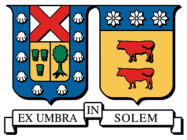

UNIVERSIDAD TECNICA
FEDERICO SANTA MARIA

DEPARTAMENTO DE
ELECTRONICA

PWM STM32-L4

Laboratorio Nikola Tesla - PowerLab

Fecha: 5 de febrero de 2025
Autores: Vicente Andrade
Guía: César Silva

Índice

1. Introducción

La presente documentación describe la implementación de un firmware para el microcontrolador `STM32L476RG`, utilizando **CubeIDE v1.17.0** y las librerías **HAL**. Además, se integra la **lectura de un encoder** (mediante `TIM2` en modo Encoder) y la **lectura de ADC** con **DMA**, disparada externamente por `TIM1_TRGO`. El objetivo primordial es demostrar que, usando un solo microcontrolador, se puede:

- Generar señales PWM en 3 fases (con `TIM1`).
- Sincronizar lecturas del ADC mediante `TRGO` para adquirir datos en tiempo real sin sobrecargar la CPU.
- Capturar la posición de un encoder (con `TIM2` en modo Encoder).
- Procesar todo **en la misma rutina de interrupción** del PWM si se desea un control en tiempo real.

2. Marco Teórico

A continuación, se señalan tres elementos centrales:

1. Principios del PWM Trifásico.
2. Descripción de la captura de encoder (modos TI1, TI2, TI12).
3. Configuración de ADC con DMA para lectura continua (disparo externo TRGO).

2.1. Principios del PWM Trifásico

El PWM trifásico consiste en generar tres señales PWM desfasadas 120° entre sí, típicamente para excitar un motor trifásico (inducción, BLDC, PMSM, etc.). En el `STM32L476RG`, se acostumbra usar `TIM1` o `TIM8` por sus funcionalidades de timer avanzado.

2.1.0.1 Center-Aligned vs. Edge-Aligned

- **Edge-Aligned Mode**: el contador se incrementa desde 0 hasta `ARR` y se reinicia. Genera una onda *diente de sierra*. El periodo del timer en Edge-Aligned se calcula mediante:

$$T_{\text{timer}} \;=\; \frac{(\text{psc}+1)\,(\text{arr}+1)}{f_{\text{clk}}}.$$

- **Center-Aligned Mode**: el contador sube de 0 a `ARR` y baja de `ARR` a 0, generando una señal triangular con dos comparaciones por ciclo. Esto reduce armónicos y mejora la simetría de la modulación.[1] El periodo del timer en Center-Aligned:

$$T_{\text{timer}} \;=\; \frac{2\,(\text{psc}+1)\,(\text{arr}+1)}{f_{\text{clk}}}.$$

[1] Ver *Reference Manual for STM32L4* (RM0351) en la sección de Timers Avanzados.

2.1.0.2 Dead Time

- Evita cortocircuitos entre transistores altos y bajos en la misma pierna de un inversor.
- Se configura en *Break and Dead Time Management* (registro `BDTR`).
- Usualmente se elige un valor en `DTG` (Dead Time Generator) según la frecuencia de conmutación y las características de los dispositivos de potencia. Ver snippet de ejemplo a continuación:

```
// 0<DTG-1<255
TD = (80*10^6)^{-1};
if (DTG <= 127) {
    t_dead = DTG * TD;
} else if (DTG <= 191) {
    t_dead = (DTG-128+64) * 2 * TD;
} else if (DTG <= 223) {
    t_dead = (DTG-192+32) * 8 * TD;
} else if (DTG <= 255) {
    t_dead = (DTG-224+32) * 16 * TD;
} else {
    return -1;
}
```

Script 1: Snippet BDTR

2.1.0.3 Auto-Reload Register (ARR)

- Define la frecuencia de la portadora PWM. Con un *clock* interno de 80 MHz, `PSC`=0 y `ARR`=7679 en modo *center-aligned*, se obtiene un periodo en torno a 192 µs (aprox. 5.2 kHz).
- Note, `PSC` corresponde a un divisor de frecuencias con entrada el *clock* interno, tal que si `PSC` es igual a `0`, entonces el timer trabajará con la frecuencia máxima disponible.
- `ARR` corresponde a la cota superior del contador, note que nuestra portadora corresponde a `CNT`, el contador asignado al timer.

2.1.0.4 Compare Registers (CCR1, CCR2, CCR3)

- Controlan el *duty cycle*. Cada `CCR` actualiza la salida de su canal PWM cuando el contador (`CNT`) iguala el valor del `CCR`.
- En un esquema trifásico, se configuran `CCR1`, `CCR2`, `CCR3` para cada fase; y sus complementarios con *dead time*.

Nota: En este proyecto, se aprovecha `CH4` de `TIM1` como *canal de comparación* con `CCR4`=0, para generar interrupciones exclusivamente al inicio de la cuenta ascendente (CNT=0), evitando el `UPDATE`[2] (CNT=ARR).

2.2. Descripción de la Captura de Encoder

El encoder incremental provee dos señales (A y B), con flancos escalonados. El `STM32L476RG` decodifica dichas señales en modo `Encoder`:

[2]NOTE: En usos generales, se suele asumir que los callbacks son gatillados solo por este evento, es decir, al ocurrir un desbordamiento en el contador. Más, es relativamente simple configurar la rutina, para que los callbacks se gatillen por la comparación del contador con el valor que queramos, aplicación descrita en la configuración de periféricos.

2.2.0.1 Modos TI1, TI2, TI12

- **TI12** (cuadratura x4): aprovecha flancos *rising* y *falling* de ambos canales (A y B) para mayor resolución.
- El registro `CNT` se incrementa o decrementa según el sentido de giro (indicado por el orden de los flancos).

2.2.0.2 Referencia Manual

- Según el RM0351 (sección `Encoder Interface`), se configura `TIM2` para *Encoder Mode TI12*, polaridades `RISING`, `PSC`=0, `ARR` = 0xFFFFFFFF, etc.

2.3. Configuración de ADC con DMA (Lectura Continua, Disparo Externo TRGO)

El `STM32L476RG` posee un ADC de 12 bits capaz de dispararse por `TRGO` (p.ej., `TIM1_TRGO`) para sincronizar las conversiones con el ciclo PWM. Se configura `Oversampling` (128) y `DMA Continuous Requests` para reducir la carga del CPU. Note, el tiempo de conversión del ADC se determina por la siguiente:

$$T_{conv} = 128\,(T_{sample} + 12{,}5) \div f_{clkADC}$$

Se descarta el tiempo que toma el corrimiento de 7 bits. T_{sample} se determina en parameter settings, para el caso se elige 2.5 ciclos. Note que el ADC trabaja como máximo con la mitad del clock del integrado, tal que en el periodo de $192\mu s$ se realizan 4 conversiones (cada una con sobre muestreo como se ha visto). Cada una de estas conversiones se guardan en un arreglo (buffer), luego en el callback de la portadora se promedian, para calcular nuevos `CCR`.

2.3.0.1 Rango de Conversión y Referencia

- Típicamente, V_{REF} puede ser hasta 3.6 V (3.3 V recomendado).
- En modo `Single-Ended`, el ADC mapea 0–3.3 V a 0–4095 (12 bits).
- En modo `Diferencial`, se obtiene un offset en 2048 para 0 V.[3]

2.3.0.2 DMA vs Interrupciones

- **DMA** transfiere resultados directamente al `adc_buffer` sin requerir la CPU para cada muestra.
- Se puede procesar el buffer al final de la conversión (`HAL_ADC_ConvCpltCallback`), o en la interrupción del PWM (`TIM1_CH4`).

3. Desarrollo

3.1. Configuración de Periféricos en CubeIDE (v1.17.0)

Nota importante: Solo configure lo que corresponda, existen más opciones, déjelas como están. En la sección *Pinout & Configuration* del archivo `.ioc`, se configuran:

[3] Ver *RM0351*, Sección ADC, tabla 105 y figura 66 para detalles.

- `TIM1` en modo PWM trifásico (`CH1, CH2, CH3`) y su canal de comparación (CH4) para generar interrupciones al inicio de la cuenta ascendente.
- `TIM2` en modo Encoder (TI12) para la captura del encoder.
- `ADC1` configurado con `Oversampling+DMA` y disparo por `TRGO` proveniente de `TIM1`.

A continuación se detallan los pasos a seguir en CubeIDE (v1.17.0) utilizando el archivo `.ioc`:

Paso 1: Abrir el archivo .ioc: Inicia CubeIDE y abre el proyecto; luego abre el archivo .ioc contenido en tu proyecto, accede a la pestaña *Pinout & Configuration*.

Paso 2: Configurar los pines: Estos se seleccionan automáticamente al configurar dentro de la sección correspondiente a cada parámetro. También lo puede hacer manualmente, dependiendo de las funciones que tenga disponible cada GPIO.

Paso 3: Configurar TIM1:

- En Timers, seleccionar `TIM1` y en Mode, establecer el modo `PWM Generation CH1, CH2, CH3`. Cada uno con su complementario.
- Activa el canal `CH4` en modo de comparación para generar la interrupción al inicio del ciclo (CNT=0). `"Output Compare No Output"`.
- En Configuration, Paramter Settings, Counter Settings, seleccione `PSC = 0`, Counter Mode Center Aligned mode2, `AutoReloadRegister = 7679`.
- Configura el *Dead Time* Break and Dead Time management - `Output Configuration`, `Dead Time = 80`.
- En Output Compare No Output Channel 4 seleccione `Mode = Frozen`

Paso 4: Configurar TIM2:

- Selecciona `TIM2`, Mode, Comined Channels = `Encoder Mode`.
- Verifica que los canales tengan la polaridad correcta y que el contador (`CNT`) tenga el rango deseado (por ejemplo, ARR=0xFFFFFFFF).

Paso 5: Configurar ADC1:

- En ADC1, Mode, selecciona tus entradas `INx` en Single-ended, según tu aplicación.
- En Configuration, Parameter Settings, ADC_Settings, habilita Scan Conversion y Continuous Conversion. En ADC_Regular_ConversionMode habilita Conversions y Oversampling, seleciona Right Shift de 7 bits, selecciona Ratio de 128, selecciona Regular Conversion Oversampling Mode en modo continuo.
- Selecciona en Rank según el orden en que operen sus canales. En la aplicación realizada se seleccionó el Sampling Time mínimo de 2.5 ciclos.
- Importante: En External Trigger Conversion Source: `Timer 1 Trigger Out event`. Para sincronizar conversiones del ADC con la portadora.

Paso 6: Configurar NVIC: Esto es fundamental para la IRQn que desea implementar. En el caso, en TIM1, Configuration, NVIC Settings, habilite TIM1 capture compare interrupt. El nombre de la interrupción lo dice todo. Este paso es crítico, de lo contrario no levantará el callback que se desea.

A continuación se muestran los screenshots tomados desde la GUI de CubeIDE (archivo `.ioc`):

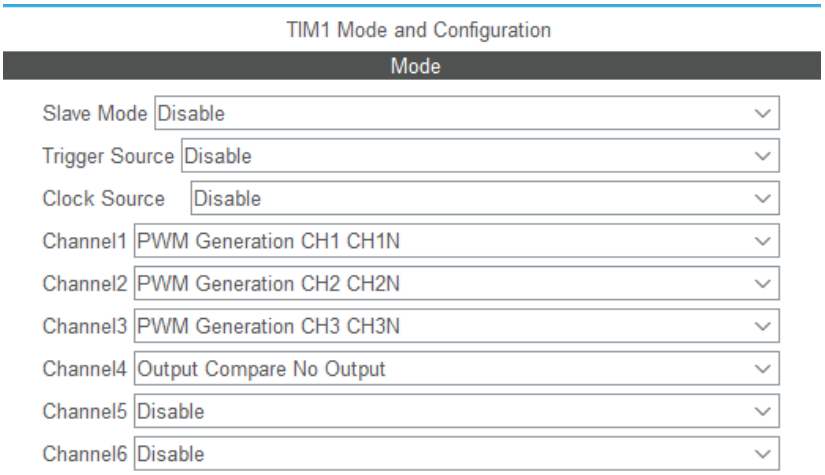

Figura 1: Configuración de `TIM1` en modo PWM trifásico (GUI del `.ioc`).

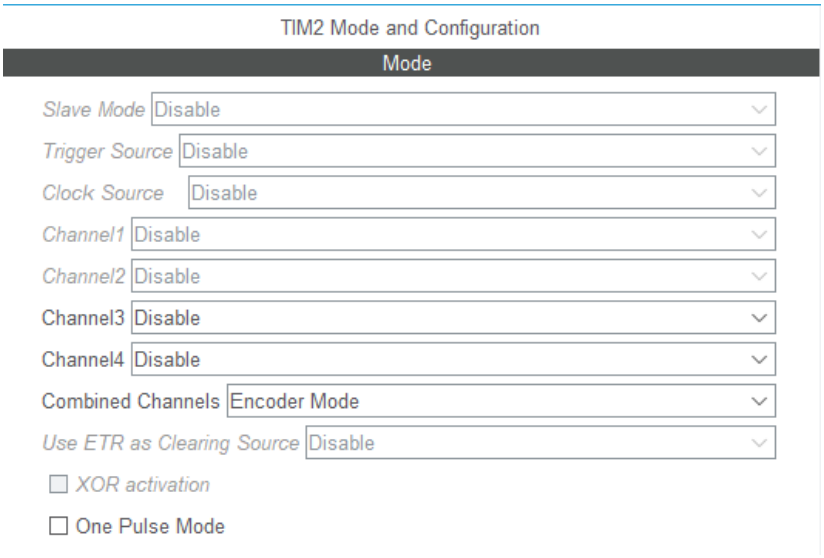

Figura 2: Configuración de `TIM2` en modo Encoder TI12 (GUI del `.ioc`).

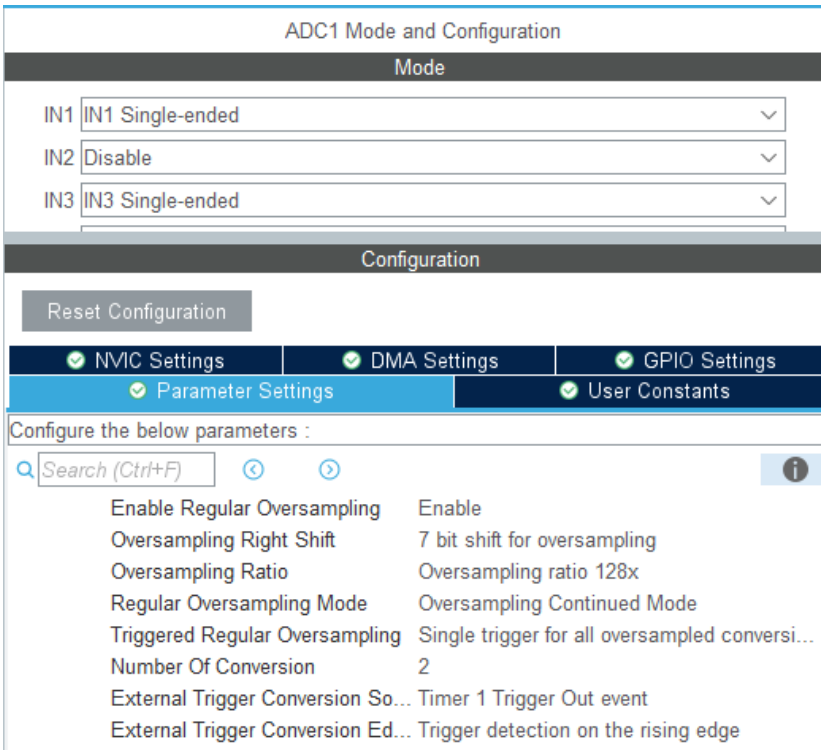

Figura 3: Configuración de `ADC1` con oversampling y DMA (GUI del `.ioc`).

Después de haber configurado correctamente los periféricos, el pinout debería verse como la Figura 4.

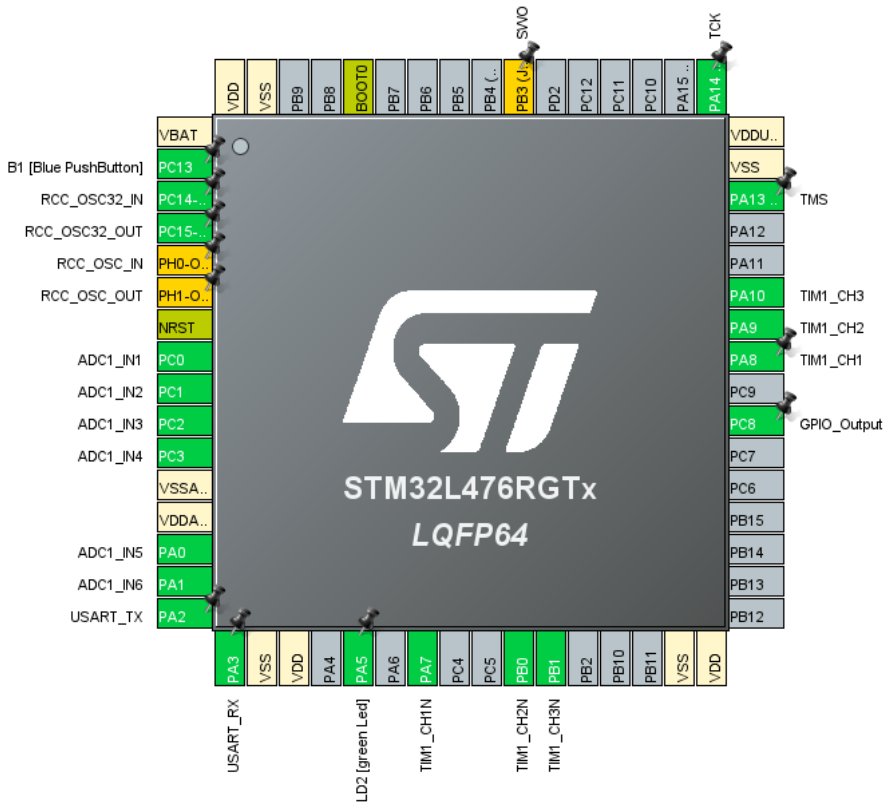

Figura 4: Pinout del sistema configurado en .ioc.

3.2. Implementación en el `main.c`

Tras guardar y generar código, `CubeIDE` crea funciones `MX_TIM1_Init`, `MX_TIM2_Init`, `MX_ADC1_Init`, etc. En `USER CODE BEGIN 2`, se agregan las llamadas para iniciar el PWM trifásico, el encoder, y el ADC en DMA, además de la interrupción `OC` en canal 4 de `TIM1`.

3.2.0.1 NVIC Initialization

Cada periférico con interrupción (p.ej., `TIM1_CC_IRQn`, `DMA1_Channel1_IRQn`, `DMA1_Channel7_IRQn`, `TIM2_IRQn`, etc.) requiere asignación de prioridad mediante `HAL_NVIC_SetPriority` y luego `HAL_NVIC_Enable`

3.2.0.2 Pasos finales (resumen)

- Iniciar cada canal PWM y su complementario (CH1N, CH2N, CH3N).
- Determinar prioridades de interrupciones `NVIC_SetPriority`.
- `__HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_4, 0)` para congelar canal4 en 0.
- Deshabilitar interrupciones de DMA no requeridas (o ajustar prioridad).
- Llamar `HAL_TIM_OC_Start_IT(&htim1, TIM_CHANNEL_4)` y `HAL_TIM_Base_Start(&htim1)`.
- `HAL_ADC_Start_DMA(&hadc1, (uint32_t*)adc_buffer, 8)` para iniciar la conversión ADC.

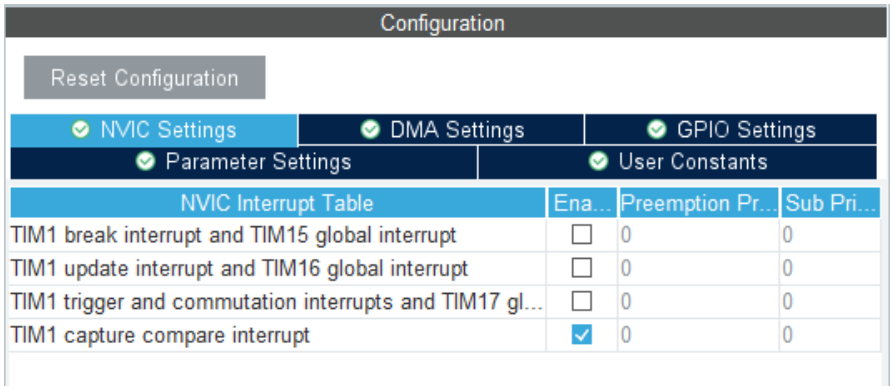

Figura 5: Habilitación de `TIM1 capture compare interrupt` en la GUI.

3.3. Descripción de Callbacks

```c
void HAL_TIM_OC_DelayElapsedCallback(TIM_HandleTypeDef *htim)
{
    if ((htim->Instance == TIM1) &&
        (htim->Channel == HAL_TIM_ACTIVE_CHANNEL_4))
    {
        // Ejemplo: l gica para leer ADC, calcular CCRs, etc.
    }
}
```

Script 2: Ejemplo de callback: Compare en TIM1 canal 4

```c
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef *hadc)
{
    if (hadc->Instance == ADC1) {
       // Manejar buffer ADC si se desea
    }
}
```

Script 3: Ejemplo de callback: Conversión completada en ADC

```c
void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart)
{
    if (huart->Instance == USART2) {
       // Indicar fin de la transmisi n
    }
}
```

Script 4: Ejemplo de callback: Transmisión DMA UART completada

4. Resultados y Discusión

Se validaron:

- **PWM Trifásico con Dead Time**: en la figura 6, se aprecia la separación entre la señal CHx y CHxN, previniendo cortocircuitos. Con un dead time prácticamente exacto de $1\mu s$, para BDTR igual a 80[4].
- **Encoder**: registró variaciones correctas de `CNT` al girar el eje manualmente, incrementando o decrementando según el sentido. Se recomienda encarecidamente hacer uso de la instancia de depuración mediante JTAG, seleccionando `Debug`, `Live Expressions` en la ventana de Watch.
- **ADC con Oversampling**: se verificó la adquisición estable en `adc_buffer` sin sobrecargar la CPU, en parte gracias a `DMA` y disparo por `TRGO`. Note en figura 7 la correcta reconstrucción de una señal senoidal de $50Hz$[5]. Esta obtiene una fase debido al tiempo que toma al STM32 digitalizar la entrada, calcular y generar salidas complementarias, previendo cortes, logrando una respuesta aceptable.
- En último lugar, al observar la Figura 8, podemos verificar que la configuración del periodo de la portadora corresponde a $192\mu s$, tal como se describe en la sección 2.

[4]Compruebe con Script 1

[5]Ripple propio de la carga capacitiva utilizada para reconstruir, posterior a amplificación mediante INA121P. Se hizo uso de un capacitor cerámico de $100nF$ para aislar el ruido producido por el generador de señales

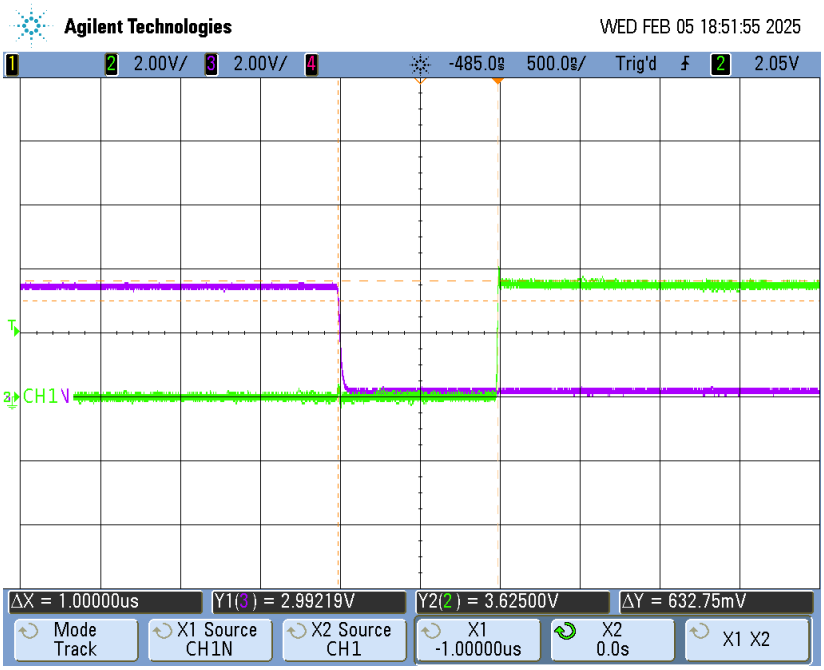

Figura 6: Señal CH1 vs CH1N con dead time configurado en BDTR.

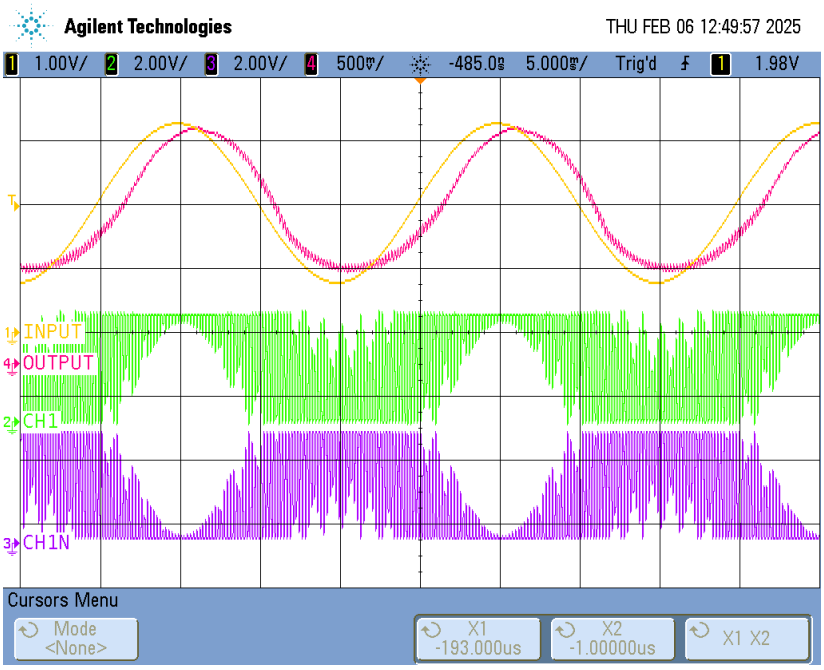

Figura 7: Reconstrucción de `INPUT`(verde). Se observa `TIM1_CH1` y `TIM1_CH1N` variando `CCR` (duty cycle). Naturalmente el duty cycle varía constantemente. Señal obtenida `OUTPUT`(rosa) con ripple propio de la carga.

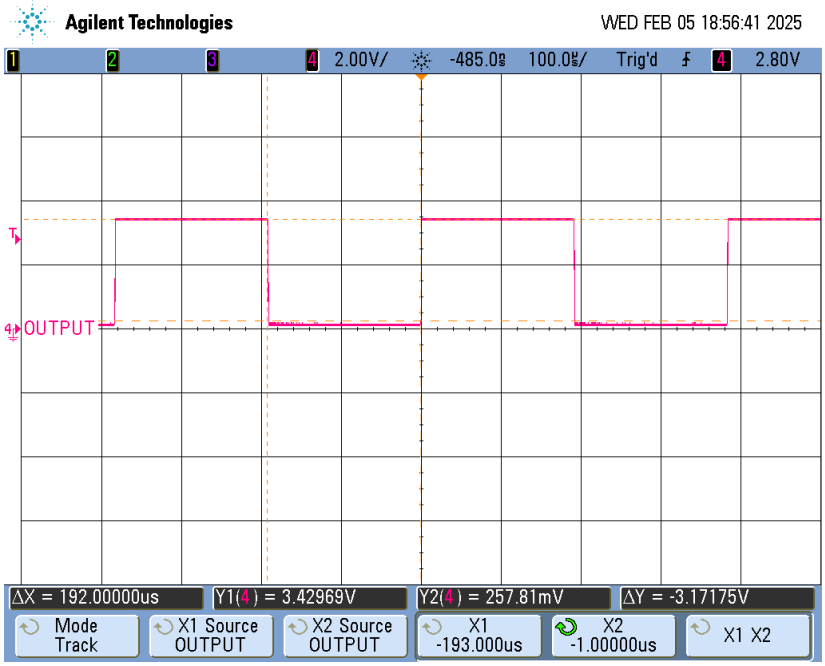

Figura 8: Señal en el pin PC8, negado al comienzo de cada callback producido por la equivalencia del contador de TIM1 con el valor constante del canal 4 de este mismo, igual a 0. [6]

5. Conclusiones

- Se estableció un **control PWM trifásico** sincronizado con la lectura ADC y la captura de encoder, todo en un `STM32L476RG`.
- El disparo externo (`TIM1_TRGO`) demostró efectividad para muestrear el ADC en tiempo real.
- La lectura del encoder (`TIM2` en modo TI12) permitió estimar la posición y/o velocidad del eje con alta resolución.
- Este esquema sienta las bases para un **control a lazo cerrado** sofisticado (p.ej., FOC o control vectorial), ampliando las posibilidades de la plataforma.

Con esta configuración se abordan los aspectos relevantes en un `STM32L476RG`: el uso de `TIM1` para generar PWM en modo trifásico con el canal 4 en compare (`CNT=0`), el ADC en modo *Oversampling + DMA* disparado por `TRGO`, y la lectura de un `Encoder` (TIM2).

La tecnología STM32 es cómoda para el usuario, la instancia .ioc ahorra dolores de cabeza y explicita la configuración de los periféricos. De esta forma, el usuario puede concentrarse en lo que le compete.

Existe vasta información, guías y discusiones en las cuales probablemente se hayan tratado soluciones varias. No tiene mucho sentido inventar la rueda, mas sí entender cómo funciona, y en base a eso lograr soluciones más eficaces y precisas.

El STM32L476RG en particular es un integrado sólido. Existen más equipos para aplicaciones más especializadas, que cuentan con incluso más timers, conversores y relojes más rápidos. Otro punto fuerte de esta tarjeta es la vasta cantidad de pines, maximizando el uso de sus periféricos, a diferencia de tecnologías como ESP32 DevKit, en las cuales se debe elegir qué herramientas descartar, ya que muchas veces comparten canales.

A continuación se expresa como anexo el resultado de implementación de la aplicación descrita en la presente. Para más consultas no dude en contactar al autor ó buscar en las referencias dispuestas por STMicroelectronics.

Lista1 : Anexo de scripts

1. **Archivo main.c**

```
/* main.c - Descripción breve del objetivo de este archivo */

int main(void)
{
  /* USER CODE BEGIN 1 */
  /* Inicializaciones tempranas del usuario */
  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------*/
  HAL_Init();
  /* USER CODE BEGIN Init */
  /* USER CODE END Init */

  /* Configure el sistema de reloj */
  SystemClock_Config();
  /* USER CODE BEGIN SysInit */
  /* USER CODE END SysInit */

  /* Inicializar periféricos configurados en .ioc */
  MX_GPIO_Init();
  MX_DMA_Init();
  MX_USART2_UART_Init();
  MX_TIM1_Init();
  MX_ADC1_Init();
  MX_TIM2_Init();

  /* USER CODE BEGIN 2 */

  // Iniciar TIM2 en modo Encoder
  HAL_TIM_Encoder_Start(&htim2, TIM_CHANNEL_ALL);

  // Pin PC8 a RESET
  HAL_GPIO_WritePin(GPIOC, GPIO_PIN_8, GPIO_PIN_RESET);

  // Iniciar PWM trifásico en TIM1 (CH1, CH2, CH3)
  HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_1);
  HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_2);
  HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_3);

  // Iniciar canales complementarios
  HAL_TIMEx_PWMN_Start(&htim1, TIM_CHANNEL_1);
  HAL_TIMEx_PWMN_Start(&htim1, TIM_CHANNEL_2);
  HAL_TIMEx_PWMN_Start(&htim1, TIM_CHANNEL_3);

  // Calibrar ADC y setear prioridad NVIC
```

```
  HAL_ADCEx_Calibration_Start(&hadc1, ADC_SINGLE_ENDED);
  HAL_NVIC_SetPriority(TIM1_CC_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(TIM1_CC_IRQn);

  // Congelar canal4 en cero
  __HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_4, 0);

  // Configurar y habilitar DMA1 Channel1
  HAL_NVIC_SetPriority(DMA1_Channel1_IRQn, 1, 0);
  HAL_NVIC_EnableIRQ(DMA1_Channel1_IRQn);

  // Configurar y habilitar DMA1 Channel7 (TX en UART2), p.ej
  HAL_NVIC_SetPriority(DMA1_Channel7_IRQn, 3, 0);
  HAL_NVIC_EnableIRQ(DMA1_Channel7_IRQn);

  // Iniciar interrupción canal4 (compare) y base
  HAL_TIM_OC_Start_IT(&htim1, TIM_CHANNEL_4);
  HAL_TIM_Base_Start(&htim1);

  // Iniciar ADC con DMA
  HAL_ADC_Start_DMA(&hadc1, (uint32_t*)adc_buffer, 8);

  // Iniciar encoder con interrupciones
  HAL_TIM_Encoder_Start_IT(&htim2, TIM_CHANNEL_ALL);

  HAL_Delay(1);
  ARR = __HAL_TIM_GET_AUTORELOAD(&htim1);
  HAL_Delay(1);

  /* USER CODE END 2 */

  /* Bucle principal */
  while (1)
  {
     if (do_uart) {
       do_uart = false;
       // Enviar la matriz en binario vía UART2 DMA
       HAL_UART_Transmit_DMA(&huart2, (uint8_t*)dataMatrix, sizeof(dataMatrix));
     }
  }
}
```

2. **Archivo callbacks.c**

```
/* callbacks.c - Rutinas de interrupción */

#include "header.h"
```

```
volatile uint16_t ccr1, ccr2, ccr3;
uint32_t sumA=0, sumB=0;
bool do_uart=false;
uint64_t interrupt_index=0;

float vA, vB, vC;
uint32_t avgA, avgB, avgC;
uint16_t j=0;
int16_t position=0;
float revolutions=0;

void HAL_TIM_OC_DelayElapsedCallback(TIM_HandleTypeDef *htim)
{
    if ((htim->Instance == TIM1) &&
        (htim->Channel == HAL_TIM_ACTIVE_CHANNEL_4))
    {
        HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_8);
        interrupt_index++;

        sumA = adc_buffer[0] + adc_buffer[2] + adc_buffer[4] + adc_buffer[6];
        sumB = adc_buffer[1] + adc_buffer[3] + adc_buffer[5] + adc_buffer[7];
        avgA = sumA >> 2;
        avgB = sumB >> 2;
        avgC = 6144 - (avgA + avgB);

        ccr1 = (uint16_t)((avgA * ARR) >> 12);
        ccr2 = (uint16_t)((avgB * ARR) >> 12);
        ccr3 = (uint16_t)((avgC * ARR) >> 12);

        // Clamping
        ccr1 = (ccr1 > ARR) ? ARR : ccr1;
        ccr2 = (ccr2 > ARR) ? ARR : ccr2;
        ccr3 = (ccr3 > ARR) ? ARR : ccr3;

        __HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_1, ccr1);
        __HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_2, ccr2);
        __HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_3, ccr3);

        j = interrupt_index % N_Samples;
        do_uart = (j == 0) ? true : false;
        interrupt_index = (interrupt_index >= UINT64_MAX - 1) ? 0 : interrupt_index

        dataMatrix[j].A = avgA;
        dataMatrix[j].B = avgB;
        dataMatrix[j].C = avgC;
        dataMatrix[j].index = interrupt_index;

        HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_8);
```

```
            position = __HAL_TIM_GET_COUNTER(&htim2);
            revolutions = (float)position / (float)CPR;
        }
    }

    // Ejemplo de callback ADC (si se desea)
    void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef *hadc)
    {
        if (hadc->Instance == ADC1) {
           // Lógica si se desea procesar buffer inmediatamente
        }
    }

    // Ejemplo de callback UART TX
    void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart)
    {
        if (huart->Instance == USART2) {
           // Indicar fin de la transmisión
        }
    }
```