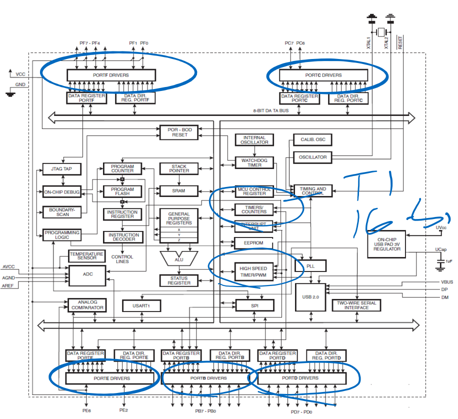


Async. Ripple Counter

$2^3\ 2^2\ 2^1\ 2^0$

0000…0000
0000…0001 — Next bit over toggles
0000…0010
0000…0011
0000…0100
0000…0101
0000…0110
0000…0111
0000…1000
:
1111…1111
000…0000

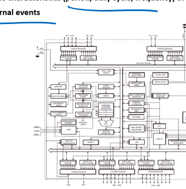
ripple	vs.	synchronous
+ cheap		− $
+ scalable		− hard to scale
+ low noise		− noise
− slow		+ fast

ATmega32U4 Timing Subsystem[1]

The ATmega32U4 is equipped with one 8-bit, two 16-bit and one 10-bit timer/counter. These timer/counters let you…

1. **Turn on or turn off an external device at a programmed time.**
2. **Generate a precision output signal (period, duty cycle, frequency). For example, generate a complex digital waveform with varying pulse width to control the speed of a DC motor**
3. **Measure the characteristics (period, duty cycle, frequency) of an incoming digital signal**
4. **Count external events**

T1 16 bit

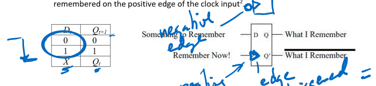


[1] Source: ATmega328P Data Sheet http://www.atmel.com/dyn/resources/prod_documents/8161S.pdf page 5

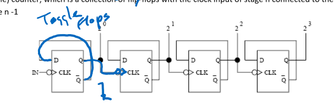
What is a Flip-Flop and a Counter

1-bit memory

- You can think of a D flip-flop as a one-bit memory. The *something to remember* on the D input of flip-flop is remembered on the positive edge of the clock input[2].



- The counter part of an ATmega32U4 Timer/Counter peripheral subsystem is an example of an asynchronous (ripple) counter, which is a collection of flip-flops with the clock input of stage n connected to the output of stage n -1



- When compared with a synchronous counter, an asynchronous "ripple" counter: generates less noise and is less expensive. On the negative side, an asynchronous "ripple" counter is slower than a synchronous counter.

[2] Source: http://sandbox.mc.edu/~bennet/cs314/slides/ch5me-4.pdf

Timing Terminology

Frequency

The number of times a particular event repeats within a 1-s period. The unit of frequency is Hertz, or cycles per second. For example, a sinusoidal signal with a 60-Hz frequency means that a full cycle of a sinusoid signal repeats itself 60 times each second, or every 16.67 ms. For the digital waveform shown, the frequency is 2 Hz.

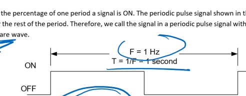
Period

The flip side of a frequency is a period. If an event occurs with a rate of 1 Hz, the period of that event is 1000 ms. To find a period, given a frequency, or vice versa, we simply need to remember their inverse relationship, $F = 1/T$ where F and T represent a frequency and the corresponding period, respectively.

Duty Cycle

In many applications, periodic pulses are used as control signals. A good example is the use of a periodic pulse to control a servo motor. To control the direction and sometimes the speed of a motor, a periodic pulse signal with a changing duty cycle over time is used.

Duty cycle is defined as the percentage of one period a signal is ON. The periodic pulse signal shown in the Figure is ON for 50% of the signal period and off for the rest of the period. Therefore, we call the signal in a periodic pulse signal with a 50% duty cycle. This special case is also called a square wave.



Duty Cycle = $\frac{T_H}{T_P}$

Timer 1 Modes of Operation

= 16 bit

$2^{16} - 1 = 65{,}535$

Table 15-4. Waveform Generation Mode Bit Description[(1)]

Mode	WGM13	WGM12 (CTC1)	WGM11 (PWM11)	WGM10 (PWM10)	Timer/Counter Mode of Operation	TOP	Update of OCR1x at	TOV1 Flag Set on
0	0	0	0	0	Normal	0xFFFF	Immediate	MAX
1	0	0	0	1	PWM, Phase Correct, 8-bit	0x00FF	TOP	BOTTOM
2	0	0	1	0	PWM, Phase Correct, 9-bit	0x01FF	TOP	BOTTOM
3	0	0	1	1	PWM, Phase Correct, 10-bit	0x03FF	TOP	BOTTOM
4	0	1	0	0	CTC	OCR1A	Immediate	MAX
5	0	1	0	1	Fast PWM, 8-bit	0x00FF	BOTTOM	TOP
6	0	1	1	0	Fast PWM, 9-bit	0x01FF	BOTTOM	TOP
7	0	1	1	1	Fast PWM, 10-bit	0x03FF	BOTTOM	TOP
8	1	0	0	0	PWM, Phase and Frequency Correct	ICR1	BOTTOM	BOTTOM
9	1	0	0	1	PWM, Phase and Frequency Correct	OCR1A	BOTTOM	BOTTOM
10	1	0	1	0	PWM, Phase Correct	ICR1	TOP	BOTTOM
11	1	0	1	1	PWM, Phase Correct	OCR1A	TOP	BOTTOM
12	1	1	0	0	CTC	ICR1	Immediate	MAX
13	1	1	0	1	(Reserved)	–	–	–
14	1	1	1	0	Fast PWM	ICR1	BOTTOM	TOP
15	1	1	1	1	Fast PWM	OCR1A	BOTTOM	TOP

Note: 1. The CTC1 and PWM11:0 bit definition names are obsolete. Use the WGM12:0 definitions. However, the functionality and location of these bits are compatible with previous versions of the timer.

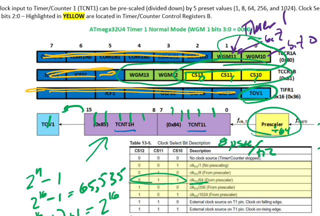
Clear Timer on Compare (CTC) mode
http://maxembedded.com/2011/07/avr-timers-ctc-mode/

GPIO Registers DDR, PORT, PIN
What are the Registers for Timers?
For Normal Mode we only need to learn 5.
TCNTH:TCNTL, TCCR1A/B, TIFR1

Normal Mode[3]

- The simplest AVR Timer mode of operation is the *Normal mode*. Waveform Generation Mode for Timer/Counter 1 (WGM1) bits 3:0 = 0 – Highlighted in **GREEN.** These bits are located in Timer/Counter Control Registers A/B (**TCCR1A** and **TCCR1B**). In this mode the Timer/Counter 1 Register (TCNT1H:TCNT1L) counts up (incrementing), and no counter clear is performed. The counter simply overruns when it passes its maximum 16-bit value 0xFFFF and then restarts 0x0000. There are no special cases to consider in the Normal mode, a new counter value can be written anytime.
- In normal operation the Timer/Counter Overflow Flag (TOV1) bit – Highlighted in **LIGHT BLUE.** located in the Timer/Counter1 Interrupt Flag Register (T1FR1) will be set in the same timer clock cycle as the Timer/Counter 1 Register (TCNT1H:TCNT1L) becomes zero. The TOV1 Flag in this case behaves like a 17th bit, except that it is only set, not cleared.
- The clock input to Timer/Counter 1 (TCNT1) can be pre-scaled (divided down) by 5 preset values (1, 8, 64, 256, and 1024). Clock Select Counter/Timer 1 (CS1) bits 2:0 – Highlighted in **YELLOW** are located in Timer/Counter Control Registers B.

ATmega32U4 Timer 1 Normal Mode (WGM 1 bits 3:0 = 0000)



$f_{clk} = 8 MHz$

$f_{T1} = \frac{f_{clk}}{N} = \frac{8 MHz}{64} = \frac{1}{8} MHz = 125 KHz$

$t_{T1} = \frac{1}{f_{T1}} = \frac{1}{(\frac{1}{8} MHz)} = 8 \mu sec/tic$

$2^n - 1$

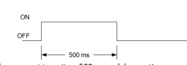
$2^{16} - 1 = 65{,}535$

$(2^{16} - 1) + 1 = 2^{16}$

65,536

[3] ATmega32U4_doc7766.pdf Table 14-5 Waveform Generation Mode Bit Description

Timer/Counter 1 Normal Mode – Design Example



- In this design example, we want to write a 500 msec delay routine assuming a system clock frequency of 8.000 MHz and a prescale divisor of 64.
- The first step is to discover if our 16-bit Timer/Counter 1 can generate a 500 ms delay.

Variable Definitions

t_{clk_T1} : period of clock input to Timer/Counter1
f_{clk} : AVR system clock frequency
$f_{Tclk\_I/O}$: AVR Timer clock input frequency to Timer/Counter Waveform Generator

How to Calculate Maximum Delay (Normal Mode)

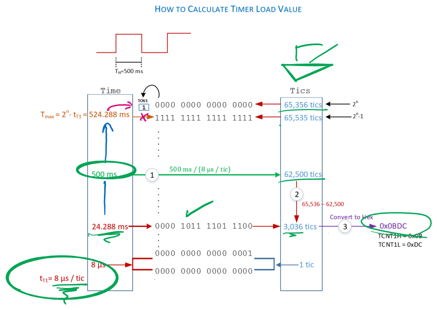
- The largest time delay possible is achieved by setting both TCNT1H and TCNT1L to zero, which results in the overflow flag TOV1 flag being set after 2^{16} = 65,536 tics of the Timer/Counter1 clock.

$f_{T1} = f_{Tclk\_I/O}/64$, given $f_{Tclk\_I/O} = f_{clk}$ then f_{T1} = 8.000 MHz / 64 = 125 KHz

and therefore T1max = 65,536 tics / 125 KHz = 524.288 msec

- Clearly, Timer 1 can generate a delay of 500 msec
- Our next step is to calculate the TCNT1 load value needed to generate a 500 ms delay.

How to Calculate Timer Load Value



Steps to Calculate Timer Load Value (Normal Mode)

Problem

Generate a 500 msec delay assuming a clock frequency of 8 MHz and a prescale divisor of 64.

Solution

1. Divide desired time delay by tclkT1 where tclkT1 = 64/fclkI/O = 64 / 8.000 MHz = 8 µsec/tic
   500msec / 8 µs/tic = 62,500 tics
   short-cut: TCNT1H = high(-62,500) and TCNT1L = low(-62,500)
2. Subtract 65,536 – step 1
   65,536 – 62,500 = 3,036
3. Convert step 2 to hexadecimal.
   3,036 = 0x0BDC
   For our example TCNT1H = 0x0B and TCNT1L = 0xDC
4. Check Answer
   3,036 tics x 8 µs/tic = 24.288 msec
   524.288 msec – 500 msec = 24.288 msec √

Steps to Calculate Clock Divisor (Normal Mode)

- In the previous example we assumed a divisor of 64, and then by calculating the maximum delay T_{MAX} verified that this assumption was correct. After that we simply followed the steps defined in the previous slide to calculate the value to be loaded into 16-bit timer/counter TCNT1.

$$T_{MAX} = \frac{2^n \cdot N}{f_{clk}} \qquad \text{eq1.}$$

Where: T_{MAX} = maximum delay
N = divisor
n = number of flip-flops making-up the timer
f_{clk} = system clock frequency

- But what if we are not given N and need to find TCNT1 for a given delay T. In this case we know that T ≤ T_{MAX} and applying a little algebra can find an equation for N.

$$N \geq \frac{T \cdot f_{clk}}{2^n} \qquad \text{eq2.}$$

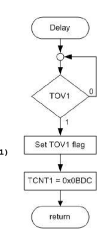
- Let's take a second look at our 500 msec delay problem. This time we will not assume a divisor of 64. Applying equation 2 we have:

$$N \geq (500 \text{ msec x } 8 \text{ MHz}) / 2^{16} = 61.035\ldots$$

- From Table 13.5 "Clock Select Bit Description" on page 10, we see that the possible clock divisors are **1, 8, 64, 256, and 1024**. From this list we want to select the divisor that is the closest value, yet greater than or equal to N. For our example, not surprisingly the answer is again 64.

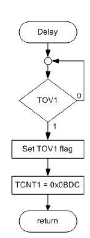
Polling Example – Assembly Version

```
; --------------------------
; ------ Delay 500ms ------
; Called from main program
; Input: none    Output: none
; no registers are modified by this subroutine
Delay:
    push  r15
    in    r15, SREG
    push  r16
wait:
    sbis  TIFR1, TOV1
    rjmp  wait
    sbi   TIFR1, TOV1   // clear flag bit by writing a one (1)
    ldi   r16,0x0B      // load value high byte 0x0B
    sts   TCNT1H,r16
    ldi   r16,0xDC      // load value low byte 0xDC
    sts   TCNT1L,r16
    pop   r16
    out   SREG, r15
    pop   r15
    ret
```



Polling Example – C Version

```
; --------------------------
; ------ Delay 500ms ------
; Called from main program
; Input: none    Output: none
void T1Delay()
{
    while (!(TIFR & (1<<TOV1))) // eq. to Ex: 9-42 expression
    TIFR = 1<<TOV1;             // clear timer overflow flag
    TCNT1H = 0x0B;
    TCNT1L = 0xDC;
}
```



More Looping Examples

Here are six (6) other ways of implementing the looping part of the Polling Example written in assembly. See if you can come up with a few more.

wait:
    sbis TIFR1, TOV1 // targets a specific bit
    rjmp wait
	wait:
	    in r16, TIFR1
	    sbrs r16, TOV1
	    rjmp wait

wait:
    in r16, TIFR1
    bst r16, TOV1
    brtc wait	

wait:
    in r16, TIFR1
    andi r16, 0x01 // bitwise operation
    breq wait
	wait:
	    in r16, TIFR1
	    cbr r16, 0xFE
	    breq wait

wait:
    in r16, TIFR1
    ror r16
    brcc wait
	wait:
	    in r16, TIFR1
	    lsr r16
	    brcc wait