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

MINF-PJH

Niels Gandraß <Niels.Gandrass@haw-hamburg.de>

September 02, 2021

Hamburg University of Applied Sciences
Faculty of Engineering & Computer Science

Table of Contents

1. Introduction
2. The Problem of Abstracting Timer Hardware
3. A Unified Timer-API for RIOT-OS
4. Validation and Evaluation
5. Conclusion

Introduction

Introduction

- Timers are fundamental parts of every embedded system
- MCU manufacturers offer a wide range of peripherals, including:
    - General-purpose timers
    - Low-power timers
    - High-speed timers
    - Real-time clocks (RTCs)
- Each timer type possesses its own feature-set

The Challenge

Embedded OSs need to keep up with the ever-growing variety of timers. Offering broad out-of-the-box peripheral support while maintaining application portability is challenging.

The Problem of Abstracting Timer Hardware

The Problem of Abstracting Timer Hardware

The Abstraction Trade-off vs.

Direct HW-register access	Strongly abstracted timer-API
Yields Near-optimal performance, but is highly error-prone, laborious, and prevents portability.	Is portable and user-friendly, but decreased performance.

The Time-memory Trade-off vs.

Mitigation of performance loss is partly possible by sacrificing system memory to reduce computational complexity

Choosing an appropriate level of abstraction therefore is challenging.

Current Low-level Timer-APIs in RIOT-OS

Three generic modules interface common timer types and two special-purpose modules provide higher-level features, such as signal generation.

- Only basic timer operations supported
- API functionality overlaps, but they differ in use and exposed features
- Underlying timer types differ between MCUs
- Peripherals can simultaneously be used by multiple APIs (resource allocation conflicts)
- Timer selection and configuration via platform-dependent headers files

Modules

`periph/`

- `timer` ()
- `rtc` ()
- `rtt` ()
- `pwm` ()
- `wdt` ()

STM32L476RG Timer Support in RIOT-OS

RIOT-OS Modules

`periph/`

- `timer` ()
- `rtc` ()
- `rtt` ()
- `pwm` ()
- `wdt` ()

STM32L476RG Peripherals

- General-purpose timer (1/7) (2/7)
  - 32- and 16-bit
- Basic timer (0/2)
- Advanced-control timer (1/2)
- Low-power timer (1/2)
- Real-time-clock (1/1)
- SysTick timer (0/1)
- Watchdog (1/2)

Peripheral Availability

- Only 35% of the available timers are actually usable
- 2 timer types are not exposed by any `periph` module

Application developers should not (re-)write low-level driver code!

A Unified Timer-API for RIOT-OS

Our Goal

Streamline existing APIs into a uniform interface, fostering a transparent and interchangeable use of all available timer peripherals. Provide basic timer functions and out-of-the-box support for device-specific feature, while preserving platform-independence whenever possible.

To base our API design on, we conducted ...

- Large-scale analysis of timer hardware
  - Covering 43 device families from 8 different manufacturers
- Review of existing low-level timer modules
- Survey of related work

Key Design Aspects

- Separation of hardware-facing (hAPI) and user-facing API (uAPI)
- One low-level driver `utim_driver_t` per timer type
- Exposed timers represented by a `utim_periph_t` instance, referencing corresponding `utim_driver_t` and providing static timer properties
- Interactive timer selection and configuration via KConfig

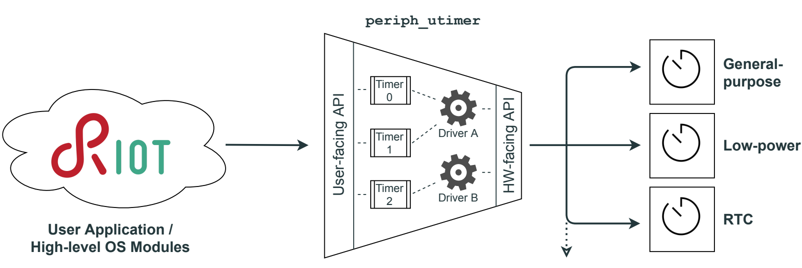

Figure 1: Architecture of the proposed low-level timer-API design

Key Design Aspects – Hardware-facing API

One low-level driver struct `utim_driver_t` per timer type

- Consisting of minimal function pointer sets
- Common basic features are directly accessible and device-specific features are exposed via a compact property interface
- Related functions are bundled into single calls
- Driver granular reusability across timer types
- Function granular reusability across drivers
- Virtual drivers allow representation of chained timers as one atomic timer instance, re-using existing driver code

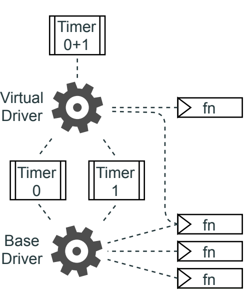

Figure 2: Virtual timer driver

Key Design Aspects – User-facing API

A single set of functions, independent of the underlying timer type

- Function calls are either directly delegated to the driver or implemented as multiple subsequent hAPI driver calls
- Previously bundled hAPI functions are unbundled
- Static attributes and run-time dynamic properties are made available
- Compare match and overflow interrupts can be handled separately
- Clock source is run-time configurable

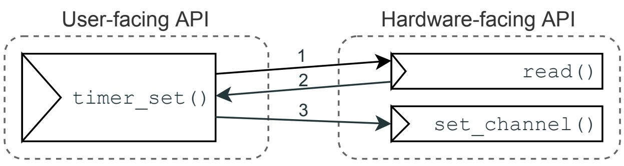

Figure 3: Compound uAPI function consisting of multiple hAPI driver calls

Validation and Evaluation

Validation and Evaluation

1. Cross-platform Validation via automated platform-independent test suites with CI integration.

2. Performance Benchmarks using a HiL testbed wit CI integration, comparing the existing `periph_timer` to our novel `periph_utimer`.

Scope

- STMicroelectronics: STM32L476RG (Nucleo-L476RG)
- STMicroelectronics: STM32F070RB (Nucleo-F070RB)
- Silicon Labs: EFM32PG12B500 (SLSTK3402A)
- Espressif Systems: ESP32 (ESP32-WROOM)

Selected MCUs cover different manufacturers, CPU architectures, counter widths from 16 to 64 bit, common basic timers, advanced ultra low-power peripherals and chainable timers.

Performance Benchmarks – Setup Architecture

Benchmarks consist of a RIOT-based test firmware and a Robot Framework (RF) test suite.

GPIO traces are captured during benchmarks. Measurement start and stop is signaled by consecutive rising and falling edges. Hardware limits like GPIO latency and hold-off times are accounted for.

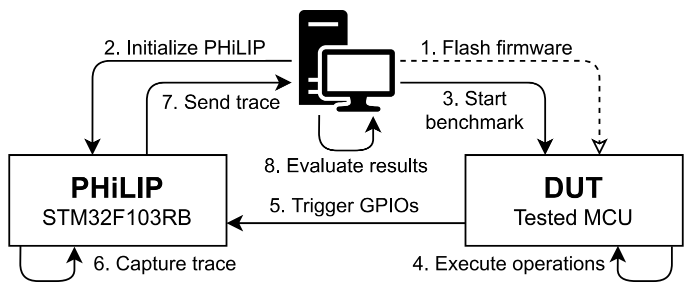

Figure 4: Architecture of our benchmarking setup

Performance Benchmarks – Conducted Benchmarks

The following aspects were assessed by our benchmarks:

1. PHiLIP hardware limits
2. GPIO Latency
3. API abstraction overhead
   - User-facing API
   - Hardware-facing API
   - No additional abstraction
4. Timer base operations
   - Read counter register
   - Write counter register
   - Set channel
   - Clear channel
5. Timeout latency

Performance Benchmarks – API Abstraction Overhead

Isolating the APIs abstraction overhead:

- Read and write operations replaced with no operations (NOPs)
- Operations performed via both uAPI and direct hAPI driver calls
- Measured execution time converted to equivalent CPU cycles

API Abstraction Overhead

- Abstraction via uAPI introduces 6 CPU cycles
- No generic hAPI overhead
  - One additional CPU cycle on STM32 due to pipeline refill artifacts.

Performance Benchmarks – Timeout Latency

Timeout latency, jitter and error were assessed:

- Timer frequencies between 10 kHz and 10 MHz
- Timeout durations between 10 μs and 1 s
- Duration between arming and callback execution is measured
- Difference of expected and actual timeout length is calculated

Timeout Latency and Error 1 ms @ 1 MHz

Timeout latency L_{tout} increased by between 0.76 μs $\pm$ 0.02 μs on the SLSTK3402A (best) and 2.34 μs $\pm$ 0.01 μs on the ESP32 (worst).

The respective timeout error E_{tout} therefore increased by between 0.0751 % (best) and 0.2338 % (worst).

Performance Benchmarks – Timeout Latency

⚠ Edge case: Very short timeouts ($\leq 10\,\mu s$)

- Every slight increase in timeout latency L_{tout} significantly contributes to the timeout error E_{tout}
- In such cases, unnecessary indirection should be avoided
- Direct hAPI use or active waiting (i.e. spinning) is recommended

---

✔ Edge case: Long-running timeouts ($\geq 1\,s$)

- Impact of timeout latency L_{tout} increase on timeout error E_{tout} becomes insignificantly small
- Other factors, such as oscillator accuracy, become dominant

Additional Aspects

During our evaluation we further found:

- **ROM size** on 32-bit devices increased by 12 bytes per configured timer peripheral and 28 bytes for every required timer type driver.
- **RAM use** was not affected by uTimer.
- Number of out-of-the-box **available peripherals and channels** significantly increased and advanced timer types are supported.
- **Code quality and usability** benefited from the streamlined API.

Conclusion

Conclusion

The proposed uTimer API streamlines existing RIOT-OS modules into a uniform interface.

- Abstraction and time-memory trade-offs were successfully balanced, allowing convenient use while maintaining performance.
- Both platform-independent and platform-specific timer features are exposed, preserving portability whenever possible.
- Application developers are relieved from modifying OS code and deep diving into vendor datasheets or SDKs.

Questions ?

Discussion !