application and other OS modules. We therefore argue that using a separate register is the preferred method for reloading.

Compare Channels

Timers can trigger system events, such as CPU interrupts, at specific counter values through the use of compare channels. Each compare channel can be armed to an arbitrary value, which is continuously compared with the counter register. This operation is performed directly by the timer hardware, thus no polling or active waiting is required. A compare 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 larger the number of parallel timeouts can be and the more freely different timeouts can be split across channels. This, for example, allows to separate short from long-running timeouts, what benefits the distribution of virtual timers across hardware peripherals and hereby reduces 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 within an optimized timer subsystem. Otherwise, active polling of the current counter value would be required, thereby occupying the CPU with timer maintenance tasks and preventing the use of power-saving modes. 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 usage within timer subsystems.

Interrupt Handling and Event Flags

Timer events, such as over- or underflows and compare matches, may generate interrupts (INTs) upon occurrence. Corresponding interrupt service routines (ISRs) can be used by timer drivers

to get informed about expired timeouts and to execute internal maintenance tasks as well as user-defined callbacks. Interrupt handling heavily depends on the actual CPU but was found to usually be implemented uniform among device families from the same manufacturer. When handling such events it is important to know whether an exclusive interrupt for every single event exists or if multiple events share one common interrupt vector. The former does not require manual resolving of the interrupt cause upon occurrence and therefore is the preferred method. With the latter, additional status register reads are inevitable to determine the exact interrupt cause.

Our analysis revealed that out of all timers only 19 % provide fully independent overflow and only 31 % offer fully independent compare match interrupts for each channel, as indicated by `R-05` and `R-06`. An additional layer of indirection for resolving the interrupt cause therefore is introduced in many cases, hereby negatively impacting timeout latency. Flags that indicate peripheral status and signal event occurrence are available throughout all platforms, as shown by `R-07`. Three platforms, namely Espressif ESP8266 (see Table A.6), Espressif ESP32 (see Table A.7), and Nordic Semiconductor nRF51x/52x (see Table A.15), do not state the availability of event status bits in their manufacturer provided documentation (see Section 4.3.4). These device families were therefore excluded from the total platform count. Nonetheless, we highly doubt that there is no way to access such information but were not able to reliably confirm its availability either.

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 former are generated solely within the MCU and therefore are always available. The latter are either supplied as an external signal that is applied to a specific input pin or generated internally with the help of additional 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 source, as shown by `R-10` and `R-11`. It was furthermore found that on 84 % of all platforms multiple timer clock sources are configurable, as shown by `R-09`. This yields numerous clock configuration options and indicates that clock configuration should be considered an important aspect when designing a timer API. Especially power constrained devices can benefit from run-time clock source reconfiguration. It aids dynamic power mode transitions and allows utilization of special purpose low-power oscillators. As these applications entail a large set of specific requirements, implications for low-power operation and further energy saving considerations are discussed separately in the following section.

Low-power Operation and Energy Saving

Proper utilization of low-power operation modes is particularly crucial with resource constrained devices. The ability to operate timers of a low-power oscillator therefore is of high importance for many IoT applications. 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 a range of power-saving modes modern MCUs provide. This, for example, includes powering down the CPU and main peripheral clock while keeping the required timers operational, hereby greatly reducing the idle power consumption. Allowing utilization of such features therefore is of utmost importance when designing a timer subsystem for embedded OSs like RIOT and IoT applications in general.

We found that all platforms provide at least one timer type that can run of a low-power clock. Our 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 fulfill these criteria as *always-on peripherals*. Among these, real-time counters and real-time clocks are the prevalent type. Five platforms additionally provide designated ultra low-power timer peripherals (see Tables A.2, A.3, A.8, A.11, and A.17), as for example the Cryotimer on the Silicon Labs EFM32/EFR32 platform (see Table A.8). Especially when dealing with long timeouts and deep-sleep periods, these timer types allow significant energy-consumption optimizations and therefore must be made available to the user application through a well-designed timer API.

Application Suitability of Timer Types

Even though we covered all types of timer peripherals within our analysis for the sake of completeness, we think that not every timer type is applicable for the use in a generic timing subsystem. We suggest that especially watchdog timers fall into this category. These timers are designed to recover a system from firmware malfunctions, endless loops, or critical error states. They achieve this through performing a full device reset if they are not periodically serviced by the application. Our analysis discovered that 67 % of all watchdogs offer the possibility to generate an interrupt prior to or even instead of performing a system reset, as shown by R-15. While this ability can theoretically be used to generate generic timeouts, it must be used with great caution to prevent unintentional system resets. Watchdog timers are usually limited to a very basic feature set and often behave non-uniform across MCU platforms. Moreover, they commonly are already occupied by other components of the application or system modules. We therefore do not recommend their usage within a generic timer subsystem.

Hardware Diversity

Being able to choose timer peripherals from a wide range of types opens up a broad spectrum of opportunities for application optimizations. 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 just one platform that guarantees the availability of only a single general-purpose timer, namely the SiFive FE310-Gx (see Table A.16). All other platforms provide at least two or even more general-purpose timers, as indicated by `R-14`. The availability of other timer types varies greatly between manufacturers and even within device families of the same manufacturer. Those platform-specific peripheral types, however, do provide advanced features that are important to many applications, such as high-resolution timeouts or low-power operation capabilities. We therefore conclude, that both the proper utilization of the available timer modules and the incorporation of platform specific peripherals alongside generic general-purpose timers is a necessity for a uniform yet flexible timer interface.

We moreover encountered MCUs that only leave the possibility of multiplexing high-level software timers onto a single general-purpose hardware timer, as outlined by `R-14` and `R-02`. These namely are the SiFive FE310-Gx (see Table A.16), with only a single general-purpose timer, and the Espressif ESP8266 (see Table A.6), with only one compare channel while leaving the alarm functionality of the RTC undocumented. We therefore argue, that a timer subsystem must be able to cope with situations in which only a single hardware timer is available. A way to determine the available hardware and its capabilities at run-time can hugely aid this aspired flexibility while also benefiting application portability when changing target platforms.

Missing Information

Not every property of the analyzed timers could be determined with sufficient confidence, based on the available documentation or further information sources, as described in Section 4.3.2. Six platforms therefore still suffer such unresolved properties, as indicated by `R-16`. Five of these can more precisely be grouped into the two Espressif MCUs (see Table A.6 and A.7) as well as three of the Cortex-M based platforms, namely the Microchip / Atmel SAMD21 (see Table A.5) and the Silicon Labs MCUs (see Table A.8 and A.9). The documentation of the Espressif devices, especially the ESP8266, remains unclear about many of our analysis criteria and only some properties could be resolved by inspecting the manufacturer provided SDKs. The Cortex-M based MCUs, in contrast, only leave the *SysTick* timer, which commonly is found across processor cores of this type, largely or even fully 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 A.15) remains unclear about the availability of event flags.

4.3.4 Outstanding Tasks and Issues

Even though our analysis of timer hardware 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 in this section.

Unresolved Properties

During our analysis we were able to gather data for nearly all the defined analysis criteria and timer types within our scope. Nonetheless, some properties still remained unresolved, as outlined in the previous section. Resolving these may further improve our results. When assessing these properties, however, the required effort must be justifiable when put into comparison with the estimated information gain and expected impact on the overall results.

Timer Resolution and Clock Tree Properties

Its resolution, i.e., the period of a single timer tick, and the maximum timeout length it can achieve without requiring intermediate maintenance wake-ups both are essential characteristics of every hardware timer. A timer subsystem can use this information to allocate requested timeouts to the most appropriate hardware peripherals that are available. Determining these properties generically for a whole class of timers is unfortunately not feasible due to their strong dependence on MCU oscillator frequencies. As the system clocks depend on both the microcontroller and its current configuration, their actual operation frequencies can vary greatly, hereby 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 different clocks each timer peripheral is able to run of. Since each device family or even single MCUs provide different oscillators and methods of routing the generated clock signals to peripherals, a standardized and comparable way of analyzing these has yet to be defined. As this clock tree analysis is an entire complex topic on its own, it was excluded from our hardware analysis for now. We nonetheless expect promising results from it, especially with respect to power-saving optimizations and therefore suggest conducting an in-depth analysis of the clock trees within future work (see Section 8.1).

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 executing certain hardware tasks based on specific events whenever generated by applicable peripherals. Both methods benefit the overall system performance by removing the need to execute a designated ISR upon event occurrence,

hereby leaving the CPU available for application use at all times. Such peripheral interconnect and event systems might prove valuable for some applications, e.g., timer chaining. Furthermore, a reduction of maintenance overhead could be possible by exploiting event systems in order to execute simple maintenance tasks autonomously within respective hardware peripherals.

Configuration and Maintenance Costs

Each read, write, status check, or reconfiguration of timer peripherals requires a certain amount of system resources, such as CPU time or battery power. Information about the exact resource consumption of such operations is unfortunately only sparsely available. Timer operations can furthermore entail side effects that need to be taken into account. For example, it can be required to enable a high-power oscillator in order to read or write registers of a timer that is running of a low-power clock. As a result, frequent timer operations drastically increase power-consumption, as the high-power oscillator is always started prior to execution of the desired operation. We therefore consider information on such costs beneficial to the timer peripheral selection process and suggest further research.

4.3.5 Summary

To gain insight into the diverse landscape of hardware timers, we conducted a large-scale analysis of timer peripherals. It includes 43 device families from eight manufacturers, hereby covering all MCUs currently supported by RIOT OS. Assessed aspects derive from our survey of existing timer interfaces and related work. Their initial set was extended, whenever our explorative data acquisition revealed additional points of interest. We examined basic characteristics such as counter width, prescalers, and compare channels as well as advanced aspects such as interrupt generation, timer chaining, and low-power features. Gathered information was transformed into uniform result tables, referred to as *Timer Comparison Matrices* (TCMs), found in Section A. These allow a detailed comparison of timer peripherals across all assessed device families. The TCMs were then used as a base to derive inter-MCU-platform findings from.

Our analysis identified counter register widths to be at least 16 bit among all platforms, while 90 % also provide 32-bit and 21 % even 64-bit timers. On 71 % of all platforms, smaller timers in particular support range extension through timer chaining. Frequency prescalers are always available, and many timers provide at least two (64 %) or even four (24 %) channels. Only 31 % feature fully independent interrupt vectors for every compare channel. On 84 % of all platforms, multiple timer clock sources are selectable and 71 % of all timers can be driven by a designated low-power clock. All platforms offer at least one timer that is capable of operating in the lowest power mode and waking the CPU on designated events.

Susnea and Mitescu [54] previously found common characteristics and operation principles among all their three devices. Our analysis confirmed this to also apply on a larger scale. Basic feature sets of general-purpose timers were found uniform across all MCUs, allowing to expose them in a platform-independent way. Other timer types differed largely in function and modes of operation, requiring a more complex interface as a consequence. Simply extending a generic interface to support device-specific features contradicts its platform-independence. We therefore argue, that a uniform timer API should provide both a basic platform-independent and a platform-specific interface to be capable of effectively abstracting and presenting the available timer hardware.

4.4 Use Case Analysis

Knowing the different use cases and requirements for a software module is crucial when designing and evaluating it. Applications within the Internet of Things are highly diverse and all have their own characteristics. These include smart devices, environmental monitoring, process automation, vehicle-to-vehicle communication, and many more [57]. What they all have in common, however, is that they are controlled and exchange data via different types of networks [23]. Therefore, timers are not only required for the local tasks of an embedded device, e.g., periodically reading a sensor, but also for the associated network communication.

The durations of requested timeouts constitute a key indicator to judge the performance and applicability of a timer interface. We quantify it for both above described cases by analyzing the usage of timer functions within the whole RIOT codebase and assessing specifications of network communication protocols that are commonly used with IoT and WSN applications. Results of these analyses and their implications for the aspired timer API are discussed in this section.

4.4.1 Timeouts within RIOT OS Modules and Drivers

To determine characteristics of real-world timer usage, timeouts within all RIOT modules, CPU implementations, and peripheral drivers were evaluated. This includes both low-level timeouts via `periph_timer` and high-level timeouts via `xtimer` and `ztimer`. For each module, all non-blocking timeout calls were evaluated. This includes absolute, relative, and periodic timeouts but excludes tick-based active waiting (i.e., spinning). If a module uses multiple applicable timeouts, the shortest timeout duration was always selected, since it entails the strictest performance requirements. If a module does not use any applicable timer function, it was excluded from the analysis. This resulted in a total of 97 modules. Figure 4.1 shows the distribution of the shortest timeout durations among each of these.

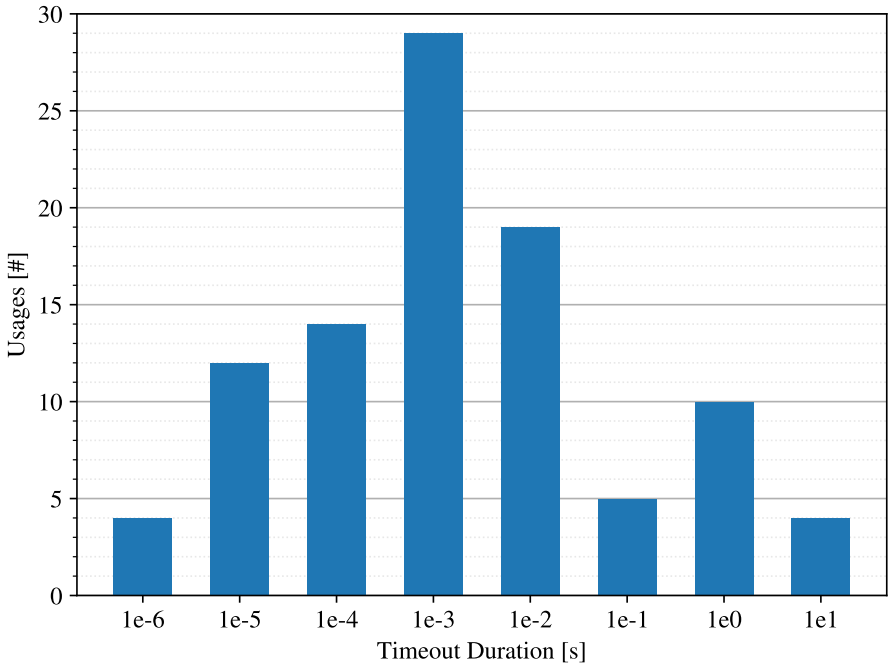


Figure 4.1: Distribution of shortest timeouts used within RIOT OS modules and drivers

Timeouts are grouped into bins according to their order of magnitude. The 1e-3 bin, for example, covers the right-open interval $\left[10^{-3}, 10^{-2}\right)$. The most common range for timeout durations is 1 ms to 10 ms, as found within 30 % of the analyzed modules. Moreover, 69 % of all modules solely use timeouts of at least 1 ms and 14 % even exclusively use timeouts that are at least 1 s long. The latter are used to model message timeouts within different communication protocols, such as CAN, LoRaWAN, or BLE. Due to their long durations, they usually do not require high-precision timeouts. Shorter timeouts ($\tau \leq 1\,\mathrm{s}$) are frequently used within drivers to reliably surpass a certain hold-off time that is required by a peripheral device. Precision of these is no issue either, as these only require to surpass a certain time threshold but do not suffer from a slightly delayed response.

Only four peripheral drivers use timeouts that are shorter than 10 µs. Two of those represent minimum hold-off times (`hd44780` and `mcp2515`), hence are insensitive to timeout latency and jitter. The remaining two are used for software clock generation (`sdcard_spi` and `soft_uart` @ 11 520 bps[6]), hence require precise timing. These clock signals are generated by periodic timeouts that are used to trigger a General-purpose input/output (GPIO) pin. A reliable clock signal possesses a stable period but is insensitive to absolute offsets due to its periodicity. It therefore can tolerate a certain amount of timeout latency but is highly sensitive to jitter.

[6]The actual timeout period depends on the selected UART bus speed. For common configurations it ranges between 104.2 µs @ 9600 bps down to 8.68 µs @ 115 200 bps.

4.4.2 Network Protocol Timing Requirements

The RFC 1122 communication layer model [10] groups network protocols into four distinct layers. It is often also referred to as the *TCP/IP model*. The generic (*GNRC*) network stack and application layer protocol implementations of RIOT are structured in the same way [33]. We inspected popular members from each of its layers to systematically assess timing requirements of network communication protocols. An overview of this model is illustrated in Figure 4.2.

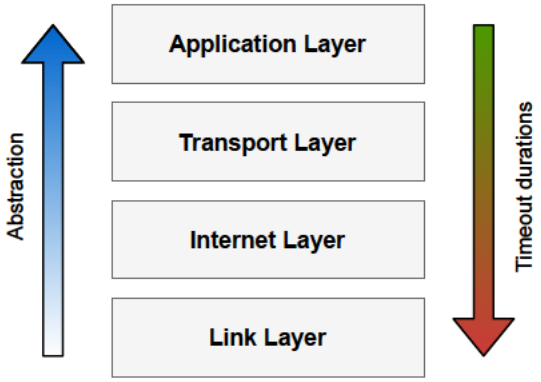


Figure 4.2: Overview of the RFC 1122 communication layer model

The rapid growth in size and diversity of both the IoT and WSNs fostered the development of numerous novel protocols on most network and adaption layers [21]. Our analysis shows that timeout durations generally are inversely proportional to the abstraction level of respective layers. Within application layer protocols, such as MQTT [5], CoAP [53], or DHCP(v6) [18], all time durations are in the order of seconds or more. This also applies to transport layer protocols, such as TCP [45] or UDP [46], as well as internet layer protocols, such as IPv6 [15] or RPL [60]. This is due to the fact that timers within these layers predominantly are used to schedule and terminate high-level activities, model message timeouts, and recognize absence of events or lack of activity [44].

Timing demands of link layer protocols, in contrast, are significantly higher. IEEE 802.15.4 [27], for example, is used as a base for many communication protocols in the IoT and operates at a symbol time of 16 µs [32]. It requires a minimum of 12 symbols (192 µs) between frames when operated in short inter frame spacing mode and a commonly used ACK timeout is 40 symbols (640 µs). Another widely used protocol is LoRaWAN [40]. It has a maximum data transmission rate of 27 kbit/s, which corresponds to a minimum symbol time of 37 µs [1]. Last, Bluetooth Low Energy (BLE) [9] can operate at over-the-air symbol rates between 125 kbit/s and 2 Mbit/s, corresponding to signal times between 8 µs and 0.5 µs. However, effective application payload throughput is limited by additional factors, such as protocol overhead and the 150 µs inter frame spacing. Timing of link layer protocols is only rarely implemented in software, but instead often is handled directly within designated networking peripherals. Data is passed to such modules

either bytewise or even via direct memory access (DMA). All data is buffered prior to the time critical transmission phase, hereby relaxing timing requirements that are based on the symbol rate. The application nonetheless must be able to manage the data fast enough to keep up with the effective payload throughput that is desired.

Besides the requirements of network protocols themselves, Gundogan *et al.* [21] outline technical timing requirements for industrial IoT use cases. Deployed devices must respond within less than 10 ms for safety critical network traffic. Control and monitoring messages are less important, but should nonetheless be transmitted in less than 100 ms.

4.4.3 Summary

Among all timer usages within the RIOT codebase, 1 ms was found to be the prevalent timeout duration. This is consistent with the findings from our previous review of timer interfaces from other embedded operating systems, as conducted in Section 4.1. Out of all RIOT modules, 69 % only use timeouts that are at least 1 ms long and only two modules require precise timeouts shorter than 10 µs. Generating such very short timeouts is challenging due to constant abstraction and maintenance overheads. With those two modules, however, said timeouts are used for the generation of periodic signals, hence can tolerate a certain amount of timeout latency while being highly sensitive to timeout jitter.

Our analysis of network communication protocols showed that protocols of all layers above the link layer solely require timeouts in the range of seconds or more. On the link layer, however, timeout durations can get as short as 0.5 µs, hence are not accurately achievable with a highly abstracted timer API. Therefore, such timeouts are usually generated by protocol specific hardware peripherals. They relax the timing requirements by providing buffered access to transmitted and received data, while allowing sequential asynchronous writing of application payload. Last, industrial IoT devices must be capable of signaling emergencies within less than 10 ms.

We conclude that a comprehensive timer API evaluation should include timeout lengths between 10 µs and 10 s. This range covers over 95 % of all timeout usages within our analyses. To assess the performance within different scenarios, both the latency and the jitter of timeouts should be measured. Quantifying the platform-independent abstraction overhead moreover allows determining the general accuracy limits of an application programming interface.

5 Design of a Uniform Low-level Timer API

Within this chapter, we employ the insights from our four extensive analyses to develop *uTimer*, a low-level timer API for the RIOT operating system. It provides user applications and high-level OS modules with a uniform and streamlined interface for hardware timers. Different timer types can be used interchangeably, device-specific hardware features are supported, and future extensibility is ensured. uTimer preserves application portability for as long as possible, or until the application developer deliberately sacrifices it.

Prior to all conceptual decisions, requirements for an application programming interface, as we aspire it, are defined in Section 5.1. Addressed aspects include feature support, application portability, resource consumption, and ease of use. We subsequently derive a sound API design from them. It is described in detail in Section 5.2, where we discuss its two-layered architecture, the distinct abstraction layers, and individual aspects, such as interrupt handling or platform agnosticism. User-space code examples follow in Section 5.3 before we discuss various trade-offs and issues of the proposed design in Section 5.4.

5.1 Demands and Goals

RIOT positions itself as a general-purpose IoT operating system that supports a diverse range of resource constrained embedded devices. The OS provides users with generic peripheral interfaces that can be used largely independently of the target hardware, striving for effortless application portability. Use cases and deployment contexts are as vast as their requirements. Regarding timekeeping tasks in particular, ease of use, timeout accuracy, resource consumption, and optimized low-power operation are essential to most applications. Because low-level timer APIs constitute the base layer of abstraction, they must comprehensively address these aspects to be capable of exposing hardware timers in a feature-rich yet flexible and portable way.

Requirements for a low-level timer API, as we aspire it for the RIOT OS, are compiled in this section. Each aspect has a unique identifier and similar ones are combined into a total of six groups, as indicated by the subsection titles. We distinguish between application and operating

system developers. Application developers, on one hand, program the firmware that implements all control logic for a respective use case. They rely on OS modules and peripheral drivers to achieve the desired device behavior. Operating system developers, on the other hand, provide application developers with this sound base. They are responsible for programming any module and driver that is provided by the embedded operating system. Our requirements are defined in the remainder of this section.

5.1.1 Features and Usability

The purpose of hardware interfaces is to comprehensively present on-board peripherals to user applications and operating system modules alike. Features to be exposed and how they are to be used are important aspects to consider when designing respective software interfaces.

A low-level timer API therefore ...

(R-1) should be intuitive and straightforward to use. It should encourage uniformity of timer code throughout applications and OS modules.

(R-2) must allow different timer types, such as general-purpose, low-power, or high-precision, to be used transparently interchangeable via the same uniform interface.

(R-3) must be capable of exposing multiple peripherals, i.e., timer instances, of the same timer type including all compare and alarm channels. Peripherals and channels must be addressable by unique identifiers.

(R-4) should allow to use all timers that a microcontroller offers. This also includes timer types that are only available on a small subset of target devices.

(R-5) must expose generic timer features, which are supported by all peripherals, in a platform-independent way. Advanced timer features, which are only supported on a subset of all devices, should moreover be exposed whenever possible.

(R-6) must expose identical features in a standardized way among all timers. This allows using them regardless of their actual type, as long as they fulfill application requirements.

(R-7) should provide ready-to-use peripheral drivers for all targeted microcontroller boards to relieve application developers from the error-prone task of writing low-level driver code.

(R-8) should allow to present multiple hardware timers as a single timer instance. This includes exposing multiple chained timer modules as an atomic timer instance.

(R-9) should encourage OS developers to follow a common pattern for platform-dependent driver code. This fosters maintainability and guides the integration of new devices. Similar functionality should be implemented within across all platform implementations.

5.1.2 Peripheral State and Capabilities

Numerous system components require timers. Providing convenient and extensive access to the properties of a hardware timer is as important as comprehensively exposing its current state.

A low-level timer API therefore ...

(R-10) should 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 properties* that can only be changed prior to compilation and therefore remain fixed during run-time, such as available timer drivers and modules.
  (c) *run-time properties* that can change at any point in time, such as counting mode or pending interrupts.

(R-11) must expose run-time dynamic properties, allowing user applications and OS modules to determine the peripheral status at any time.

(R-12) should expose static hardware properties, hereby fostering the platform-independent use of timer peripherals.

(R-13) should allow user applications and OS modules to determine whether a timer supports a specific feature during run-time.

(R-14) must support user-defined callback functions for both compare match (CMP) and overflow (OVF) interrupts. The exact interrupt cause should be easy to determine.

5.1.3 Application Portability

Embedded operating systems strive to support a large number of target devices. User applications should easily be portable to different microcontroller boards without requiring a large amount of manual code changes.

A low-level timer API therefore ...

(R-15) must present timer instances in a standardized way across all platforms.

(R-16) should provide both a platform-independent and an explicit way of addressing peripheral instances. This allows
  (a) application developers to explicitly select specific timers from the full set of available peripherals.
  (b) user applications to automatically select timers based on their features.
  (c) using timers inside other OS modules, such as the network stack or sensor drivers.
  (d) optional management of peripheral allocation by some form of external resource allocator, providing mutually exclusive timer usage.
  (e) to define generic application requirements like ”at least n low-level timers”.

(R-17) should allow selecting timer clock sources as either a platform-independent clock class or an explicit platform-specific clock.

(R-18) should maintain platform-independence as long as possible and only be platform-specific whenever inevitable. Target device compatibility should only gradually be reduced, based the required feature set.

(R-19) must allow application developers to consciously sacrifice cross-platform portability whenever platform-specific features are required by an application.

5.1.4 Configuration Management

With the ever-growing number of target devices come not only new peripherals, but also numerous configuration options. Managing their diverse range of configuration options in a generic yet extensive way can be challenging.

A low-level timer API therefore ...

(R-20) must support to select timer instances during compile-time that are then made available to the application at run-time.

(R-21) must permit the dynamic (re-)configuration of hardware timers during run-time. This includes, in particular, changing timer clock sources to facilitate power mode transitions.

(R-22) should encourage a common pattern for peripheral configuration management. It should guide OS developers in storing configuration data and provide application developers with a convenient way of accessing it. If one configurable parameter applies to a relevant number of platforms, it should globally be configured for all affected platforms.

5.1.5 System Impact and Resources

Because timer peripherals are required for numerous system functions and most user applications, they can have an significant impact on system performance. Hence, possible side effects and resource efficiency are important aspects.

A low-level timer API therefore ...

(R-23) must be modular so that code that is not required for the current application can be excluded from the binary. Implementations should automatically be selected for compilation, based on the used peripherals.

(R-24) can define optional convenience functions that may be excluded from the build if desired.

(R-25) must not interfere with other OS modules nor introduce side effects, whenever avoidable.

(R-26) should allow timer instances to be managed by an external resource allocator.

(R-27) should use system resources efficiently and minimize maintenance overhead.

(R-28) should keep the abstraction overhead that is inherent to the API itself to the bare minimum. Timer performance should primarily depend on the platform-specific drivers.

5.1.6 Miscellaneous

Requirements that did not fall into any of the above categories are listed in this section.

A low-level timer API ...

(R-29) must allow to verify the behavior of platform implementations by generic unit tests.
(R-30) should enforce a defined behavior upon hardware failures or invalid parameters. Both erroneous cases should be signaled accordingly.
(R-31) should seamlessly integrate with existing OS modules. This especially applies to high-level timer subsystems, such as `xtimer` or `ztimer`.
(R-32) should be able to coexist with other low-level timer APIs, such as `periph_timer`.
(R-33) should encourage uniform driver implementations and prevent code duplication.

5.2 Design

The design of *uTimer* is proposed in this section. This novel API streamlines existing low-level timer modules and exposes hardware timers via a uniform interface, fostering an interchangeable use of all available timer peripherals. Besides common timer functions, out-of-the-box support for device-specific features is provided, while platform-independence is preserved whenever possible. The developed software design is based on the requirements we defined in the previous section and supported by insights from our four comprehensive analyses.

An architectural overview of our two-layered design is given in Section 5.2.1. The presentation of timer peripheral instances and the used data models are described in Section 5.2.2. It is followed by detailed descriptions of both abstraction layers in the Sections 5.2.3 and 5.2.4. All subsequent sections discuss individual aspects of our design, such as interrupt handling and configuration management.

5.2.1 Overview

Our design is split into a hardware-facing API (hAPI) and a user-facing API (uAPI), as illustrated in Figure 5.1. The hAPI consists of timer-type specific drivers that interact directly with the peripherals. The uAPI then builds upon the hAPI to provide a convenient hardware-agnostic interface to user applications and higher-level OS modules alike. This design decouples the abstract timer logic from hardware-dependent driver code. Application developers are free to

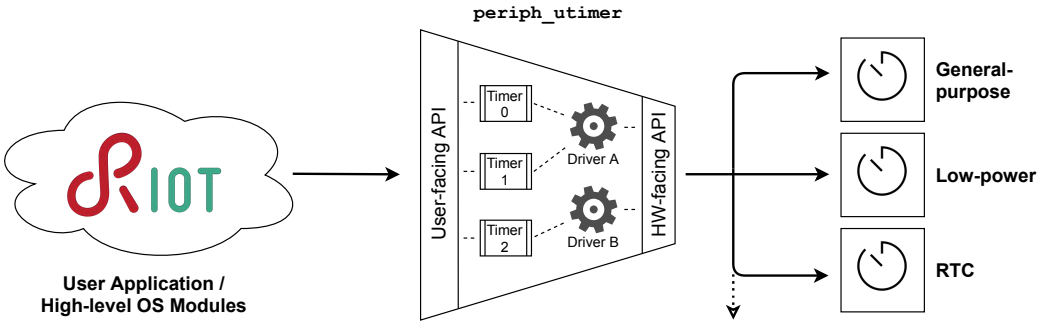


Figure 5.1: Overview of the proposed low-level timer API design

choose the interface that best suits their current needs. Both abstraction layers are steadily available and can be mixed within the same application. uTimer hereby encourages uniformity of timer code throughout the whole RIOT ecosystem.

The additional layer of abstraction inevitably causes computational overhead. Our API design minimizes the negative impact on performance, though cannot fully eliminate it. However, using only a single-layered design would not allow for the aspired interface uniformity, application portability, and extensive feature support. We quantify the abstraction overhead that is caused by both the hAPI and the uAPI individually within our evaluation in Chapter 7.

5.2.2 Timer Types and Instances

Each hardware timer is represented by a designated timer instance struct, called `utim_periph_t`. An excerpt of its definition is given in Listing 1. It uniquely identifies the peripheral device, provides static timer properties, and specifies the hAPI driver to use. This allows the transparent and interchangeable use of different hardware timers via a mutual API. Binding all mandatory timer information directly to the device identifier eliminates the need for platform-dependent preprocessor defines and fosters code uniformity.

```
typedef struct utim_periph {
    const utim_t dev;
    const utim_driver_t *const driver;

    const uint16_t width    :8;
    const uint16_t channels :4;
    // 8< -----
} utim_periph_t;
```

Listing 1: Excerpt of the `utim_periph_t` struct. Its full definition is given in appendix Listing 10.

Timer instances can be addressed in a platform-independent fashion using abstract identifiers, such as *Timer 0*. They can also explicitly be addressed via their platform-specific names, such as *STM32_LPTIM2*. If the user application requests a specific timer that is either not available on the MCU or currently disabled, an error will be thrown during compilation. This guards against unexpected timer behavior after changing peripherals or target platforms. Selection of the timers that are made available to the application is done via *Kconfig* at compile-time. Exposed peripherals can interactively be selected and required drivers are automatically included. Configuration of API features and peripherals is described in detail in Section 5.2.6.

Counter values are represented by the `utim_cnt_t` type, defined in Listing 11. Its width can either be automatically determined to match the exposed peripherals or manually selected during compile-time. uTimer always uses the fastest minimum-width unsigned integer types, as defined in `<stdint.h>`. This accounts for speed benefits from matching the CPU word size[1].

5.2.3 Hardware-facing API

The hardware-facing API (hAPI) provides low-level drivers that directly interact with timer hardware registers. They share a common interface that is lightweight yet flexible. One such driver exists for every timer type, e.g., *general-purpose*, *low-power*, or *RTC*, that is used by at least one timer instance. Drivers are implemented as structs of the `utim_driver_t` type. They consist of minimal function sets, each represented as a group of function pointers. An overview of the available functions is given in Listing 2.

```
typedef struct utim_driver {
    int (*const init)(/*timer, freq, clk, cmp_cb, cmp_args, ovf_cb, ovf_args */);
    utim_propval_t (*const get_property)(/* timer, prop */);
    int (*const set_property)(/* timer, prop, value */);
    int (*const enable)(/* timer, run */);
    utim_cnt_t (*const read)(/* timer */);
    int (*const write)(/* timer, count */);
    int (*const set_channel)(/* timer, channel, mode, count */);
    #ifndef CONFIG_UTIMER_USE_REDUCED_API
    int (*const is_valid_freq)(/* timer, clk, freq */);
    #endif
} utim_driver_t;
```

Listing 2: Excerpt of the `utim_driver_t` struct. Its full definition can be found in appendix Listing 15.

Common basic features are directly accessible through designated functions, allowing for platform-independent use. Device-specific features, on the other hand, are exposed via a compact and flexible property interface that supports optional feature availability. This combination permits

[1] Integers with a width different from CPU word size can impair performance by requiring additional processor instructions during access and arithmetic operations.

comprehensive presentation of arbitrary timer features in a standardized way, independent of the underlying peripheral type. The hAPI is extensively specified to establish a sound pattern for uniform platform implementations. Defining drivers both board-specific and cross-platform allows to conveniently model peripheral similarities and aids code quality.

Timer Features and Property Access

uTimer distinguishes between *static properties* of the timer hardware, such as counter register width or channel count, and *dynamic properties*, such as counting mode or pending interrupts. While the former are constant, the latter can change at any time during firmware execution. Both types are made available by the hardware-facing API.

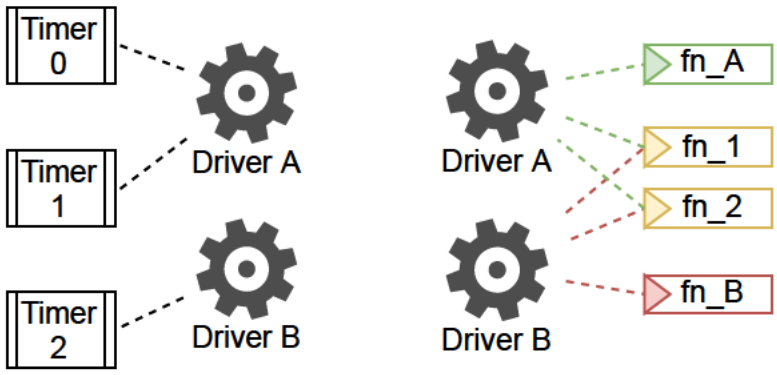
Static properties that are common to all timer types are encoded within the `utim_periph_t` structs, defined in Listing 1. System memory is preserved by combining multiple static properties into bit fields, whenever appropriate. Important hardware parameters are hereby directly bound to the timer instance itself. This provides a simple and convenient way to determine whether a peripheral is suitable for a specific task, e.g., based on its maximum counter value. Applications and high-level system modules can hereby dynamically adjust to the available hardware and utilize it to its full extent. This, for example, can be the dynamic distribution of timeouts, based on the number of available compare channels.

```
typedef enum {
    UTIM_PROP_MODE              = 0x01,  //!< Timer counting mode
    UTIM_PROP_CNT_DIR           = 0x02,  //!< Counting direction
    UTIM_PROP_INT_CMP_MATCH     = 0x03,  //!< IRQ generation on compare match
    // 8< -----
    UTIM_PROP_OVF_PENDING       = 0xF0,  //!< Overflow flag is set
    UTIM_PROP_CMP_MATCH_PENDING = 0xF1,  //!< Unhandled compare match pending
    // 8< -----
} utim_prop_t;

typedef struct utim_driver {
    // 8< -----
    utim_propval_t (*const get_property)(/* timer, prop */);
    int (*const set_property)(/* timer, prop, value */);
    // 8< -----
} utim_driver_t;
```

Listing 3: Excerpt of the Dynamic property interface that is provided by the hAPI. Available properties are encoded within the `utim_prop_t` enum and can be accessed using the listed driver calls. Its full definition can be found in appendix Listing 13.

Properties that are either timer-type specific or can change during run-time are instead exposed via a slim and straightforward interface, as defined in Listing 3. Available properties are encoded within the `utim_prop_t` enum. They are extended by property-specific value definitions, found



(a) Driver-granular code reusability (b) Function-granular code reusability

Figure 5.2: hAPI driver design concepts for code reusability

in appendix Listing 12. All properties can be read and written using the hAPI functions `get_property()` and `set_property()`.

Advanced features that are not available on every device are likewise exposed via this interface. These can be configured by writing the respective property and their current status can be determined by reading it accordingly. If a timer does not support a particular feature, it can be left unimplemented. Such cases are signaled via the return values of the access functions. This solution is a future-proof way to support new features as they become available without requiring changes to the application programming interface. This applies to both platform-specific and generic features alike.

Function Bundling

Maintaining a slim and lightweight interface is important for the hAPI. Providing individual functions for every timer operation inflates the interface and requires numerous pointers to be stored. The memory footprint that is inherent to the API itself would thereby significantly increase. The flexible property interface, described in the previous section, efficiently addresses this problem for all advanced and platform-specific features. However, some basic timer operations do not require individual functions, hence offer further optimization opportunities. Such closely related operations are bundled into single functions whenever appropriate. An example for this is the combination of `start()` and `stop()` into a single `enable(bool)` call. Even though this somewhat impairs the ease of use, the reduction in ROM consumption easily makes up for this small loss. Moreover, all bundled function calls are conveniently unbundled again within the uAPI, as described in Section 5.2.4.

Code Reusability

Both our timer hardware analysis and our survey of related work found that microcontrollers typically offer multiple timer peripherals of the same type (see Sections 4.3.3 and 4.1). The hardware-facing API accounts for this by allowing to map a single driver to multiple timer instances. An example of this is depicted in Figure 5.2a. Here, *Timer 0* and *Timer 1* share the same hAPI *Driver A*, while *Timer 2* is interfaced by *Driver B* instead. We refer to this concept as *driver-granular code reusability*.

Our analyses furthermore revealed that certain timer types can be available in multiple versions that share most of their features. The hAPI therefore enables OS developers to map a single implementation to multiple drivers, allowing selective re-use of driver functions. Figure 5.2b illustrates two timers that share parts of their driver code. Here, *Driver A* and *Driver B* share the functions *fn_ 1* and *fn_ 2*, but also possess individual implementations for the functions *fn_ A* and *fn_ B*. We refer to this as *function-granular code reusability*.

Virtual Drivers

In most cases, drivers interface hardware timers directly. *Virtual drivers* can instead also use other *base drivers* to control peripherals. They allow to freely mix direct hardware access and foreign driver calls within a single hAPI driver. In addition, functions of the base driver can be overwritten if necessary. An important use case for virtual drivers is the representation of multiple small timers as a large atomic timer instance. Such a scenario is illustrated in Figure 5.3.

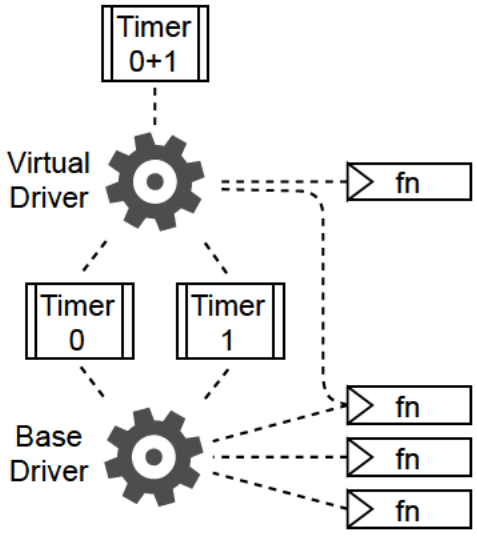


Figure 5.3: A virtual hAPI driver used for timer chaining

In the example above, *Timer 0* and *Timer 1* can be linked in hardware to form a single timer with an extended counter range. Both peripherals can individually be controlled using their base drivers. However, combining them requires additional configuration and special behavior during counter operations. The virtual driver implements the required control logic and automatically configures the peripherals accordingly. Both timers are then exposed as the atomic *Timer 0+1* instance. User applications and high-level system modules can transparently interface the chained timer via the existing API. However, removing the base timers from the set of exposed peripherals is strongly advised when using timer chaining. Simultaneous access to the same hardware timer via two different interfaces can lead to undefined behavior of the combined timer instance.

5.2.4 User-facing API

The user-facing API (uAPI) provides a single set of functions that abstracts from the underlying timer type. It is designed for convenient use by all components of the embedded firmware. Provided platform-agnostic timeouts allow the development of generic applications and high-level OS components, such as sensor drivers. Function calls are either directly delegated to the respective driver, such as read and write operations, or performed as compound operations of multiple subsequent hAPI calls, such as relative timer arming. The uAPI moreover defines additional convenience functions to enhance usability.

In contrast to the hAPI, the uAPI does not need to limit the number of functions it provides. Unused functions are automatically excluded during the build process and therefore do not negatively affect the final binary size. This allows to provide an extensive interface without raising memory consumption by default. A list of all uAPI functions can be found in Section B.3.

Generic Timer Functions

Basic functions that are found on all timer types are straightforwardly exposed by the hAPI. This includes peripheral initialization, counter operations, and compare channel configuration. The uAPI transforms this limited set of features into user-friendly functions. It autonomously determines the responsible driver and invokes the requested hAPI methods with appropriate arguments. The `utimer_read()` call, for example, executes the hAPI `read()` function, based on the respective timer driver.

Advanced Timer Functions and Status Information

The above discussed set of basic timer functions is extended by operations that are only available on a limited number of MCUs. We refer to such functions as *advanced timer functions*. These include widespread features like counting mode selection as well as peripheral-specific functions.

The hAPI exposes them via its slim but somewhat inconvenient property interface. The uAPI builds upon this base to provide designated functions for many advanced features. Whether a device supports a specific feature can be determined during run-time and is indicated via function return values. However, offering an individual function for each potential feature would quickly inflate the API. The user-facing API therefore continues to provide the property interface via the `utimer_get_property()` and `utimer_set_property()` functions. All not widely used features hereby remain available while the API is kept streamlined. This not only enables uTimer to expose up-to-date timer features, but also ensures future expandability. Whenever a feature becomes available on a relevant number of target devices, an exclusive uAPI function can easily be added for it.

Static attributes such as counter width or channel count, and run-time dynamic properties such as counting mode or pending interrupts, are moreover exposed by the hAPI. They are either stored within the timer instance struct or accessible via the property interface. The user-facing API can provide wrapper functions for convenient access to both types. These include, among others, `utimer_has_pending_ovf()`, `utimer_set_irq_ovf(bool)`, and `utimer_set_count_mode(utim_mode_t)`.

Compound Functions and Unbundling

Operations that can be achieved by a combination of multiple subsequent hAPI function calls are exclusively implemented within the uAPI. We refer to these as *compound functions*. An example of such is depicted in Figure 5.4. Here, relative timer arming is performed by (1.) reading the current counter value, (2.) calculating the target counter value, and (3.) arming a compare channel to the calculated value. Compound functions keep the driver structs compact by eliminating partially redundant functions.

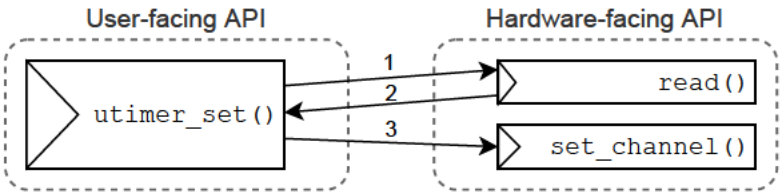


Figure 5.4: Exemplary uAPI compound function. Here, relative timer arming is implemented exclusively within the uAPI as a combination of two hAPI function calls.

Closely related to this is the unbundling of hAPI calls. Each function that was previously bundled within the hardware-facing API is now split into parts. The hAPI function `set_channel()`, for example, is unbundled into the four uAPI functions `utimer_set()`, `utimer_set_absolute()`, `utimer_set_periodic()`, and `utimer_clear()`. This relieves developers from

the somewhat inconvenient use of compressed functions. All unused functions again are automatically excluded from the firmware binary during build, hence do not waste system memory.

Convenience Functions

All above described uAPI functions always entail a call to the hardware-facing API. *Convenience functions* are an exception to this. They provide the user with out-of-the-box solutions for typical problems, but are not mandatory for timer use. These include, for example, the conversion between wall-clock time and timer ticks or checking whether a timer can be configured to a given frequency and, if not, determining the closest achievable one (see Section B.3.5). The set of convenience functions can be excluded as a whole during compile-time configuration.

Error Behavior

Sound hardware interfaces must define their behavior upon errors. This becomes particularly important when developing for a diverse range of target devices, each possessing different driver implementations. uTimer therefore extensively specifies the behavior upon hardware errors or invalid API use. All errors are signaled via negative function return values. Their causes are defined within the interface specification of the respective function for both all hAPI (see Section B.2) and all uAPI (see Section B.3) functions. A rich HTML documentation is automatically generated from the Doxygen[2] compatible definitions. These measures guide OS developers during the integration of new devices and ensure consistent API behavior across all platforms.

5.2.5 Interrupt Handling

Timers can generate interrupt requests to signal overflows or elapsed timeouts, i.e., their counter register reached a certain value. uTimer allows the attachment of separate user-defined callbacks for both compare match and overflow events. All mandatory maintenance tasks are performed by the hAPI driver within the corresponding low-level interrupt service routines. This happens automatically prior to user-callback execution.

Interrupt support is indicated by the static properties within every timer instance struct, and all interrupts can individually be (un-)masked during run-time. Associated function signatures and interrupt context definitions can be found in appendix Listing 14. The interrupt cause is indirectly determined by the invoked callback. With compare channel matches, the channel that triggered the compare match event is passed to the function as an argument. All callbacks furthermore allow to bind an arbitrary context that is made available during invocation. Separate callbacks for every channel are not natively supported, as discussed in Section 5.4.

[2]Doxygen source code documentation generator: `https://www.doxygen.nl/` (Accessed: 21.02.2022)

5.2.6 Configuration Management

Establishing a uniform scheme for peripheral configuration is just as important as its convenient use. We distinguish between *compile-time* and *run-time configuration*. The former includes all settings that cannot be altered after the firmware binary is built, such as the exposed timer peripherals. The latter instead includes all properties that can be changed during run-time, such as timer frequency.

We use Kconfig [36] to provide a solid and user-friendly management of compile-time configurations. This solution seamlessly integrates with the existing configuration system of RIOT. Kconfig allows to define both platform-specific and global settings in a uniform way. Stored data is easily accessible from all parts of the firmware. It can either conveniently be altered during build via the console-based user interface or statically be set via a target configuration file. Figure 5.5 shows the interactive selection of timer peripherals during build.

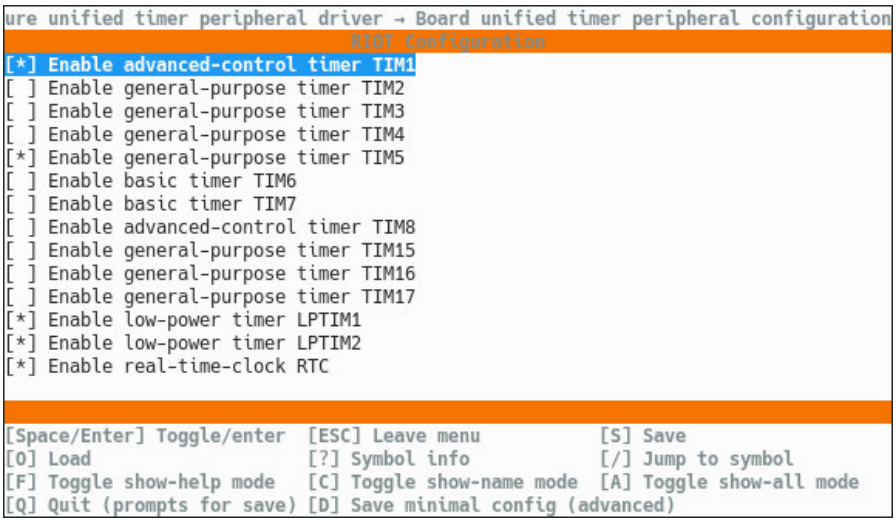


Figure 5.5: Interactive selection of timer peripherals using Kconfig

Configurable parameters are defined within designated `Kconfig` files. They allow to specify data type, parameter range, and dependencies. Entries can be created on different hierarchical levels, allowing system-wide, CPU-specific, and board-specific settings to be modeled alike. We use this to provide device-family wide peripheral definitions that are automatically adjusted to the target board. Generic settings, such as overwriting the counter type width, are employed on a system-wide scale instead. This not only significantly reduces code duplication, but also fosters a uniform and maintainable configuration management.

The proposed API also enables applications to reconfigure timers during operation. All run-time configurations can be performed via the user-facing API. However, one parameter that requires special attention is the timer clock, with which two crucial restrictions apply. First, explicitly

using a platform-specific clock comes at the cost of loosing application portability. uTimer copes with this by allowing to select clock sources as either a generic clock class, such as *high-frequency*, *low-power* and *default*, or explicitly as a platform-specific clock. Second, altering a clock source during run-time must not affect other timers or peripherals in any way. Certain clock configurations can therefore be unavailable, based on the target hardware and the configuration of other peripherals. We continue to discuss this problem in-depth within Section 5.4.2.

5.2.7 Application Portability

Operating systems provide user applications with the necessary abstraction to run independently of the underlying hardware. With embedded systems, this becomes particularly important and challenging at the same time. OSs like RIOT are expected to support a highly heterogeneous range of hardware targets while ensuring identical behavior on all MCUs. Being able to write code in a platform-independent fashion fosters reusable software solutions, widens the range of compatible microcontrollers, and can in turn lower development costs as well as the time-to-market [31]. On one hand, peripheral APIs must therefore put on enough abstraction to satisfy the requirements for application portability. Too much abstraction, on the other hand, quickly turns negative because it narrows the exposed feature set and impairs system performance. This problem is in detail described in Section 3.2.

uTimer implements effortless cross-platform portability of user applications in multiple ways. Generic peripheral identifiers and the class-based clock selection allow for platform-independent timer configurations. The abstract uAPI can transparently interchange different timer types, also across target devices. All functions that are not common to every peripheral are exposed via the flexible property interface. It permits comprehensive interchangeability, as long as all required timer features are supported by the targeted microcontroller. Platform-agnosticism is only fully relinquished once the application developer actively decides to use explicit peripheral or clock mappings. In this case, the timekeeping code is bound to the current target device and peripheral mappings must be manually adjusted whenever changing the MCU. The extensive specification of the application programming interface ensures its consistent behavior. Although not recommended for new applications, uTimer can also coexist with current low-level timer APIs. This facilitates the incremental migration of old firmware to the uniform API.

5.3 Usage Examples

This section presents two simple code examples that use the uTimer API to control low-level timers. Error handling is omitted within both examples to keep them concise.

```c
#include "periph/utimer.h"

static void cb(void *arg, int channel) {
    (void) arg;
    printf("Callback executed. channel=%d\n", channel);
}

int main(void) {
    utim_periph_t timer = utimer_get_periph(0);
    utimer_init(&timer, TIMER_SPEED, UTIM_CLK_DEFAULT, &cb, NULL, NULL, NULL);
    utimer_set(&timer, 0, 10000);
    utimer_start(&timer);
    puts("Timer 0 armed and started.");

    return 0;
}
```

Listing 4: Code Example: Platform-independent use of a single timer to generate an absolute timeout. The attached callback function is executed upon compare match.

```c
#include "periph/utimer.h"

static void cb(void *arg, int channel) {
    (void) arg;
    printf("Callback executed. channel=%d\n", channel);
}

int main(void) {
    utim_periph_t timers[] = {
        utimer_get_periph(TIMERS_STM32_TIM5),    // 32-bit, 4 CC, general-purpose
        utimer_get_periph(TIMERS_STM32_TIM1),    // 16-bit, 4 CC, advanced-control
        utimer_get_periph(TIMERS_STM32_LPTIM1), // 16-bit, 1 CC, low-power
        utimer_get_periph(TIMERS_STM32_LPTIM2)  // 16-bit, 1 CC, low-power
    };

    for (size_t i = 0; i < ARRAY_SIZE(timers); i++) {
        utim_periph_t* tim = &timers[i];
        utimer_init(tim, TIMER_SPEED, UTIM_CLK_DEFAULT, &cb, NULL, NULL, NULL);

        for (size_t chan = 0; chan < tim->channels; chan++) {
            utimer_set(tim, chan, 10000);
            printf("Timer %d channel %d armed.\n", tim->dev, chan);
        }

        utimer_start(tim);
        printf("Timer %d started.\n", tim->dev);
    }

    return 0;
}
```

Listing 5: Example Code: Interchangeable use of different timer types. Peripherals are explicitly selected among platform-specific timers. All available compare channels are automatic armed, based on the static properties of the respective timer instance.

An introductory code example is given in Listing 4. Here, the first generic timer instance is initialized and set to wait for 10000 timer ticks. It subsequently is started and the main application halts. The callback function `cb` is asynchronously invoked upon timeout and prints the number of the compare channel that triggered it.

The interchangeable use of different platform-specific timers is demonstrated in Listing 5. A total of four timer instances are explicitly addressed here. The set consists of one general-purpose, one advanced-control, and two low-power timers from the STM32 device family. This explicit selection limits the application compatibility to all devices that possess at least the selected peripherals and causes the automatic use of a 32 bit wide counter value type. Requesting timers that are not available on a desired build target raises a respective error during compilation. Each timer is initialized to a common counting frequency, using its type-specific default clock. It is followed by the arming of all available compare channels and the subsequent timer start. This produces a total of six callback executions for the chosen set of timers.

5.4 Discussion

Various trade-offs had to be weighed during the design of uTimer. The most important of these are discussed in the first part of this section. Even though the proposed API was developed with great care, certain issues remain. The second part of this section therefore focuses on implications that arise from certain design decisions, as well as generic limitations of the architecture itself. However, only conceptual aspects that are inherent to the API design are discussed here. The run-time performance of uTimer is separately assessed by the comprehensive benchmarks we develop in Chapter 7.

5.4.1 Design Trade-offs

Compare match (CMP) and overflow (OVF) callbacks are separated due to their disjoint use cases. However, the compare match callback is not further split into individual channel-specific functions because only 31 % of all timer types provide distinct interrupts for every compare channel. Even though applicable platforms may experience a slightly lower timeout latency, the remaining majority (69 %) only suffers the additional memory overhead for storing respective function pointers. We found the potential performance gain on some devices to not nearly outweigh the general drawbacks for all platforms. Individual functions can nonetheless still be dispatched within the user-defined callback, whenever required.

All timer functions are registered within hAPI driver structs which in turn are assigned to timer instances. Each uAPI call must therefore resolve the corresponding driver upon invocation, hereby introducing pointer dereferencing overhead. Functions could instead be mapped directly

within each timer instance struct, which would remove a layer of indirection. However, this not only significantly increases the memory footprint, but also results in a non-negligible amount of duplicate code. We therefore decided against this performance optimization. The exact overhead caused by the different abstraction layers of uTimer is quantified in Section 7.4.5.

Bundling hAPI functions impairs their ease of use but reduces driver struct size. Because the hardware-facing API is only rarely used by applications directly, the improved resource efficiency undoubtedly outweighs the somewhat reduced convenience. The memory footprint of uTimer is quantified during our Evaluation in Section 7.5.3. This includes a breakdown of all contributing factors, including hAPI function pointers.

Exposing widely available advanced functions and timer-type specific features via the property interface likewise impedes usability of the hardware-facing API. These functions could instead be implemented directly as individual driver calls, allowing easy access and removing the additional layer of indirection. However, this would again entail a massive increase in memory requirements. Every driver would need to store a pointer for each potential timer function, regardless of whether the interfaced peripheral actually supports it.

The width of the counter value type `utim_cnt_t` is defined globally. Smaller timers are thereby forced to use superfluously wide data types, whenever required by at least one exposed timer. Being unaware of such counter width differences can theoretically lead to unexpected behavior during bit shifting and delay desired integer overflows. Using each timer with its exact width requires separate function calls for every timer width that is available. Assuming all unused functions are excluded during build, this still requires a minimum of 3 uAPI function sets on 57 % of the evaluated MCUs (see Table 7.2). This is not feasible because it drastically limits peripheral interchangeability, impairs application portability, and greatly inflates the user-facing API without any significant gain in functionality.

5.4.2 Issues

In Section 5.2.6 we already stressed that altering timer clock sources during run-time entails two potential problems. One of them is to ensure that a clock change does not affect other hardware timers or system components. Determining the impact domain of such reconfigurations requires an examination of the clock tree. The following example illustrates this problem:

Given a timer clock *TIMx_CLK* can be driven by three different sources, namely *CLK_A*, *CLK_B*, and *CLK_C*. If *TIMx_CLK* is only used by a single hardware module, reconfiguring it does not cause any side effects and can therefore be performed at any time without restrictions. In such cases, applications are free to select among the following clock sources: *CLK_A*, *CLK_B*, and *CLK_C*. If *TIMx_CLK* is used by multiple timers or other peripherals instead, changing it for one timer will likewise change it for all other peripherals. Now it is no longer possible to freely

chose among the full range of clock sources. The clocking options of all affected timers therefore must be limited to only *TIMx_ CLK*. In other words, when traversing the hierarchical clock tree, only those clocks are applicable that lay on the path originating from the timer peripheral up to the first branch that leads to other peripherals.

One solution to this problem is proposed by Rottleuthner *et al.* [50]. FlexClock combines a lightweight yet flexible scheme for modeling microcontroller clock trees with a generic software implementation. It is specifically designed for low-power and resource constrained IoT devices. This novel solution makes clock trees explorable during run-time and manages the dynamic reconfiguration of all system clocks. We are very much in favor of using techniques like this to allow dynamic and platform-independent clock selection. Because such changes not only affect timers but all device peripherals alike, we suggest implementing them at a system-wide scale. uTimer therefore does not evaluate clock trees, hence is unable to automatically determine possible clock configurations. As a consequence, appropriate clock sources must be specified by the operating system developer during implementation. Effects of this design decision are revisited during our Evaluation in Section 7.5.9.

Mutual exclusive use of hardware timers is likewise not currently ensured by uTimer. This can lead to undefined behavior if a single timer instance is used simultaneously by multiple system components. As with clock management, peripheral allocation should not be implemented by every low-level API itself. Hence, the need for an external resource allocator component arises. Nonetheless, this remains a currently still unsolved problem when using uTimer in RIOT OS.

We explained our decision to enforcement a global counter value type width in the previous section. Closely related is the transparent interchangeability of different timer types, which is one key feature of the proposed API. Offering a single uniform interface likewise requires time to be represented in a uniform way too. uTimer achieves this by using counter ticks as a common base unit. While this approach is fine for most timers, some real-time clocks expose their current ”counter value” as wall-clock time instead. Conversion between timer ticks and alternative time representations therefore becomes mandatory. The uniformity of the uAPI, however, contradicts a transparent conversion of time formats and hides the wall-clock time from user applications. Even though respective helper functions can aid in usability, the conversion overhead remains inevitable. Separate counter-type based interfaces solve this problem, but were decided against in the preceding trade-off section.

6 Implementation

Our implementation of the proposed API design is divided into a platform-independent operating system module and six platform-specific packages. The latter include all low-level drivers for the timer types of the respective MCUs, already providing support for a total of 129 boards. We use RIOT release `2021.07`[1] as the code base on which to build all our changes.

This chapter first presents the software architecture of RIOT and which parts of it are touched upon by uTimer in Section 6.1. All hardware-independent aspects of the `periph_utimer` module are addressed in Section 6.2. Section 6.3 continues by discussing each of our six platform implementations individually, highlighting important aspects we found during the development of hAPI drivers and within existing low-level timer drivers. This chapter closes with an automated validation of the contributed software components in Section 6.4. The performance of uTimer is evaluated separately in Chapter 7.

6.1 RIOT System Architecture

The structure of the RIOT code base is illustrated in Figure 6.1. It is divided into four layers that build upon each other. All parts that are relevant for uTimer are highlighted in green.

Starting from the device hardware, `boards` and `cpus` are the first components. Peripheral configurations, pin mappings, clock settings, and certain peripheral-specific initialization routines are placed inside the `boards` directory for each board the is supported. uTimer extends these definitions with a registry of all timer peripherals that targeted devices provide. Each board maps to exactly one CPU, as defined in the `cpu` directory. It contains all CPU-specific code, including processor configurations, interrupt handling routines, startup code, and clock initialization. Both `cpu` and `boards` allow entities to share parts of their code via designated directories, identified by a `_common` suffix. It is strongly encouraged to place code at the highest possible level to minimize redundant segments. The `periph` subdirectory constitutes the next hardware-dependent layer. It contains implementations of low-level peripheral drivers for each individual

[1] RIOT release `2021.07`: `https://github.com/RIOT-OS/RIOT/releases/tag/2021.07` (Accessed: 06.02.2022)

[2] This illustration is largely based on the following image: `https://doc.riot-os.org/riot-structure.svg` (Accessed: 05.02.2022)

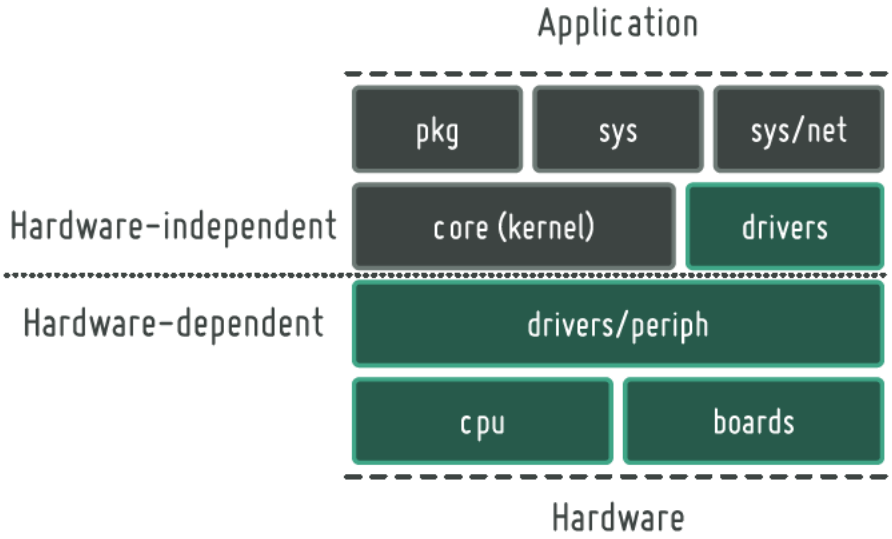


Figure 6.1: Structural overview of the RIOT OS. Components are displayed in their hierarchical order from the device hardware up to the user application. All areas that are affected by our implementation of uTimer are highlighted in green[2].

CPU, including `periph_timer/rtc/rtt`. uTimer is likewise implemented here as the exclusive `periph_utimer` module, containing all hAPI drivers for the respective MCU.

The first hardware-independent layer comprises `drivers` and the system `core`. The former directory contains drivers for external devices such as sensors, network interfaces, and radios. All device drivers rely on the low-level `periph` implementations to interface external hardware in a platform-independent fashion. The `periph_common` subdirectory allows extending the CPU-specific `periph` implementations by additional platform-agnostic components. It is predominantly used for common peripheral driver initialization, to implement compound functions, and to provide utility code. We therefore implement the user-facing API here.

All remaining components are not related to uTimer and therefore are only briefly outlined below. The `core` directory contains the RIOT kernel. It provides scheduling, threading, and inter-process-communication as well as all corresponding data structures and type definitions. Above that are `pkg` and `sys`, including the `sys/net` subdirectory. All external libraries and respective patches reside inside the `pkg` directory. Following the micro-kernel architecture of RIOT, `sys` contains all system modules that implement platform-independent functions besides device drivers, such as cryptographic operations or the shell. The comprehensive generic (*GNRC*) network stack of RIOT resides in its exclusive `net` subdirectory [33].

User applications are implemented on top of all four operating system layers but can access each layer individually whenever required. The described components are further extended by example

applications (examples), user documentation (doc), various automation scripts (dist), and unit tests (tests). We extend the latter by porting the full set of existing low-level timer test suites to uTimer as well as developing unit tests for all new API functions.

6.2 Low-level Timer Module

uTimer is implemented as a self-contained peripheral driver module, namely periph_utimer. It can be included during build by adding it to the list of used modules via USEMODULE += periph_utimer in the application Makefile. Other low-level timer drivers can be used in conjunction with it, but a single hardware timer should only be used through one of the interfaces to avoid undefined behavior.

This section exclusively describes interface definitions and common code segments of uTimer, placed inside the drivers/periph_common directory. First, the implementation of compile-time module configuration is discussed. It is followed by datatype definitions and associated defaults. The section finishes with all platform-independent aspects to both the hardware-facing and user-facing APIs.

6.2.1 Compile-time Configuration Management

All uTimer settings, including the selection of timer peripherals, are performed via Kconfig and thus blend seamlessly with the uniform configuration management of RIOT. The design and capabilities of this system are described in Section 5.2.6. This section focuses on its implementation instead. Kconfig files are hierarchically structured, as illustrated in Figure 6.2.

Generic API settings are defined inside a designated Kconfig.utimer file that resides in the drivers/periph_common directory. Entries include, for example, the global counter type width overwrite or the exclusion of uAPI convenience functions. All definitions are automatically sourced into the main peripheral Kconfig to fulfill the *separation of concerns* software design principle. It in turn is loaded by the driver configuration file that finally is included inside the root Kconfig.

Timer peripherals are defined within the CPU-specific configuration files. They list every hardware timer that is part of a microcontroller family[3] at a central location. Boards select these definitions inside their board-specific Kconfig files to indicate the subset of actually available peripherals. The list of low-level timers that can be used by the application is then automatically generated prior to compilation, based on the BOARD variable. This supersedes the cumbersome

[3]Some manufacturers use standardized peripheral names throughout all their device families. In such cases, available peripherals are defined within a common package for the respective manufacturer.

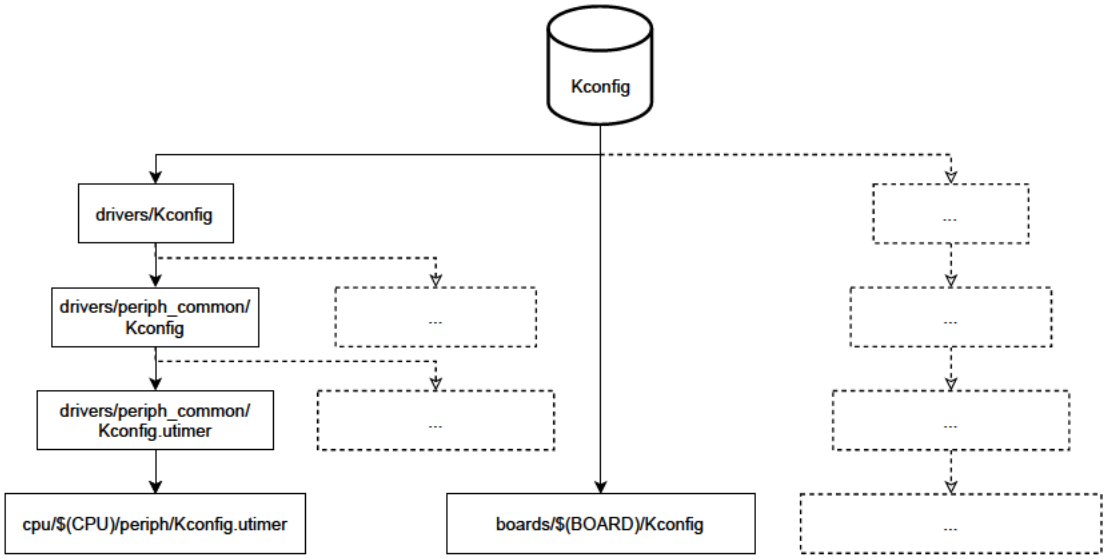


Figure 6.2: Hierarchical structure of the Kconfig definition files

and error-prone peripheral configuration via header files and relives application developers from looking up available timers within vendor datasheets.

6.2.2 Datatype Definitions and Defaults

Common data types are defined inside the module header file `drivers/include/periph/utimer.h`. It provides defaults for all types, which can be overwritten if necessary. However, certain data types and enumerations should be defined individually by the platform-specific implementations. Otherwise, limited fallback definitions will be used instead.

An unsigned integer is used as the standard peripheral identifier `utim_t`. Its maximum value `UINT_MAX` is reserved by default to indicate an invalid device. The actual width of the `unsigned int` type varies between processor architectures, but is at least 8 bits. This leaves more than enough room to identify all timer peripherals. The counter value type `utim_cnt_t` should be selected individually by the CPU implementations. If this is not the case, `uint_fast32_t` is used as a fallback. When the corresponding compile-time option is set, both fallback and platform-specific selections will be overwritten (see appendix Listing 11). Available clock sources shall likewise be provided by the platform implementations. Otherwise, `UTIM_CLK_DEFAULT` remains the only possible selection (see appendix Listing 12).

All attributes that are part of the flexible property interface are registered within the `utim_prop_t` type, as defined in appendix Listing 13. Each entry is identified by a distinct hexadecimal value. The enumeration is split into peripheral-wide and channel-specific properties, as indicated

by the most significant bit. If the MSB is set, the following five bits indicate the channel number and all remaining bits define the respective channel property. Two examples of this encoding are given in Listing 6. Example 1 encodes the *Mode* property of timer channel four and Example 2 encodes the *Compare Match Pending* status flag of timer channel two. The interface distinguishes between 1024 properties and 32 individual channels. This even allows modeling all 12 channels of the Texas Instruments LM4F120 (see Table A.13) and leaves enough room for future extensions.

```
/** Prefix indicating a channel property in utim_prop_t */
#define UTIM_PROP_CHAN_PFX      0x8000
/** Channel number bits in utim_prop_t */
#define UTIM_PROP_CHAN_NUM_Pos  10
#define UTIM_PROP_CHAN_NUM_Msk  (0b11111 << UTIM_PROP_CHAN_NUM_Pos)
/** Channel property indicator bits in utim_prop_t */
#define UTIM_PROP_CHAN_PROP_Pos 0
#define UTIM_PROP_CHAN_PROP_Msk (0b1111111111 << UTIM_PROP_CHAN_PROP_Pos)

typedef enum {
    // 8< -----
    /** Example 1: Mode of timer channel 4
     *
     *  -Prefix-    -Channel 4-      -Mode-
     * ((0x8000) | (0x04 << 10) | (0x00 << 0)) = 0x9000
     *                                         = 0b 1001 0000 0000 0000
     */
    UTIM_PROP_CHAN4_MODE = (
        (UTIM_PROP_CHAN_PFX) |
        (0x04 << UTIM_PROP_CHAN_NUM_Pos) |
        (0x00 << UTIM_PROP_CHAN_PROP_Pos)
    ),

    /** Example 2: Compare match status of timer channel 2
     *
     *  -Prefix-    -Channel 2-      -CMP-
     * ((0x8000) | (0x02 << 10) | (0x01 << 0)) = 0x8801
     *                                         = 0b 1000 1000 0000 0001
     */
    UTIM_PROP_CHAN2_CMP_MATCH_PENDING = (
        (UTIM_PROP_CHAN_PFX) |
        (0x02 << UTIM_PROP_CHAN_NUM_Pos) |
        (0x01 << UTIM_PROP_CHAN_PROP_Pos)
    ),
    // 8< -----
} utim_prop_t;
```

Listing 6: Encoding of channel-specific properties within `utim_prop_t`

6.2.3 Hardware-facing API

The hardware-facing API is likewise specified within the common module header file. The `utim_driver_t` type encapsulates all function pointers that make up an hAPI driver. They are listed in full detail within appendix Section B.2. Their implementation is entirely performed by the platform-specific code inside the `cpu` directory, which is why `utimer.h` only defines the generic hAPI interface.

It is extended by supplementary type definitions that provide named values for certain function arguments, such as channel operation modes. The signatures for compare match and overflow callbacks are also defined at this place, including a default interrupt context type for both events (see appendix Listing 14). It facilitates implementations to bind the optional argument pointer directly to the respective callback function.

6.2.4 User-facing API

All functions of the user-facing API are fully implemented by the common uTimer `periph` module. The only exception to this is `utimer_get_periph()`. Since platform developers are free to choose how they store peripheral instances, the corresponding access function must reflect this accordingly. Its implementation is therefore left to the individual platforms themselves.

The signatures of all uAPI functions are defined in the module header file `drivers/include/periph/utimer.h`. A complete list of these can be found in appendix Section B.3. Interface specification, error handling, and documentation are either referenced from the corresponding hAPI functions or defined here, as is the case with compound and unbundled functions. All uAPI code resides inside the `drivers/include/periph_common/utimer.c` file.

6.3 Platform Implementations

To date we have implemented uTimer for six different microcontroller platforms. This makes the novel low-level timer API already available on a total of 129 boards. All supported device families are listed in Table 6.1. These were selected based on their timer hardware diversity, popularity, and availability.

Each platform-specific uTimer implementation has to ...

a) implement a `utim_driver_t` hAPI driver for every timer type that is available.
b) create and store the `utim_periph_t` instance structs for all selected timers.
c) implement the `utim_get_periph()` uAPI function for exposing available timer instances to the user application.

CPU	Boards [#]
atmega	16 boards
efm32	14 boards
esp32	$\geq$ 10 boards
esp8266	$\geq$ 3 boards
msp430fxyz	4 boards
stm32	82 boards
	129 boards

The esp32-wroom-32 and esp8266-esp-12x boards are designed to be used with various generic ESP32/ESP8266 modules.

Table 6.1: Total number of boards with uTimer implementations

d) configure all necessary device interrupts and provide corresponding low-level ISRs.
e) provide the required platform-specific type definitions.
f) supply Kconfig peripheral mappings for supported boards.

Low-level hAPI drivers reside inside the `cpu/$(CPU)/periph` directory for each respective CPU. The `utimer.c` source code file contains all hAPI functions, creates driver structs, and sets up timer instances based on the Kconfig peripheral selection. The well-known `UTIMER_NUMOF` preprocessor macro, which indicates the total number of available timer instances, is also automatically calculated by the latter. Whenever multiple MCUs have the same timer type, its associated driver is shared among them and therefore implemented within a common module.

The remaining files are either placed inside the `cpu` or `boards` directory, depending on the genericity of the respective platform. As long as an implementation does not require to distinguish between boards, source code and system headers are preferably placed within the `cpu` folder. The `boards` directory should only be used if inevitable to prevent unnecessary code duplication and facilitate software maintenance. When used, it typically contains board-specific mappings of interrupt vectors and peripheral base addresses. It is normally only required on large and highly diverse platforms, such as the STM32 microcontroller series. Platform-specific type definitions, such as available clock sources, are appended to the existing peripheral configuration header files `periph_cpu.h` and `periph_conf.h`. Data structures for storing timer instances and all available hAPI drivers are provided within the new `periph_utimer_common.h` header.

The remainder of this section briefly discusses each platform individually. In doing so, we will go into details that stood out during development of the respective uTimer implementation as well as certain aspects of existing low-level timer drivers. The platforms are discussed in their lexicographical order.

6.3.1 Atmel / Microchip ATmega AVR

The Microchip ATmega AVR series, formerly produced by Atmel, is the predominant processor family on Arduino boards. It is therefore a popular choice among hobbyists, but ATmega MCUs are also widely found in the industry due to their cost-efficient design. It is the only platform that provides separate interrupt vectors for every compare channel[4].

Low-level peripheral code is exclusively placed within the `atmega_common` package, which is included by all ATmega CPUs during build. While implementing periodic timeouts for uTimer, we noticed a problem within the existing `periph_timer_periodic` module. Periodic timeouts can be configured to reset the counter value upon compare match, as indicated by the `TIM_FLAG_RESET_ON_MATCH` option. However, the existing implementation failed at restoring it after timer restarts, causing an invalid counting mode, called *CTC*, to be set. As a result, the counter value is not reset after the first compare match and the timer continues to count until it overflows. In such cases, no periodic timeout events are generated until the counter value fully wrapped around and reached the configured compare match threshold again. We contributed a fix for this issue with pull request #17387[5].

We moreover identified a peripheral allocation conflict between the `periph_rtt` and `periph_rtc` modules. Both use *TIMER2* to implement their APIs upon. This results in undefined timer behavior when the conflict remains unnoticed by the developer and both modules are used within the same application.

6.3.2 Espressif ESP8266 and ESP32

The ESP8266 and ESP32 microcontrollers, manufactured by Espressif, provide low-cost access to Wi-Fi and Bluetooth communication. Compared to other devices, they possess a sophisticated system architecture that is backed by an extensive vendor-provided software development kit (SDK). However, their timer peripherals are very limited in both number and features.

Even though their peripheral drivers are separated within RIOT, timers of both MCUs are very similar. The vendor SDK can be used to access general-purpose timers, but only exposes a small feature set. We therefore implement all timer drivers via direct hardware register access instead. During development of the clock configuration, we noticed that although prescalers are available on all general-purpose timers, `periph_timer` only supports a fixed 1 MHz counting frequency. This may be sufficient for many applications, but it drastically limits peripheral configuration options. As on most other platforms, we found that some peripherals are simultaneously exposed via multiple of the existing low-level APIs.

[4] The ESP8266 and ESP32 are excluded from this assertion because they only offer a single compare channel and have no overflow interrupt support.

[5] See: `https://github.com/RIOT-OS/RIOT/pull/17387` (Accessed: 04.01.2022)

6.3.3 Silicon Labs EFM32

The Silicon Labs EFM32/EFR32 platform combines powerful Cortex processor architectures with a comprehensive selection of timekeeping peripherals. It, for example, provides the designated ultra low-power *Cryotimer*, which unfortunately is not usable via any of the existing peripheral interfaces. Furthermore, despite the availability of a designated real-time counter, current APIs expose solely the real-time clock through both `periph_rtt` and `periph_rtc`.

Timer chaining is implemented on this platform to test the virtual hAPI drivers of uTimer, as introduced in Section 5.2.3. We use the general-purpose *TIMER0* as a configurable prescaler that is then fed into *TIMER1*. The combined peripherals are presented to the user application as an atomic timer instance, which can be used interchangeably with other types. This structure yields a greatly extended counter range and enables comprehensive fine-tuning of the counting frequency. It likewise applies to the wide general-purpose timers *WTIMER0* and *WTIMER1*, allowing for counter ranges up to 2^{64} ticks. The example in Listing 7 demonstrates the creation of both regular and virtual timer instances. It is strongly recommended to use either the chained timer instance *TIMER01* or contained timers individually. Simultaneous use of both results in undefined behavior. Such selections can either be completely prohibited via Kconfig definitions or the application developer can be notified by a preprocessor warning as in our example.

During development, we noticed that `periph_timer` is able to expose two different timer types on this platform, namely *General-purpose* and *Low-energy*. While this is desirable from a feature perspective, it has one crucial consequence. The peripheral driver thereby needs to distinguish between the two types within each function call individually. This not only impairs code maintainability but also lowers performance, as confirmed by our benchmarks in Section 7.4. uTimer circumvents this explicit distinction by implicitly determining the peripheral type through the associated hAPI driver.

6.3.4 STMicroelectronics STM32

The STM32 microcontroller family possesses the most comprehensive range of timer peripherals among all targeted platforms. This implementation alone provides support for a total of 82 boards. Devices provide various general-purpose, advanced-control, basic, and low-power timers as well as up to two real-time clock peripherals per MCU. All hAPI drivers are implemented inside the common `stm32` package and are selected accordingly within the individual board mappings. Problems with the Kconfig files for the `nucleo-l152re` board, for example, could be fixed during uTimer implementation by our pull request #16999[6].

[6] See: `https://github.com/RIOT-OS/RIOT/pull/16999` (Accessed: 04.01.2022)

```
const utim_periph_t timer_periphs[] = {
    #if CONFIG_EFM32_PERIPH_UTIMER_USE_TIMER0
    {
        .dev=TIMERS_EFM32_TIMER0,
        .driver=&UTIMER_DRIVER_GPTIMER,
        .width=16,
        .channels=3,
        .int_ovf=true,
        .int_cmp_match=true
    },
    #endif /* CONFIG_EFM32_PERIPH_UTIMER_USE_TIMER0 */

    #if CONFIG_EFM32_PERIPH_UTIMER_USE_TIMER1
    {
        .dev=TIMERS_EFM32_TIMER1,
        .driver=&UTIMER_DRIVER_GPTIMER,
        .width=16,
        .channels=3,
        .int_ovf=true,
        .int_cmp_match=true
    },
    #endif /* CONFIG_EFM32_PERIPH_UTIMER_USE_TIMER1 */

    #if CONFIG_EFM32_PERIPH_UTIMER_USE_TIMER01
        #if (CONFIG_EFM32_PERIPH_UTIMER_USE_TIMER0 || \
             CONFIG_EFM32_PERIPH_UTIMER_USE_TIMER1)
            #warning "TIMER0 or TIMER1 are exposed but TIMER01 is selected."
        #endif /* TIMER01 && (TIMER0 || TIMER1) */
    {
        .dev=TIMERS_EFM32_TIMER01,
        .driver=&UTIMER_VDRIVER_GPTIMER_CHAINED,
        .width=32,
        .channels=3,
        .int_ovf=true,
        .int_cmp_match=true
    },
    #endif /* CONFIG_EFM32_PERIPH_UTIMER_USE_TIMER01 */

    // 8< -----
};
```

Listing 7: Creation of regular and virtual timer instances. Their mutual exclusive use is encouraged by a respective preprocessor warning.

Although most timer types behave identical, certain ones differ slightly between device subseries. An example for this is the real-time clock. Besides the usual variations in register names and addresses, the STM32F1 series requires a different RTC driver code than others. On said platform, a peripheral allocation conflict between `periph_rtc` and `periph_rtt` exists.

The large diversity of timers can also lead to false assumptions about the available hardware. Existing timer APIs, for example, assume that "all STM timers have 4 capture-compare channels"[7]. However, this is not the case as our hardware analysis revealed. General-purpose timers can have between one and four, advanced-control timers either four or six, and low-power timers only a single compare channel, see Table A.1 for full details. This prevents their exhaustive utilization via existing interfaces and results in erroneous timer behavior whenever peripherals with less than four channels are used. uTimer addresses this by binding static hardware properties to each timer instance.

6.3.5 Texas Instruments MSP430

Hardware timers on the Texas Instruments MSP430x platform are rather simplistic. Besides its watchdog, the microcontroller only offers two 16-bit general-purpose timers. These namely are *TIMER_A*, which provides three compare channels, and *TIMER_B*, which provides a total of seven channels. Neither of both supports range extension through timer chaining.

Existing low-level timer APIs are only capable of exposing *TIMER_A* via `periph_timer`. *TIMER_B* remains unusable and none of the other low-level timer modules are implemented. Since this uTimer implementation was straight forward on the one hand and the last one we conducted on the other, no further noteworthy aspects stood out.

6.4 Validation

To validate our API implementations, all existing low-level timer test suites of RIOT were ported to `periph_utimer`. This was accomplished by replacing each `periph_timer` call with the corresponding uTimer function. Test concepts and procedures were left unchanged. Existing test suites assess fundamental timer functions, one-shot timeouts on all channels, periodic timeouts, and include a robustness test for very short timeouts. We developed additional test suites for features that are unique to uTimer, such as counter register writes and overflow interrupt handling. To validate advanced peripheral features, further device-specific tests were conducted. These naturally only succeed on a subset of our targeted platforms, since hardware support for the assessed feature is required. Virtual timer drivers, for example, were tested by exposing a

[7] See source code file `cpu/stm32/include/periph_cpu.h`: https://github.com/RIOT-OS/RIOT/blob/2021.07-branch/cpu/stm32/include/periph_cpu.h#L104 (Accessed: 07.02.2022)

pair of chained hardware timers as a single atomic timer instance to the user application, as described in Section 6.3.3. Behavioral equivalence was verified by executing the identical test suites on both the virtual and all other non-virtual timer instances of the assessed MCU.

Both types of tests revealed small implementation errors, which then could be fixed prior to conducting our benchmarks, as described in Chapter 7. All test suites are platform-independent and the provided Continuous Integration (CI) support allows their automated execution. uTimer tests therefore integrate seamlessly into the existing development workflow of RIOT. They hereby not only reveal errors during the initial driver implementation phase, but also allow regression testing to detect bugs and possible side effects within nightly builds and pending pull requests.

7 Evaluation

Proper behavior of our uTimer implementations was verified by unit tests in the previous section. Within this chapter, we continue by assessing its performance and comparing it to the current low-level timer APIs of RIOT. First, we provide a brief introduction to the applied concepts and techniques, followed by a detailed description of our evaluation setup. We develop benchmarks that quantify the performance of both uTimer and `periph_timer`. These assess basic timer operations, compare channel timeouts, interrupt handling, and the platform-independent abstraction overhead that is inherent to the APIs themselves. All measurements are conducted in an automated fashion by using a hardware in the loop testbed with CI support. This chapter finishes with a discussion of benchmark results, issues, memory requirements, and peripheral diversity as well as qualitative aspects, including usability, code quality, and influence on the systems development life cycle (SDLC).

7.1 Concepts

In this section, we briefly introduce the main concepts and techniques applied in our evaluation. Their use and how we integrate them into our test setup is described in detail in Section 7.3.

7.1.1 Hardware in the Loop

Hardware in the Loop (HiL) testing is a technique where a programmable hardware controller is used to stimulate a device under test (DUT). The HiL controller acts as an external reference device and is capable of generating arbitrary input signals on the GPIOs of the DUT while constantly monitoring their state. Any change in signal level is recorded and can be acted upon. This allows to test embedded software directly on the target hardware, without having to connect all external peripherals that would be present during normal operation. HiL controllers usually run mathematical simulation models to emulate the sensors and actuators required by the specific application of the DUT. They can moreover be used to drive signal buses, such as CAN, or solely as automated logic analyzers. We use this technique throughout all our tests and benchmarks.

7.1.2 Primitive Hardware in the Loop Integration Product

The Primitive Hardware in the Loop Integration Product (PHiLIP) [58] is a low-cost solution for HiL testing. This open-source firmware[1] is specifically designed to test hardware abstraction layers and peripheral APIs of embedded applications and OSs. It runs on both the nucleo-f103rb and the bluepill board. PHiLIP can be used either directly via a serial connection or via a convenient python shell interface. Both interfaces support setting and clearing arbitrary GPIOs and offer platform-independent mappings for specific functions, such as DUT reset or universal asynchronous receiver-transmitter (UART) communication. This allows to develop abstract test suites that can be run on a wide range of target devices. Moreover, a designated input capture pin for precise timing measurements is available at all times. Recorded traces are stored in an internal buffer and can be retrieved via the shell interface. Our test setup uses PHiLIP to conduct all measurements in a hardware-independent fashion.

7.1.3 Robot Framework

The Robot Framework (RF) [49] is an open-source test automation framework that is written in Python and released under the terms of the Apache License 2.0. It initially was developed by Nokia Networks to test robotic process automation systems. Test cases are specified as human-readable keywords in a tabular format. They can be executed both individually or as a part of larger test suites. Open and extensible software interfaces foster the integration with other tools, such as CI systems, wherefore a rich ecosystem exists around the framework. Test results are output in both a rich HTML and a machine-readable XML format. The first provides quick insight, while the latter facilitates data post-processing and allows further automated analyses. The set of built-in functions is readily extended with additional libraries that can be written in Python, Java, or many other programming languages. We use the Robot Framework to specify and execute all our test cases as well as our performance benchmarks in an automated fashion.

7.2 Methodology

We evaluate uTimer on a total of 14 distinct boards, ranging from low-end to high-end devices. Selected microcontrollers cover a comprehensive range of embedded hardware and are commonly deployed within various IoT contexts. Table 7.1 lists all assessed boards including their MCU, vendor, word size and CPU architecture. Table 7.2 extends this list by a quantitative overview of available hardware timers and compare channels. Boards were selected to represent a diverse spectrum of timer peripherals. They offer general-purpose timers between 8 and 64 bit, basic

[1] PHiLIP GitHub repository: `https://github.com/riot-appstore/PHiLIP` (Accessed: 18.01.2022)

Board	Microcontroller	Vendor	Word	Architecture	CI
arduino-mega2560	ATmega2560	Microchip Technology	8 bit	AVR	✓
esp32-wroom-32	ESP32	Espressif Systems	32 bit	LX6	✓
esp8266-esp-12x	ESP8266	Espressif Systems	32 bit	LX106	✓
nucleo-f103rb	STM32F103RB	ST Microelectronics	32 bit	Cortex-M3	✓
nucleo-f767zi	STM32F767ZI	ST Microelectronics	32 bit	Cortex-M7	✓
nucleo-g474re	STM32G474RE	ST Microelectronics	32 bit	Cortex-M4	✓
nucleo-l152re	STM32L152RE	ST Microelectronics	32 bit	Cortex-M3	✓
slstk3400a	EFM32HG322	Silicon Labs	32 bit	Cortex-M0+	✓
slstk3401a	EFM32PG1B200	Silicon Labs	32 bit	Cortex-M4F	✓
stk3200	EFM32ZG222	Silicon Labs	32 bit	Cortex-M0+	✓
z1	MSP430F2617	Texas Instruments	16 bit	MSP430	✓
nucleo-f070rb	STM32F070RB	ST Microelectronics	32 bit	Cortex-M0	×
nucleo-l476rg	STM32L476RG	ST Microelectronics	32 bit	Cortex-M4	×
slstk3402a	EFM32PG12B500	Silicon Labs	32 bit	Cortex-M4F	×

Table 7.1: Overview of evaluated microcontroller boards. Their availability within our continuous integration system is indicated by the *CI* column.

Board	CPU Clock	Product Range	Timers (total)	Channels (total)	8-bit Timers	16-bit Timers	24-bit Timers	32-bit Timers	48-bit Timers	64-bit Timers
z1	8 MHz	-	2	10		✓				
arduino-mega2560	16 MHz	-	6	16	✓	✓				
slstk3400a	24 MHz	-	5	11		✓	✓			
stk3200	24 MHz	-	4	8		✓	✓			
nucleo-f070rb	48 MHz	-	10	14		✓	✓	✓		
nucleo-l152re	32 MHz	○	11	22		✓	✓	✓		
slstk3401a	40 MHz	○	6	11		✓	✓	✓		
slstk3402a	40 MHz	○	8	18		✓	✓	✓		
nucleo-f103rb	72 MHz	○	6	17		✓	✓	✓		
esp32-wroom-32	80 MHz	○	5	5					✓	✓
esp8266-esp-12x	80 MHz	○	2	2				✓		
nucleo-l476rg	80 MHz	+	15	32		✓	✓	✓		
nucleo-g474re	170 MHz	+	14	47		✓	✓	✓		
nucleo-f767zi	216 MHz	+	17	35		✓	✓	✓		

Product range is rated as: low-end (-), mid-range (○), or high-end (+)

Table 7.2: Overview of hardware timers as available on the evaluated microcontroller boards. Devices appear sorted by *Product Range* and *CPU Clock*.

as well as feature-rich modules, and some provide even advanced ultra low-power timers. Moreover, three distinct interrupt handling techniques, various advanced counting modes, and timer chaining capabilities are included. Our selection covers 10 different CPU architectures[2] and all predominant word sizes. Devices are produced by five manufacturers and incorporate MCUs that are frequently used for resource-constrained applications in the Internet of Things.

We perform the following steps for each board:

1.) Executing timer test suites against all APIs on the target device. This happens prior to any evaluative measurement, ensuring that assessed implementations meet their specifications.
2.) Conducting timer performance benchmarks and comparing the results of uTimer against the existing `periph_timer` API.
3.) Assessing further quantitative aspects that are not covered by our benchmarks. These include memory consumption and the availability of timer peripherals to user applications and high-level OS modules.
4.) Examining qualitative aspects, such as user-friendliness, software quality, and compatibility.
5.) Investigating encountered problems and issues, if any.

Results and findings from the evaluated boards are compiled into individual summaries for all assessed aspects. These are subsequently judged as part of a comprehensive discussion.

7.3 Test Setup

The setup we use to conduct all software tests and performance benchmarks is described in detail in this section. At first, a comprehensive overview of its architecture is given. We highlight all applied concepts and techniques, as well as individual components and their interaction. System accuracy is subsequently quantified and hardware limits of our setup are discussed. We carry out individual test series to determine the stability of all our measurements. This ensures reliable and reproducible results. Last, we outline the integration of our evaluation setup into the current CI system and hardware testbed of RIOT.

7.3.1 Architecture

Automation is a necessity for obtaining comparable measurement results from a large number of target devices. We therefore combine generic test firmware and benchmark routines with Robot Framework test suites to automate execution. Since we are assessing the timer subsystem of the target device, we cannot rely on its own system clock for measuring execution durations.

[2]The Cortex-M0+ architecture supports the full instruction set of the Cortex-M0 architecture while having a lower energy consumption and only a two-stage CPU pipeline.

Incorporating HiL testing provides us with an external reference device that possesses a qualified clock source, which is independent of the DUT. PHiLIP is used as a HiL controller that is able to conduct all our measurements in a hardware-independent fashion. It integrates directly with RF through a Python library that allows to control the DUT and exposes recorded GPIO event traces to test suites. The architecture of our setup and the interaction of its components is illustrated in Figure 7.1.

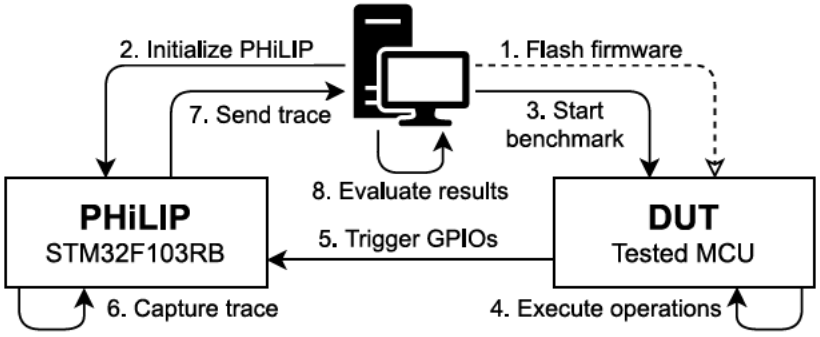


Figure 7.1: Architecture of our benchmark setup

Process flow is controlled by a computer that runs the Robot Framework software. Prior to suite execution, the corresponding test firmware is flashed onto the DUT once (1). It implements all benchmark routines and is controlled through a simple shell interface, which is exposed to the host via UART. The corresponding RF test suite subsequently starts. It initializes the PHiLIP device (2), issues control commands via the DUT shell (3) and uses PHiLIP to record GPIO event traces (6–7). DUTs signal begin and end of time periods via their GPIOs during execution of the requested operations (4–5). Measurements are evaluated by processing the captured event traces within the RF test suite (8). All results are exported as machine-readable XML files in the xUnit format[3]. These are further processed into CSV files that combine the results from all assessed platforms.

Prior to any measurement, the DUT firmware version and board timing parameters, such as oscillator frequencies, are verified. In case of mismatch, test suite execution is aborted to obviate erroneous results. Each test case starts by resetting the DUT, flushing the trace buffer, and configuring DUT interrupt requests (IRQs). That follows a short pause to surpass PHiLIPs minimum hold-off time. During test teardown, GPIOs are cleared and IRQ state is restored. Every benchmark is repeated 500 times to ensure accurate results. Certain microbenchmarks additionally repeat the executed function calls 10 times per benchmark pass to safely exceed the required hold-off time. We quantify the minimum hold-off time of PHiLIP and discuss appropriate sample sizes for all our benchmarks in detail within the next section.

[3]The xUnit.net v2 XML Format: `https://xunit.net/docs/format-xml-v2` (Accessed: 20.01.2022)

7.3.2 Accuracy and Hardware Limits

Start and stop of measured time durations is indicated by consecutive rising and falling edges on a GPIO pin. The accuracy of this measurement therefore depends on both the signaling overhead of the DUT and the input capture performance of PHiLIP. The first is accounted for during trace post-processing, as quantified by our GPIO overhead measurements that are described in Section 7.4.1. The latter depends on the selected trigger mode. We use the interrupt-based dual-edge trigger mode of PHiLIP for all our tests. This mode is rated to require at least 1 µs hold-off time between two consecutive edges and faces 200 ns timestamp jitter [58, p. 14]. However, we found its accuracy to improve when using a Nucleo-F103RB instead of the smaller Blue Pill board. We hereby were able to obtain accurate and stable results with as little as 600 ns hold-off time and only 45 ns jitter. When undercutting this limit, however, samples start to get lost, what ultimately results in erroneous measurements. A hold-off time of at least 1 µs is therefore enforced throughout all our test and benchmark routines to provide a reasonable safety margin and to preserve board compatibility.

The temporal resolution of captured traces is limited by the CPU operation frequency of the MCU that runs PHiLIP. As both supported boards operate at 72 MHz, 14 ns quantization steps can be observed within our measurements.

Each test or benchmark pass may generate multiple GPIO events and can be repeated multiple times. PHiLIP records each event occurrence within a circular trace buffer that can hold up to 128 events at once. All our test and benchmark suites therefore fetch captured event traces not later than after every 50 consecutive measurement periods. Traces from multiple runs may be combined to reduce their total execution time. This is accounted for within the corresponding RF suites, whenever applied.

7.3.3 Measurement Stability

The wall-clock time that an operation takes to fully execute can vary due to multiple factors. These include task scheduling, concurrency control, and interrupt servicing. Repeating the set of measured operations is therefore required to achieve stable benchmark results. Choosing a sample size that is too small yields inaccurate and unreliable results. Simply selecting a superfluously large sample size may reliably produce accurate results, but skyrockets benchmark execution time. We ran all benchmark suites using sample sizes ranging from 50 to 1000 with a step size of 50 to determine an appropriate value for our measurements. Result stability over full suite executions was verified by repeating the described procedure within batches of five consecutive benchmark suite runs on three different days.

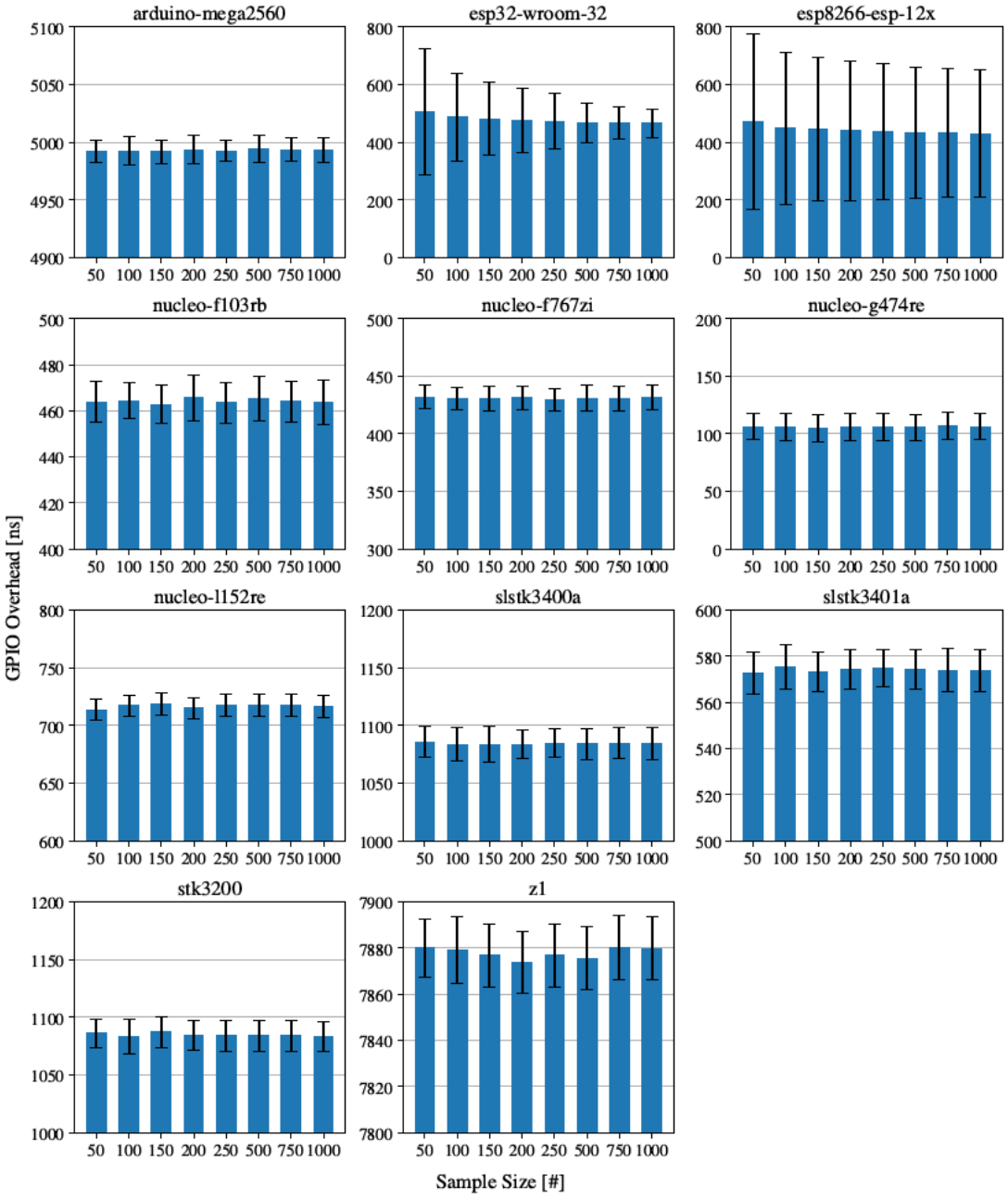


Figure 7.2: Comparison of GPIO overhead on all boards using different sample sizes. Error bars indicate the standard deviation of each sample set. Note that the vertical axis scales differ to improve the readability of error bars.

Figure 7.2 shows the results of GPIO overhead measurements (see Section 7.4.1) on all boards for a subset of the assessed sample sizes. Results are stable on all boards at a sample size of at least 250. The basic timer operations benchmark (see Section 7.4.2) becomes stable at much lower sample sizes, compared to other benchmarks. This is due to the fact, that the measurement routine itself already executes each operation 10 times per benchmark pass. It produces stable results after no more than 50 repetitions across all boards. Benchmarks that assess various types of timeouts (see Section 7.4.3) produce stable results on 82 % of the boards with 100 samples. They become stable on all boards when at least 200 repetitions are conducted.

Our analysis identified a sample size of 250 to be the absolute common minimum across all benchmarks and devices. Therefore, we repeat all our measurements 500 times to provide a sound safety margin while keeping benchmark execution times as short as possible.

7.3.4 Continuous Integration and Benchmark Automation

Comprehensive software tests and benchmarks are especially important when dealing with a diverse and ever-changing hardware landscape. RIOT therefore actively fosters automated cross-platform testing by combining Hardware in the Loop (HiL) testbeds with Continuous Integration (CI). Its HiL system is operated by the RIOT developer community, currently features 30 different boards, and is being further extended right now. All devices are arranged within racks, such as the one depicted in Figure 7.3. DUTs are connected to STM32 Blue Pill boards that fulfill the role of the HiL controllers by running the PHiLIP firmware. Each device pair possesses an individual Raspberry Pi control node that executes all tests using the Robot Framework. A Jenkins automation server[4] executes unit tests and performance benchmarks for every nightly build and pull request. In addition, any existing as well as custom test suites can manually be run against arbitrary RIOT firmware versions.

We leverage this system to execute the developed test suites and performance benchmarks. Our evaluation setup therefore seamlessly integrates with this solution and can easily be extended to include additional microcontroller boards in the future. OS and application developers are provided with ready-to-use unit tests and benchmarks, hereby encouraging and guiding the development of performant API implementations.

7.4 Benchmarks

Within this section, we develop benchmarks to quantify the performance of both uTimer, internally referred to as `periph_utimer`, and the current `periph_timer` API. These assess basic

[4] Jenkins Project Website: `https://www.jenkins.io/` (Accessed: 21.01.2022)

Figure 7.3: A single rack from the HiL testbed of the RIOT community

timer operations (see Section 7.4.2), different types of timeouts (see Section 7.4.3), interrupt handling (see Section 7.4.4), and the platform-independent abstraction overhead that is inherent to the API itself (see Section 7.4.5). Before that, we quantify the board-specific GPIO overhead (see Section 7.4.1), allowing its compensation within subsequent measurements.

As we conduct all measurements using the HiL testbed of RIOT, performance evaluation is limited to the available boards. Table 7.1 lists the 11 of our total of 14 targeted boards that are integrated into the CI system. Including the remaining three boards would be possible by manually executing the benchmarks on these. We nonetheless decided against this approach, since using different test architectures can potentially yield incomparable results.

7.4.1 GPIO Overhead

The time that is required to set and clear a GPIO pin varies between MCUs due to differences in CPU architecture, operation frequencies, and low-level peripheral code. As we signal start and stop of measurements by these events, knowing the board-specific GPIO overhead O_{GPIO} is essential for achieving accurate results. We measure it according to pseudocode Listing 7.4a.

Active waiting, also referred to as *spinning*, is used to represent a set of executed operations. The `spin()` function is inlined to avoid any function call overhead. Subtracting its calibrated execution time from the recorded trace yields the GPIO overhead O_{GPIO}. It is calculated individually for each board within our measurement setup according to Equation (7.1). We

```
Setup
REPEAT 500 times:
  gpio_set(GPIO_IC);
  spin(SPIN_DURATION);
  gpio_clear(GPIO_IC);
  spin(PHILIP_HOLDOFF_TIME);
Teardown
```

(a) Benchmark pseudocode

$$\Delta t = t_{\texttt{gpio\_clear()}} - t_{\texttt{gpio\_set()}} \tag{7.1a}$$

$$O_{GPIO} = \frac{1}{N} \sum_{i=1}^{N} [\Delta t_i - t_{spin}] \tag{7.1b}$$

(b) Evaluation rules

Figure 7.4: Definition of the GPIO overhead benchmark

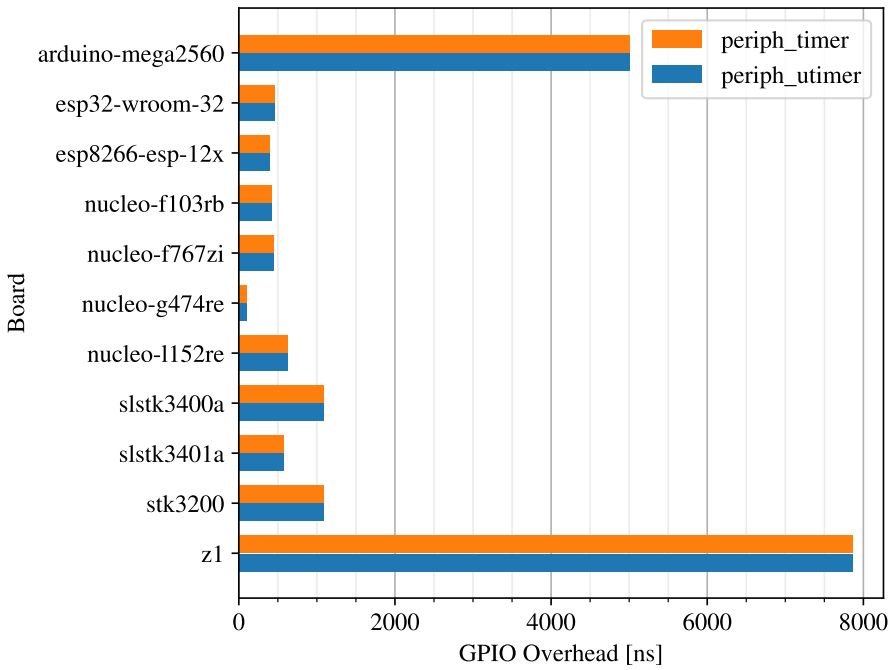


Figure 7.5: GPIO overhead for all boards within our CI setup

tested spin durations up to 1 s and found results to be stable for periods between 1 µs and 1 ms. Longer spins were increasingly affected by clock drift and CPU pipelining artifacts, which is why we decided to use a spin duration of 1 µs for all our measurements.

Figure 7.5 illustrates the measured GPIO overhead on all boards. On nine of the 11 assessed boards, O_{GPIO} is in the nanosecond range. On the remaining two boards, GPIO overheads of 5.01 µs ± 21 ns and 7.86 µs ± 25 ns were observed. This is due to the significantly lower CPU frequencies of 16 MHz on the `arduino-mega2560` and 8 MHz on the `z1`. Our measurements also confirmed the expected independence of O_{GPIO} from the used timer API.

7.4.2 Basic Timer Operations

Accessing the counter value and configuring compare channels are the most essential operations a hardware timer can perform. We measure their execution time according to pseudocode Listing 7.6a. Because the assessed operations can complete very quickly, each function is invoked ten times per measurement period to ensure that the required hold-off time is reliably exceeded on all devices. Function calls are textually repeated via preprocessor macros to avoid any form of loop overhead. Repetitions are factored out during evaluation, according to Equation (7.2). Recorded execution durations t_{op} are converted from wall-clock time to equivalent CPU cycles. This allows comparison across MCUs despite their different clock frequencies. uAPI and hAPI driver calls are assessed separately. The latter is executed solely as the desired driver operation, e.g., `driver->read()` instead of the equivalent uAPI function `utimer_read()`.

```
Setup
REPEAT 500 times:
  START Measurement
    timer_operation();
    timer_operation();
    // ... repeated 10 times
    timer_operation();
  STOP Measurement
  spin(PHILIP_HOLDOFF_TIME);
Teardown
```

(a) Benchmark pseudocode

$$\Delta t = t_{\text{STOP}} - t_{\text{START}} \tag{7.2a}$$

$$t_{op} = \frac{1}{N} \sum_{i=1}^{N} \left[\frac{1}{10} \left(\Delta t_i - O_{GPIO} \right) \right] \tag{7.2b}$$

(b) Evaluation rules

Figure 7.6: Definition of the timer base operation benchmark

Figure 7.7 provides an overview of the number of CPU cycles that are consumed by read, write, set, and clear operations. Each board is represented by a radar chart in which the performance of all base operations is plotted for the different APIs. Counter value access is shown on the horizontal axis and timer channel usage is shown on the vertical axis. Spikes and large surface areas indicate long execution times. The standard deviation is less than 0.1 CPU cycles for all measurements and is therefore not explicitly stated.

Observed execution times vary between timer operations and target hardware. Eight boards complete all benchmarked operations within approximately 100 CPU cycles or less. The other three MCUs take no more than 200 cycles to complete all operations except `set()`. On two of the boards, arming a compare channel takes up to 275 cycles, where uTimer outperforms `periph_timer` on the ESP32 but is inferior on the arduino-mega2560. Read and write operations are on average three times faster than setting and clearing timer channels, while arming a single channel takes on average about 50 % longer than clearing it.

Each assessed operation is individually discussed in the remainder of this section. Before that, however, it can be noted that all observed performance differences, both positive and negative,

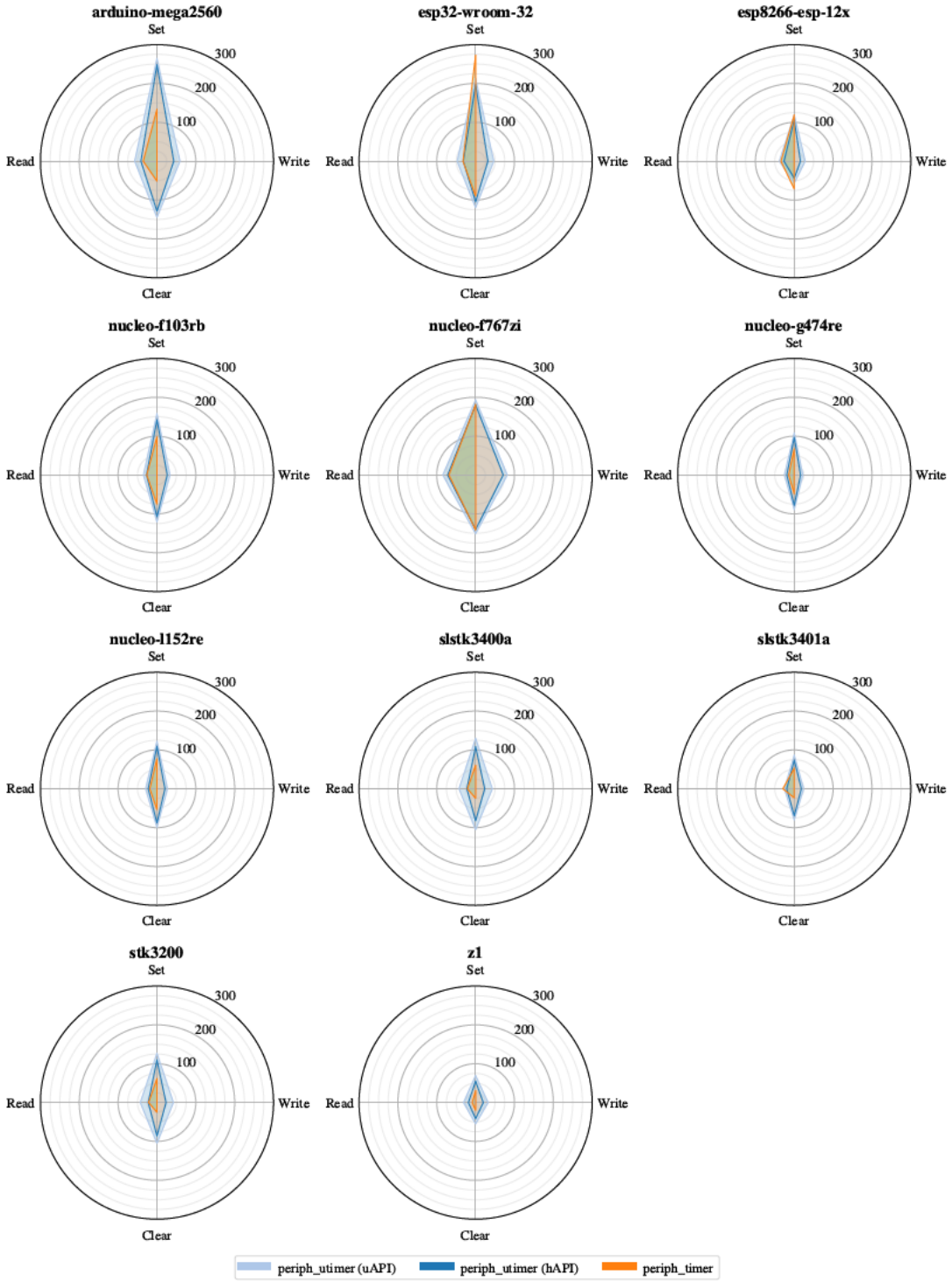


Figure 7.7: CPU cycles consumed by read, write, set, and clear operations. Write operations are not supported by `periph_timer` and are therefore not shown in the plot.

are negligibly small for most use cases. As CPU frequencies are typically at least ten times higher than timer base frequencies, multiple instructions are executed within a single timer tick period. Minor increases in computational overhead therefore do not noticeably affect timer performance. With 1 MHz timers on the ESP32, for example, the processor executes 80 instructions per timer tick, hereby virtually eliminating the difference in CPU cycles regarding timer behavior.

Reading the Counter Value

Direct reads of the raw counter value are usually performed only when the user application needs to obtain a current system timestamp or during relative timer arming. Actively polling the counter register at a high frequency becomes superfluous when using compare channels to generate timeouts. Our extensive hardware analysis confirmed their general availability on all assessed microcontrollers, hence active polling is never required (see Section 4.3.3).

All boards except the z1 show an increase in execution time by less than a factor of two, when compared to `periph_timer`. The z1, however, faces an increase by a factor of 3.57 while uTimer slightly outperforms the existing API on the slstk3410a. On average, uTimer increases the CPU cycles that are consumed by timer read operations by a factor of 1.69.

Writing the Counter Value

Manually writing the raw counter value can be required when operating a timer in one-shot mode or during initialization. It moreover allows to align multiple counters and enables high-level timer subsystems to perform certain optimizations. As this feature is currently not supported by `periph_timer` it could only be assessed for our proposed API. With uTimer, writing the counter register is as fast as reading it. This applies to all platforms and is explained by the fact that both read and write operations usually only require a single register access to complete.

Timer Reloading

The auto-reload feature allows handling counter reloading in a CPU-independent fashion. It can remove any overhead that is caused by timer write operations upon counter overflows and thereby relieves the CPU from overflow maintenance. At least one type of auto-reloading is supported by all timer peripherals (see Section 4.3.3).

Writing the counter register requires between 16 and 71 CPU cycles among the assessed boards. Eliminating manual counter register writes via the auto-reload features therefore allows saving at least 129 ns and up to 4 µs per counter overflow[5]. It moreover removes any overhead that

[5]CPU cycles are converted to wall-clock time t based on the CPU frequency as $t = cycles \cdot f_{CPU}$

is caused by interrupt servicing, as quantified in Section 7.4.4, allowing to additionally save between 870 ns and 29 µs. This benefits high-level timer subsystems in particular and improves the precision of long-running or periodic timeouts.

Compare Channel Arming

Initializing a timer channel is required once per timeout. While this means that the `set()` function is invoked prior to every one-shot timeout, consecutive periods of periodic timeouts do not require its repeated execution. Compare channel arming performance is therefore primarily relevant for one-shot timeouts.

Observed uTimer overhead is below a factor of two on nine of the boards and caps off at a worst-case increase by a factor of 2.28 on the z1. On the ESP32, uTimer outperformed `periph_timer` by more than 20 %. As the number of required CPU cycles is stable ($\sigma \leq 0.1$), the static computational overhead that is inherent to the `set()` function can easily be compensated for during timer arming. Automated execution of the developed benchmarks readily provides developers with all the required parameters and allows their convenient verification.

Compare Channel Disarming

With one-shot timeouts, compare channels are automatically disarmed upon compare match. Periodic timeouts require manual channel disarming instead. Whenever a timer channel is cleared prior to timeout expiry, all actions to be executed upon expiry become null. We therefore consider this a non time-critical operation. The maximum number of processor cycles that are required for clearing a channel is 142 cycles with `periph_timer` and 151 cycles with uTimer.

7.4.3 Timeouts

The latency of a timeout L_τ is the difference between the expected and the actually observed end of a time period. It can be used as an indicator for the abstraction and maintenance overheads of timeout handling. Negative values for L_τ may occur due to oscillator inaccuracies (e.g., constant frequency offsets, jitter, and drift), faulty driver code[6], or a combination of both. Its standard deviation is referred to as timeout jitter J_τ. The timeout error E_τ puts timeout latency in relation to the expected timeout duration τ. Keeping it within application appropriate limits is crucial. Our benchmarks assess the performance of one-shot and periodic timeouts. Both types are discussed individually in the remainder of this section.

[6] A significant proportion of the timeout latency is constant. Some peripheral drivers therefore try to compensate it by subtracting a fixed value from all timeout durations during channel arming. If this value is larger than the actual overhead, timeout durations will be shorter than expected, resulting in negative timeout latencies.

One-shot Timeouts

One-shot timeouts are used to model nonrecurring and independent time periods. They are automatically disarmed at their first compare match event, hence trigger only once. We measure the performance of one-shot timeouts according to pseudocode Listing 7.8a. First, a timer is initialized and stopped. It is then armed to a counter value that corresponds to the desired timeout duration τ at the configured counting frequency f. Relative arming is required due to the missing support for writing the counter register via `periph_timer`. Using absolute arming instead would lead to unpredictably inaccurate results, since no assertion regarding the counter value after peripheral initialization can be made with the existing API. Elapsed time Δt between the subsequent timer start $t_{\texttt{START}}$ and the awaited callback execution $t_{\texttt{STOP}}$ then is measured (7.3a). Results for all assessed parameters of a complete benchmark pass are calculated according to Equation (7.3b-d).

```
Setup
REPEAT 500 times:
  timer_init(frequency, callback);
  timer_stop();
  cnt = timer_read() + timeout;
  timer_set(cnt);

  timer_start();
  START Measurement
    WAIT for callback execution
  STOP Measurement
  spin(PHILIP_HOLDOFF_TIME);
Teardown
```

(a) Benchmark pseudocode

$$\Delta t = t_{\texttt{STOP}} - t_{\texttt{START}} \tag{7.3a}$$

$$L_\tau = \frac{1}{N} \sum_{i=1}^{N} [\Delta t_i - \tau - O_{GPIO}] \tag{7.3b}$$

$$J_\tau = \sigma(L_\tau) \tag{7.3c}$$

$$= \sqrt{\sigma(\Delta t)^2 + \sigma(O_{GPIO})^2}$$

$$E_\tau = \text{abs}\left(\frac{L_\tau}{\tau}\right) \tag{7.3d}$$

(b) Evaluation rules

Figure 7.8: Definition of the one-shot timeout latency benchmark

Evaluated timer frequencies vary between 10 kHz and 10 MHz while timeout durations vary between 10 µs and 1 s. Results show that compared to `periph_timer`, uTimer suffers an increased timeout latency on 90 % of all boards. This behavior reveals the inevitable drawbacks of abstraction and was observed throughout all tested timeout durations alike. Changing the counting frequency had no effect on benchmark results. The increased latency is caused by a combination of resolving the timer type dependent low-level ISR upon IRQ occurrence and the now supported timer overflow handling[7]. Whether this overhead is tolerable depends on both the timeout duration and the application requirements. The respective timeout error therefore must be considered when judging performance impacts. Figure 7.9 compares the performance of both APIs for common 1 ms one-shot timeouts. For all but one boards, the absolute timeout error increased by between 0.04 % on the nucleo-g474re (best) and 0.34 % on the esp8266-esp-12x board (worst). On the arduino-mega2560, `periph_utimer` outperformed `periph_timer` by

[7] This effect is not present on platforms with independent interrupts for both compare match and overflow events.

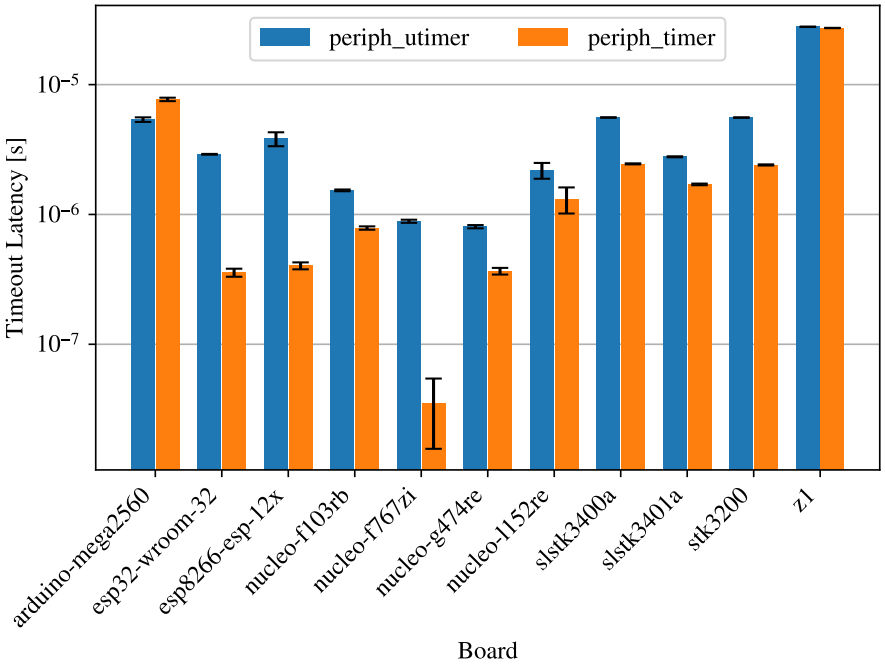


Figure 7.9: Latencies of 1 ms timeouts at the platform-default timer frequency

0.23 %. Jitter remained stable on all boards except the esp8266-esp-12x where it increased from 160 ns to 495 ns.

Timeout latency is of no concern for long-running timeouts ($\tau \geq 1\,\text{s}$), as other factors, such as oscillator accuracy, become increasingly dominant. Negative performance impacts, however, are problematic with very short timeouts ($\tau \leq 10\,\mu\text{s}$), as even just a slight increase in latency contributes significantly to the total timeout error. In such cases, unnecessary indirection should be avoided at any cost. Directly using the hAPI or active waiting, i.e., spinning, is therefore recommended. Most applications, however, do not require such very short timeouts, as confirmed by our timer use case analysis in Section 4.4. A slightly increased latency is therefore negligible when put into perspective to the timeout durations that are common with typical IoT use cases.

Our benchmarks showed that the timeout latency is generally independent of the selected counting frequency. However, a significant increase was observed for certain timeout and frequency combinations when using `periph_timer`. On the nucleo-f070rb, for example, generating a 1 s timeout using a timer that runs at 10 kHz base frequency resulted in a timeout latency of 11.5 µs. This corresponds to a timeout error of less than 0.01 %. Requesting the same 1 s timeout from a peripheral that is configured to 1 MHz instead, increased timeout latency to 983.0 ms. This constitutes an abnormally high timeout error of 98.30 %. Since system behavior for invalid parameters is not explicitly specified by the API, implementations usually neither verify the requested frequency nor timeout duration. An established convention is to select the closest

achievable timer frequency or compare channel value. This leads to unpredictable inaccuracies if parameters are not chosen carefully by the developer, as required for each MCU and its current clock configuration individually. Both erroneous cases result in the observed timeout latency increase. uTimer is not affected by this problem, because it requires implementations to signal an error if a requested frequency is invalid or a desired timeout is out of range.

Periodic Timeouts

Periodic timeouts trigger compare match events at a fixed time interval until manually disarmed by the application. Repeating a periodic timeout only once, effectively results in a one-shot timeout. Any subsequent cycle, however, can potentially outperform one-shot timeouts in terms of timeout latency. This is due to the fact that the compare channel only needs to be armed once, and therefore only one initial API call is performed. All necessary timer maintenance tasks can moreover be executed solely within hardware, as supported by 85 % of all timer types we analyzed within our large-scale hardware analysis (see Section 4.3.3). This reduces timeout latency to the execution time of the low-level ISR that needs to be executed prior to callback invocation[8]. Drivers can also implement periodic timeouts exclusively in software if no adequate hardware support is available. However, this has a negative impact on performance compared to in-hardware solutions.

The `periph_timer` API requires the additional module `periph_timer_periodic` to be loaded to generate periodic timeouts. It is available for five of the 11 assessed boards. The proposed uTimer API instead provides native support for periodic timeouts. We have currently implemented it on eight boards, as this is sufficient for our comparative evaluation. These are all five boards that offer `periph_timer_periodic` support, as well as three additional boards to demonstrate the ease of implementing this feature with uTimer.

With periodic timeouts, we likewise assess timeout latency, jitter, and error. In contrast to one-shot timeouts, periodic timeouts can be repeated multiple times prior to disarming. Our benchmarks account for this by decrementing a volatile loop counter during each compare match callback execution, as shown in pseudocode Listing 7.10a. Elapsed time is measured in the form of a combined time period Δt (7.4a) and timeout latency L_τ is calculated as the mean value of all timeout cycles, i.e., the number of times a timeout is repeated prior to disarming (7.4b). Timeout jitter and timeout error are calculated the same way as for one-shot timeouts (7.4c–d).

Prior to all measurements, we examined if clearing the `GPIO_IC` pin, which signals the end of a time measurement, within the ISR context vs clearing it within the control loop affects our results. We found no significant impact on measured time durations within both our periodic timeouts and parallel callbacks (see Section 7.4.4) benchmarks.

[8] Timeout latency can further be affected by other high-priority system tasks and pending IRQs.

```
Setup
REPEAT 500 times:
  timer_init(frequency, callback);
  timer_stop();
  timer_set_periodic(cnt, RST_MATCH);
  cycles_left = cycles;

  timer_start();
  START Measurement
    WHILE cycles_left > 0:
      WAIT for callback execution
      cycles_left--;
  STOP Measurement
  spin(PHILIP_HOLDOFF_TIME);
Teardown
```

(a) Benchmark pseudocode

$$\Delta t = \frac{t_{\text{STOP}} - t_{\text{START}}}{cycles} \tag{7.4a}$$

$$L_\tau = \frac{1}{N} \sum_{i=1}^{N} [\Delta t_i - \tau - O_{GPIO}] \tag{7.4b}$$

$$J_\tau = \sigma(L_\tau) \tag{7.4c}$$

$$= \sqrt{\sigma(\Delta t)^2 + \sigma(O_{GPIO})^2}$$

$$E_\tau = \text{abs}\left(\frac{L_\tau}{\tau}\right) \tag{7.4d}$$

(b) Evaluation rules

Figure 7.10: Definition of the periodic timeout latency benchmark

We assessed periodic timeouts between 1 and 1000 repetitions. On all boards except two, any differences in timeout latency vanished after 100 timeout cycles at the latest. This behavior was, for example, observed on the nucleo-f103rb board and is illustrated in the left chart of Figure 7.11. The two exceptions to this are the arduino-mega2560, where uTimer suffers a constant increase in timeout error by 0.39 %, and the nucleo-g474re, where uTimer outperforms `periph_timer` after no more than 10 timeout cycles. This second case is illustrated in the right chart of Figure 7.11.

7.4.4 Interrupt Handling

Interrupt handling is an important performance indicator for low-level timer drivers. It affects timeout accuracy and timer maintenance to a significant extent. In this section, we discuss various aspects of both compare match and overflow interrupts. We develop a benchmark suite that assesses timer performance under stress and compare the scaling behavior of uTimer and `periph_timer` within such situations.

Compare Match Interrupts

Compare match (CMP) interrupts are issued whenever an armed timer channel expired. A low-level interrupt service routine (ISR) is executed first upon event occurrence. It performs all necessary maintenance tasks before the user-defined callback function is invoked. ISRs are executed within a special interrupt context and should always be kept as short as possible. Their efficient design becomes particularly important whenever multiple interrupt requests (IRQs) are pending simultaneously. As only a single function can be executed by the CPU at any point in

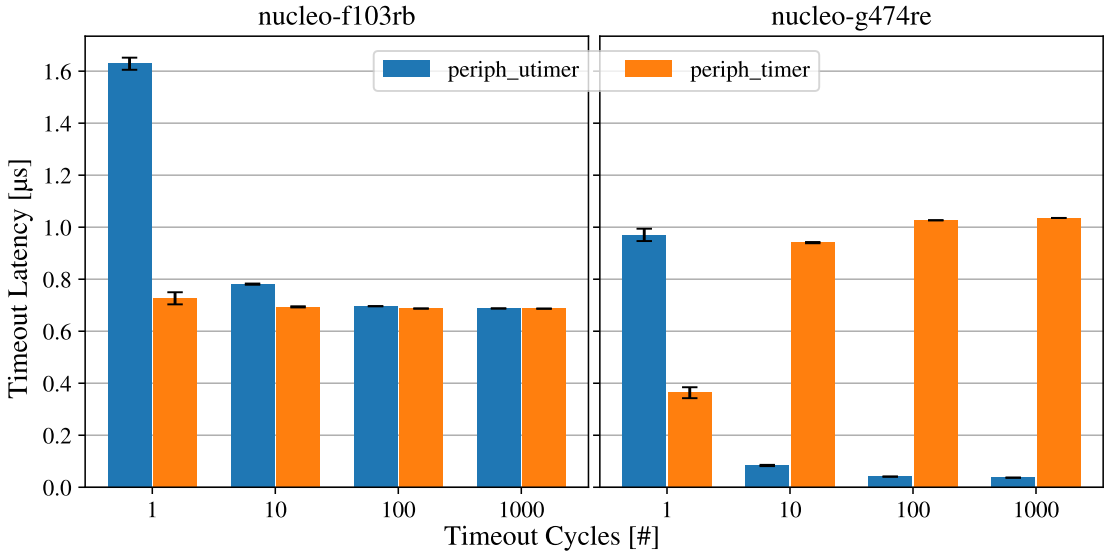


Figure 7.11: Latencies of periodic 1 ms timeouts after different numbers of timeout cycles

time, sequential processing of system events is inevitable[9]. Hence, the longer the execution of a single ISR takes and the more IRQs are pending, the poorer the reactivity of the entire system becomes. The latency of timeouts in particular therefore is directly affected by the number of currently elapsed but still unhandled compare match events.

Our parallel timeouts benchmark provides a quantitative indicator for the timeout performance under stress. It measures the callback execution delay D_{cb} for different numbers of simultaneously expiring timeouts. This is the time it takes between the raise of compare match IRQs and the invocation of all associated user-callback functions. The callback execution delay hereby indicates the best-case reactivity of the firmware application under the respective stress conditions. Our benchmark measures it according to pseudocode Listing 7.12a. At first, a timer is initialized and the desired number of compare channels is armed to the same absolute counter value. To reduce the influence of other system events, all maskable INTs that are not required during suite execution are disabled. The number of armed channels is stored in a volatile counter variable. After starting the timer, the counter variable is decremented every time a user callback is executed. Measurement stop is signalled right after the last callback was invoked (7.5a). The callback execution delay is then determined by subtracting the timeout duration τ and the GPIO overhead O_{GPIO} (7.5b) from the measured time period Δt. Its standard deviation is referred to as callback execution jitter J_{cb} and calculated according to Equation (7.5c).

[9]Some interrupt controllers prioritize certain interrupt vectors. Execution of a low-priority ISR can therefore get interrupted by a high-priority ISR. This potentially induces priority inversion problems within complex systems [55]. Timer interrupts, however, are usually within the same priority class and therefore always processed one after the other.

These steps are repeated for timeouts from one up to the maximum number of available compare channels on every board. Since uTimer provides this information via its static property interface, as described in Section 5.2, the peripheral-specific value range can automatically be determined by the benchmark firmware. Likewise to the periodic timeouts benchmark (see Section 7.4.3), we found no difference in results whether measurement end is signaled directly within the interrupt context or the control loop.

```
Setup
REPEAT 500 times:
  timer_init(frequency, callback);
  timer_stop();
  cnt = timer_read() + timeout;

  channels_left = channels;
  FOR chan = 0 to (channels-1):
    timer_set(chan, cnt);

  timer_start();
  START Measurement
    WHILE channels_left > 0:
      WAIT for callback execution
      channels_left--;
  STOP Measurement
  spin(PHILIP_HOLDOFF_TIME);
Teardown
```

(a) Benchmark pseudocode

$$\Delta t = t_{\text{STOP}} - t_{\text{START}} \tag{7.5a}$$

$$D_{cb} = \frac{1}{N} \sum_{i=1}^{N} [\Delta t_i - \tau - O_{GPIO}] \tag{7.5b}$$

$$J_{cb} = \sigma(D_{cb}) = \sqrt{\sigma(\Delta t)^2 + \sigma(O_{GPIO})^2} \tag{7.5c}$$

(b) Evaluation rules

Figure 7.12: Definition of the parallel callbacks benchmark

We benchmarked callback execution delays with up to four simultaneously expired timeouts. The board-specific limit depends on the number of compare channels that are available on the respective microcontroller. Scaling behavior of all MCUs, including their channel counts, is illustrated in Figure 7.13. The two Espressif devices (18 %) both only have a single compare channel, which is why no more than one compare match IRQ can be pending at any point in time. Scaling behavior can therefore only be assessed on the nine remaining MCUs. Those possess timers with at least three (46 %) or even four channels (36 %).

Both APIs show a virtually identical scaling behavior. Differences in callback execution delay gradient ∇D_{cb} are less than 1 % on all boards. However, on six of the nine MCUs, uTimer performed interrupt servicing slightly faster than `periph_timer` was able to. This speed advantage is due to uTimers support for separate handling of overflow and compare match events. Although its effect is only marginal within our isolated test system (≤ 0.1 %), it can become relevant for real-world systems. Whenever multiple IRQs, including those from other peripherals and components, are pending, even small reductions in ISR execution time improve reaction time in busy situations and benefit overall system performance.

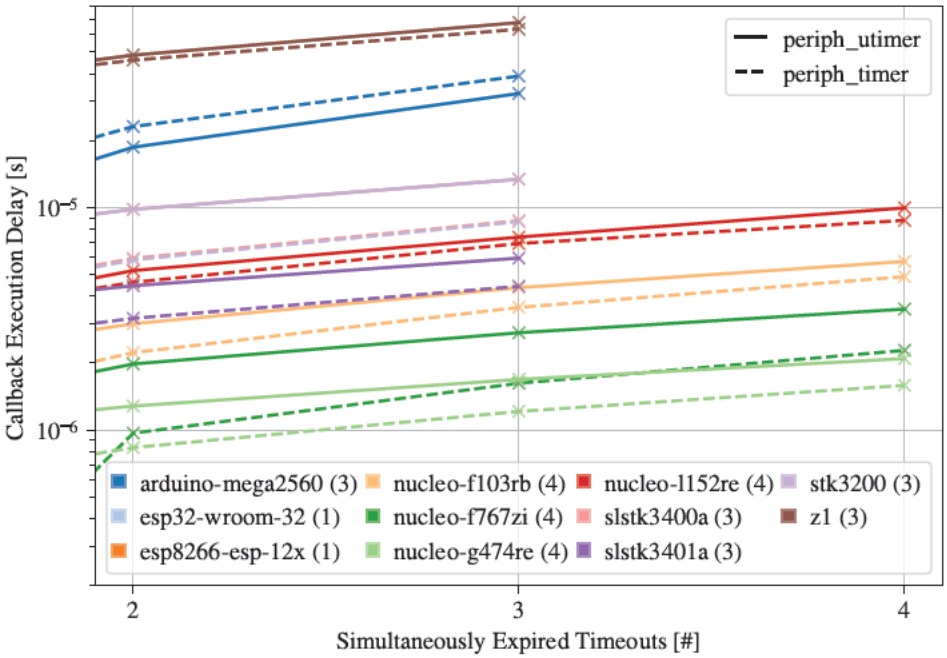


Figure 7.13: Delay of compare match callback execution with simultaneously expiring timeouts

With only a single IRQ pending on the nucleo-f767zi, `periph_timer` outperforms uTimer by 0.9 µs, explaining the kink within Figure 7.13. This is consistent with the results of the one-shot timeouts benchmark (see Section 7.4.3), but this superiority disappears as soon as at least one additional IRQ is pending simultaneously.

Overflow Interrupts

Overflow (OVF) interrupts are issued whenever the internal counter register of a timer reached its maximum value and wraps around[10]. They can also occur prematurely, when an arbitrary auto-reload value is configured. Although overflows can be utilized to generate periodic timeouts, they are predominantly used to trigger timer maintenance tasks. High-level timer subsystems, for example, use them to maintain their software timeouts and to support timeout durations beyond a full overflow period.

uTimer provides access to counter overflow events via a designated timer status flag and a respective callback function can furthermore be invoked upon occurrence. `periph_timer`, on the other hand, does neither exposes overflow status flags nor allows a callback to be attached. Overflow handling performance therefore cannot be compared between the two APIs.

[10] Timers can also underflow depending on the selected counting mode and direction. For the sake of simplicity, we refer to both cases as *timer overflows*.

7.4.5 Abstraction Overhead

The two-layered design of uTimer inevitably causes some computational overhead because of its higher abstraction compared to `periph_timer`. We previously looked at this from a software-design point of view in Section 5.4 and now continue by quantifying the exact overhead that is inherent to the uniform API.

Execution times of all function calls are composed of the device-specific driver code and the generic API logic itself. To isolate the latter, we replaced the driver implementations of counter read and write operations with no operations (NOPs)[11] and measured their execution time. Listing 8 shows the disassembly of a `timer_read()` function after replacing its body. The two remaining instructions move a fixed value to the first general-purpose register of the CPU and return immediately afterwards. Because the `periph_timer` interface does not support write operations, only its read function was assessed. Measurement procedure and result calculation are mostly identical to our basic timer operations benchmark, described in Section 7.4.2. The recorded time duration Δt includes resolving the hAPI driver, invoking the respective function, executing the NOP, and returning a dummy result to the application. We evaluate it for using the full user-facing API as well as using the hardware-facing API directly and compare both results to `periph_timer`.

```
;-- unsigned int timer_read(tim_t dev);
;-- timer.c:257
0x0000297E      nop
0x00002980      movs r0, 0
0x00002982      bx lr
```

Listing 8: Disassembly of the `timer_read()` function (NOP version)

Our measurements confirmed that the uAPI inevitably introduces computational overhead on all platforms due to its additional abstraction layer. The exact amount was found to be platform-dependent and varies between six and 21 CPU cycles. All results are stable with a standard deviation of less than 0.1 cycles. Figure 7.14 illustrates the performance of NOP calls via the different interfaces across all boards. Directly using the hardware-facing API requires a similar amount of CPU cycles[12] compared to `periph_timer` on six boards (55 %). On the remaining five boards (45 %), the hAPI surprisingly outperforms `periph_timer`, hereby even almost compensating for the additional abstraction overhead of the uAPI. This observation is explained by the fact that on those boards, `periph_timer` explicitly distinguishes between different timer types inside each function call. The underlying implementation problem is in detail described in Section 6.3.3. With uTimer, on the other hand, hAPI drivers implicitly specify the timer

[11] No operations (NOPs) are processor instructions that have no effect on the processor state. They take a well-defined number of CPU cycles to execute and are unaffected by compile-time optimizations. If NOP is not implemented as a specific instruction, the assembler treats it as a pseudo-instruction and generates an alternative instruction that does nothing.

[12] A deviation of up to two cycles is considered as similar, due to the influence of CPU pipelining artifacts.

type, whereby this distinction is transparently performed by its abstraction layers. Therefore, the performance differences disappear as soon as different timer types are made available via a common interface.

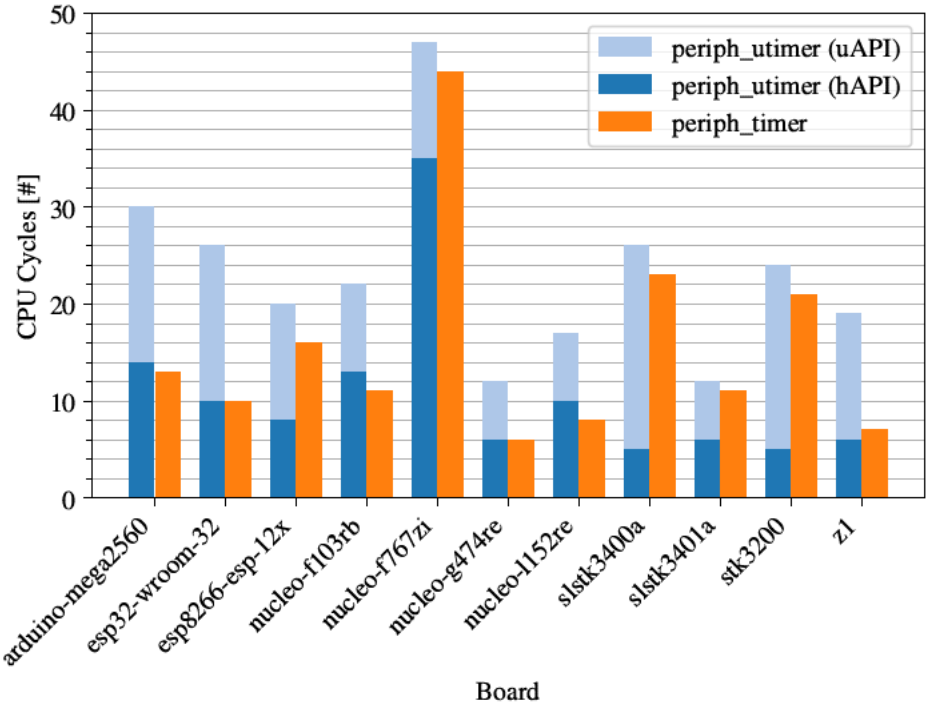


Figure 7.14: CPU cycles consumed by NOP calls via APIs with different degrees of abstraction

In any real-world scenario, execution times of API operations are influenced by a multitude of extrinsic factors. These include task scheduling, pending interrupts, external events, and CPU pipelining. Taking their possible impact into account, the hardware-facing uTimer API and `periph_timer` can be considered to perform alike. Nonetheless, making use of the platform-independent uAPI inevitably consumes additional CPU cycles. The observed overhead ranges from 14 ns (best-case) to 500 ns (worst-case). However, when put into perspective to common timeout durations, as identified in Section 4.4, even the worst-case overhead affects timeout latency by only less than 0.05 %. This minimal increase can furthermore remain unnoticed by applications, depending on the ration between CPU speed and timer frequency, as previously explained in Section 7.4.2. Therefore, all low-level timer interfaces are considered valid choices for every-day applications. The only exception to this are cases where ultra-short timeouts ($\tau \leq 10\,\mu s$) with high precision are required. Then it is recommended to either use the hAPI directly or perform active waiting, i.e., spinning.

7.5 Discussion

This section first summarizes our benchmark results and breaks down factors that contribute to the abstraction overhead of the user-facing API. What follows is an evaluation of both memory footprint and the set of exposed timer peripherals. Qualitative aspects, such as usability and code quality, are assessed and the suitability of uTimer for high-level timer APIs is discussed. We end with an overview of issues that are inherent to our evaluation or were revealed by it.

7.5.1 API Performance

We evaluated the performance of uTimer and compared it to `periph_timer` with respect to various aspects of low-level hardware timers. Prior to any measurement, a sample size of 500 benchmark passes was confirmed to reproducibly yield stable results and our test setup was parameterized individually for each target board.

Basic timer operations, namely `read()`, `write()`, `set()`, and `clear()`, were assessed by measuring their execution times via all available APIs. `periph_timer` outperformed uTimer in most cases, but performance differences were marginal and remained mostly unnoticed by user applications. In our benchmarks, the auto-reload feature moreover reduced overflow maintenance time by at least 1 µs and up to 33 µs on the assessed boards.

The accuracy of timeouts was determined for durations between 10 µs and 1 s as well as counter frequencies between 10 kHz and 10 MHz. One-shot timeouts suffered an absolute worst-case increase in timeout error E_τ of 0.34 % while lowering it by 0.23 % in the best case. Periodic timeouts produced similar results when repeated only once. However, after 100 timeout cycles at the latest, the performance differences between APIs disappeared on 82 % of all boards. On the remaining devices, one or the other API performed better.

Performance under stress was characterized by measuring the callback execution delay D_{cb} with up to four simultaneously expiring timeouts. Both timer interfaces showed nearly identical scaling behavior, with all differences being less than 1 %. uTimer serviced compare match interrupts slightly faster than `periph_timer` was able to on 66 % of all boards. Although this advantage was only marginal within our isolated test scenario ($\leq 0.1\,\%$), it can become beneficial for time-critical real-world applications.

In summary, uTimer generally scored slightly worse than `periph_timer` within most of our benchmarks. However, all observed performance differences were found to have little to no impact on typical use cases within the IoT or WSNs. Only ultra short timeouts ($\tau \leq 10\,\mu s$) that require high precision were identified as problematic. However, these are just rarely used and leveraging the hAPI directly or performing active waiting is suggested instead. We therefore consider uTimer a valid drop-in replacement to existing APIs from a performance point of view.

7.5.2 Abstraction Trade-off

We isolated the computational effort inherent to the timer APIs themselves within Section 7.4.5. Results showed that the user-facing API requires between six and 21 additional CPU cycles. Using the hardware-facing API directly yielded comparable performance to the existing API on about half of all boards. On the other half, the hAPI of uTimer outperformed `periph_timer` by an amount that nearly offset the additional uAPI overhead. However, all performance differences, both positive and negative, were found negligible when taking extrinsic factors of real-world scenarios, differences in CPU and timer frequency, and the impact on typical timeouts into account. A closer look at the uAPI overhead was taken nonetheless. The user-facing API constitutes an additional layer of functions that abstract from the underlying device-specific driver. To identify the different factors that contribute to the observed overhead, firmware binaries of our uTimer benchmark suite were disassembled and analyzed.

Invoking uAPI functions naturally produces additional branching instructions. These can potentially be eliminated by declaring the respective functions as `inline`. While this could make a function faster, its impact on code size is hard to predict because it depends on the number of invocations and their individual contexts[13]. Our performance benchmarks confirmed uTimer to perform sufficiently well for typical applications, even with compiler size optimizations enabled. In addition, microcontrollers are usually more limited in terms of available ROM than CPU power. We therefore decided against the small performance gains from inlining uAPI functions in favor of valuable flash memory.

Besides the branching instructions, resolving the corresponding hAPI driver function was found to be another source of overhead. We analyzed the executed processor instructions for multiple consecutive timer function calls to identify potential optimizations. Despite the timer instance as well as its driver being constant and the executed operation remained identical, dereferencing of the driver function pointer was repeated prior to every branch instruction. Building with different `gcc` optimization levels (`-Os`, `-O[0-3]`) yielded identical results at all times, while the `const` pointer dereferencing was never optimized. Even though there are solutions to this problem, their effect heavily depends on the used compiler toolchain and selected optimization level [59]. Both can greatly differ between MCU platforms in RIOT. We therefore consider the observed dereferencing overhead as inevitable at a generic level. Application developers can nonetheless fine-tune such aspects, for example, by instructing the compiler to perform certain optimizations via the `CFLAGS_OPT` variable inside the application Makefile.

Remaining instructions are predominantly caused by either compound or convenience uAPI functions, as explained in Section 5.2.4. Since they contribute new functionality, they naturally introduce a number of additional processor instructions for performing their tasks.

[13] Inline Functions - `gcc` documentation: `https://gcc.gnu.org/onlinedocs/gcc/Inline.html` (Accessed: 08.02.2022)

7.5.3 Binary Size and Memory Usage

The additional abstraction layers and time-memory optimizations both contribute to the overall memory footprint of applications. With uTimer, each exposed timer instance is represented by one `utim_periph_t` struct. It consists of two pointers and one 16-bit field, requiring a total of 12 bytes on 32-bit devices[14]. Each `utim_driver_t` hAPI driver struct consists of seven pointers, hence requires 28 bytes on 32-bit devices. However, drivers are only included in the final binary if used by at least one active timer, as selected during compile-time via Kconfig. The size of uTimer data structures s_{utim} therefore can be calculated according to Equation (7.6). Here, n_{timers} indicates the number of exposed timer instances, $n_{drivers}$ indicates the number of required hAPI drivers, and s_{ptr} specifies the pointer size of the target device in bytes. Additional struct padding p may occur on CPUs with a word size other than 16 bit. Using four timers of two different types requires a total of 104 bytes to store timer instances and driver structs on 32-bit MCUs. The absolute minimum memory requirement for a single timer is 20 bytes and increases by only 6 bytes for every additional timer instance of the same type.

$$s_{utim} = n_{timers} \cdot [2 \cdot s_{ptr} + 2\,\text{byte} + p] \;+\; n_{drivers} \cdot [7 \cdot s_{ptr}] \tag{7.6}$$

Besides the generic API footprint, the platform-dependent driver code largely affects memory consumption as well. The more timer types are used, the more driver code has to be stored. Having only a single unified API, in turn, potentially frees up memory that previously was consumed by multiple timer-type specific APIs. The final ROM requirement is therefore highly dependent on both the individual application and the target platform implementation. RAM use remained constant throughout all our evaluations.

Since we found no significant differences between `gcc` optimization levels with respect to timer performance, optimizing for binary size (`-Os`) is generally suggested. This is already by far the most frequently used optimization level within RIOT. Additional flash memory can be required if the developer explicitly instructs the compiler to make extensive use of function inlining, as described in the previous subsection. However, this is not the default case and its impact must be weighed against the individual application requirements and available hardware resources.

7.5.4 Peripheral Availability

One of the main goals of the proposed API design is allowing application developers to choose the best suiting timers from the full range of peripherals the target board has to offer. Table 7.3 compares the total number of available hardware timers and their channels between a combined use of all three existing low-level interfaces (`periph_timer`/`rtt`/`rtc`) and the exclusive use

[14] Two additional bytes are added due to struct padding. This effect is not present on 16-bit architectures.

Board	Timers (Channels) Available	Existing APIs	uTimer	Increase
arduino-mega2560	6 (16)	3 (7)	6 (16)	+3 (+9)
esp32-wroom-32	5 (5)	4 (4)	4 (4)	0 (0)
esp8266-esp-12x	2 (2)	1 (1)	2 (2)	+1 (+1)
nucleo-f070rb	10 (14)	2 (5)	9 (14)	+7 (+9)
nucleo-f103rb	6 (17)	3 (9)	5 (17)	+2 (+8)
nucleo-f767zi	17 (35)	3 (6)	16 (35)	+13 (+29)
nucleo-g474re	14 (47)	3 (6)	13 (47)	+10 (+41)
nucleo-l152re	11 (22)	2 (5)	10 (22)	+8 (+17)
nucleo-l476rg	15 (32)	3 (6)	14 (32)	+11 (+26)
slstk3400a	5 (11)	3 (4)	4 (11)	+1 (+7)
slstk3401a	6 (11)	4 (6)	5 (11)	+1 (+5)
slstk3402a	9 (18)	4 (9)	8 (18)	+4 (+9)
stk3200	4 (8)	3 (4)	3 (8)	0 (+4)
z1	2 (10)	1 (3)	2 (10)	+1 (+7)

Table 7.3: Exposure of timer peripherals and channels: Combined use of existing APIs compared to uTimer. The *Increase* column indicates the number of timers that are only usable via uTimer. Watchdog timers are excluded from this overview.

of uTimer. Results confirm that our novel API makes nearly all timers and channels available to the application. It is capable of exposing up to 13 additional timers on 86 % and up to 41 additional compare channels on 93 % of all boards. This not only mitigates resource conflicts between user applications and operating system modules, but also allows utilizing the available peripherals to their full extent.

Besides their sheer number, the types of available timers are equally important. With STM32 based boards, existing APIs only expose one *Low-power Timer*, leaving further peripherals of this type unusable. *Advanced-control Timers* require manual configuration, including changes to system header files, and *Basic Timers* remain fully unsupported. *SysTick Timers*, commonly found on all ARM Cortex MCUs, are left unexposed by all APIs. However, these are usually exclusively occupied by the scheduler or other OS modules, hence are neither exposed via existing APIs nor uTimer. On the ESP32, all available timers are exposed, except one general-purpose timer that is reserved for the OS. Its predecessor, the ESP8266, offers a real-time clock, but the required driver is currently not provided. The MSP430x possesses two general-purpose timers, of which only the smaller *TIMER_A* is exposed by `periph_timer`. uTimer additionally allows using the more powerful *TIMER_B*. Finally, none of the existing APIs allows using EFM32 ultra low-power *Cryotimer* peripherals or *Pulse Counters*. This common lack of support for advanced timer types impedes low-power optimizations in particular. Using uTimer, developers are able to select the best suited timers for their specific applications among the full range of available peripherals and utilize their full set of features.

7.5.5 Virtual Timer Drivers

The concept of virtual drivers is described in Section 5.2.3. They introduce hierarchical structures to the hardware-facing API, allowing to inherit certain functionality from other base drivers. The feasibility of this concept was put to the test using the example of timer chaining on the EFM32 platform. Details of the respective implementation are described in Section 6.3.3.

Applying virtual hAPI drivers, we were able to expose two distinct hardware timers as an atomic timer instance to the application. It possessed a greatly extended counter range and allowed for comprehensive fine-tuning of the timer frequency. All virtual timer instances could transparently be interchanged with regular ones. The reusable driver could moreover successfully be used to control the larger timers on that device, hereby yielding counter ranges of up to 2^{64} ticks. Both configurations were interactively selectable and their mutual exclusive use was ensured. We therefore consider virtual drivers as a beneficial part of our proposed API design.

7.5.6 Code Quality and Usability

Besides the pure performance of an application programming interface stands its usability and maintainability. We have found current interfaces to have a great potential for optimization regarding both. This section discusses the most important qualitative aspects and how they are addressed by uTimer.

Individual timer peripherals are simultaneously exposed via at least two of the existing low-level APIs on almost all platforms. This frequently results in resource allocation conflicts, which in turn can lead to undefined behavior if remained unnoticed. uTimer eliminates this problem by managing timer selection inherently and providing a uniform interface that is independent of the underlying timer type. Moreover, on some platforms, as for example all EFM32 based devices, different timer types are exposes via the same low-level API. This forces developers to combine multiple timer-type specific implementations within a single function, making distinction between them necessary during every function call. This not only degrades performance, but also lowers code quality and hides information about the peripheral type from the user. Some drivers furthermore require error-prone workarounds to determine timer properties, again impairing maintainability. During initialization of EFM32 timers, for example, the peripherals internal memory base address is evaluated to determine the timer type. It is then used to deduce the maximum counter width and set the auto-reload register accordingly. uTimer instead natively supports multiple timer types, and timer properties can dynamically be accessed during run-time. User applications, peripheral drivers, and other OS modules can directly use this information without error-prone workarounds.

Timeout τ	Frequency f	Latency L_τ	Error E_τ
1 s	10 kHz	11.5 µs	$< 0.01\,\%$
1 s	100 kHz	655.4 ms	$65.54\,\%$
1 s	1 MHz	983.0 ms	$98.30\,\%$
1 s	10 MHz	996.8 ms	$99.68\,\%$

Table 7.4: Timeout latencies and timeout error with unsupported parameters using `periph_timer` on the Nucleo-F070RB

Existing low-level timer modules are very similar in terms of the functions they expose, their interfaces nonetheless differ largely. Neither can available timer features thereby be comprehensively exposed to the user, nor can peripherals be addressed in a uniform way. Developers therefore often find themselves required to (re-)implement certain hardware features. This leads to inconsistent and hard to maintain timer code within user applications, high-level OS modules, and peripheral drivers. With uTimer, multiple timer types are natively supported and enabled timers can be addressed both explicitly (e.g. *STM32_LPTIM2*) and platform-independent (e.g., *Timer 0*). Ready to use peripheral mappings and their convenient selection relieve application developers from modifying OS code and deep-diving into vendor datasheets or SDKs. A common pattern for device configuration is established and existing APIs are streamlined into a single interface, fostering uniformity of timer code throughout the whole RIOT ecosystem.

During our benchmarks we noticed irregularities with certain parameter combinations. The lack of specification regarding behavior for invalid timer configurations and inconsistent error handling were identified as their cause. Table 7.4 shows an example in which the performance of 1 s one-shot timeouts via `periph_timer` is measured for varying counter frequencies. Generating a timeout on a 10 kHz timer results in a timeout error of less than 0.01 %. Instructing the API to configure the same timer to 1 MHz instead, skyrockets the timeout error to 98.30 %. Since behavior for invalid parameters is not explicitly specified, implementations usually neither verify requested frequencies nor timeout durations. An established convention is to select the closest achievable timer frequency or compare channel value. Both cases lead to unpredictable timeout lengths, resulting in the observed failure. uTimer addresses this problem by requiring implementations to signal an error if the requested frequency is invalid or the desired timeout is out of range. A uAPI method to determine the closest achievable frequency is provided and counter limits can be determined by the static properties within every timer instance struct.

All the above highlighted measures significantly benefit the API usability and foster uniform timer code that is easy to maintain. Therefore, we likewise consider uTimer to be a user-friendly and future-proof complement to the existing low-level timer interfaces in RIOT from a qualitative point of view.

7.5.7 Automation

The developed unit tests and benchmark suites integrate seamlessly with the CI and HiL infrastructure that is operated by the RIOT community[15]. Its architecture is described in detail within Section 7.3.4. Test suites are executed every night on the main branch as well as for each individual pending pull request. Figure 7.15 shows a summary of the basic timer test suite.

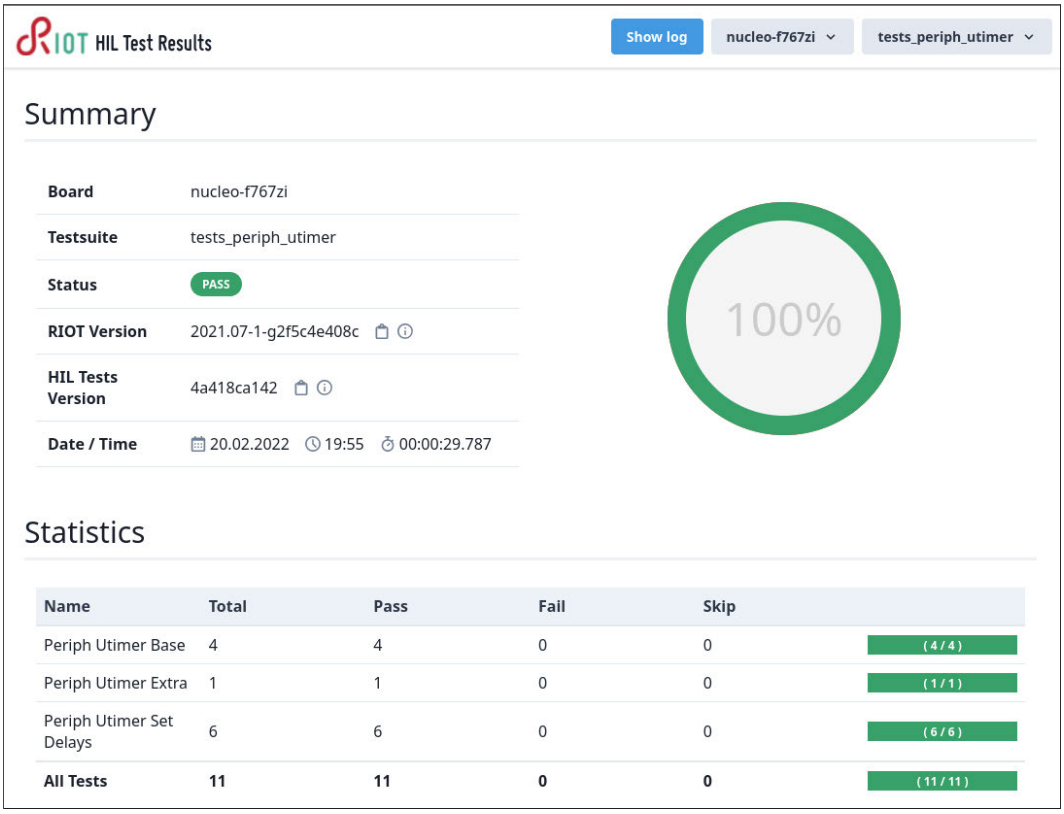


Figure 7.15: Screenshot of a HiL unit test report

Detailed reports are automatically generated for unit tests and performance benchmarks alike. Results are made available online and can interactively be explored down to their individual instructions for each board. All test and benchmark suites can also be executed at home using arbitrary hardware, requiring only a PHiLIP HiL controller board and the desired device under test. The data that is produced by tests and benchmarks provides developers with crucial insights and guidance. It encourages the development of performant driver implementations and simultaneously allows for comprehensive regression testing to ensure the reliability of RIOT OS across supported boards. We consider the broad and effortless availability of such information to be fundamental to providing a high-quality embedded operating system.

[15]RIOT HiL test results overview: `https://hil.riot-os.org/results` (Accessed: 09.02.2022)

7.5.8 Integration

Various components of an embedded operating system interact with hardware timers. Among these, high-level timer subsystems place the highest demands on the low-level timer API and the underlying peripheral drivers. Although a great variety of such systems exist, our analysis identified common requirements, which we therefore accounted for within the design of uTimer.

High-level timer modules usually multiplex virtual software timeouts onto a number of hardware timers and their channels. Exposing all provided compare channels improves the performance of timer maintenance by allowing the separation of virtual timeouts based on duration or priority. It furthermore can reduce the number of hardware timers that are required to generate all desired timer ticks down to a single low-power peripheral. This leaves more peripherals available to the operating system and user applications, while also significantly lowering energy consumption for common IoT use cases, such as autonomous sensor nodes. Reliable determination of timer characteristics and capabilities, as comprehensively provided by uTimer, further aids the dynamic utilization of available peripherals. It permits the implementation of heuristic solutions to high-level timeout management.

Certain peripheral maintenance must be performed by a timer subsystem whenever a hardware timer overflows. Exposing the corresponding OVF interrupts hugely benefits the performance of this recurring task. Otherwise, periodically polling the respective status flag is inevitable. The separation of overflow and compare match interrupts within uTimer furthermore decouples the low-priority maintenance of the underlying timer hardware from the high-priority task of servicing running timeouts. This enables timer subsystems to prioritize the signaling of expired timeouts over the non time-critical peripheral maintenance.

Most high-level timer APIs require timer features beyond the generation of just a monotonic clock tick. uTimer comprehensively exposes even advanced timer features while allowing to determine their availability during run-time. Timer subsystems can thereby dynamically adjust their behavior based on the hardware they are given. However, benefits are not limited to timekeeping tasks. The access to power-gating features and the run-time reconfigurability of clock sources, for example, can be leveraged by power saving frameworks. The uniform timer identifiers, to give another example, allow the management of timer peripherals by an external allocator module.

Considering the above, it can be concluded that uTimer constitutes a sound base to build high-level timer subsystems on. The novel API easily integrates with other modules of the operating system and can dynamically be utilized by user applications.

7.5.9 Issues

uTimer was implemented and evaluated to the best of our knowledge, nonetheless some issues naturally remain. This section exclusively discusses problems that are related to our evaluation or were revealed by it. All implications and trade-offs that arise from the API design itself were previously discussed in Section 5.4.

Some platforms are still missing implementations for advanced timer features and types. This issue is not inherent to uTimer itself, but rather caused by a sheer lack of time. Driver support for such device-specific features is desired nonetheless. However, the primary goal of this work is not to provide fully-featured peripheral drivers for every MCU that RIOT targets, but the contribution of a design for a sound low-level timer API and its evaluation instead. We have therefore decided against striving for most complete driver implementations in favor of a comprehensive evaluation. As a consequence, platform-specific features are not yet covered by the automated unit tests.

Conflicting clock configurations are not dynamically determined by uTimer, as explained within Section 5.2.6. Driver developers instead specify a fixed set of applicable clocking options for each timer type during platform implementation. However, in the course of our evaluation we have found this approach to exclude clock configurations whose feasibility depends on the configuration of other peripherals during run-time. We intentionally decided against performing clock-tree evaluations within uTimer, but suggested implementing clock management on a system-wide scale instead. This nonetheless currently prevents the described clock configurations despite their potential feasibility.

Unused timer drivers are automatically excluded during build to preserve flash space. An analysis of the memory footprint of uTimer revealed that such compiler optimizations only happen on a per-driver basis. This means that whenever only a single function from a driver is used, its whole set of hAPI functions is included into the firmware binary. We assume this issue to be a side effect of the function pointer approach and that it relates to the `const` pointer optimization described in Section 7.5.2. Unfortunately, we were unable to solve this problem in a timely manner, wherefore it is still present.

8 Conclusion and Outlook

Peripheral APIs should expose underlying hardware to its fullest extent. A low-level timer API in particular must also contain its computational overhead to maintain timeout accuracy and ensure system responsiveness. Tight resource constrains of target devices and the heterogeneous spectrum of boards supported by modern embedded OSs render this a complex problem. The challenge in designing a sound and future-proof low-level timer API lies therefore in choosing an appropriate amount of abstraction to extensively expose timer hardware while achieving portability and maintaining performance at the same time.

With this work, we contributed four detailed analyses of various aspects of timekeeping tasks on embedded devices, including a large-scale survey of hardware timers on 43 microcontroller platforms. Gathered insights enabled us to develop a comprehensive and sustainable design for a uniform low-level timer API. The proposed *uTimer* was implemented for RIOT and already supports six device families on 129 boards in total. Cross-platform timer test suites verified the proper behavior of all our implementations in an automated fashion. We developed extensive timer benchmarks and compared the performance of uTimer to the existing low-level timer APIs of RIOT. Results confirm uTimer to be a sound alternative to existing APIs from a performance point of view. Its hardware-facing API is as fast as `periph_timer`, while its uniform user-facing API adds between six and 21 CPU cycles overhead due to the extra layer of abstraction. Once at least two timer types are available, the uAPI and `periph_timer` perform alike. Even the worst-case uAPI overhead increases the timeout error by no more than 0.05 %, thus is negligible for typical IoT use cases. If ultra-high performance is required, the hAPI can optionally remove any abstraction overhead at the cost of deliberately narrowing application portability.

uTimer expands the set of hardware timers that are made available to user applications and high-level OS modules in both number and diversity. On average, it triples the amount of timers and channels, while also exposing numerous timer types, including ultra low-power peripherals, which previously remained unexposed by all three existing APIs. The novel uTimer allows to use all timers transparently interchangeable via its platform-independent interface, hereby fostering portability and uniformity of timer code throughout the whole RIOT ecosystem. Its flexible design enables high-level system modules and timer subsystems in particular to base their generic solutions on uTimer. Application developers, on one hand, are provided with an encompassing and reliable timer hardware abstraction that relieves them from writing low-level

driver code. OS developers, on the other hand, benefit from the contributed unit tests and performance benchmarks, encouraging the development of sustainable API implementations and permitting their profound evaluation.

To conclude, the proposed uTimer API is a performant, user-friendly, and future-proof addition to the existing low-level timer interfaces of RIOT. Our comprehensive analyses and the contributed evaluation methods moreover provide developers with valuable insights and guidance throughout the entire systems development life cycle.

8.1 Future Work

This work constitutes a solid foundation for future enhancements to the timekeeping features of RIOT. First, we aspire to develop a uTimer backend for the high-level `ztimer` API. This integration makes uTimer usable with all hard- and software timeouts throughout RIOT. After laying this ground, we suggest leveraging the timer types and features that are now available to improve the existing higher-level timekeeping solutions. This includes, for example, using ultra-low power peripherals as tick sources, class-based timeout multiplexing, and dynamic peripheral reconfiguration. In parallel to this, current driver implementations and the range of supported microcontrollers shall be extended.

Conceptually, the definition of abstract timer classes is aspired. These shall group timer types that provide a certain set of features, allowing application developers to select peripherals based on the required properties in a platform-independent fashion. The exploration of compile-time optimizations may reveal additional opportunities to further reduce the memory footprint. Moreover, an in-depth analysis of timer clocking options will yield valuable information to base the design of a generic clock-management module on. Last, the feasibility of integrating peripheral interconnect systems into the API design shall be evaluated, as those possess the potential to obviate CPU wake-ups during timer maintenance.

Bibliography

[1] F. Adelantado, X. Vilajosana, P. Tuset-Peiro, B. Martinez, J. Melia-Segui, and T. Watteyne, "Understanding the Limits of LoRaWAN," *IEEE Communications Magazine*, vol. 55, no. 9, pp. 34–40, 2017. DOI: 10.1109/mcom.2017.1600613.

[2] Arm Ltd., *Common Microcontroller Software Interface Standard (CMSIS)*. [Online]. Available: https://www.arm.com/why-arm/technologies/cmsis (visited on 10/26/2021).

[3] E. Baccelli, C. Gündogan, O. Hahm, P. Kietzmann, M. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch, "RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT," *IEEE Internet of Things Journal*, vol. 5, pages 4428–4440, Dec. 2018. DOI: 10.1109/jiot.2018.2815038.

[4] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, and T. C. Schmidt, "RIOT OS: Towards an OS for the Internet of Things," in *Proceedings of the 2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, Piscataway, NJ, USA: IEEE Press, Apr. 2013, pages 79–80. DOI: 10.1109/INFCOMW.2013.6970748.

[5] A. Banks and R. G. (Eds.), "MQTT Version 3.1.1," OASIS, OASIS Standard, Oct. 2014. [Online]. Available: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html.

[6] R. Barry and Amazon Web Services, *FreeRTOS: Real-time Operating System for Microcontrollers. Project Website*. [Online]. Available: https://www.freertos.org/ (visited on 12/07/2021).

[7] P. Bellasi, "Linux Power Management Architecture: A review on Linux PM frameworks," Politecnico di Milano, Dipartimenti di Elettronica e Informazione, Tech. Rep., Dec. 2010.

[8] BlackBerry and QNX Software Systems, *Blackberry QNX Website*, English. [Online]. Available: https://blackberry.qnx.com/ (visited on 12/22/2021).

[9] Bluetooth Special Interest Group, "Bluetooth Core Specification," Bluetooth SIG, Bluetooth Specification 5.1, Jan. 2019. [Online]. Available: https://www.bluetooth.com/specifications/bluetooth-core-specification.

[10] R. Braden, "Requirements for Internet Hosts - Communication Layers," IETF, RFC 1122, Oct. 1989.

[11] Contiki Developer Community, *Contiki-OS Source Code Repository*. [Online]. Available: https://github.com/contiki-os/contiki (visited on 12/07/2021).

[12] Contiki-NG Developer Community, *Contiki-NG Source Code Repository*. [Online]. Available: https://github.com/contiki-ng/contiki-ng (visited on 12/07/2021).

[13] *corbet* (Pseudonym), *Clockevents and dyntick*, News Article. Released in Linux Weekly News (LWN), Feb. 2007. [Online]. Available: https://lwn.net/Articles/223185/ (visited on 01/30/2020).

[14] ——, *Deferrable timers*, News Article. Released in Linux Weekly News (LWN), Mar. 2007. [Online]. Available: https://lwn.net/Articles/228143/ (visited on 01/30/2020).

[15] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," IETF, RFC 8200, Jul. 2017.

[16] DeviceTree Community, *Devicetree Specification, Release v0.3*, English, Online, Feb. 2020. [Online]. Available: https://www.devicetree.org/specifications/.

[17] A. Dimitrievski, S. Filiposka, B. Cico, and V. Trajkovik, "Energy conservation using ultra low power timers for sustainable environmental monitoring," IEEE, Jun. 2021. DOI: 10.1109/meco52532.2021.9460145.

[18] R. Droms, "Dynamic Host Configuration Protocol," IETF, RFC 2131, Mar. 1997.

[19] T. Gleixner and D. Niehaus, "Hrtimers and Beyond: Transforming the Linux Time Subsystems," in *Proceedings of the 2006 Ottawa Linux Symposium (Volume One)*, Jul. 2006, pages 333–346.

[20] G. Gracioli, D. Santos, R. Matos, L. Wanner, and A. Fröhlich, "One-shot time management analysis in EPOS," in *Proceedings of the International Conference of the Chilean Computer Science Society*, Nov. 2008, pages 92–99. DOI: 10.1109/SCCC.2008.13.

[21] C. Gundogan, P. Kietzmann, M. S. Lenders, H. Petersen, M. Frey, T. C. Schmidt, F. Shzu-Juraschek, and M. Wahlisch, "The Impact of Networking Protocols on Massive M2M Communication in the Industrial IoT," *IEEE Transactions on Network and Service Management*, vol. 18, no. 4, pp. 4814–4828, Dec. 2021. DOI: 10.1109/tnsm.2021.3089549.

[22] V. Handziski, J. Polastre, J.-H. Hauer, C. Sharp, A. Wolisz, and D. Culler, "Flexible Hardware Abstraction for Wireless Sensor Networks," in *Proceeedings of the Second European Workshop on Wireless Sensor Networks*, Feb. 2005, pages 145–157, ISBN: 0-7803-8801-1. DOI: 10.1109/EWSN.2005.1462006.

[23] D. Hanes, G. Salgueiro, and R. Barton, *IoT Fundamentals: Networking Technologies, Protocols, and Use Cases for the Internet of Things*, ser. Cisco Press Fundamentals Series. Cisco Press, 2017, ISBN: 9780134307091.

[24] M. Hellman, "A cryptanalytic time-memory trade-off," *IEEE Transactions on Information Theory*, vol. 26, no. 4, pages 401–406, 1980. DOI: 10.1109/tit.1980.1056220.

[25] W. Hofer, "Sloth: The Virtue and Vice of Latency Hiding in Hardware-Centric Operating Systems," Doctoral Thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Aug. 2014.

[26] W. Hofer, D. Danner, R. Muller, F. Scheler, W. Schröder-Preikschat, and D. Lohmann, "Sloth on Time: Efficient Hardware-Based Scheduling for Time-Triggered RTOS," in *Proceedingss of the 33rd IEEE Real-Time Systems Symposium*, Dec. 2012, pages 237–247. DOI: 10.1109/RTSS.2012.75.

[27] IEEE 802.15 Working Group, "IEEE Standard for Low-Rate Wireless Networks," IEEE, New York, NY, USA, Tech. Rep. IEEE Std 802.15.4™–2015 (Revision of IEEE Std 802.15.4-2011), 2016, pp. 1–709.

[28] A. Ismail, "Automated Testing of the RIOT-OS Timer Subsystem," Dec. 2020. [Online]. Available: https://inet.haw-hamburg.de/thesis/completed/ba_aiman_ismail.pdf.

[29] R. Kamal, *Embedded Systems: Architecture, Programming and Design*, second. Tata McGraw Hill Education, 2011, ISBN: 978-0-070-66764-8.

[30] V. B. Kleeberger, S. Rutkowski, and R. Coppens, "Design & Verification of Automotive SoC Firmware," in *Proceedings of the 52nd Annual Design Automation Conference*, ser. DAC '15, San Francisco, California: ACM, Jun. 2015, ISBN: 9781450335201. DOI: 10.1145/2744769.2747918.

[31] P. Koopman, "Embedded system design issues (the rest of the story)," in *Proceedings International Conference on Computer Design. VLSI in Computers and Processors*, IEEE Comput. Soc. Press, Oct. 1996, pp. 310–317. DOI: 10.1109/iccd.1996.563572.

[32] A. Köpke and J.-H. Hauer, "IEEE 802.15.4 Symbol Rate Timer for TelosB," Telecommunication Networks Group (TKN), TU Berlin (TUB), Berlin, Germany, Technical Report TKN-08-006, May 2008.

[33] M. Lenders, P. Kietzmann, O. Hahm, H. Petersen, C. Gündoğan, E. Baccelli, K. Schleiser, T. C. Schmidt, and M. Wählisch, "Connecting the World of Embedded Mobiles: The RIOT Approach to Ubiquitous Networking for the Internet of Things," Jan. 2018. arXiv: 1801.02833 [cs.NI].

[34] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, "TinyOS: An Operating System for Sensor Networks," in *Ambient Intelligence*, W. Weber, J. M. Rabaey, and E. Aarts, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 115–148, ISBN: 978-3-540-27139-0. DOI: 10.1007/3-540-27139-2_7.

[35] P. Lindgren, E. Fresk, M. Lindner, A. Lindner, D. Pereira, and L. M. Pinho, "Abstract Timers and Their Implementation onto the ARM Cortex-M Family of MCUs," *ACM SIGBED Review*, vol. 13, pages 48–53, Mar. 2016. DOI: 10.1145/2907972.2907979.

[36] Linux Foundation, *The Linux Kernel documentation: Kconfig language*, English, version v. 5.16.0-rc4. [Online]. Available: https://www.kernel.org/doc/html/latest/kbuild/kconfig-language.html (visited on 12/09/2021).

[37] Linux Foundation and Wind River Systems, *Zephyr API Reference: RTC*, version 2.7.99. [Online]. Available: https://docs.zephyrproject.org/latest/reference/peripherals/rtc.html (visited on 12/07/2021).

[38] ——, *Zephyr API Reference: Timers*, version 2.7.99. [Online]. Available: https://docs.zephyrproject.org/latest/reference/kernel/timing/timers.html (visited on 12/07/2021).

[39] ——, *Zephyr Project Website*. [Online]. Available: https://zephyrproject.org/ (visited on 12/07/2021).

[40] LoRa Alliance – Technical Committee, "Lorawan 1.1 specification," LoRa Alliance, Tech. Rep., Oct. 2017. [Online]. Available: https://lora-alliance.org/sites/default/files/2018-04/lorawantm_specification_-v1.1.pdf.

[41] S. Maji, U. Banerjee, S. H. Fuller, M. R. Abdelhamid, P. M. Nadeau, R. T. Yazicigil, and A. P. Chandrakasan, "A low-power dual-factor authentication unit for secure implantable devices," IEEE, Mar. 2020. DOI: 10.1109/cicc48029.2020.9075945.

[42] Microsoft, *Windows Embedded Compact | Microsoft Docs*, English. [Online]. Available: https://go.microsoft.com/fwlink/p/?linkid=335870 (visited on 01/24/2021).

[43] V. Mincev and D. Milicev, "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, 1998, pages 1037–1046, ISBN: 978-3-540-69756-5. DOI: 10.1007/3-540-64359-1_769.

[44] C. Neely, G. Brebner, and W. Shang, "Flexible and modular support for timing functions in high performance networking acceleration," IEEE, Aug. 2010. DOI: 10.1109/fpl.2010.102.

[45] J. Postel, "Transmission Control Protocol," IETF, RFC 793, Sep. 1981.

[46] ——, "User Datagram Protocol," IETF, RFC 768, Aug. 1980.

[47] F. Reverter, "Toward Non-CPU Activity in Low-Power MCU-Based Measurement Systems," *IEEE Transactions on Instrumentation and Measurement*, vol. 69, no. 1, pp. 15–17, Jan. 2020. DOI: 10.1109/tim.2019.2953374.

[48] RIOT Developer Community, *RIOT OS Project Website*. [Online]. Available: https://www.riot-os.org/ (visited on 12/19/2021).

[49] Robot Framework Foundation, *Robot Framework Project Website*. [Online]. Available: https://robotframework.org/ (visited on 01/19/2022).

[50] M. Rottleuthner, T. C. Schmidt, and M. Wählisch, *FlexClock: Generic Clock Reconfiguration for Low-end IoT Devices*, 2021. arXiv: 2102.10353 [eess.SY].

[51] P. Sagar and V. Agarwal, "Embedded Operating Systems for Real-Time Applications," Electronic Systems Group, EE Dept, IIT Bombay, Tech. Rep., Nov. 2002. [Online]. Available: https://www.ee.iitb.ac.in/~esgroup/es_mtech02_sem/es02_sem_rep_sagar.pdf.

[52] C. Sharp, M. Turon, and D. Gay, *TinyOS Enhancement Proposal #102: Timers*, English, Online, TinyOS Alliance, Sep. 2007. [Online]. Available: https://github.com/tinyos/tinyos-main/raw/master/doc/txt/tep102.txt.

[53] Z. Shelby, K. Hartke, and C. Bormann, "The Constrained Application Protocol (CoAP)," IETF, RFC 7252, Jun. 2014.

[54] I. Susnea and M. Mitescu, *Using the MCU Timers: Microcontrollers in Practice (Springer Series in Advanced Microelectronics)*, first. Berlin, Heidelberg: Springer-Verlag, 2005, pages 67–91, ISBN: 978-3-540-28308-9. DOI: 10.1007/3-540-28308-0_6.

[55] A. S. Tanenbaum, *Moderne Betriebssysteme*, 3., aktualisierte Auflage. Pearson Studium, Apr. 2009, ISBN: 978-3-8273-7342-7.

[56] TinyOS Alliance, *TinyOS Project Website*. [Online]. Available: https://www.riot-os.org/ (visited on 12/07/2021).

[57] S. Vongsingthong and S. Smanchat, "Internet of Things: A Review of Applications and Technologies," *Suranaree Journal of Science and Technology*, vol. 21, pp. 359–374, Jun. 2014. DOI: 10.14456/SJST.2014.38.

[58] K. Weiss, M. Rottleuthner, T. C. Schmidt, and M. Wählisch, "PHiLIP on the HiL: Automated Multi-platform OS Testing with External Reference Devices," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 20, no. 5s, pp. 1–26, Aug. 2021. DOI: 10.1145/3477040.

[59] C. Wellons, *Const and Optimization in C*, Jul. 2016. [Online]. Available: https://nullprogram.com/blog/2016/07/25/ (visited on 02/08/2022).

[60] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J. Vasseur, and R. Alexander, "RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks," IETF, RFC 6550, Mar. 2012.

A Hardware Analysis Results

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

A.1 Column Key for Analysis Criteria

All analysis criteria that are listed in the result tables are defined in this section.

A.1.1 Timer Type

Name of a timer type. Generic timer modules are grouped under the type name ”General-purpose” across all platforms to allow for easy identification 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 different 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 provide a raw counter value, 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 discrete values.

A.1.4 Prescaler Type

Availability of a prescaler that can divide the timer clock. Can be one of the following:

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

A.1.5 Max Prescaler

Maximum value that can be selected as a prescaler with respect to *Prescaler Type*. This is the largest configurable clock divider, hence resulting in the longest time to over- or underflow. 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 single larger one, e.g., combining two 16-bit timers into one 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.

A.1.7 Compare INT

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

- × Non-existing. Compare channel 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. No status bit or event flag read is necessary to identify the triggered compare channel.

A.1.8 Overflow INT

Type of the interrupt that is generated on a counter register over- or underflow. Can be one of the following:

- × Non-existing. A counter over- or 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. No status bit or event flag read is necessary to distinguish it from compare matches.

A.1.9 Event Flags

Availability of status bits that indicate if an event, e.g., compare match or overflow, 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 or flags available.
- ✓ Event status bits or 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 reloads only at counter over- or 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. An exclusive auto-reload value register is provided (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 counts as not available.
- ✓ PWM generation available.

A.1.12 Internal CLKs

The number of internal clocks the timer is able to run of. A single clock is categorized as internal, if it can be driven by an internal oscillator in at least one configuration. 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`. However, if 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

The number of external clocks the timer is able to run of. A single clock is categorized as external, if it can be driven by an external oscillator in at least one configuration. 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, either internal or external. A low-power clock is defined as allowing the CPU and high-frequency peripheral clocks 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 driven by a low-power clock source, hence 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 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

Occasionally one of the above described properties does not apply to a timer module, e.g., counter width for some real-time-clocks. It can also be currently unknown or require further 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

A.2 Timer Comparison Matrices (TCMs)

Timer comparison matrices for all analyzed device families are listed below.

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 A.1: 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
[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	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 A.2: 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 A.3: 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)°	16 bit	3	F	2^{7}	✓	○	○	✓	□	✓	2	5	✓	×
Pulse Width Modulation (PWM)°	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 A.4: Timer Comparison Matrix: Microchip / Atmel SAM3

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 A.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	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 A.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	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 A.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	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 A.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	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 A.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	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 A.10: 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 A.11: 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 A.12: 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	✓[1]	×
Watchdog (SysClk) Watchdog (PIOSC)	32 bit	0	×	-	×	×	✓	✓	×	×	1 1	1 0	✓ ×	✓

Table A.13: 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 A.14: 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
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 A.15: 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 A.16: Timer Comparison Matrix: SiFive FE310-Gx

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 A.17: 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 A.18: 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 A.19: Timer Comparison Matrix: NXP Semiconductors LPC2387

B uTimer API Definition

Header definitions of the uTimer API, i.e., `periph_utimer`, are listed in this chapter. Code listings are split into type definitions (see Section B.1), hardware-facing API (hAPI) (see Section B.2), and user-facing API (uAPI).

The full interface definition can be found in `/drivers/include/periph/utimer.h`.

B.1 Type Definitions

Type definitions that are used by both the hAPI and the uAPI are listed in this section.

```
/**
 * @brief   Default timer device identifier type
 */
typedef unsigned int utim_t;

/**
 * @brief   Default timer definition macro
 *
 * Overwrite this in your CPUs periph_cpu.h file if needed
 */
#ifndef UTIMER_DEV
#define UTIMER_DEV(x) (x)
#endif

/**
 * @brief   Value for utim_t to indicate an invalid device
 *
 * Needs to be redefined if a custom utim_t is used
 */
#ifndef UTIMER_DEV_INVALID
#define UTIMER_DEV_INVALID (UINT_MAX)
#endif
```

Listing 9: Definition of the internal device identifier type `utim_t`. Additional preprocessor macros to represent an invalid device and for optional type conversions are provided.

```c
/**
 * @brief   Type for timer peripheral instances available on a MCU
 *
 * Each exposed hardware timer is represented as an instance of this struct.
 */
typedef struct utim_periph {
    const utim_t dev;                   /**< Timer device identifier */
    const utim_driver_t *const driver;  /**< Timer class driver */

    const uint16_t width          :8;   /**< Counter register width in bits */
    const uint16_t channels       :4;   /**< Number of available compare channels */
    const uint16_t int_ovf        :1;   /**< Overflow interrupt availability */
    const uint16_t int_cmp_match  :1;   /**< Compare match interrupt availability */
} utim_periph_t;
```

Listing 10: Definition of the `utim_periph_t` struct. It maps drivers to exposed hardware timers and provides information about static timer properties.

```c
/**
 * @brief   Counter value type
 *
 * The selected type must be wide enough to hold the maximum counter width
 * among all used timers on a MCU.
 */
#ifdef CONFIG_UTIMER_OVERWRITE_MAX_WIDTH
    #ifdef CONFIG_UTIMER_OVERWRITE_MAX_WIDTH_8
        typedef uint_fast8_t utim_cnt_t;
    #endif

    #ifdef CONFIG_UTIMER_OVERWRITE_MAX_WIDTH_16
        typedef uint_fast16_t utim_cnt_t;
    #endif

    #ifdef CONFIG_UTIMER_OVERWRITE_MAX_WIDTH_24
        typedef uint_fast32_t utim_cnt_t;
    #endif

    #ifdef CONFIG_UTIMER_OVERWRITE_MAX_WIDTH_32
        typedef uint_fast32_t utim_cnt_t;
    #endif

    #ifdef CONFIG_UTIMER_OVERWRITE_MAX_WIDTH_64
        typedef uint_fast64_t utim_cnt_t;
    #endif
#else
    #ifndef HAVE_UTIMER_COUNT_T
        typedef uint_fast32_t utim_cnt_t;
    #endif
#endif
```

Listing 11: Default definition of the counter value type `utim_cnt_t`. It usually is automatically inferred from the selected timer peripherals within the board specific configuration, as indicated by `HAVE_UTIMER_COUNT_T`. If not, a default value is used. It can also be overwritten by the optional Kconfig setting `UTIMER_OVERWRITE_MAX_WIDTH`.

```c
/**
 * @brief   Type for timer clock sources
 *
 * All available timer clocks are identified by a systemwide unique id
 */
#ifndef HAVE_UTIMER_CLK_T
typedef enum {
    UTIM_CLK_DEFAULT,   /**< Default clock source for the peripheral.
                             Unspecified or don't care. */
} utim_clk_t;
#endif

/**
 * @brief   Operation mode of a timer peripheral
 */
typedef enum {
    UTIM_DISABLED    = 'D', /**< Timer is disabled / not counting */
    UTIM_CONTINUOUS  = 'C', /**< Timer is counting continuously (autoreload) */
    UTIM_ONESHOT     = 'O'  /**< Timer is counting in one-shot mode */
} utim_mode_t;

/**
 * @brief   Operation mode of a timer compare channel
 */
typedef enum {
    UTIM_CHAN_DISABLED       = 'D', /**< Timer channel is disabled */
    UTIM_CHAN_ONESHOT        = 'O', /**< Timer channel is armed in one-shot mode */
    UTIM_CHAN_PERIODIC       = 'P', /**< Timer channel is armed in periodic mode */
    UTIM_CHAN_PERIODIC_RESET = 'R'  /**< Timer channel is armed in periodic mode.
                                         Counter value is automatically reset after
                                         each compare match */
} utim_chan_mode_t;

/**
 * @brief   Counting direction of a timer
 */
typedef enum {
    UTIM_CNT_DIR_UP      = 'U', /**< Timer is counting up */
    UTIM_CNT_DIR_DOWN    = 'D', /**< Timer is counting down */
    UTIM_CNT_DIR_UPDOWN  = 'C'  /**< Timer counts up to max value,
                                     than counts down to min value. */
} utim_cnt_dir_t;
```

Listing 12: Definition of enumeration types for timer properties: `utim_clk_t`, `utim_mode_t`, `utim_chan_mode_t`, `utim_cnt_dir_t`. These types are used in conjunction with the respective timer property defined in Listing 13.

```
/**
 * @brief   Type for timer property values
 */
typedef unsigned int utim_propval_t;

/** Prefix indicating a channel property in utim_prop_t */
#define UTIM_PROP_CHAN_PFX      0x8000
/** Channel number bits in utim_prop_t */
#define UTIM_PROP_CHAN_NUM_Pos  10
#define UTIM_PROP_CHAN_NUM_Msk  (0b11111 << UTIM_PROP_CHAN_NUM_Pos)
/** Channel property indicator bits in utim_prop_t */
#define UTIM_PROP_CHAN_PROP_Pos 0
#define UTIM_PROP_CHAN_PROP_Msk (0b1111111111 << UTIM_PROP_CHAN_PROP_Pos)

/**
 * @brief   Properties and flags a timer peripheral can have
 */
typedef enum {
    UTIM_PROP_RESERVED          = 0x00, /**< Reserved for future use */

    /* Basic timer peripheral properties */
    UTIM_PROP_MODE              = 0x01, /**< Timer counting mode (utim_mode_t) */
    UTIM_PROP_CNT_DIR           = 0x02, /**< Counting direction (utim_cnt_dir_t) */
    UTIM_PROP_INT_CMP_MATCH     = 0x03, /**< IRQ generation on compare match */
    UTIM_PROP_INT_OVF           = 0x04, /**< IRQ generation on counter overflow */

    /* Status and event properties */
    UTIM_PROP_OVF_PENDING       = 0xF0, /**< Overflow flag is set */
    UTIM_PROP_CMP_MATCH_PENDING = 0xF1, /**< At least one compare match flag set */

    // 8< -----

    /* Compare channel specific properties */
    /**
     * MSB indicates that this is a channel property. The next 5 bits hold the
     * channel number, followed by n bits that determine the addressed channel
     * property.
     *
     * Example:
     *  -Prefix-    -Channel 3-      -Mode-
     * ((0x8000) | (0x03 << 10) | (0x00 << 0))
     */
    UTIM_PROP_CHAN0_MODE                = ((UTIM_PROP_CHAN_PFX) |
        (0x00 << UTIM_PROP_CHAN_NUM_Pos) | (0x00 << UTIM_PROP_CHAN_PROP_Pos)),
    UTIM_PROP_CHAN0_CMP_MATCH_PENDING   = ((UTIM_PROP_CHAN_PFX) |
        (0x00 << UTIM_PROP_CHAN_NUM_Pos) | (0x01 << UTIM_PROP_CHAN_PROP_Pos)),
    // 8< -----
    UTIM_PROP_CHAN7_MODE                = ((UTIM_PROP_CHAN_PFX) |
        (0x07 << UTIM_PROP_CHAN_NUM_Pos) | (0x00 << UTIM_PROP_CHAN_PROP_Pos)),
    UTIM_PROP_CHAN7_CMP_MATCH_PENDING   = ((UTIM_PROP_CHAN_PFX) |
        (0x07 << UTIM_PROP_CHAN_NUM_Pos) | (0x01 << UTIM_PROP_CHAN_PROP_Pos)),
} utim_prop_t;
```

Listing 13: Definition of the hAPI property interface types and their encoding. Each property possesses a unique identifier within `utim_prop_t`. Its value depends on the respective property but is limited by the `utim_propval_t` type.

```c
/**
 * @brief   User-defined compare match interrupt callback function
 *
 * @param[in] arg       optional context for the callback
 * @param[in] channel   timer channel that triggered the interrupt.
 *                      -1 if channel is unknown.
 */
typedef void (*utim_cmp_cb_t)(void *arg, int channel);

/**
 * @brief   User-defined overflow interrupt callback function
 *
 * @param[in] arg   optional context for the callback
 */
typedef void (*utim_ovf_cb_t)(void *arg);

/**
 * @brief   Default interrupt context entry holding compare match callback
 *          and argument.
 */
typedef struct {
    utim_cmp_cb_t cb;   /**< callback executed at compare match interrupt */
    void *arg;          /**< optional argument given to the callback */
} utim_cmp_isr_ctx_t;

/**
 * @brief   Default interrupt context entry holding overflow callback and
 *          argument
 */
typedef struct {
    utim_ovf_cb_t cb;   /**< callback executed at overflow interrupt */
    void *arg;          /**< optional argument given to the callback */
} utim_ovf_isr_ctx_t;
```

Listing 14: Definition of timer callback function signatures and ISR contexts. Callbacks of the types `utim_cmp_cb_t` and `utim_ovf_cb_t` are combined with their optional arguments into the ISR context types `utim_cmp_isr_ctx_t` and `utim_ovf_isr_ctx_t`.

B.2 Hardware-facing API

Data types and functions of the hAPI are listed in this section. Listing 15 gives an overview of all functions that are available. It is followed by detailed definitions of all its functions.

```
/**
 * @brief   Driver type that groups driver functions for a timer type
 *
 * One such driver must be provided for every timer type. It is mapped within
 * the respective utim_periph_t structs.
 */
typedef struct utim_driver {
    int (*const init)(/* timer, freq, clk, cmp_cb, cmp_args, ovf_cb, ovf_args */);
    utim_propval_t (*const get_property)(/* timer, prop */);
    int (*const set_property)(/* timer, prop, value */);
    int (*const enable)(/* timer, run */);
    utim_cnt_t (*const read)(/* timer */);
    int (*const write)(/* timer, count */);
    int (*const set_channel)(/* timer, channel, mode, count */);
    #ifndef CONFIG_UTIMER_USE_REDUCED_API
    int (*const is_valid_freq)(/* timer, clk, freq */);
    #endif
} utim_driver_t;
```

Listing 15: Definition of the `utim_driver_t` struct. One such struct exists for every timer type and contains pointers to its respectively implemented hAPI functions. All functions, including their arguments, are described in detail in the remainder of this section.

```
/**
 * @brief   Initializes the timer module
 *
 * @param[in] tim       The timer device to operate on
 * @param[in] freq      Requested frequency (number of timer ticks per second)
 * @param[in] clk       Clock source to select for the timer
 * @param[in] cmp_cb    Callback to be executed on compare match interrupt
 * @param[in] cmp_arg   Optional arguments passed to CMP callback function
 * @param[in] ovf_cb    Callback to be executed on overflow interrupt
 * @param[in] ovf_arg   Optional arguments passed to OVF callback function
 *
 * @retval  0 on success
 * @retval -1 if frequency not applicable
 * @retval -2 if clock source invalid
 * @retval -3 if OVF interrupt requested but not supported by the timer
 */
int (*const init)(
    utim_periph_t *const tim,
    unsigned long freq,
    utim_clk_t clk,
    utim_cmp_cb_t cmp_cb,
    void *cmp_cb_arg,
    utim_ovf_cb_t ovf_cb,
    void *ovf_cb_arg
);
```

Listing 16: Definition of the hAPI function: `init()`

```
/**
 * @brief   Retrieves information about a specific timer property
 *
 * @param[in] tim   The timer device to operate on
 * @param[in] prop  Desired property
 *
 * @return  Value of the property. Undefined if property is not supported
 *          by the timer.
 */
utim_propval_t (*const get_property)(
    utim_periph_t *const tim,
    utim_prop_t prop
);
```

Listing 17: Definition of the hAPI function: `get_property()`

```
/**
 * @brief   Modifies a specific timer property
 *
 * @param[in] tim   The timer device to operate on
 * @param[in] prop  Desired property
 * @param[in] val   Value to set the property to
 *
 * @retval  0 on success
 * @retval -1 if property update failed
 * @retval -2 if property is unsupported or read-only
 */
int (*const set_property)(
    utim_periph_t *const tim,
    utim_prop_t prop,
    utim_propval_t val
);
```

Listing 18: Definition of the hAPI function: `set_property()`

```
/**
 * @brief   Enables or disables the timer, thus starting or stopping
 *          incrementing the internal counter of the hardware peripheral.
 *
 * @param[in] tim   The timer device to operate on
 * @param[in] run   if true, the timer is started
 *                  if false, the timer is stopped
 *
 * @retval  0 on success
 * @retval -1 if timer device failed to start or stop
 */
int (*const enable)(
    utim_periph_t *const tim,
    bool run
);
```

Listing 19: Definition of the hAPI function: `enable()`

```
/**
 * @brief   Reads the current raw counter value of the internal counter
 *          register.
 *
 * @param[in] tim   The timer device to operate on
 *
 * @return Current raw value of the internal counter register
 */
utim_cnt_t (*const read)(
    utim_periph_t *const tim
);
```

Listing 20: Definition of the hAPI function: `read()`

```
/**
 * @brief   Sets the current raw counter value of the internal counter
 *          register.
 *
 * @param[in] tim   The timer device to operate on
 * @param[in] cnt   Counter value to write
 *
 * @retval  0 on success
 * @retval -1 if counter write failed or unsupported
 */
int (*const write)(
    utim_periph_t *const tim,
    utim_cnt_t cnt
);
```

Listing 21: Definition of the hAPI function: `write()`

```
/**
 * @brief   Configures a timer channel to either generate a timeout event
 *          exactly at the given counter value or disables it.
 *
 * @param[in] tim       The timer device to operate on
 * @param[in] channel   Timer channel to (dis-)arm
 * @param[in] mode      Desired channel operation mode (e.g. disabled,
 *                      one-shot, periodic, ...).
 * @param[in] cnt       Absolute counter value to arm for
 *
 * @retval  0 on success
 * @retval -1 if channel invalid
 * @retval -2 if cnt value invalid (e.g. higher than max counter value)
 * @retval -3 if mode invalid or not supported by timer
 */
int (*const set_channel)(
    utim_periph_t *const tim,
    unsigned int channel,
    utim_chan_mode_t mode,
    utim_cnt_t cnt
);
```

Listing 22: Definition of the hAPI function: `set_channel()`

```
#ifndef CONFIG_UTIMER_USE_REDUCED_API
/**
 * @brief   Indicates if the timer can run at the given frequency
 *
 * @param[in] tim   The timer device to operate on
 * @param[in] clk   Clock source to decide frequency validity for
 * @param[in] freq  Frequency (number of ticks per second) to check
 *
 * @retval  0 if frequency is valid for the timer
 * @retval -1 if frequency not applicable for the timer
 */
int (*const is_valid_freq)(
    utim_periph_t *const tim,
    utim_clk_t clk,
    unsigned long freq
);
#endif
```

Listing 23: Definition of the optional hAPI function: `is_valid_freq()`

B.3 User-facing API

All public functions of the uAPI are listed in this section. They are split into five categories: B.3.1) Peripheral Management, B.3.2) Basic Timer Functions, B.3.3) Compare Channels and Timeouts, B.3.4) Dynamic Property Interface, and B.3.5) Helper Functions.

B.3.1 Peripheral Management

```
/**
 * @brief   Provides the timer instance struct for a given timer
 *
 * @param[in] dev   Timer device identifier
 *
 * @return  On success: Corresponding timer peripheral struct
 * @return  If timer device not found: Timer peripheral struct with
 *          .dev = UTIMER_DEV_INVALID and .driver = NULL
 */
utim_periph_t utimer_get_periph(utim_t dev);
```

Listing 24: Definition of the uAPI function: `utimer_get_periph()`

```
/**
 * @brief   Initializes the timer module.
 *
 * @param[in] tim       The timer device to operate on
 * @param[in] freq      Requested frequency (number of timer ticks per second)
 * @param[in] clk       Clock source to select for the timer
 * @param[in] cmp_cb    Callback to be executed on compare match interrupt
 * @param[in] cmp_arg   Optional arguments passed to CMP callback function
 * @param[in] ovf_cb    Callback to be executed on overflow interrupt
 * @param[in] ovf_arg   Optional arguments passed to OVF callback function
 *
 * @retval  0 on success
 * @retval -1 if frequency not applicable
 * @retval -2 if clock source invalid
 * @retval -3 if OVF interrupt requested but not supported by the timer
 */
int utimer_init(
    utim_periph_t *const tim,
    unsigned long freq,
    utim_clk_t clk,
    utim_cmp_cb_t cmp_cb,
    void *cmp_cb_arg,
    utim_ovf_cb_t ovf_cb,
    void *ovf_cb_arg
);
```

Listing 25: Definition of the uAPI function: `utimer_init()`

B.3.2 Basic Timer Functions

```
/**
 * @brief   Enables the timer, i.e., starting to increment the internal
 *          counter of the hardware peripheral.
 *
 * @param[in] tim   The timer device to operate on
 *
 * @retval  0 on success
 * @retval -1 if timer device failed to start
 */
int utimer_start(utim_periph_t *const tim);
```

Listing 26: Definition of the uAPI function: `utimer_start()`

```
/**
 * @brief   Disables the timer, i.e., stopping the internal counter
 *          of the hardware peripheral.
 *
 * @param[in] tim   The timer device to operate on
 *
 * @retval  0 on success
 * @retval -1 if timer failed to stop
 */
int utimer_stop(utim_periph_t *const tim);
```

Listing 27: Definition of the uAPI function: `utimer_stop()`

```
/**
 * @brief   Reads the current raw counter value of the internal counter
 *          register.
 *
 * @param[in] tim   The timer device to operate on
 *
 * @return Current raw value of the internal counter register
 */
utim_cnt_t utimer_read(utim_periph_t *const tim);
```

Listing 28: Definition of the uAPI function: `utimer_read()`

```
/**
 * @brief   Sets the current raw counter value of the internal counter
 *          register.
 *
 * @param[in] tim   The timer device to operate on
 * @param[in] cnt   Counter value to write
 *
 * @retval  0 on success
 * @retval -1 if counter write failed or unsupported
 */
int utimer_write(utim_periph_t *const tim, utim_cnt_t cnt);
```

Listing 29: Definition of the uAPI function: `utimer_write()`

B.3.3 Compare Channels and Timeouts

```
/**
 * @brief   Arms a timer channel to generate a timeout event after the given
 *          time.
 *
 * The specified timeout value is relative to the current internal counter
 * value of the timer.
 *
 * @param[in] tim       The timer device to operate on
 * @param[in] channel   Desired timer channel to arm
 * @param[in] timeout   Relative timeout value (number of ticks) to set
 *
 * @retval  0 on success
 * @retval -1 if channel invalid
 * @retval -2 if timeout invalid
 */
int utimer_set(
    utim_periph_t *const tim,
    unsigned int channel,
    unsigned int timeout
);
```

Listing 30: Definition of the uAPI function: `utimer_set()`

```
/**
 * @brief   Arms a timer channel to generate a timeout event exactly at the
 *          given counter value.
 *
 * @param[in] tim       The timer device to operate on
 * @param[in] channel   Desired timer channel to arm
 * @param[in] cnt       Absolute counter value to arm for
 *
 * @retval  0 on success
 * @retval -1 if channel invalid
 * @retval -2 if cnt invalid (e.g. higher than max counter value)
 */
int utimer_set_absolute(
    utim_periph_t *const tim,
    unsigned int channel,
    utim_cnt_t cnt
);
```

Listing 31: Definition of the uAPI function: `utimer_set_absolute()`

```
/**
 * @brief   Arms a timer channel to periodically generate timeout events
 *          exactly at the given counter value.
 *
 * @param[in] tim       The timer device to operate on
 * @param[in] channel   Desired timer channel to arm
 * @param[in] cnt       Absolute counter value to arm for
 * @param[in] flags     Flags to control counter reset behavior
 *
 * @retval  0 on success
 * @retval -1 if channel invalid
 * @retval -2 if cnt invalid (e.g. higher than max counter value)
 */
int utimer_set_periodic(
    utim_periph_t *const tim,
    unsigned int channel,
    utim_cnt_t cnt,
    uint8_t flags
);
```

Listing 32: Definition of the uAPI function: `utimer_set_periodic()`

```
/**
 * @brief   Disarms a timer channel.
 *
 * The specified timer channel is disarmed regardless of its current state.
 *
 * @param[in] tim       The timer device to operate on
 * @param[in] channel   Desired timer channel to disarm
 *
 * @retval  0 on success
 * @retval -1 if channel invalid
 */
int utimer_clear(
    utim_periph_t *const tim,
    unsigned int channel
);
```

Listing 33: Definition of the uAPI function: `utimer_clear()`

B.3.4 Dynamic Property Interface

```
/**
 * @brief   Retrieves information about a specific timer property
 *
 * @param[in] tim   The timer device to operate on
 * @param[in] prop  Desired property
 *
 * @return  Value of the property. Undefined if property is not supported
 *          by the timer.
 */
utim_propval_t utimer_get_property(
    utim_periph_t *const tim,
    utim_prop_t prop
);
```

Listing 34: Definition of the uAPI function: `utimer_get_property()`

```
/**
 * @brief   Modifies a specific timer property
 *
 * @param[in] tim   The timer device to operate on
 * @param[in] prop  Desired property
 * @param[in] val   Value to set the property to
 *
 * @retval  0 on success
 * @retval -1 if property update failed
 * @retval -2 if property is unsupported or read-only
 */
int utimer_set_property(
    utim_periph_t *const tim,
    utim_prop_t prop,
    utim_propval_t val
);
```

Listing 35: Definition of the uAPI function: `utimer_set_property()`

B.3.5 Convenience Functions

```
#ifndef CONFIG_UTIMER_USE_REDUCED_API
/**
 * @brief   Indicates if the timer can run at the given frequency
 *
 * @param[in] tim   The timer device to operate on
 * @param[in] clk   Clock source to decide frequency validity for
 * @param[in] freq  Frequency (number of ticks per second) to check
 *
 * @retval  0 if frequency is applicable for the given timer
 * @retval -1 if frequency not applicable
 */
int utimer_is_valid_freq(
    utim_periph_t *const tim,
    utim_clk_t clk,
    unsigned long freq
);
#endif
```

Listing 36: Definition of the optional uAPI function: `utimer_is_valid_freq()`

```
#ifndef CONFIG_UTIMER_USE_REDUCED_API
/**
 * @brief   Calculates the nearest possible frequency the timer is
 *          capable of running at.
 *
 * @param[in] tim   The timer device to operate on
 * @param[in] clk   Clock source to decide frequency validity for
 * @param[in] freq  Target frequency (number of ticks per second)
 *
 * @return Nearest achievable frequency with respect to the
 *         specified clock
 */
unsigned long utimer_get_nearest_freq(
    utim_periph_t *const tim,
    utim_clk_t clk,
    unsigned long freq
);
#endif
```

Listing 37: Definition of the optional uAPI function: `utimer_get_nearest_freq()`

Glossary

API Well-defined software interface between user applications and system components. APIs specify a set of functions and provide their implementations. *Peripheral APIs* are a subset of these, used to access the hardware peripherals of an MCU. See Section 2.2.

Benchmark A set of test routines that assess the performance of software. Quantitative results allow for comparison between different implementations.

Hardware Abstraction Layer Software layer between device hardware and application firmware or other OS modules. See Section 2.2.

Hardware in the Loop Testing Technique where a programmable hardware controller is used to stimulate a device under test. See Section 7.1.1.

Hardware-facing API Part of the uTimer API. A minimal device-specific set of functions. It contains all low-level driver code for a timer type. See Section 5.2.3.

MCU Platform A set of microcontrollers that share most of their properties, such as CPU and peripheral types. Also referred to as *device family*. See Section 2.1.

Microcontroller Unit Integrated circuit (IC) that combines a CPU, memory, and on-board peripherals within a single chip package. See Section 2.1.

periph_timer The current generic low-level timer API of RIOT. See Section 4.2.

PHiLIP The Primitive Hardware in the Loop Integration Product (PHiLIP) [58] is a low-cost solution for HiL testing. See Section 7.1.2.

Regression Testing The process of re-running software tests to ensure that imminent changes does not cause the software to behave incorrectly.

RIOT An open-source embedded OS for resource constrained devices in the IoT and WSNs. See Section 2.3.

Robot Framework The Robot Framework (RF) [49] is an open-source test automation framework. See Section 7.1.3.

Timer Microcontroller peripheral that is used for timing and counting. Runs asynchronously to the main program flow, i.e., independent of the CPU. Also referred to as *hardware timer* or *low-level timer*. See Section 2.1.

Timer Type Specific class or version of hardware timers, such as *general-purpose timer*, *low-power timer*, or *real-time clock*. All peripherals that belong to one timer type can be controlled by the same low-level driver code..

Unit Testing Testing a specific section of code for proper behavior.

User-facing API Part of the uTimer API. A uniform set of functions that is independent of underlying timer types. See Section 5.2.4.

uTimer The novel low-level timer API we contribute with this work. The corresponding RIOT OS module is internally referred to as `periph_utimer`. See Chapter 5.

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original