2021 IEEE International Conferences on Internet of Things (iThings) and IEEE Green Computing & Communications (GreenCom) and IEEE Cyber, Physical & Social Computing (CPSCom) and IEEE Smart Data (SmartData) and IEEE Congress

uTimer: A Uniform Low-level Timer API for RIOT OS

Niels Gandraß
HAW Hamburg
Germany
Niels.Gandrass@haw-hamburg.de

Michel Rottleuthner
HAW Hamburg
Germany
Michel.Rottleuthner@haw-hamburg.de

Thomas C. Schmidt
HAW Hamburg
Germany
T.Schmidt@haw-hamburg.de

***Abstract*—Microcontrollers offer a large range of hardware timers. As peripherals grow in diversity, the complexity of supporting them increases for embedded operating systems. Well-established software interfaces struggle with covering the full feature set of novel timers. This raises the need for a flexible hardware abstraction of timer peripherals.**

In this paper, we first contribute an analysis of timer hardware that covers 43 device families from eight manufacturers and a survey of existing driver modules. We then derive uTimer, a uniform low-level timer API for the RIOT operating system. Its interface is independent of the timer type and allows for transparent use of all available peripherals. This fosters application portability across MCUs. We develop an automated benchmark suite that quantifies the abstraction overhead and compare the API performance to current solutions. Results show that uTimer introduces only six additional CPU cycles and preserves the timer performance.

***Index Terms*—embedded systems, hardware abstraction, hardware timers, operating systems, resource constrained devices**

1. Introduction

Microcontroller units (MCUs) offer a constantly growing variety of hardware timers. Especially resource constrained embedded devices can benefit from supporting those novel peripherals. With the emerging Internet of Things (IoT), devices, applications, and deployment contexts of embedded controllers increase in number and heterogeneity, and so does the need for robust hardware abstractions that fosters portability. Choosing an appropriate level of abstraction is complex, but mandatory to balance todays requirements on performance and time to market. Embedded operating systems (OSs) are the prevalent solution for developing sustainable applications in the IoT. One increasingly popular embedded OS is RIOT [2]. This open-source OS targets low-power and resource constrained embedded devices.

RIOT offers five distinct low-level timer modules, all differing in use and feature availability. With this work, we propose a low-level timer interface that unifies current APIs and hereby streamlines timer usage throughout the RIOT ecosystem. We start by defining the problem of hardware abstraction, followed by a survey of existing timer interfaces and related work in Section 2. We conduct a large-scale analysis of timer peripherals on which we ground our low-level timer API design within Section 3. After that, we develop benchmarks to quantify its performance and isolate the entailed abstraction overhead in Section 4. We conclude with an outlook on future work in Section 5.

2. The Problem of Hardware Abstraction

Timers are a fundamental part of every embedded system and required by most applications. As a result, their efficient yet convenient use is essential. 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*. Mitigation of performance loss is partly possible by sacrificing system memory to reduce computational complexity. Design decisions regarding this *time-memory trade-off* are discussed by us in Section 3.2.3. The problem of abstracting timer hardware therefore lies in balancing both trade-offs to foster convenient timer use while confining performance degradation.

2.1. Low-level Timer Interfaces

RIOT offers three generic and two special purpose low-level timer drivers. The generic modules (`periph_timer`/`rtt`/`rtc`) interface common timer types, such as general-purpose and low-power timers or real-time clocks (RTCs). The special purpose drivers (`periph_pwm`/`wdg`) do not expose timers directly, but instead use the underlying hardware to provide higher-level features, such as signal generation. Separating 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 drivers are limited to the small set of features that is common to all MCU-platforms, leaving additional timer features unsupported. Though functionality

978-1-6654-1762-4/21/$31.00 ©2021 IEEE

DOI 10.1109/iThings-GreenCom-CPSCom-SmartData-Cybermatcis53846.2021.00017

of most modules overlaps, we found their APIs to differ in use and feature availability. This includes, for example, counter value representation, interrupt (INT) 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 through multiple APIs, leading to resource allocation conflicts and undefined behavior. Special timer types, such as high-speed counters or ultra low-power timers, frequently remain unsupported. Three modules statically map to a single hardware timer and channel, hereby leaving a significant number of available peripherals and compare channels unusable. Registration of user-defined callback functions, to be executed upon timer overflow interrupts, is only supported by one module. 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 MCUs. Considering the above, the application developer is often required to manually (re-)write low-level driver code whenever additional timer features or specific timer peripherals are needed.

2.2. Related Work

Operation principles and common features of hardware timers are outlined by Kamal [5] as well as Susnea and Mitescu [8]. The latter conducted a comparison of timer peripherals from three MCU families. They found that even though specific timer features differ, assessed peripherals still share many common operation principles. Further analyses of MCU hardware exist, but they do not cover timer peripherals at the required level of detail to derive interface design decisions from them.

The current timer subsystem and hardware abstraction layers (HALs) in RIOT are described by Baccelli *et al.* [2]. 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.* [6] separates register access from peripheral driver code. This makes abstract device drivers independent of the underlying hardware, enabling the authors to mock MCU hardware during unit tests. Handziski *et al.* [3] present a multi-layered 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 loosing application portability.

The *Common Microcontroller Software Interface Standard (CMSIS)* [1] is a set of uniform APIs for Cortex-based devices. It exposes core components, including the *SysTick* timer. Additional timer peripherals cannot be accessed, since these are not part of the generic Cortex processor family. A platform-independent timer API is proposed by Lindgren *et al.* [7]. They discuss similarities between timers of the STM32F4 and LPC1700 MCU families and outline their impact on the proposed API. Identified generic characteristics include counter width, interrupt capability, prescalers, auto-reload, and compare channel count. We likewise cover these characteristics in our larger-scale hardware analysis. The authors furthermore emphasize, that having many compare channels available benefits timer maintenance.

Comparative benchmarks of the current RIOT high-level timer software modules are performed by Ismail [4]. The author uses PHiLIP [9] 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 in Section 4.

3. A Unified Timer API for RIOT

3.1. Timer Hardware Analysis

Prior to designing the timer API, we conducted a large-scale analysis of timer peripherals. It includes 43 device families of 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 further, whenever our explorative data acquisition revealed additional points of interest. We examine 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 is transformed into uniform result tables, referred to as *Timer Comparison Matrices* (TCMs). These allow detailed comparison of timer peripherals across all assessed device families. The TCMs are the 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 [8] 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, hereby requiring a more complex interface. 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 timer hardware.

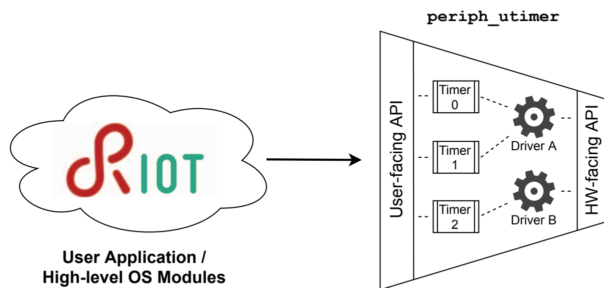

Figure 1. Architecture of the proposed low-level timer API design

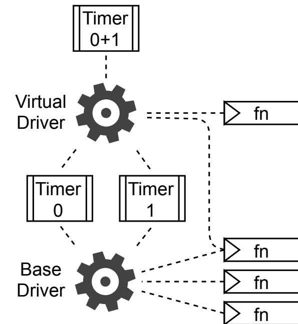

Figure 2. Virtual timer driver

3.2. A Uniform Low-level Timer API

We address the outlined challenges with uTimer, a low-level timer API for RIOT. It streamlines existing APIs and exposes hardware timers via a uniform interface, fostering a transparent and 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.

Our design is split into a hardware-facing API (hAPI) and a user-facing API (uAPI), decoupling timer logic from hardware-dependent driver code. It incorporates key aspects of the hardware abstraction architecture proposed by Handziski *et al.* [3]. The hAPI consists of timer type specific drivers that interact directly with the peripherals. The uAPI then uses the hAPI to provide a convenient hardware-agnostic interface to both applications and higher-level OS modules, such as the network stack or sensor drivers. It hereby encourages uniformity of timer code throughout the whole RIOT ecosystem. An overview of this architecture is illustrated in Figure 1.

With uTimer, each hardware timer is represented by a designated timer instance struct. It identifies the peripheral device, provides static timer properties, and specifies the hAPI driver to use. Selection and configuration of timers that are made available to the application is done via *Kconfig* at compile-time. Exposed peripherals can interactively be configured and required drivers are automatically included.

3.2.1. Hardware-facing API. The hAPI provides low-level drivers that directly interact with timer hardware registers and possess a lightweight yet flexible interface. One such driver exists for every timer type (e.g., *general-purpose*, *low-power*, or *RTC*) that is used by at least one exposed timer instance. Drivers are implemented as structs that consist of minimal function sets, each represented as a group of function pointers. Common basic features are directly accessible through designated functions, whereas device-specific features are exposed via a compact and flexible property interface, supporting optional feature availability. Closely related operations are bundled into single functions whenever appropriate, e.g., combining `start()` and `stop()` into a single `enable(bool)` call. Our hardware analysis revealed, that some timer types are available in multiple versions, hence share many common features. Their implementations can therefore be mapped to multiple drivers, allowing selective re-use of single functions as well as entire drivers. Representing chained timers as a single instance is furthermore made possible by combining peripherals using virtual drivers, as illustrated in Figure 2.

3.2.2. User-facing API. The uAPI provides one single set of functions that is independent of the underlying timer type. Previously bundled hAPI functions are unbundled for convenient use and function calls are either directly delegated to the respective driver, for example, read and write operations, or performed as compound operations of multiple subsequent hAPI driver calls such as relative timer arming. Static attributes such as counter width or channel count, and run-time dynamic properties such as counting mode or pending interrupts, are made available. Additional convenience functions, such as determining available timer frequencies, are furthermore provided. Support of device-specific features can dynamically be determined during run-time and is indicated via function return values. Separate user-defined callbacks can be attached to both compare match and overflow interrupts, whenever supported by the respective timer. Interrupts can be (un-)masked at any time, and all mandatory maintenance tasks are automatically performed by the corresponding hAPI driver. Clock sources can be run-time selected as either a generic clock class, such as *high-frequency*, *low-power* and *default*, or explicitly as a platform-specific clock.

3.2.3. Design Trade-offs. Compare match (CMP) and overflow (OVF) callbacks are separated due to their disjoint use cases. CMP is not split further, since only 31 % of all timer types provide distinct interrupts for every compare channel. Even though applicable platforms may benefit from a slightly lower timeout latency, the majority only suffer the additional pointer memory overhead. Individual functions can nonetheless still be dispatched within the user-defined callback, whenever required.

Mapping timer functions within hAPI driver structs and assigning these to timer instances introduces pointer dereferencing overhead. Functions could instead be mapped

directly within each timer instance struct, hereby removing one layer of indirection. This, however, raises the memory footprint and induces code duplication. Closely related is our design decision to bundle certain hAPI functions and later unbundle them within the uAPI to reduce driver struct size. We assess both trade-offs in detail by isolating and quantifying the exact abstraction overhead in Section 4.2.

4. Evaluation

We implemented uTimer on four MCUs that feature a representative and diverse range of timer peripherals: (I) STM32L476RG (Nucleo-L476RG), a powerful Cortex-M4 device featuring six timer types and a total of 15 timer peripherals; (II) STM32F070RB (Nucleo-F070RB), a mainstream Cortex-M0 MCU that has only basic 16-bit timers; (III) EFM32PG12B500 (SLSTK3402A), a low-power ARM chip with four chainable 16/32-bit timers and two types of ultra low-power timer peripherals; (IV) ESP32 (ESP32-WROOM), a Xtensa MCU that possesses four 64-bit timers and is a popular choice for generic IoT applications.

Automating software tests and benchmarks is especially important when dealing with a diverse and ever-changing hardware landscape. RIOT therefore actively supports automated cross-platform testing by combining Hardware in the Loop (HiL) testbeds with Continuous Integration (CI) [9]. Our evaluation setup integrates with this solution. It provides developers with ready-to-use unit tests and benchmarks, fostering development of performant API implementations.

4.1. Cross-platform Validation

To validate our implementations, all existing RIOT low-level timer test suites were ported to `periph_utimer`. These 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 newly supported features, such as counter register writes and overflow interrupt handling. To validate advanced features, further device-specific tests were developed. Virtual timer drivers for example were tested by exposing a pair of chained hardware timers as a single timer instance. Both types of tests revealed small implementation errors, which then could be fixed prior to our benchmarks. Test suites are platform-independent and a CI integration for automated execution is provided.

4.2. Performance Benchmarks

We quantified the performance of uTimer (`periph_utimer`) and compared it to the existing `periph_timer` API on all four MCUs. Our test setup uses the Primitive Hardware in the Loop Integration Product (PHiLIP) [9] to conduct all measurements in a hardware-independent fashion. The test system architecture and our benchmarking procedure is illustrated in Figure 3.

Our benchmarks consist of a RIOT-based test firmware and a Robot Framework (RF) test suite, similar to the

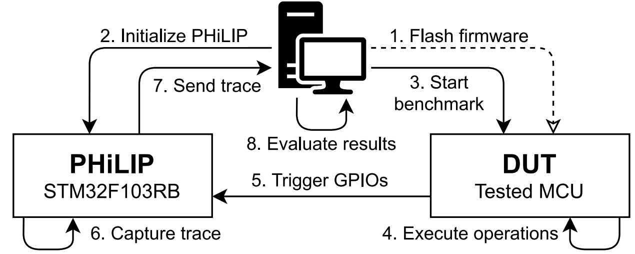

Figure 3. Architecture of our benchmarking setup

high-level benchmarks conducted by Ismail [4]. Prior to execution, the test firmware is flashed onto the device under test (DUT) once. It implements all benchmark routines and is controlled through a simple shell interface that is exposed via UART. The corresponding RF test suite then is started. It sends control commands to the DUT shell and uses PHiLIP to record GPIO event traces. Measurements are evaluated by processing the captured traces within the RF test suite. This setup seamlessly integrates with the existing HiL and CI infrastructure and can easily be extended to additional MCUs, as API implementations become available.

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. Each test case starts by resetting the DUT, flushing the trace buffer, and configuring DUT interrupt requests (IRQs) followed by 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 executed function calls 10 times per benchmark pass to safely exceed the required hold-off time.

4.2.1. Accuracy and Hardware Limits. Measurement start and stop is indicated by consecutive rising and falling edges on an output pin. Its accuracy therefore depends on the GPIO overhead of the DUT and the input capture performance of PHiLIP. The first is accounted for during trace evaluation, as described and quantified in Section 4.2.2. The latter depends on the selected trigger mode. We use PHiLIPs interrupt-based dual-edge trigger mode for all our benchmarks. It is rated to require at least $1\,\mu s$ hold-off time between two consecutive edges and faces $200\,ns$ timestamp jitter [9, 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 results with only $600\,ns$ hold-off time and $45\,ns$ jitter. Undercutting this limit leads to loss of samples and erroneous measurements. A hold-off time of at least $1\,\mu s$ is therefore enforced throughout all our benchmarks to provide a reasonable safety margin and preserve board compatibility.

The resolution of captured traces is limited by PHiLIPs fixed CPU operation frequency of $72\,MHz$, resulting in $14\,ns$ quantization steps that can be observed in measurements. PHiLIPs circular trace buffer can hold a maximum of 128

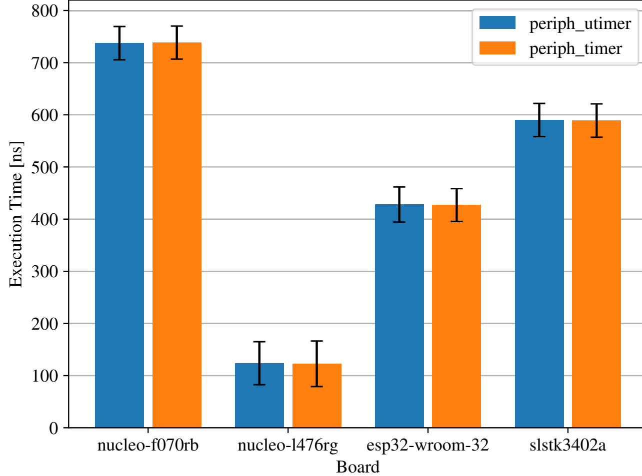

Figure 4. Average GPIO overheads including their standard deviations

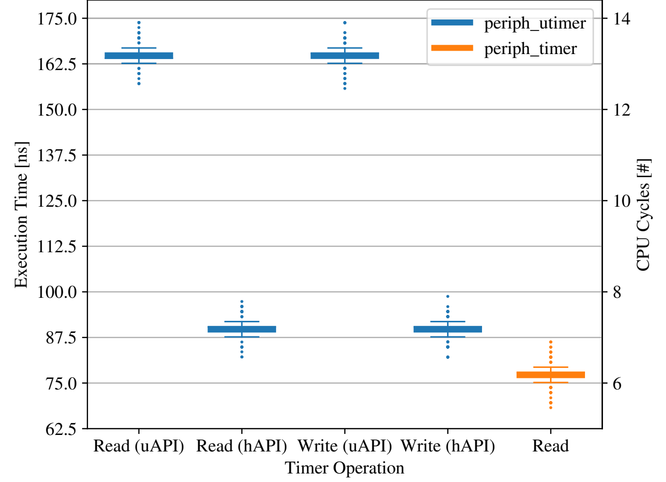

Figure 5. Read and write NOPs execution time on Nucleo-L476RG board running at 80 MHz core clock speed

events at once. All our test suites therefore fetch captured traces after every 50 consecutive benchmark passes.

4.2.2. 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 measurement start and stop by these events, knowing the board specific GPIO overhead O_{GPIO} is essential for achieving accurate results. We measure it according to pseudocode Listing 1.

```
1 Setup
2 Repeat 500 times:
3    gpio_set(GPIO_IC);
4    spin(SPIN_DURATION);
5    gpio_clear(GPIO_IC);
6    spin(PHILIP_HOLDOFF_TIME);
7 Teardown
```

Listing 1. GPIO overhead benchmark pseudocode

$$\Delta t = t_{\texttt{gpio_clear()}} - t_{\texttt{gpio_set()}} \quad (1a)$$

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

Active waiting (i.e., spinning) is used to represent a set of benchmarked operations. The `spin()` function is inlined to avoid function call overhead. Subtracting its calibrated execution time from the recorded trace yields the overhead of our measurement setup (1). We found it stable for spin durations between 1 µs and 1 ms, while longer spins were increasingly affected by clock drift and CPU pipelining artifacts. Observed stable values range from 123 ns ± 42 ns on the Nucleo-L476RG up to 737 ns ± 32 ns on the Nucleo-F070RB, as depicted in Figure 4. Our measurements also confirmed O_{GPIO} to be independent of the used timer API.

4.2.3. Timer Base Operations Benchmark. We measure basic timer operations such as read or write according to pseudocode Listing 2. As these operations can complete quickly, each function is invoked 10 times per benchmark run to ensure that the PHiLIP 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 (2). Execution durations t_{op} are converted 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()`.

```
1  Setup
2  Repeat 500 times:
3    START Measurement
4      timer_operation();
5      timer_operation();
6      // ... repeated 10 times
7      timer_operation();
8    STOP Measurement
9    spin(PHILIP_HOLDOFF_TIME);
10 Teardown
```

Listing 2. Timer base operation benchmark pseudocode

$$\Delta t = t_{\texttt{STOP}} - t_{\texttt{START}} \quad (2a)$$

$$t_{op} = \frac{1}{N}\sum_{i=1}^{N}\left[\frac{1}{10}\left(\Delta t_i - O_{GPIO}\right)\right] \quad (2b)$$

Execution times are composed of both MCU specific code and the abstraction overhead that is inherent to our API design. To isolate the latter, we replaced all timer read and write operations with no operations (NOPs) and measured the remaining execution time. As the current `periph_timer` API does not support write operations, only its read function was assessed.

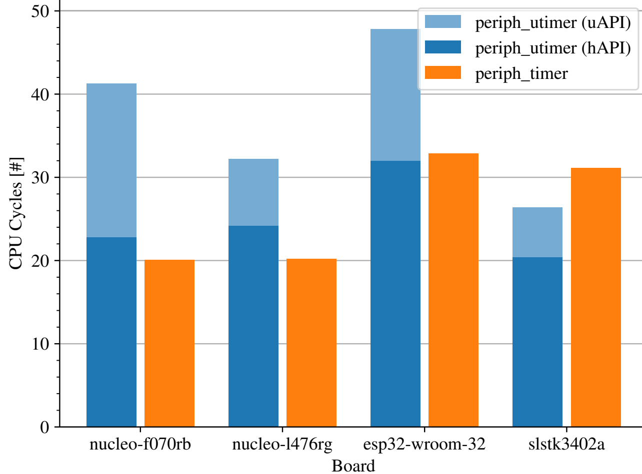

Figure 6. CPU cycles consumed by read operations

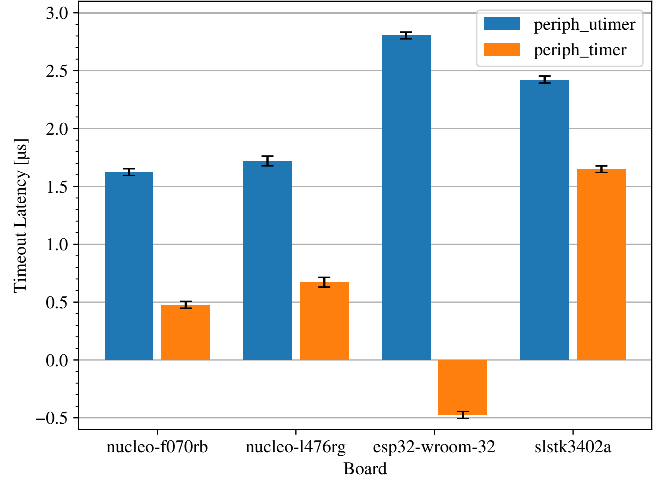

Figure 7. Latency and jitter of 1 ms timeouts at 1 MHz timer frequency

Results show that the applied abstraction causes an overhead of six clock cycles, as depicted in Figure 5. On ARM-based devices, however, one additional clock cycle is introduced within the hAPI. `periph_timer` uses a plain integer to address peripherals on STM32 microcontrollers. Compiling with `gcc` and size optimization (`-Os`) results in a `MOVS` instruction prior to the branch (`BL`). With `periph_utimer`, a pointer to the respective timer instance struct is passed instead. This results in an `ADD` instruction followed by a branch and an additional instruction set change (`BLX`). Though these instructions require the same amount of CPU cycles, both sets are subjected to pipeline refills. One such pipeline refill takes between one and three CPU cycles, hereby causing the observed hAPI overhead. The ESP32 with its Xtensa CPU architecture suffered no such increase.

Besides required branching instructions, resolving the corresponding hAPI driver function was found to be another source of overhead. We further analyzed the disassembly to identify potential optimizations for the latter. Despite the timer instance and its driver being constant and the executed operation never changed, 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. We therefore consider the six clock cycle overhead inevitable with the used compiler toolchains. However, once quantified, this overhead can be compensated by the application.

After assessing the sole API overhead, we ran the benchmark suite for read, write, set, and clear operations with their full implementations. Figure 6 compares the timer read performance for all boards. On the Nucleo and ESP32 boards, `periph_utimer` shows the quantified abstraction overhead. Board specific implementation differences are furthermore reflected by our results. On the SLSTK3402A, however, uTimer performs better than the existing implementation by five CPU cycles. Directly using the hAPI instead of the uAPI even results in a total speed-up of 11 CPU cycles. As `periph_timer` exposes three different timer types on EFM32 CPUs, it is required to determine the corresponding implementation within every API function. This constitutes the observed overhead. The uTimer API mitigates this by natively supporting multiple timer types.

CPU frequencies are usually at least ten times higher than timer base frequencies. All observed performance differences, both positive and negative, therefore become negligibly small for many use cases. With 1 MHz timers on the Nucleo-L476RG, for example, the CPU performs 80 cycles per timer tick, hereby virtually eliminating the effects of the additionally required CPU cycles regarding timer behavior.

4.2.4. Timeout Latency Benchmark. The latency of a timeout L_{tout} is the difference between the expected and the actually observed end of a timeout period. Negative values for L_{tout} may occur due to oscillator inaccuracies or faulty driver code. Its standard deviation is referred to as timeout jitter J_{tout}. Timeout error E_{tout} puts L_{tout} in relation to the expected timeout duration $tout$. Keeping it within application appropriate limits is crucial. We measure it according to pseudocode Listing 3. First, a timer is initialized and stopped. It then is armed to a counter value corresponding to the desired timeout duration at the selected frequency. Relative arming is required due to the missing support for writing the counter register via the existing `periph_timer` API. Elapsed time Δt between the subsequent timer start and the awaited callback execution then is measured (3a). Results of a complete benchmark pass are calculated according to (3b-d).

$$\Delta t = t_{\text{STOP}} - t_{\text{START}} \tag{3a}$$

$$L_{tout} = \frac{1}{N} \sum_{i=1}^{N} [\Delta t_i - tout - O_{GPIO}] \tag{3b}$$

$$J_{tout} = \sigma(L_{tout}) = \sqrt{\sigma(\Delta t)^2 + \sigma(L_{GPIO})^2} \tag{3c}$$

$$E_{tout} = \text{abs}\left(\frac{L_{tout}}{tout}\right) \tag{3d}$$

```
1  Setup
2  Repeat 500 times:
3    timer_init(frequency, callback);
4    timer_stop();
5    cnt = timer_read() + timeout;
6    timer_set(cnt);
7    timer_start();
8    START Measurement
9      WAIT for callback execution
10   STOP Measurement
11   spin(PHILIP_HOLDOFF_TIME);
12 Teardown
```

Listing 3. Timeout latency benchmark pseudocode

To cover a comprehensive range of use cases, assessed timer frequencies vary between 10 kHz and 10 MHz while timeout durations vary between 10 µs and 1 s. Compared to `periph_timer`, `periph_utimer` suffered a slight increase in timeout latency while jitter remained stable, as depicted in Figure 7. Our benchmarks found L_{tout} to increase by between 0.77 µs on the SLSTK3402A (best) and 2.32 µs on the ESP32 board (worst). This behavior shows the inevitable drawbacks of abstraction and was observed throughout all tested duration and frequency combinations alike. It is caused by a combination of resolving the timer type dependent low-level interrupt service routine (ISR) upon IRQ occurrence and the now supported timer overflow handling[1]. Whether this increase is tolerable depends on both the timeout duration and the application requirements. Table 1 compares the performance of both APIs for common 1 ms timeouts. Here, timeout error increased by between 0.08 % on the SLSTK3402A (best) and 0.23 % on the ESP32 board (worst). We consider this minor increase to be insignificant for most applications. It, however, becomes an issue with very short timeouts ($\leq$ 10 µs), as even just a slight increase in latency contributes significantly to the overall timeout error. In such cases, unnecessary indirection should always be avoided. Directly using the hAPI or active waiting (i.e., spinning) therefore is recommended. Most applications, however, do not require this very short timeouts. A slightly increased latency therefore is negligible when put into perspective to the timeout durations that are common with typical IoT use case scenarios. Moreover, with long-running timeouts ($\geq$ 1 s), other factors, such as oscillator accuracy, become dominant.

Our benchmarks revealed a significant increase in timeout latency for certain timeout and frequency combinations when using `periph_timer`. On the Nucleo-F070RB, for example, generating a 1 s timeout using a timer running at 10 kHz base frequency resulted in a 11.5 µs timeout latency. 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 increase in timeout error to 98.30 %. Since behavior for invalid parameters is not explicitly specified by the API, implementations usually neither verify the requested frequency nor timeout durations. An established convention is to select the closest achievable timer frequency or compare channel value. This leads to unpredictable timeout lengths if parameters are not chosen carefully by the developer, as individually required for each MCU and its current clock configuration. Both erroneous cases result in the observed timeout latency increase. The `periph_utimer` API addresses this problem by requiring implementations to signal an error if the requested frequency is not exactly achievable or the desired timeout is out of range. A uAPI method to determine the closest achievable frequency is furthermore provided and counter limits can be determined by the static properties within every timer instance struct.

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

TABLE 1. TIMEOUT LATENCY BENCHMARK RESULTS FOR 1 ms TIMEOUTS AT 1 MHz TIMER FREQUENCY

Board	`periph_timer` $L_{tout} \pm J_{tout}$	`periph_utimer` $L_{tout} \pm J_{tout}$
nucleo-f070rb	0.48 µs ± 43.28 ns	1.62 µs ± 43.27 ns
nucleo-l476rg	0.67 µs ± 60.55 ns	1.72 µs ± 58.91 ns
esp32-wroom-32	−0.48 µs ± 43.94 ns	2.80 µs ± 44.52 ns
slstk3402a	1.65 µs ± 42.70 ns	2.42 µs ± 43.87 ns

4.3. Additional Aspects

4.3.1. Memory Consumption. The additional abstraction layers and time-memory optimizations both contribute to the overall memory footprint. With uTimer, every exposed timer instance entails one `utim_periph_t` struct. It consists of two pointers and one 16-bit field, requiring a total of 12 bytes[2] on 32-bit devices. 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 activated timer, as dynamically configured via Kconfig. Besides the generic API footprint, differences in the platform-dependent driver code may further affect memory consumption. Using only a single unified API, in turn, can free up memory that previously was consumed by the multiple specialized APIs. The final data read-only memory (ROM) size therefore strongly depends on both the individual application and the target platform. Random-access memory (RAM) use remained constant throughout all our experiments.

4.3.2. Peripheral Availability. Making all timers of a board available to the application is one of our aspired goals. Table 2 compares the total number of exposed peripherals and channels between a combined use of existing APIs (`periph_timer`/`rtt`/`rtc`) and exclusively using uTimer. It shows that our novel API makes all available timers and channels usable. Besides their sole quantity, the types of available timers are equally important. On STM32 boards, existing APIs only expose one *Low-power Timer*, leaving additional peripherals unusable. *Advanced-control Timers*

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

TABLE 2. EXPOSURE OF TIMER PERIPHERALS AND CHANNELS: COMBINED USE OF EXISTING APIS COMPARED TO UTIMER

Board	Timers (Channels)		
	Available	Existing APIs	uTimer
nucleo-f070rb	10 (14)	2 (5)	10 (14)
nucleo-l476rg	15 (32)	3 (6)	15 (32)
esp32-wroom-32	5 (5)	4 (4)	4 (4)
slstk3402a	8 (18)	4 (9)	8 (18)

require additional configuration and *Basic Timers* remain fully unusable. On the ESP32, all available timers are exposed, except one general-purpose timer that is reserved for the OS. Lastly, none of the existing APIs allow the use of EFM32 ultra low-power *Cryotimer* peripherals or *Pulse Counters*. This common lack of support for advanced timer types especially limits low-power optimizations. Using uTimer instead, developers are able to select the best suited timers for their specific application among the full range of available peripherals.

4.3.3. Code Quality and Usability. Besides performance of an API stands its usability and maintainability. Existing interfaces still offer room for optimization regarding these aspects. As previously indicated, `periph_timer` on EFM32 devices simultaneously exposes three different timer types, promoting to combine all implementations within a single function. This not only lowers performance, but also decreases code quality and hides information about the peripheral type from the user. Some implementations 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.

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*). Static and run-time dynamic timer properties can safely and conveniently be accessed. User applications, drivers, and other OS modules thereby can directly use this information without error-prone workarounds. Ready to use peripheral mappings and their flexible selection furthermore relieve application developers from modifying OS code and deep-diving into vendor datasheets or SDKs. Existing APIs are streamlined into a single interface, fostering uniformity of timer code throughout applications. API behavior upon error or invalid use is explicitly defined and a common pattern for peripheral configuration is established. During our evaluation, we found all these measures to significantly benefit both usability and code maintainability.

5. Conclusion and Outlook

The challenge in designing a hardware API lies in balancing the abstraction and time-memory trade-offs. We assessed this systematically for RIOT with a survey of existing low-level timer interfaces and an extensive analysis of timer peripherals. Based on both, we proposed uTimer, a uniform low-level timer API that exposes all available hardware timers and fosters application portability. It provides both common platform-independent and advanced platform-specific timer features. The abstraction overhead that is inherent to our two-layered design was isolated and quantified to a total of six CPU cycles. We evaluated the proposed API with HiL-based benchmarks on four distinct MCUs, covering basic timer operations and timeout specific metrics. Compared to current solutions, relative error of typical $1\,\mathrm{ms}$ timeouts merely increased by between $0.08\,\%$ (best) and $0.23\,\%$ (worst). All negative and positive performance differences were found to be negligible for most applications, while usability and code quality benefited significantly.

We will continue our research by extending uTimer implementations to additional RIOT devices. Unit tests and performance benchmarks will be integrated into the existing HiL and CI environment. This work shall lay the ground for higher-level advancements of the RIOT timer subsystem such as software-counter width extensions and class-based multiplexing of virtual timeouts.

References

[1] Arm Ltd., *Common Microcontroller Software Interface Standard (CMSIS)*, Oct. 2021.

[2] 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.

[3] 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*, 2005, pages 145–157.

[4] A. Ismail, “Automated Testing of the RIOT-OS Timer Subsystem,” Dec. 2020.

[5] R. Kamal, *Embedded Systems: Architecture, Programming and Design*, second edition. Tata McGraw Hill Education, 2011.

[6] V. B. Kleeberger, S. Rutkowski, and R. Coppens, “Design & Verification of Automotive SoC Firmware,” in *DAC ’15 Proceedings*, ACM, Jun. 2015.

[7] 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, Mar. 2016.

[8] I. Susnea and M. Mitescu, *Microcontrollers in Practice (Springer Series in Advanced Microelectronics)*, first edition. Berlin, Heidelberg: Springer-Verlag, 2005.

[9] 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.