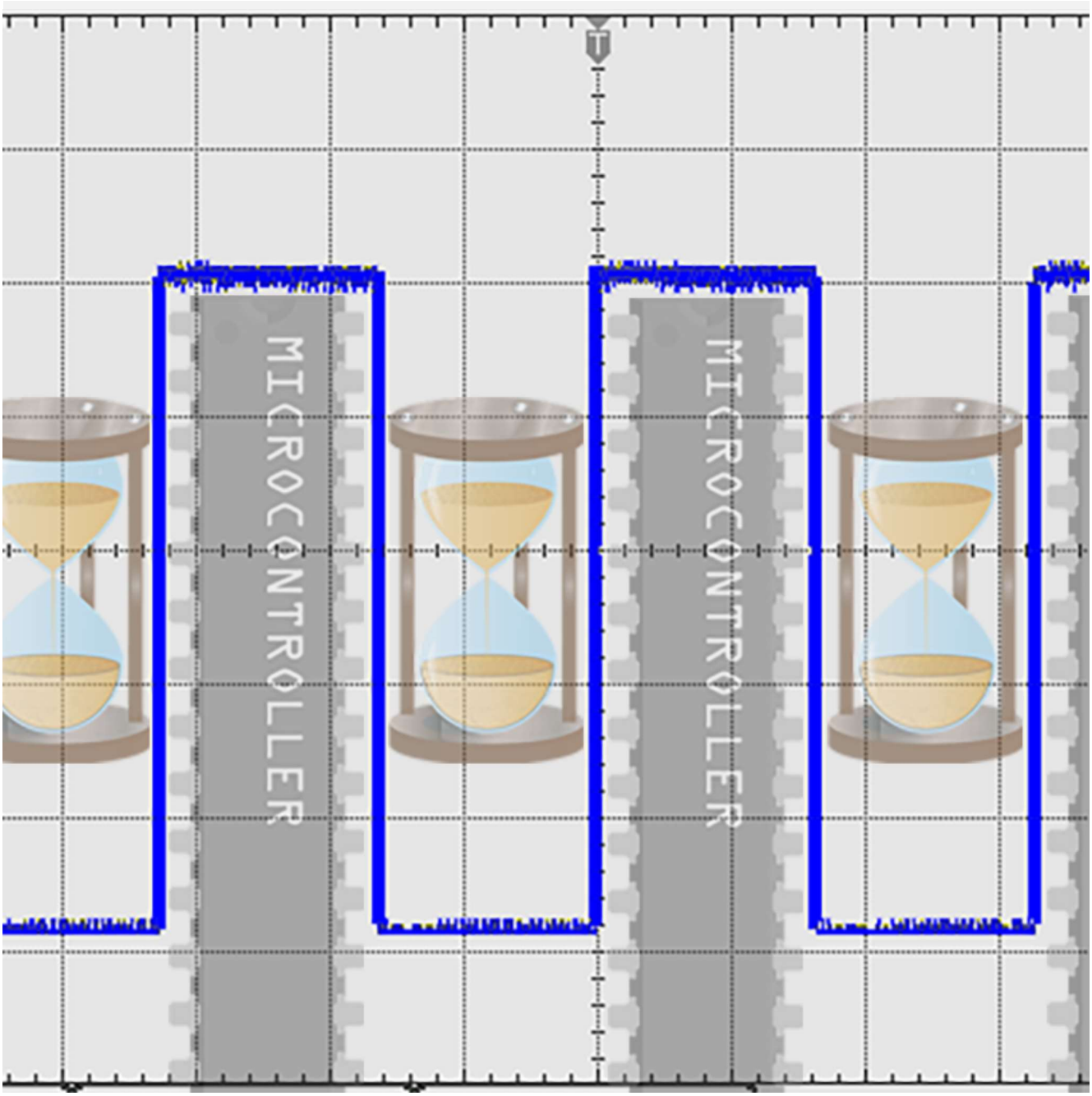
Wolles Elektronikkiste (https://wolles-elektronikkiste.de/en)

Timer and PWM – Part 1 (8-Bit Timer0/2)

POSTED ON NOVEMBER 20, 2020 (HTTPS://WOLLES-ELEKTRONIKKISTE.DE/EN/TIMER-AND-PWM-PART-1-8-BIT-TIMER0-2) BY WOLFGANG EWALD (HTTPS://WOLLES-ELEKTRONIKKISTE.DE/EN/AUTHOR/W0ELLELLE_EWOOD)



About this post

In this article about timer and pulse width modulation (PWM) I will dive into the depths of the Arduino UNO and the ATmega328P respectively. The Arduino is a great invention that makes it easy to enter the world of microcontrollers. However, the price of simplification is that not all capabilities of the underlying microcontroller have been transferred to the Arduino world. Timer and PWM are definitely part of this. These features are still accessible (with few limitations), but only through the somewhat cryptic logic (bit) operations.

I will try to explain the subject in a comprehensible way with many small examples. As a result, the post has become extremely long – more of a book chapter than a blog post. But I also want that more inexperienced "Arduinoists" understand it. However, to understand this article you will need some knowledge of logic operations and port manipulation. If you still have some catching up to do, look at my last post (https://wolles-elektronikkiste.de/en/logical-operations-and-port-manipulation?lang=en).

Because of the size, I limit myself to the 8 bit timers in this article. I deal with the 16 bit timer in part 2 (https://wolles-elektronikkiste.de/en/timer-and-pwm-part-2-16-bit-timer1).

Content

So, that's what I will talk about:

- What are timers and PWM
- Which timer does the Arduino UNO (or ATmega 328 P) have?
- The four timer modes:
  - Normal
  - CTC (clear timer on compare match)
  - Fast PWM
  - Phase-correct PWM
- C – sketches in Atmel Studio

What is a timer, what is PWM?

I start quite simply: first, a counter belongs to a timer. And it does what you would expect – namely it counts. It counts repeatedly up or down to a limit or up to adjustable limits. The counter becomes a timer when it counts time-dependently. Finally, the timer does not make sense until reaching the limit or an intermediate value can trigger an event.

PWM is simply the technique of periodically switching the digital outputs of your microcontroller HIGH and LOW, i.e. generating square wave signals with certain patterns. And since this is supposed to happen rapidly and in the background, you use timers for it. You have certainly already generated PWM signals, e.g. via analogWrite, servo motor control or the tone function. This is ready-made food, so to speak. Here you will learn how to "cook" PWM signals yourself.

The timers of the Arduino UNO

In this post (and part 2), I'll cover the following timers:

- Timer0: 8 bit
- Timer1: 16 bit (part 2 (https://wolles-elektronikkiste.de/en/timer-and-pwm-part-2-16-bit-timer1))
- Timer2: 8 bits

There are more specialized timers (e.B. watchdog timer). However, the ones mentioned here are those that are used for PWM and similar purposes.

The corresponding counters, control registers and I/O pins

The Timer/Counter Register TCNTx

Now it's getting dry and theoretical. But don't worry, the examples will make things clearer. Learn now – and understand later.

Depending on their size, there are one or two counter registers for each timer, namely TCNT0 (Timer/Counter 0), TCNT1L, TCNT1H and TCNT2. Since the Timer1 is 16 bit, it needs two registers. In the following, however, I limit myself to the two 8 bit timers.

The Timer/Counter Control Registers TCCRxy

The main settings for the timers are made in the Timer/Counter Control Registers. TCCR0A and TCCR0B belong to Timer1, TCCR2A and TCCR2B belong to Timer2. Here is an example of the Timer2 Control Registers:

Bit No	7	6	5	4	3	2	1	0	
Identifier	COM2A1	COM2A0	COM2B1	COM2B0	-	-	WGM21	WGM20	TCCR2A
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W	

(https://wolles-elektronikkiste.de/wp-content/uploads/2019/12/TCCR2A.png)

TCCR2A Control Register for Timer2

Bit No	7	6	5	4	3	2	1	0	
Identifier	FOC2A	FOC2B	-	-	WGM22	CS22	CS21	CS20	TCCR2B
Read/Write	W	W	R	R	R/W	R/W	R/W	R/W	

(https://wolles-elektronikkiste.de/wp-content/uploads/2019/12/TCCR2B.png)

TCCR2B Control Register for Timer2

You can also find the registers and their description in the rather elongated data sheet (http://ww1.microchip.com/downloads/en/DeviceDoc/ATmega48A-PA-88A-PA-168A-PA-328-P-DS-DS40002061A.pdf) for the ATmega 48 / 88 / 168 / 328 family.

The designations for the registers and bits are defined in the AVR libraries by #define instructions. And since these libraries are an integral part of the Arduino IDE, this makes access easy.

Basically, the bits of the register are like switches: set bit = 1 = switch on, bit not set = 0 = switch off. How to combine the switches in a meaningful way is the subject of this article.

The Timer/Counter Interrupt Mask Registers TIMSKx

If you want a timer/counter overflow or the match with a comparison value (Compare Match) to trigger an interrupt, you can set this in the corresponding registers TIMSK0 or TIMSK2. Here TIMSK2 as an example:

Bit No	7	6	5	4	3	2	1	0	
Identifier	-	-	-	-	-	OCIE2B	OCIE2A	TOIE2	TIMSK2
Read/Write	R	R	R	R	R	R/W	R/W	R/W	

(https://wolles-elektronikkiste.de/wp-content/uploads/2019/12/TIMSK2.png)
Setting TOIE2 (Timer/Counter2 Overflow Interrupt Enable) causes a register overflow of TCNT2 (at 8 bits that's after 255) to trigger an interrupt. Setting OCIE2A and OCIE2B bits (= Timer/Counter2 Output Compare Match Interrupt Enable A or B) cause an interrupt to be triggered if TCNT2 matches the comparison values in the Output Compare Registers.

TIMSK0 is organized accordingly. Simply replace 2 with 0 each.

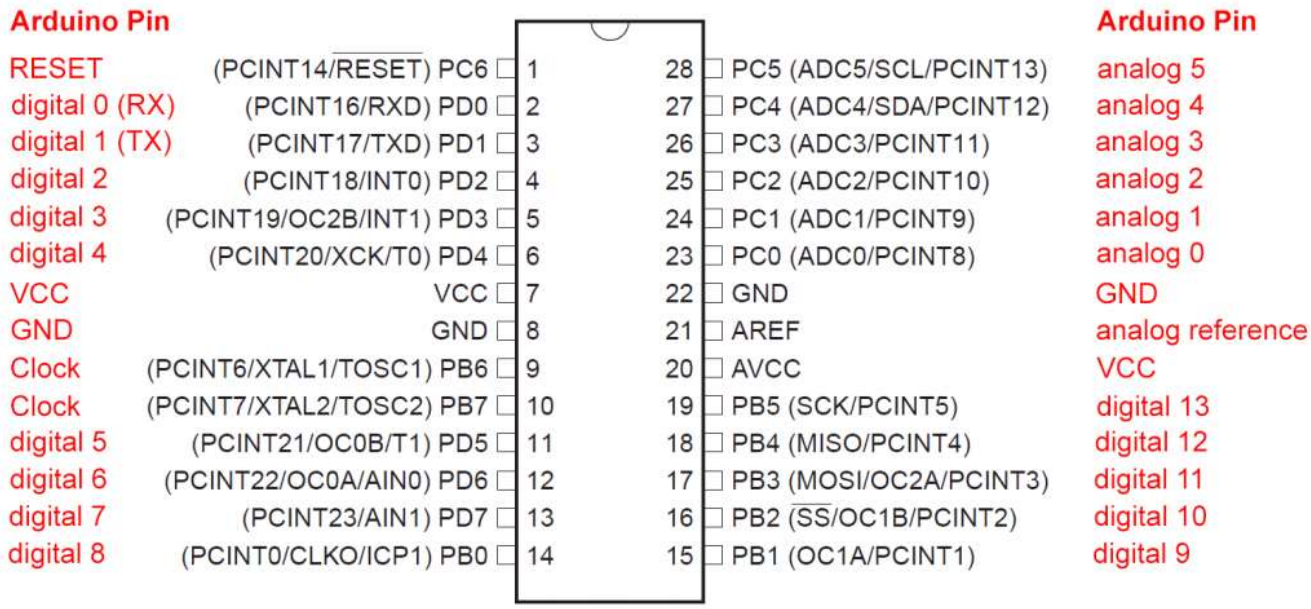
The Output Compare Register OCRxy

For Timer0 and the Timer2 there are the Output Compare Registers OCR0A and OCR0B respectively OCR2A and OCR2B. When you use them, the content in TCNT0 or TCNT2 is constantly compared with the OCR0A/OCR0B or OCR2A/OCR2B register values. What happens in case of a match is specified in the Timer/Counter Control Registers (TCCRxy).

The Output Compare Pins OCxy

The timers/counters each have two pins assigned, OCxA and OCxB. In the Timer/Counter Control Registers (TCCRxy) you specify the status of the pins for a compare match or timer overflow.

The following diagram shows where these pins are located on the ATmega328P and which are the corresponding Arduino pins.



(https://wolles-elektronikkiste.de/wp-content/uploads/2019/12/Pinout-Atmega-328P-vs-Arduino-UNO.png)

Pinout des Atmega328P vs. Arduino UNO

Overview of the settings

Hold out – I have to torment you a little with more information until we get to the examples.

First, you set the Wave Form Generation Mode in the Timer/Counter Control Registers (TCCRxy). The three bits WGMx0, WGMx1 and WGMx2 are responsible for this. Why they had to spread these bits into two registers is a mystery to me. Here is an overview for the Timer2:

Mode	WGM22	WGM21	WGM20	Timer/Counter Mode of Operation	TOP	Update of OCRx at	TOV Flag set on
0	0	0	0	Normal	0xFF	Immediate	MAX
1	0	0	1	PWM, phase correct	0xFF	TOP	BOTTOM
2	0	1	0	CTC	OCRA	Immediate	MAX
3	0	1	1	Fast PWM	0xFF	BOTTOM	MAX
4	1	0	0	Reserved	-	-	-
5	1	0	1	PWM, phase correct	OCRA	TOP	BOTTOM
6	1	1	0	Reserved	-	-	-
7	1	1	1	Fast PWM	OCRA	BOTTOM	TOP

(https://wolles-elektronikkiste.de/wp-content/uploads/2019/12/WGMTable_2.png)

Wave Form Generation Mode Description for Timer2

The Output Compare Modes

The Compare Output Mode, i.e. the effect of the bits COMxyz (with x = 0 or 2, y = A or B, z = 0 or 1) depends on the selected WGM mode.

COM2A1	COM2A0	Description
0	0	Normal port operation, OC2A disconnected.
0	1	Toggle OC2A on Compare Match
1	0	Clear OC2A on Compare Match
1	1	Set OC2A on Compare Match

(https://wolles-elektronikkiste.de/wp-content/uploads/2019/12/CompOutpMode_nonPWM_2.png)

1a) Compare Output Mode for non-PWM (Normal Mode 0), Timer2, COM2Ax

COM2B1	COM2B0	Description
0	0	Normal port operation, OC2B disconnected.
0	1	Toggle OC2B on Compare Match
1	0	Clear OC2B on Compare Match
1	1	Set OC2B on Compare Match

(https://wolles-elektronikkiste.de/wp-content/uploads/2019/12/CompOutpMode_nonPWM_2B.png)

1b) Compare Output Mode for non-PWM (Normal Mode 0), Timer2, COM2Bx

COM2A1	COM2A0	Description
0	0	Normal port operation, OC2A disconnected.
0	1	WGM22 = 0: Normal port operation, OC2A disconnected. WGM22 = 1: Toggle OC2A on Compare Match
1	0	Clear OC2A on Compare Match, set OC2A at BOTTOM (non-inverting mode).
1	1	Set OC2A on Compare Match, clear OC2A at BOTTOM (inverting mode).

(https://wolles-elektronikkiste.de/wp-content/uploads/2019/12/CompOutpMode_FastPWM_2.png)

2a) Compare Output Mode for Fast PWM (WGM modes 3 and 7), Timer2, COM2Ax

COM2B1	COM2B0	Description
0	0	Normal port operation, OC2B disconnected.
0	1	Reserved
1	0	Clear OC2B on Compare Match, set OC2B at BOTTOM (non-inverting mode).
1	1	Set OC2B on Compare Match, clear OC2B at BOTTOM (inverting mode).

(https://wolles-elektronikkiste.de/wp-

content/uploads/2019/12/CompOutpMode_FastPWM_2B.png)

2b) Compare Output Mode for Fast PWM (WGM modes 3 and 7), Timer2, COM2Bx

COM2A1	COM2A0	Description
0	0	Normal port operation, OC2A disconnected.
0	1	WGM22 = 0: Normal port operation, OC2A disconnected. WGM22 = 1: Toggle OC2A on Compare Match
1	0	Clear OC2A on Compare Match when up-counting. Set OC2A on Compare Match when down-counting.
1	1	Set OC2A on Compare Match when up-counting. Clear OC2A on Compare Match when down-counting.

(https://wolles-elektronikkiste.de/wp-content/uploads/2019/12/CompOutpMode_PhaseCorrectPWM_2.png)

3a) Compare Output Mode for Phase Correct PWM (WGM Modes 1 and 5), Timer2, COM2Ax

COM2B1	COM2B0	Description
0	0	Normal port operation, OC2B disconnected.
0	1	Reserved
1	0	Clear OC2B on Compare Match when up-counting. Set OC2B on Compare Match when down-counting.
1	1	Set OC2B on Compare Match when up-counting. Clear OC2B on Compare Match when down-counting.

(https://wolles-elektronikkiste.de/wp-content/uploads/2019/12/CompOutpMode_PhaseCorrectPWM_2B.png)

3b) Compare Output Mode for Phase Correct PWM (WGM Modes 1 and 5), Timer2, COM2Bx

These were the settings for Timer2. The good news: all previous tables of the Timer2 correspond to those of the Timer0. You only have to replace 2 with a 0 where 2 represents the timer.

Prescaler settings

The speed at which the Timer/Counter register TCNTx counts is based on the system clock. At the Arduino UNO, that's 16 MHz. For many applications, this is far too fast. Therefore, there is the prescaler, which causes the counter to be incremented only every n^{th} clock. You can make the settings for this in the Timer/Counter Control Register B (TCCRxB) with the Clock Select Bits (CSx0, CSx1,CSx2). For the Timer2:

CS22	CS21	CS20	Description
0	0	0	No Clock Source
0	0	1	System Clock
0	1	0	Prescaler = 8
0	1	1	Prescaler = 32
1	0	0	Prescaler = 64
1	0	1	Prescaler = 128
1	1	0	Prescaler = 256
1	1	1	Prescaler = 1024

(https://wolles-elektronikkiste.de/wp-content/uploads/2019/12/ClockSelectBit2.png)

Setting the prescaler with the Clock Select bits for Timer2

For the Timer0 the table is different, more about this later. There is a reason why I start with the Timer2. More about that later, too.

The timer in normal mode

Done, now it's time to put it in practice. So complex the settings, so simple the sketches. The confusion will disappear.

We start with normal mode. This is suitable for rather slow applications.

The first sketch in Normal Mode

For the following sketch, you connect an LED to the Arduino Pin 7 (PD7). You set the control register A of the Timer2, i.e. TCCR2A, to 0. This deactivates the OC2A pin. All CS2x bits are set in Timer/Counter Control Register B (TCCR2B). The prescaler is therefore 1024. No WGM2x bit is set, so Normal Mode is active. In the Timer/Counter2 Interrupt Mask Register, the bit for the Timer Overflow Interrupt is set. This means that every time the TCNT2 overflows (> 255), an interrupt is triggered. What you do with the interrupt is defined in the ISR (Interrupt Service Routine). The ISR handles TIMER2_OVF_vect, i.e. the Timer2 Overflow Interrupt. A table of available interrupts can be found in the data sheet.

In this case, the interrupt causes the pin 7 to be inverted. *Note: with DDRD |= (1<<PD7) pin 7 was defined as output. This setting should always be made after the other register settings.*

timer2_normal_mode_minimum_frequenz.ino

```
1.  void setup(){
2.    TCCR2A = 0x00; // Wave Form Generation Mode 0: Normal Mode; OC2A
    disconnected
3.    TCCR2B = (1<<CS22) + (1<<CS21) + (1<<CS20); // prescaler = 1024
```

```
4.    TIMSK2 = (1<<TOIE2); // interrupt when TCNT2 is overflowed
5.    DDRD |= (1<<PD7);  // Portmanipulation: replaces pinMode(7, OUTPUT);
6.  }
7.
8.  void loop() {
9.   // do something else
10. }
11.
12. ISR(TIMER2_OVF_vect){
13.     PORTD ^= (1<<PD7); // toggle PD7
14. }
```

Upload the sketch and see what happens. This is the result on the oscilloscope:

(https://wolles-elektronikkiste.de/wp-content/uploads/2019/12/timer2einfachst-1.png)

Minimum frequency in Normal Mode. The frequency refers to a HIGH/LOW pair.

If you transfer the sketch to the Timer0, you will receive an error message:

```
wiring.c.o (symbol from plugin): In function `__vector_16':

(.text+0x0): multiple definition of `__vector_16'
```

(https://wolles-elektronikkiste.de/wp-content/uploads/2019/12/Fehlermeldung.png)

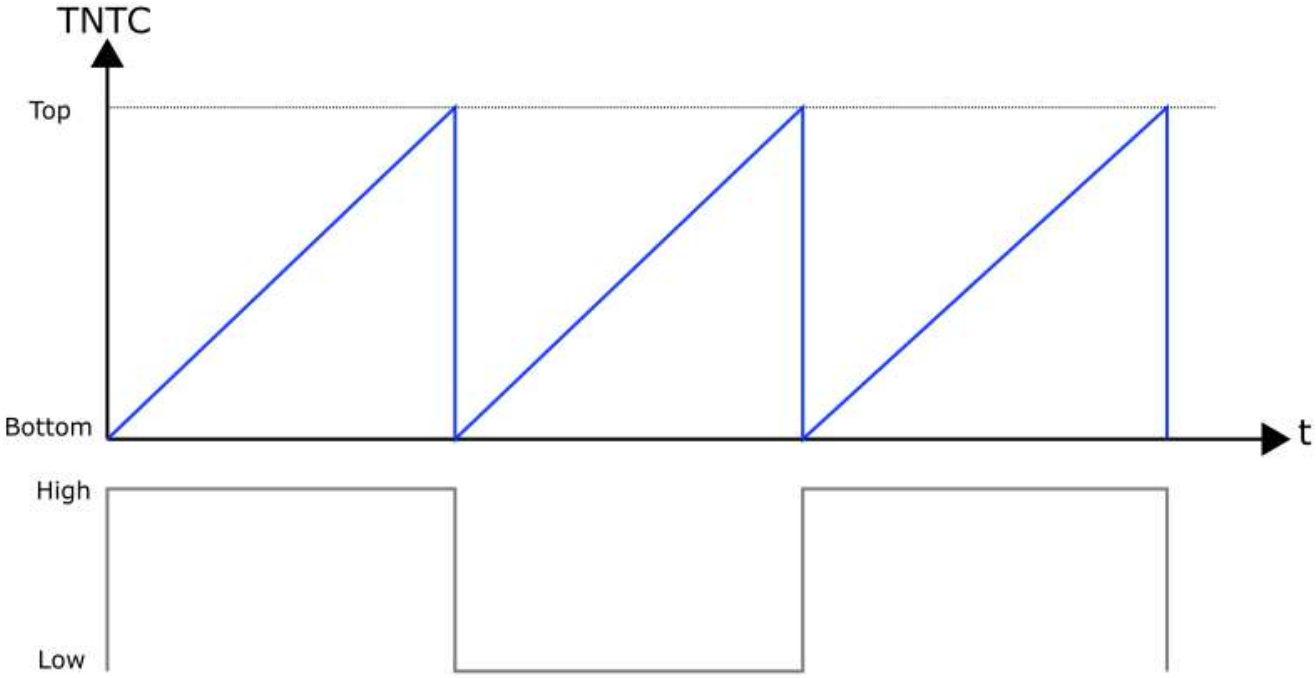
The problem is that in the Arduino environment the Timer0_OVF_vect is used for other things. That's why I started with the Timer2. In other environments, such as Atmel Studio, the problem does not exist. At the end of the article, I will come back to that.

Calculation of the frequency

You can see that the LED flashes very fast when using the last sketch. The Arduino clocks 16 million times per second. Due to the prescaler setting, TCNT2 is increased every 1024th clock. TNCT2 starts at 0 and overflows after **255.** That's **256** steps (like a `for(i = 0; i<256; i++)`). Therefore, the general formula for frequency f is:

$$f = \frac{system\_clock}{prescaler \cdot 256} = \frac{16000000}{1024 \cdot 256} = 61.035...[\mathrm{s}^{-1}]$$

Graphically, it looks like this:



(https://wolles-elektronikkiste.de/wp-content/uploads/2019/12/Grafik_Toggle.png)

Graph 1: TNTC vs. pin status

TCNT starts at Bottom (here: 0) and counts to Top. In the WGM table you find for Normal Mode: Top = 0xFF and TOV flag set on MAX (=TOP). PD7 inverts with each overflow.

Further downgrade of the frequency

So even with the maximum prescaler, the frequency for a blinksketch is still very high. To further slow down the process, we introduce a scale factor into the ISR (counter = 60). This results in a frequency of about 1. Strictly speaking, 61,035... / 60 [s^{-1}].

timer2_normal_mode_verlangsamt.ino

```
1.  void setup(){
2.    TCCR2A = 0x00; // Wave Form Generation Mode 0: Normal Mode, OC2A
    disconnected
```

```
  TCCR2B = (1<<CS22) + (1<<CS21) + (1<<CS20); // prescaler = 1024
  TIMSK2 = (1<<TOIE2); // interrupt when TCNT2 is overflowed
  DDRD |= (1<<PD7);
}

void loop() {
 // do something else
}

ISR(TIMER2_OVF_vect){  // Interrupt Service Routine
  static int counter = 0;
  counter++;
  if(counter==60){
    PORTD ^= (1<<PD7);
    counter = 0;
  }
}
```

Setting an exact frequency

In the next step, we want to set a frequency of exactly 1 Hz. For this, we use the option to define a starting value for the TCNTx register. Then it's only `256 - starting_value` steps to the overflow. The general formula is:

$$f_{desired} \cdot scalefactor = \frac{system\_clock}{prescaler \cdot (256 - starting\_value)}$$

With $f_{desired}$ = 1 results:

$$starting\_value = 256 - \frac{system\_clock}{prescaler \cdot scalefactor}$$

The system clock is 16 MHz. But it's still an equation with three unknowns. However, we know that the prescaler can take only certain values, that the starting value must be less than 256 and it must be an integral number. So let's try a bit:

$$starting\_value = 256 - \frac{15625}{scalefactor} \quad with\ prescaler = 1024$$

$$starting\_value = 256 - \frac{62500}{scalefaktor} \quad with\ prescaler = 256$$

$$starting\_value = 256 - \frac{250000}{scalefaktor} \quad with\ prescaler = 64$$

The prescaler 1024 delivers “crooked” values. With a prescaler of 256, it looks better. If we take a scale factor of 500, we get a starting value of 131. With a prescaler of 64 and a scale factor of 1000, a starting value of 6 results. Both are good.

If you don’t want to calculate, then you can use calculators, which you find e.g. here (http://evolutec.publicmsg.de/index.php?menu=software&content=prescalertools) or here (https://eleccelerator.com/avr-timer-calculator/). Although these calculators do not consider scale factors, they are still a help.

The previous sketch is slightly changed. TCNT2 must be set back to the start value at the beginning and after each overflow.

timer2_normal_mode_genau_1Hz.ino

```
byte counterStart = 131;  // alternative: 6
unsigned int scaleFactor = 500; // alternative: 1000

void setup(){
  TCCR2A = 0x00; // OC2A and OC2B disconnected; Wave Form Generation Mode 0: Normal Mode
  TCCR2B = (1<<CS22) + (1<<CS21); // prescaler = 256
  // TCCR2B = (1<<CS22); // prescaler = 64;
  TIMSK2 = (1<<TOIE2); // interrupt when TCNT2 is overflowed
  TCNT2 = counterStart;
  DDRD |= (1<<PD7);
}

void loop() {
 // do something else
}

ISR(TIMER2_OVF_vect){
  static int counter = 0;
  TCNT2 = counterStart;
  counter++;
  if(counter==scaleFactor){
    PORTD ^= (1<<PD7);
    counter = 0;
  }
}
```

An application in normal mode: asynchronous LEDs

You may have noticed that the main loop was empty in the previous sketches. The control of the flashing LED happens in the background. If the blink code was part of the main loop, then all additional code has to be programmed “around” that. This can become quite challenging, if other time-dependent tasks shall take place in parallel.

As a simple example, we let two LEDs flash asynchronously in the next sketch. Try programming this without a timer.

timer2_normal_mode_asynchrone_LEDs.ino

```
byte counterStart = 131;  // alternative: 6
unsigned int scaleFactor = 500; // alternative: 1000

void setup(){
  TCCR2A = 0x00; // OC2A and OC2B disconnected; Wave Form Generator: Normal Mode
  TCCR2B = (1<<CS22) + (1<<CS21); // prescaler = 256
  // TCCR2B = (1<<CS22); // prescaler = 64;
  TIMSK2 = (1<<TOIE2); // interrupt when TCNT2 is overflowed
  TCNT2 = counterStart;
  DDRD |= (1<<PD7) + (1<<PD6); // Pin 6 und Pin 7 als Ausgang
}

void loop() {
 PORTD ^= (1<<PD6);
 delay(723);
}

ISR(TIMER2_OVF_vect){
  static int counter = 0;
  TCNT2 = counterStart;
  counter++;
  if(counter==scaleFactor){
    PORTD ^= (1<<PD7);
    counter = 0;
  }
}
```

The timer in CTC mode

In “Clear Timer on Compare Match” mode, or CTC for short, TCNTx is reset to zero not after 256 steps, but after reaching the value stored in OCRxA. Again we want to have an LED flashing every second. The formula for frequency calculation is:

$$f_{desired} \cdot scalefactor = \frac{system\_clock}{prescaler \cdot (1 + Top)}$$

And why (1 + Top) and not just Top? Quite simply: because the 0 counts! Top is therefore with $f_{desired}$ = 1:

$$Top = \frac{system\_clock}{prescaler \cdot scalefactor} - 1$$

A system clock of 16 MHz, a prescaler of 256 and a scale factor of 500 results in a top value of 124. We write this value into OCRxA register, here: OCR2A = 124.

According to the WGM table, we have to set the bit WGM21 for the CTC mode. For the prescaler 256 we set CS22 and CS21. In the Timer/Counter Interrupt Mask Register we set OCIE2A (Output Compare Interrupt Enable A, Timer2) because this time there is no timer overflow, but a compare match. Accordingly, we also have to change the ISR routine and pass TIMER2_COMPA_vect as parameter.

timer2_ctc_mode_genau_1Hz.ino

```
unsigned int scaleFactor = 500;

void setup(){
  TCCR2A = (1<<WGM21); // Wave Form Generation Mode 2: CTC, OC2A disconnected
  TCCR2B = (1<<CS22) + (1<<CS21) ; // prescaler = 256
  TIMSK2 = (1<<OCIE2A); // interrupt when Compare Match with OCR2A
  OCR2A = 124;
  DDRD |= (1<<PD7);

}

void loop() {
 // do something else
}

ISR (TIMER2_COMPA_vect){  // Interrupt Service Routine
  static int counter = 0;
  counter++;
  if(counter==scaleFactor){
    PORTD ^= (1<<PD7); //
    counter = 0;
  }
}
```

If we compare normal mode with CTC mode, we have counted in one case from a certain TCNT value to 255, and in the other case from 0 to the value stored in OCRA.

The timer in Fast PWM mode

In Fast PWM mode, you usually work with the pins associated with the timer. This is important for the

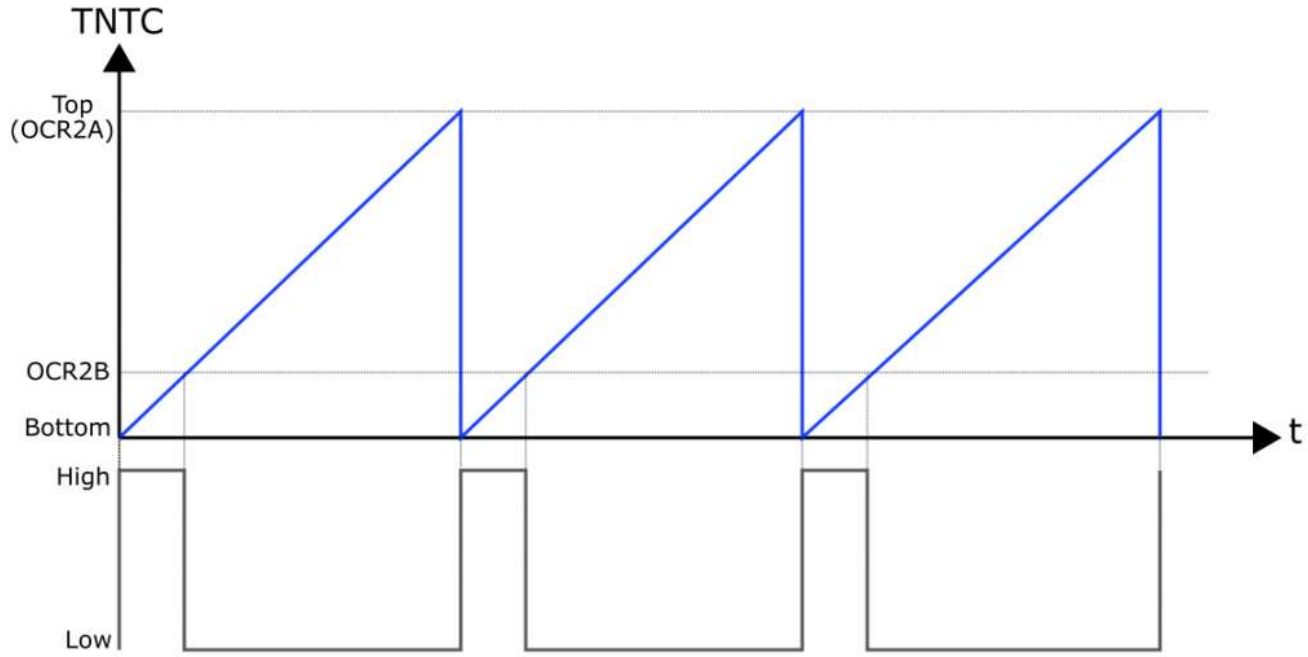
- Timer0: OC0A (=PD6, Arduino Pin 6) / OC0B (=PD5, Arduino Pin 5)
- Timer2: OC2A (=PB3, Arduino Pin 11) / OC2B (=PD3, Arduino Pin 3)

The PWM mode works in Mode 3 with a timer overflow after 255 (0xFF). In Mode 7, PWM mode works with a Compare Match. Top is the value stored in OCRxA.

Fast PWM on OC2B

As an example, let's take Mode 7. The target is a square wave signal with a frequency of 1 kHz at OC2B. Per period, the signal should be HIGH 20% of the time and 80% LOW. Another expression for this is: the duty cycle is 20%.

We set the COM2B1 bit for this purpose. According to the table, this means: "clear OC2B at Compare Match, set OC2B at BOTTOM". The Compare Match refers to the value stored in OCR2B. Graphically, it looks like this:



(https://wolles-elektronikkiste.de/wp-content/uploads/2019/12/PWM_20_80_OC2B.png)

Graph 2: PWM signal on OC2B

First, we have to do some calculations again. We need the top value for the frequency and the value for OCR2B for the duty cycle. Because of the high frequency, we don't need a scale factor. The following applies:

$$f_{desired} = \frac{system\_clock}{prescaler \cdot (1 + Top)}$$

$$Top = \frac{16000}{prescaler} - 1 \quad with \quad clock = 16000000 \quad and \quad f = 1000$$

A prescaler of 64 results in 249 for top. That's 250 steps. One-fifth of it is 50. Since the counter starts at 0, OCR2B is 49, at least theoretically. In practice, you have to try it out. I have hit the desired signal better with the pairing 249 / 50. Of course, it also depends on how exactly the microcontroller clocks.

Here's what the sketch looks like:

timer2_fast_pwm_mode_80_20_an_OC2B.ino

```
1.  // Period = 1 ms => Frequenz = 1kHz
2.  void setup(){
3.    // WGM22/WGM21/WGM20 all set -> Mode 7, fast PWM
4.    TCCR2A = (1<<COM2B1) + (1<<WGM21) + (1<<WGM20); // Set OC2B at bottom,
    clear OC2B at compare match
5.    TCCR2B = (1<<CS22) + (1<<WGM22); // prescaler = 64;
6.    OCR2A = 249;
7.    OCR2B = 49;
8.    DDRD |= (1<<PD3);
9.  }
10.
11.  void loop() {}
```

… and this is what it looks like on the oscilloscope:

(https://wolles-elektronikkiste.de/wp-content/uploads/2019/12/timer2_Fast_PWM_an_OC2B.png)

PWM signal on OC2B

Fast PWM on OC2A

For a PWM signal on OC2A, Mode 7 is of limited suitability, as the Compare Match is equal to Top. There is no possibility to divide the signal into a HIGH and a LOW part. For better understanding you might look into the corresponding tables. In Mode 3, the timer counts up to 255, so any duty cycles can be realized there. The disadvantage, however, is that not any frequency can be set because Top is fixed.

PWM signals with 50% duty cycle, on the other hand, are easy to set on OC2A. This is what you do with the combination: Mode 7 / set COM2A0. According to the table, this means: Toggle O2CA on Compare Match. The period is doubled (frequency is halved).

timer2_fast_pwm_mode_50_50_OC2A.ino

```
1.  // Period = 2 ms / Frequenz = 500 Hz.
2.  void setup(){
3.    // WGM22/WGM21/WGM20 all set -> Mode 7, fast PWM, TOP = OCR2A
4.    TCCR2A = (1<<COM2A0) + (1<<WGM21) + (1<<WGM20); // Toggle OC2A at
    compare match
5.    TCCR2B = (1<<CS22) + (1<<WGM22); // prescaler = 64;
6.    OCR2A = 249;
7.    DDRB |= (1<<PB3); // PB3 = OC2A = Arduino Pin 11
8.  }
9.
10. void loop() {
11. }
```

AnalogWrite – a PWM application

You can see on the oscilloscope that analogWrite is not an analog but a PWM signal:

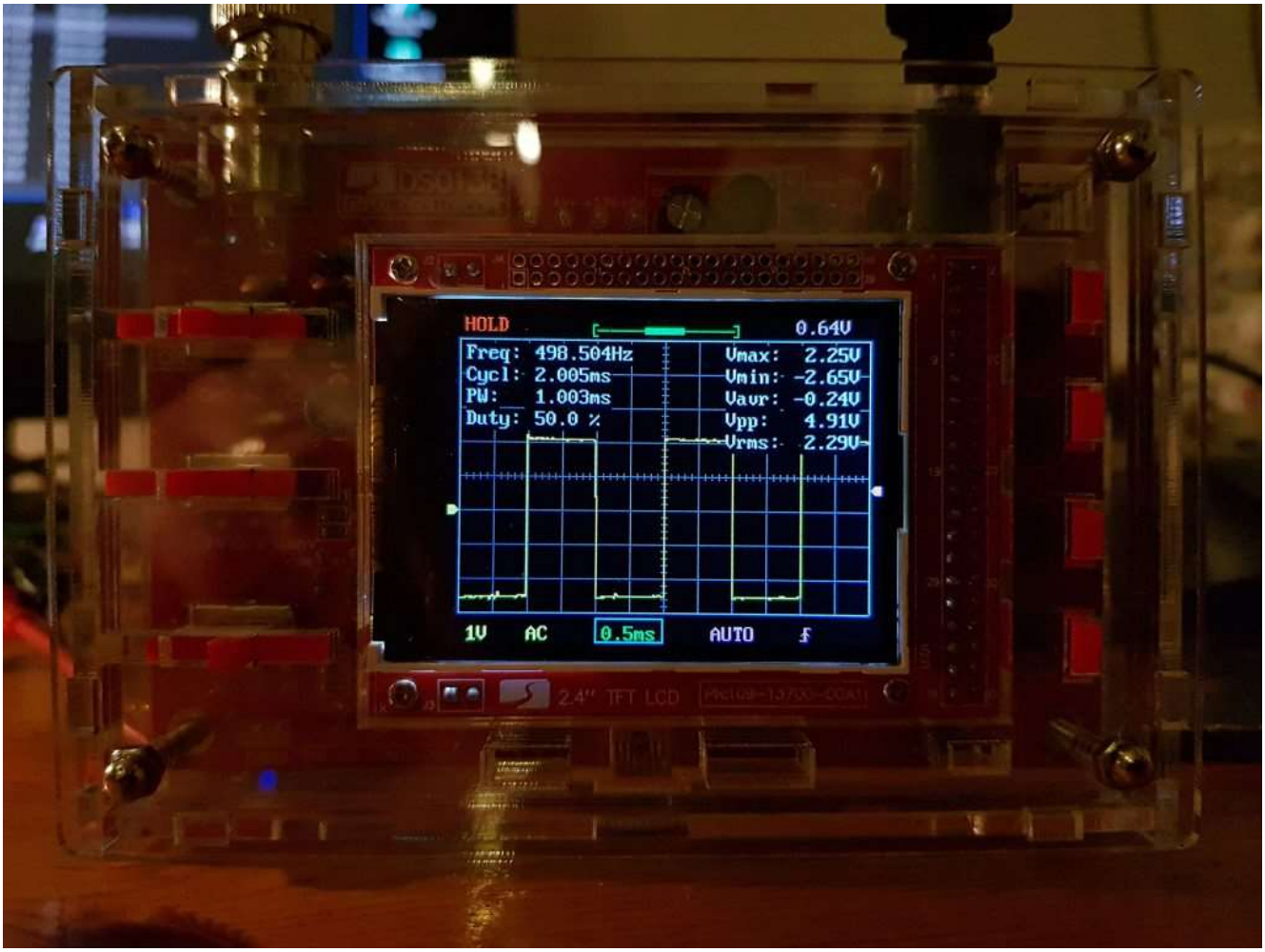
(https://wolles-elektronikkiste.de/wp-content/uploads/2019/12/analogWrite.png)

analogWrite(*pin*, 50) on the oscilloscope

You don't have an oscilloscope?

Some may be frustrated that they cannot check the result of the Fast PWM sketches because they do not have an oscilloscope. Three suggestions:

- also in Fast PWM mode you can work with Compare Match Interrupts, insert a counter in the ISR and observe the PWM signal in slow motion. You can even set OCIE2A and OCIE2B and use two ISR routines (TIMER2_COMPA_vect / TIMER2_COMPB_vect).
- uses the technology explained in my post about IR remote controls (https://wolles-elektronikkiste.de/en/ir-remote-controls) to analyze fast signals
- buy a DSO 138 oscilloscope for less than 30 euros. Search for it on Amazon or eBay. This device works amazingly well:



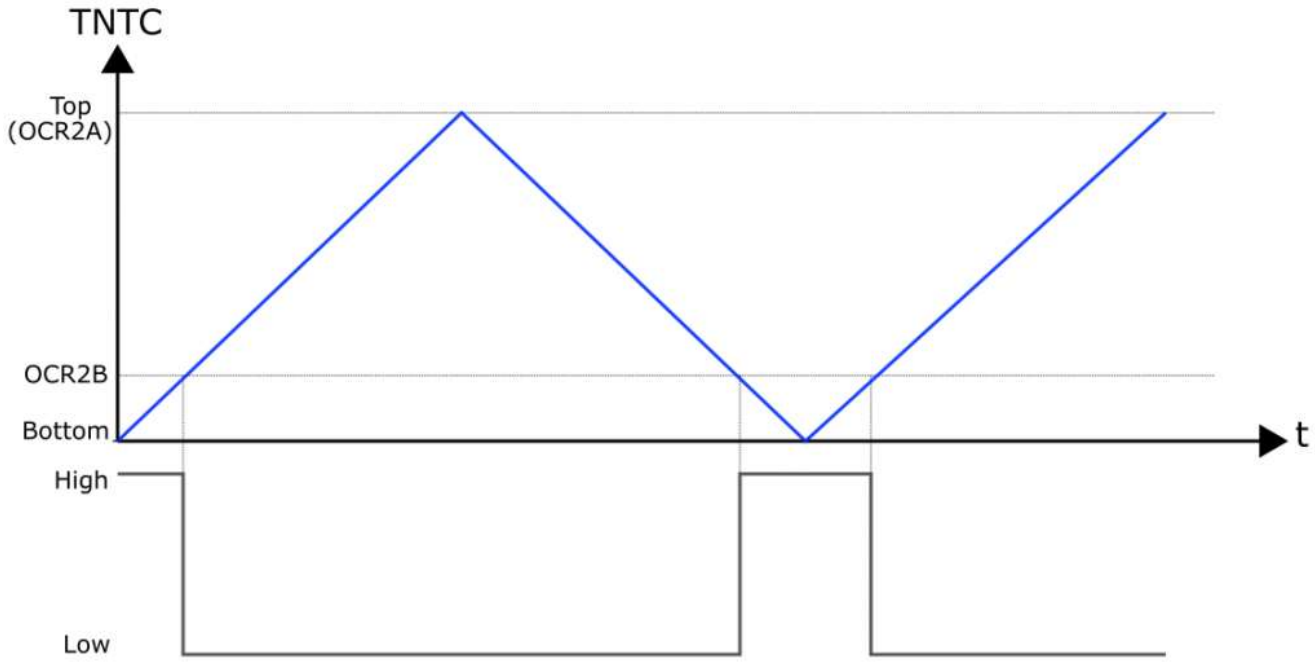
(https://wolles-elektronikkiste.de/wp-content/uploads/2019/12/Mini-Oskar_klein.jpg)

DSO 138 oscilloscope in action

The timer in Phase Correct PWM Mode

In phase-correct PWM mode (Phase Correct PWM Mode), the counter counts up from BOTTOM to TOP and then down again to BOTTOM. As an example, let's take the combination of Mode 5 and COM2B1 set, i.e.: "Clear OC2B when up-counting, set OC2B when down-counting".

Graphically, this looks like this:



(https://wolles-elektronikkiste.de/wp-content/uploads/2019/12/Grafik_phase_correct_PWM.png)

Graph 3: Phase Correct PWM with Timer2 on OC2B

This halves the frequency compared to the Fast PWM method.

Phase correct means that the reversal points from HIGH to LOW (or vice versa) are at the same distance before and after the bottom or top of the timer. This results in a symmetric signal, even if the OCR2B Compare value is changed dynamically. A very helpful animation, which shows the difference to the Fast PWM, can be found here (http://www.aquaticus.info/pwm-modes/). The phase-correct PWM is mainly used for motor controllers. If you want to know more about the PWM modes, see e.g. here (http://modelleisenbahn-steuern.de/controller/atmega8/12-11-4-phasenkorrekt-pwm-modus-atmega8.htm). There is another PWM mode, namely the “phase- and frequency-correct PWM” mode. This subject will be treated part 2 (https://wolles-elektronikkiste.de/en/timer-and-pwm-part-2-16-bit-timer1) (16 bit timer). There I will also go into a little more detail about the differences of the PWM modes.

And here’s a sketch example:

timer2_phase_correct_pwm_mode_80_20_an_OC2B.ino

```
1.  // Period = 2 ms
2.  void setup(){
3.    // WGM22/WGM20 all set -> Mode 5, phase correct PWM
4.    TCCR2A = (1<<COM2B1) + (1<<WGM20); // Set OC2A at bottom, clear OC2B at
    compare match
5.    TCCR2B = (1<<CS22) + (1<<WGM22); // prescaler = 64;
6.    OCR2A = 249;
```

```
    OCR2B = 49;
    DDRD |= (1<<PD3);
}

void loop() { }
```

Timer 0 vs. Timer 2

As mentioned above, the structure of Timer0 and Timer2 is very similar. You can see the main difference in the Clock Select Bit table for the timer0:

CS02	CS01	CS00	Description
0	0	0	No Clock Source
0	0	1	System Clock
0	1	0	Prescaler = 8
0	1	1	Prescaler = 64
1	0	0	Prescaler = 256
1	0	1	Prescaler = 1024
1	1	0	External clock source on T0 pin. Clock on falling edge.
1	1	1	External clock source on T0 pin. Clock on rising edge.

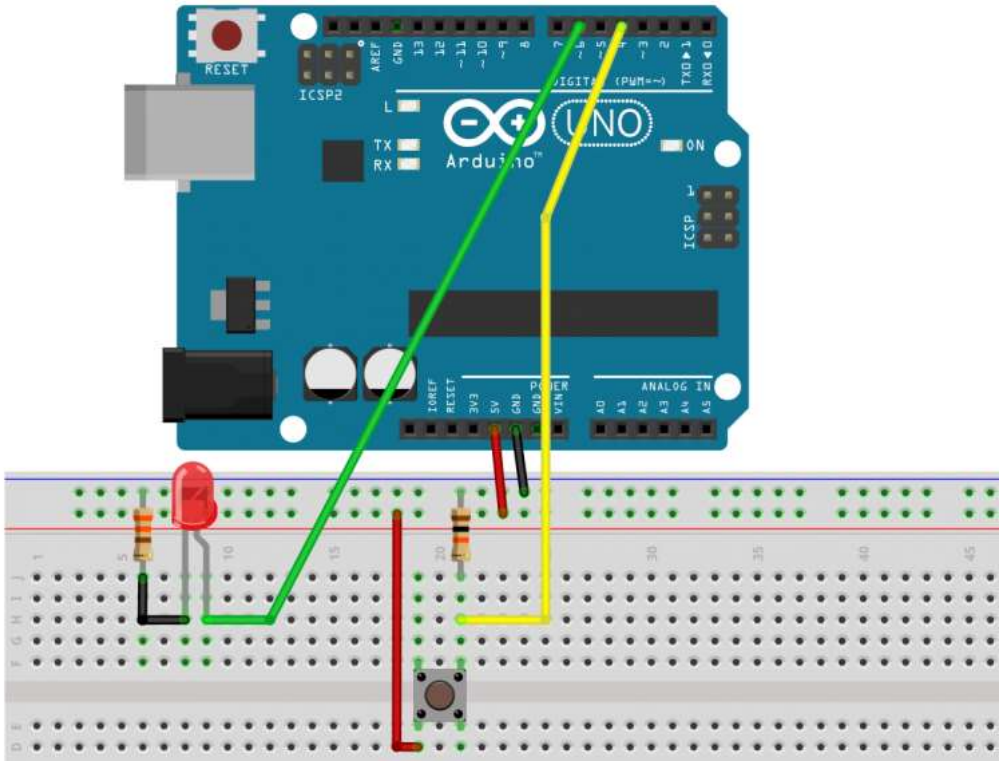
(https://wolles-elektronikkiste.de/wp-content/uploads/2019/12/ClockSelectBit0.png)

Prescaler / Clock Select with the Clock Select Bits for Timer0

There are fewer prescalers to choose from, instead you can use an external clock here. You connect it to T0 (PD4, Arduino pin 4). In addition, you can choose whether to count at the rising or falling edge.

Simple example of external clocks

As a clock, we use a simple push button. The OC0A output pin (PD6, Arduino Pin 6) is connected to an LED.



(https://wolles-elektronikkiste.de/wp-content/uploads/2019/12/Ext_Clock_an_T0.png)

Timer0 with push button as external clock

Now let's choose Mode 7 and set all CS0x bits (Clock on rising edge). OCR0A we set to 10.

Timer0_External_Clock_TOGGLE_OC0A.ino

```
 void setup(){
   TCCR0A = (1<<COM0A0) + (1<<WGM01) + (1<<WGM00); // WGM 7: fast PWM; since WGM02 = 1 --> Toggle OC0A on Compare Match;
   TCCR0B = (1<<CS02) + (1<<CS01) + (1<<CS00) + (1<<WGM02);  // External clock source on T0 (rising edge)
   OCR0A = 10;
   DDRD |= (1<<PD6);
 }

 void loop() {
 }
```

Theoretically, the LED should switch on or off every eleventh press of the push button. The LED switches accordingly earlier by pushing the button.

Counting external events in the background can be very useful for certain applications. By the way, there are also special counter ICs, which I have described here. (https://wolles-elektronikkiste.de/en/counter-ics)

If you use a clock quartz as a clock set, you can build a quartz watch. This is described in detail here. (https://www.mikrocontroller.net/articles/AVR-GCC-Tutorial/Die_Timer_und_Z%C3%A4hler_des_AVR#Timer2_im_Asynchron_Mode) Pretty cool.

Using Microchip (Atmel) Studio

At the very beginning I mentioned that the Timer0 Overflow Interrupt is not accessible in the Arduino environment. If you use the free software Microchip Studio (formerly known as Atmel Studio), you do not have this limitation. An introduction to Microchip Studio can be found here (https://wolles-elektronikkiste.de/en/atmel-studio-7-an-introduction). You have to invest some time to get used to it, but it's worth it from my point of view.

The sketch for the two asynchronously flashing LEDs, transferred to the Timer0, looks like this in Atmel Studio:

Asynchronous_LEDs.cpp

```
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
uint8_t counterStart = 131;
uint16_t scaleFactor = 500;

int main(void)
{
  TCCR0A = 0x00; // OC0A and OC0B disconnected; Wave Form Generation Mode 0: Normal Mode
  TCCR0B = (1<<CS02); // prescaler = 256
  TIMSK0 = (1<<TOIE0); // interrupt when TCNT0 is overflowed
  TCNT0 = counterStart;
  DDRB |= (1<<PB1) + (1<<PB0);
  sei();  // activate interrupts

  while (1)
  {
    PORTB ^= (1<<PB0);
    _delay_ms(723);
  }
}


ISR(TIMER0_OVF_vect)
{
  static int counter = 0;
  TCNT0 = counterStart;
  counter++;
  if(counter==scaleFactor)
```

```
    {
      PORTB ^= (1<<PB1);
      counter = 0;
    }
  }
```

Final words

So, I hope one or the other has persevered up to here. Personally, I found it very exciting when I had my first experiences with the timers and got a deeper insight into the inner structure of the ATmega328P.

What follows is, as announced above, part 2 (https://wolles-elektronikkiste.de/en/timer-and-pwm-part-2-16-bit-timer1), which deals with the 16 bit Timer1.

Acknowledgement

I owe the hourglass in the post image “derGestalterCottbus (https://pixabay.com/de/users/derGestalterCottbus-5321869/?utm_source=link-attribution&utm_medium=referral&utm_campaign=image&utm_content=3308818)” on Pixabay. (https://pixabay.com/de/?utm_source=link-attribution&utm_medium=referral&utm_campaign=image&utm_content=3308818)

Posted in Boards and Microcontrollers (https://wolles-elektronikkiste.de/en/category/boards-and-microcontrollers) Tagged compare match (https://wolles-elektronikkiste.de/en/tag/compare-match-en), counter (https://wolles-elektronikkiste.de/en/tag/counter-en), CTC (https://wolles-elektronikkiste.de/en/tag/ctc-en-2), duty cycle (https://wolles-elektronikkiste.de/en/tag/duty-cycle-en), fast PWM (https://wolles-elektronikkiste.de/en/tag/fast-pwm-en-2), ISR (https://wolles-elektronikkiste.de/en/tag/isr-en), normal mode (https://wolles-elektronikkiste.de/en/tag/normal-mode-en), output compare (https://wolles-elektronikkiste.de/en/tag/output-compare-en-2), phase-correct PWM (https://wolles-elektronikkiste.de/en/tag/phase-correct-pwm-en-2), phasenkorrekte PWM (https://wolles-elektronikkiste.de/en/tag/phasenkorrekte-pwm-en), prescaler (https://wolles-elektronikkiste.de/en/tag/prescaler-en), PWM (https://wolles-elektronikkiste.de/en/tag/pwm-en), timer (https://wolles-elektronikkiste.de/en/tag/timer-en), timer 0 (https://wolles-elektronikkiste.de/en/tag/timer-0-en-2), timer 2 (https://wolles-elektronikkiste.de/en/tag/timer-2-en-2), TIMER0_OVF_vect (https://wolles-elektronikkiste.de/en/tag/timer0_ovf_vect-en), TIMER2_OVF_vect (https://wolles-elektronikkiste.de/en/tag/timer2_ovf_vect-en-2), wave form generation mode (https://wolles-elektronikkiste.de/en/tag/wave-form-generation-mode-en)

← Logical operations and port manipulation (https://wolles-elektronikkiste.de/en/logical-operations-and-port-manipulation)

Timer and PWM – Part 2 (16 Bit Timer1) → (https://wolles-elektronikkiste.de/en/timer-and-pwm-part-2-16-bit-timer1)

2 thoughts on “Timer and PWM – Part 1 (8-Bit Timer0/2)”

Cee Wee **(https://wolles-elektronikkiste.de/en/timer-and-pwm-part-1-8-bit-timer0-2#comment-3374)** *July 8, 2021*

A really comprehensive write-up for atmega328p timer and pwm, far better than reading from the datasheet which I found it hard to get a good big picture.

Reply

Wolfgang Ewald **(https://wolles-elektronikkiste.de/en/timer-and-pwm-part-1-8-bit-timer0-2#comment-3375)** *July 8, 2021*

Thank you!

Reply

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name *

Email *

Website

☐ **Ja, informiere mich über neue Beiträge! Inform me about new posts! You will NOT be notified about new or follow-up comments!**

Post Comment

ABO / SUBSCRIPTION?

Vorname

E-Mail *

Zur Datenschutzerklärung. (http://wolles-elektronikkiste.de/privacy-policy)

Abonniere / subscribe!

RECENT POSTS

- Strain gauges (https://wolles-elektronikkiste.de/en/strain-gauges)
- Programming the ESP32 with MicroPython (https://wolles-elektronikkiste.de/en/programming-the-esp32-with-micropython)
- MicroPython – Switching from Arduino (https://wolles-elektronikkiste.de/en/micropython-switching-from-arduino)
- ICM-20948 9-Axis Sensor Part II (https://wolles-elektronikkiste.de/en/icm-20948-9-axis-sensor-part-ii)
- ICM-20948 9-Axis Sensor Part I (https://wolles-elektronikkiste.de/en/icm-20948-9-axis-sensor-part-i)

RECENT COMMENTS

- Wolfgang Ewald on ADXL345 – The Universal Accelerometer – Part 1 (https://wolles-elektronikkiste.de/en/adxl345-the-universal-accelerometer-part-1#comment-5776)
- Wolfgang Ewald on INA219 Current and Power Sensor (https://wolles-elektronikkiste.de/en/ina219-current-and-power-sensor#comment-5775)
- Ivan on INA219 Current and Power Sensor (https://wolles-elektronikkiste.de/en/ina219-current-and-power-sensor#comment-5774)
- Andrea on ADXL345 – The Universal Accelerometer – Part 1 (https://wolles-elektronikkiste.de/en/adxl345-the-universal-accelerometer-part-1#comment-5758)

- Wolfgang Ewald on ADXL345 – The Universal Accelerometer – Part 1 (https://wolles-elektronikkiste.de/en/adxl345-the-universal-accelerometer-part-1#comment-5484)

ARCHIVES

Select Month

THEMEN

- Boards and Microcontrollers (https://wolles-elektronikkiste.de/en/category/boards-and-microcontrollers)
- Other parts (https://wolles-elektronikkiste.de/en/category/other-parts)
  - Port expansion (https://wolles-elektronikkiste.de/en/category/other-parts/port-expansion)
- Other stuff (https://wolles-elektronikkiste.de/en/category/other-stuff)
- Projects (https://wolles-elektronikkiste.de/en/category/projects)
- Sensors (https://wolles-elektronikkiste.de/en/category/sensors)
  - Acceleration (https://wolles-elektronikkiste.de/en/category/sensors/acceleration)
  - Current, voltage (https://wolles-elektronikkiste.de/en/category/sensors/current-voltage)
  - Distance, light, movement (https://wolles-elektronikkiste.de/en/category/sensors/distance-light-movement)
- Software and tools (https://wolles-elektronikkiste.de/en/category/software-and-tools)
- Sound (https://wolles-elektronikkiste.de/en/category/sound-en)
- Uncategorized (https://wolles-elektronikkiste.de/en/category/uncategorized)
- Wireless (https://wolles-elektronikkiste.de/en/category/wireless)
  - 433 MHz (https://wolles-elektronikkiste.de/en/category/wireless/433-mhz-en)
  - Bluetooth (https://wolles-elektronikkiste.de/en/category/wireless/bluetooth-en)
  - Infrared (https://wolles-elektronikkiste.de/en/category/wireless/infrared)
  - WLAN (https://wolles-elektronikkiste.de/en/category/wireless/wlan-en)
- wireless (https://wolles-elektronikkiste.de/en/category/wireless-en-2)

FORMALES / FORMAL STUFF

- Legal notice (https://wolles-elektronikkiste.de/en/legal-notice)

ANDERES / OTHER STUFF

- Home (https://wolles-elektronikkiste.de/?lang=en)
- Visit me on Github (https://github.com/wollewald)