PROGRAMACIÓN COMENTADA (**generación de una señal SPWM unipolar con ATmega 8**)

Vista general de la programación.

Esta programación está diseñada en base a la teoría de los inversores modulados por ancho de pulso con tensión unipolar, para ello se investigó la forma de obtener las dos señales necesarias, senoidal y triangular, en el ATmega 8, y finalmente compararlas y obtener a la salida del microcontrolador una señal SPWM unipolar.

Adicionalmente se puede agregar una manera para variar la amplitud y la frecuencia de forma proporcional como lo indica la teoría para un **CONTROL ESCALAR,** la cual nos permite variar la velocidad de un motor asíncrono de corriente alterna.

Para la generación de la señal triangular utilizamos uno de los periféricos que integra el microcontrolador, este es el TIMER. En el TIMER podemos generar distintas señales PWM, pero la señal que más se acerca a la teoría de inversores, es la **PWM de fase correcta** la cual tendremos que configurar y luego habilitar en el programa principal. Siguiendo la teoría de inversores la frecuencia de la triangular tendrá que ser m_f veces la frecuencia de la senoidal deseada.

Para la generación de la señal senoidal, utilizaremos una librería especial que incluye el programa Atmel Studio (`#include <math.h>`) el cual nos permite utilizar una función `(sin)` en la cual tendremos que insertar el argumento y la amplitud para generar una señal senoidal.

Debido a que el microcontrolador no puede generar señales continuas, lo que haremos es muestrear la señal senoidal de acuerdo al índice de modulación de frecuencia m_f, ya que estamos utilizando al TIMER en modo **PWM de fase correcta,** habrá dos momentos de comparación en el TIMER uno de subida y otro de bajada para ello tendremos que muestrear el doble de veces, lo que daría $2 \times m_f$ de muestras, para luego colocarlo todos estos valores en una tabla o array con un tamaño de $2 \times m_f$, `array [2 x mf]`.

Algo positivo a la hora de generar una señal SPWM unipolar es que podemos utilizar las mismas ondas generadas en el semiperiodo positivo para el semiperiodo negativo, ahorrándonos conversiones como seria en el caso de generar una señal SPWM bipolar. Por ello ahora ya no necesitaremos una tabla de $2 \times m_f$, `array [2 x mf]`, sino una tabla de m_f, `array [mf]`.

Por ejemplo si el m_f es 21 entonces dividiremos en 21 partes un semiperiodo de la senoidal, cada una con distinta amplitud debido a que los 21 valores están generados para distintos ángulos.

Todos estos valores de la tabla estarán relacionados con el registro `OCR1X` del TIMER debido a que es este registro que está en constante comparación con la señal triangular generado por el TIMER. Esto quiere decir que actualizaremos m_f veces el valor del registro `OCR1X` en un semiperiodo de la onda senoidal.

Es de esta manera que emularemos la comparación de una señal senoidal con una triangular, en el ATmega 8, y al resultado de su comparación, como indica la teoría, obtendremos la señal SPWM unipolar.

Comentario de la programación

Se comentara por bloques la programación y se le asignara un cuadro de flujo a cada bloque, al final integraremos todos los cuadros en un diagrama de flujo.

```
/*
 * control_spwm_con_atmega8.c
*/
/*****************************************************************************
/*

En este primer bloque incluiremos las librerías necesarias para el resto de la
programación.

*/

#include <avr/io.h>        //para utilizar las entradas y salidas del
microcontrolador
#include <avr/interrupt.h> //para utilizar las interrupciones internas generadas por
el TIMER
#include <math.h>          //para generar la función senoidal

/*****************************************************************************
/*

En este segundo bloque generamos las variables globales utilizadas a lo largo del
programa, ya sea en la función principal como en las demás funciones.

Estas variables tienen distintos tipos de datos ya sea del tipo int (16 bits) o char
(8 bits) estáticas y volátiles.

SinDivisions (21), es el índice de modulación de frecuencia (mf) según teoría de
inversores decimos que el mf debe ser un valor cercano a 21. Para que no sea ni tan
grande evitando generar perdidas por conmutación ni tan pequeño evitando utilizar un
filtro más grande para atenuar las armónicas desplazadas.

microMHz, es la variable para designar con qué frecuencia de CPU estamos trabajando
y realizar el cálculo de la frecuencia de la señal triangula de forma correcta.

Freq, es la variable para designar la frecuencia de la senoidal que también estará
presente en el cálculo de la frecuencia de la triangular.

period, es la variable para designar el máximo valor que toma el TIMER en su conteo,
multiplicando esta variable por el periodo del TIMER (con o sin prescalador)
tendremos el periodo real de la señal triangular.

lookUp[SinDivisions], es un array en donde estarán alojadas todas las muestras de la
senoidal en nuestro caso  SinDivisions muestras.

theTCCR1A, es una variable de un tamaño de 8 bits o un byte la cual representa un
registro de configuración para el TIMER. La utilizaremos para conmutar las salidas
de PWM, es decir durante un semiperiodo positivo de la senoidal estará activada la
salida OC1 y desactivada la salida OC2 y viceversa durante el semiperiodo negativo.

Esto nos sirve para poder controlar un inversor de puente completo una de las ramas
diagonales con el pin OC1 y la otra rama con el pin OC2.
```

TOP, esta variable representa la amplitud máxima de la senoidal en este primer proyecto consideramos que sea de la misma amplitud que la triangular es decir, ***TOP = period***, relacionando con la teoría de los inversores decimos que el índice de modulación de amplitud (ma) es igual a 1.

num, esta variable vendría a ser un puntero o índice la cual ubicara un valor del array lookUp[SinDivisions]

*/

```
#define SinDivisions (21)                       // Sub divisiones de la onda senoidal
static int microMHz = 8;                        // Frecuencia del CPU en microherz
static int freq = 50;                           // Frecuencia senoidal
static long int period;                         // Periodo de la PWM
static unsigned int lookUp[SinDivisions];       // Tabla de muestras de la senoidal
static char theTCCR1A = 0b10000010;             // variable para TCCR1A
long int TOP;                                   // Amplitud de la senoidal
volatile int num;                               // Indice de la table de muestras

/*****************************************************************************
/*
```

En este tercer bloque generamos los prototipos de funciones para hacer la programación ordenada y poder colocar estas funciones después de la función principal.

void setup_TIMER(void), esta función se ejecutara solo una vez en el programa debido a que es una función de configuración.

void table_block(int TOP), en este primer proyecto esta función se ejecutara solo una vez debido a que las muestras de la senoidal se mantendrán constantes, en un siguiente proyecto cuando necesitemos variar la amplitud de la senoidal tendremos que llamar a esta función cada vez que quisiéramos variar la amplitud de la senoidal y sobrescribir sobre la tabla nuevos valores o muestras de la nueva senoidal.

*/

```
void table_block(int TOP);        //Prototipo de la tabla de valores muestreados
void setup_TIMER(void);           //Prototipo para configurar el TIMER

/*****************************************************************************

/*
```

En este cuarto bloque se encuentra la función principal la cual será primera en ejecutarse por la memoria del programa,

- En un inicio se calcula el periodo de la PWM que será el mismo valor para la amplitud de la senoidal.
- Luego llamamos a la función ***table_block(period)*** para generar las muestras de la senoidal, retornando a la función principal con toda la tabla llena.

- Seguidamente hacemos otra llamada a la función ***setup_TIMER()*** para configurar al TIMER de acuerdo a nuestras especificaciones, retornando a la función principal con el TIMER encendido y listo para recibir mandos.
- Luego se habilita todas las interrupciones con la función ***sei()***
- Por último se deja en bucle infinito al programa principal, para que genere una señal SPWM en todo su funcionamiento del microcontrolador.

*/

```
int main(void)              // Funcion principal
{
  period = microMHz*1e6/freq/(2*SinDivisions);  // Periodo de la PWM

  table_block(period);      // Funcion de valores muestreados de la senoidal
  setup_TIMER();            // Funcion para configurar el TIMER
  sei();                    // Habilitar interruciones globales.

  while(1)                  // Bucle infinito
  {
  }
}

/*****************************************************************************
/*
```

En este quinto bloque se encuentra la función para configurar el TIMER 1. Elegimos este TIMER debido a que este posee dos salidas de PWM las cuales podemos conmutar y así generar las señales de control para un inversor de puente completo.

En el programa se seguirá el siguiente flujo.

- Se configura en modo PWM de fase correcta, sin prescalar.
- Se habilita las interrupciones por sobre flujo y por input capture,
- Luego se toma el valor del registro ICR1 como TOP para la onda senoidal
- y por último se configura como salidas a PB1 y PB2.

Finalmente regresa a la función principal, main(void), con el TIMER 1 configurado y encendido.

*/

```
void setup_TIMER(void)
{
       TCCR1A = theTCCR1A; // 0b10000010;
        /*10 compA se limpia a cero con una comparación, set at BOTTOM for compA.
          00 compB incialmente desconectado, luego se conmuta por el compA.
          00
          10 WGM1 1:0 para la forma de onda 11.
        */
       TCCR1B = 0b00010001;
        /*000
          10 WGM1 3:2 para la forma de onda 11.
          001 sin prescalador para el conteo  / 000 El conteo es parado.
        */
```

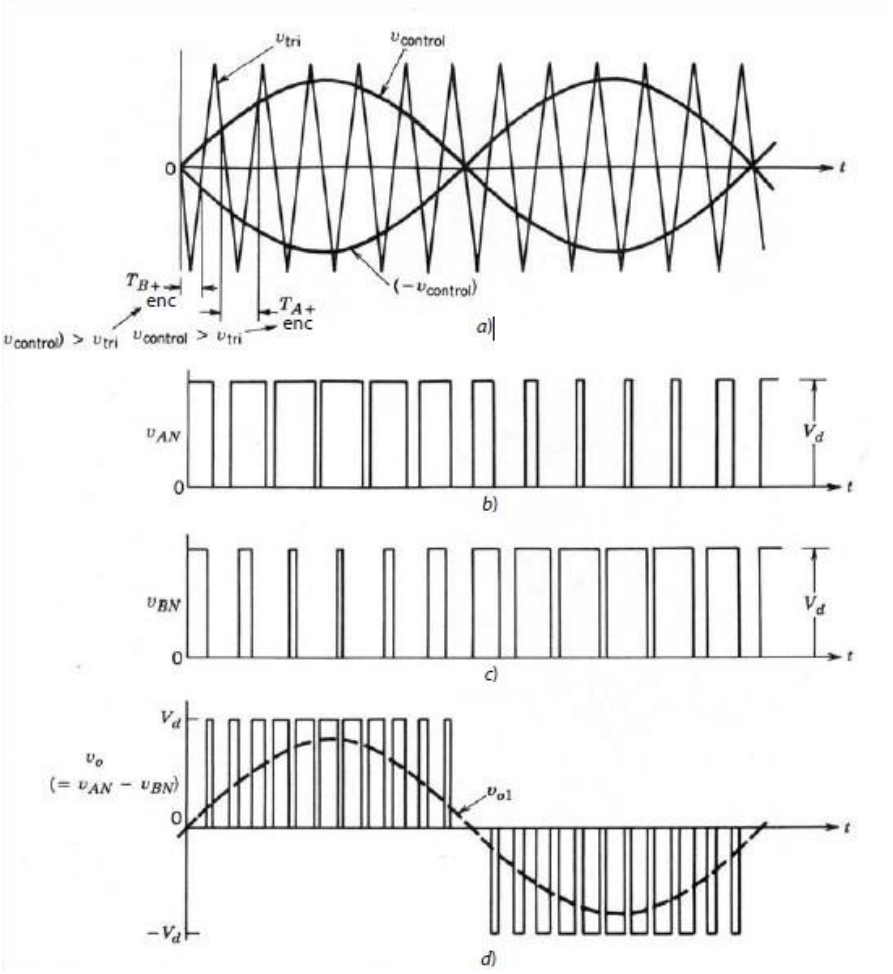
```
	TIMSK = 0b00100100;
	 /*
	   1 TOV1 Interrupción por sobre flujo habilitado.
	   1 ICF1 Interrupción por input capture habilitado.
	 */
	ICR1   = period;   /* Periodo para 8MHz de reloj interno, para una frecuencia
	de conmutación de 2kHz en 21 subdivisiones por 50Hz de onda senoidal */

	DDRB = 0b00000110; // Configurar PB1 y PB2 como salidas.
}

/*****************************************************************************
/*

En este sexto bloque generamos la tabla que incluyen las muestras para la onda
senoidal.

   • Primero creamos una variable temp del tipo double, el cual tiene un rango de
     ±1.18E-38 a ±3.39E+38 con una precisión de 7 bits.
   • Generamos un bucle hasta llenar la tabla de todas las muestras necesarias,
     como ya mencionamos tendrá mf muestras.
   • Dentro del bucle se guardara estos valores muestreados de la senoidal en
     nuestra variable temp, ya que estos serán del tipo decimal.
   • Finalmente estos valores se redondean a una variable entera y colocadas en
     cada lugar de la tabla según el indice.

*/

void table_block(int TOP)
{
	double temp; //Variable doble para funciones <math.h>.

	for(int i = 0; i < SinDivisions; i++)
	{ // Generando tabla de muestras para la onda senoidal.
		temp = sin((i)*M_PI/SinDivisions)*TOP;
		lookUp[i] =  int (temp);	// Redondeado a un entero.
	}
}

/*****************************************************************************
/*

En este séptimo bloque se encuentra la función de interrupción del TIMER por sobre
flujo, este sobre flujo varía según el modo del PWM que se escoja, en nuestro caso
escogimos un PWM de fase correcta esto hará que nuestro sobre flujo se active cuando
el valor continuo del TIMER llegue a cero o su valor BOTTOM, según la hoja de datos.

En esta función una vez que se termine el semiperiodo positivo se desactiva la
salida OC1 y se activa la salida OC2 y además se resetea el índice a 0 para
reutilizar la tabla generada para el semiperiodo positivo en el semiperiodo
negativo.

Por último cada vez que se genera esta interrupción se actualiza el valor del
registro OCR1A y OCR1B, de esta manera se varía el ciclo de trabajo o ancho de pulso
de la PWM a la salida de OC1 y OC2 según cual este activada.
```

```
*/

ISR(TIMER1_OVF_vect)
{
	static int delay1;		// variable estática

	if (delay1 == 1) {

		theTCCR1A ^= 0b10110000;	// Toggle connect and disconnect of compare
						   output A and B // TCCR1A = 0b10000010
		TCCR1A = theTCCR1A;		//TCCR1A = 0b10000010
		delay1 = 0;
	}
	 if (num >= SinDivisions){
		num = 0;			// Reset num
		delay1++;
	}

	OCR1A = OCR1B = lookUp[num];	// change duty-cycle every period.
	num++;
}

/*****************************************************************************
/*

En este octavo bloque se encuentra la función de interrupción del TIMER por input
capture, en este caso también influye el modo en que se trabaje con el PWM del
TIMER, en nuestro caso PWM de fase correcta para ello esta interrupción se activa
cada vez que el valor continuo del TIMER llega a su valor máximo o TOP.

Cada vez que se ingresa a esta función por interrupción actualizamos los valores de
OCR1A y OCR1B para variar el ancho de pulso a la salida del PWM en el OC1 y OC2
según la salida que estuviera activada.

*/

ISR(TIMER1_CAPT_vect)
{
	OCR1A = OCR1B = lookUp[num];
	num++;
}

/*****************************************************************************
```

DATOS Y TABLAS ADICIONALES

Tabla de variables y tipos de datos del lenguaje C		
Tipo de dato	**Tamaño en bits**	**Rango de valores que puede adoptar**
char	8	0 a 255 ó -128 a 127
signed char	8	-128 a 127
unsigned char	8	0 a 255
(signed) int	16	-32,768 a 32,767
unsigned int	16	0 a 65,536
(signed) short	16	-32,768 a 32,767
unsigned short	16	0 a 65,536
(signed) long	32	-2,147,483,648 a 2,147,483,647
unsigned long	32	0 a 4,294,967,295
(signed) long long (int)	64	-2^{63} a $2^{63} - 1$
unsigned long long (int)	64	0 a $2^{64} - 1$
float	32	±1.18E-38 a ±3.39E+38
double	32	±1.18E-38 a ±3.39E+38
double	64	±2.23E-308 a ±1.79E+308

GENERACION DE UNA SEÑAL SPWM UNIPOLAR



OTRA FORMA DE GENERAR LA SEÑAL SPWM UNIPOLAR

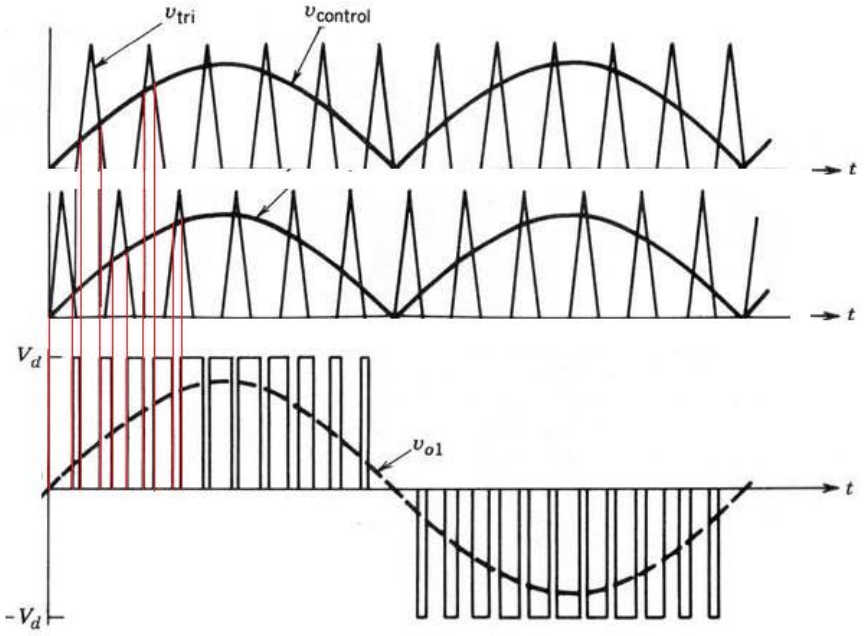
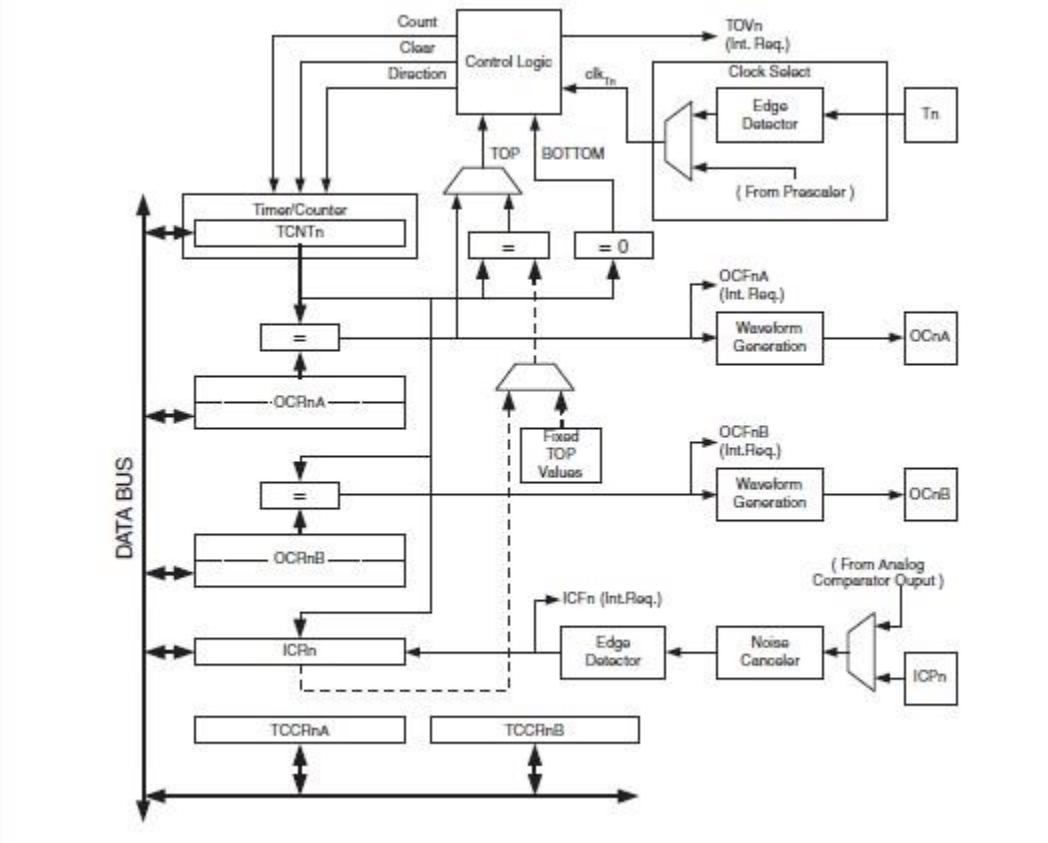
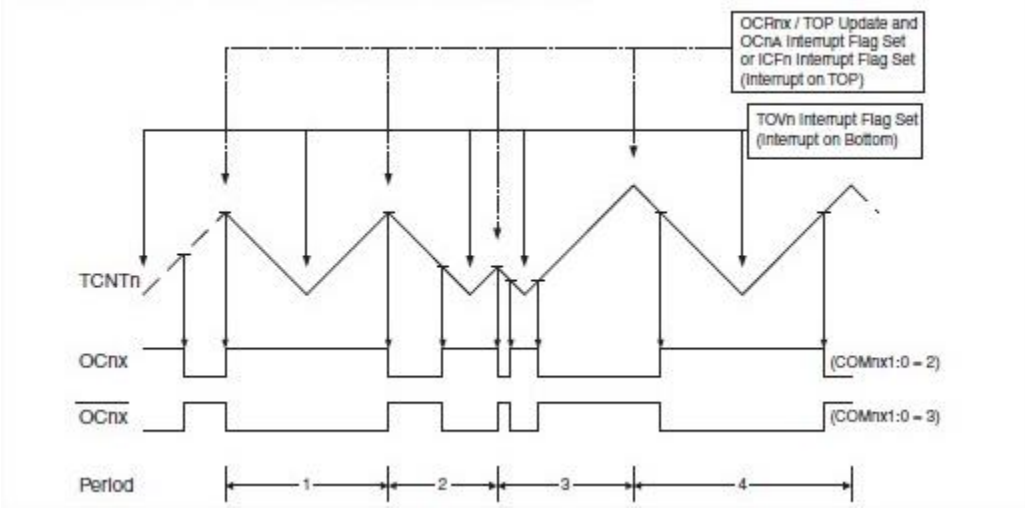


DIAGRAMA DE BLOQUES DEL TIMER



PWM DE FASE CORRECTA (ATmega 8)



Rampa con el valor actualizado después de la interrupción por sobre flujo

Rampa con el valor actualizado después de la interrupción por input capture