

Pulse Width Modulation

WHY DO WE NEED PWM?: To control the amt of power supplied to a device, also, it can be used to convey information and configure/control a device. **HOW**

The duration of "on time" is called the pulse width. To get varying analog values, you change, or modulate, that pulse width.

1) Switching Frequency: Lesser the switching frequency, higher the flickering. Higher the switching frequency lower the flickering.

2) Duty Cycle: Tells the fraction of time period for which the pulse remains in on condition.

$$D = T(\text{ON})/T$$

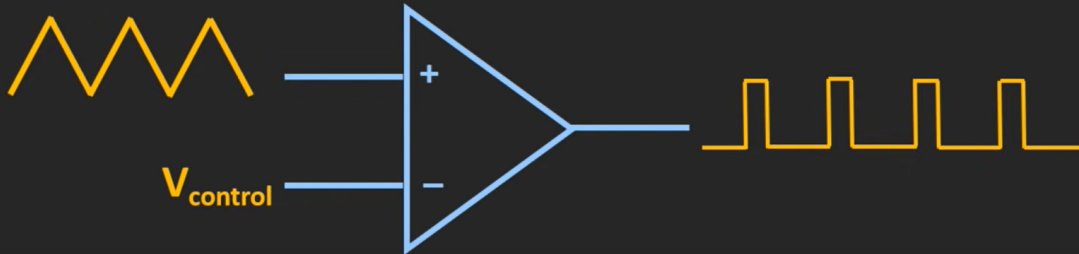
3) MOSFETs are used as electronic switches for PWM. Because during switching the power dissipated across the switch is negligible. How??

4) Methods of generating PWM:

(a) Comparator Ckt: **WHAT AND HOW??**

What is Pulse Width Modulation? How to generate PWM signal ? Pulse Width Modulation Explained

How to Generate PWM Signal



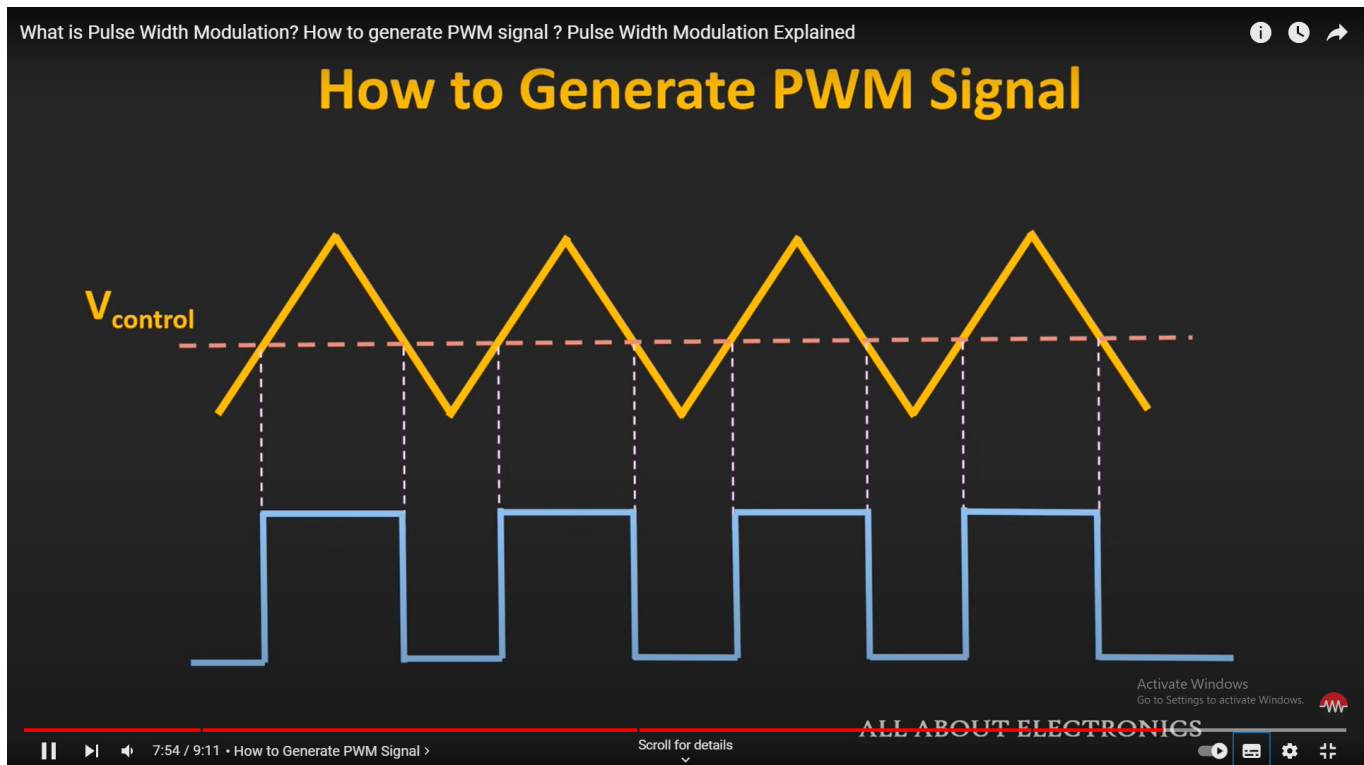
*Inverting Node

Activate Windows
Go to Settings to activate Windows.

ALL ABOUT ELECTRONICS

7:13 / 9:11 • How to Generate PWM Signal >

Scroll for details



Lower the $V_{control}$ higher is the duty cycle for the generated pulse.

NOTE: Triangular Wave is applied at the non-inverting node and $V_{control}$ is applied at the inverting node.

(b) In microcontrollers like ARDUINO:

Pulse-width modulation (PWM) can be implemented on the Arduino in several ways.

(1) Simple Pulse Width Modulation with `analogWrite()`

Use the syntax `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.)

(2) BIT BANGING PWM

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

CODE

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

```
void loop()
{
    digitalWrite(13, HIGH);
    delay(250); // Approximately 25% duty cycle
    digitalWrite(13, LOW);
    delay(1000 - 250);
}
```

This technique has the advantage that it can use any digital output pin. In addition, you have full control the duty cycle and frequency.

BUT, 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.

(3)Using the ATmega PWM Registers Directly

The ATmega328P has three timers known as Timer 0, Timer 1, and Timer 2. By manipulating the chip's timer registers directly, we can obtain more control than the `analogWrite` function provides.

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).

How the OCR Register Works

1. **Timer Counting:** The microcontroller has a timer that continuously counts up from 0 to a maximum value (TOP). For an 8-bit timer, this maximum value is 255. Once it reaches the maximum, it resets to 0 and starts counting again.
2. **PWM Signal:** The PWM signal is generated on a specific output pin. The signal is either HIGH (ON) or LOW (OFF).
3. **Output Compare Register (OCR):** The OCR register holds a value between 0 and the maximum count value (TOP). This value is used to determine when to change the state of the PWM signal within one period.

Example to Illustrate

- **Timer Period:** Suppose the timer counts from 0 to 255 (8-bit timer).
- **OCR Value:** If we set the OCR register to 127, the following happens:

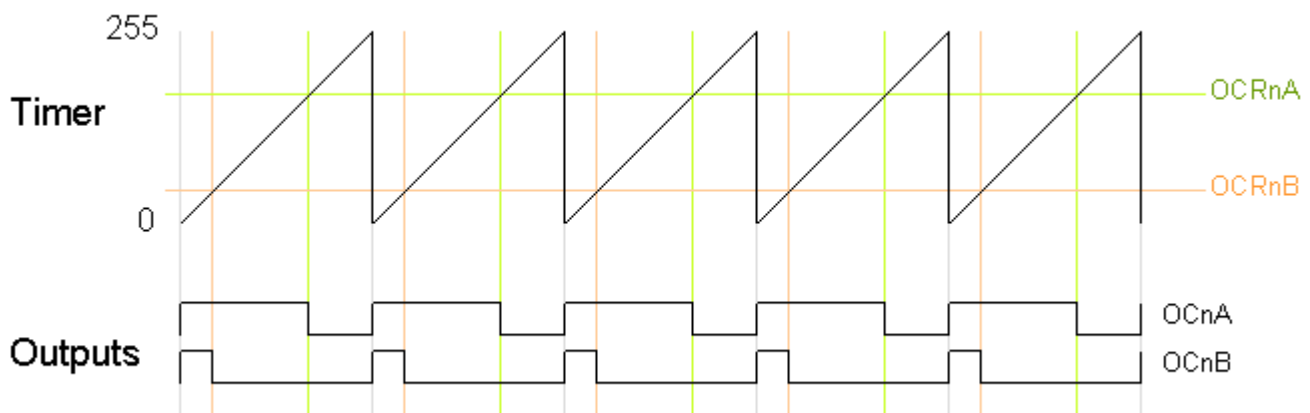
- When the timer starts from 0, the PWM signal is set to HIGH.
- As the timer counts up and reaches the value 127, the PWM signal changes to LOW.
- The signal stays LOW until the timer reaches 255 and resets to 0.
- The cycle then repeats.

The timers are complicated by several different modes. The main PWM modes are "Fast PWM" and "Phase-correct PWM"

Types of PWM:

(1) 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 frequency, matching the frequency of a complete timer cycle.



How to code fast PWM

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

Explanation:

Setting Up the Pins

```
pinMode(3, OUTPUT);  
pinMode(11, OUTPUT);
```

- `pinMode(3, OUTPUT);` : This line configures pin 3 as an output pin.
- `pinMode(11, OUTPUT);` : This line configures pin 11 as an output pin.

Configuring Timer2 for PWM

```
TCCR2A = _BV(COM2A1) | _BV(COM2B1) | _BV(WGM21) | _BV(WGM20);
```

- `TCCR2A` : This is the Timer/Counter Control Register A for Timer2. This register is used to configure the behavior of Timer2.
 - `_BV(COM2A1)` : This sets the COM2A1 bit in TCCR2A, configuring the timer to clear OC2A on compare match when up-counting (non-inverting mode for OC2A).
 - `_BV(COM2B1)` : This sets the COM2B1 bit in TCCR2A, configuring the timer to clear OC2B on compare match when up-counting (non-inverting mode for OC2B).
 - `_BV(WGM21)` : This sets the WGM21 bit, selecting one of the Fast PWM modes (WGM21 and WGM20 together set Fast PWM mode with TOP at 0xFF).
 - `_BV(WGM20)` : This sets the WGM20 bit, completing the selection of Fast PWM mode.

Setting the Prescaler

```
TCCR2B = _BV(CS22);
```

- `TCCR2B` : This is the Timer/Counter Control Register B for Timer2. This register is used to configure the clock source and the prescaler for Timer2.
 - `_BV(CS22)` : This sets the CS22 bit, configuring the timer's prescaler to 64. With an Arduino clocked at 16 MHz, this means the timer frequency is $(\frac{16 \text{ MHz}}{64} = 250 \text{ kHz})$.

Setting Output Compare Registers

```
OCR2A = 180;  
OCR2B = 50;
```

- `OCR2A` : This is the Output Compare Register A for Timer2. It sets the duty cycle for the PWM signal on pin 11 (OC2A).

- `OCR2A = 180;` : Sets the value at which the timer will compare to toggle the PWM output. With the timer counting from 0 to 255, a value of 180 means the duty cycle will be approx 70.6%).
- `OCR2B` : This is the Output Compare Register B for Timer2. It sets the duty cycle for the PWM signal on pin 3 (OC2B).
 - `OCR2B = 50;` : Sets the value at which the timer will compare to toggle the PWM output. With the timer counting from 0 to 255, a value of 50 means the duty cycle will be approx 19.6%).

Summary

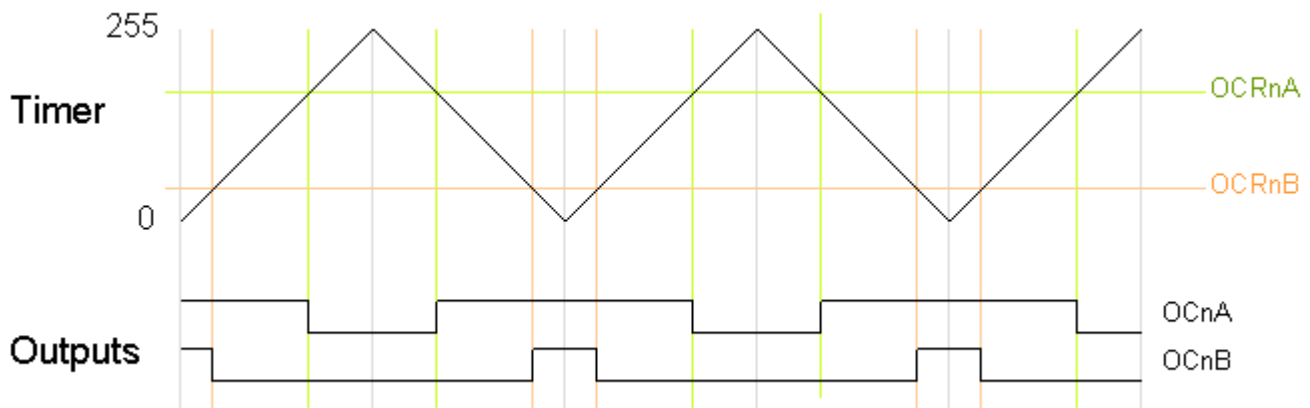
This code sets up Timer2 to generate PWM signals on pins 3 and 11 of the Arduino. Here's a summary of the setup:

- Pins 3 and 11 are configured as outputs.
- Timer2 is set to Fast PWM mode with non-inverting outputs.
- The prescaler is set to 64, resulting in a timer frequency of 250 kHz.
- The duty cycle for the PWM signal on pin 11 (OC2A) is set to approximately 70.6%.
- The duty cycle for the PWM signal on pin 3 (OC2B) is set to approximately 19.6%.

This configuration will generate PWM signals with the specified duty cycles on the respective pins.

(2)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.



How to code for Phase-Correct PWM

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

Explanation;

Setting Up the Pins

```
pinMode(3, OUTPUT);
pinMode(11, OUTPUT);
```

Configuring Timer2 for PWM

```
TCCR2A = _BV(COM2A1) | _BV(COM2B1) | _BV(WGM20);
```

- **TCCR2A** : This is the Timer/Counter Control Register A for Timer2. This register controls several aspects of Timer2, including the mode of operation and how the outputs are managed.
 - **_BV(COM2A1)** : This macro sets the COM2A1 bit in the TCCR2A register. Setting this bit configures the timer to clear the OC2A output (pin 11) on a compare match when up-counting and to set the OC2A output on a compare match when down-counting (non-inverting mode for OC2A).
 - **_BV(COM2B1)** : This macro sets the COM2B1 bit in the TCCR2A register. Setting this bit configures the timer to clear the OC2B output (pin 3) on a compare match when up-counting and to set the OC2B output on a compare match when down-counting (non-inverting mode for OC2B).
 - **_BV(WGM20)** : This macro sets the WGM20 bit in the TCCR2A register. Setting this bit selects the Fast PWM mode when combined with appropriate bits in TCCR2B. Here, WGM21 is not set, so it results in Phase Correct PWM mode.

Setting the Prescaler

```
TCCR2B = _BV(CS22);
```

- **TCCR2B** : This is the Timer/Counter Control Register B for Timer2. This register is used to configure the clock source and prescaler for Timer2.
 - **_BV(CS22)** : This macro sets the CS22 bit in the TCCR2B register. Setting this bit selects a prescaler of 64 for the timer. With the Arduino running at 16 MHz, this means the timer's frequency is divided by 64, resulting in a timer frequency of 250 kHz.

Setting Output Compare Registers

```
OCR2A = 180;
OCR2B = 50;
```

- **OCR2A** : This is the Output Compare Register A for Timer2. It determines the duty cycle of the PWM signal on pin 11 (OC2A).
 - **OCR2A = 180;** : This line sets the value of the OCR2A register to 180. With the timer counting from 0 to 255, the duty cycle for pin 11 is calculated as $(\frac{180}{255}) \approx 70.6\%$.
- **OCR2B** : This is the Output Compare Register B for Timer2. It determines the duty cycle of the PWM signal on pin 3 (OC2B).
 - **OCR2B = 50;** : This line sets the value of the OCR2B register to 50. With the timer counting from 0 to 255, the duty cycle for pin 3 is calculated as $(\frac{50}{255}) \approx 19.6\%$.

Summary

- **Pins Setup:** Pins 3 and 11 are set as output pins to be used for PWM signals.
- **Timer Configuration:**
 - **TCCR2A** is configured to:
 - Use non-inverting mode for both OC2A and OC2B.
 - Set the timer to Phase Correct PWM mode.
 - **TCCR2B** is configured to set a prescaler of 64, resulting in a timer frequency of 250 kHz.
- **PWM Duty Cycle:**
 - The duty cycle for the PWM signal on pin 11 (OC2A) is set to approximately 70.6%.
 - The duty cycle for the PWM signal on pin 3 (OC2B) is set to approximately 19.6%.

This configuration allows Timer2 to generate PWM signals with the specified duty cycles on pins 3 and 11 of the Arduino Uno.

Timer output	Arduino output
OC0A	6
OC0B	5
OC1A	9
OC1B	10
OC2A	11
OC2B	3

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.

PWM in Data Transfer

Pulse Width Modulation (PWM) plays a crucial role in various communication systems and protocols. While PWM itself is not typically used directly for communication like UART or SPI, it can be utilized as a method for encoding information within a signal.

1. Analog Data Transmission:

PWM can be used to transmit analog information over digital communication channels. By varying the duty cycle of the PWM signal, analog data such as sensor readings, audio signals, or motor speed commands can be encoded and transmitted. The receiver interprets the duty cycle variations to reconstruct the original analog signal.

2. Infrared Communication:

PWM is used in infrared (IR) communication protocols such as Pulse Position Modulation (PPM) or Infrared Remote Control (IRC). In IRC applications, for instance, different commands are encoded as variations in the PWM signal's duty cycle, allowing remote controls to send commands to electronic devices like TVs or air conditioners.

3. Audio Transmission:

In some applications, PWM is used for audio transmission. By modulating the duty cycle of the PWM signal at high frequencies, audio signals can be encoded and transmitted efficiently. While not as common as other audio transmission methods like PCM (Pulse Code Modulation), PWM can be suitable for specific low-cost applications.

TIMERS(IN DEPTH):

Types of Timers in ATmega328P

The ATmega328P has three timers:

1. **Timer/Counter0**: 8-bit timer
2. **Timer/Counter1**: 16-bit timer
3. **Timer/Counter2**: 8-bit timer

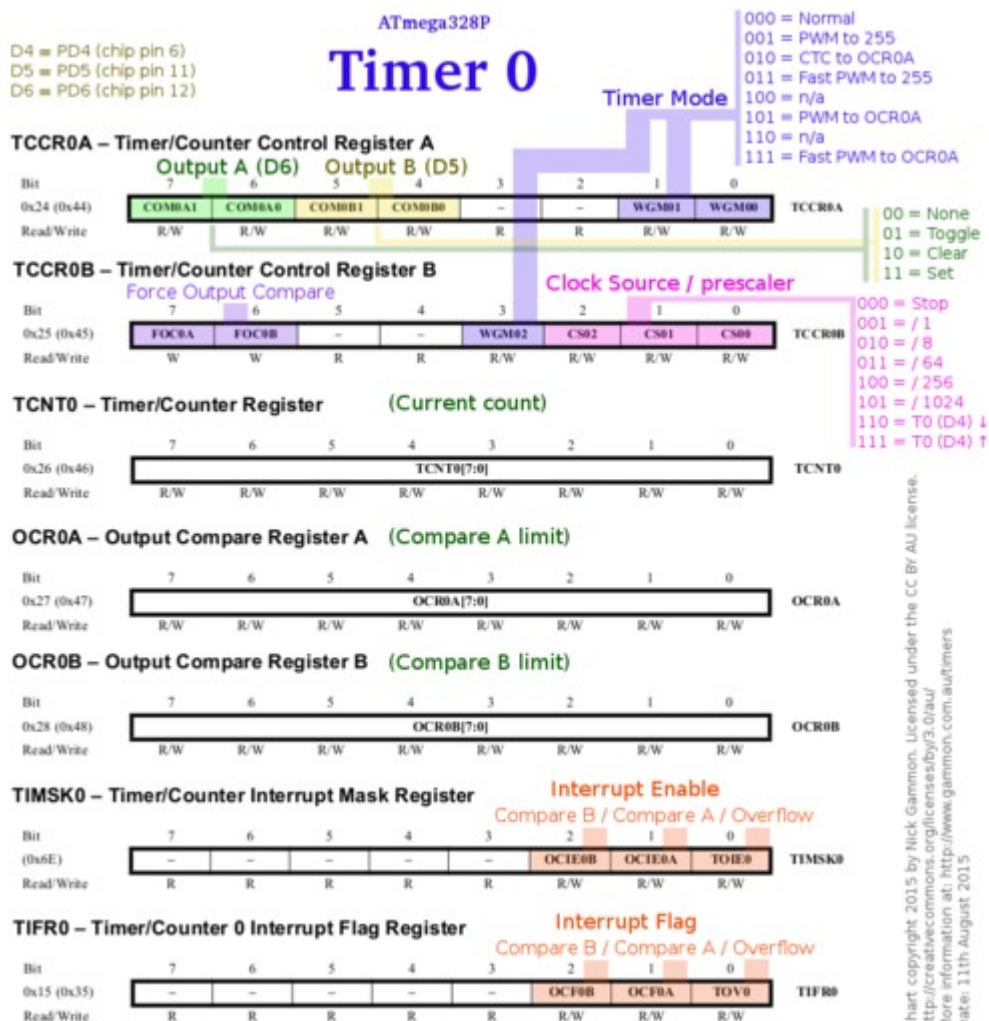
Timer Basics

- **8-bit Timer**: Can count from 0 to 255.
- **16-bit Timer**: Can count from 0 to 65535.

Timer Registers

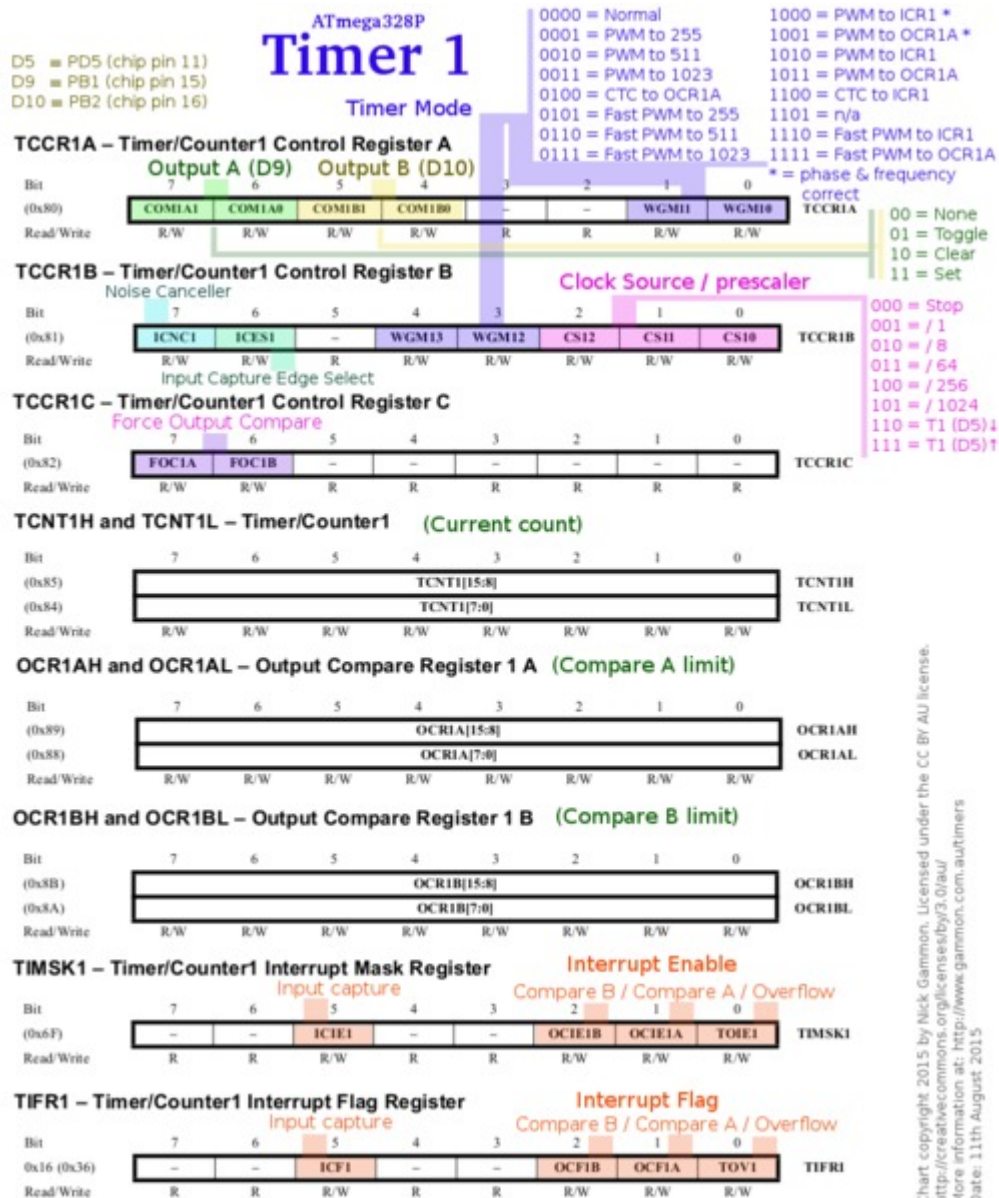
Each timer has specific registers associated with it for control and data storage.

Timer/Counter0 (8-bit)



- **TCCR0A**: Timer/Counter Control Register A
- **TCCR0B**: Timer/Counter Control Register B
- **TCNT0**: Timer/Counter Register
- **OCR0A**: Output Compare Register A
- **OCR0B**: Output Compare Register B
- **TIMSK0**: Timer/Counter Interrupt Mask Register
- **TIFR0**: Timer/Counter Interrupt Flag Register

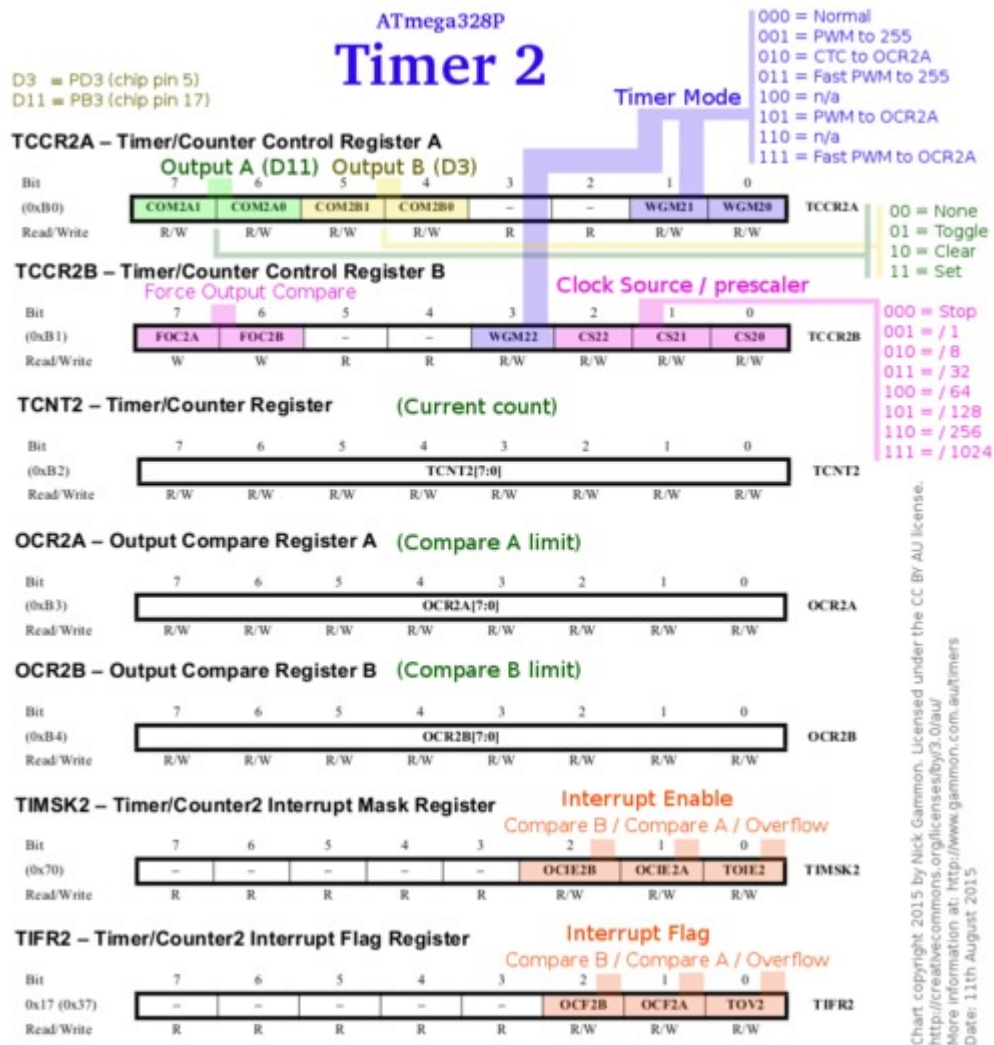
Timer/Counter1 (16-bit)



- **TCCR1A**: Timer/Counter Control Register A
- **TCCR1B**: Timer/Counter Control Register B
- **TCNT1**: Timer/Counter Register
- **OCR1A**: Output Compare Register A

- **OCR1B**: Output Compare Register B
- **ICR1**: Input Capture Register
- **TIMSK1**: Timer/Counter Interrupt Mask Register
- **TIFR1**: Timer/Counter Interrupt Flag Register

Timer/Counter2 (8-bit)



- **TCCR2A**: Timer/Counter Control Register A
- **TCCR2B**: Timer/Counter Control Register B
- **TCNT2**: Timer/Counter Register
- **OCR2A**: Output Compare Register A
- **OCR2B**: Output Compare Register B
- **TIMSK2**: Timer/Counter Interrupt Mask Register
- **TIFR2**: Timer/Counter Interrupt Flag Register

Here is a detailed explanation of the roles of each type of register:

Timer/Counter Control Registers (TCCRnA, TCCRnB)

These registers configure the mode of operation(A) and prescaling(B) for the timers.

TCCRnA (Timer/Counter Control Register A)

- **WGMn1, WGMn0 (Waveform Generation Mode):** Select the mode of operation for the timer (e.g., Normal mode, CTC mode, PWM modes).
- **COMnA1, COMnA0 (Compare Output Mode A):** Control the behavior of the OCnA output pin (e.g., toggle, clear, or set on compare match).
- **COMnB1, COMnB0 (Compare Output Mode B):** Control the behavior of the OCnB output pin.

TCCRnB (Timer/Counter Control Register B)

- **CSn2, CSn1, CSn0 (Clock Select):** Set the prescaler, determining the timer clock frequency.
- **WGMn3, WGMn2 (Waveform Generation Mode):** Together with WGMn1 and WGMn0, these bits select the timer mode.
- **ICESn (Input Capture Edge Select, Timer1 only):** Selects the edge (rising or falling) for the input capture event.
- **ICNCn (Input Capture Noise Canceler, Timer1 only):** Enables noise canceling for the input capture.

Timer/Counter Registers (TCNTn)

- **TCNTn:** This register holds the current count value of the timer. Writing to this register resets the timer count.

Output Compare Registers (OCRnA, OCRnB)

- **OCRnA, OCRnB:** These registers hold the compare values. When the timer count matches the value in OCRnA or OCRnB, certain actions can be triggered (like generating an interrupt or toggling an output pin).

Input Capture Register (ICRn, Timer1 only)

- **ICRn:** Used to capture the timer count value at the time of an external event (e.g., a rising edge on the input capture pin).

Timer Interrupt Mask Registers (TIMSKn)

- **TIMSKn**: These registers enable or disable interrupts for various timer events.

TIMSK0 (Timer/Counter Interrupt Mask Register for Timer0)

- **OCIE0A (Output Compare A Match Interrupt Enable)**: Enables interrupt on compare match A.
- **OCIE0B (Output Compare B Match Interrupt Enable)**: Enables interrupt on compare match B.
- **TOIE0 (Overflow Interrupt Enable)**: Enables interrupt on timer overflow.

TIMSK1 (Timer/Counter Interrupt Mask Register for Timer1)

- **ICIE1 (Input Capture Interrupt Enable)**: Enables interrupt on input capture event.
- **OCIE1A (Output Compare A Match Interrupt Enable)**: Enables interrupt on compare match A.
- **OCIE1B (Output Compare B Match Interrupt Enable)**: Enables interrupt on compare match B.
- **TOIE1 (Overflow Interrupt Enable)**: Enables interrupt on timer overflow.

TIMSK2 (Timer/Counter Interrupt Mask Register for Timer2)

- **OCIE2A (Output Compare A Match Interrupt Enable)**: Enables interrupt on compare match A.
- **OCIE2B (Output Compare B Match Interrupt Enable)**: Enables interrupt on compare match B.
- **TOIE2 (Overflow Interrupt Enable)**: Enables interrupt on timer overflow.

Timer Interrupt Flag Registers (TIFRn)

- **TIFRn**: These registers contain the flags for the various timer interrupts. Each flag is set when the corresponding interrupt condition occurs and must be cleared by the software.

TIFR0 (Timer/Counter Interrupt Flag Register for Timer0)

- **OCF0A (Output Compare A Match Flag)**: Set when a compare match A occurs.
- **OCF0B (Output Compare B Match Flag)**: Set when a compare match B occurs.
- **TOV0 (Overflow Flag)**: Set when the timer overflows.

TIFR1 (Timer/Counter Interrupt Flag Register for Timer1)

- **ICF1 (Input Capture Flag)**: Set when an input capture event occurs.
- **OCF1A (Output Compare A Match Flag)**: Set when a compare match A occurs.

- **OCF1B (Output Compare B Match Flag)**: Set when a compare match B occurs.
- **TOV1 (Overflow Flag)**: Set when the timer overflows.

TIFR2 (Timer/Counter Interrupt Flag Register for Timer2)

- **OCF2A (Output Compare A Match Flag)**: Set when a compare match A occurs.
- **OCF2B (Output Compare B Match Flag)**: Set when a compare match B occurs.
- **TOV2 (Overflow Flag)**: Set when the timer overflows.

Summary

- **TCCRnA, TCCRnB**: Configure timer modes, prescalers, and output compare behavior.
- **TCNTn**: Stores the current timer count.
- **OCRnA, OCRnB**: Hold compare values for generating output compare events.
- **ICRn**: Capture timer count on external events (Timer1 only).
- **TIMSKn**: Enable or disable timer interrupts.
- **TIFRn**: Indicate the occurrence of timer events and need to be cleared by software.

Timer Modes

Each timer can operate in several modes:

1. **Normal Mode**: The timer counts from 0 to its maximum value and then overflows.
2. **CTC (Clear Timer on Compare Match) Mode**: The timer is cleared to 0 when it matches a compare value.
3. **Fast PWM Mode**: Used for Pulse Width Modulation with high-frequency output.
4. **Phase Correct PWM Mode**: Generates a PWM signal with frequency controlled by a compare value.

Configuring Timers

Timers are configured by setting appropriate bits in the control registers (`TCCRnA` , `TCCRnB` , etc.). This includes setting the mode of operation, the clock source (prescaler), and enabling interrupts.

Example: Configuring Timer0 in CTC Mode

```
// Set CTC mode
TCCR0A |= (1 << WGM01);
// Set prescaler to 64
TCCR0B |= (1 << CS01) | (1 << CS00);
// Set compare value
```



```
OCR0A = 249;  
// Enable Timer Compare Interrupt  
TIMSK0 |= (1 << OCIE0A);
```

Explanation: Setting the OCIE0A bit to 1 enables the Output Compare Match A Interrupt for Timer0. This means that when the timer counter (TCNT0) matches the value in OCR0A (249), an interrupt service routine (ISR) will be triggered.

Timer Interrupts

Timers can generate interrupts on events such as overflow or compare match. These interrupts can be enabled/disabled using the `TIMSKn` register and flags can be checked using the `TIFRn` register.

Timer Applications

1. **Delays:** Using the timer to create precise time delays.
2. **PWM Generation:** Controlling motor speed, LED brightness, etc.
3. **Event Counting:** Counting external events or signals.
4. **Timing Measurements:** Measuring time intervals between events using input capture.

Smoothing of PWM:

Smoothing of PWM (Pulse Width Modulation) signals refers to the process of converting a PWM signal, which is essentially a digital signal with varying duty cycles, into a smooth analog signal. This is typically done using a low-pass filter, which removes the high-frequency components of the PWM signal, leaving behind a steady voltage that corresponds to the average value of the PWM signal.

Why Smooth PWM?

1. **Analog Control:** Many applications, such as motor control and audio signal generation, require smooth analog signals rather than a rapidly switching digital signal.
2. **Noise Reduction:** Smoothing reduces the noise and electromagnetic interference (EMI) generated by the high-frequency switching of PWM signals.
3. **Stable Output:** Provides a more stable and precise control for devices that respond better to continuous signals.

How to Smooth PWM Using Low-Pass Filter

A simple and effective way to smooth a PWM signal is to use a low-pass filter, typically consisting of a resistor and a capacitor (RC filter).

Implementation:

- The PWM signal is fed into the RC filter.
- The output of the RC filter is a smooth analog voltage.

Low-Pass Filter Basics

A low-pass filter allows low-frequency signals to pass through while attenuating high-frequency signals. The most basic type of low-pass filter is the RC (resistor-capacitor) filter.

How It Works

1. **PWM Input:** The PWM signal is a square wave that switches between 0V and the supply voltage (e.g., 5V) with a certain duty cycle.
2. **Resistor (R):** Limits the current charging and discharging the capacitor.
3. **Capacitor (C):** Stores and releases electrical energy, smoothing the input signal.

Charging and Discharging Process

The capacitor in the RC filter charges and discharges based on the duty cycle of the PWM signal, effectively averaging the high and low states to produce a smooth DC voltage corresponding to the PWM duty cycle.

Charging (When PWM is High):

When the PWM signal is high, the capacitor charges through the resistor. The voltage across the capacitor (V_C) increases exponentially towards the supply voltage V_{supply} according to the following formula:

$$V_C(t) = V_{supply}(1 - e^{-(t/RC)})$$

Discharging (When PWM is Low):

When the PWM signal is low, the capacitor discharges through the resistor. The voltage across the capacitor decreases exponentially towards 0V:

$$V_C(t) = V_{initial}(e^{-(t/RC)})$$

- ($V_{initial}$): Initial voltage across the capacitor at the start of discharge.

Steady-State Behavior

When the PWM signal switches at a high frequency relative to the RC time constant, the capacitor doesn't fully charge or discharge during each cycle. Instead, it reaches a steady-state

voltage that is the average of the PWM signal. The average voltage (V_{out}) is proportional to the duty cycle

(D) of the PWM signal:

$$V_{out} = D \cdot V_{supply}$$

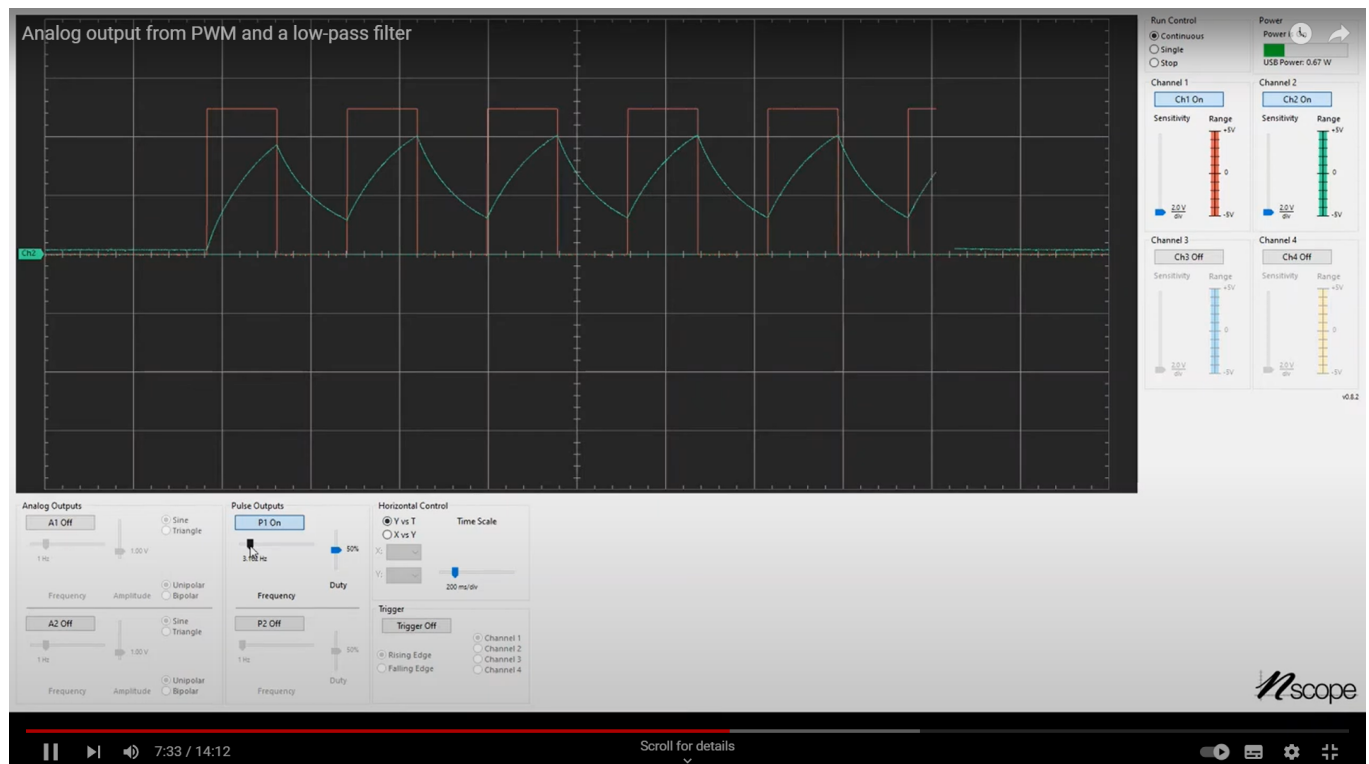
- (D): Duty cycle (0 to 1)

Output Behavior

The RC filter will smooth the PWM signal, producing a steady output voltage proportional to the duty cycle of the PWM. For a 50% duty cycle, the output will be around 2.5V. Changing the duty cycle will change the average voltage accordingly, allowing for analog control using a digital signal.

Conclusion

Smoothing a PWM signal using a low-pass filter involves using an RC circuit to average the high-frequency switching of the PWM, resulting in a steady analog voltage. This technique is widely used for generating analog signals from digital PWM outputs in applications such as motor control, audio signal generation, and other analog control scenarios.



NOTE:When we increase the frequency of PWM signal the capacitor gets lesser time to charge and discharge, so the peak value decreases.

The peak Value depends on the duty cycle of the PWM Signal:
Higher the duty cycle, higher the peak value.