05: PWM

Microcontrollers

Stefan Huber
www.sthu.org

Dept. for Information Technologies and Digitalisation
FH Salzburg

Winter 2024

PWM

Modes of operation for timers

Timers can

- count upwards or downwards,
- fire IRQs on overflows or when certain values are matched,
- and manipulate pin levels on these events.

→ This gives rise to different modes of operation.

The three timers of the ATmega32 provide at least four modes:

- Normal mode
  We used this mode in the example of the periodic timer interrupt every 1024 µs.
- Clear Timer on Compare Match (CTC) mode
- Single slope PWM mode
- Phase correct PWM mode

Do something when TCNT matches with the Output Compare Register (OCR).

- Raise an interrupt
- Clear TCNT register
- Change the output compare pin (OC), e.g., toggle pin to generate a square wave.[1]

Can be used to generate a pulse-width modulation (PWM) signal.

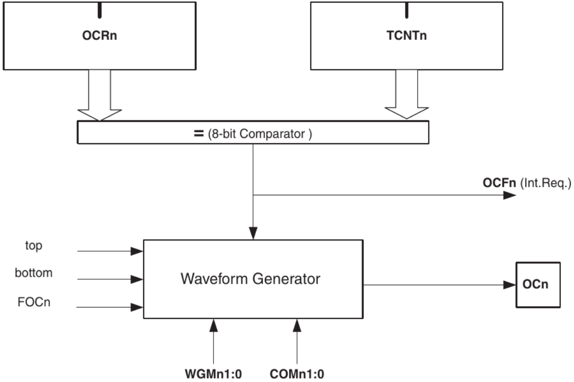


Figure: Output compare unit, [ATmega32]

[1] This is set by the compare match output mode. See [ATmega32] for timer 2.

Normal mode

The timer simply counts from `BOTTOM` to `MAX`, after which it restarts at `BOTTOM`.

- `BOTTOM` is zero. `MAX` is $2^n - 1$ for n-bit timers.
- A timer overflow flag `TOVn` is set when the timer becomes zero again and an interrupt is raised.
- The counter value can be updated at any time.

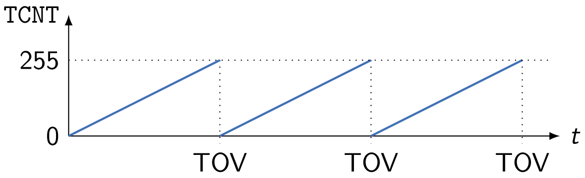


Figure: 8-bit timer example for normal mode. `MAX` is 255.

Example

```
#include <avr/interrupt.h>

ISR (TIMER0_OVF_vect) {
    /* TCNT0 became zero and raised an overflow interrupt. */
}

void init() {
    /* Timer/counter control register for timer/counter 0. (p80)
     * Set (waveform generation) mode to 'normal'.
     * Set compare match output mode to 'normal' (OC0 pin disconnected).
     * Set clock select to internal clock with prescaler 8. */
    TCCR0 = (1 << CS01);
    /* Set TOIE0 bit (timer overflow interupt enable for timer/counter 0) */
    TIMSK |= (1 << TOIE0);

    /* Set global interrupt enable bit. */
    sei();
}
```

See register description in [**ATmega32**].

Custom periods

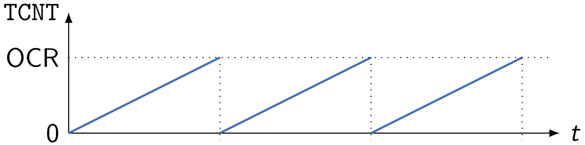
Assume we want shorter, arbitrary timer periods than the full period from 0 to $2^n - 1$.

- We could re-initialize the counter register to some value $b > 0$ in the overflow ISR.
- But if we do that in software then we have to take an vague interrupt latency into account.

Custom periods

Assume we want shorter, arbitrary timer periods than the full period from 0 to $2^n - 1$.

- We could re-initialize the counter register to some value $b > 0$ in the overflow ISR.
- But if we do that in software then we have to take an vague interrupt latency into account.
- The timer mode Clear Timer on Compare Match Mode (CTC) resets the timer when `TCNT` reaches `TOP`.
  - The `TOP` value is stored in the register `OCR`.
  - It holds that $0 \leq \texttt{TCNT} \leq \texttt{OCR}$.



Typically, OC pin is either disconnected or in toggle mode.

Example applications:

- Generate a square wave signal at the OC pin with a period of $2 \cdot (1 + \texttt{OCR})$ timer ticks.[2]
- Count external events and raise an interrupt after k events. (Pulse accumulator mode.)

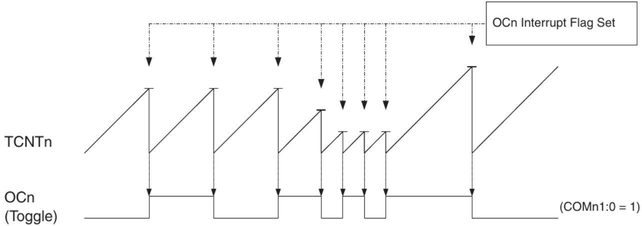


Figure: [**ATmega32**]

Changing `OCR` is done in the ISR of the compare match.

- Otherwise, `TCNT` may have overtaken `OCR`, never match, and overflow after hitting `MAX`.

2 See [**ATmega32**].

Example: Toggle a pin every 43 ticks

In this example we would like to toggle the pin OC0 precisely every 43 CPU cycles.

- We use CTC mode to have the timer being cleared upon compare match.
- We set the output mode to toggle the OC0 pin.
- We set compare match register to 42.

```c
void init() {
    /* Toogle pin OC0 after 43 cycles. TCNT0 cycles in [0, 42]. */
    OCR0 = 42;

    /* Timer/counter control register for timer/counter 0. (p80)
     * Set (waveform generation) mode to 'CTC'.
     * Set compare match output mode of OC0 to 'toggle'.
     * Set clock select to internal clock without prescaler. */
    TCCR0 = (1 << WGM01) | (1 << COM00) | (1 << CS00);
}
```

- There is no need for an ISR.

PWM

The pulse-width modulation (PWM) is a digital modulation:

- It allows to encode a value $d \in [0,1]$ using a digital signal $x(t)$ in a time period p.
- Simply set x high for a d-th fraction of the time. The value d is called the duty cycle[3].
- To sum up, a PWM signal is a function $[0,1] \to \{0,1\}$ to encode a value $[0,1]$ as its mean.
- Of course, microcontrollers generate *time-discrete* PWM signals.

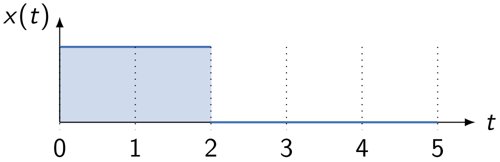
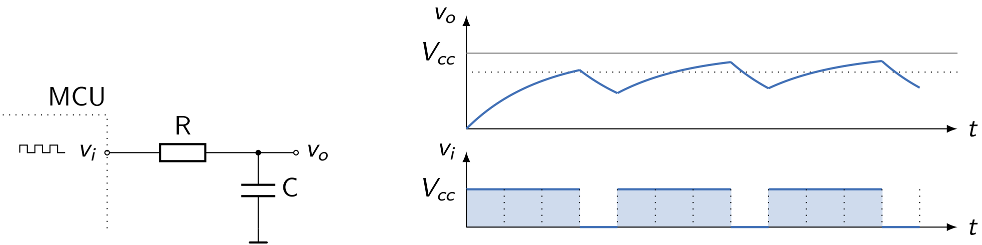


Figure: A single period of a (time-discrete) PWM signal with a duty cycle of 0.4 or 40 %.

3 Dt. Tastverhältnis

We demodulate a PWM by a low pass to get the steady component. This gives analog output.



With duty cycle $d \in [0, 1]$, we have $d \cdot V_{cc}$ as steady component.

- Low-pass cut-off frequency is $f_c = \frac{1}{2\pi RC}$.
- Increasing time constant $\tau = RC$ is costly (larger components) and causes longer response times.
- To reduce oscillation of v_o we therefore strive for a higher PWM frequency.

PWM applications for microcontrollers

- digital-analog conversion
- dimming LEDs or displays
- fan control
- frequency converter for motor control

In many applications the load has a low-pass characteristic:

- receiving electrical system
- inertial of a motor or mechanical system
- human visual perception of light

If the PWM frequency is sufficiently high then need no explicit filters.

- Otherwise, we need external capacitors and/or inductors to filter the PWM signal.

Single slope PWM generation

- The PWM is output at the output compare pin.
- Let a timer `TCNT` repeatedly count from a value `BOTTOM` to a value `MAX`.
  - Set the PWM output high on timer overflow.
  - Set the PWM output low on `ORC` match.

- The PWM period is 2^n timer clock ticks for an n-bit timer.
- The duty cycle is `OCR` $/2^n$, if output is not inverted.
- There is also an inverted mode.

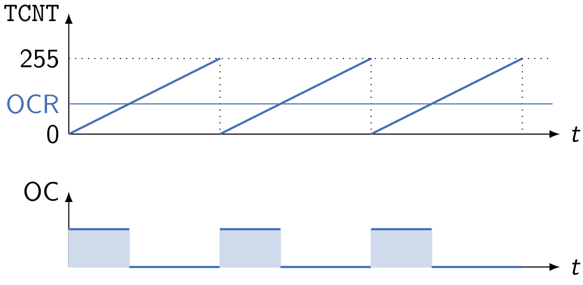


Figure: 8-bit timer single-slope example.

Single slope PWM generation

- The PWM is output at the output compare pin.
- Let a timer `TCNT` repeatedly count from a value `BOTTOM` to a value `MAX`.
  - Set the PWM output high on timer overflow.
  - Set the PWM output low on `ORC` match.

- The PWM period is 2^n timer clock ticks for an n-bit timer.
- The duty cycle is `OCR` $/2^n$, if output is not inverted.
- There is also an inverted mode.

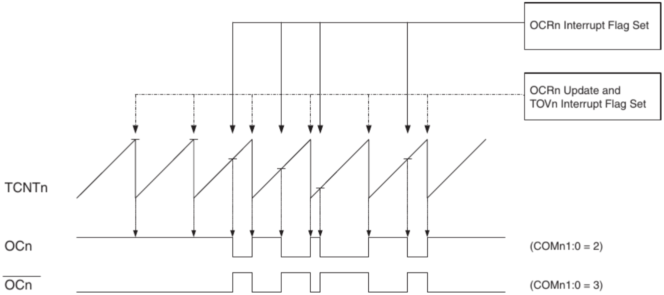


Figure: See **[ATmega32]**

Single slope PWM generation

```c
#include <avr/io.h>

void init() {
    /* Set PB3 (OC0 pin) to output */
    DDRB |= (1 << PB3);
    /* Duty cycle of 192/256 = 75% */
    OCR0 = 192;

    /* Timer/counter control register for timer/counter 0. (p80)
     *
     * Set (waveform generation) mode to 'Fast PWM'.
     * Set compare match output mode of OC0 to 'non-inverted PWM'.
     * Set clock select to internal clock without prescaler. */
    TCCR0 = (1 << WGM01) | (1 << WGM00) | (1 << COM01) | (1 << CS00);
}
```

Dual slope PWM generation

The timer register `TCNT` runs from `BOTTOM` to `MAX` and back again.

- On `OCR` match while upcounting, `OC` is cleared.
- On `OCR` match while downcounting, `OC` is set.

- The PWM period is 2^{n+1} timer clock ticks for an n-bit timer.
- The duty cycle is again `OCR` $/2^n$ for non-inverted mode.
- There is also an inverted mode.

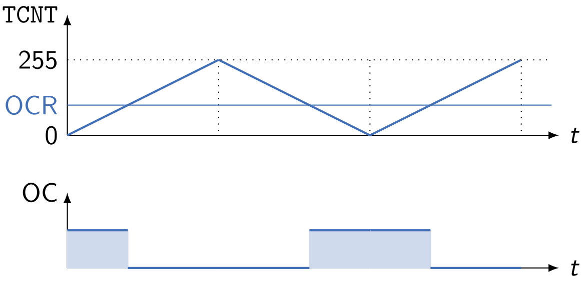


Figure: 8-bit timer dual-slope example.

Dual slope PWM generation

The timer register `TCNT` runs from `BOTTOM` to `MAX` and back again.

- On `OCR` match while upcounting, `OC` is cleared.
- On `OCR` match while downcounting, `OC` is set.

- The PWM period is 2^{n+1} timer clock ticks for an n-bit timer.
- The duty cycle is again `OCR` $/2^n$ for non-inverted mode.
- There is also an inverted mode.

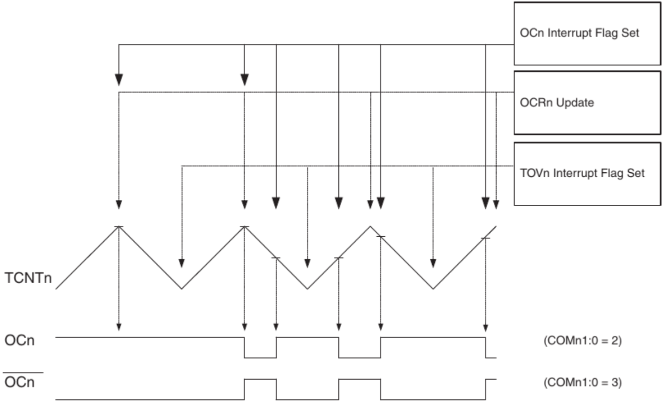


Figure: See **[ATmega32]**

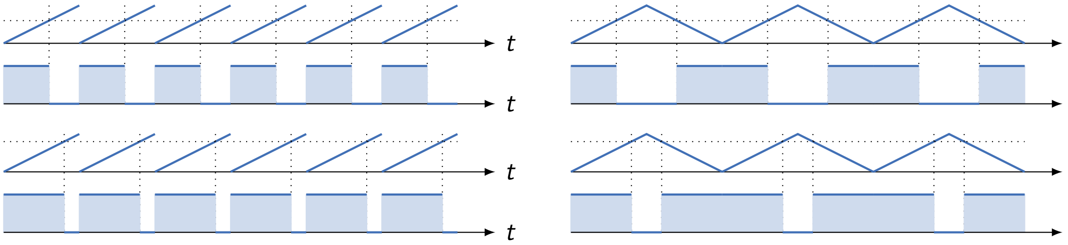
Dual slope PWM generation

```c
#include <avr/io.h>

void init() {
    /* Set PB3 (OC0 pin) to output */
    DDRB |= (1 << PB3);
    /* Duty cycle of 192/256 = 75% */
    OCR0 = 192;

    /* Timer/counter control register for timer/counter 0. (p80)
     *
     * Set (waveform generation) mode to 'Phase correct PWM'.
     * Set compare match output mode of OC0 to 'clear on compare match when
     * up-counting'.
     * Set clock select to internal clock without prescaler. */
    TCCR0 = (1 << WGM00) | (1 << COM01) | (1 << CS00);
}
```

Single slope versus dual slope



ATmega32 calls single slope mode the *fast PWM mode*:

- It has double the frequency (half the period) of the dual slope mode.[4]
- Higher frequency allows for smaller external components, e.g., capacitors for DAC applications.

ATmega32 calls dual slope mode the *phase correct PWM mode*:

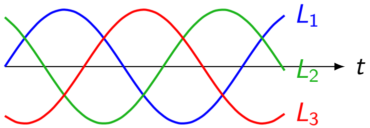
- Increasing the duty cycle in the single slope mode makes a phase shift to the right.
- In the dual slope mode the pulse width changes *in phase* with the timer period.

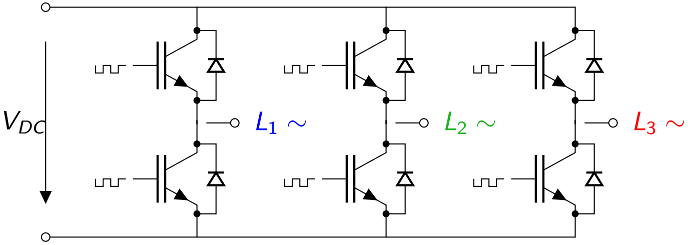
[4] The period in dual slope mode is not exactly doubled, but it is $2(2^n - 1)$ timer ticks for an n-bit timer, as the timer values `BOTTOM` and `TOP` are not repeated.

The issue with phase shifts

Example: Three-phase inverter for motion control.

- Phases are 120°-shifted sinus signals and generated by PWM signals. The current signal value corresponds to the duty cycle of a PWM signal.





- In single slope mode, the change of the duty cycle introduces changes in the phase shift in the PWM signal. Hence, the sinus signals are shifted – depending on the signal value!

Conclusion: Phase shifts of PWM signals can be pathological when we have multiple PWM signals that are related to each other.

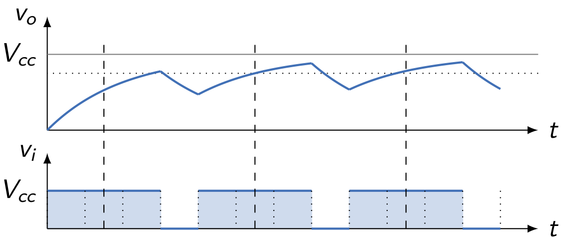
Dual slope PWM and measuring analog input

For closed-loop control, like motor control, we need to measure feedback signals.

- Thought experiment: Read back analog output v_o generated by a low-pass filtered PWM v_i.

Let's read back in the timer overflow ISR:

- Single slope PWM: Here v_o is at a local minimum, which is bad.
- Dual slope PWM: Timer overflow is in the middle of a high phase. Here v_o is closer to its steady component, so error is lower.

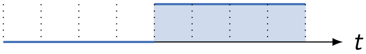


PWM on the Raspberry Pi

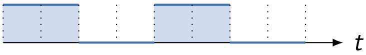
Any digital signal x with period p and mean value d, i.e., with the property $d = \frac{1}{p}\int_0^p x(t)\,\mathrm{d}t$ is called a PWM signal with duty cycle d. What are good candidates?[5]

Assume we would like to have a duty cycle of 50 % and a period of 8 cycles. Examples are:

(a) ATmega32, non-inverted mode



(b) ATmega32, inverted mode



(c) Doubling the frequency

(d) Maximum frequency

The BCM2835 SoC on the Raspberry Pi generates the last: it gives the highest frequency.

5 And how many candidates do exist?

References I

The PWM generation algorithm of the BCM2835

Assume we would like to generate a PWM signal with duty cycle $3/7$ and a period of 7. How do we achieve a good (high) resulting frequency?

[6] Strictly speaking, the algorithm listing in [**BCM2835**] is incomplete due to broken typesetting. This is a translation into Python. The function pwm() is a so-called *generator* function that works like an iterator. Each time it is called the *yield* instruction returns the next PWM signal value.

The PWM generation algorithm of the BCM2835

Assume we would like to generate a PWM signal with duty cycle $3/7$ and a period of 7. How do we achieve a good (high) resulting frequency?

According to [**BCM2835**], the algorithm to generate n high cycles out of m cycles, is as follows:[6]

```python
def pwm(n, m):
  k = 0
  while True:
    k += n
    if k >= m:
      k = k % m
      yield 1      # Output a high
    else:
      yield 0      # Output a low
```

Two questions:

- Correctness: Why is this giving a duty cycle of n/m?
- Quality: Why is this giving a good PWM signal frequency?

6 Strictly speaking, the algorithm listing in [**BCM2835**] is incomplete due to broken typesetting. This is a translation into Python. The function pwm() is a so-called *generator* function that works like an iterator. Each time it is called the *yield* instruction returns the next PWM signal value.

Algorithm analysis: correctness

Lemma

Given that $n \leq m$ then after m invocations of `pwm(n, m)` exactly n highs are output. Furthermore, the output is periodic with period m.

Proof. Let us rephrase the algorithm:

```python
def pwm(n, m):
  k = 0
  while True:
    k += n
    if k >= m:
      k = k % m
      yield 1      # Output a high
    else:
      yield 0      # Output a low
```

```python
def pwm2(n, m):
  k = 0
  while True:
    k += n
    if k passes an m-multiple:   # (k-n, k] contains
                                 # a multiple of m
      yield 1      # Output a high
    else:
      yield 0      # Output a low
```

After m invocations of `pwm2(n, m)`:

- k is equal to $n \cdot m$. Hence, k passed an m-multiple exactly n times.
- Also, in `pwm(n, m)` the variable k is zero again, so the output repeats after m cycles. □

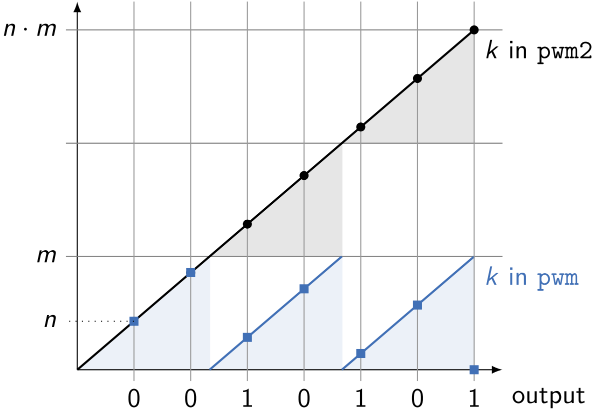


Figure: The variables k in the two algorithms for $m = 7$ and $n = 3$. They output a 1 when the black line crosses a horizontal line resp. when the blue line starts at zero again. The horizontal lines are equally spaced, hence the 1's in the output are more ore less equally spaced: the PWM frequency is good.

Additional capabilities of ATmega32 timers

Some features are only provided by some timers:

	TCNT0	TCNT1	TCNT2
Input capture unit		•	
Phase and frequency correct mode		•	
Two output compare match pins		•	
Asynchronous mode with ext. crystal			•

Many details have been skipped:

- Different timer resolution modes
- Custom PWM periods
- Details on the double buffering of certain registers.

The TCNT1 alone provides 15 different modes in total.[7]

[7] See [**ATmega32**].