LaurTec

ATmega 2560

Corso di programmazione per microcontrollori Atmel

Autore : *Marcello Pinna*

ID: UT0005-IT

Copyright © 2020 Mauro Laurenti

INFORMATIVA

Come prescritto dall'art. 1, comma 1, della legge 21 maggio 2004 n.128, l'autore avvisa di aver assolto, per la seguente opera dell'ingegno, a tutti gli obblighi della legge 22 Aprile del 1941 n. 633, sulla tutela del diritto d'autore.
Tutti i diritti di questa opera sono riservati. Ogni riproduzione ed ogni altra forma di diffusione al pubblico dell'opera, o parte di essa, senza un'autorizzazione scritta dell'autore, rappresenta una violazione della legge che tutela il diritto d'autore, in particolare non ne è consentito un utilizzo per trarne profitto.
La mancata osservanza della legge 22 Aprile del 1941 n. 633 è perseguibile con la reclusione o sanzione pecuniaria, come descritto al Titolo III, Capo III, Sezione II.
A norma dell'art. 70 è comunque consentito, per scopi di critica o discussione, il riassunto e la citazione, accompagnati dalla menzione del titolo dell'opera e dal nome dell'autore.

AVVERTENZE

I progetti presentati non hanno la marcatura CE, quindi non possono essere utilizzati per scopi commerciali nella Comunità Economica Europea.

Chiunque decida di far uso delle nozioni riportate nella seguente opera o decida di realizzare i circuiti proposti, è tenuto pertanto a prestare la massima attenzione in osservanza alle normative in vigore sulla sicurezza.
L'autore declina ogni responsabilità per eventuali danni causati a persone, animali o cose derivante dall'utilizzo diretto o indiretto del materiale, dei dispositivi o del software presentati nella seguente opera.
Si fa inoltre presente che quanto riportato viene fornito così com'è, a solo scopo didattico e formativo, senza garanzia alcuna della sua correttezza.

L'autore ringrazia anticipatamente per la segnalazione di ogni errore.

Tutti i marchi citati in quest'opera sono dei rispettivi proprietari.

Indice

Capitolo VII - Esempi....4
Programmi di esempio....4
Modalità NORMAL con interrupt....5
Modalità CTC (Clear Time on Compare)....10
Modalità' FAST PWM....13
Modalità PHASE CORRECT PWM....16
History....19

Capitolo VII - Esempi

Programmi di esempio

Ora di applicherà in pratica tutto ciò che è stato descritto in modo teorico in precedenza, con ulteriori chiarimenti. Il tutto può sembrare apparentemente complicato ma in realtà non lo è affatto, basta avere le idee chiare sui termini utilizzati e consultare i vari registri descritti. Riporto innanzitutto in Figura 1 lo schema elettrico, valido per tutti gli esempi che seguiranno.

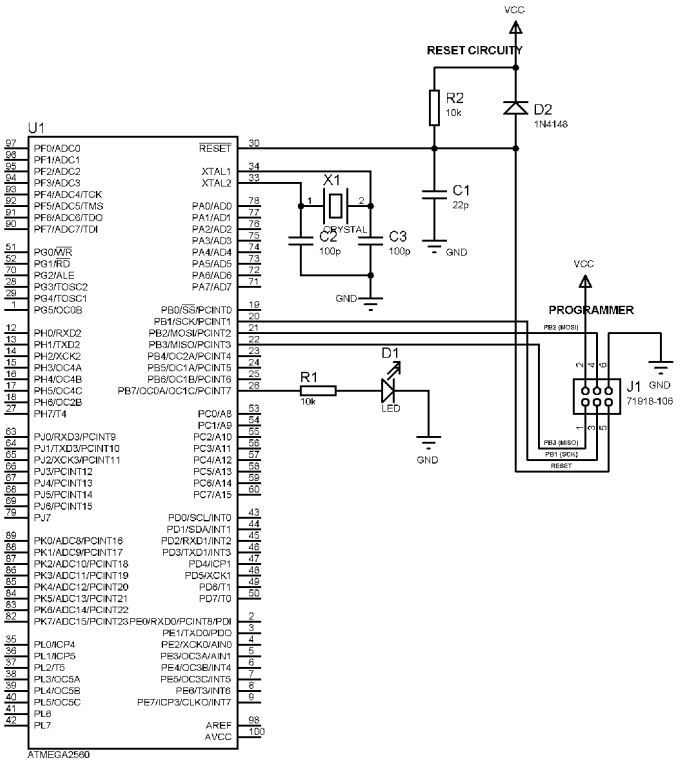

Figura 1: *Schema elettrico usato negli esempi.*

Nelle schede Arduino il Timer0 - Counter0 è destinato esclusivamente alla gestione delle temporizzazioni (delay) del sistema, in quanto è programmato con una ISR (*Interrupt Service Routine*) che esegue costantemente la funzione principale "*millis*".

Modalità NORMAL con interrupt

In questo primo esempio si utilizza l'interrupt generato dal bit TOV0, per la cattura dell'*overflow*. Questo bit è un Flag di segnalazione di interrupt, presente nel registro TIFR0 (Timer0/Counter0 *Interrupt Flag Register*). Innanzitutto ricordo che ogni interrupt dell'Atmega ha un suo indirizzo (*Interrupt Vector*) a cui è abbinata una funzione specifica (*handler*) che in questo caso corrisponde a ISR(TIMER0_OVF_vect), dove OVF significa OVerFlow (vettore del timer0)

```
//  Interrupt Service Routine del TIMER0
ISR(TIMER0_OVF_vect){

 // qui va scritto il programma di gestione dell'ISR
}
```

ISR(TIMER0_OVF_vect) non farà altro che eseguire la stessa routine ogni qualvolta il timer TCNT0 andrà in *overflow*, contemporaneamente il bit TOV0 viene resettato.

Il problema da risolvere è il seguente: si vuole accendere e spegnere ogni secondo un LED, collegato al pin PB7. Come visto il Timer0, con un prescaler di 1024, può arrivare a garantire un tempo massimo Tmax di 0,016384 secondi. Per cui raggiungere 1s sembrerebbe una cosa impossibile da fare. Questo limite si supera facilmente utilizzando la routine di interrupt; infatti, basta contare il numero degli *overflow* totali da essa generati ed impostare il timer TCNT0 con un numero di conteggi iniziale per raggiungere 1s esatto, o controllare il valore finale prima dell'ultimo *overflow*. Per calcolare matematicamente questi due parametri è necessario conoscere la frequenza di clock in ingresso. Nella demoboard usata nel corso, è presente un quarzo con una frequenza *fc* = 16 MHz, quindi servono questi dati:

fc (espressa in Hz) = 16000000 Hz

T da raggiungere = 1s = 1000 ms

a questo punto si sceglie un prescaler, ad esempio:

N = 256
si calcola il valore della frequenza in uscita (*fc*/N) con il prescaler = 256

fc / N = 16000000 / 256 = 62500 Hz

si calcola il valore di Tc corrispondente:

Tc = 1 / (*fc* / N) = 1 / 62500 = 0,000016 secondi = 0,016 ms

Considerando che impostando il prescaler = 256, il tempo necessario per un *overflow* è pari a Tmax = 0,004096s = 4,096 ms, di può calcolare il numero di *overflow*.

numero degli *overflow* = T/Tmax = 1000 / 4,096 = 244, 140

il numero intero 244, ricavato da questa formula, è uguale al numero degli *overflow* totali. Ricordo che il registro ad 8 bit lavora solo con numeri interi, non potrebbe essere diversamente. Poiché è inferiore a 255 lo accetto come parametro valido, se così non fosse stato si sarebbe dovuto scegliere il prescaler successivo.

A questo punto si moltiplica il numero degli *overflow* per Tmax:

244 * 4,096 = 999,424 ms

e si fa la sottrazione tra 1000 ms (il tempo richiesto) ed il risultato ottenuto dal calcolo precedente (999,424 ms)

1000 - 999,424 = 0,576 ms

si ottiene come risultato della sottrazione 0,576 e lo si divide per Tc

numero dei conteggi = 0,576 / 0,016 = 36

il numero 36 è il numeri dei conteggi dell'ultimo *overflow*. Questo valore assieme a 244 *overflow*, ci fornisce 1s di ritardo.

Ora si hanno tutti i dati necessari per scrivere il programma. Nel `main` si definisce la frequenza di clock e si includono i file header. Si imposta quindi il Timer0 con un prescaler uguale a 256 e si azzera il contatore:

```
// Definisco la frequenza della CPU
#define F_CPU  16000000UL

// Includo i file di testata per la gestione I/O e dell'interrupt
#include <avr/io.h>
#include <avr/interrupt.h>

// imposto il timer0 con prescaler = 256
TCCR0B |= (1 << CS02);

// inizializzo il contatore
TCNT0 = 0;
```

Si abilita l'interrupt sull'*overflow* e poi l'interrupt globale. TOIE0 (Timer0 *Interrupt Enable*) è un bit presente nel registro TIMSK0 che abilita l'interrupt sull'*overflow*, mentre I-bit è un bit presente nel registro SREG che abilita l'interrupt globale. Quando entrambi vengono impostati ad 1, l'interrupt viene generato.

```
// abilito l'interrupt sugli overflow
TIMSK0 |= (1 << TOIE0);

// abilito l'interrupt globale
SREG|=0x80;    // 0b10000000
```

Ora si azzera la variabile per il conteggio degli *overflow*, variabile che verrà incrementata

all'interno dell'*Interrupt Service Routine* assegnata al Timer.

```
// Inizializzo la variabile per il conteggio degli overflow
tot_overflow = 0;

//  Interrupt Service Routine del TIMER0
//  (chiamata ogni qualvolta TCNT0 va in overflow)
ISR(TIMER0_OVF_vect){

// incremento il numero degli overflow avvenuti
tot_overflow++;
}
```

All'interno del `main` si scrivono le istruzioni di verifica, ovvero se il numero degli *overflow* è uguale a 244, ed il numero dei conteggi del timer è uguale a 36. Ogni secondo avverrà l'inversione logica dello stato del LED. Qui sotto è riportato il listato completo dell'esempio:

```
/*
 * Created: 08/05/2018 17:41:20
 * Author : marcello
 */

#pragma config BODLEVEL = BODLEVEL_2V7
#pragma config OCDEN = CLEAR
#pragma config JTAGEN = SET
#pragma config SPIEN = SET
#pragma config WDTON = CLEAR
#pragma config EESAVE = CLEAR
#pragma config BOOTSZ = BOOTSZ_4096W_1F000
#pragma config BOOTRST = SET
#pragma config CKDIV8 = CLEAR
#pragma config CKOUT = CLEAR
#pragma config SUT_CKSEL = SUT_CKSEL_EXTXOSC_8MHZ_XX_16KCK_65MS

// Definisco la frequenza della CPU
#define F_CPU   16000000UL

// Definisco le funzioni per accendere e spegnere il LED
#define LED_ON     PORTB |=  (1<<PORTB7)
#define LED_OFF    PORTB &= ~(1<<PORTB7)

// Definisco una bitmask per la verifica del pulsante premuto
#define BUTTON_PRESSED   0b10000000

// Includo i file di testata per la gestione I/O e dei ritardi
// Definisco la frequenza della CPU
#define F_CPU   16000000UL

// Includo i file di testata per la gestione I/O e dell'interrupt
#include <avr/io.h>
#include <avr/interrupt.h>

// Variabile globale per il conteggio del numero degli overflow
volatile char tot_overflow;
```

```
// initializzo il timer0
void timer0_init(){

 // imposto il timer0 con prescaler = 256
 TCCR0B |= (1 << CS02);

 // Inizializzo il contatore
 TCNT0 = 0;

 // abilito l'interrupt sugli overflow
 TIMSK0 |= (1 << TOIE0);

 // abilito l'interrupt globale
 SREG|=0x80;    // 0b10000000

 // Inizializzo la variabile per il conteggio degli overflow
 tot_overflow = 0;
}

//  Interrupt Service Routine del TIMER0
//  (chiamata ogni qualvolta TCNT0 va in overflow)
ISR(TIMER0_OVF_vect){

 // incremento il numero degli overflow avvenuti
 tot_overflow++;
}

int main(void){

 // Imposto PB7 come uscita
 // DIGITAL PIN 13
 DDRB |= (1 << DDB7);

 // inizializzo il timer
 timer0_init();

 while(1){

    // (CALCOLO DA FOGLIO DI LAVORO TIMER0)
    // verifico che il numero degli overflow sia uguale a 244
    if (tot_overflow >= 244){

    // verifico che il conteggio del contatore del timer0 (TCNT0)
   //sia uguale a 36
       if (TCNT0 >= 36){

          // inverto lo stato del led ogni secondo (1000 ms)
          PORTB ^= (1 << PORTB7);

          // azzero i contatori
          TCNT0 = 0;
          tot_overflow = 0;
       }
    }
 }
}
```

Questa è un'immagine del tempo richiesto, ricavata dall'oscilloscopio, durante l'esecuzione del programma. Come si può notare, si sono rispettate le specifiche del progetto.

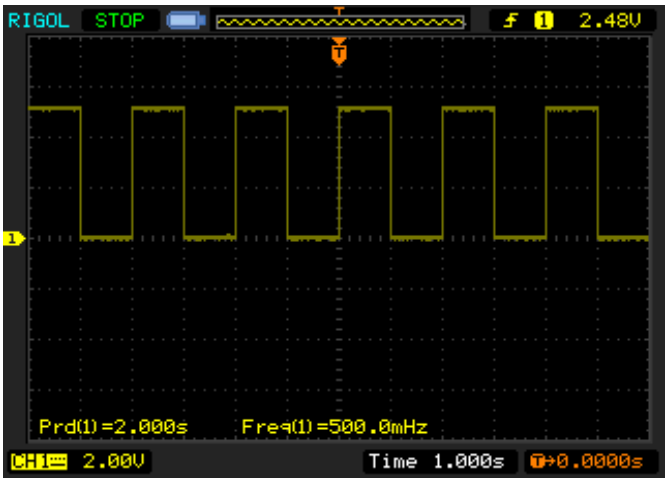

Figura 2: *Verifica del tempo di accensione del LED.*

Il ritardo massimo richiesto tramite l'utilizzo della routine di interrupt non può superare i 4,194304s. Questo limite è dovuto alla risoluzione massima (8 bit) del contatore.

Modalità CTC (Clear Time on Compare)

Prima di presentare l'esempio è bene chiarire alcuni concetti a mio avviso fondamentali: anche questa modalità può essere usata per ottenere un tempo non superiore a 0,016384 secondi, utilizzando gli stessi calcoli visti in precedenza per la modalità *Normal*. Nel caso si dovesse utilizzare il pin PB7 (OC0A) non è necessario un evento d'interrupt. Nel caso si utilizzi un pin diverso si deve procedere come per l'esempio precedente. La modalità CTC è usata soprattutto per generare frequenze in uscita a OC0A, con un *duty cycle* uguale al 50%. La generazione della frequenza richiesta si ottiene impostando un valore intero, non superiore a 255, nel registro OCR0A mentre il *duty cycle*. si ottiene ponendo ad 1 il solo bit COM0A0 (*Toggle on Compare Match*). Il calcolo di questo valore viene eseguito applicando la seguente formula:

$$OCR0A = \left[\frac{fc}{2 * N * F_{OC0A}} \right] - 1$$

dove:
fc = frequenza di clock
N = prescaler
F_{OC0A} = frequenza richiesta

Supponendo di voler ottenere una frequenza di 20 KHz con *fc* = 16000000Hz e con il prescaler più basso possibile, ovvero 1, il calcolo è presto fatto:

OCR0A = (16000000 / (2 * 1 * 20000)) - 1 = (16000000 / 40000) – 1 = 400 -1 = 399

Poiché questo valore supera 255 va scartato e si procede applicando lo stesso calcolo al prescaler successivo, ovvero 8.

OCR0A = (16000000 / (2 * 8 * 20000)) - 1 = (16000000 / 320000) – 1 = 50 -1 = 49

In questo caso il valore 49 è da ritenersi valido, visto che è inferiore a 256 e può essere scritto nel registro OCR0A

Per quanto riguarda la frequenza minima di lavoro, generata con il prescaler più alto, essa si calcola in questo modo: sapendo che il Tmax ottenuto con prescaler 1024 è uguale a 0,016384 secondi, si evince che il periodo P è uguale a 0,016384 * 2 = 0,032768 secondi e di conseguenza la frequenza ottenuta è uguale ad 1 / 0,032768 = 30,51758125Hz.

Il listato dell'esempio che segue genera un'onda quadra di 20 KHz sul pin PB7 con *duty cycle* uguale al 50%, come dai calcoli appena visti. Data la sua semplicità non credo sia necessario descrivere le righe di codice essenziali per la sua impostazione, i commenti dovrebbero chiarire il tutto.

```
#pragma config BODLEVEL = BODLEVEL_2V7
#pragma config OCDEN = CLEAR
#pragma config JTAGEN = SET
#pragma config SPIEN = SET
#pragma config WDTON = CLEAR
#pragma config EESAVE = CLEAR
#pragma config BOOTSZ = BOOTSZ_4096W_1F000
#pragma config BOOTRST = SET
#pragma config CKDIV8 = CLEAR
#pragma config CKOUT = CLEAR
#pragma config SUT_CKSEL = SUT_CKSEL_EXTXOSC_8MHZ_XX_16KCK_65MS


// Definisco la frequenza della CPU
#define F_CPU 16000000  // 16 MHz

// Includo i file di testata per la gestione I/O
#include <avr/io.h>

int main(void){

 // Imposto PB7  (OC0A)
 // DIGITAL PIN 13
 DDRB |= (1<<DDB7);

 // Imposto il modo CTC
 TCCR0A |= (1<<WGM01);

 // Imposto modalità Toggle OC0A on Compare Match
 TCCR0A |= (1<<COM0A0);

 // Imposto il Prescaler = 8
 TCCR0B |= (1<<CS01);

 // Azzero il contatore
 TCNT0 = 0;

  // Imposto il registro OCR0A con un valore (n° di conteggi)
  // calcolato per ottenere una    frequenza di 20 KHz
 OCR0A = 49;

 while(1){

 }
}
```

Dall'immagine in Figura 3, ricavata dall'oscilloscopio, si può vedere chiaramente che la frequenza misurata è esattamente quella calcolata.

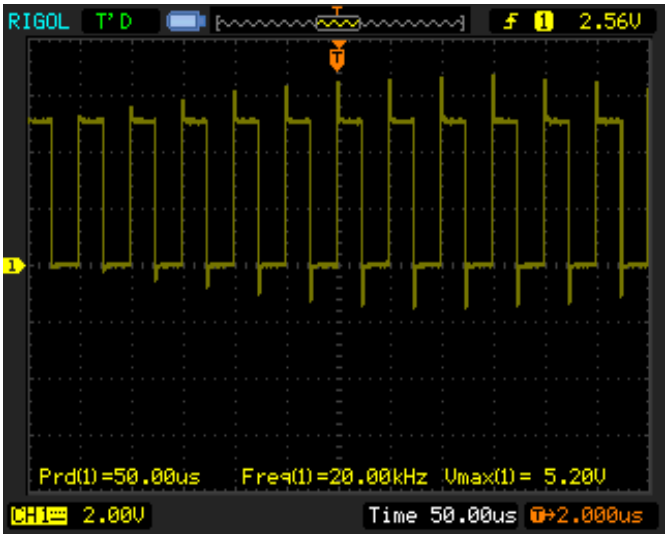

Figura 3: *Verifica della frequenza del segnale in uscita dal pin PB7.*

Prima di proseguire con gli altri esempi credo sia meglio approfondire alcuni concetti. Ricordo che le modalità *Fast e Phase Correct PWM* permettono di variare la frequenza richiesta mantenendo però un *duty cycle* fisso uguale al 50% (*Toggle on Compare Match*). In questo caso la formula da utilizzare per il calcolo del valore da assegnare a OCR0A è la stessa utilizzata per la modalità CTC, ovvero:

$$OCR0A = \left[\frac{fc}{2 * N * F_{OC0A}} \right] - 1$$

dove:
fc = frequenza di clock
N = prescaler
F_{OC0A} = frequenza richiesta

Nel caso si voglia invece modificare il *duty cycle* si deve tenere presente che questo può essere fatto solo utilizzando le frequenze fisse ottenute dal prescaler, frequenze che nel caso fast sono doppie rispetto a quelle generate dalla *phase correct*. Il valore da assegnare al registro OCR0A deve essere un numero superiore a 0 ed inferiore a 256. La formula da applicare per il calcolo del *duty cycle*, espresso in percentuale e per entrambe le modalità, è la seguente:

$$D.C. = \left[\frac{(OCR0A + 1) * 100}{256} \right]$$

dove:
D.C. = *duty cycle*
OCR0A = valore da impostare nel registro OCR0A

Modalità' FAST PWM

Gli utilizzatori di Arduino sanno senz'altro che nel suo IDE è presente uno sketch che esegue il *Fade* di un LED, ovvero lo illumina gradualmente fino a raggiungere un valore massimo che poi decresce fino a spegnerlo. Il programma che segue si può definire un *clone* di questo sketch in quanto pilota il LED usando il *Fast PWM* (modo 3), in modalità non invertente e senza prescaler. Il LED deve essere collegato a OC0A (via hardware), corrispondente al pin PB7. Il valore medio della tensione in uscita a questo pin è proporzionale al valore scritto nel registro OCR0A. Il valore incrementa fino a 255 e decrementa fino a 0, in funzione di una variabile che gli viene assegnata e che rappresenta il *duty cycle*.

```
// Definisco la frequenza della CPU
#define F_CPU 16000000  // 16 MHz

// Includo i file di testata per la gestione I/O e dei ritardi
#include <avr/io.h>
#include <util/delay.h>
```

All'interno del main si imposteranno la variabile assegnata al registro OCR0A ed il pin OC0A in uscita, per pilotare il LED:

```
// Imposto la variabile per la luminosità
int brightness;

// Imposto PB7 come uscita
// DIGITAL PIN 13
DDRB  |= (1<<DDB7);
```

Per avere un Fast PWM, bisogna impostare ad 1 il bit WGM00 e WGM01 del registro TCCR0A. Per avere la modalità non invertente "*Clear OC0A on Compare Match - Set OC0A on Compare Match*" bisogna impostare ad 1 il bit COM0A1 dello stesso registro (ometto COM0A0 in quanto il suo valore è già a 0). Per impostare il prescaler ad 1 (nessun prescaler) bisogna impostare ad 1 il bit CS00 del registro TCCR0B. Il tutto si riduce alle poche istruzioni necessarie:

```
// Imposto il modo Fast PWM
TCCR0A|=(1<<WGM00)|(1<<WGM01);

// Imposto modalità Non Invertente
TCCR0A|=(1<<COM0A1);

// Volendo, si può scrivere anche così, anche se meno chiaro:
// TCCR0A|=(1<<WGM00)|   (1<<WGM01)|(1<<COM0A1);

// Nessun prescaler (prescaler = 1)
TCCR0B|=(1<<CS00);
```

Il programma intero, con relativi commenti, è riportato di seguito:

```
#pragma config BODLEVEL = BODLEVEL_2V7
#pragma config OCDEN = CLEAR
#pragma config JTAGEN = SET
#pragma config SPIEN = SET
#pragma config WDTON = CLEAR
#pragma config EESAVE = CLEAR
#pragma config BOOTSZ = BOOTSZ_4096W_1F000
#pragma config BOOTRST = SET
#pragma config CKDIV8 = CLEAR
#pragma config CKOUT = CLEAR
#pragma config SUT_CKSEL = SUT_CKSEL_EXTXOSC_8MHZ_XX_16KCK_65MS

// Definisco la frequenza della CPU
#define F_CPU 16000000  // 16 MHz

// Includo i file di testata per la gestione I/O e dei ritardi
#include <avr/io.h>
#include <util/delay.h>

int main(void){

 // Imposto la variabile per la luminosità
 int brightness;

 // Imposto PB7 come uscita (OC0A)
 // DIGITAL PIN 13
 DDRB|= (1<<DDB7);

 // Imposto il modo Fast PWM - modo 3 - TOP = 0xFF
 TCCR0A|=(1<<WGM00)|(1<<WGM01);

 // Imposto modalità Non Invertente
 TCCR0A|=(1<<COM0A1);

 // Nessun prescaler (prescaler = 1)- frequenza = 62,5 KHz
 TCCR0B|=(1<<CS00);

 while(1){

    // Aumento la luminosità da 0 a 254
    for (brightness = 0; brightness < 255; ++brightness){
       // Imposto il registro OCR0A con il valore della luminosità
      //ovvero imposto il duty cycle non espresso in percentuale
       OCR0A = brightness;

       // piccolo ritardo per rendere il tutto visibile all'occhio umano
       _delay_ms(10);
    }

    // Diminuisco la luminosità fino a portarla a 0
    for (brightness = 255; brightness > 0; --brightness){
       OCR0A = brightness;
       _delay_ms(10);
    }
 }
}
```

In Figura 4 si può vedere un'immagine ricavata dall'oscilloscopio durante l'esecuzione del programma, in particolare il *duty cycle* è circa 80%.

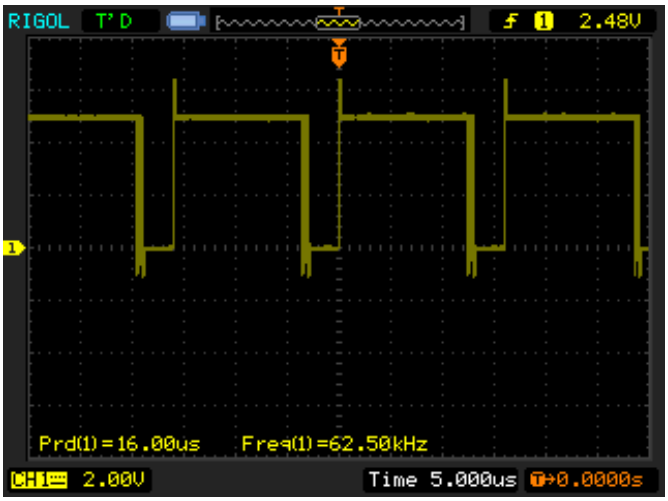

Figura 4: *Verifica del PWM e relativo duty cycle.*

Eseguendo la contro verifica, ovvero applicando la formula inversa per il calcolo della frequenza in modalità *Fast PWM*, estrapolata dal datasheet e riportata nel paragrafo ad essa dedicato, con un quarzo da 16 MHz, si ha che:

fc = 16000000 Hz
N = 1
16000000 / 1 * 256 = 62500 Hz = 62,5 KHz

Anche in questo caso, la frequenza misurata dall'oscilloscopio corrisponde a quella calcolata.

Modalità PHASE CORRECT PWM

Ultimo esempio, di cui riporto solo il listato, genera un segnale PWM sul pin PB7 con frequenza di 20KHz, impostando il registro OCR0A con il valore precedentemente calcolato con la formula della modalità CTC, ovvero:

$$OCR0A = \left[\frac{fc}{2 * N * F_{OC0A}} \right] - 1$$

dove:
fc = frequenza di clock
N = prescaler
F_{OC0A} = frequenza richiesta

```
#pragma config BODLEVEL = BODLEVEL_2V7
#pragma config OCDEN = CLEAR
#pragma config JTAGEN = SET
#pragma config SPIEN = SET
#pragma config WDTON = CLEAR
#pragma config EESAVE = CLEAR
#pragma config BOOTSZ = BOOTSZ_4096W_1F000
#pragma config BOOTRST = SET
#pragma config CKDIV8 = CLEAR
#pragma config CKOUT = CLEAR
#pragma config SUT_CKSEL = SUT_CKSEL_EXTXOSC_8MHZ_XX_16KCK_65MS


// Definisco la frequenza della CPU
#define F_CPU 16000000  // 16 MHz

// Includo i file di testata per la gestione I/O e dei ritardi
#include <avr/io.h>
#include <util/delay.h>

int main(void){
 // Imposto la variabile per la luminosità
 int brightness;

 // Imposto PB7 come uscita (OC0A)
 // DIGITAL PIN 13
 DDRB |= (1<<DDB7);

 // imposta il Timer0 in modo Phase Correct (su OCR0A) - modo 1
 TCCR0A |= (1<< WGM00);

 // Pulisco OC0A all'evento compare quando il timer incrementa
 // Imposto OC0A all'evento compare quando il timer decrementa
 TCCR0A |= (1<<COM0A1);

 // Prescaler = 8 - frequenza = 3,906 KHz
 TCCR0B |= (1<<CS01);

 while(1){
    // Aumento la luminosità da 0 a 254
    for (brightness = 0; brightness < 255; ++brightness){
       // Imposto il registro OCR0A con il valore della luminosità,
```

```
        // ovvero imposto il duty cycle non espresso in percentuale
        OCR0A = brightness;

        // piccolo ritardo per rendere il tutto visibile all'occhio umano
        _delay_ms(10);
    }

    // Diminuisco la luminosità fino a portarla a 0
    for (brightness = 255; brightness > 0; --brightness){
        OCR0A = brightness;

        _delay_ms(10);
    }
 }
}
```

In Figura 5 è mostrata l'immagine ricavata dall'oscilloscopio durante l'esecuzione del programma.

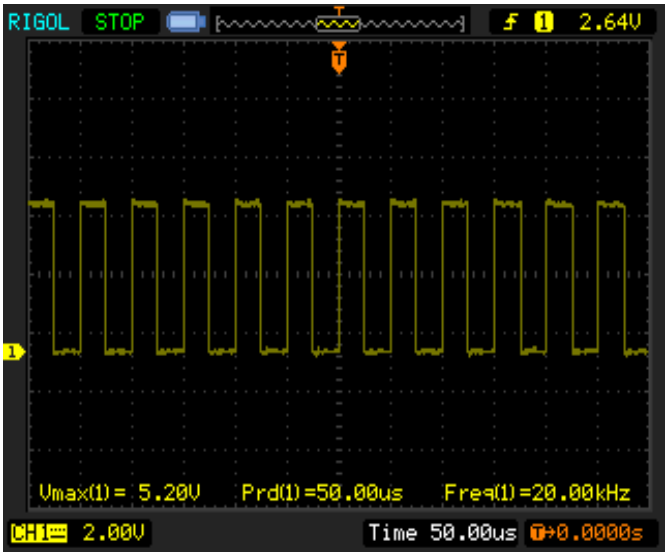

Figura 5: *Verifica del PWM e relativa frequenza di 20KHz.*

Applicando la formula inversa per il calcolo della frequenza, si ottiene lo stesso risultato sperimentale:

f_c = 16000000 Hz
N = 8
OCR0A = 49

16000000 / (2 * 8 * (1+49)) = 16000000 / (16 * 50) = 16000000 / 800 = 20000 Hz

Indice Alfabetico

A
Arduino....4, 13
C
COM0A0....10, 13
COM0A1....13
CTC....10
D
delay....4
H
handler....5
I
Interrupt Flag Register....5
interrupt globale....6
Interrupt Vector....5
ISR....4
L
l-bit....6
M
millis....4
N
Normal....10
O
OC0A....10, 13
OCR0A....10
OVF....5
P
phase correct....12
Phase Correct PWM....12
prescaler....5
S
sketch....13
SREG....6
T
TCCR0B....13
TIFR0....5
Timer0....4
TIMER0_OVF_vect....5
TIMSK0....6
TOIE0....6
TOV0....5

History

Data	Versione	Autore	Revisione	Descrizione Cambiamento
25/06/20	1.0	Marcello Pinna	Mauro Laurenti	Versione Originale.