AVR 8-bit Microcontrollers

AVR130: Setup and Use of AVR Timers

APPLICATION NOTE

Introduction

This application note describes how to use the different timers of the Atmel® AVR®. The intention of this document is to give a general overview of the timers, show their possibilities, and explain how to configure them. The code examples will make this clearer and can be used as guidance for other applications. The Atmel ATmega328PB is used as an example in this document to explain the timers.

Starting from a general overview of the timers, several examples will show how the timers work and how they are configured. Experienced users can start directly with the section Setting Up the Timers. The last section is a short description of the PWM mode. A zip file is available with this application note which contains C and Assembly code for all these examples. It can be downloaded from the Atmel Website.

Additional information is available in the data sheets and application notes for specific devices.

Features

- Description of Timer/Counter events
- Timer/Counter Event notification
- Clock options
- Example Code for various timers
  - Overflow interrupt
  - Input Capture interrupt
  - Asynchronous operation
  - Compare Match interrupt
- PWM Basics

Table of Contents

1. Overview

In principle, a timer is a simple counter. Its advantage is that the input clock and operation of the timer is independent of the program execution. The deterministic clock makes it possible to measure time by counting the elapsed cycles and take the input frequency of the timer into account.

1.1. About Timers

Generally, the megaAVR® microcontrollers have two 8-bit and one 16-bit timer. A timer with 16-bit resolution is certainly more flexible to use than one with 8-bit resolution. However, the saying “bigger is better” does not necessarily apply to the microcontroller world. For many applications, it is sufficient to have 8-bit resolution. Using a higher resolution means a larger program overhead, which costs processing time and should be avoided in speed optimized code. It also means higher device cost.

Because of the flexibility of the AVR timers, they can be used for different purposes. The number of timers determines the amount of independent configurations. In the following, the different configuration options will be described more closely.

1.2. Timer Events

The timer of the AVR can be specified to monitor several events. Status flags in the TIMSK register show if an event has occurred. The ATmega328PB can be configured to monitor up to three events for the 8-bit timers TC0 and TC2. It also has three 16-bit timers (TC1, TC3, and TC4) each of them support four events.

Timer Overflow

A timer overflow means that the counter has counted up to its maximum value and is reset to zero in the next timer clock cycle. The resolution of the timer determines the maximum value of that timer. There are two timers with 8-bit resolution and three timers with 16-bit resolution on the ATmega328PB. The maximum value a timer can count to can be calculated by Equation 1. Res is here the resolution in bits.

$$MaxVal = 2^{Res} - 1 \qquad (1)$$

The timer overflow event causes the Timer Overflow Flag (TOVx) to be set in the Timer Interrupt Flag Register (TIFRn).

Compare Match

In cases where it is not sufficient to monitor a timer overflow, the compare match interrupt can be used. The Output Compare Register (OCRx) can be loaded with a value [0 .. MaxVal] which the TCNTn will be compared against in every timer cycle. When the timer reaches the compare value, the corresponding Output Compare Flag (OCFx) in the TIFRn register is set. The Timer can be configured to clear the count register to zero on a compare match.

Related output pins can be configured to be set, cleared, or toggled automatically on a compare match. This feature is very useful to generate square wave signals of different frequencies. It offers a wide range of possibilities, which makes it possible to implement a DAC. The PWM mode is a special mode which is even better suited for wave generation. See the device datasheet for details.

Input Capture

Timers in AVR have input pins to trigger the input capture event. A signal change at such a pin causes the timer value to be read and saved in the Input Capture Register (ICRx). At the same time the Input Capture Flag (ICFx) in the TIFRn will be set. This is useful to measure the width of external pulses.

2. Timer Event Notification

The timer operates independently of the program execution. For each timer event there is a corresponding status flag in the Timer Interrupt Flag Register (TIFRn). The occurrence of timer events require a notification of the processor to trigger the execution of corresponding actions. This is done by setting the status flag of the event which occurred.

There are three different ways to monitor timer events and respond to them:

1. Constantly polling of status flags – interrupt flags and execution of corresponding code.
2. Break of program flow and execution of Interrupt Service Routines (ISR).
3. Changing the level of output pins automatically.

Polling of Interrupt Flags

This method makes use of the fact that the processor marks the timer events by setting the corresponding interrupt flags. The main program can frequently check the status of these flags to see if one of these events occurred. This requires some program overhead, which will cost additional processing time. The advantage of this solution is the very short response time when tight loops are used.

The assembler implementation for the Timer0 can look like the following code example. These three lines of code have to be located in the main loop so that they are executed frequently.

```
loop:               ; Label
lds     r16,TIFRn   ; Load TIFRn in
sbrs    r16,TOV0    ; Skip next instruction if bit (zero) in register
                  ; (r16) is set
rjmp    loop        ; Jump to loop
                     ; Event Service Code starts here
```

Interrupt Controlled Notification

The AVR can be configured to execute interrupts if a timer event has occurred (the corresponding interrupt flag in the TIFRn is set). Normal program execution will be interrupted (almost) immediately and the processor will execute the code of the Interrupt Service Routine. The advantage compared to polling of interrupt flags is zero overhead in the main loop. This saves processing time. The section *Setting Up the Timers* shows a few examples of how this can be implemented.

Timer interrupts are enabled by setting the corresponding bit in the Timer Interrupt Mask Register (TIMSKn). The following example shows steps to enable the Output Compare Interrupt on channel A of Timer2:

```
ldi     r16,1<<OCIE2A
sts     TIMSK2,r16    ; Enable timer output compare interrupt
sei                    ; Enable global interrupts
```

Automatic Reaction on Events

Timers on this device support the possibility to react on timer interrupt events on purely hardware basis without the need to execute code. Related output pins can be configured to be set, cleared, or toggled automatically on a compare match. In contrast to the two other solutions this happens in parallel to normal code execution and requires no processing time.

The following code example shows steps to set the compare value and enable pin toggling. The configuration of Timer2 can be as follows:

```
ldi     r16, (1<<COM2A1)|(1<<WGM21)|(1<<WGM20)
sts     TCCR2A,r16      ; OC2A toggling on compare match/timer
                        ; Clock = system clock
```

```
ldi     r16,32
sts     OCR2A,r16       ; Set output compare value to 32
```

To enable pin toggling, the data direction register bit corresponding to OCx has to be set to make it an output pin.

3. Clock Options

The clock unit of the AVR timers consists of a prescaler connected to a multiplexer. A prescaler can be considered as a clock divider. Generally, it is implemented as a counter with several output signals at different counting stages. In the case of the ATmega328PB, a 10-bit counter is used to divide the input clock in four (six in case of the Timer2) different prescaled clocks. The multiplexer is used to select which prescaled clock signal to use as input signal for the Timer. Alternatively, the multiplexer can be used to bypass the prescaler and configure an external pin to be used as input for the Timer.

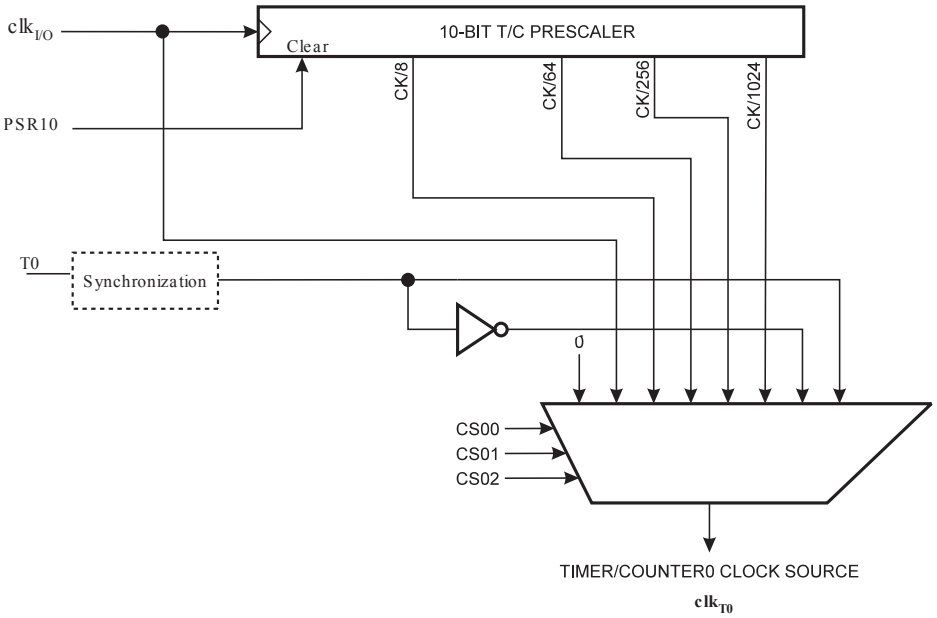
There are two prescaler blocks available in the device. Since Timer0 and Timer1 are synchronous timers and use the system clock (CPU clock) as input source, they can use the same prescaler block (as long as each timer can be configured separately using the corresponding CSxx bits). However, the asynchronous clocked Timer2 needs its own prescaler to be independent of the system clock.

The following figure shows the prescaling and the configuration unit for Timer0/1. The data sheets contain more detailed drawings showing all prescalers and multiplexers. An overview of the possible clock settings is given in Table 3-1 Overview of the Clock Settings . In the following sections these settings will be described more clearly.

Note:

1. The prescaler is constantly running during operation. In cases where the timer has to count very accurately it has to be ensured that the prescaler starts counting from zero.
2. On devices with shared prescaler, executing a prescaler reset will affect all connected timers.

Figure 3-1. Prescaler for Timer0/1



Clocking by System Clock

In this case, the system clock is used as input signal for the prescaler. Even if a prescaled value is chosen instead of the system clock, this clock is based on the system clock. The timer clock is therefore synchronous to the system clock. All five timers of the ATmega328PB and most timers on other AVR parts support this option. Small time frames can be implemented or monitored because of the high frequency of the system clock. The timer overflow frequency is a good indication of the size of the time frame a timer covers. Equation 1 shows the correlation between the timer overflow frequency TOVCK, the

maximum value (MaxVal) of the timer, the system clock (CK), and the division factor of the prescaler (PVal).

Table 3-1. Overview of the Clock Settings

TCCRx			Synchronous Timer0 and Timer1 P_{CK} = CK	Synchronous/Asynchronous Timer2 P_{CK2} = f (AS2)
Bit 2	Bit 1	Bit 0		
CSx2	CSx1	CSx0	$T_{CK0/1}$	T_{CK2}
0	0	0	0 (Timer Stopped)	0 (Timer Stopped)
0	0	1	P_{CK} (System Clock)	P_{CK2} (System Clock/Asynchronous Clock)
0	1	0	$P_{CK}/8$	$P_{CK2}/8$
0	1	1	$P_{CK}/64$	$P_{CK2}/32$
1	0	0	$P_{CK}/256$	$P_{CK2}/64$
1	0	1	$P_{CK}/1024$	$P_{CK2}/128$
1	1	0	External Clock on Tn pin	$P_{CK2}/256$
1	1	1	External Clock on Tn pin	$P_{CK2}/1024$

$$TOV_{CK} = \frac{f_{CK}}{MaxVal} = \frac{(P_{CKx} / PVal)}{MaxVal} = \frac{P_{CKx}}{(PVal \bullet MaxVal)} \qquad (2)$$

Assume that the CPU is running with f_{CPU} = 1MHz and the resolution of the timer is 8 bit (MaxVal = 256). A prescale value of 64 will then cause the timer to be clocked with TCK = 1MHz/64 so that there will be about 61 timer overflows per second. See Equation 2 for the correct mathematical description:

To get 61 timer overflow events per second means that every 16ms an overflow occurs. The maximum prescaler value will generate a timer overflow every ~262ms while the minimum prescaler value generates a timer overflow every 256µs.

In most cases a different approach will be used to determine the settings. The requirements of the application will specify the frequency of the timer overflows. Based on this and the given clock frequency of the CPU together with the timer resolution the prescaler settings will be calculated according to the following equation.

$$PVal = \frac{P_{CKx}}{(TOV \bullet MaxVal)} \qquad (3)$$

The assembler implementation for Timer0 can look like the following code example. These lines set the prescaler values in the TCCR0B to a clock division factor of 1024.

```
ldi     r16,(1<<CS02)|(1<<CS00)
sts     TCCR0B,r16     ; Timer clock = system clock/1024
```

Clocking by Asynchronous Clock

In contrast to the two other timers, which do not support this option, Timer2 of the ATmega328PB can be clocked by an asynchronous external clock. For this purpose a crystal or a ceramic resonator can be connected to the on-board oscillator via the pins TOSC1 and TOSC2. The oscillator is optimized for a watch crystal of 32.768kHz. This frequency is well suited for the implementation of Real Time Clocks (RTC). For more information, See AVR134: Real Time Clock (RTC) using the Asynchronous Timer. The main advantage of a separate clock is that it is independent of the system clock. This makes it possible to run the part at a high processing frequency while the timer is clocked by an external clock with a frequency optimized for accurate timing. Additional power save mode support allows putting the part in sleep mode while the asynchronous timer is still in duty.

Asynchronous operation requires some additional consideration. Because the clocking of Timer2 is asynchronous, the timer events have to be synchronized by the CPU. This requires a timer clock frequency, which is at least four times lower than the system clock. On the other hand, conflicts between the synchronous and the asynchronous access have to be avoided. This is done by using temporary registers. Status bits signalize when an update of the configuration registers is in process. See the description of the Asynchronous Status Register (ASSR) in the data sheet for details.

The TOVCK is calculated according to Equation 2, but by using the oscillator frequency instead of the system clock. The settings of TCCR2B are given in Table 3-1 Overview of the Clock Settings . The prescaler input clock PCK2 is a function of the AS2 bit in the ASSR register. If this bit is cleared, the timer runs in synchronous mode with the system clock as input frequency. If this bit is set, the asynchronous clock signal on the pins TOSC1 and TOSC2 is used as input signal of the prescaler.

The assembler implementation for Timer2 can look like the following code example. These two lines set the prescaler values in TCCR2B to a clock division factor of 1024 (see Table 3-1 Overview of the Clock Settings).

```
ldi     r16, (1<<CS22)|(1<<CS21)|(1<<CS20)
sts     TCCR2B,r16     ; Timer clock = system clock/1024
```

External Clocking

External clocking is supported by Timer0 and Timer1 only. This mode allows the use of a wide range of external signals as timer clock signals. This is synchronous clocking, which means that the CPU detects the status of the pin and clocks the timer synchronously to the system clock if an external clock signal was detected. The T1/T0 pin is sampled once every system clock cycle by the pin synchronization logic. The synchronized (sampled) signal is then passed through an edge detector. This clocking option is selected in TCCRx, the settings of the bits CS00, CS01, and CS02 can be found in Table 3-1 Overview of the Clock Settings .

The assembler implementation for the Timer0 can look like the following code example. These lines set pin T0 as input pin for the timer clock with the rising edge as the active clock edge.

```
ldi     r16,(1<<CS02)|(1<<CS01)|(1<<CS00)
sts     TCCR0B,r16     ; Timer clock = external pin T0, rising edge
```

Note: It is important to ensure that pin T0 is an input pin in the Data Direction Register of Port B (DDRB). The setting of the direction register will not be overwritten by the timer setup, because it is also allowed to implement a software clocked timer in the AVR. T0 and T1 are inputs by default.

How to Stop the Timer

Stopping the timer from counting is simple. A value of zero as prescaler values in the TCCRx stops the corresponding timer (see Table 3-1 Overview of the Clock Settings). However, remember that the prescaler is still running.

The assembler implementation for the Timer0 can look like the following code example.

```
clr         r16
sts         TCCR0B,r16    ; writing zero to TCCR0B stops Timer 0
```

Note: Other TCCRx may contain configuration bits beside the clock select (CSxx) bits. The command lines above will clear these bits. This has to be avoided if these bits were set. That costs one extra line of code, as shown in the following code snippet.

```
lds         r16,TCCR0B    ; Load current value of TCCR0B
andi    r16,~((1<<CS02)|(1<<CS01)|(1<<CS00))
                    ; Clear CS02,CS01,CS00
sts         TCCR0B,r16    ; Writing zero to CS02, CS01, and CS00 in TCCR0B stops Timer0. The
other bits are not affected
```

4. Setting Up the Timers

This section shows concrete examples for how to set up the different timers. The data sheet and the application notes listed in the final section of this document should be read in addition. Especially when transforming the settings to other parts than the ATmega328PB.

Using interrupts is the most common way to react on timer events. The examples described in the following use interrupts.

Independent of the different features of these timers, they all have two things in common. The timer has to be started by selecting the clock source and, if interrupts are used, they have to be enabled.

Shared Registers

If the same registers are used in the interrupt service routines as in the main code, these registers have to be saved at the beginning of the ISR and restored at the end of the ISR. If not all the 32 registers are needed in the application, the save and restore operations can be avoided by using separate registers in the main code and the ISR.

It is also very important to remember to store the Status Register (SREG), as this is not automatically done by the interrupt handler.

Note: The C compiler handles this automatically. If assembly language is used, it has to be done manually by using push and pop instructions.

4.1. Example 1 : Timer0 Overflow Interrupt

The 8-bit Timer0 is a synchronous Timer. This means that it is clocked by the system clock, a prescaled system clock, or an external clock, which is synchronized with the system clock (see section Clock Options for more details about this). This timer is the least complex of the three. Only a few settings have to be made to get it running.

The following example will show how the Timer0 can be used to generate Timer Overflow Interrupt. With every interrupt, pin PB5 on Port B will be toggled. To observe this, the STK®600 Development Board or the ATmega328PB Xplained Mini can be used. On STK600, PB5 has to be connected to an LED. The LED will blink with a frequency (fLED) that is determined by the following formula:

$$f_{LED} = \frac{f_{CK}}{MaxVal} = \frac{(CK/PVal)}{2 \bullet \text{MaxVal}} = \frac{CK}{2(\text{PVal} \bullet \text{MaxVal})}$$

A system consisting of an 8-bit timer (MaxVal = 256) and a system clock of CK = 1 MHz, which is divided by a prescaler value of PVal = 1024, will cause the LED to blink with a frequency (fLED) of approximately 3.8Hz. The following initialization routine shows how to set up such a system:

```
init_Ex1:
    ldi r16,(1<<CS02)|(1<<CS00)
    out TCCR0B,r16 ; Timer clock = system clock / 1024
    ldi r16,1<<TOV0
    out TIFR0,r16 ; Clear TOV0/ Clear pending interrupts
    ldi r16,1<<TOIE0
    sts TIMSK0,r16 ; Enable Timer/Counter0 Overflow Interrupt
    ret
```

The corresponding C code looks like this:

```
void init_Ex1(void)
{
```

```
    /* Timer clock = I/O clock / 1024 */
    TCCR0B = (1<<CS02)|(1<<CS00);
    /* Clear overflow flag */
    TIFR0  = 1<<TOV0;
    /* Enable Overflow Interrupt */
    TIMSK0 = 1<<TOIE0;
}
```

In the next step, the interrupt service routine has to be implemented. This routine will be executed with every timer overflow. The purpose of the routine in this example is to toggle PB5.

```
ISR_TOV0:
    push r16
    in r16,SREG
    push r16
    call TOGGLEPIN
    pop r16
    out SREG,r16
    pop r16
    reti
```

```
TOGGLEPIN:
    sbic portb,PORTB5
    rjmp CLEARPIN
    nop
    sbi portb,PORTB5
    jmp RET1
CLEARPIN:
    cbi portb,PORTB5
RET1:
    ret
```

The corresponding C code:

```
ISR (TIMER0_OVF_vect)
{
    /* Toggle a pin on timer overflow */
    PORTB  ^= (1 << USER_LED);
}
```

4.2. Example 2 : Timer1 Input Capture Interrupt

The 16-bit timer1 is a synchronous timer. This means that it is clocked by the system clock, a prescaled system clock or an external clock which is synchronized with the system clock. To ensure that the 16-bit registers of the Timer1 are written and read simultaneously, a temporary register (Temp) is used. This makes it necessary to access these registers in a specific order. See the application note “AVR072: Accessing 16-bit I/O Registers” and the device datasheet for details. The correct way to access the registers is shown in following table.

Table 4-1. Accessing 16-bit Registers

Operation	1st Access	2nd Access
Read	Low Byte	High Byte
Write	High Byte	Low Byte

According to this, a read operation of a 16-bit register can look like this:

```
lds    r16,TCNT1L
lds    r17,TCNT1H
```

A write operation to this register has to access the registers in the opposite order:

```
sts    TCNT1H,r17
sts    TCNT1L,r16
```

The C Compiler automatically handles 16-bit I/O read and write operations in the correct order.

This example will show the implementation of a very simple use of the input capture event and interrupt. The port pin PB0 is the input capture pin (ICP1). If the value of this pin changes, a capture will be triggered; the 16-bit value of the counter (TCNT1) is written to the Input Capture Register (ICR1).

Measurement of an external signal's duty cycle requires that the trigger edge is changed after each capture. Changing the edge sensing must be done as early as possible after the ICR1 Register has been read. This is to make sure that the change in sense configuration is done before the next falling edge occurs. After a change of the edge, the Input Capture Flag (ICF1) must be cleared by software (writing a logical one to the I/O bit location). For measuring frequency only, the clearing of the ICF1 Flag is not required (if an interrupt handler is used).

The following initialization routine shows how to set up such a system:

```
init_Ex2:
    ldi r16,(1<<CS11)|(1<<CS10)
    sts TCCR1B,r16    ; timer clock = system clock/64
    ldi r16,1<<ICF1
    out TIFR1,r16    ; Clear ICF1/clear pending interrupts
    ldi r16,1<<ICIE1
    sts TIMSK1,r16    ; Timer/Counter1 Capture Event Interrupt
    ldi r16,1<<ICNC1
    sts TCCR1B,r16  ; Enable noise canceler
    cbi DDRB,PORTB0    ; Set PB0/ICP1 as input
    ret
```

The corresponding C code looks like this:

```
void init_Ex2(void)
{
    /* Timer clock = I/O clock / 64 */
    TCCR1B = (1<<CS11)|(1<<CS10);
    /* Clear ICF1. Clear pending interrupts */
    TIFR1   = 1<<ICF1;
    /* Enable Timer 1 Capture Event Interrupt */
    TIMSK1  = 1<<ICIE1;
}
```

In the next step, the interrupt service routine has to be implemented. This routine will be executed with every input capture event. The ISR will toggle PB5 on each capture event and cleat TCNT register before for next measurement.

```
ISR_TIM1_CAPT:
    push r16
    in r16,SREG
    push r16
    clr r16
    sts TCNT1H,r16    ; Write Temp register
    sts TCNT1L,r16    ; Clear the 16 bit register
    call TOGGLEPIN
    pop r16
    out SREG,r16
    pop r16
    reti
```

The corresponding C code looks like this:

```
ISR (TIMER1_CAPT_vect)
{
```

```
    /* Toggle a pin after input capture */
    PORTB  ^= (1 << USER_LED);
    /* Clear counter to restart counting */
    TCNT1 = 0;
}
```

Note: This implementation has one disadvantage: A timer overflow is not detected. Thus, if the duty cycle of the wave has a longer period than what can be covered the 16-bit counter, an overflow will happen before the next edge occurs. A global variable which is set in a timer overflow ISR can be used to sense whether a timer overflow has happened before the capture is performed. If this variable is set, then the effective capture value will be (0xFFFF + contents of ICR1).

4.3. Example 3 : Timer2 Asynchronous Operation

Timer2 can be used in synchronous mode like Timer0 and Timer1. In addition, an asynchronous mode can be used. See the description of the asynchronous clocking in section Clocking by Asynchronous Clock or the data sheet for details.

Example – Timer Output Compare Interrupt

This example shows how to use the timer output compare interrupt of Timer2. The timer will be configured so that the compare match event occurs every second. This feature could be used to implement a RTC. In this example, however, the port pins will be inverted with every compare match event so that PB5 will be blinking with a frequency of 0.5Hz.

Similar to the previous example, PB5 has to be connected to an LED. In addition, a 32.768kHz crystal has to be mounted on the pins TOSC1/PB6 and TOSC2/PB7 of Port B.

The timer settings can be calculated according to Equation 2. As Timer maximum value (MaxVal) the value of the OCR2 has to be used instead. The prescaler clock (PCKx) is in this case the clock signal of the watch crystal (fOSCCK). TOVCK is the overflow frequency which is specified by the application to 1 second. The mathematical description of this relation is shown by the following equation:

$$1 = TOV_{CK} = \frac{f_{OSCCK}}{\text{PVal} \bullet \text{OCR2}} = \frac{32.768\ \text{kHz}}{\text{PVal} \bullet \text{OCR2}}$$

A prescaler value of 1024 is selected plus a corresponding OCR2 value of 32 to get the delay time of one second between two Timer compare match events.

The following initialization routine shows how to set up such a system:

```
init_Ex3:
    ldi r16, (1<<AS2)
    sts ASSR,r16 ; Enable asynchronous mode
                 ; Clear Timer on compare match. Toggle OC2A on Compare Match
    ldi r16, (1<<COM2A0) | (1<<WGM21)
    sts TCCR2A,r16
                 ; Timer clock = 32768/1024
    ldi r16, (1<<CS22)|(1<<CS21)|(1<<CS20)
    sts TCCR2B,r16
    ldi r16,32
    sts OCR2A,r16 ; Set output compare value to 32
    sbi DDRB, PORTB3 ; Set OC2A pin as output
    lds r16, ASSR ; Refer ATmega328PB datasheet section
                  ;'Asynchronous Operation of Timer/Counter2'
    andi r16, ((1 << OCR2AUB) | (1 << OCR2BUB) | (1 << TCR2AUB) | (1 << TCR2BUB) | (1<<
TCN2UB))
    brne PC-0x03
    ldi r16, (1 << TOV2) | (1 << OCF2A) | (1 << OCF2B)
    sts TIFR2,r16 ; Clear pending interrupts
    ldi r16,1<<OCIE2A
```

```
    sts TIMSK2,r16 ; Enable timer output compare interrupt
    ret
```

The corresponding C code can be as follows:

```
void init_Ex3(void)
{
    /* Select clock source as crystal on TOSCn pins */
    ASSR |= 1 << AS2;
    /* Clear Timer on compare match. Toggle OC2A on Compare Match */
    TCCR2A = (1<<COM2A0) | (1<<WGM21);
    /* Timer Clock = 32768 Hz / 1024 */
    TCCR2B = (1<<CS22)|(1<<CS21)|(1<<CS20);
    /* Set Output Compare Value to 32. Output pin will toggle every second */
    OCR2A  = 32;
    /* Wait till registers are ready
     * Refer ATmega328PB datasheet section
     * 'Asynchronous Operation of Timer/Counter2' */
    while ((ASSR & ((1 << OCR2AUB) | (1 << OCR2BUB) | (1 << TCR2AUB)
        | (1 << TCR2BUB) | (1<< TCN2UB))));
    /* Clear pending interrupts */
    TIFR2  = (1 << TOV2) | (1 << OCF2A) | (1 << OCF2B);
    /* Enable Timer 2 Output Compare Match Interrupt */
    TIMSK2 = (1 << OCIE2A);
    /* Set OC2A pin as output */
    DDRB |= (1 << OC2A_PIN);
}
```

In the next step the interrupt service routine has to be implemented. This routine will be executed with every output compare event. The purpose in this example is to toggle an LED connected to PB5.

```
ISR_OCIE2A:
    push r16
    in r16,SREG
    call TOGGLEPIN
    out SREG,r16
    pop r16
    reti
```

The corresponding C code can be as follows:

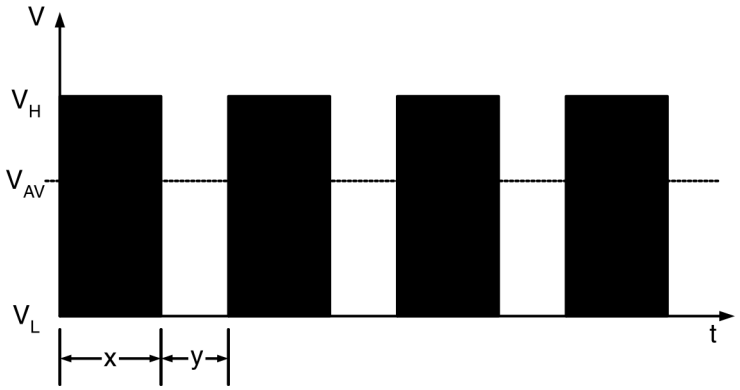
```
ISR(TIMER2_COMPA_vect)
{
    /* Toggle a pin to indicate compare interrupt */
    PORTB  ^= (1 << USER_LED);
}
```

5. PWM Basics

PWM is an abbreviation for Pulse Width Modulation. In this mode, the timer acts as an up/down counter. This means that the counter counts up to its maximum value and then clears to zero. The advantage of the PWM is that the duty cycle relation can be changed in a phase consistent way.

If the PWM is configured to toggle the Output Compare pin (OCx), the signal at this pin can look like shown in the following figure.

Figure 5-1. Output Signal of PWM



VH: Output Voltage high level

VL: Output Voltage low level

VAV: Average Output Voltage level

x: Duty cycle high level

y: Duty cycle low level

A low-pass filter at the output pin combined with the relative high speed of the PWM will cause a constant voltage level instead of a square wave signal as output signal. Equation 4 shows how this voltage level can be calculated:

$$V_{AV} = \frac{(V_H \bullet x + V_L \bullet y)}{(x + y)} \qquad (4)$$

where

$$x = OCRx \bullet 2$$
$$y = (MaxVal - OCRx) \bullet 2$$

$$V_{AV} = \frac{(V_H \bullet OCRx + V_L \bullet (MaxVal - OCRx))}{MaxVal} \qquad (5)$$

The fact that this method allows the timer to generate voltage levels between VCC and GND means that a DAC can be implemented using the PWM. Details about this are described in the application notes *AVR314: DTMF Transmitter* and *AVR335: Digital Sound Recorder with AVR and Serial DataFlash*.

5.1. Example 4 : Timer2 PWM Generation

This example shows how to generate voltages between VCC and GND at the output pin of the PWM (PB3/OC2A). To observe this, PB3 should be connected an LED. This PWM output signal has a duty relation of 1/8 to 7/8 (OCR2A = 0xE0).

The following initialization routine shows how to set up such a system:

```
init_Ex4:
    ; 8 bit PWM non-inverted
    ldi r16, (1<<COM2A1)|(1<<WGM21)|(1<<WGM20);
    sts TCCR2A,r16 ; 8 bit PWM non-inverted
    ldi r16,(1<<CS20)
    sts TCCR2B,r16 ; Timer clock = I/O clock
    ldi r16,0xE0
    sts OCR2A,r16 ; Set compare value/duty cycle ratio
    sbi DDRB, PORTB3 ; Set OC2A pin as output
    ret
```

The corresponding C code looks like this:

```
void init_Ex4(void)
{
    /* Enable non inverting 8-Bit PWM */
    TCCR2A = (1<<COM2A1)|(1<<WGM21)|(1<<WGM20);
    /* Timer clock = I/O clock */
    TCCR2B = (1<<CS20);
    /* Set the compare value to control duty cycle */
    OCR2A  = 0xE0;
    /* Enable Timer 2 Output Compare Match Interrupt */
    TIMSK2 = (1 << OCIE2A);
    /* Set OC2A pin as output */
    DDRB |= (1 <<     OC2A_PIN);
}
```

6. Further Reading

1. AVR072: Accessing 16-bit I/O Registers
2. AVR134: Real Time Clock (RTC) using the Asynchronous Timer
3. AVR314: DTMF Transmitter
4. AVR335: Digital Sound Recorder with AVR and Serial DataFlash
5. AVR131: Using the 8-bit AVR High-speed PWM

7. Revision History

Doc Rev.	Date	Comments
2505B	03/2016	Updated for newer device
2505A	02/2002	Initial document release

Atmel | Enabling Unlimited Possibilities®

Atmel Corporation 1600 Technology Drive, San Jose, CA 95110 USA **T:** (+1)(408) 441.0311 **F:** (+1)(408) 436.4200 | **www.atmel.com**