# Principios de mecatrónica
# Clase 5–10

SDI-11561-004, SDI-11561-002

**IN vs LDS**

Como mencionamos anteriormente, se puede usar instrucción LDS para copiar el contenido de una localidad de memoria al GPR. Esto significa que nosotros podemos cargar un registro de I/O dentro del GPR, usando la instrucción LDS. Así, cual es la ventaja de usar instrucción IN para leer el contenido de los registros I/O sobre el uso de la instrucción LDS?

La instrucción IN tiene las siguientes ventajas:

- El CPU ejecuta la instrucción IN más rápido que la instrucción LDS, ya que IN es una instrucción de 2 bytes, mientras que LDS lo es de cuatro. Esto significa que la instrucción IN ocupa menos memoria.
- Cuando usamos la instrucción IN, se puede usar el nombre de los registros I/O en lugar de su dirección.
- La instrucción IN esta disponible en todos los AVR's, mientras que LDS no se ha implementado en algunos AVR's.

Sin embargo, es importante notar que usando la instrucción IN se puede acceder solamente a la memoria standard I/O, mientras que usando la instrucción LDS se puede acceder a todas partes de los datos de memoria.

*Ejemplo 1.-* Realizar un programa que reste 5 del contenido de la localización $0x300$ y lo almacene en la localidad $0x320$

*Ejemplo 2.-* Realizar un programa que envíe inicialmente al DDRB $0x55$ y después su complemento.

**Incondicionante útil en la memoria(JUMP)**

Básicamente, cada programa ensamblador necesita uno de los diferentes tipos de saltos. Los saltos son en un sólo sentido: el programa no puede regresar del código al que saltó, a menos que se conozca la ubicación del reotorno.

**JMP:** Salta a una dirección dentro de la memoria de programa de xM (palabras). Esta instrucción no está disponible en todos los dispositivos.

*Ejemplo 1.-*

**.org 0x00**
**jmp main**
**.....** //espacio para subrutinas y códigos
**.org 0x900** // configurar contador de programa para dirección 0x900
**main:** // está es la aplicación de la rutina principal
**rjmp main** //y es sólo un bucle sin fin.

En este ejemplo, se coloca una instrucción jmp en el vector de reinicio (0x00), y se utiliza para saltar a la rutina principal de la aplicación, que está etiquetada como main. La cual se ha colocado en la dirección 0x900 utilizando la **directiva .org**.

.org no es una instrucción de salto, sino una directiva de ensamblador. Le dice al ensamblador que coloque las siguientes expresiones en una dirección específica.

Es importante mencionar que JMP es una instrucción (brinco incondicional) de 4 bytes y que puede ir a **cualquier localidad de memoria** en espacios de memoria de (4M), $2^{22}$ palabras, lo que permitedar brincos entre las localidades 000000 a $3FFFFF$.

*Ejemplo 2.-* El programa obtiene datos del PINB y lo envia al registro

I/O del PORT C continuamente:

```
AGAIN: IN    R16,    PINB   ; bring data from PortB into R16
       OUT   PORTC, R16    ; send it to Port C
       JMP AGAIN    ; keep doing it forever
```

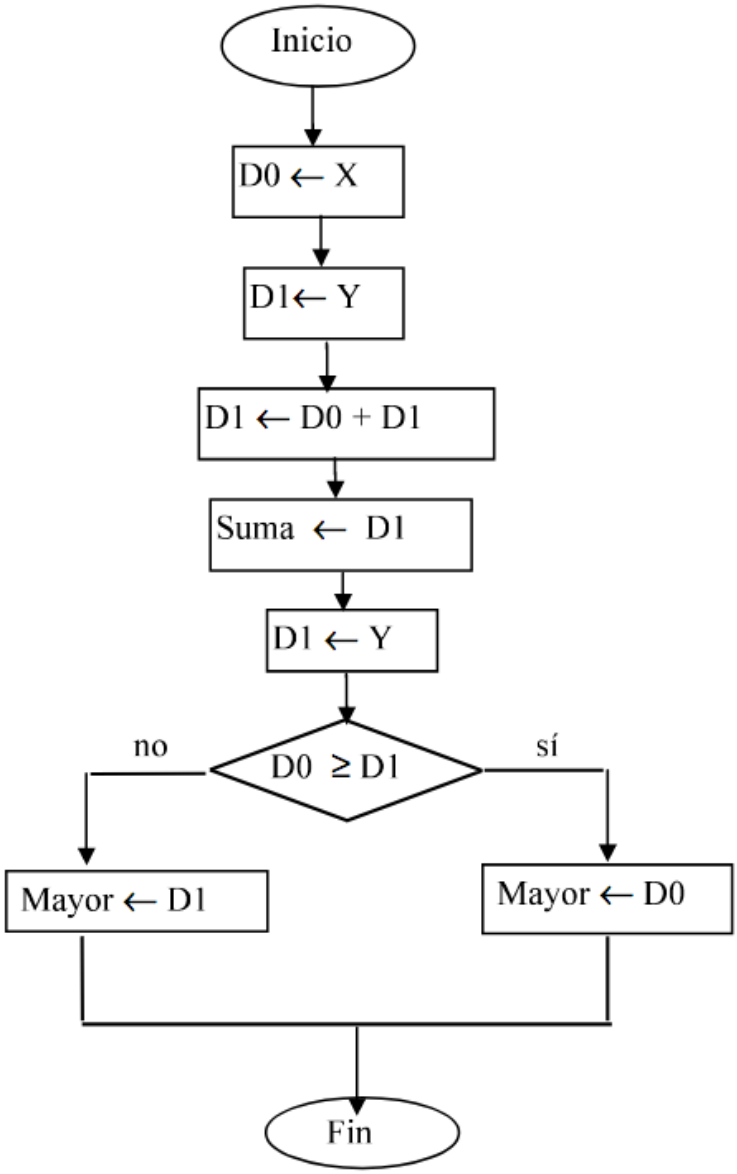
*Ejemplo 3.-* Realizar un programa que muestre como se alternan los bits de un número (0 a 1 y viceversa) en el registro de I/O PORT B de forma continua.

```
       LDI   R20,   0x55
       OUT   PORTB,   R20
L1:    COM   R20
       OUT   PORTB,   R20
       JMP L1
```

*Ejemplo 4.-* Desarrollar un programa que realice las tareas presentadas en la siguiente figura.

**CP**: *Esta instrucción realiza una comparación entre dos registros $R_d$ y $R_r$. Ninguno de los registros es cambiado. Todas las instrucciones condicionales se pueden utilizar después de esta instrucción.*
**BRSH** :*Verifica la bandera de acarreo (C) y brinca relativamente (PC+k+1) si C=0. Si la instrucción se ejecuta inmediatamente después de la ejecución de cualquiera de las instrucciones CP, CPI, SUB o SUBI, se producirá el brinco si y solo si el número binario (sin signo) representado en $R_d$ es mayor o igual que el número binario sin signo representado en $R_r$.*

**BRGE**: *Brinco condicional. Verifica la bandera signo (S) y se brinca relativamente (Pc+K+1) en relación con el PC si S es cero. Si la instrucción se ejecuta inmediatamente después de cualquiera de las instrucciones CP, CPI, SUB o SUBI, el brinco ocurrirá si y solo si el número binario(signado) representado en $R_d$ es mayor o igual que el número binario representado en Rr.*


Inicio
D0 ← X
D1← Y
D1 ← D0 + D1
Suma ← D1
D1 ← Y
no
D0 ≥ D1
sí
Mayor ← D1
Mayor ← D0
Fin

Hay dos razones para proporcionar una instrucción de salto con capacidades algo limitadas: los AVR normales no tienen tanta memoria, y la mayoría de los destinos de saltos se encuentran dentro de una región cercana a la dirección donde se almacena el salto. El salto relativo aprovecha esta situación.

**JUMP relativo (RJMP)**

Es una instrucción de 2 bytes, donde las direcciones relativas tienen un rango entre 000 y $FFF$, la cual es dividida para brincos positivos y negativos, que es dentro de -2048 a 2048 palabras de memoria. De esta forma, al ser de 2 bytes es preferida sobre JMP ya que toma menos espacio de RAM.

En el fragmento de código que acompaña a la explicación de jmp (ver arriba), rjmp se usa para construir un bucle sin fin:

**.....** //espacio para subrutinas y códigos
**.org 0x900** // configurar contador de programa para dirección 0x900
**main:** // está es la aplicación de la rutina principal
**rjmp main** //y es sólo un bucle sin fin.

La instrucción que sigue al rjmp sería en la dirección 0x901. El destino del salto es 0x900, por lo que la longitud relativa del salto es 0x900-0x901 = -1. De nuevo: repetir la misma instrucción una y otra vez se almacena como un salto hacia atrás. En consecuencia, un salto relativo de cero palabras.

Es similar a un **NOP**, pero necesita dos ciclos de CPU para la ejecución. Esté es un ciclo menos que los necesarios en jmp.

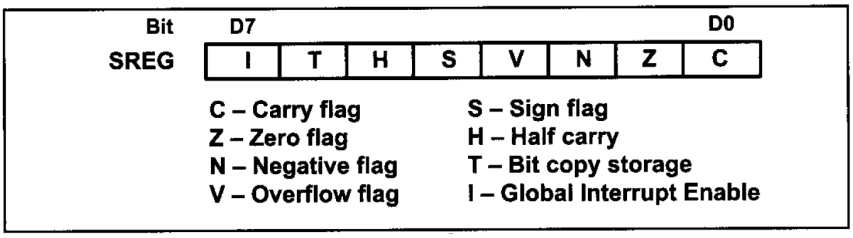
*Ejemplo 1.-*

```
cpi R16,0x42        ;Compara r16 to 0x42
brne error          ;Branch if r16 no es 0x42
rjmp ok             ;Unconditional branch
```

```
error: ADD R16 R17   ; Suma R17 to R16
INC R16              ;Incrementa R16
ok: nop              ;Destino de rjmp
```

## 1. Registro de status en el AVR

Al igual que otros microprocesadores, el AVR tiene un registro de banderas para indicar condiciones aritméticas tal como el bit de acarreo. El registro de banderas en el AVR es llamado el *status register(SReg)*. En esta sesión, nosotros discutiremos varios bits de estos registros y proporcionaremos algunos ejemplos de como son alterados.
El registro de estatus es un registro de 8 bits. Es además referido como el registro de banderas. Los bits C, Z, N, V, S y H son llamadas banderas de condiciójales, lo cual significa que ellas indican algunas condiciones que resultan después de que una instrucción ha sido ejecutada. Cada una de estas banderas condicionales pueden ser usadas para realizar una condición de branch (o jump).




- *Carry (c):* Esta bandera es afectada después de una suma o resta.
- *Zero (z):* La bandera de cero refleja el resultado de una operación aritmética u operación lógica/ Si el resultado es cero, entonces $Z = 1$, de lo contrario $Z = 0$.
- *Negative (N)* Representación binaria de números con signo usan D7 como bit de signo. La bandera de negativo refleja el resultado de una operación aritmética. Si elbit D7 de el resultado es cero, entonces N=0 y el resultado es positivo. Si el bit D7 es uno, entonces N=1 y el

resultado es negativo. El negativo y el bit de la bandera V son usados para operaciones aritméticas de números con signo.

- *Overflow (O)* Esta bandera es usada si el resultado de una operación de números con signo es muy grande, causando un bit de mayor orden que sobre pase el bit del signo.
- *Sign* Esta bandera es el resultado de una Or-exclusiva de las banderas N y V.
- *Half carry* Si hat un carry de D3 a D4 durante una operación de suma o resta, este bit es activado. Este bit es usado por instrucciones que realizan aritmetica BCD.

*Ejemplo.-* Realizar la suma de 0x38 y 0x2F y mostrar el status de las banderas C, H y Z después de la ejecución de la suma.

*Ejemplo.-* Realizar la suma de 0x9C y 0x64 y mostrar el status de las banderas C, H y Z.

## 2. Bits de banderas y decisiones

Hay instrucciones que harán una condición de jump (o branch) basado sobre el status de los bits de las banderas. Los nombres de las instrucciones son los siguientes.

### Instrucciones de Branch y lazos

En esta sección discutiremos como se puede realizar una acción de lazo en el AVR y la instrucción branch (jump), ambas condicional e incondicional.

| Instruction | Action |
|---|---|
| BRLO | Branch if C = 1 |
| BRSH | Branch if C = 0 |
| BREQ | Branch if Z = 1 |
| BRNE | Branch if Z = 0 |
| BRMI | Branch if N = 1 |
| BRPL | Branch if N = 0 |
| BRVS | Branch if V = 1 |
| BRVC | Branch if V = 0 |

**Lazos en AVR**

Repitiendo una secuencia de instrucciones o varias veces la misma operación a ciertos números, es llamado lazo(loops). Los lazos o bucles son las técnicas mas usados en programación. En el AVR, hay varias maneras de repetir una operación varias veces. La forma más facil es repetir la operación una y otra vez.

*Ejemplo*

```
LDI R16,0         ;R16 = 0
LDI R17,3         ;R17 = 3
ADD R16,R17       ;add value 3 to R16 (R16 = 0x03)
ADD R16,R17       ;add value 3 to R16 (R16 = 0x06)
ADD R16,R17       ;add value 3 to R16 (R16 = 0x09)
ADD R16,R17       ;add value 3 to R16 (R16 = 0x0C)
ADD R16,R17       ;add value 3 to R16 (R16 = 0x0F)
ADD R16,R17       ;add value 3 to R16 (R16 = 0x12)
```

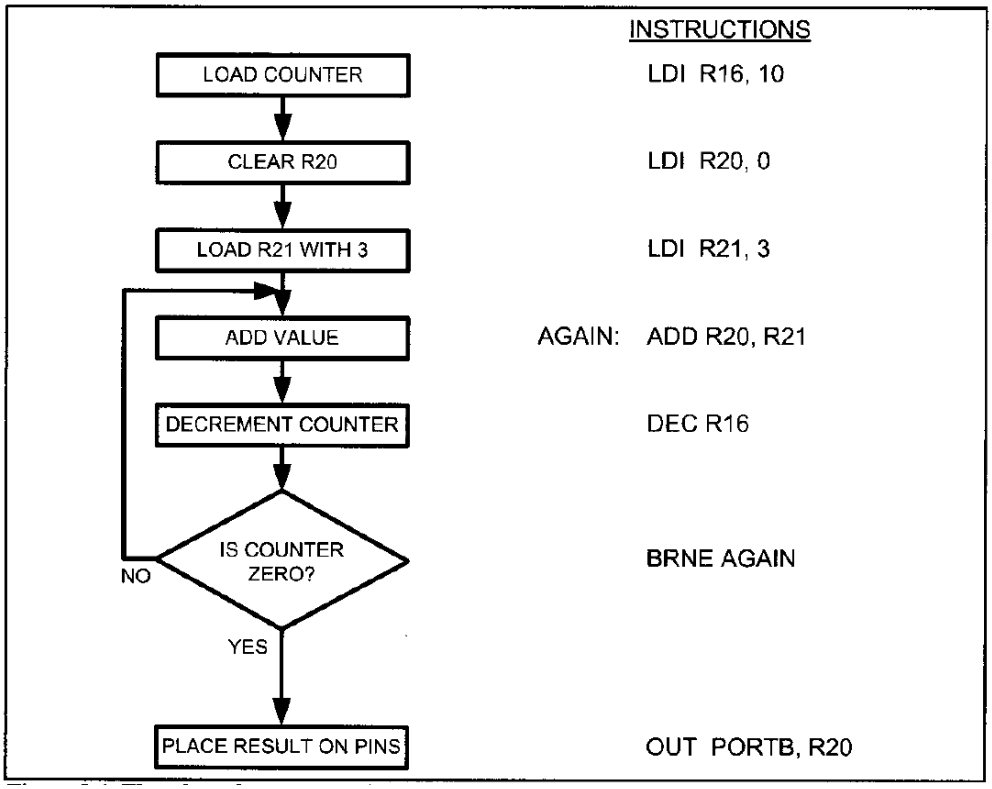
En el programa, se ha sumado 5 veces el número 3 en el registro 16. Un problema con el ejemplo anterior es que se necesita mucho espacio de código, esto implica que si se desea incrementar el numero de repeticiones resultaría complicado y tedioso. Una mejor manera es usar un lazo.

**Instrucción BRNE para lazos**

La instrucción (branch if not equal) BRNE usa la bandera zero en registro de estatus. La instrucción es usada como sigue:

```
BACK: .........   ;start of the loop
      .........   ;body of the loop
      .........   ;body of the loop
      DEC Rn      ;decrement Rn, Z = 1 if Rn = 0
      BRNE BACK   ;branch to BACK if Z = 0
```

Ejemplo.- Escribir un programa para (a) limpiar el registro 20, entonces (b) sumar diez veces el número 3 en el R20, y (c) enviar la suma al puerto B, usando la bandera zero y BRNE



```
;this program adds value 3 to the R20 ten times
 .INCLUDE "M32DEF.INC"
      LDI   R16, 10      ;R16 = 10 (decimal) for counter
      LDI   R20, 0       ;R20 = 0
      LDI   R21, 3       ;R21 = 3
AGAIN:ADD   R20, R21     ;add 03 to R20 (R20 = sum)
      DEC   R16          ;decrement R16 (counter)
      BRNE  AGAIN        ;repeat until COUNT = 0
      OUT   PORTB,R20    ;send sum to PORTB
```

**BREQ(branch if equal, branch if Z=1)**

En esta instrucción, la bandera Z es revisada. Si es activa, el CPU brinca a la dirección objetivo. Por ejemplo,

```
OVER:   IN   R20,   PINB
        TST   R20
        BREQ   OVER
```

En este programa, si PINB es cero, el CPU brincará a la etiqueta OVER. Se quedará en el lazo hasta que PIB tenga otro valor que no sea cero. Es importante notar que la instrucción **TST**(Tests if a register is zero or negative) es usada para examinar un registro estableciendo las banderas de acuerdo a su contenido sin tener que realizar una instrucción aritmética tal como un decremento.

*Ejemplo: Cual es la función del siguiente programa?*

```
        .EQU  MYLOC=0x200
        LDS   R30, MYLOC
        TST   R30              ;set the flag
                               ;(Z=1 if R30 has zero value)
        BRNE  NEXT             ;branch if R30 is not zero (Z=0)
        LDI   R30, 0x55        ;put 0x55 if R30 has zero value
        STS   MYLOC,R30        ;and store a copy to loc $200
NEXT: ...
```

El programa determina si la localización de RAM $0x200$ contiene el valor cero, entonces si es cierto, pone el valor 0x55 sobre él.

**BRSH (branch if C=0)**

En esta intrucción la bandera de carry en el Status register es usado para tomar la decisión de brincar.

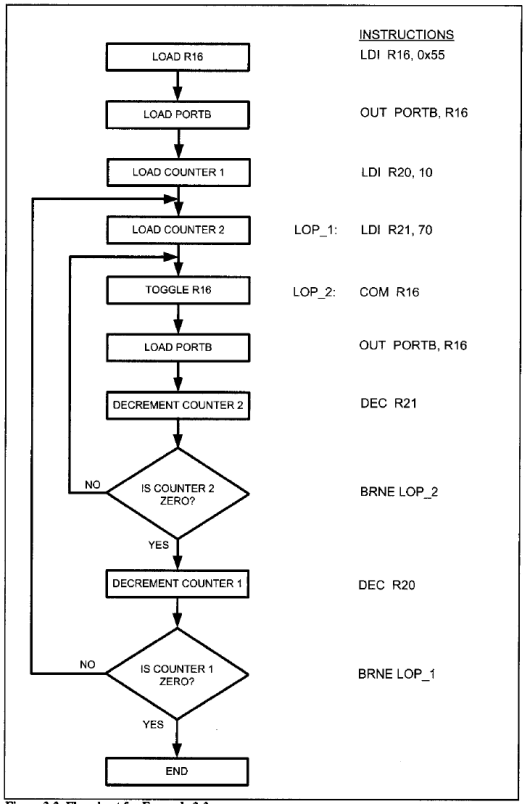
```
.INCLUDE "M32DEF.INC"
.ORG 0
      LDI   R21, 0      ;clear high byte (R21 = 0)
      LDI   R20, 0      ;clear low byte (R20 = 0)
      LDI   R16, 0x79
      ADD   R20, R16    ;R20 = 0 + 0x79 = 0x79, C = 0
      BRSH  N_1         ;if C = 0, add next number
      INC   R21         ;C = 1, increment (now high byte = 0)
N_1:  LDI   R16, 0xF5
      ADD   R20, R16    ;R20 = 0x79 + 0xF5 = 0x6E and C = 1
      BRSH  N_2         ;branch if C = 0
      INC   R21         ;C = 1, increment (now high byte = 1)
N_2:  LDI   R16, 0xE2
      ADD   R20, R16    ;R20 = 0x6E + 0xE2 = 0x50 and C = 1
      BRSH  OVER        ;branch if C = 0
      INC   R21         ;C = 1, increment (now high byte = 2)
OVER:                   ;now low byte = 0x50, and high byte = 02
```

El programa suma los valores 0x79, 0xF5 y 0xE2 y pone la suma en R20(low byte).

**Lazo dentro del lazo**

Nosotros sabemos que el contenido máximo de un registro es 255; la pregunta es: Qué pasa si nosotros queremos repetir una acción más de 255 veces?, para hacer esto es necesario un lazo dentro de otro lazo, lo cual es llamado lazo anidado, de esta forma nosotros cumplimos el objetivo usando dos o mas registros.

```
.INCLUDE "M32DEF.INC"
.ORG 0
      LDI   R16, 0x55     ;R16 = 0x55
      OUT   PORTB, R16    ;PORTB = 0x55
      LDI   R20, 10       ;load 10 into R20 (outer loop count)
LOP_1:LDI   R21, 70       ;load 70 into R21 (inner loop count)
LOP_2:COM   R16           ;complement R16
      OUT   PORTB, R16    ;load PORTB SFR with the complemented value
      DEC   R21           ;dec R21 (inner loop)
      BRNE  LOP_2         ;repeat it 70 times
      DEC   R20           ;dec R20 (outer loop)
      BRNE  LOP_1         ;repeat it 10 times
```



Ejemplo: Qué hace el siguiente código de programa?

```
        LDI   R16, 0x55
        OUT   PORTB, R16
        LDI   R23, 10
LOP_3:LDI   R22, 100
LOP_2:LDI    R21, 100
LOP_1:COM   R16
        DEC   R21
        BRNE  LOP_1
        DEC   R22
        BRNE  LOP_2
        DEC   R23
        BRNE  LOP_3
```

**Instrucciones de comparación sin el registro de banderas.**

**(Compare Skip if Equal ) CPSE:** *Esta instrucción realiza una comparación entre dos registros $R_d$ y $R_r$, y omite la siguiente instrucción si $R_d = R_r$.*

## Representación de formato de datos

Existen cuatro formas de representar un byte de datos en el AVR. Los números pueden ser hexadecimal, binario, decimal o ASCII. Los siguientes son ejemplos de como trabaja cada uno.

### Números hexadecimales

Como habíamos mencionado previamente, hay dos formas de mostrar un número hexadecimal

- Poner $0x$ enfrente del número: LDI  R16,  0x99
- Poner $ enfrente del número: LDI  R22, $99

*Ejemplo*
LDI  R28,  $75

```
SUBI    R28,    0x11
SUBI    R28,    0x20
ANDI    R28,    0xF
```

**Números binarios**

Existe una forma de representar números binarios en el ensamblador de AVR. Es como sigue:

```
LDI    R16,    0b10011001.
```

La letra mayúscula también funciona:

```
LDI    R23, 0b00100101
ORI    R23, 0B00010001
```

**Números decimales**

Para indicar números decimales en un ensamblador de AVR, se usa simplemente el numero sin poner nada antes o después de él.

```
LDI    R17,    12
SUBI    R17,    2
```

**ASCII**

Para representar datos ASCII, las cantidades se usan como sigue

```
LDI    R23, '2'  ; R23=00110010:= 32 en hexadecimal
```

Otro ejemplo:

```
LDI    R20, '9'   ;R20=0x39
SUBI    R20, '1'   ; R20= 0x39-0x31=0x8
```

## 3. Directivas en ensamblador

Mientras que las instrucciones llaman al CPU para saber que hacer, las directivas (también llamadas pseudo–instrucciones) dan direcciones al ensamblador. Por ejemplo LDI y ADD son instrucciones comandadas por el CPU, pero .EQU, .DEVIDE, .ORG son directivas para el ensamblador. Una ventaja del lenguaje ensamblador es que las directivas ayudan a desarrollar nuestro programa más fácil y hacen el programa más legible. Esto debido a que indican aspectos como la ubicación del código o definiciones. Además de que no generan código maquina.

**.EQU(equate):** Este es usado para definir valores constantes o fijar direcciones. La directiva .EQU no reserva un almacenamiento para un elemento de datos, pero asocia a números constantes con un dato o un nivel de dirección así que cuando el nivel aparece en el programa, su constante será sustituida por el nivel. La siguiente instruccione usa .EQU para contador(COUNT) constante y entonces la constante es usada para cargar el registro R21:

.EQU  COUNT = 0x25

... ...

LDI  R21, COUNT

Mencionamos anteriormente que se puede usar el nombre del registros de I/O en lugar de sus direcciones (e,g., se puede escribir "OUT PORTA, R20" en lugar de " OUT 0x1B, R20"). Esto se realiza con la ayuda de la directiva .EQU. En "include files", tal como M32DEF.INC los nombres de los registros I/O son asociados con su dirección usando la directiva .EQU.

Por ejemplo, en M32DEF.INC, la siguiente pseudo instrucción existe, a la cual se ha asociado 0x1B (dirección de PORTB) con el PORTB

.EQU  PORTB=0x1B

.EQU es ampliamente usado para asignar direcciones del SFR

```
.EQU COUNTER = 0x00     ;counter value 00
.EQU PORTB = 0x18       ;SFR Port B address
LDI  R16, COUNTER       ;R16 = 0x00
OUT  PORTB, R16         ;Port B (loc 0x18) now has 00 too
```

Otro uso común de .EQU es para asignar la dirección de la SRAM interna.

```
.EQU  SUM = 0x120       ;assign RAM loc to SUM
LDI   R20, 5            ;load R20 with 5
LDI   R21, 2            ;load R21 with 2
ADD   R20, R21          ;R20 = R20 + R21
ADD   R20, R21          ;R20 = R20 + R21
STS   SUM, R20          ;store the result in loc 0x120
```

**.SET** Esta directiva es usada para definir un valor constante o una dirección fija. En este sentido, .SET y .EQU son idénticas. La única diferencia es que el valor asignado por la directiva .SET puede ser reasignado más tarde.

**.ORG** La directiva .ORG es usado para indicar el inicio de la dirección y puede ser usado para código y datos.

**.INCLUDE** La directiva .INCLUDE llama al ensamblador del AVR para agregar el contenido de un archivo a nuestro programa. En la siguiente tabla, se puede ver los archivos que se puede, incluir dependiendo del tipo de AVR.

| ATMEGA | | ATTINY | | Special Purpose | |
|---|---|---|---|---|---|
| ATmega8 | m8def.inc | ATtiny11 | tn11def.inc | AT90CAN32 | can32def.inc |
| ATmega16 | m16def.inc | ATtiny12 | tn12def.inc | AT90CAN64 | can64def.inc |
| ATmega32 | m32def.inc | ATtiny22 | tn22def.inc | AT90PWM2 | pwm2def.inc |
| ATmega64 | m64def.inc | ATtiny44 | tn44def.inc | AT90PWM3 | pwm3def.inc |
| ATmega128 | m128def.inc | ATtiny85 | tn85def.inc | AT90USB646 | usb646def.inc |
| ATmega256 | m256def.inc | | | | |
| ATmega2560 | m2560def.inc | | | | |

Por ejemplo, cuando se quiere usar el ATmega32, debemos escribir la siguiente instrucción al inicio del programa

.INCLUDE “M32DEF.INC”

*Ejemplo 1.-*

```
.include “m256def.inc”
.org 0x00
ADD R16, R17          ; Add R17 to R16
MOV R17, R18          ; Copy R18 to R17
```

De esta forma .include indica al ensamblador que ensamble el código contenido en el archivo especificado como parte del programa.

Además, .org 0x00 indica al ensamblador dónde se encuentra dentro de la memoria del microcontrolador para comenzar a almacenar el código que sigue. Es decir, comience a almacenar el código de máquina desde la dirección 0x00 en la memoria del programa.

**.DEVICE**.- Define cual microcontrolador será usado, el cual define la aplicabilidad del conjunto de instrucciones.

### Estructura del lenguaje ensamblador

Lenguaje ensamblador se refiere a un lenguaje de bajo nivel debido a que trata directamente con la estructura interna del CPU. Para programar en lenguaje ensamblador, el programador debe de conocer todos los registros del CPU, sus direcciones, su tamaño, entre otros detalles.

Ahora, uno puede usar muchos programas de lenguaje, tales como BASIC, Pascal, C, C++, Java y muchos más. Estos lenguajes son llamados de alto nivel debido a que el programador no necesita conocer los detalles internos del CPU. Mientras que ensamblador es usado para traducir un programa dentro de un código maquina. Mientras que lenguajes de alto nivel son traducidos hacia lenguaje maquina por medio de un compilador.

De esta forma, un programa de lenguaje ensamblador consiste de una serie de líneas u oraciones de instrucciones de ensamblador. Una instrucción de ensamblador consiste cuatro campos: etiquetas, mnemonicos, opcional-

mente seguido de uno o dos operandos. Los operandos son los datos a ser manipulados y comentarios, el cual es opcional.

$$[etiqueta:]\ mnemonico\ [operandos]\ [;comentarios]$$

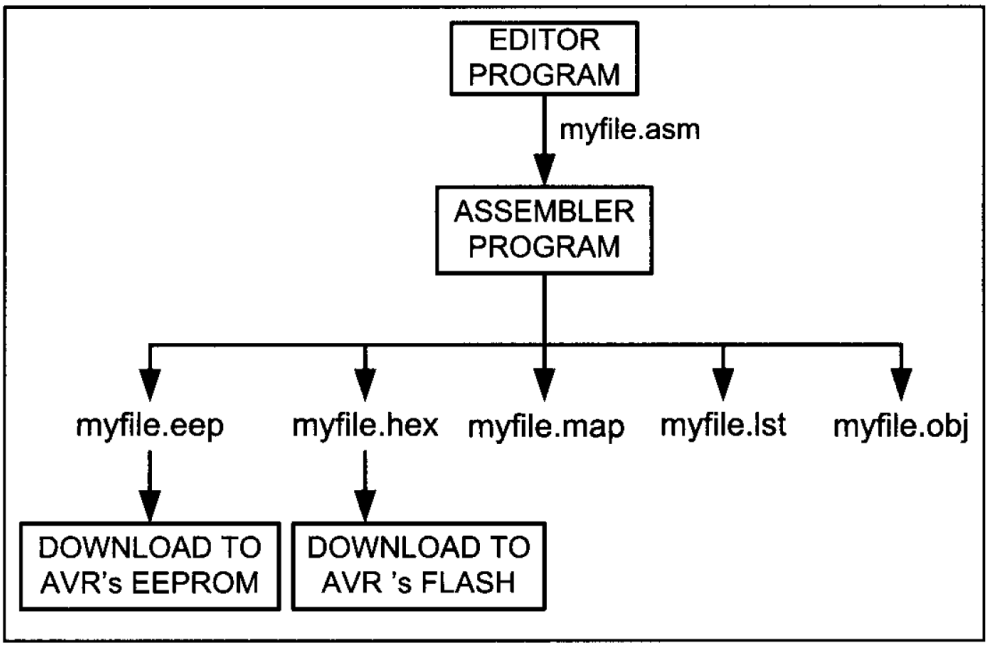
Un simple programa con todos los elementos de una instrucción se muestra a continuación:

```
      ;AVR Assembly Language Program To Add Some Data.
      ;store SUM in SRAM location 0x300.

      .EQU  SUM   = 0x300     ;SRAM loc $300 for SUM

      .ORG 00                 ;start at address 0
      LDI R16, 0x25           ;R16 = 0x25
      LDI R17, $34            ;R17 = 0x34
      LDI R18, 0b00110001     ;R18 = 0x31
      ADD R16, R17            ;add R17 to R16
      ADD R16, R18            ;add R18 to R16
      LDI R17, 11             ;R17 = 0x0B
      ADD R16, R17            ;add R17 to R16
      STS SUM, R16            ;save the SUM in loc $300
HERE: JMP HERE                ;stay here forever
```

**Ensamblando un programa en AVR**

Para ensamblar un programa del microcontrolador AVR utiliza un editor de texto para escribir un programa como el del ejemplo anterior. En el caso de los microcontroladores AVR, se utiliza el **AVRStudio–x** el cual tiene un editor de text, simulador y ensamblador. Es un excelente desarrollador de software que soporta toda la familia de los AVR's. Para ensambladores, los nombres de los archivos siguen la convención usual de MS–DOS(MicroSoft Disk Operating System), pero el archivo fuente en los AVR's tienen la extensión "asm".

El archivo fuente "asm" contiene el código de programa creado en el editor y es alimentado al ensamblador AVR. El ensamblador produce un archivo objeto(.obj), un archivo hex(.hex), un archivo eeprom(.eep), un




archivo list(.lst) y un archivo map(.map).

Después, el archivo hex esta listo para ser quemado en el programa AVR ROM y es descargado en el AVR trainer. El ensamblador convierte el archivo asm en lenguaje maquina y proporciona el archivo objeto (obj). Este archivo es usado como entrada para un simulador o emulador.

**Archivos "lst" y "map"**

Los archivos map muestran las etiquetas definidas en el programa junto con sus valores.

```
AVRASM ver. 2.1.2  F:\AVR\Sample\Sample.asm Sun Apr 06 23:39:32 2008


EQU   SUM           00000300
CSEG  HERE          00000009
```

El archivo lst, el cual es opcional, es útil para el programador. Este archivo muestra el código binario y el código fuente, además muestra cuales instrucciones se han usado en el código fuente y la cantidad de memoria que el programa ha usado.

```
AVRASM ver. 2.1.2  F:\AVR\Sample\Sample.asm Tue Mar 11 11:28:34 2008

               ;store SUM in SRAM location 0x300.
                  .DEVICE ATMega32
                  .EQU  SUM   = 0x300     ;SRAM loc $300 for SUM

                  .ORG 00                 ;start at address 0
000000 e205       LDI R16, 0x25           ;R16 = 0x25
000001 e314       LDI R17, $34            ;R17 = 0x34
000002 e321       LDI R18, 0b00110001     ;R18 = 0x31
000003 0f01       ADD R16, R17            ;add R17 to R16
000004 0f02       ADD R16, R18            ;add R18 to R16
000005 e01b       LDI R17, 11             ;R17 = 0x0B
000006 0f01       ADD R16, R17            ;add R17 to R16
000007 9300 0300  STS SUM, R16            ;save the SUM in loc $300
000009 940c 0009 HERE: JMP HERE           ;stay here forever

RESOURCE USE INFORMATION
------------------------
...
Memory use summary [bytes]:
Segment    Begin     End       Code    Data    Used    Size    Use%
---------------------------------------------------------------------
[.cseg]  0x000000 0x000016      22       0      22 unknown       -
[.dseg]  0x000060 0x000060       0       0       0 unknown       -
[.eseg]  0x000000 0x000000       0       0       0 unknown       -

Assembly complete, 0 errors, 0 warnings
```

### 3.1. Contador de programa en el AVR

El registro más importante en el microcontrolador AVR es el program counter(PC). El PC es usado por el CPU para apuntar a la dirección de la siguiente instrucción a ser ejecutada. Como el CPU obtiene el código de operación de la ROM del programa, el PC es incrementado automáticamente a apuntar a la siguiente instrucción. Cuanto más amplio sea el contador del programa, más ubicaciones de memoria podrá acceder una CPU. Estó significa que un PC de 14 bits puede acceder a un máximo de 16K localizaciones de memoria $2^{14} = 16K$.

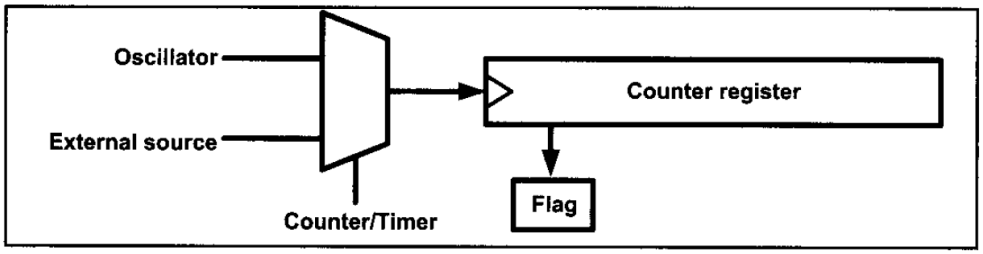
En los microconotradores AVR, cada localización de memoria Flash es de 2 bytes. Por ejemplo, en el ATMega 32, cuyo Flash es de 32K bytes, the Flash es organizada, y su PC es de 14 bits de amplitud ($2^{14} = 16$K localidades de memoria). El ATmega64 tiene un contador de 15 bits, así la Flash tiene 32K de direcciones ($2^{15} = 32$) con cada localización conteniendo dos bytes (32K$\times$ 2 bytes=64K bytes).

| | On-chip Code ROM (Bytes) | Code Address Range (Hex) | ROM Organization |
|---|---|---|---|
| ATtiny25 | 2K | 00000–003FF | 1K × 2 bytes |
| ATmega8 | 8K | 00000–00FFF | 4K × 2 bytes |
| ATmega32 | 32K | 00000–03FFF | 16K × 2 bytes |
| ATmega64 | 64K | 00000–07FFF | 32K × 2 bytes |
| ATmega128 | 128K | 00000–0FFFF | 64K × 2 bytes |
| ATmega256 | 256K | 00000–1FFFF | 128K × 2 bytes |

Figura 1: Espacio de direcciones

## Programación del timer del AVR

Muchas aplicaciones necesitan contar un evento o generar tiempos de retardo. Así, hay registros de contadores en un microcontrolador para estos propósitos. **Cuando queremos contar un evento, podemos conectar la fuente del evento externo al pin del reloj del registro de contador**. De esta forma, cuando un evento ocurre externamente, el contador es incrementado; así , el contenido del contador representa el numero de veces que el evento ha ocurrido. **Cuando queremos generar tiempos de retardo, se conecta el oscilador al pin del reloj del contador.** Entonces, cuando el oscilador marque, el contenido del contador será incrementado. Como resultado, el contenido del registro del contador representará cuantas marcas han ocurrido desde que se ha limpiado el contador. Ya que la velocidad del oscilador en un microcontrolador es conocido, se puede calcular el periodo de la marca, y del contenido del registro del contador nosotros conoceremos cuantos tiempos han transcurrido.




De esta manera para generar un retardo, se borra el contador en el

tiempo inicial y se espera hasta que el contador alcance un cierto numero.

Por ejemplo, considere un microcontrolador con un oscilador de una frecuencia de 1MHz; en el microcontrolador, el contenido del registro del contador incrementa una vez por microsegundo. Entonces, si queremos un retardo de 100 microsegundos, debemos limpiar el contador y esperar hasta que sea igual a 100.

En el microcontrolador, hay una bandera para cada uno de los contadores. La bandera se establece cuando el contador se desborda, y es borrada por software. El segundo método para generar un retardo es cargar el registro del contador y esperar hasta que el contador se desborde y la bandera se establezca.

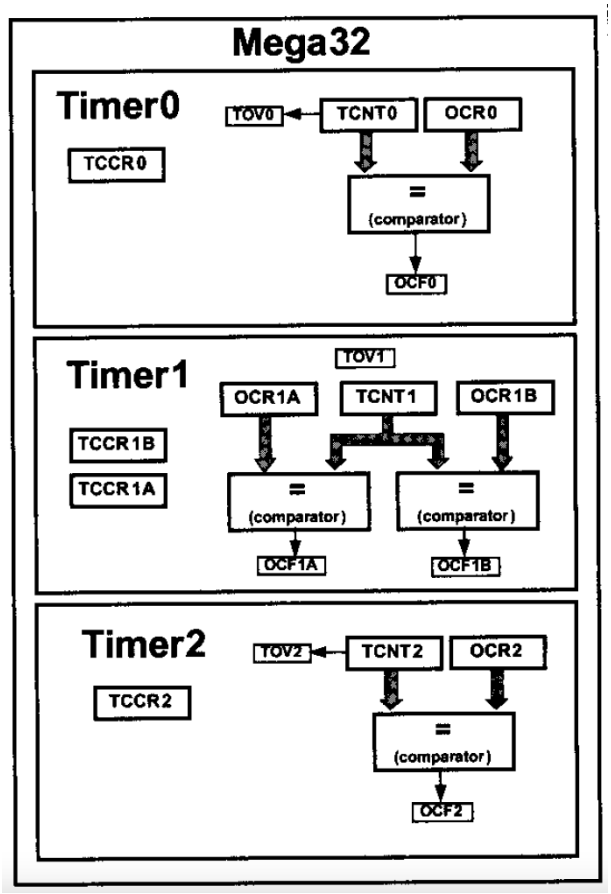
Por ejemplo, en un microcontrolador, con una frecuencia de 1MHz, con un registro contador de 8 bits, si queremos un retardo de 3 microsegundos, necesitamos cargar el contador con $ FD y esperar hasta que la bandera se establezca después de 3 ticks. Después del primer tick, el contenido del registro se incrementara a $FE, después del segundo tick, llegará hacer $FF, y después del tercer tick, el contenido del registro llegará hacer(overflow) $00 y la bandera se establecerá.

El AVR tiene de 1 a 6 timers dependiendo del tipo de familia. Ellos son referidos como 0, 1..5. Pueden ser usados como timers para generar retardo o como contadores para contar eventos sucediendo fuera del microcontrolador.

En el AVR algunos de los timers/contadores son de 8 bits y algunos son de 16 bits. El ATmega32, tiene 3 timers: Timer0, Timer1 y Timer2. Timer0 y Timer2 son de 8 bits, mientras que el timer T1 es de 16 bits. En específico el ATMEGA 2560 utilizado en el laboratorio de principios de mecatrónica tiene 6 timers, 2 de 8 bits y 4 de 16 bits.

**Programación de Timers**

Cada timer necesita un pulso de reloj para marcar. La fuente del reloj puede ser interna o externa. Si se usa la fuente de reloj interna, entonces la frecuencia del oscilador es alimentada al timer. Por lo tanto, es usado para generar retardos y consecuentemente es llamado timer. Si nosotros elegimos la opción de un reloj externo, se necesitará alimentar pulsos a través de uno de los pines del AVR. Esto es llamado contador.




Para cada timer, hay un registro TCNTn (timer/contador). Lo cual significa que en el ATmega32 tenemos TCNT0, TCNT1 y TCNT2.

- El registro TCNTn contiene ceros. El registro (contador) cuenta con cada pulso.
- Se puede acceder al contenido de los timers/ contadores usando TCNTn.
- Se pueden cargar valores sobre TCNTn o leer.

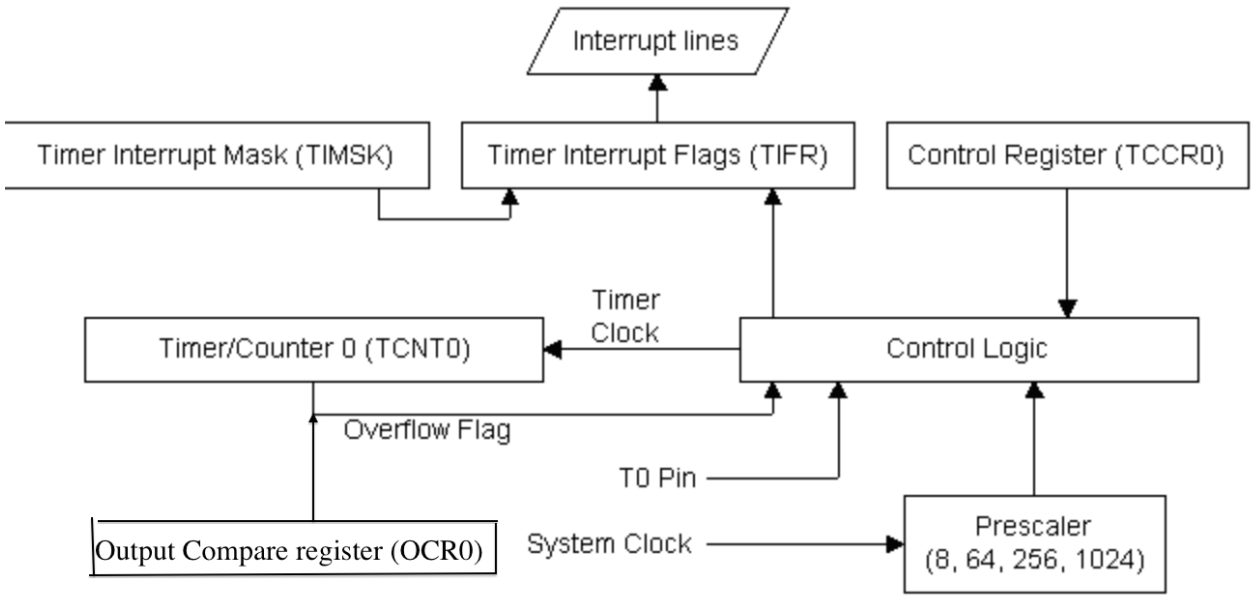
- Cada timer tiene una bandera TOVn (Timer overflow).
- Cada timer tiene además un registro (TCCRn) para definir su modo de operación. Por ejemplo, se puede especificar Timer0 para trabajar como un timer o un contador cargando su valor en TCCR0.
- Cada timer tiene también un registro OCRn (output compare register). El contenido de OCRn es comparado con el contenido del TCNTn. Cuando ellos son iguales la bandera OCFn (bandera de salida de comparación) será establecida.

Los registros de timer están localizados en los registros de memoria I/O. Por lo tanto, se puede leer o escribir en los registros usando las instrucciones IN y OUT, como en los otros registros I/O.

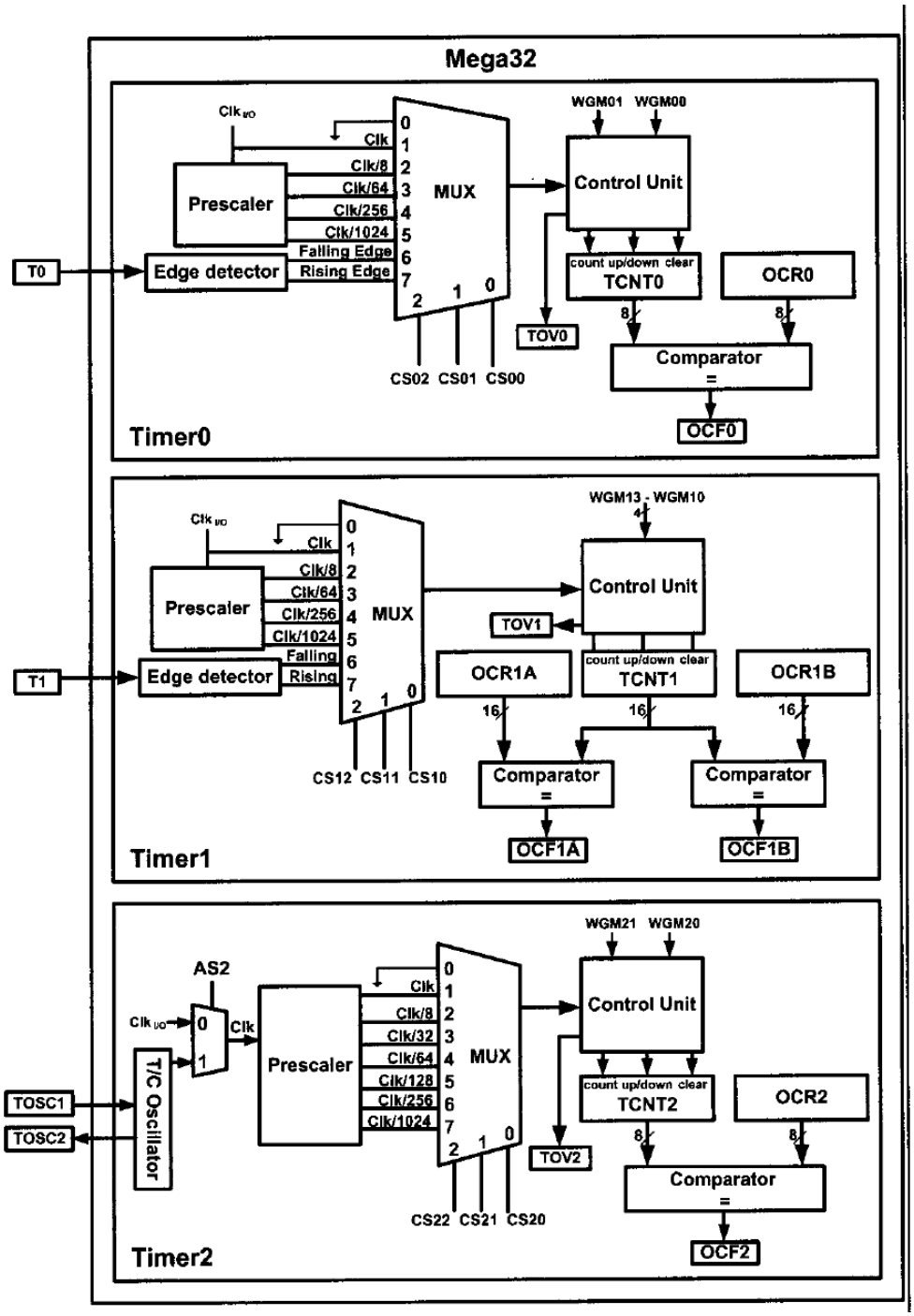
Por ejemplo, la siguiente instrucción carga TCNT0 con 25:

```
LDI   R20, 25
OUT   TCNT0, R20
```

Un bosquejo de la estructura de los timers puede apreciarse en la siguiente figura.

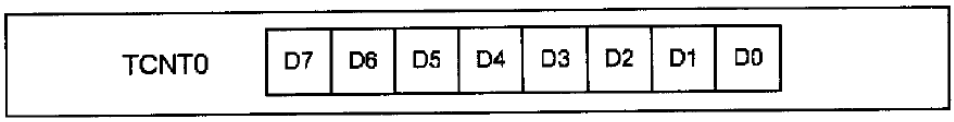


La estructura interna de los timers del ATmega32 se muestra a continuación.



### Programación del timer 0

Timer 0 es de 8 bits, así TCNT0 es de 8 bits y su estructura se muestra en la siguiente figura:




#### Registro TCCR0 (Registro de Control del Timer/contador)

TCCR0 es un registro de 8 bits usado para control del Timer0. Los bits para el TCCR0 son mostrados a continuación.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| | FOC0 | WGM00 | COM01 | COM00 | WGM01 | CS02 | CS01 | CS00 |
| Read/Write | W | RW | RW | RW | RW | RW | RW | RW |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**FOC0** D7 Force compare match: This is a write-only bit, which can be used while generating a wave. Writing 1 to it causes the wave generator to act as if a compare match had occurred.

**WGM00, WGM01**

| D6 | D3 | Timer0 mode selector bits |
|---|---|---|
| 0 | 0 | Normal |
| 0 | 1 | CTC (Clear Timer on Compare Match) |
| 1 | 0 | PWM, phase correct |
| 1 | 1 | Fast PWM |

**COM01:00** D5 D4 Compare Output Mode:
These bits control the waveform generator

**CS02:00**

| D2 | D1 | D0 | Timer0 clock selector |
|---|---|---|---|
| 0 | 0 | 0 | No clock source (Timer/Counter stopped) |
| 0 | 0 | 1 | clk (No Prescaling) |
| 0 | 1 | 0 | clk / 8 |
| 0 | 1 | 1 | clk / 64 |
| 1 | 0 | 0 | clk / 256 |
| 1 | 0 | 1 | clk / 1024 |
| 1 | 1 | 0 | External clock source on T0 pin. Clock on falling edge. |
| 1 | 1 | 1 | External clock source on T0 pin. Clock on rising edge. |

**FOC0:** Este es un bit de solo escritura, que se puede usar al generar una onda. Al escribir 1 en él, el generador de ondas actúa como si se

hubiera producido una coincidencia de comparación.

**CS02:CS00 (fuente del reloj Timer0):** Estos bits en el registro TCRR0 son usados para elegir la fuente del reloj. Si CS02:CS00=000, entonces el contador se detiene. Si CS02-CS00 tienen valores entre 001 y 101, el oscilador es usado como fuente de reloj y el timer/contador actúa como un timer. Recuerde que para este caso, los timers son frecuentemente usados para generación de retardos.

**WGM01:00** El timer 0 puede trabajar de 4 formas diferentes: Normal, corrección de fase del PWM, CTC (limpiar el contador coincidir con el valor a comparar) y rapido PWM. Particularmente, En modo CTC, se puede establecer el valor máximo a comparar(TOP). Por lo tanto, si establece el valor de TOP como 100 en lugar de 255 (valor TOP predeterminado), el TCNT cuenta desde INFERIOR (0) a 100 y luego nuevamente a INFERIOR (0). Entonces, si los conteos de TCNT disminuyen, el período de tiempo de la oscilación disminuye. En este modo podemos ajustar la frecuencia. Para ajustar el valor TOP en este caso usamos el OCRn (registro de comparación de salida). Así, el pin OCn (para temporizadores de 8 bits, OC0 y OC2) es el pin de salida para PWM

#### TIFR(Registro de interrupciones de banderas del timer/contador)

El registro TIFR contiene las banderas de los diferentes timers. Donde TOV0 esta relacionada al timer 0.

**TOV0 (Overflow del timer 0)** : La bandera se activa cuando el contador se desborda(overflow), yendo de $FF a $00. Cuando los timers pasan de $FF a $00, la bandera TVO0 se establece en 1 y así pertemanece hasta que el software la borra.

**Lo extraño de esta bandera es que para limpiarla se necesita escribir 1 sobre ella. De hecho esta regla se aplica para todas las banderas del chip del AVR. Así, cuando queremos limpiar la bandera del registro, necesitamos escribir 1 en ella y cero en los otros bits.**

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| | OCF2 | TOV2 | ICF1 | OCF1A | OCF1B | TOV1 | OCF0 | TOV0 |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**TOV0** D0 Timer0 overflow flag bit
0 = Timer0 did not overflow.
1 = Timer0 has overflowed (going from $FF to $00).

**OCF0** D1 Timer0 output compare flag bit
0 = compare match did not occur.
1 = compare match occurred.

**TOV1** D2 Timer1 overflow flag bit

**OCF1B** D3 Timer1 output compare B match flag

**OCF1A** D4 Timer1 output compare A match flag

**ICF1** D5 Input Capture flag

**TOV2** D6 Timer2 overflow flag

**OCF2** D7 Timer2 output compare match flag

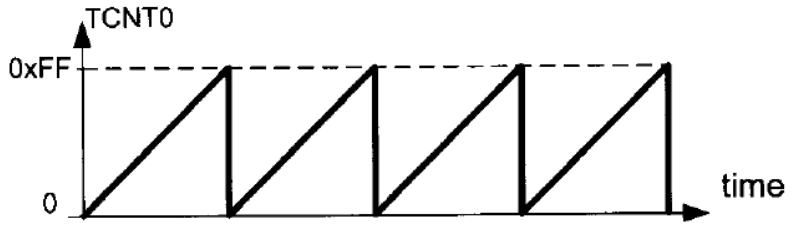
Figura 2: Registro TIFR

Por ejemplo; el siguiente programa limpia TOV0:

```
LDI   R20, 0x01
OUT TIFR, R20
```

## Modo normal

En este modo el contenido del timer/contador aumenta cada ciclo de reloj. El contador se incrementa hata alcanzar su máximo $0xFF$, cuando él pasa de $0xFF$ a $0x00$, la bandera TVO0 se activa. Esta bandera se puede monitorear.



Qué hace el siguiente programa?

```
.INCLUDE "M32DEF.INC"

       LDI   R16,1<<5    ;R16 = 0x20 (0010 0000 for PB5)
       SBI   DDRB,5      ;PB5 as an output
       LDI   R17,0
       OUT   PORTB,R17   ;clear PORTB
BEGIN:RCALL DELAY        ;call timer delay
       EOR   R17,R16     ;toggle D5 of R17 by Ex-Oring with 1
       OUT   PORTB,R17   ;toggle PB5
       RJMP BEGIN

;-------------------Time0 delay
DELAY:LDI   R20,0xF2     ;R20 = 0xF2
      OUT   TCNT0,R20    ;load timer0
      LDI   R20,0x01
      OUT   TCCR0,R20    ;Timer0, Normal mode, int clk, no prescaler
AGAIN:IN    R20,TIFR     ;read TIFR
      SBRS  R20,TOV0     ;if TOV0 is set skip next instruction
      RJMP  AGAIN
      LDI   R20,0x0
      OUT   TCCR0,R20    ;stop Timer0
      LDI   R20,(1<<TOV0)
      OUT   TIFR,R20     ;clear TOV0 flag by writing a 1 to TIFR
      RET
```

**SBRS:** Esta instrucción verifica un solo bit en un registro y omite la siguiente instrucción si el bit está en alto (1).
**SBI**: Establece en alto (1) un bit especificado de un registro.

**Cual es el valor del retardo generado por el timer asumiendo que el cristal(XTAL) es de 8 MHZ?**

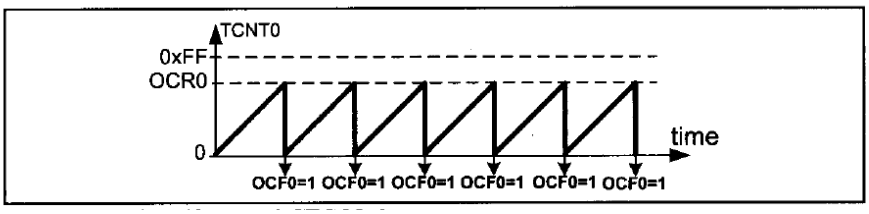
Nosotros sabemos que la frecuencia del timer es de 8MHz, entonces cada ciclo de reloj tiene un periodo de T= $1/(8MHz)$ =0.125 micro segundos. Esto significa que el Timer0 cuenta hacia arriba cada 0.125 microsegundos, de esta forma

$$\text{Delay=número de cuentas (0.125 microsegundos).}$$

Si el contador empieza en 0xF2 y la bandera TOV0 se activa después de llegar a 0xFF, entonces, sumaríamos un ciclo extra (0xFF-0xF2=13+1). Así, tendremos que el retardo es de 1.75 microsegundos para la mitad del pulso.

**Modo de programar–Limpiar Timer 0 usando comparación de coincidencia (CTC)**

El registro OCR0 es usado con el modo CTC. Al igual que en el modo normal, en el modo CTC, el timer es incrementado con un reloj. Pero cuenta hasta que el contenido del registro TCNT0 llegue a ser igual al contenido de OCR0; entonces, el timer será limpiado y la bandera OCF0 será activada cuando el siguiente ciclo de reloj ocurra. La bandera OCF0 esta localiza en el registro TIRF.



Qué hace el siguiente problema?

```
.INCLUDE "M32DEF.INC"

      LDI   R16,0x08
      SBI   DDRB,3            ;PB3 as an output
      LDI   R17,0
BEGIN:OUT   PORTB,R17         ;PORTB = R17
      RCALL DELAY
      EOR   R17,R16           ;toggle D3 of R17
      RJMP  BEGIN
;---------------- Timer0 Delay
DELAY:LDI   R20,0
      OUT   TCNT0,R20
      LDI   R20,9
      OUT   OCR0,R20          ;load OCR0
      LDI   R20,0x09
      OUT   TCCR0,R20         ;Timer0, CTC mode, int clk
AGAIN:IN    R20,TIFR          ;read TIFR
      SBRS  R20,OCF0          ;if OCF0 is set skip next inst.
      RJMP  AGAIN
      LDI   R20,0x0
      OUT   TCCR0,R20         ;stop Timer0
      LDI   R20,1<<OCF0
      OUT   TIFR,R20          ;clear OCF0 flag
      RET
```

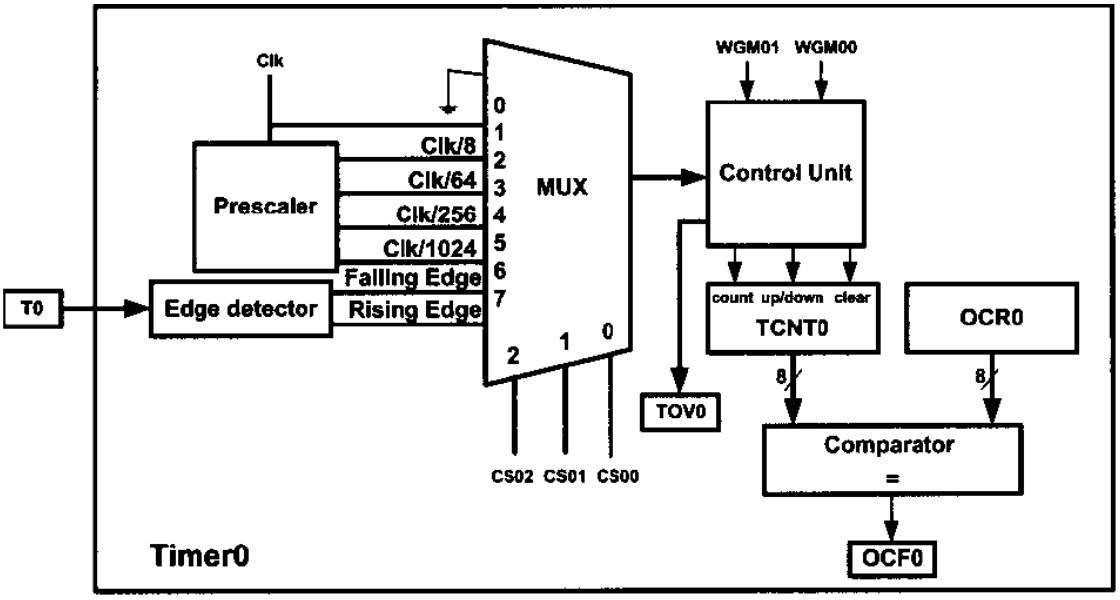
## Programando el contador

En las secciones previas, usamos el timer del AVR para generar retardos. **El AVR puede además ser usado para contar, detectar y medir el tiempo de eventos pasando fuera del AVR.** El uso del timer como un contador de eventos se cubre en esta sección. Cuando el timer es usado para generar retardos, el cristal del AVR es usado como la fuente de frecuencia. Cuando es usado como un contador, es un pulso externo el que incrementa el registro TCNTx. Note que, en modo contador cada registro como TCCR, OCR0 y TCNT son los mismos como para el timer discutido previamente.

### Bits CS00, CS01 y CSO2 en el registro TCCR0

Recordando de la sección previa que bit CS (selector de reloj) en el registro TCCR0 decide la fuente del reloj para el timer. Si CS02:00 esta

entre 1 y 5, el timer obtiene pulsos del cristal oscilador. En contraste, cuando CS02:00 esta entre 6 y 7, el timer es usado como un contador y obtiene sus pulsos de una fuente externa del AVR.

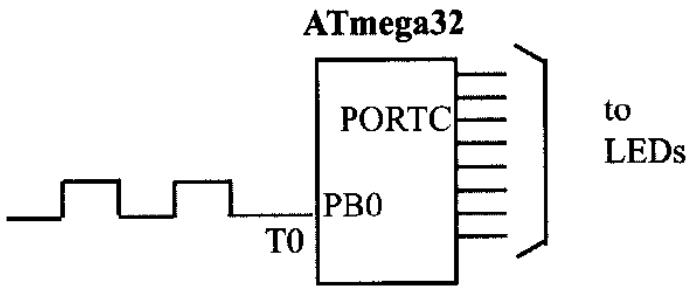



Por lo tanto, cuando CS02:00 es 6 ó 7, el contador TCNT0 cuenta hacia arriba (pulso subida o de bajada) cada que los pulsos son alimentados del pin T0 (Timer/ contador 0–entrada externa de reloj). En Atmega32, T0 es la función alternativa de PORTB.0.

En el siguiente ejemplo, nosotros usamos el Timer0 como un contador de eventos que cuenta hacia arriba cada que los pulsos son alimentados en PB0.

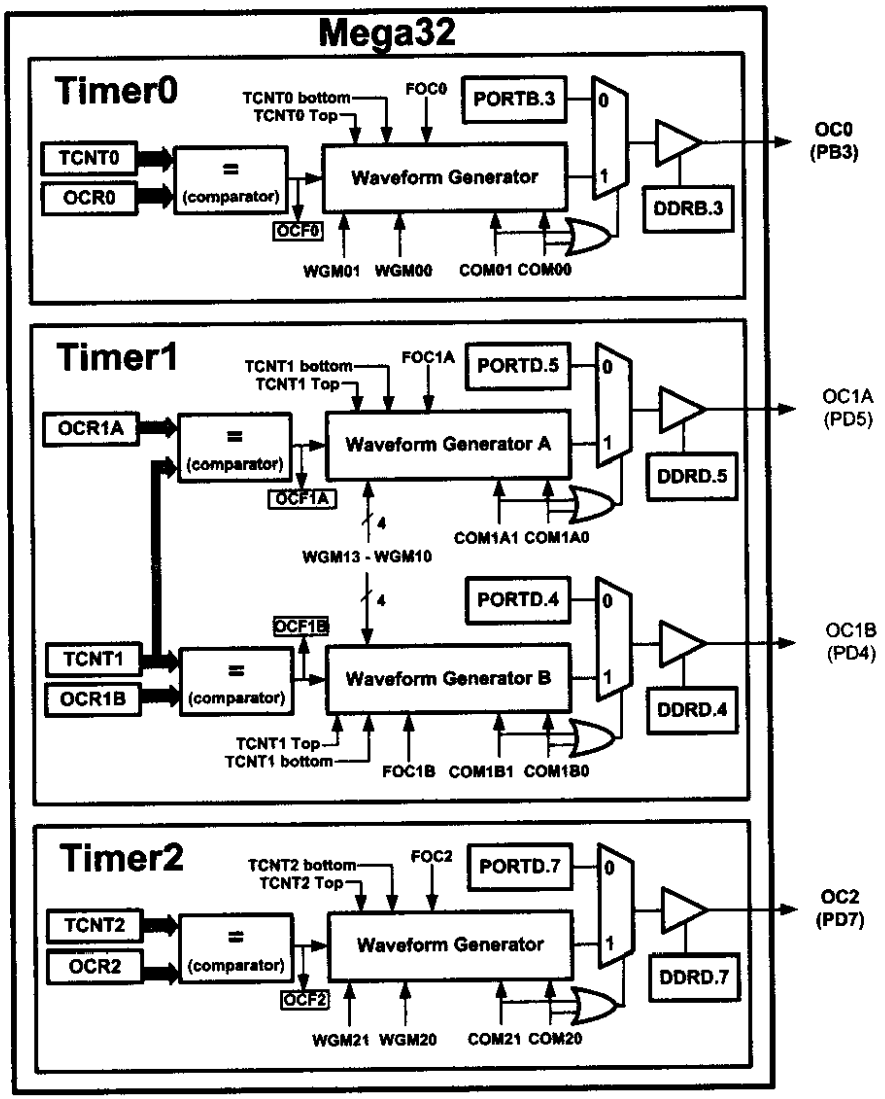
```
.INCLUDE "M32DEF.INC"
      CBI   DDRB,0            ;make T0 (PB0) input
      LDI   R20,0xFF
      OUT   DDRC,R20          ;make PORTC output
      LDI   R20,0x06
      OUT   TCCR0,R20         ;counter, falling edge
AGAIN:
      IN    R20,TCNT0
      OUT   PORTC,R20         ;PORTC = TCNT0
      IN    R16,TIFR
      SBRS  R16,TOV0          ;monitor TOV0 flag
      RJMP  AGAIN             ;keep doing if Timer0 flag is low
      LDI   R16,1<<TOV0
      OUT   TIFR, R16         ;clear TOV0 flag
      RJMP  AGAIN             ;keep doing it
```

PORTC is connected to 8 LEDs and input T0 (PB0) to 1 Hz pulse.



## Modos de PWM en timers de 8 bits

Como hemos mencionado la familia de microcontroladores ATmega32 viene con al menos 3 timers, los cuales pueden ser usados como generadores de onda como se muestra en la siguiente figura.

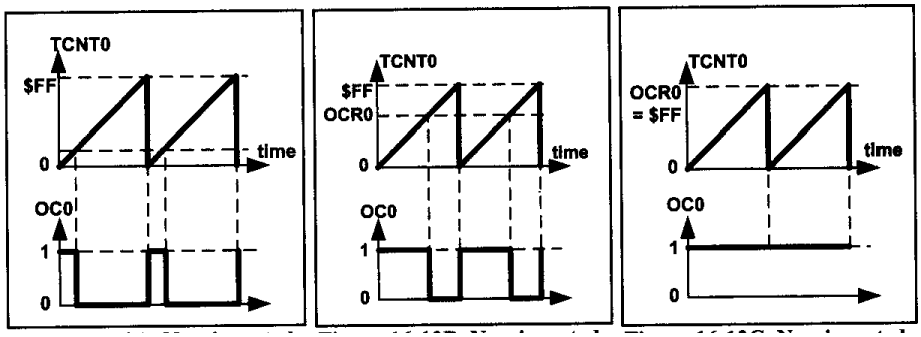


Veremos como usar el CPU para crear el equivalente a salidas PWM. La ventaja de utilizar el PWM incorporado del AVR es que nos da la opción de programar el período y el ciclo de trabajo.

#### Modo PWM rápido

En el PWM rápido, el contador cuenta como el modo normal.En la siguiente figura se puede ver la reacción del generador de onda cuando el comparador de emparejamiento ocurre mientras el timer esta en modo rápido PWM.

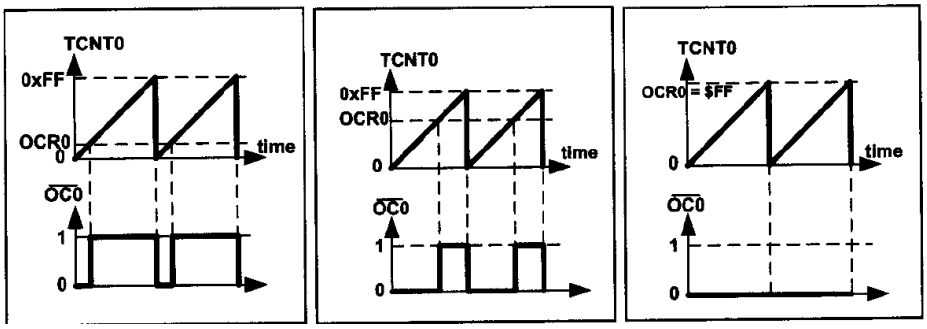
Cuando COM01:00=00 el pin OC0 opera como puerto de I/O. Cuando COM01:00=10, el generador de forma de onda borra el pin OC0 y cada vez que se produce una comparación lo establece en alto (1). Este modo es llamado *PWM no invertido*.




Como se puede observar en las figuras, en el PWM no invertido, el ciclo de trabajo del generador de ondas incrementa cuando el valor de OCR0 incrementa.

Cuando COM01:00=11, el generador de forma de onda activa el pin OC0 cuando se produce la coincidencia de comparación y lo borra al terminar el conteo.

Podemos ver que en el modo de PWM invertido cuando el valor de OCR0 se incrementa, el ciclo de trabajo del generador de onda se decrementa.




**Frecuencia de la onda generada en modo PWM rápido**

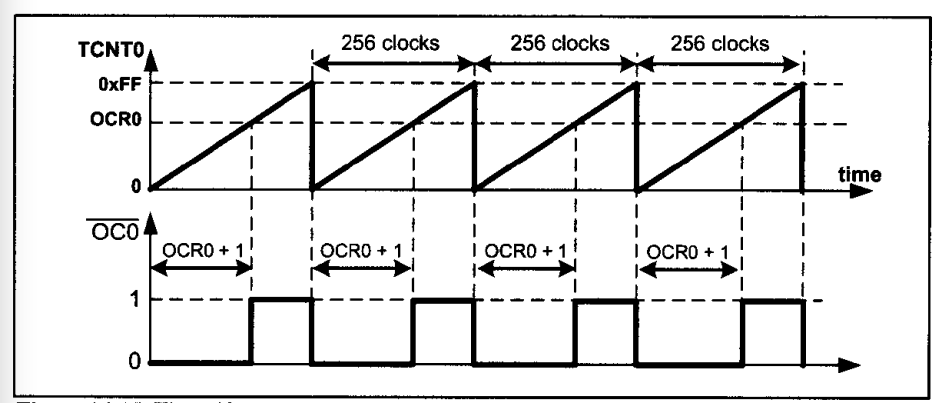
En modo PWM rápido, el timer cuenta de 0 a 0xFF y entonces regresa. Así, la frecuencia del generador de onda es de 1/256 de la frecuencia del timer del reloj. La frecuencia del reloj del timer puede ser seleccionado usando el preescalador. Así, en timers de 8 bits la frecuencia del generador de onda puede ser calculada como sigue:

$$
\begin{aligned}
F_{onda\ generada} &= F_{\frac{frec.\ del\ retardo\ de\ reloj}{256}} \\
F_{frec.\ del\ retardo\ de\ reloj} &= \frac{F_{oscilador}}{N} \\
F_{onda\ generada} &= \frac{F_{frec.\ del\ oscilador}}{256N}
\end{aligned}
$$

**Ciclo de trabajo (duty cycle) del generador de onda en modo PWM rápido**

El ciclo de trabajo del modo generador puede ser determinado usando registro OCR0. Cuando COM01:00=10 (modo no invertible), el valor más grande de OCR0 resulta en el ciclo de trabajo más grande (OCR0=255, el OC0 es 256 ciclos de reloj lo cual significa siempre en alto y ciclo de trabajo del 100 %).

De esta forma, el ciclo de trabajo puede ser calculado usando la siguiente




formula para el modo no invertible.

$$Duty\ cycle = \frac{OCR0 + 1}{256} \times 100$$

Similarmente, la formula para calcular el ciclo de trabajo para el modo invertible es

$$Duty\ cycle = \frac{255 - OCR0}{256} \times 100$$

*Ejemplo 1.-* Para generar una onda con ciclo de trabajo del 75 % en modo no invertible, calcular el valor del registro OCR0.

**75 = (OCR0 + 1) × 100 / 256 ➔ OCR0 + 1 = 75 × 256 / 100 = 192 ➔ OCR0 = 191**

*Ejemplo 2.-* Encontrar el valor de TCCR0 para inicializar Timer0 en modo rápido PWM, modo de onda PWM no invertido, y no preescalado.

WGM01:00=11-Fast PWM mode
COM01:00=10= Non-inverted PWM
CS02:00=001= No prescaler
De esta forma, el valor delregistro TCCR0 será:

$$TCCR0 = 01101001$$

*Ejemplo 3.-* Asumiendo que XTAL=8MHz, usando modo no invertible, escribir un programa que genere una onda con frecuencia de 31,250 Hz y un ciclo de trabajo del 75 %.

**31,250 = 8M / (256 × N) ➔ N = 8M / (31,250 × 256) = 1 ➔ N = 1 ➔ No prescaler**

Después

```
.INCLUDE "M32DEF.INC"
     SBI   DDRB,3
     LDI   R20,191
     OUT   OCR0,R20
     LDI   R20,0x69
     OUT   TCCR0,R20
```

## Ejemplos a resolver

*Ejemplo 4.-*Asumiendo XTAL=8MHz, usando modo no invertido, escribir un **programa** que genere una onda con frecuencia de 3906.25 Hz y un ciclo de trabajo de 37.5 %.

**3906.25 = 8M / (256 × N) ➔ N = 8M / (3906.25 × 256) = 8 ➔ the prescaler value = 8**
**37.5 = 100 × (OCR0 + 1) / 256 ➔ OCR0 + 1 = (256 × 37.5) / 100 = 96 ➔ OCR0 = 95**

```
.INCLUDE "M32DEF.INC"
     SBI   DDRB,3
     LDI   R20,95
     OUT   OCR0,R20   ;OCR0 = 95
     LDI   R20,0x6A
     OUT   TCCR0,R20  ;Fast PWM, N = 8, non-inverted
HERE: RJMP  HERE
```

*Ejemplo 5.-* Reescribir el ejemplo anterior usando modo invertido.

## Encontrando valores para ser cargados en el timer (continuación de retardos)

**Asumiendo** que nosotros sabemos el valor del retardo que necesitamos; la pregunta sería: Cómo encontrar los valores necesarios para el registro TCNT0?, para calcular los valores a ser cargados en el registro TCNT0; podemos seguir los siguientes pasos:

- Calcular el período del reloj usando la formula:

$$T_{clock} = 1/F_{timer}$$

  Donde $F_{timer}$ es la frecuencia del reloj usado. Por ejemplo, en modo no escalado, $F_{timer} = F_{frec.\ oscilador}$. Donde, $T_{clock}$ da el periodo en el cual el timer se incrementa.

- Dividir el retardo deseado por $T_{clock}$. Esto determinará cuantos pulsos de reloj son necesarios.

- Realizar $256 - n$, donde n es el número decimal que se obtuvo del paso anterior.

- Convertir el número decimal del paso anterior en hexadecimal, siendo este valor el que se cargará en el registro del timer.

*Ejemplo.-* Asumiendo XTAL=8MHz, escribir un programa que genere una onda cuadrada con un período de 12.5 microsegundos.

Para una onda cuadrada con un período de $T = 12{,}5\mu s$ tendríamos un retardo de 6.25 $\mu s$. Por lo tanto, el contador cuenta cada 0.125 $\mu s$ ya que XTAL es de 8MHz. Esto significa que el número de ciclos de reloj para generar el retardo esta dado por: $\frac{6{,}25\mu s}{0{,}125\mu} = 50$. Entonces, TCNT0 debe ser igual a 256-50=206= 0xCE.

Asumiendo que XTAL=8 MHz, modificar el programa anterior para generar una onda cuadrada de 16 KHz de frecuencia.

```
.INCLUDE "M32DEF.INC"

      LDI   R16,0x08
      SBI   DDRB,3       ;PB3 as an output
      LDI   R17,0
      OUT   PORTB,R17
BEGIN:RCALL DELAY
      EOR   R17,R16      ;toggle D3 of R17
      OUT   PORTB,R17    ;toggle PB3
      RJMP BEGIN
;---------------- Timer0 Delay
DELAY:LDI   R20,0xCE
      OUT   TCNT0,R20    ;load Timer0
      LDI   R20,0x01
      OUT   TCCR0,R20    ;Timer0, Normal mode, int clk, no prescaler
AGAIN:IN    R20,TIFR     ;read TIFR
      SBRS  R20,TOV0     ;if TOV0 is set skip next instruction
      RJMP  AGAIN
      LDI   R20,0x00
      OUT   TCCR0,R20    ;stop Timer0
      LDI   R20,(1<<TOV0)
      OUT   TIFR,R20     ;clear TOV0 flag
      RET
```

## Preescalamiento

Como hemos visto en los ejemplos hasta ahora, el tamaño del retardo de tiempo depende de dos factores: (a) la frecuencia del cristal, y (b) el registro de 8 bits del temporizador. Ambos factores están fuera del control del programador AVR. Además hemos visto hasta ahora que, sin un preescalador habilitado, la frecuencia del oscilador de cristal se alimenta directamente en Timer0. Sin embargo, si habilitamos el bit del preescalador en el registro TCCR0, podemos **dividir el reloj antes de alimentarlo a Timer0**. Existen 3 bits en el registro TCCR0 que dan las opciones del número por el que podemos dividir. Como se mostrado, este número puede ser 8, 64, 256, 1024.

*Ejemplo 1.-* Encontrar el valor para TCCR0 si queremos programar el Timer0 en modo normal con un preescalamiena de 64 usando clock interno para la fuente del reloj:

TCCR0= 0000 0011.

*Ejemplo2.-* Asumir un cristal XTAL=8MHz, (a) encontrar el período de reloj alimentado en el Timer0 si la opción de preescalamiento de 1024 es elegida. (b) Mostrar cual es el tiempo de retardo más grande que podemos usar en este preesacalamiento en el Timer0.

(a) 8 MHz $\frac{1}{1024}$= 7812.5 Hz, debido al preescalamiento 1:1014. Entonces

$$T = \frac{1}{7812{,}5Hz} = 0{,}128ms \tag{1}$$

(b) Para obtener el delay más grande, hacemos TCNT0 cero. Haciendo esto el timer contará de 00 a 0xFF, y luego dará vuelta al activar la bandera TOV0. Como resultado, se moverá a través de 256 estados. Por lo tanto, se tendrá un delay= (256-0) (128) $\mu s = 0.032768$ segundos.