# SERVOCOMANDI E TECNICA PWM

**Maggio 2012**

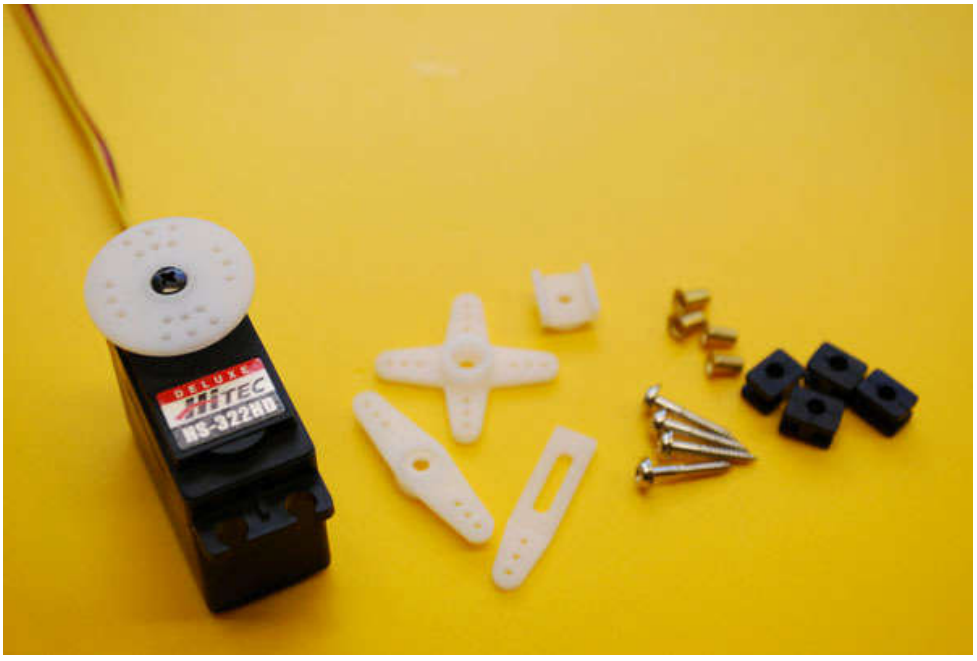
I **servocomandi**, o più semplicemente **servo**, sono molto usati nel modellismo e nella robotica amatoriale. Si tratta di piccoli dispositivi, generalmente contenuti in uno scatolotto di materiale plastico, muniti di un motorino, una serie di ingranaggi di riduzione e circuiteria. Parleremo quindi in questa brief-note, dei servocomandi "da hobby" anche noti come **servocomandi RC**, perché generalmente utilizzati in abbinamento alle riceventi dei RadioComandi. Più che altro la denominazione di "Servocomandi RC" assicura che i servo funzionino seguendo un preciso standard di comunicazione.

Dal corpo di un servocomando fuoriesce un perno di rotazione, al quale possono essere fissate, mediante un incastro zigrinato e una vite centrale, le cosiddette "squadrette" (*Horn* in inglese), di varie forme, materiali e dimensioni e che servono per collegare il servocomando al dispositivo che esso dovrà "muovere". Le squadrette generalmente sono forate per permettere l'ancoraggio, tramite viti autofilettanti, a staffe oppure per permetterci di inserire al loro interno gli alberini di comando. Per applicazioni di robotica sono spesso necessarie anche delle Staffe (*Frames* in inglese) in alluminio, plastica o materiali vari, da ancorare alle squadrette tonde, che permettono di realizzare articolazioni complesse formate da più servo.

Il perno di rotazione è in grado di effettuare un'escursione minima garantita di 90°, ma in genere si arriva tranquillamente a 180°. Il servo ha una posizione centrale e può ruotare di 45° (o 90°) a destra e 45° (o 90°) a sinistra.

Attenzione: io qui mi riferisco spesso ad una rotazione di 90° (45-0-45), per motivi di sicurezza, ma al 99.9999% il servo che avete tra le mani in questo momento è in grado di ruotare di 180° (90-0-90), anche se alcuni servo economici sforzano arrivati alle estremità.

Questi dispositivi sono molto utilizzati, ad esempio, sulle automobili radiocomandate per aprire/chiudere la valvola del gas, per effettuare la sterzata, oppure ancora sugli aeroplani radiocomandati o ancora nei piccoli robot in cui vanno a costituire, mediante squadrette, staffe e leveraggi vari, le varie giunture degli arti dei robot.



I servocomandi, a dispetto delle loro ridotte dimensioni, grazie a sistemi di riduzione ad ingranaggi più o meno sofisticati, hanno delle forze torcenti spaventose e **sono in grado di spostare diversi Kg** (tanto per fare un esempio, l'Hitec HS-805BB è in grado di spostare circa 25Kg, ma alcuni modelli di servo arrivano anche oltre 45Kg).

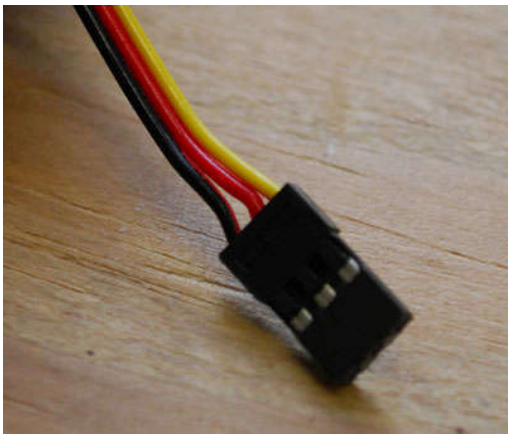
Un'altra peculiarità è quella di mantenere la posizione: una volta *detto* al servocomando di ruotare di un certo numero di gradi, e se tale comando viene inviato di continuo, rimarrà in quella posizione trattenendo il carico a cui è collegato. Ovviamente bisogna sempre scegliere un servocomando che abbia la potenza necessaria a svolgere il compito assegnatogli. Spesso, nei servocomandi economici, capita che gli ingranaggi, generalmente costruiti in Nylon, si spacchino proprio perchè il servocomando viene utilizzato al limite della propria forza. Per tale motivo esistono servocomandi con ingranaggi metallici o materiali plastici di un certo livello. Quello della foto in alto, ad esempio, utilizza ingranaggi in Karbonite: un brevetto della Hitec che permette di avere un buon compromesso tra prezzo, peso e resistenza.

Bisogna inoltre distinguere tra servocomandi analogici e digitali. In questa sede tratterò solo gli analogici perché sono i più diffusi e più economici. Vengono chiamati analogici in quanto il posizionamento del perno viene fatto controllando la rotazione di un piccolo potenziometro posto all'interno del servocomando e messo in movimento dagli stessi ingranaggi che muovono il perno. Nei servi analogici il motorino non è alimentato quando il servo si trova nella posizione comandata. I servo digitali, invece, hanno sistemi digitali per controllare il posizionamento e forniscono quindi una risposta più rapida e precisa, hanno altre opzioni tra le quali il settaggio della velocità, il controllo di assorbimento, temperatura ecc (possono essere programmati", raggiungono forze torcenti più elevate ma per contro costano un occhio della testa e consumano più corrente in quanto il motorino è sempre alimentato. In ogni caso **i servocomandi RC sia digitali che analogici seguono lo stesso standard di comunicazione**: per cui generalmente sono intercambiabili.

Il servocomando, abbiamo detto, ha quindi una rotazione limitata: 90° (45 a destra e 45 a sinistra o 90° a destra e sinistra per i servo che lo consentono), questo perché hanno all'interno un perno che ne blocca l'escursione.

In molti effettuano la modifica dei servocomandi rimuovendone i blocchi: in questo caso il servocomando può essere utilizzato, ad esempio, come un *motoriduttore* per collegarci su delle ruote per movimentare il nostro robottino o la nostra macchinina: è una soluzione valida ed economica. Arriviamo quindi alla parte cruciale: **come si pilota un servocomando**?

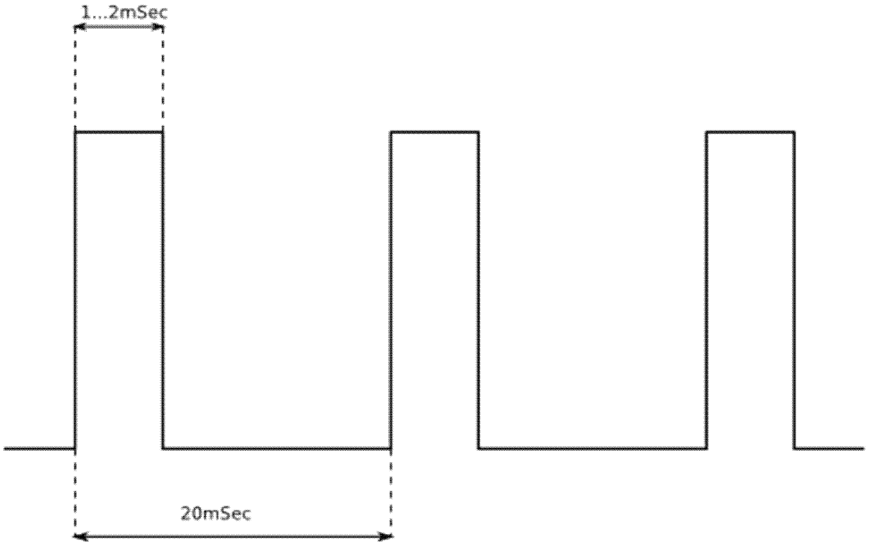
Da un servocomando fuoriescono 3 fili: due sono per l'alimentazione e generalmente sono nero o marrone per il negativo e rosso per il positivo; il terzo filo, quello destinato al **segnale di comando**, generalmente è giallo (Hitec), bianco (Futaba) o arancione o nero.



Cavetto di un servo prodotto dalla Hitec. La piedinatura segue quella standard da 2,54mm. Non tutti i servocomandi, però, presentano il connettore fatto in questa maniera

L'alimentazione generalmente è compresa tra 4.8 e 6volts (ma fate riferimento al datasheet del vostro servocomando) . Alla massima tensione il servo fornisce il massimo della potenza.

Il segnale di comando è costituito da un'onda quadra inviata ripetutamente: il fronte positivo deve avere una durata compresa tra 1 e 2 (tra 0,5 e 2,5 per i servo che ruotano di 180°) millisecondi e la somma del fronte positivo e quello negativo (ovvero: il periodo) deve essere di circa 20mSec (frequenza: 50 Hz). Il segnale fatto in questo modo deve essere inviato di continuo se si vuole che il servocomando, sotto sforzo, mantenga la posizione desiderata:




Possiamo notare che tale segnale è in pratica un segnale di tipo **PWM** (**Pulse Width Modulation**) : per pilotare un servocomando andiamo a modulare la durata della semionda positiva.

I servocomandi in grado di ruotare di 180° accettano un impulso tra 0,5 e 2,5mSec.

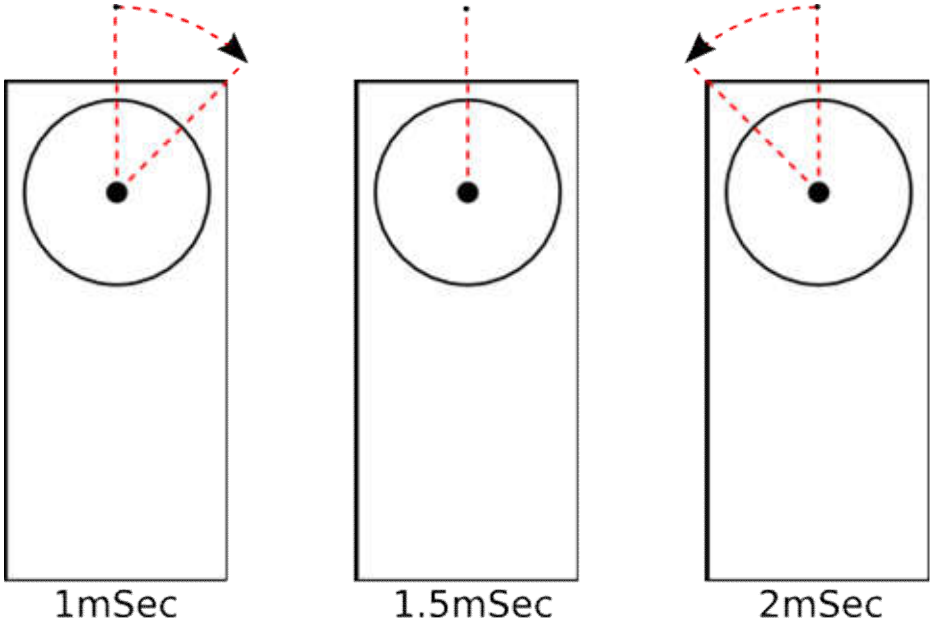
Ne approfitto quindi per introdurre il concetto di Duty Cycle (d): esso rappresenta il rapporto, espresso in percentuale, tra la durata della semionda positiva (τ) e il periodo (T) del segnale. Quando il fronte positivo dura 1mSec, essendo il periodo di 20mSec, il duty cycle varrà:

$$d = \frac{\tau}{T} \cdot 100 = \frac{1\text{mSec}}{20\text{mSec}} \cdot 100 = 5\%$$

Possiamo quindi dire che un servocomando va pilotato mediante un segnale modulato in PWM alla frequenza di 50Hz, con un duty cycle compreso tra 5% e 10% . Valori al di fuori di questi possono portare a malfunzionamenti: il servo fischia, si surriscalda e può rompersi.

In realtà alcuni servocomandi possono essere pilotati non curandosi del periodo di 20mSec ma solo della esatta durata della semionda positiva: quella negativa potrà variare tra 10 e 40mSec circa, ma tale comportamento non è standard. Utilizzando un periodo (e quindi una durata totale) di 20mSec saremo certi di riuscire a far funzionare correttamente qualsiasi tipo di servocomando senza correre il rischio di romperlo.

Con un duty cycle del 5% (quindi: impulso di 1mSec), il servo si ruoterà tutto da un lato (45° a destra), con un duty cycle dal 10% (impulso di 2mSec), il servo si ruoterà tutto dall'altro lato (45° a sinistra). Il servo raggiungerà quindi la posizione centrale con un impulso di 1,5mSec:




I servo che hanno un'escursione di 180° ruoteranno di 90° a destra con un impulso di 0,5mSec e 90° a sinistra con un impulso da 2,5mSec. In pratica possiamo dire che c'è una rotazione di 45° ogni 0,5mSec.

Realizzare un'onda quadra adatta allo scopo è relativamente semplice. In questo articolo potete vedere **come pilotare un servocomando usando un picmicro** (codice sorgente in Hitech-C).

# Articoli che potrebbero interessarti

- Demo PIC10F322 : PWM + ADC + Interrupt Timer0. Moduliamo l'intensità luminosa con un trimmer
- Corso programmazione PICMicro in C – Approfondimenti – Pilotare un servocomando usando un trimmer
- [User Project] Controllare un servocomando a distanza con i moduli RF della Aurel
- CENT4UR™ – Driver di potenza per coppia di motori DC a spazzole. Esempio di pilotaggio con PICmicro
- Come Arduino gestisce il PWM. Esempio pilotaggio CENT4UR™ con Arduino
- Generare segnali PWM con i moduli Output Compare sui PICmicro a 16 e 32 bit
- Scheda controllo motori by Laurtec
- AIR Module Booster Pack
- Easy Bee – Guida alla scelta e alla comprensione dei moduli XBee. Free EBook
- chipKIT Basic I/O Shield™

## Il PWM su Arduino

Arduino possiede 3 **moduli OC** (**Output Compare**), che fanno capo a 6 uscite PWM (pin 3, 5, 6, 9, 10 e 11: ogni modulo ha 2 uscite). I 3 moduli sono contraddistinti dalle sigle OC0, OC1 e OC2 e vengono gestiti ognuno da un diverso timer:

| Pin Arduino | Pin ATMega 328 | Modulo | Timer |
|---|---|---|---|
| 6 | PD6 (OC0A) | OC0 | TCCR0 |
| 5 | PD5 (OC0B) | | |
| 9 | PB1 (OC1A) | OC1 | TCCR1 |
| 10 | PB2 (OC1B) | | |
| 11 | PB3 (OC2A) | OC2 | TCCR2 |
| 3 | PD3 (OC2B) | | |

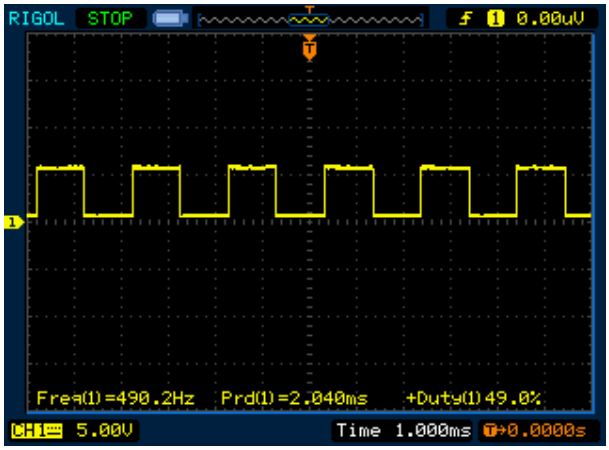
Ogni modulo, essendo pilotato da un diverso Timer, ha la possibilità di operare ad una frequenza diversa dagli altri. E' ovvio che i due pin appartenenti allo stesso modulo opereranno sempre alla stessa frequenza pur potendo variare individualmente il duty cycle.

Normalmente chi utilizza Arduino, per poter sfruttare il PWM, è abituato ad utilizzare la seguente funzione:

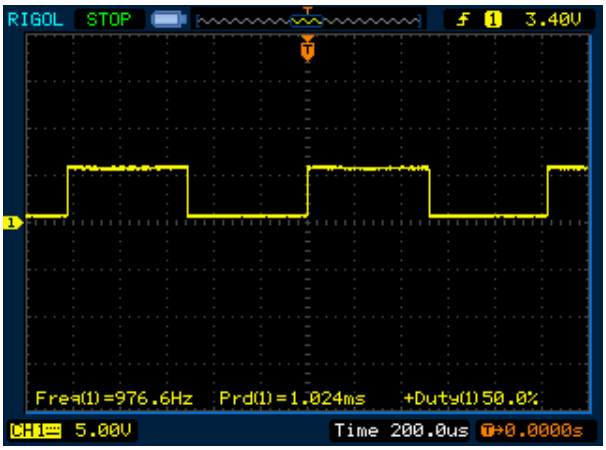
```
analogWrite(pin, valore);
```

dove *pin* assume il valore 6, 5, 9, 10, 11 o 3, e *valore* è un numero tra 0 (duty cycle:0%) e 255 (duty cycle:100%). Molti, purtroppo, ignorano la *questione della frequenza*: ok sto impostando il duty cycle… ma a che frequenza sto lavorando? E soprattutto: sono vincolato ad una frequenza fissa oppure ho la possibilità di variarla?

Di default Arduino imposta i pin 9, 10, 11 e 3 (moduli OC1 e OC2) per lavorare a 488Hz, e i pin 6 e 5 (modulo OC0) a 976Hz.




**Segnale PWM Arduino sui pin 3, 9, 10, 11**




**Segnale PWM Arduino sui pin 5 e 6**

Queste frequenze così basse potrebbero andare bene per variare la luminosità di un led ma non certo per pilotare un motore o generare un segnale analogico, come purtroppo molti fanno. Ancora peggio quando su "certi siti" si vedono collegati addirittura i driver motori a due pin che generano frequenze diverse (per esempio ho visto in giro qualcuno che sul suo fantastico sito di elettronica di terza mano collegava i motori ai pin 3 e 5, che hanno due frequenze diverse). Il sistema per poter

variare le frequenze alle quali lavorano i moduli Output Compare dell'ATmega328 è quello di agire nei registri di configurazione dei Timer che li controllano.

Partiamo col dire che per la maggior parte delle applicazioni è sconveniente variare la frequenza operativa del modulo OC0 (pin 6 e 5) dal momento che questo si appoggia al Timer0, al quale fanno riferimento anche le routine di ritardo: *ci sistemiamo* il PWM su questi pin, ma poi le varie funzioni *delay()*, *millis()*, libreria servo e probabilmente molte altre funzioni che fanno uso del Timer0, produrranno risultati del tutto inaspettati.

I Timers (che nei documenti Atmel vengono più propriamente chiamati *Timer/Counter*) sull'ATmega328 hanno due registri di controllo: **TCCRxA** e **TCCRxB** (dove x è 0,1 o 2). Il registro A controlla più propriamente le modalità operative dei Timer (normale, Fast PWM, Phase-Correct PWM ecc, per le quali vi rimando ai link a fine articolo). Il registro B consente, tra le altre cose, di impostare il prescaler agendo sui bit 0,1 e 2, chiamati **CS0**, **CS1** e **CS2**. La soluzione più rapida di tutte è quella di modificare unicamente il prescaler dei Timer lasciando invariate tutte le altre impostazioni, agendo nei registri TCCRxB, utilizzando la seguente comoda funzione disponibile sul playground di Arduino:

```
void setPwmFrequency(int pin, int divisor) {
  byte mode;
  if(pin == 5 || pin == 6 || pin == 9 || pin == 10) {
    switch(divisor) {
      case 1: mode = 0x01; break;
      case 8: mode = 0x02; break;
      case 64: mode = 0x03; break;
      case 256: mode = 0x04; break;
      case 1024: mode = 0x05; break;
      default: return;
    }
    if(pin == 5 || pin == 6) {
      TCCR0B = TCCR0B & 0b11111000 | mode;
    } else {
      TCCR1B = TCCR1B & 0b11111000 | mode;
    }
  } else if(pin == 3 || pin == 11) {
    switch(divisor) {
      case    1: mode = 0x01; break;
      case    8: mode = 0x02; break;
      case   32: mode = 0x03; break;
      case   64: mode = 0x04; break;
      case  128: mode = 0x05; break;
      case  256: mode = 0x06; break;
      case 1024:mode = 0x07; break;
      default: return;
    }
    TCCR2B = TCCR2B & 0b11111000 | mode;
  }
}
```

A dispetto di quello che potrebbe far capire il nome della funzione (*setPwmFrequency*), non andiamo ad impostare direttamente la *frequenza del PWM* bensì il **divisore del prescaler** associato al timer che controlla il pin.

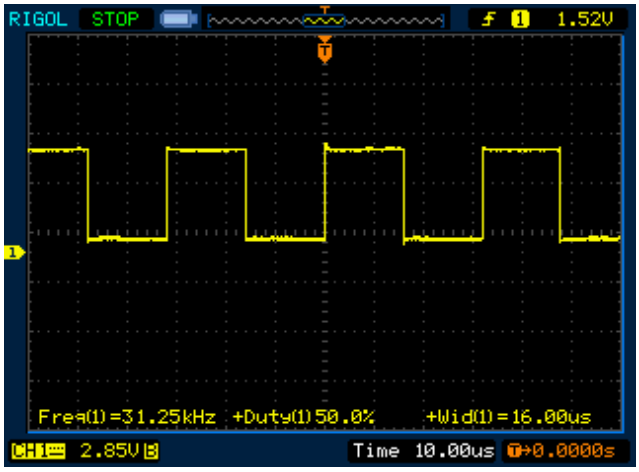
Per i pin 9,10, 11 e 3 si parte dalla frequenza di base di **31250Hz**, per cui su tali pin impostando il parametro *divisor* sul valore 64, ad esempio, otterremo una frequenza PWM pari a 31250/64 = 488Hz (la frequenza di default).

La frequenza di base per i pin 6 e 5 è invece **62500Hz**, per cui impostando anche qui *divisor* a 64, ad esempio, otteniamo 62500/64=976Hz (di nuovo la frequenza di default!). I valori validi per il parametro *divisor* sono: 1, 8, 64, 256 e 1024. Per i pin 3 e 11 sono disponibili anche i valori 32 e 128 oltre a quelli elencati prima.

I valori 32 e 128 per i pin 5,6,9 e 10, non sono disponibili perché i Timers/Counter 0 e 1 hanno due impostazioni per i bit CS0÷CS2 che prevedono il clocking dei timer da una sorgente esterna, opzione non disponibile per il Timer/Counter2 che in compenso ha due valori di divisore in più. Vedi datasheet completo nei link.

## Pilotaggio CENT4UR™ con Arduino

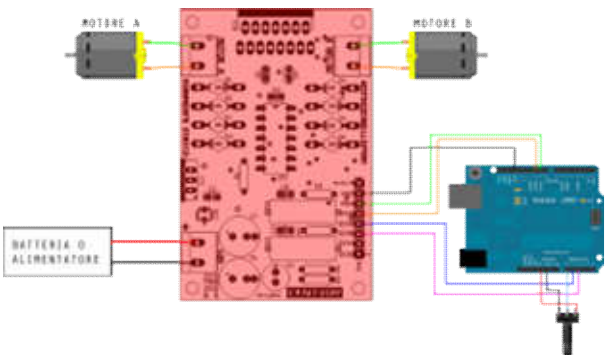
Ho detto prima che non intendo modificare la frequenza del PWM sui pin 6 e 5 dato che potrei avere bisogno di funzioni di ritardo o di utilizzare servocomandi. Per pilotare i miei motori userò la minima frequenza ultrasonica disponibile su Arduino: 31250Hz impostando il prescaler a 1 sul Timer/Counter1 (pin 9 e 10)




A tal scopo, facendo uso della funzione sopra esposta, nel setup imposto i pin per il PWM nel seguente modo:

```
// PWM sui pin 9 e 10 a 31,2KHz
setPwmFrequency(9, 1);
setPwmFrequency(10, 1);
```

Non c'è bisogno di impostare i pin del PWM come uscite dal momento che la funzione analogWrite già esegue questa operazione ogniqualvolta viene richiamata. Lo schema di collegamento per poter sfruttare l'esempio, allegato a fine articolo, con CENT4UR™ è il seguente:



**Esempio collegamento CENT4UR su Arduino. Parte del disegno è stata realizzata con Fritzing**

- GND -> GND
- PWA -> 9
- PWB -> 10
- SENA -> A1
- SENB -> A2

Utilizzare quindi un trimmer o un potenziometro lineare da 10K e collegare il polo centrale su A0 di Arduino e le estremità a 5V e GND. La batteria (o la tensione di alimentazione di CENT4UR™/motori) va ovviamente scelta in base alla tensione di funzionamento dei motori. Fate riferimento al manuale operativo di CENT4UR™ per maggiori informazioni.

Ruotando il potenziometro si varia il valore di duty cycle. La lettura analogica fornisce un valore a 10bit (da 0 a 1023) mentre il valore di duty cyle è a 8 bit (da 0 a 255), per cui il valore letto dal potenziometro viene scalato a 8 bit utilizzando la funzione *map* di Arduino. Si passa dal valore 0 (motori a massima velocità in un verso), al valore 128 (motori fermi) per arrivare al valore 255 (motori a massima velocità nel verso opposto).

Aprendo il terminale seriale, impostato alla velocità di default di 9600bps, è possibile leggere il valore di duty cycle e l'assorbimento dei motori espresso in mA

```
Duty cycle: 60 Assorbimento motore A: 130mA, motore B: 100mA
Duty cycle: 60 Assorbimento motore A: 110mA, motore B: 80mA
Duty cycle: 60 Assorbimento motore A: 110mA, motore B: 130mA
Duty cycle: 60 Assorbimento motore A: 130mA, motore B: 70mA
Duty cycle: 60 Assorbimento motore A: 80mA, motore B: 0mA
Duty cycle: 60 Assorbimento motore A: 80mA, motore B: 130mA
Duty cycle: 60 Assorbimento motore A: 120mA, motore B: 90mA
Duty cycle: 60 Assorbimento motore A: 160mA, motore B: 80mA
Duty cycle: 60 Assorbimento motore A: 110mA, motore B: 130mA
Duty cycle: 60 Assorbimento motore A: 120mA, motore B: 200mA
Duty cycle: 60 Assorbimento motore A: 140mA, motore B: 110mA
Duty cycle: 60 Assorbimento motore A: 150mA, motore B: 110mA
Duty cycle: 60 Assorbimento motore A: 100mA, motore B: 110mA
Duty cycle: 60 Assorbimento motore A: 90mA, motore B: 100mA
```

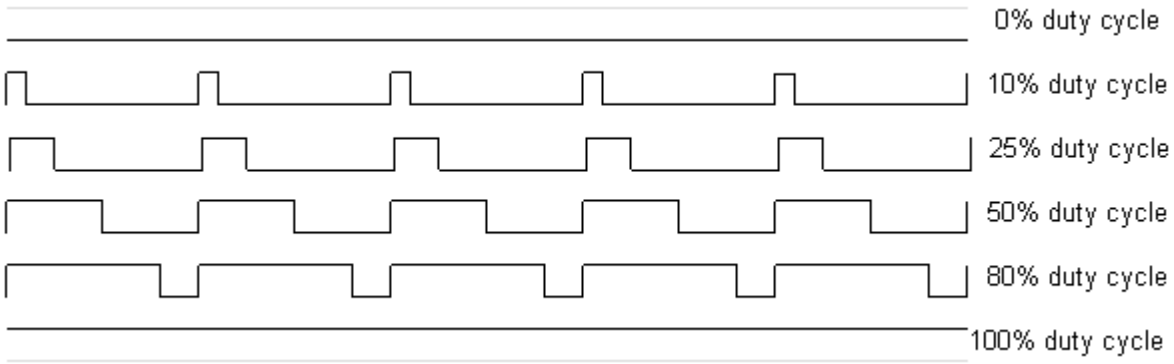
CENT4UR™ è un driver motori di piccola/media potenza. Se ci tieni a settorezero.com, **richiedi un PCB di CENT4UR™ e contribuisci a mantenere settorezero.com sempre attivo.**

# Secrets of Arduino PWM

by Ken Shirriff
with further editing by Paul Badger
the original document

Pulse-width modulation (PWM) can be implemented on the Arduino in several ways. This article explains simple PWM techniques, as well as how to use the PWM registers directly for more control over the duty cycle and frequency. This article focuses on the Arduino Diecimila and Duemilanove models, which use the ATmega168 or ATmega328.

If you're unfamiliar with Pulse Width Modulation, see the tutorial. Briefly, a PWM signal is a digital square wave, where the frequency is constant, but that fraction of the time the signal is on (the duty cycle) can be varied between 0 and 100%.




# PWM examples

PWM has several uses:

- Dimming an LED
- Providing an analog output; if the digital output is filtered, it will provide an analog voltage between 0% and 100% .
- Generating audio signals.
- Providing variable speed control for motors.
- Generating a modulated signal, for example to drive an infrared LED for a remote control.

## Simple Pulse Width Modulation with analogWrite

The Arduino's programming language makes PWM easy to use; simply call analogWrite(pin, dutyCycle), where dutyCycle is a value from 0 to 255, and pin is one of the PWM pins (3, 5, 6, 9, 10, or 11). The analogWrite function provides a simple interface to the hardware PWM, but doesn't provide any control over frequency. (Note that despite the function name, the output is a digital signal, often referred to as a square wave.)

Probably 99% of the readers can stop here, and just use analogWrite, but there are other options that provide more flexibility.

# Bit-banging Pulse Width Modulation

You can "manually" implement PWM on any pin by repeatedly turning the pin on and off for the desired times. e.g.

```
void setup()
{
  pinMode(13, OUTPUT);
}

void loop()
{
  digitalWrite(13, HIGH);
  delayMicroseconds(100); // Approximately 10% duty cycle @ 1KHz
  digitalWrite(13, LOW);
  delayMicroseconds(1000 - 100);
}
```

This technique has the advantage that it can use any digital output pin. In addition, you have full control the duty cycle and frequency. One major disadvantage is that any interrupts will affect the timing, which can cause considerable jitter unless you disable interrupts. A second disadvantage is you can't leave the output running while the processor does something else. Finally, it's difficult to determine the appropriate constants for a particular duty cycle and frequency unless you either carefully count cycles, or tweak the values while watching an oscilloscope.

A more elaborate example of manually PWMing all pins may be found here.

# Using the ATmega PWM registers directly

The ATmega168P/328P chip has three PWM timers, controlling 6 PWM outputs. By manipulating the chip's timer registers directly, you can obtain more control than the analogWrite function provides.

The AVR ATmega328P datasheet provides a detailed description of the PWM timers, but the datasheet can be difficult to understand, due to the many different control and output modes of the timers.

A word on the relationship between the Arduino language and the datasheet may be in order here.

## The Atmega 168/328 timers.

The ATmega328P has three timers known as Timer 0, Timer 1, and Timer 2. Each timer has two output compare registers that control the PWM width for the timer's two outputs: when the timer reaches the compare register value, the corresponding output is toggled. The two outputs for each timer will normally have the same frequency, but can have different duty cycles (depending on the respective output compare register).

Each of the timers has a prescaler that generates the timer clock by dividing the system clock by a prescale factor such as 1, 8, 64, 256, or 1024. The Arduino has a system clock of 16MHz and the

timer clock frequency will be the system clock frequency divided by the prescale factor. Note that Timer 2 has a different set of prescale values from the other timers.

The timers are complicated by several different modes. The main PWM modes are "Fast PWM" and "Phase-correct PWM", which will be described below. The timer can either run from 0 to 255, or from 0 to a fixed value. (The 16-bit Timer 1 has additional modes to supports timer values up to 16 bits.) Each output can also be inverted.

The timers can also generate interrupts on overflow and/or match against either output compare register, but that's beyond the scope of this article. Timer Registers Several registers are used to control each timer. The Timer/Counter Control Registers TCCRnA and TCCRnB hold the main control bits for the timer. (Note that TCCRnA and TCCRnB do not correspond to the outputs A and B.) These registers hold several groups of bits:

- Waveform Generation Mode bits (WGM): these control the overall mode of the timer. (These bits are split between TCCRnA and TCCRnB.)
- Clock Select bits (CS): these control the clock prescaler
- Compare Match Output A Mode bits (COMnA): these enable/disable/invert output A
- Compare Match Output B Mode bits (COMnB): these enable/disable/invert output B

The Output Compare Registers OCRnA and OCRnB set the levels at which outputs A and B will be affected. When the timer value matches the register value, the corresponding output will be modified as specified by the mode.

The bits are slightly different for each timer, so consult the datasheet for details. Timer 1 is a 16-bit timer and has additional modes. Timer 2 has different prescaler values.

## Fast PWM

In the simplest PWM mode, the timer repeatedly counts from 0 to 255. The output turns on when the timer is at 0, and turns off when the timer matches the output compare register. The higher the value in the output compare register, the higher the duty cycle. This mode is known as Fast PWM Mode. The following diagram shows the outputs for two particular values of OCRnA and OCRnB. Note that both outputs have the same frequncy, matching the frequency of a complete timer cycle.

## Fast PWM Mode

The following code fragment sets up fast PWM on pins 3 and 11 (Timer 2). To summarize the register settings, setting the waveform generation mode bits WGM to 011 selects fast PWM. Setting the COM2A bits and COM2B bits to 10 provides non-inverted PWM for outputs A and B. Setting the CS bits to 100 sets the prescaler to divide the clock by 64. (Since the bits are different for the different timers, consult the datasheet for the right values.) The output compare registers are arbitrarily set to 180 and 50 to control the PWM duty cycle of outputs A and B. (Of course, you can modify the registers directly instead of using pinMode, but you do need to set the pins to output.)

```
pinMode(3, OUTPUT);
pinMode(11, OUTPUT);
TCCR2A = _BV(COM2A1) | _BV(COM2B1) | _BV(WGM21) | _BV(WGM20);
```

```
TCCR2B = _BV(CS22);
OCR2A = 180;
OCR2B = 50;
```

On the Arduino Duemilanove, these values yield:

- Output A frequency: 16 MHz / 64 / 256 = 976.5625Hz
- Output A duty cycle: (180+1) / 256 = 70.7%
- Output B frequency: 16 MHz / 64 / 256 = 976.5625Hz
- Output B duty cycle: (50+1) / 256 = 19.9%

The output frequency is the 16MHz system clock frequency, divided by the prescaler value (64), divided by the 256 cycles it takes for the timer to wrap around. Note that fast PWM holds the output high one cycle longer than the compare register value.

## Phase-Correct PWM

The second PWM mode is called phase-correct PWM. In this mode, the timer counts from 0 to 255 and then back down to 0. The output turns off as the timer hits the output compare register value on the way up, and turns back on as the timer hits the output compare register value on the way down. The result is a more symmetrical output. The output frequency will be approximately half of the value for fast PWM mode, because the timer runs both up and down.

## Phase-Correct PWM example

The following code fragment sets up phase-correct PWM on pins 3 and 11 (Timer 2). The waveform generation mode bits WGM are set to to 001 for phase-correct PWM. The other bits are the same as for fast PWM.

```
pinMode(3, OUTPUT);
pinMode(11, OUTPUT);
TCCR2A = _BV(COM2A1) | _BV(COM2B1) | _BV(WGM20);
TCCR2B = _BV(CS22);
OCR2A = 180;
OCR2B = 50;
```

On the Arduino Duemilanove, these values yield:

- Output A frequency: 16 MHz / 64 / 255 / 2 = 490.196Hz
- Output A duty cycle: 180 / 255 = 70.6%
- Output B frequency: 16 MHz / 64 / 255 / 2 = 490.196Hz
- Output B duty cycle: 50 / 255 = 19.6%

Phase-correct PWM divides the frequency by two compared to fast PWM, because the timer goes both up and down. Somewhat surprisingly, the frequency is divided by 255 instead of 256, and the duty cycle calculations do not add one as for fast PWM. See the explanation below under "Off-by-one".

## Varying the timer top limit: fast PWM

Both fast PWM and phase correct PWM have an additional mode that gives control over the output frequency. In this mode, the timer counts from 0 to OCRA (the value of output compare register A), rather than from 0 to 255. This gives much more control over the output frequency than the previous modes. (For even more frequency control, use the 16-bit Timer 1.)

Note that in this mode, only output B can be used for PWM; OCRA cannot be used both as the top value and the PWM compare value. However, there is a special-case mode "Toggle OCnA on Compare Match" that will toggle output A at the end of each cycle, generating a fixed 50% duty cycle and half frequency in this case. The examples will use this mode.

In the following diagram, the timer resets when it matches OCRnA, yielding a faster output frequency for OCnB than in the previous diagrams. Note how OCnA toggles once for each timer reset.

## Fast PWM Mode with OCRA top

The following code fragment sets up fast PWM on pins 3 and 11 (Timer 2), using OCR2A as the top value for the timer. The waveform generation mode bits WGM are set to to 111 for fast PWM with OCRA controlling the top limit. The OCR2A top limit is arbitrarily set to 180, and the OCR2B compare register is arbitrarily set to 50. OCR2A's mode is set to "Toggle on Compare Match" by setting the COM2A bits to 01.

```
pinMode(3, OUTPUT);
pinMode(11, OUTPUT);
TCCR2A = _BV(COM2A0) | _BV(COM2B1) | _BV(WGM21) | _BV(WGM20);
TCCR2B = _BV(WGM22) | _BV(CS22);
OCR2A = 180;
OCR2B = 50;
```

On the Arduino Duemilanove, these values yield:

- Output A frequency: 16 MHz / 64 / (180+1) / 2 = 690.6Hz
- Output A duty cycle: 50%
- Output B frequency: 16 MHz / 64 / (180+1) = 1381.2Hz
- Output B duty cycle: (50+1) / (180+1) = 28.2%

Note that in this example, the timer goes from 0 to 180, which takes 181 clock cycles, so the output frequency is divided by 181. Output A has half the frequency of Output B because the Toggle on Compare Match mode toggles Output A once each complete timer cycle.

## Varying the timer top limit: phase-correct PWM

Similarly, the timer can be configured in phase-correct PWM mode to reset when it reaches OCRnA. Phase-Correct PWM with OCRA top The following code fragment sets up phase-correct PWM on pins 3 and 11 (Timer 2), using OCR2A as the top value for the timer. The waveform generation mode bits WGM are set to to 101 for phase-correct PWM with OCRA controlling the top limit. The OCR2A top limit is arbitrarily set to 180, and the OCR2B compare register is arbitrarily set to 50. OCR2A's mode is set to "Toggle on Compare Match" by setting the COM2A bits to 01.

```
pinMode(3, OUTPUT);
pinMode(11, OUTPUT);
TCCR2A = _BV(COM2A0) | _BV(COM2B1) | _BV(WGM20);
TCCR2B = _BV(WGM22) | _BV(CS22);
OCR2A = 180;
OCR2B = 50;
```

On the Arduino Duemilanove, these values yield:

- Output A frequency: 16 MHz / 64 / 180 / 2 / 2 = 347.2Hz
- Output A duty cycle: 50%
- Output B frequency: 16 MHz / 64 / 180 / 2 = 694.4Hz
- Output B duty cycle: 50 / 180 = 27.8%

Note that in this example, the timer goes from 0 to 180 and back to 0, which takes 360 clock cycles. Thus, everything is divided by 180 or 360, unlike the fast PWM case, which divided everything by 181; see below for details. Off-by-one You may have noticed that fast PWM and phase-correct PWM seem to be off-by-one with respect to each other, dividing by 256 versus 255 and adding one in various places. The documentation is a bit opaque here, so I'll explain in a bit of detail.

Suppose the timer is set to fast PWM mode and is set to count up to an OCRnA value of 3. The timer will take on the values 012301230123... Note that there are 4 clock cycles in each timer cycle. Thus, the frequency will be divided by 4, not 3. The duty cycle will be a multiple of 25%, since the output can be high for 0, 1, 2, 3, or 4 cycles out of the four. Likewise, if the timer counts up to 255, there will be 256 clock cycles in each timer cycle, and the duty cycle will be a multiple of 1/256. To summarize, fast PWM divides by N+1 where N is the maximum timer value (either OCRnA or 255).

Now consider phase-correct PWM mode with the timer counting up to an OCRnA value of 3. The timer values will be 012321012321... There are 6 clock cycles in each timer cycle (012321). Thus the frequency will be divided by 6. The duty cycle will be a multiple of 33%, since the output can be high for 0, 2, 4, or 6 of the 6 cycles. Likewise, if the timer counts up to 255 and back down, there will be 510 clock cycles in each timer cycle, and the duty cycle will be a multiple of 1/255. To summarize, phase-correct PWM divides by 2N, where N is the maximum timer value.

The second important timing difference is that fast PWM holds the output high for one cycle longer than the output compare register value. The motivation for this is that for fast PWM counting to 255, the duty cycle can be from 0 to 256 cycles, but the output compare register can only hold a value from 0 to 255. What happens to the missing value? The fast PWM mode keeps the output high for N+1 cycles when the output compare register is set to N so an output compare register value of 255 is 100% duty cycle, but an output compare register value of 0 is not 0% duty cycle but 1/256 duty cycle. This is unlike phase-correct PWM, where a register value of 255 is 100% duty cycle and a value of 0 is a 0% duty cycle. Timers and the Arduino The Arduino supports PWM on a subset of its output pins. It may not be immediately obvious which timer controls which output, but the following table will clarify the situation. It gives for each timer output the output pin on the Arduino (i.e. the silkscreened label on the board), the pin on the ATmega chip, and the name and bit of the output port. For instance Timer 0 output OC0A is connected to the Arduino output pin 6; it uses chip pin 12 which is also known as PD6. Timer output Arduino output Chip pin Pin name OC0A 6 12 PD6 OC0B 5 11 PD5 OC1A 9 15 PB1 OC1B 10 16 PB2 OC2A 11 17 PB3 OC2B 3 5 PD3

The Arduino performs some initialization of the timers. The Arduino initializes the prescaler on all three timers to divide the clock by 64. Timer 0 is initialized to Fast PWM, while Timer 1 and Timer 2 is initialized to Phase Correct PWM. See the Arduino source file wiring.c for details.

The Arduino uses Timer 0 internally for the millis() and delay() functions, so be warned that changing the frequency of this timer will cause those functions to be erroneous. Using the PWM outputs is safe if you don't change the frequency, though.

The analogWrite(pin, duty_cycle) function sets the appropriate pin to PWM and sets the appropriate output compare register to duty_cycle (with the special case for duty cycle of 0 on Timer 0). The digitalWrite() function turns off PWM output if called on a timer pin. The relevant code is wiring_analog.c and wiring_digital.c.

If you use analogWrite(5, 0) you get a duty cycle of 0%, even though pin 5's timer (Timer 0) is using fast PWM. How can this be, when a fast PWM value of 0 yields a duty cycle of 1/256 as explained above? The answer is that analogWrite "cheats"; it has special-case code to explicitly turn off the pin when called on Timer 0 with a duty cycle of 0. As a consequency, the duty cycle of 1/256 is unavailable when you use analogWrite on Timer0, and there is a jump in the actual duty cycle between values of 0 and 1.

Some other Arduino models use dfferent AVR processors with similar timers. The Arduino Mega uses the ATmega1280 (datasheet), which has four 16-bit timers with 3 outputs each and two 8-bit timers with 2 outputs each. Only 14 of the PWM outputs are supported by the Arduino Wiring library, however. Some older Arduino models use the ATmega8 (datasheet), which has three timers but only 3 PWM outputs: Timer 0 has no PWM, Timer 1 is 16 bits and has two PWM outputs, and Timer 2 is 8 bits and has one PWM output. Troubleshooting It can be tricky to get the PWM outputs to work. Some tips:

- You need to both enable the pin for output and enable the PWM mode on the pin in order to get any output.
  I.e. you need to do pinMode() and set the COM bits.
- The different timers use the control bits and prescaler differently;
  check the documentation for the appropriate timer.
- Some combinations of bits that you might expect to work are reserved,
  which means if you try to use them, they won't work.
  For example, toggle mode doesn't work with fast PWM to 255, or with output B.
- Make sure the bits are set the way you think. Bit operations can be tricky,
  so print out the register values with the binary (BIN) formatter and make sure they are what you expect.
- Make sure you're using the right output pins. See the table above.
- You'll probably want a decoupling capacitor to avoid spikes on the output.

An oscilloscope is very handy for debugging PWM if you have access to one. If you don't have one, I recommend using your sound card and a program such as xoscope.

**Conclusion**

I hope this article helps explain the PWM modes of the Arduino. I found the documentation of the different modes somewhat opaque, and the off-by-one issues unexplained. Please let me know if you encounter any errors.