Work-in-Progress: Large-scale Timer Hardware Analysis for a Flexible Low-level Timer-API Design

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

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

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

ABSTRACT

We report on our ongoing development of an optimized low-level timer-API for the RIOT operating system. Starting with a survey of hardware timer peripherals from 43 MCU-families and 8 manufacturers, we identify common properties and differences of all available timer types. Based on this hardware analysis, we propose a lightweight yet powerful low-level timer-API design. It streamlines existing timer interfaces and relieves application developers from the error-prone task of repeatedly writing additional peripheral driver code.

KEYWORDS

embedded systems, hardware abstraction, operating systems

ACM Reference Format:
Niels Gandraß, Michel Rottleuthner, and Thomas C. Schmidt. 2021. Work-in-Progress: Large-scale Timer Hardware Analysis for a Flexible Low-level Timer-API Design. In *2021 International Conference on Embedded Software Companion (EMSOFT'21 Companion), October 8–15, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3477244.3477617

1 INTRODUCTION

Timer peripherals are essential to all embedded devices [3]. Manufacturers of microcontroller units (MCUs) today offer a large variety of timer modules ranging from general-purpose to highly specialized components. With the emerging *Internet of Things* (IoT), devices, applications, and deployment contexts of embedded controllers increase in numbers and heterogeneity, and so does the need for sound hardware abstractions that foster portability. Embedded operating systems (OSs) are the prevalent solution for developing sustainable applications in the IoT. One increasingly popular embedded OS is RIOT [1]. This open-source OS explicitly 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 want to design a new low-level timer interface that unifies current APIs and hereby streamlines timer usage throughout the whole RIOT ecosystem. We start with a large-scale analysis of timer peripherals in Section 2, and sketch a low-level timer-API, which improves on the existing driver implementations in Section 3. We conclude with an outlook on our future work.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
EMSOFT'21 Companion, October 8–15, 2021, Virtual Event, USA
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8712-5/21/10…$15.00
https://doi.org/10.1145/3477244.3477617

2 HARDWARE-PLATFORM ANALYSIS

We conducted a survey covering 43 device families of eight manufacturers – those are all MCUs currently supported by RIOT-OS. For every device family, we characterize all hardware timer types by examining datasheets, reference manuals, and SDKs. Assessed aspects include basic properties such as counter register width, prescaler configuration, compare match capabilities, and auto-reload functionality [3, pp. 152-159] as well as more advanced aspects such as interrupt generation, timer chaining, and low-power features. Since our data acquisition was explorative and not limited to only the previously defined properties, it yielded additional results. These were used to extend our set of assessed aspects before we transformed gathered information into uniform result tables, referred to as *Timer Comparison Matrices (TCMs)*. These allow detailed comparison of timer peripherals across MCU families. The TCMs were then used to derive inter-MCU-platform findings from.

Our analysis identified counter register widths of timers to be at least 16 bit among all platforms while 90 % also provide 32-bit and 21 % even 64-bit timers. In addition, especially small timers often support timer chaining, as found on 71 % of all platforms. Frequency prescalers are always available, and many timers provide at least two (64 %) or even four (24 %) channels. However, only 31 % feature fully independent interrupt vectors for every compare channel. We observed that 95 % of all timers can be driven by at least one internal and 92 % by at least one external clock source. 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 also offer at least one timer that is capable of both operating in the lowest device power states and waking the CPU upon event occurrence. Feature sets of general-purpose timers were confirmed to be virtually uniform across all platforms, whereas other timer types differed largely in their offered functions and modes of operation, as also found by Susnea and Mitescu [4, pp. 67-91]. A uniform timer-API therefore can allow platform-independent use of basic common features while also being capable of exposing platform-specific timer functions.

3 TOWARDS A LOW-LEVEL TIMER-API

RIOT-OS Timer Modules. Currently, three generic and two special purpose low-level timer abstractions exist in RIOT-OS. Their functions are limited to a small set of features that is common to all MCU-platforms, leaving many timer features unexposed. Though functionality of most modules overlap, their APIs differ in use and feature availability. Advanced timer types are often left unexposed,

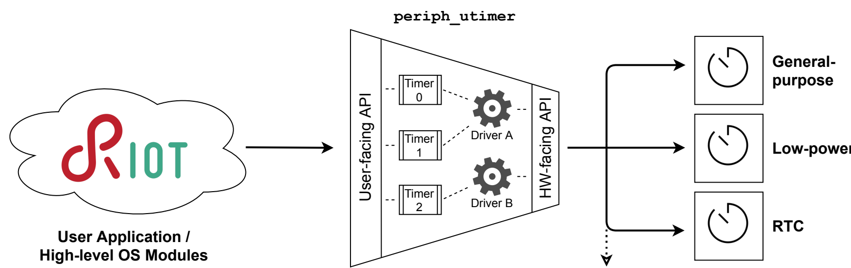

Figure 1: Architecture of the proposed low-level timer-API design

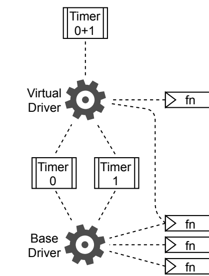

Figure 2: A virtual timer driver

and many modules statically map to a single hardware timer. This leaves a significant number of available peripherals and compare channels unusable. Registration of callback functions, to be executed upon timer overflow interrupts, is only supported by one module. Timer configuration management is highly platform dependent and differs largely across MCUs. Considering the above, the application developer often is required to manually (re-)write low-level driver code whenever advanced features, low-power operation, or specific timer peripherals are needed.

Low-level Timer-API Design. To address current shortcomings we propose a unified low-level timer-API, incorporating results of our hardware analysis and a review of related work. It exposes hardware timers homogeneous and platform-independent, thereby fostering a transparent and interchangeable use of "all" available timer peripherals. Access to advanced timer features is kept lightweight and platform-independence is preserved whenever possible.

Our design is split into a hardware-facing API (hAPI) and a user-facing API (uAPI), incorporating key aspects of the three-stage hardware abstraction architecture proposed by Handziski et al. [2]. The hAPI consists of hardware-dependent drivers for each timer type, interacting directly with the timer peripherals. The uAPI then uses the hAPI to provide a convenient hardware-agnostic interface to applications and high-level OS modules, hereby encouraging uniformity of timer code throughout the whole RIOT ecosystem. An overview of this architecture is illustrated in Figure 1.

Each hardware timer is represented by a struct that is referenced when interacting with the peripheral. For every exposed timer, as configured during compile-time through *Kconfig*, an instance of this struct exists. It identifies the peripheral device, provides static timer information (e.g., width and channel count), and specifies the hAPI driver to use. One such driver exists for every timer type (e.g., general-purpose, low-power, RTC) that is used by at least one exposed timer instance. Drivers consist of minimal function sets, each represented as a group of function pointers. Common basic features are directly accessible through designated functions, whereas more advanced 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(run)` call. Driver functions can be mapped freely and may be shared between multiple drivers, allowing re-use of single functions as well as entire drivers. Representing chained timers as a single instance is furthermore possible by combining peripherals using virtual drivers, as illustrated in Figure 2.

In contrast, the uAPI only provides one single set of functions, usable independent of the underlying timer type. Previously bundled hAPI functions are unbundled for convenient use and function calls are either delegated to the respective driver directly (e.g., read- and write operations) or performed as compound operations of multiple subsequent hAPI driver calls (e.g., relative timer channel arming). Static attributes and run-time dynamic properties are made available and additional convenience functions are provided. Feature support can dynamically be determined by the application during run-time and is also indicated via function return values. User-defined callbacks can be attached to both compare match and overflow interrupts, whenever supported by the respective timer. All required interrupt maintenance tasks are automatically performed by the corresponding driver. Clock sources can be run-time selected as either a generic clock class, such as *high-frequency*, *low-power* or *default*, or explicitly as a platform-specific clock.

4 CONCLUSION AND OUTLOOK

We have outlined our path towards a unified low-level timer-API for RIOT that exposes all hardware timers and fosters application portability. It provides both common platform-independent and advanced platform-specific timer features. Next on our agenda is a systematical evaluation and verification of the proposed design. We first will extend our implementation prototype to a relevant and diverse set of devices from different manufacturers. Automated function tests across all platforms shall reveal potential shortcomings and problems. Subsequently, benchmarks will be conducted to compare the performance of the proposed API against current implementations. Quantifying the impact of each of the different abstractions that are part of our novel API-design is crucial. Aspects to assess shall include timeout latency, jitter, and memory footprint. After a final optimization phase, we plan to implement the API for a large set of RIOT devices.

REFERENCES

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

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

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

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