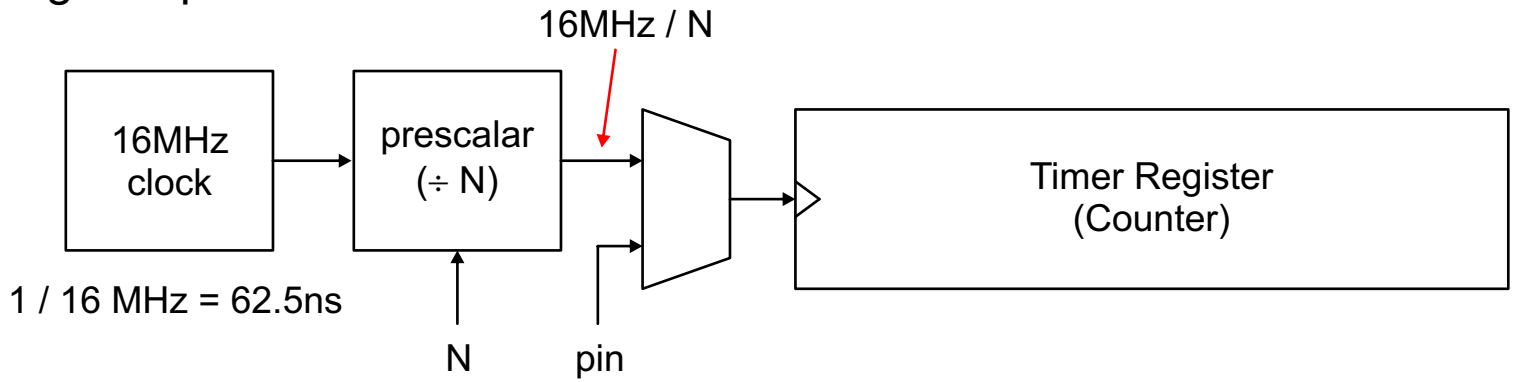
Atmega328p Timers 0 and 1:



N = 1 (no prescale), 8, 64, 256, 1024

Timer 0 and Timer 2 are both 8 bit timers (count up from 0 to 255 over and over when enabled in Normal mode)

Timer 1 is a 16 bit timer (counts up from 0 to 65535 over and over when enabled in Normal mode)

Resolution: Smallest amount of time that a timer can measure. This is the inverse of the timer clock frequency.

$$= \frac{1}{\left(\frac{16\ \text{MHz}}{N}\right)} = \frac{N}{16\ \text{MHz}} = N \times \frac{1}{16\ \text{MHz}} = N \times 62.5\text{ns}$$

Range: Maximum amount of time that a timer can measure. It equals Resolution * 2^n
(where n is the number of bits in the timer)

Prescale (N)	Resolution	Timer 1 Range	Timer 0 or 2 Range
1	0.0625us	4.096ms	16us
8	0.5us	32.768ms	128us
64	4us	262.144ms	1.024ms
256	16us	1.048576s	4.096ms
1024	64us	4.194304s	16.384ms

Note: Timer 2 has a slightly different block diagram and additional prescale values of 32 and 128

Atmega328p Timers 0 and 1:

Prescale (N)	Resolution	Timer 1 Range	Timer 0 or 2 Range
1	0.0625us	4.096ms	16us
8	0.5us	32.768ms	128us
64	4us	262.144ms	1.024ms
256	16us	1.048576s	4.096ms
1024	64us	4.194304s	16.384ms

$$\text{Target Timer Value} = \frac{\text{Desired Time}}{\text{Resolution}}$$ (We may need to round)

Prescale (N)	Resolution	Value for .5 s
1	0.0625us	8000000
8	0.5us	1000000
64	4us	125000
256	16us	31250
1024	64us	7812.5

Use 7813 or 7812 (but there will be some error)

7813 * 64us = 0.500032 s
7812 * 64us = 0.499968 s

16.11.2 **TCCR1B** – Timer/Counter1 Control Register B

bit:	7	6	5	4	3	2	1	0	
(0x81)	ICNC1	ICES1	-	WGM13	WGM12	CS12	CS11	CS10	**TCCR1B**
Read/Write:	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value:	0	0	0	0	0	0	0	0	

bits 7,6: ICNC1, ICES1
These bits control functions related to an Input Capture functionality that timer 1 can support We may explore this functionality later. Since we won't be using this functionality, we will just leave their values at 0.

bit 5: This bit is reserved for future use. For ensuring compatibility with future devices, this bit must be written to zero when TCCR1B is written.

bits 4-3: WGM13, WGM12
In conjunction with bits WGM10 and WGM11 (found in rgister TCCR1A), these bits control PWM (pulse width modulation) modes and Timer Compare modes. These modes are collectively referred to as Waveform Generation Modes (WGM) To use the timer as a basic timer, we leave these bits at their default value of 0.

bits 2-0: CS12, CS11, CS10
These are the Clock Select bits. They determine what the clock source is, including chosing the prescalar.

16.11.2 **TCCR1B** – Timer/Counter1 Control Register B

bit:	7	6	5	4	3	2	1	0	
(0x81)	ICNC1	ICES1	-	WGM13	WGM12	CS12	CS11	CS10	**TCCR1B**
Read/Write:	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value:	0	0	0	0	0	0	0	0	

Table 16-5. Clock Select Bit Description

CS12	CS11	CS10	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	$CLK_{I/O}$ / 1 (No prescaling)
0	1	0	$CLK_{I/O}$ / 8 (From prescaler)
0	1	1	$CLK_{I/O}$ / 64 (From prescaler)
1	0	0	$CLK_{I/O}$ / 256 (From prescaler)
1	0	1	$CLK_{I/O}$ / 1024 (From prescaler)
1	1	0	External clock source on T1 pin. Clock on falling edge.
1	1	1	External clock source on T1 pin. Clock on rising edge.

Note: Timer 0 uses the same CS values as Timer1, but Timer 2 has additional prescaler choices and uses different CS values.

Examples:

```
TCCR1B = 3; // Set the prescaler to 64, and all other bits to 0.
            // Assuming that the TCCR1B CS bits were previously 0,
            // this will also turn Timer 1 on.

TCCR1B = (TCCR1B & ~7) | 3; // Change the prescaler to 64 without
                            // changing any other bits
```

16.11.4 **TCNT1H** and **TCNT1L** – Timer/Counter1

bit:	7	6	5	4	3	2	1	0	
(0x85)	TCNT1[15:8]								**TCNT1H**
(0x84)	TCNT1[7:0]								**TCNT1L**
Read/Write:	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value:	0	0	0	0	0	0	0	0	

Note that the ATmega328p processor is an 8 bit processor, so reading or writing 16 bits to or from the 16 bit timer Timer 1 actually requires two 8 bit references using TCNT1H and TCNT1L.

But, the timer is still a 16 bit timer. To do a 16-bit write, the high byte must be written to TCNT1H before the low byte is written to TCNT1L. All 16 bits of data are actually transferred to the timer after the low byte is written to TCNT1L.

For a 16-bit read, the low byte must be read from TCNT1L before the high byte is read from TCNT1H. When the byte read from TCNT1L is initiated, all 16 bits of the timer are moved to TCNT1L and TCNT1H.

In C, you should use TCNT1 to access all 16 bits of the timer. The TCNT1 reference will be translated into the above two operations, performed in the correct order, by the compiler.

Example:

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

int main(void)
{
    // Set an LED attached to Pin13 (Port B bit 5) as Output
    DDRB |= (1 << DDB5);

    // Set up and start Timer1 at Prescale Value of 64
    // and the other bits (ICNC1, ICNES and WGM bits) as 0
    TCCR1B = 3;
    // Together with the 0 defaults on TCCR1A and TCCR1C this gives a
    // basic up-count timer mode in Timer1 (Normal mode)

    while (1)
    {
        if(TCNT1 >= 12500) // Careful, it is not always safe to use ==
        {
            //Toggle the LED
            PORTB ^= (1 << PORTB5);

            // Reset Timer1
            TCNT1 = 0;
        }
    }
}
```

At 16 Mhz, and a prescalar of 64, waiting for a count of 12500 on the timer takes how much time?

Example:

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

int main(void)
{
    // Set an LED attached to Pin13 (Port B bit 5) as Output
    DDRB |= (1 << DDB5);

    // Set up and start Timer1 at Prescale Value of 64
    // and the other bits (ICNC1, ICNES and WGM bits) as 0
    TCCR1B = 3;
    // Together with the 0 defaults on TCCR1A and TCCR1C this gives a
    // basic up-count timer mode in Timer1 (Normal mode)

    while (1)
    {
        if(TCNT1 >= 12500) // Careful, it is not always safe to use ==
        {
            //Toggle the LED
            PORTB ^= (1 << PORTB5);

            // Reset Timer1
            TCNT1 = 0;
        }
    }
}
```

What if we need a delay longer than that provided by a count of 65535 using Timer 1?

16.11.9 **TIFR1** – Timer/Counter1 Interrupt Flag Register

bit:	7	6	5	4	3	2	1	0	
0x16 (0x36)	-	-	ICF1	-	-	OCF1B	OCF1A	TOV1	**TIFR1**
Read/Write:	R	R	R/W	R	R	R/W	R/W	R/W	
Initial Value:	0	0	0	0	0	0	0	0	

TOV1: Timer/Counter1 Overflow Flag
OCF1A: Timer/Counter1 Output Compare A Match Flag
OCF1B: Timer/Counter1 Output Compare B Match Flag
ICF1: Timer/Counter1 Input Capture Flag

These flags are set when particular Timer1 events occur (based on the Timer1 mode). If enabled, these events can cause interrupts. If the event is handled by an interrupt, the flag will automatically be cleared by hardware. But, these flags can also be managed by software.

In Normal mode, Timer 1 will count up from 0x0000 to 0xFFFF (65535), incrementing by 1 with each pre-scaled clock cycle. On the next clock, the timer resets to 0x0000 **and** the **TOV1** flag **is set**. The flag will remain set until it is cleared. If it is not enabled as an interrupt (so that the flag would automatically be cleared by the hardware upon interrupt service), software may clear it explicitly.

16.11.9 **TIFR1** – Timer/Counter1 Interrupt Flag Register

bit:	7	6	5	4	3	2	1	0	
0x16 (0x36)	-	-	ICF1	-	-	OCF1B	OCF1A	TOV1	**TIFR1**
Read/Write:	R	R	R/W	R	R	R/W	R/W	R/W	
Initial Value:	0	0	0	0	0	0	0	0	

TOV1: Timer/Counter1 Overflow Flag
OCF1A: Timer/Counter1 Output Compare A Match Flag
OCF1B: Timer/Counter1 Output Compare B Match Flag
ICF1: Timer/Counter1 Input Capture Flag

Note: To clear any of the TIFR1 flags explicitly with software, you must write a **1** to the flag. Writing a 0 to any bit does not change its current value.

Because of this behavior **RMW** (Read Modify Write) **operations on TIFR1 must be avoided!** This applies to all the TIFRx registers.

So, to clear TOV1 we must use

```
TIFR1 = (1 << TOV1);
```

rather than

```
TIFR1 |= (1 << TOV1);
```

What happens if both OCF1B and TOV1 are both set in TIFR1 and we used the |= operation above?

Example, Suppose we want to delay for 500,000 Timer1 counts (2 sec at 16 MHz and a 64 prescale)..
Noting that 500,000 = 65,536 * 7 + 41,248:

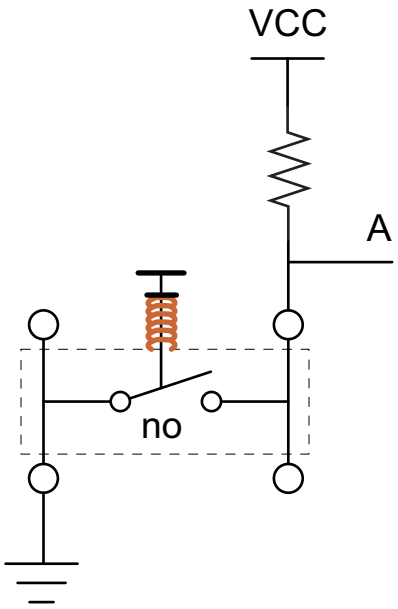
```
#include <avr/io.h>

int main(void)
{
    uint8_t NumOverflows = 0;
    // Set an LED attached to Pin13 (Port B bit 5) as Output
    DDRB |= (1 << DDB5);

    // Set up and start Timer1 at Prescale Value of 64
    // and the other bits (ICNC1, ICNES and WGM bits) as 0
    TCCR1B = 3;
    // Together with the 0 defaults on TCCR1A and TCCR1C this gives a
    // basic up-count timer mode in Timer1 (Normal mode)

    while (1)
    {
        while (NumOverflows < 7) {
            while ((TIFR1 & (1 << TOV1)) == 0); // Wait until Timer1 Overflows
            NumOverflows++;
            TIFR1 = (1 << TOV1); // Clear the TOV1 flag
        }

        While(TCNT1 < 41248); // wait for Timer1 to reach 41248

        //Toggle the LED
        PORTB ^= (1 << PORTB5);

        // Reset Timer1 and the Overflow counter
        TCNT1 = 0;
        NumOverflows = 0;

    }
}
```

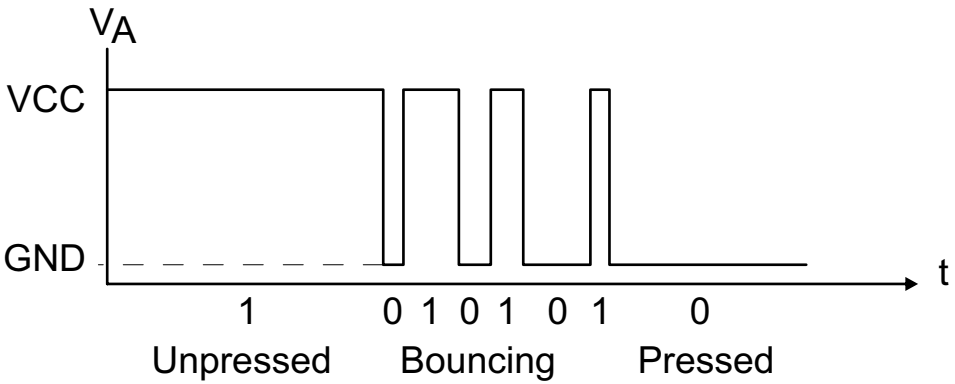
Example, Suppose we want to delay for 500,000 Timer1 counts (2 sec at 16 MHz and a 64 prescale)..
Noting that 500,000 = 65,536 * 7 + 41,248:

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

int main(void)
{
    uint8_t NumOverflows = 0;
    // Set an LED attached to Pin13 (Port B bit 5) as Output
    DDRB |= (1 << DDB5);

    // Set up and start Timer1 at Prescale Value of 64
    // and the other bits (ICNC1, ICNES and WGM bits) as 0
    TCCR1B = 3;
    // Together with the 0 defaults on TCCR1A and TCCR1C this gives a
    // basic up-count timer mode in Timer1 (Normal mode)

    while (1)
    {
        while (NumOverflows < 7) {
            while ((TIFR1 & (1 << TOV1)) == 0); // Wait until Timer1 Overflows
            NumOverflows++;
            TIFR1 = (1 << TOV1); // Clear the TOV1 flag
        }

        While(TCNT1 < 41248); // wait for Timer1 to reach 41248

        //Toggle the LED
        PORTB ^= (1 << PORTB5);

        // Reset Timer1 and the Overflow counter
        TCNT1 = 0;
        NumOverflows = 0;
    }
}
```

65536 x 7 = 458,742 which is 1.835 sec at 16 MHz and a 64 prescale.
If that was close enough to 2 sec for our purposes, we could leave out the second nested while.
Perhaps better: 65586 x 8 = 524,488 which is 2.097 sec (closer to 2 sec).

Buttons / Switches – Switch Bounce:

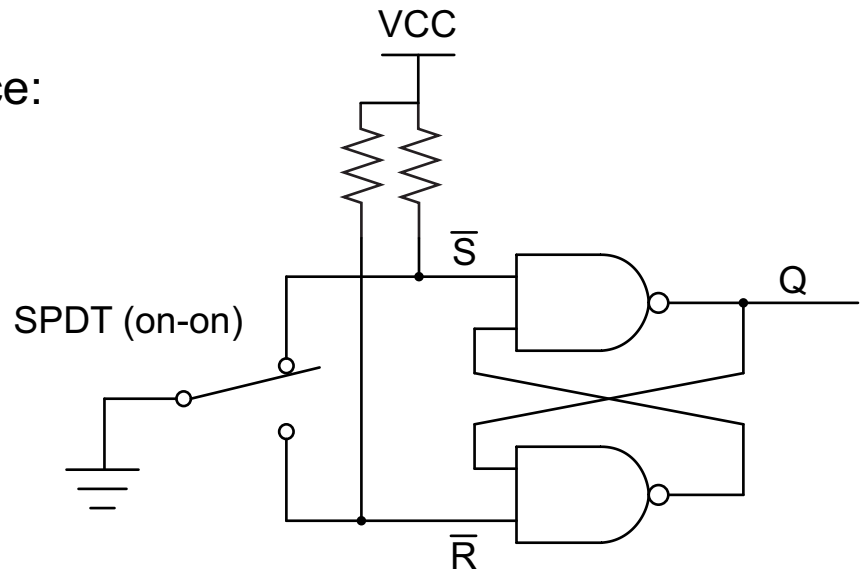




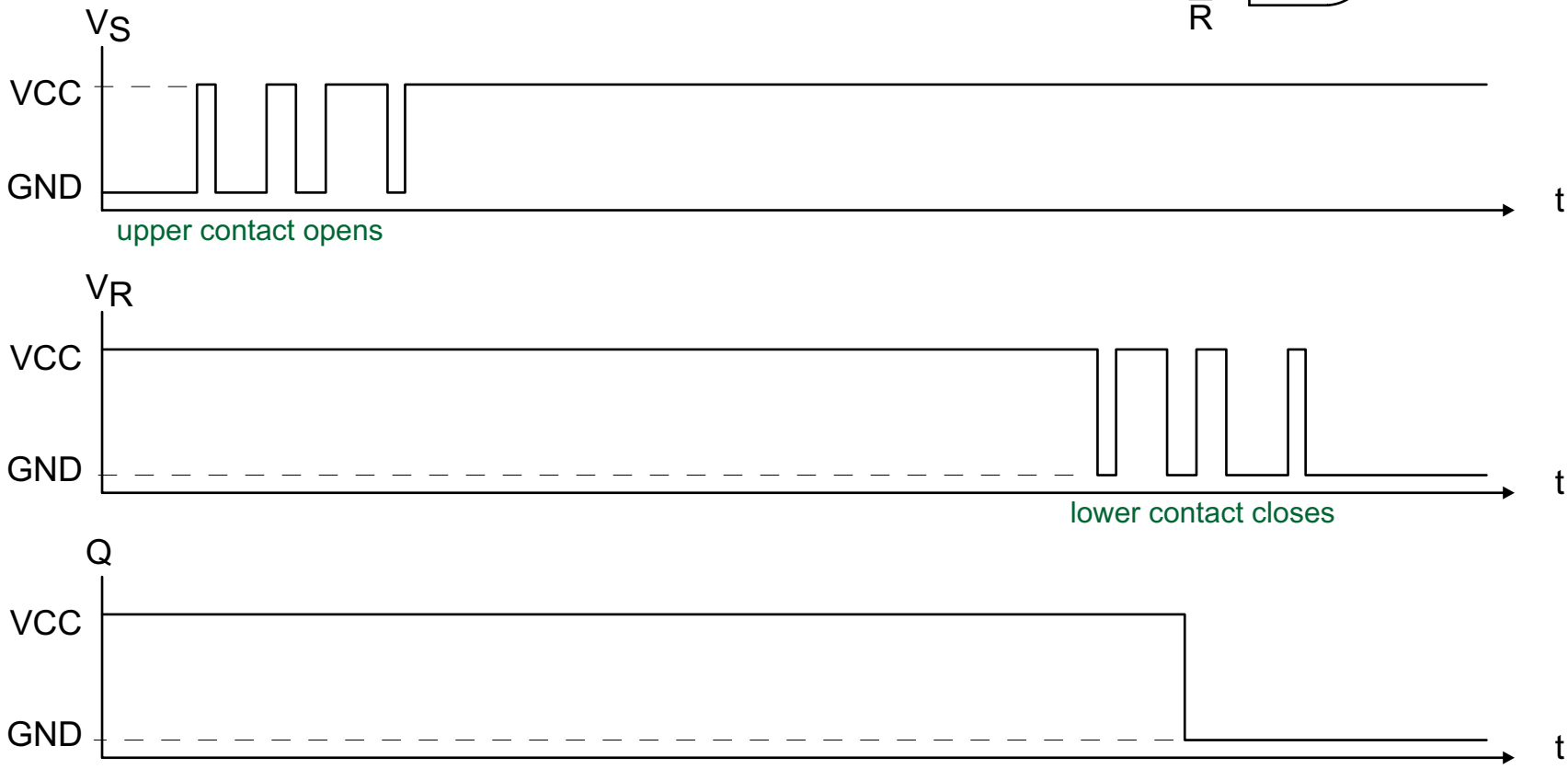
“Good” switches may bounce for 11 - 15ms.

Buttons / Switches – Hardware De-Bounce:

using a SR Latch and an SPDT switch



Example: Switch moving from top contact to bottom contact.



Note: Hardware de-bouncing can also be done with a Resistor-Capacitor circuit, but the design of these circuits is beyond the scope of this course.