Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Tackling the RIOT-OS Low-level Timer

A Progress Report about Ongoing Research

INET Seminar / MINF-PJG

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

August 6th, 2020

Hamburg University of Applied Sciences
Faculty of Engineering & Computer Science

Table of Contents

1. Introduction and Motivation

2. Goals and Conceptual Approach

3. API Design Ideas

4. Advantages and Problems

5. Outlook and Future Work

Introduction and Motivation

Motivation

Starting Point **1/3**

There currently are 5 different low-level timer modules in RIOT-OS:

`periph/`

• `timer` ()	General-purpose timer driver
• `rtc` ()	Real-time clock driver
• `rtt` ()	Real-time timer driver
• `pwm` ()	PWM peripheral driver
• `wdt` ()	Watchdog timer driver

Motivation

Starting Point **2/3**

All driver modules only offer a minimalistic set of generic features

- Provided API derived from set of functions that can commonly be found, even on small low-end MCUs
- Additional features, available on more advanced MCU platforms, are therefore not implemented and often left unused
    - Potentially leaving much driver code development to the user
    - Preventing generic optimizations, based on feature-availability, in higher-level modules (e.g. `xtimer`). For example: Dynamically switching between software emulation of a feature and utilization of a hardware implementation if available

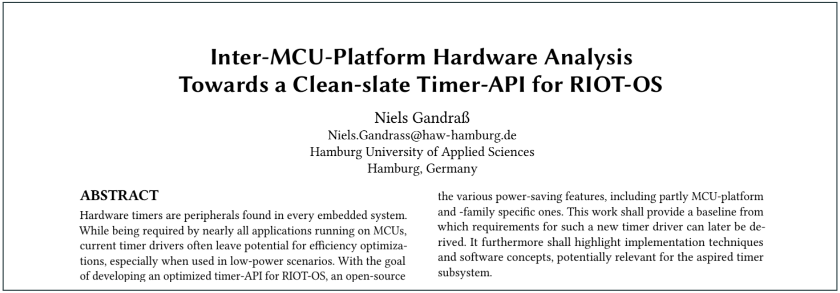
Motivation

Starting Point **3/3**

Peripheral configuration management, if available, is implemented very heterogeneous across different MCU platforms

- Current timer drivers leave configuration management decisions to the platform developer
  - No uniform separation of configuration layers: board-specific, compile-time static, runtime
- Sometimes only a fraction of the available peripherals is exposed, while others are left completely unused
- Static mapping of hardware timers inside the drivers can lead to peripheral allocation conflicts

Introduction



Inter-MCU-Platform Hardware Analysis Towards a Clean-slate Timer-API for RIOT-OS

Niels Gandraß
Niels.Gandrass@haw-hamburg.de
Hamburg University of Applied Sciences
Hamburg, Germany

ABSTRACT

Hardware timers are peripherals found in every embedded system. While being required by nearly all applications running on MCUs, current timer drivers often leave potential for efficiency optimizations, especially when used in low-power scenarios. With the goal of developing an optimized timer-API for RIOT-OS, an open-source

the various power-saving features, including partly MCU-platform and -family specific ones. This work shall provide a baseline from which requirements for such a new timer driver can later be derived. It furthermore shall highlight implementation techniques and software concepts, potentially relevant for the aspired timer subsystem.

Figure 1: Previously conducted timer hardware analysis

Introduction

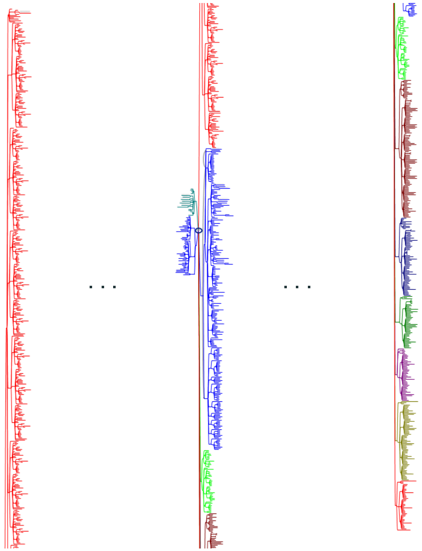


Figure 2: Mind-map containing data for half of the platforms

Introduction

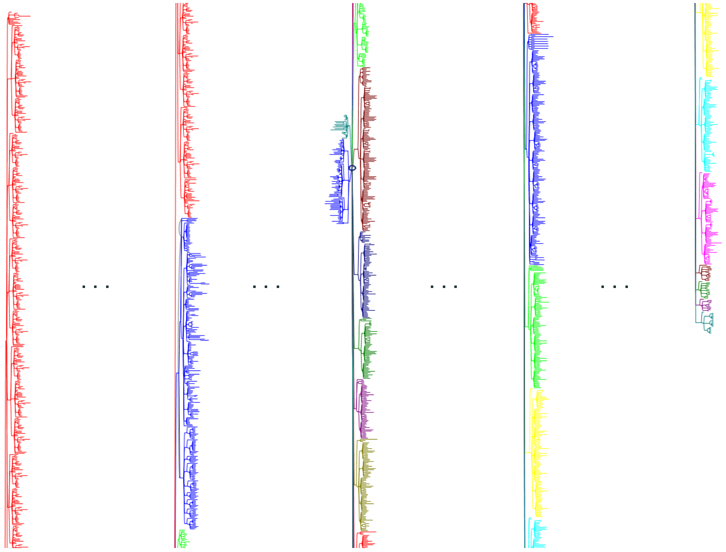


Figure 3: Updated Mind-map containing data for most of the platforms

Goals and Conceptual Approach

Goals

Primary Goals

- Usage of available timer types besides general-purpose timers
- Unified API for different timers (e.g. general-purpose and RTC)
- Allowing access to basic peripheral functions such as:
    - periodic auto-reload or
    - clock source selection
- Exposure of advanced and timer specific features such as:
    - Low-power operation modes or
    - Timer chaining
- Low ROM & RAM footprint
- Support for timer type specific implementations

Goals

Meta Goals

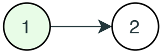
- Backing design decisions on facts from hardware analysis
- Assess different design and implementation approaches
- Incorporating feedback from the RIOT community
- Document why certain design decisions were (not) made

Conceptual Approach

1. Create low-level API-draft from conducted hardware analysis

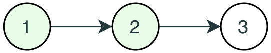
Conceptual Approach

1. Create low-level API-draft from conducted hardware analysis
2. Implement API for one selected MCU-platform



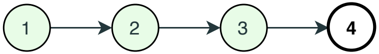
Conceptual Approach

1. Create low-level API-draft from conducted hardware analysis
2. Implement API for one selected MCU-platform
3. Iteratively improve API-draft based on findings from implementation



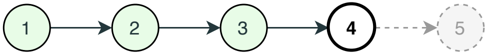
Conceptual Approach

1. Create low-level API-draft from conducted hardware analysis
2. Implement API for one selected MCU-platform
3. Iteratively improve API-draft based on findings from implementation
4. **Gather feedback for API-draft**



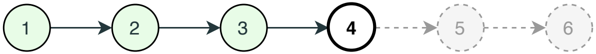
Conceptual Approach

1. Create low-level API-draft from conducted hardware analysis
2. Implement API for one selected MCU-platform
3. Iteratively improve API-draft based on findings from implementation
4. **Gather feedback for API-draft**
5. Conduct micro-benchmarks of different design decisions



Conceptual Approach

1. Create low-level API-draft from conducted hardware analysis
2. Implement API for one selected MCU-platform
3. Iteratively improve API-draft based on findings from implementation
4. **Gather feedback for API-draft**
5. Conduct micro-benchmarks of different design decisions
6. Take the discussion to the RIOT-OS community!



The NUCLEO-L476RG (STM32L476RG) Board

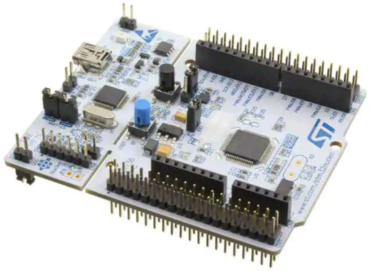


Figure 4: NUCLEO-L476RG Board

Available timer peripherals

- General-purpose timer
  - 32- and 16-bit
- Basic timer
- Advanced-control timer
- Low-power timer
- Real-time-clock (RTC)
- SysTick timer
- Watchdog (WDG)

STM32L476RG Timer Support in RIOT-OS

RIOT-OS Modules

`periph/`

- `timer` (📦)
- `rtc` (🕘), `rtt` (○)
- `pwm` (📶)
- `wdt` (🐾)

`sys/`

- `xtimer`, `ztimer`
- `evtimer`

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 🐾)

Σ Summary

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

API Design Ideas

Hardware Analysis: Key Findings

General Aspects

- Besides GP-timers, MCU platforms bring special timer types
    - Only watchdogs were classified as out of scope for the aspired API

More specific findings are discussed directly during presentation of the respective API design ideas.

Hardware Analysis: Key Findings

General Aspects

- Besides GP-timers, MCU platforms bring special timer types
    - Only watchdogs were classified as out of scope for the aspired API
- All platforms except one provide multiple GP-timers

More specific findings are discussed directly during presentation of the respective API design ideas.

Hardware Analysis: Key Findings

General Aspects

- Besides GP-timers, MCU platforms bring special timer types
    - Only watchdogs were classified as out of scope for the aspired API
- All platforms except one provide multiple GP-timers
- On 71% of all platforms, small timers ($\leq$ 16-bit) are capable of counter range extension using timer chaining

More specific findings are discussed directly during presentation of the respective API design ideas.

Hardware Analysis: Key Findings

General Aspects

- Besides GP-timers, MCU platforms bring special timer types
    - Only watchdogs were classified as out of scope for the aspired API
- All platforms except one provide multiple GP-timers
- On 71% of all platforms, small timers ($\leq$ 16-bit) are capable of counter range extension using timer chaining
- Different timer types (e.g. low-power timer) provide different platform specific features that are unexposed by current timer APIs

More specific findings are discussed directly during presentation of the respective API design ideas.

Hardware Analysis: Key Findings

General Aspects

- Besides GP-timers, MCU platforms bring special timer types
  - Only watchdogs were classified as out of scope for the aspired API
- All platforms except one provide multiple GP-timers
- On 71% of all platforms, small timers ($\leq$ 16-bit) are capable of counter range extension using timer chaining
- Different timer types (e.g. low-power timer) provide different platform specific features that are unexposed by current timer APIs
- Each individual peripheral comes with its own distinct set of capabilities that needs to be reflected by the API
  - e.g. number of compare channels or overflow-INT support

More specific findings are discussed directly during presentation of the respective API design ideas.

API Design Ideas

- Separation of hardware interface and user-facing API

API Design Ideas

- Separation of hardware interface and user-facing API
- Individual drivers per timer type (i.e. timer class)
  - Timer type specific implementations
  - Driver granular reusability across timer classes
  - Function granular reusability across drivers
  - Aim to keep memory footprint low

API Design Ideas

- Separation of hardware interface and user-facing API
- Individual drivers per timer type (i.e. timer class)
  - Timer type specific implementations
  - Driver granular reusability across timer classes
  - Function granular reusability across drivers
  - Aim to keep memory footprint low
- Each timer is represented by a `tim_periph_t` instance, referencing corresponding `tim_driver_t` and containing metadata

API Design Ideas

- Separation of hardware interface and user-facing API
- Individual drivers per timer type (i.e. timer class)
    - Timer type specific implementations
    - Driver granular reusability across timer classes
    - Function granular reusability across drivers
    - Aim to keep memory footprint low
- Each timer is represented by a `tim_periph_t` instance, referencing corresponding `tim_driver_t` and containing metadata
- Property based access to timer status and features

API Design Ideas

- Separation of hardware interface and user-facing API
- Individual drivers per timer type (i.e. timer class)
    - Timer type specific implementations
    - Driver granular reusability across timer classes
    - Function granular reusability across drivers
    - Aim to keep memory footprint low
- Each timer is represented by a `tim_periph_t` instance, referencing corresponding `tim_driver_t` and containing metadata
- Property based access to timer status and features
- Typical timer operations are provided by user-facing API functions

API Design Ideas

- Separation of hardware interface and user-facing API
- Individual drivers per timer type (i.e. timer class)
  - Timer type specific implementations
  - Driver granular reusability across timer classes
  - Function granular reusability across drivers
  - Aim to keep memory footprint low
- Each timer is represented by a `tim_periph_t` instance, referencing corresponding `tim_driver_t` and containing metadata
- Property based access to timer status and features
- Typical timer operations are provided by user-facing API functions
- Special features (e.g. chaining) as compile-time optional modules

API Design Ideas

Based on the proposed API design ideas...

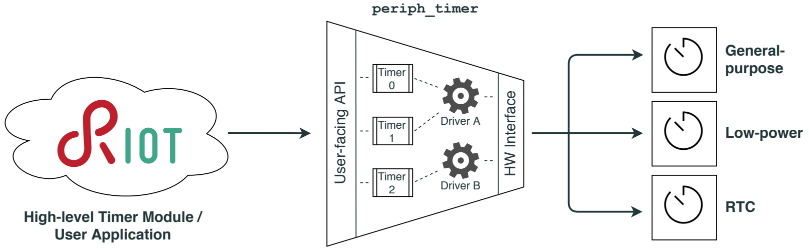


Figure 5: Overview of API design concept

Hardware Interface vs. User-facing API

User-facing API

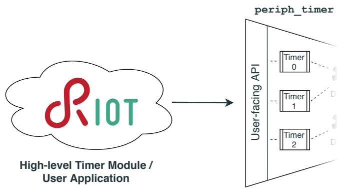
Timer type abstracted functions exposed to user or high-level timer module.

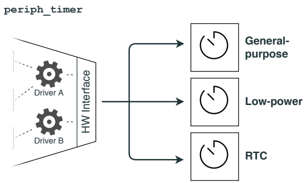
`timer_init(), timer_start(), timer_set(), timer_clear(), ...`

Hardware Interface

Compact and reusable timer drivers, directly interfacing the various hardware peripherals.

`init(), read(), write(), get_property(), ...`





Hardware Interface

Hardware Interface

Key design decisions, based on our conducted analysis.

Hardware Interface: Drivers

Analytical Finding

All platforms bring specialized timer types in addition to GP-timers.

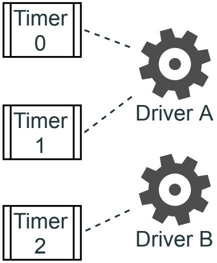
Implications for the Hardware Interface **1/5**

Driver-based approach: Drivers are implemented as minimal sets of function pointers that can be mapped to any hardware timer. They hereby can be used for interfacing various timer peripherals.

Hardware Interface: Drivers

Implications for the Hardware Interface **1/5**

Drivers are sets of function pointers that map to hardware timers.



```c
typedef struct {
    void (*fnct1)();
    int (*fcnt2)(void *args);
} driver_t;
// ...
const driver_t foo = {
    .fnct1 = &actual_fnct1_impl,
    .fnct2 = &actual_fnct2_impl
};
```

Hardware Interface: Function Mapping

Analytical Finding

Some timers vary largely, others only slightly, in their interfacing. The latter ones therefore allow (partial) sharing of driver code.

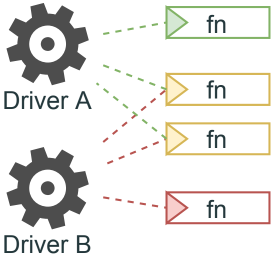
Implications for the Hardware Interface **2/5**

Function granular mapping in driver through individual pointers.
Provides reusability in order to keep maintenance efforts low and prevent large amounts of redundant code.

Hardware Interface: Function Mapping

Implications for the Hardware Interface 2/5

Function granular mapping in driver through individual pointers.



```
const driver_t foo = {
    .fnct1 = &fnct1_foo,
    .fnct2 = &fnct2_foo
};

const driver_t bar = {
    .fnct1 = &fnct1_foo,
    .fnct2 = &fnct2_bar
};
```

Hardware Interface: Static Attributes

Analytical Finding

Features and capabilities differ largely between timer classes.

Implications for the Hardware Interface **3/5**

Each peripherals individual capabilities and properties are reflected by the hardware interface. Common static attributes (e.g. counter width or compare channels) are included in `tim_periph_t`, individual or variable features are exposed via an property based access mechanism.

Hardware Interface: Static Attributes

Implications for the Hardware Interface **3/5**

Common static attributes are included in `tim_periph_t`.
Specific features are exposed via an property based access mechanism.

```
typedef struct {
    // ...
    const uint16_t width    :8;
    const uint16_t channels :4;
    // ...
} tim_periph_t;
```

```
typedef struct {
    // ...
    tim_propval_t (*get_property)(tim_prop_t prop);
    int (*set_property)(tim_prop_t prop, tim_propval_t val);
    // ...
} tim_driver_t;
```

Hardware Interface: Virtual Types

Analytical Finding

In particular 16-bit platforms (71%) strongly rely on timer chaining for range extension and long timeouts.

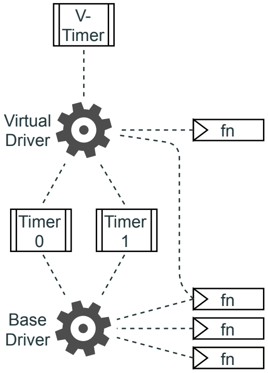
Implications for the Hardware Interface **4/5**

"Virtual" timer types that consist of multiple chained hardware modules are introduced. They can provide additional driver code and present themselves as single `tim_periph_t` instances.

Hardware Interface: Virtual Types

Implications for the Hardware Interface **4/5**

"Virtual" timer types as a compound of multiple hardware modules.



Hardware Interface: Memory Footprint

Analytical Finding

Especially for very resource-restricted devices the additionally introduced memory footprint of driver structs has to be kept as low as possible.

Implications for the Hardware Interface **5/5**

- Strongly coupled functionality is combined into a single hardware interface function. For example: `set()`, `set_periodic()`, `clear()` combined into `set_channel(mode)`.
- Compile time optional modules for specific features (e.g. PWM).
- Properties and capabilities are combined into appropriate bit fields.

Hardware Interface: Memory Footprint

Implications for the Hardware Interface **5/5**

Memory footprint of the timer and driver structs has to be kept low.

Introduced Memory Footprint **(MCU: STM32L476RG)**

For every used **timer class** (`tim_driver_t`)

$$8 \cdot \texttt{sizeof(size\_t)}$$

$$8 \cdot 4\,\text{Byte} = 32\,\text{Byte}$$

For every used **timer peripheral** (`tim_periph_t`)

$$\texttt{sizeof(tim\_t)} + \texttt{sizeof(size\_t)} + \texttt{sizeof(uint16\_t)}$$

$$4\,\text{Byte} + 4\,\text{Byte} + 2\,\text{Byte} = 10\,\text{Byte}$$

User-facing API

User-facing API

Key design decisions, based on our conducted analysis.

User-facing API: Base Functions

Analytical Finding

Platform and timer hardware agnostic use of basic features is a must.

Implications for the User-facing API **1/6**

Common timer features like `timer_start()` or `timer_read()` are exposed through well-defined functions implemented for every timer class.

User-facing API: Base Functions

Implications for the User-facing API **1/6**

Basic common timer features are exposed via well-defined functions.

```
int timer_init(/* tim, freq, clk, ovf, cb, arg */);
int timer_start(/* tim */);
int timer_stop(/* tim */);
tim_cnt_t timer_read(/* tim */);
void timer_write(/* tim, cnt */);
int timer_set(/* tim, channel, timeout */);
int timer_set_absolute(/* tim, channel, cnt */);
int timer_clear(/* tim, channel */);
```

Displaying only argument names to preserve screen space.

User-facing API: Properties & Capabilities

Analytical Finding

Access to platform specific timer properties and features is required for some applications (e.g. low-power operation).

Implications for the User-facing API **2/6**

Timer capabilities and properties are exposed to the user, thereby allowing usage of such platform specific features.

User-facing API: Properties & Capabilities

Implications for the User-facing API 2/6

Timer capabilities and properties (incl. platform specific) are exposed.

```
typedef struct {
    // ...
    const uint16_t width    :8;
    const uint16_t channels :4;
    // ...
} tim_periph_t;
```

```
typedef enum {
    // ...
    TIM_PROP_MODE              = 0x01, /**< Timer counting mode (e.g. disabled, continuous, ...) */
    TIM_PROP_CNT_DIR           = 0x02, /**< Counting direction (e.g. up, down, up/down) */
    TIM_PROP_INT_CMP_MATCH     = 0x03, /**< Interrupt generation on compare channel match */
    TIM_PROP_INT_OVF           = 0x04, /**< Interrupt generation on counter register overflow */
    // ...
} tim_prop_t;
```

```
unsigned int timer_get_property(tim_periph_t *const tim, tim_prop_t prop);
int timer_set_property(tim_periph_t *const tim, tim_prop_t prop, unsigned int val);
```

User-facing API: Compound Functions

Analytical Finding

Some frequently used functions are just a compound of base driver calls, such as relative timer arming (i.e. `timer_set()`).

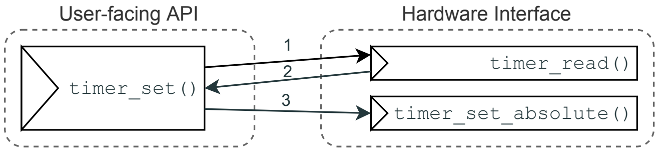
Implications for the User-facing API **3/6**

Since those functions can be realized as sole combinations of driver calls, those are platform independently implemented directly in the user-facing API. Hereby the `tim_driver_t` struct is kept small and clear.

User-facing API: Compound Functions

Implications for the User-facing API **3/6**

Compound functions are solely implemented in the user-facing API.



User-facing API: Clock Configuration

Analytical Finding

On 84% of all platforms, timers can be driven by at least two clocks, selectable during runtime (e.g. when entering a lower power mode).

Implications for the User-facing API **4/6**

Clock selection is optionally exposed at runtime. The clock source can be specified either platform agnostic (i.e. always use default CLK) or explicitly selected among the set of available CLKs. Only modifications that are free of side effects on other peripherals shall be supported.

User-facing API: Clock Configuration

Implications for the User-facing API **4/6**

MCU-Platform (in-)dependent clock selection is exposed at runtime.

```
#define HAVE_TIMER_CLK_T
typedef enum {
    TIM_CLK_DEFAULT, /**< Default clock source for the peripheral.
    ↪  Unspecified or don't care. **/
    TIM_CLK_APB,     /**< Advanced Peripheral Bus. **/
    TIM_CLK_LSI,     /**< Low-speed internal oscillator **/
    TIM_CLK_HSI16,   /**< High-speed internal oscillator **/
    // ...
} tim_clk_t;
```

Excerpt of the STM32 specific clocks

User-facing API: Overflow Interrupts

Analytical Finding

Timer peripherals can be configured to generate overflow interrupts.

Implications for the User-facing API **5/6**

Introduce overflow INT configuration and pass respective interrupt cause to timer callback during invocation.

User-facing API: Overflow Interrupts

Implications for the User-facing API **5/6**

Introduce overflow INT configuration and pass interrupt cause to timer callback during invocation.

```
typedef enum {
    TIM_INT_UNKNOWN        = 0x00, /**< Unknown event */
    TIM_INT_COMPARE_MATCH = 0x01, /**< Compare channel match */
    TIM_INT_OVERFLOW       = 0x02  /**< Counter register overflow */
} tim_int_t;
```

```
typedef void (*tim_cb_t)(void *arg, tim_int_t cause, int channel);
```

User-facing API: Status Reporting

Analytical Finding

High-level modules often require information on pending IRQs or other unhandled events, even when interrupts are currently masked.

Implications for the User-facing API **6/6**

Status reporting, including information about currently unhandled events (e.g. pending compare match, overflow, . . .), is provided by the API.

User-facing API: Status Reporting

Implications for the User-facing API 6/6

Status reporting (incl. pending events) is provided by the API.

```c
typedef enum {
  // ...
  TIM_PROP_OVF_PENDING       = 0xF0, /**< Overflow flag is set */
  TIM_PROP_CMP_MATCH_PENDING = 0xF1, /**< At least one compare match flag is set */
  // ...
} tim_prop_t;
```

```c
unsigned int timer_get_property(tim_periph_t *const tim, tim_prop_t prop);
```

Advantages and Problems

Advantages

- ⊕ Integration of currently unsupported timer types, all usable through a unified and MCU-independent API

Advantages

- Integration of currently unsupported timer types, all usable through a unified and MCU-independent API
- Benefitting both high-level modules and specialized applications by exposing features, commonly found on mid-range to high-end MCUs

Advantages

- Integration of currently unsupported timer types, all usable through a unified and MCU-independent API
- Benefitting both high-level modules and specialized applications by exposing features, commonly found on mid-range to high-end MCUs
- Driver based approach introduces flexibility that aids a large amount of diverse use cases

Advantages

- Integration of currently unsupported timer types, all usable through a unified and MCU-independent API
- Benefitting both high-level modules and specialized applications by exposing features, commonly found on mid-range to high-end MCUs
- Driver based approach introduces flexibility that aids a large amount of diverse use cases
- Widening of runtime configuration possibilities (e.g. clock selection)

Advantages

- Integration of currently unsupported timer types, all usable through a unified and MCU-independent API
- Benefitting both high-level modules and specialized applications by exposing features, commonly found on mid-range to high-end MCUs
- Driver based approach introduces flexibility that aids a large amount of diverse use cases
- Widening of runtime configuration possibilities (e.g. clock selection)
- Combining multiple hardware timers into one virtual `tim_periph_t` instance (e.g. for chaining)

Advantages

- Integration of currently unsupported timer types, all usable through a unified and MCU-independent API
- Benefitting both high-level modules and specialized applications by exposing features, commonly found on mid-range to high-end MCUs
- Driver based approach introduces flexibility that aids a large amount of diverse use cases
- Widening of runtime configuration possibilities (e.g. clock selection)
- Combining multiple hardware timers into one virtual `tim_periph_t` instance (e.g. for chaining)
- Compact hardware interface by moving all hardware agnostic code to user-facing API

Current Problems and Possible Pitfalls

- ⊖ Increased memory footprint due to introduction of dedicated drivers

Current Problems and Possible Pitfalls

- Increased memory footprint due to introduction of dedicated drivers
- `tim_cnt_t` fixed to widest utilized counter even on smaller timers
  - We neither want a specific function for every timer width (e.g. `read_16()`, `read_32()`, `read_64()`) nor error prone `void`-ptrs.

Current Problems and Possible Pitfalls

- Increased memory footprint due to introduction of dedicated drivers
- `tim_cnt_t` fixed to widest utilized counter even on smaller timers
  - We neither want a specific function for every timer width (e.g. `read_16()`, `read_32()`, `read_64()`) nor error prone `void`-ptrs.
- Integration of feature restricted compare channels, such as ones that only allow to select predefined compare match values, can be improved (e.g. STM32 periodic wakeup timer)

Current Problems and Possible Pitfalls

- Increased memory footprint due to introduction of dedicated drivers
- `tim_cnt_t` fixed to widest utilized counter even on smaller timers
  - We neither want a specific function for every timer width (e.g. `read_16()`, `read_32()`, `read_64()`) nor error prone `void`-ptrs.
- Integration of feature restricted compare channels, such as ones that only allow to select predefined compare match values, can be improved (e.g. STM32 periodic wakeup timer)
- Current `cpu` implementations of `periph/timer` need to be updated

Current Problems and Possible Pitfalls

- Increased memory footprint due to introduction of dedicated drivers
- `tim_cnt_t` fixed to widest utilized counter even on smaller timers
  - We neither want a specific function for every timer width (e.g. `read_16()`, `read_32()`, `read_64()`) nor error prone `void`-ptrs.
- Integration of feature restricted compare channels, such as ones that only allow to select predefined compare match values, can be improved (e.g. STM32 periodic wakeup timer)
- Current `cpu` implementations of `periph/timer` need to be updated
- A backwards compatability layer for high-level modules potentially needs to be provided

Outlook and Future Work

Future Work

Roadmap

- Conduct micro-benchmarks of different design decisions
- Take the discussion to the RIOT-OS community!
- Finalize API draft and implement new low-level timer driver
- Outline a migration plan for existing cpu drivers
- Broad range hardware-in-the-loop (HIL) testing
- Incorporate new and enhanced features into high-level timer modules (e.g. `xtimer` / `ztimer`)

Questions?

Discussion!