Taller de Sistemas Embebidos
STM32 MCU – Pulse Width Modulation (PWM)

Información relevante

Taller de Sistemas Embebidos

Asignatura correspondiente a la **actualización 2023** del Plan de Estudios 2020 y resoluciones modificatorias, de Ingeniería Electrónica de FIUBA

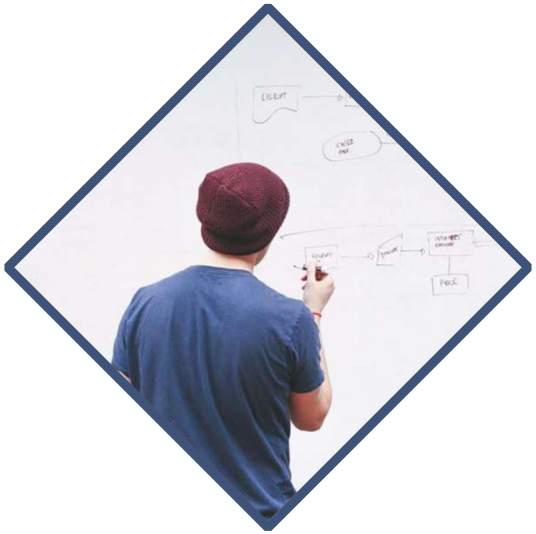
Estructura Curricular de la Carrera

El **Proyecto Intermedio** se desarrolla en la asignatura **Taller de Sistemas Embebidos**, la cual tiene un enfoque centrado en la **práctica propia de la carrera** más que en el desarrollo teórico disciplinar, con eje en la **participación de las y los estudiantes**

Más información . . .

. . . sobre la **actualización 2023** . . . https://www.fi.uba.ar/grado/carreras/ingenieria-electronica/plan-de-estudios

. . . sobre el **Taller de Sistemas Embebidos** . . . https://campusgrado.fi.uba.ar/course/view.php?id=1217



¡Hola!

Soy Juan Manuel Cruz
Taller de Sistemas Embebidos
Consultas a: jcruz@fi.uba.ar

1

Introducción

Actualización 2023 del Plan de Estudios 2020 y resoluciones . . .

Conceptos básicos

- Referencia:
  - Programming whith STM32, Getting Started with the Nucleo Board and C/C++ - Donal Norris (Author)
  - Chapter 10: Pulse Width Modulation (PWM)
    - Este capítulo trata sobre la modulación de ancho de pulso (PWM), que es una señal digital que una MCU genera fácilmente para diversos propósitos. Estos propósitos comúnmente incluyen controlar lo siguiente:
      - La posición de un servomotor estándar.
      - La velocidad de rotación de un servo de rotación continua.
      - La luminiscencia de un emisor de luz.

Conceptos básicos

- La velocidad de rotación de un motor eléctrico estándar con circuito de controlador externo
- Un cargador de baterías solares fotovoltaicas
- El maximum power-point track
- Una forma de señal de salida como sinusoidal, triangular o cuadrada.
- Un generador de sonido acústico

- Los proyectos de este capítulo demostrarán cómo controlar la luminiscencia (intensidad de la luz) de un LED normal, el color y la luminiscencia de un LED tricolor y el posicionamiento de un servomotor estándar.

Conceptos básicos

- Es aconsejable discutir primero la naturaleza y las propiedades de una señal PWM antes de proceder a mostrar cómo generarla usando el tablero de proyecto STM. La Figura 10-1 es una señal PWM típica con varias propiedades anotadas en la figura.
- Lo primero que hay que tener en cuenta es que la señal es repetitiva con una frecuencia en torno a los 50 Hz o un periodo equivalente de 20 ms. Esta frecuencia es muy común en servocontrol y será la que usaré para los proyectos del capítulo.
- Sin embargo, no hay nada limitante en esta frecuencia y puede usar cualquier valor hasta el límite de lo que la MCU puede soportar.

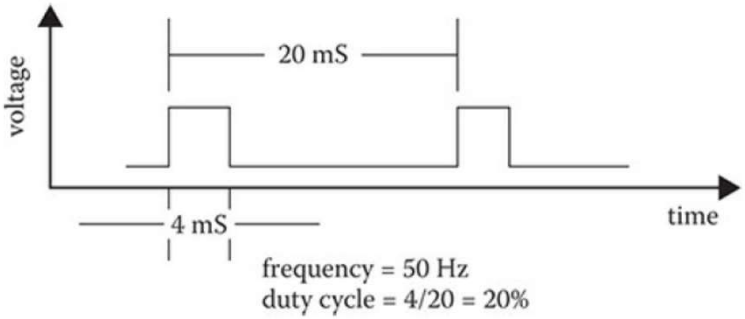


Figure 10-1 *Typical PWM signal.*

Conceptos básicos

- La segunda característica importante es el ciclo de actividad (duty cycle), que es un porcentaje de la cantidad de tiempo que la señal está en estado alto respecto del período total, que es la suma de los tiempos alto y bajo. El ciclo de actividad a menudo se considera la propiedad principal de una señal PWM, ya que determina únicamente la posición del servo y la luminiscencia del LED.
- Resulta que cada tipo de dispositivo controlado por PWM actúa de manera diferente con la señal. Describiré cada interacción específica cuando vea la demostración del dispositivo.

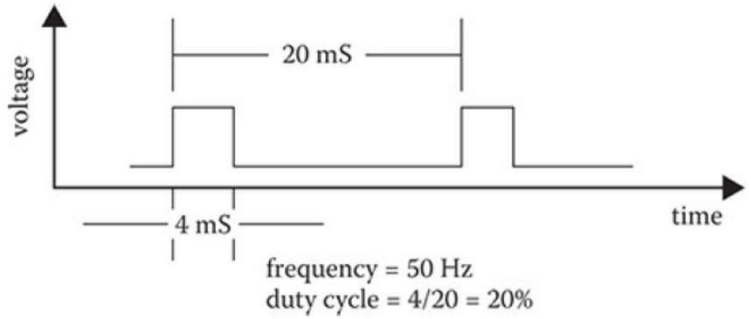
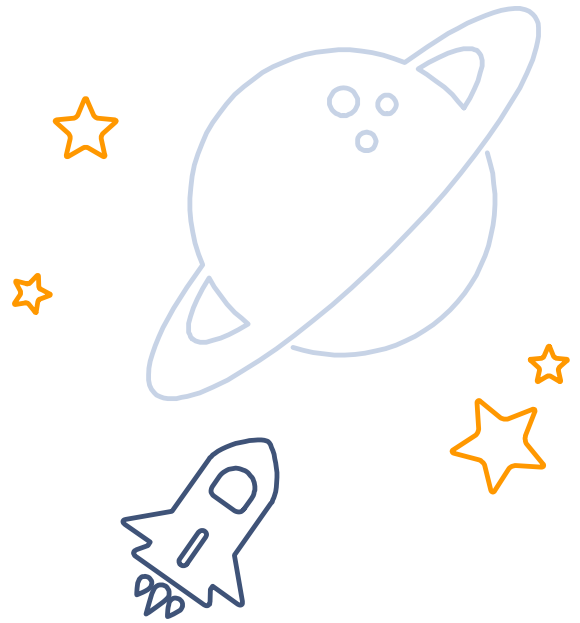


Figure 10-1 *Typical PWM signal.*

Solución Adecuada



. . . lo más simple posible, previa determinación del objetivo de excelencia a cumplir, obviamente contando con la documentación debida y recurriendo a la metodología de trabajo adecuada

2

Documentación debida

1er Cuatrimestre de 2024, dictado por primera vez . . .

“

Programming whith STM32, Getting Started with the Nucleo Board and C/C++ - Donal Norris (Author)

Chapter 10: Pulse Width Modulation (PWM)

General-Purpose Timer PWM Signal Generation

- Las señales PWM se generan únicamente mediante el uso de un temporizador avanzado o de propósito general (GP). Esta sección describe el proceso de cómo se hace esto.
- La generación de señal es relativamente simple e implica el uso de un temporizador en modo de conteo ascendente. La parte superior de la Figura 10-2 muestra un temporizador que cuenta desde 0 hasta un recuento máximo de 16 bits igual a 65535.
- Cada temporizador GP tiene una serie de registros conocidos como registros de captura de comparación (CCRx), que almacenan un número.

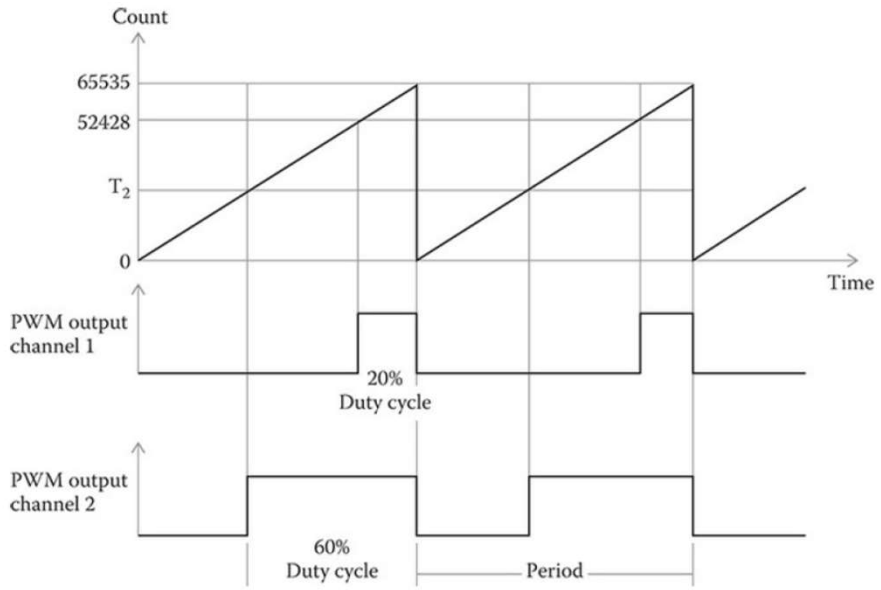


Figure 10-2 *PWM signal generation.*

General-Purpose Timer PWM Signal Generation

- Cuando el conteo alcanza ese número, se genera un evento que depende del modo del temporizador.
  - Si el temporizador está configurado para una salida directa como es el caso de este modo PWM, entonces el evento será encender el canal asociado al registro CCRx.
- En la Figura 10-2, el canal 1 cambia a nivel alto cuando el conteo llega a 52428.
  - El canal permanece alto hasta que el contador del temporizador alcanza el conteo máximo (MAX) y se desborda o se reinicia como se muestra en la forma de señal superior.

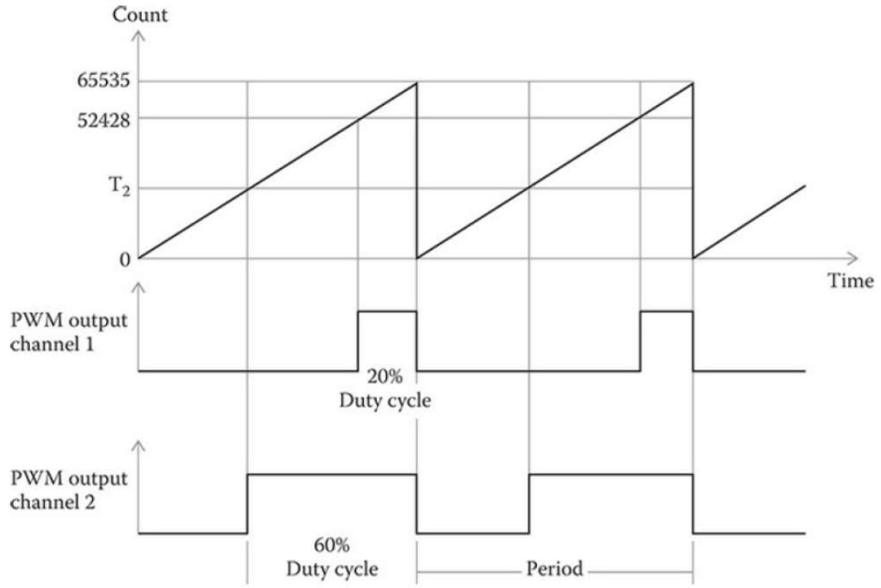


Figure 10-2 *PWM signal generation.*

General-Purpose Timer PWM Signal Generation

- El resultado neto de esta acción es que se emite un pulso repetitivo desde la línea del canal 1 con un ciclo de actividad del 20%.
- La siguiente ecuación relaciona el conteo del temporizador con el ciclo de actividad (DC):

$$DC = \frac{MAX - CCR}{MAX}$$

- Esta ecuación se puede reorganizar para encontrar el valor de CCRx dado un ciclo de actividad deseado:

$$CCR = MAX * (1 - DC)$$

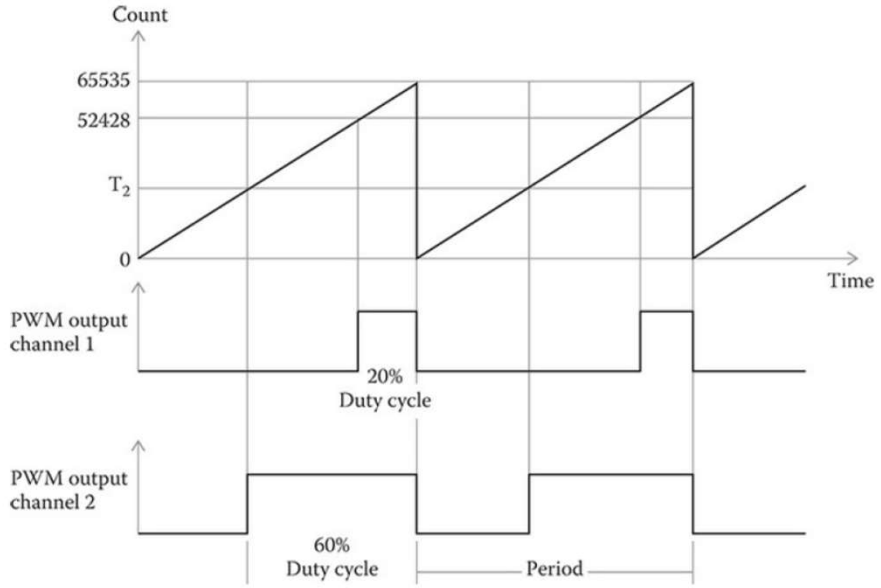


Figure 10-2 *PWM signal generation.*

General-Purpose Timer PWM Signal Generation

- En la Figura 10-2, también mostré una salida de canal adicional con un ciclo de actividad deseado del 60%, pero no se muestra un recuento específico para el registro CCRx.
- Usaré la segunda forma de la ecuación para resolver T2, que es el valor de conteo que se almacenará en el registro CCR2:

$$CCR2 = 65535 * (1 - 0.6) = 26214$$

- Por lo tanto, almacenar un valor de 26214 en CCR2 provocará que se genere un tren de pulsos de ciclo de actividad del 60% desde la salida del canal 2 de TIM2.

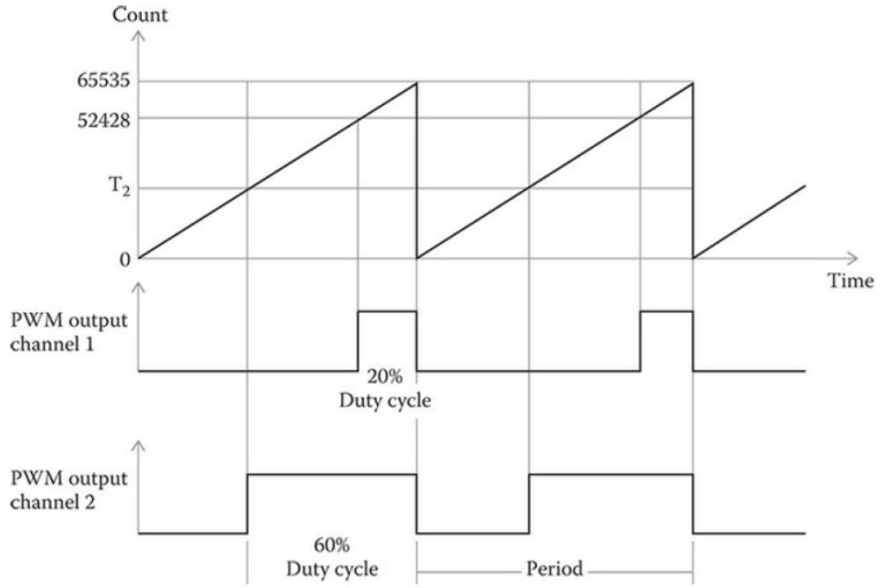


Figure 10-2 *PWM signal generation.*

General-Purpose Timer PWM Signal Generation

- Tenga en cuenta que los números que utilicé para este cálculo son relativos en el sentido de que los números CCRx reales almacenados dependen de la velocidad del reloj del temporizador real. En breve verás que los números son mucho más pequeños, pero el principio para calcularlos es el mismo.
- El temporizador TIM2 GP que utilicé para la generación de PWM tiene un máximo de cuatro canales de salida, lo que significa que es posible generar simultáneamente cuatro señales PWM sincronizadas utilizando un único temporizador GP. Una de las siguientes demostraciones utiliza tres canales, lo cual es suficiente para demostrar esta capacidad.

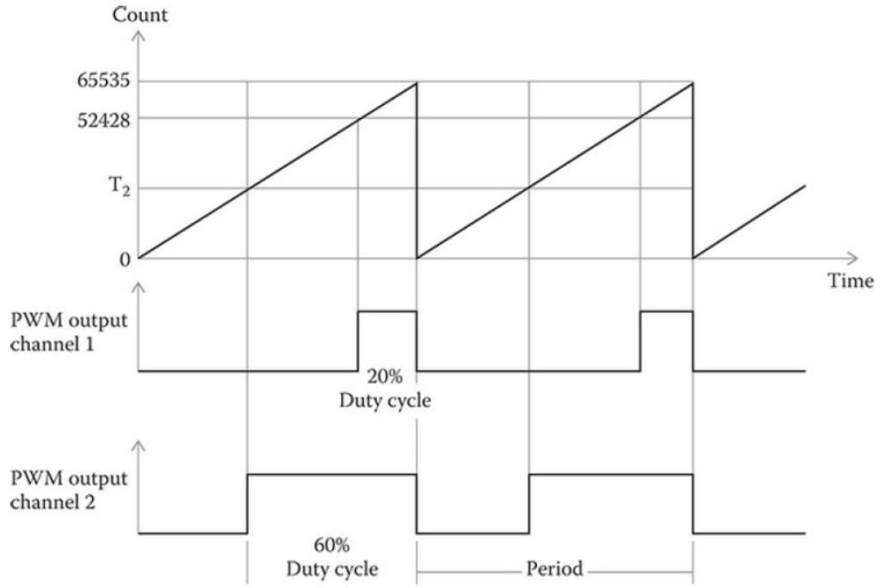


Figure 10-2 *PWM signal generation.*

Timer Hardware Architecture

- La Figura 10-3 es una parte del diagrama de bloques del temporizador GP que resalta las salidas multicanal y los registros CCRx asociados.

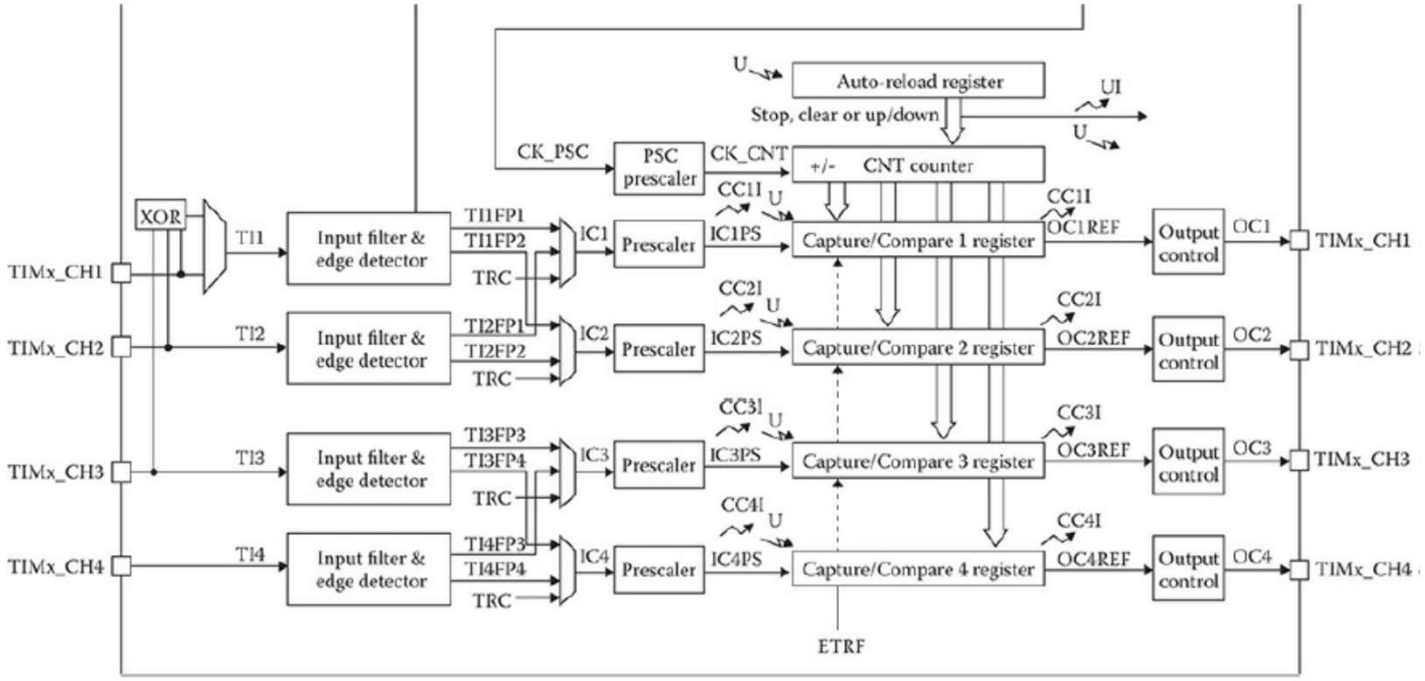


Figure 10-3 *GP timer block diagram with four output channels.*

Timer Hardware Architecture

- La arquitectura es muy sencilla y muestra que cada registro CCRx tiene una entrada del registro del contador de temporizador común (CNT), así como su propia entrada de frecuencia de reloj preescalada. Esta disposición permite implementar una salida de tren de pulsos PWM muy flexible.
- Ahora es el momento de describir cómo el software HAL permite la generación de señales PWM.

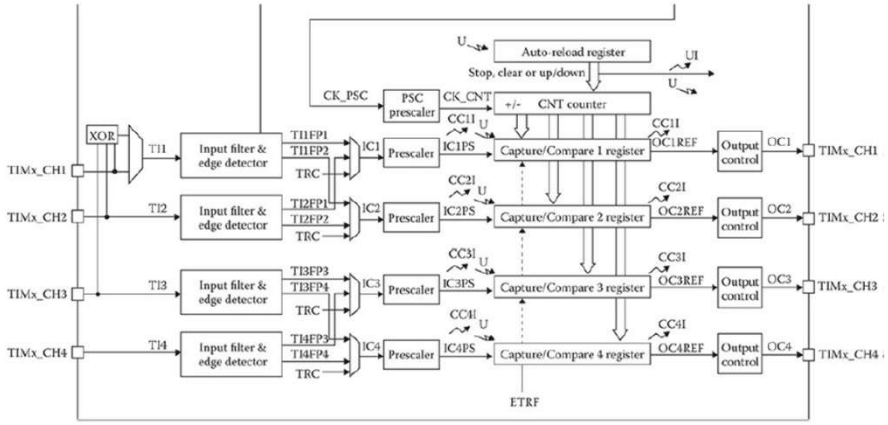


Figure 10-3 *GP timer block diagram with four output channels.*

PWM Signals with HAL

- La HAL utiliza una estructura C llamada TIM_MasterConfigTypeDef para configurar el temporizador periférico que genera una señal PWM. Cada miembro se analizará por separado después de la definición de la estructura.

```
typedef struct {
uint32_t MasterOutputTrigger; /* Trigger output (TRGO) selection */
uint32_t MasterSlaveMode; /* Master/slave mode selection */
} TIM_MasterConfigTypeDef;
```

- Los miembros de la estructura TIM_MasterConfigTypeDef se definen brevemente en la siguiente lista:
  - uint32_t MasterOutputTrigger; Este valor especifica el comportamiento de la salida del disparador (TRGO). Este parámetro puede ser cualquier valor de las siguientes definiciones:

PWM Signals with HAL

- uint32_t MasterSlaveMode; Se utiliza para habilitar/deshabilitar el modo maestro/esclavo de un temporizador. Este parámetro puede ser uno de los dos valores de las siguientes definiciones:

```
TIM_MASTERSLAVEMODE_ENABLE
TIM_MASTERSLAVEMODE_DISABLE
```

- Un temporizador PWM configurado mediante la estructura TIM_MasterConfigTypeDef debe inicializarse además mediante la estructura TIM_OC_InitTypeDef C, que se define de la siguiente manera:

```
typedef struct {
uint32_t OCMode;
uint32_t Pulse;
uint32_t OCPolarity;
uint32_t OCNPolarity;
uint32_t OCFastMode;
uint32_t OCIdleState;
uint32_t OCNIdleState;
} TIM_OC_InitTypeDef;
```

PWM Signals with HAL

- Los miembros de la estructura TIM_OC_InitTypeDef se definen brevemente en la siguiente lista:
  - uint32_t OCMode; Especifica el modo de comparación de salida. Este parámetro puede ser cualquier valor de las siguientes definiciones:
  - TIM_OCMODE_TIMING; La comparación entre el registro de comparación de salida (CCRx) y el contador del temporizador (CNT) no tiene ningún efecto en la salida, lo que se conoce como modo congelado.
  - TIM_OCMODE_ACTIVE: configura la salida del canal al nivel activo en la coincidencia de CCRx.
  - TIM_OCMODE_INACTIVE: establece la salida del canal en nivel inactivo en la coincidencia de CCRx.

PWM Signals with HAL

- TIM_OCMODE_TOGGLE: alterna la salida del canal cuando el contador del temporizador (CNT) coincide con el CCRx.

```
TIM_OCMODE_PWM1—PWM Mode 1
TIM_OCMODE_PWM2—PWM Mode 2
```

- TIM_OCMODE_FORCED_ACTIVE: fuerza la salida del canal a nivel alto independientemente del valor del contador del temporizador (CNT).
- TIM_OCMODE_FORCED_INACTIVE: fuerza la salida del canal a nivel bajo independientemente del valor del contador del temporizador (CNT).
- NOTA Modo PWM 1: Cuando el temporizador está contando, el canal estará activo mientras el Período sea menor que el Pulso; de lo contrario, estará inactivo. En un modo de cuenta regresiva, el canal está inactivo mientras el Período sea mayor que el Pulso; de lo contrario, está activo.

PWM Signals with HAL

- NOTA Modo PWM 2: Cuando el temporizador está contando, el canal estará inactivo siempre que el Período sea menor que el Pulso; de lo contrario, estará activo. En modo de conteo regresivo, el canal está activo mientras el Período sea mayor que el Pulso, de lo contrario está inactivo.
- uint32_t Pulse; Este valor se almacena dentro del registro CCRx y establece cuándo se activa la salida.
- uint32_t OCPolarity; Define la polaridad del canal de salida cuando los registros CCRx coinciden con el CNT. Este parámetro puede ser uno de los dos valores de las siguientes definiciones:

```
TIM_OCPOLARITY_HIGH
TIM_OCPOLARITY_LOW
```

PWM Signals with HAL

- uint32_t OCNPolarity; Define la polaridad de salida complementaria. Es un modo disponible sólo en los temporizadores avanzados TIM1 y TIM8, que permiten que dos canales dedicados adicionales generen señales complementarias, es decir, cuando el canal 1 está ALTO entonces el canal 1N está BAJO y viceversa. Esta característica es especialmente útil para aplicaciones de control de motores. Este parámetro puede ser uno de los dos valores de las siguientes definiciones:

```
TIM_OCNPOLARITY_HIGH
TIM_OCNPOLARITY_LOW
```

- uint32_t OCFastMode; Especifica el estado del modo rápido. Este parámetro es válido sólo en modo PWM1 y PWM2. Este parámetro puede ser uno de los dos valores de las siguientes definiciones:

```
TIM_OCFAST_DISABLE
TIM_OCFAST_ENABLE
```

PWM Signals with HAL

- uint32_t OCFastMode; Especifica el estado del modo rápido. Este parámetro es válido sólo en modo PWM1 y PWM2. Este parámetro puede ser uno de los dos valores de las siguientes definiciones: `TIM_OCFAST_DISABLE` `TIM_OCFAST_ENABLE`
- uint32_t OCIdleState; Especifica el estado del pin de comparación de salida del canal durante el estado inactivo del temporizador. Este parámetro puede ser uno de los dos valores de las siguientes definiciones: `TIM_OCIDLESTATE_SET` `TIM_OCIDLESTATE_RESET`
- NOTA Este parámetro solo está disponible en los temporizadores avanzados TIM1 y TIM8.
- uint32_t OCNIdleState; Especifica el estado del pin de comparación de salida del canal complementario durante el estado inactivo del temporizador. Este parámetro puede ser uno de los dos valores de las siguientes definiciones: `TIM_OCNIDLESTATE_SET` `TIM_OCNIDLESTATE_RESET`
- NOTA Este parámetro solo está disponible en los temporizadores avanzados TIM1 y TIM8.

PWM Signals with HAL

Enabling the PWM Function

- Ahora necesita habilitar la función alternativa PWM para tener la función de modulación de ancho de pulso. El temporizador TIM2 se configura inicialmente al crear el nuevo proyecto y mostrarse la ventana Pinout.
- La Figura 10-4 muestra cómo el periférico TIM2 con funcionalidad PWM se habilita en la vista Pinout seleccionando la salida directa PWM para el canal 1 en TIM2. El pin PA1 es la salida del canal 1.

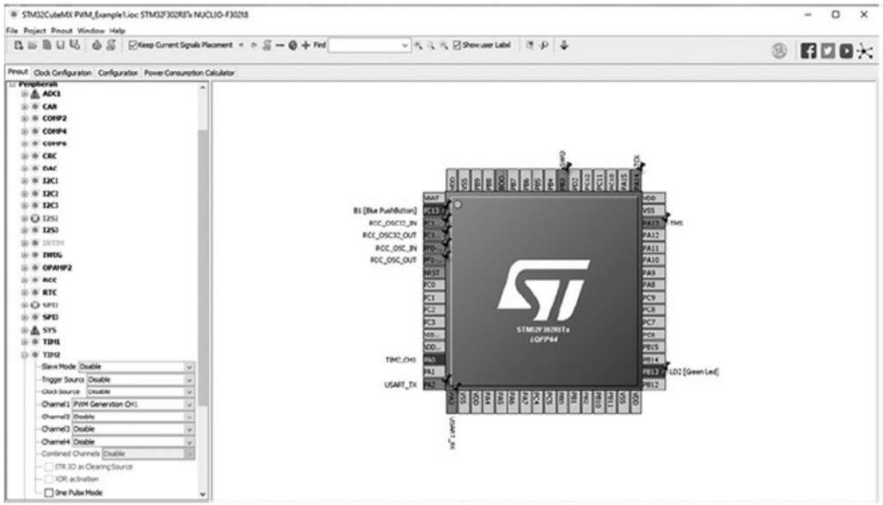


Figure 10-4 *Enabling timer TIM2 with PWM output on channel 1.*

PWM Signals with HAL

- La aplicación CubeMX insertará automáticamente un método de inicialización estático void MX_TIM2_Init(void) en el archivo main.c cuando la función TIM2 PWM esté habilitada.
- Algunos parámetros adicionales también se deben configurar en el objeto TIM2 usando la pestaña Configuración de CubeMX, como se muestra en la Figura 10-5.

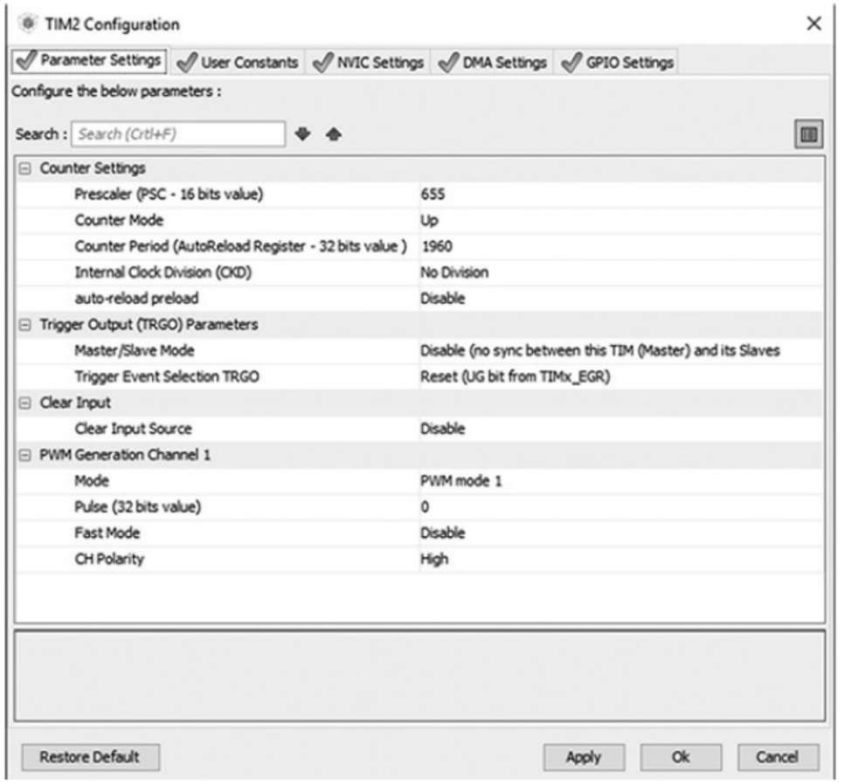


Figure 10-5 *Configuration menu settings.*

PWM Signals with HAL

- Todos los parámetros iniciales de PWM se configuran automáticamente en el código C según los parámetros que configuró en la vista de árbol IP de TIM2 PWM y el menú de configuración, como se muestra en el siguiente fragmento de código:

```
static void MX_TIM2_Init(void)
{
  TIM_MasterConfigTypeDef sMasterConfig;
  TIM_OC_InitTypeDef sConfigOC;

  htim2.Instance = TIM2;
  htim2.Init.Prescaler = 655;
  htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim2.Init.Period = 1960;
  htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_PWM_Init(&htim2) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig) !=
HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

  sConfigOC.OCMode = TIM_OCMODE_PWM1;
  sConfigOC.Pulse = 0;
  sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
  sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
  if (HAL_TIM_PWM_ConfigChannel(&htim2, &sConfigOC, TIM_CHANNEL_1) !=
HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

  HAL_TIM_MspPostInit(&htim2);

}
```

PWM Demonstration Software

- El archivo main.c recién generado ahora tiene un método que configura e inicializa el periférico interno TIM2 con una salida PWM en su canal 1.
  - No hay ningún código funcional ejecutándose dentro del bucle eterno. La generación de la señal PWM se realiza automáticamente en segundo plano. Este código se muestra en la siguiente lista de códigos:

```
// STM disclaimer goes here
// Includes
#include "main.h"
#include "stm32f3xx_hal.h"

/* USER CODE BEGIN Includes */

/* USER CODE END Includes */

// Private variables
TIM_HandleTypeDef htim2;

/* USER CODE BEGIN PV */
// Private variables

/* USER CODE END PV */

// Private function prototypes
void SystemClock_Config(void);
```

```
static void MX_GPIO_Init(void);
static void MX_TIM2_Init(void);
void HAL_TIM_MspPostInit(TIM_HandleTypeDef *htim);


/* USER CODE BEGIN PFP */
// Private function prototypes

/* USER CODE END PFP */

/* USER CODE BEGIN 0 */

/* USER CODE END 0 */

int main(void)
{

  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  // MCU Configuration

  // Reset all peripherals, initialize the flash and Systick
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_TIM2_Init();

  /* USER CODE BEGIN 2 */
  HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);

       TIM2->CCR1 = 980; // CCR1 register set for 50% duty cycle
  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
```

ware

```
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */

    }
    /* USER CODE END 3 */
}

// System Clock Configuration
void SystemClock_Config(void)
{

  RCC_OscInitTypeDef RCC_OscInitStruct;
  RCC_ClkInitTypeDef RCC_ClkInitStruct;

  // Initializes the CPU, AHB and APB buss clocks
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
  RCC_OscInitStruct.HSIState = RCC_HSI_ON;
  RCC_OscInitStruct.HSICalibrationValue = 16;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
  RCC_OscInitStruct.PLL.PLLMUL = RCC_PLL_MUL16;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

  // Initializes the CPU, AHB and APB buss clocks
  RCC_ClkInitStruct.ClockType =
  RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
  |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV2;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_2) !=
HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

  // Configure the Systick interrupt time
  HAL_SYSTICK_Config(HAL_RCC_GetHCLKFreq()/1000);

  // Configure the Systick
  HAL_SYSTICK_CLKSourceConfig(SYSTICK_CLKSOURCE_HCLK);

  /* SysTick_IRQn interrupt configuration */
  HAL_NVIC_SetPriority(SysTick_IRQn, 0, 0);
```

```
}

/* TIM2 init function */
static void MX_TIM2_Init(void)
{

  TIM_MasterConfigTypeDef sMasterConfig;
  TIM_OC_InitTypeDef sConfigOC;

  htim2.Instance = TIM2;
  htim2.Init.Prescaler = 655;
  htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim2.Init.Period = 1960;
  htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_PWM_Init(&htim2) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig) !=
HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

  sConfigOC.OCMode = TIM_OCMODE_PWM1;
  sConfigOC.Pulse = 0;
  sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
  sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
  if (HAL_TIM_PWM_ConfigChannel(&htim2, &sConfigOC, TIM_CHANNEL_1) !=
HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

  HAL_TIM_MspPostInit(&htim2);

}

/** Configure pins as
        * Analog
        * Input
        * Output
        * EVENT_OUT
        * EXTI
     PA2   ------> USART2_TX
     PA3   ------> USART2_RX
*/
```

```
static void MX_GPIO_Init(void)
{

  GPIO_InitTypeDef GPIO_InitStruct;

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_GPIOF_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();
  __HAL_RCC_GPIOB_CLK_ENABLE();

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);

  /*Configure GPIO pin : B1_Pin */
  GPIO_InitStruct.Pin = B1_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  HAL_GPIO_Init(B1_GPIO_Port, &GPIO_InitStruct);

  /*Configure GPIO pins : USART_TX_Pin USART_RX_Pin */
  GPIO_InitStruct.Pin = USART_TX_Pin|USART_RX_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  GPIO_InitStruct.Alternate = GPIO_AF7_USART2;
  HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

  /*Configure GPIO pin : LD2_Pin */
  GPIO_InitStruct.Pin = LD2_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(LD2_GPIO_Port, &GPIO_InitStruct);

}

/* USER CODE BEGIN 4 */

/* USER CODE END 4 */

/**
  * @brief  This function is executed in case of error occurrence.
  * @param  None
  * @retval None
  */
void _Error_Handler(char * file, int line)
{
  /* USER CODE BEGIN Error_Handler_Debug */
  /* User can add his own implementation to report the HAL error return
state */
```

PWM Demonstration Software

```
  while(1)
  {
  }
  /* USER CODE END Error_Handler_Debug */
}

#ifdef USE_FULL_ASSERT

/**
   * @brief Reports the name of the source file and the source line number
   * where the assert_param error has occurred.
   * @param file: pointer to the source file name
   * @param line: assert_param error line source number
   * @retval None
   */
void assert_failed(uint8_t* file, uint32_t line)
{
  /* USER CODE BEGIN 6 */
  /* User can add his own implementation to report the file name and line number,
     ex: printf("Wrong parameters value: file %s on line %d\r\n", file,
line) */
  /* USER CODE END 6 */

}

#endif

/**
  * @}
  */

/**
  * @}
*/

/**** (C) COPYRIGHT STMicroelectronics ****/
```

- Debes tener en cuenta que configuré el preescalador en un valor de 655. Esto significaba que la entrada del reloj TIM2 estaría a una frecuencia aproximadamente igual a 977 kHz con un período de 10,23 µs. Este valor aseguró que la resolución del temporizador permitiría establecer anchos de pulso muy precisos.

Demonstration One

- Esta demostración utiliza los valores establecidos durante la configuración inicial para generar una forma de señal de ciclo de actividad del 50 %.
- El programa se creó primero y luego se descargó en el tablero del proyecto. Utilicé un osciloscopio USB para observar la señal PWM generada desde la salida del canal 1 del TIM2, que se emite desde el pin PA1.
- La Figura 10-6 muestra esta forma de señal.
- Debería poder ver que el período de sobreseñal es de 20 ms y el tiempo alto del pulso es de 10 ms, lo que crea el ciclo de actividad del 50 % como se esperaba.

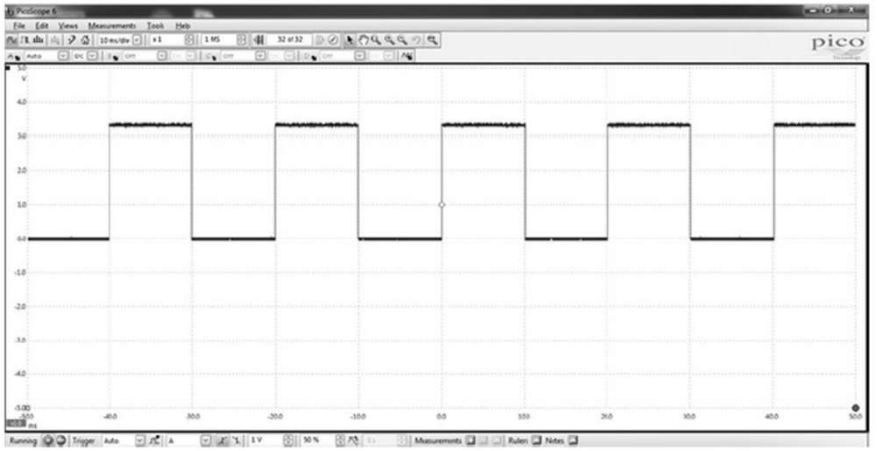


Figure 10-6 *Fifty percent duty-cycle PWM waveform.*

Demonstration One

- Esta forma de señal utilizó un valor de preescalador igual a 655, lo que provocó que el recuento de cada temporizador fuera de 10,23 µs como se analizó anteriormente.
- El valor del período se estableció en 1960, lo que significa que el tiempo del período real es 1960 veces 10,23 µs o 20050,8 µs, equivalente a 20,0508 ms.
- Eso estuvo lo suficientemente cerca para mis propósitos. El registro CCR1 tenía un valor de 980, lo que significaba que la salida conmutaba a 980 veces 10,23 µs o 10025,4 µs, equivalente a 10,0254 ms.

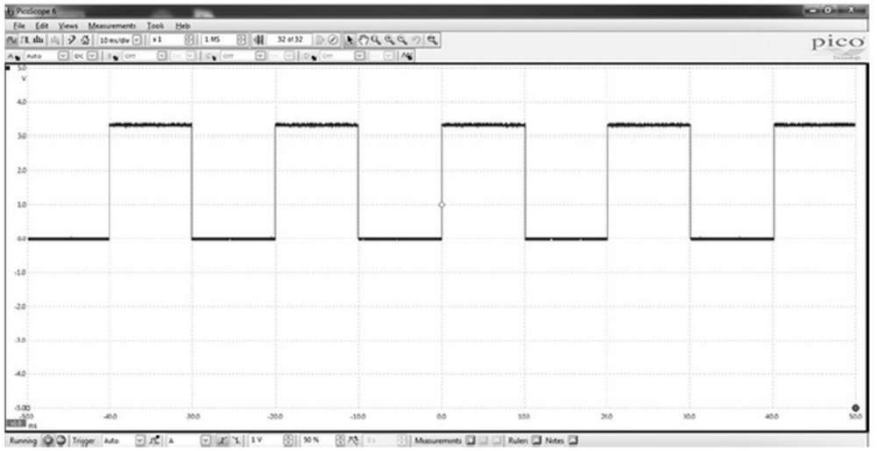


Figure 10-6 *Fifty percent duty-cycle PWM waveform.*

Demonstration One

- Esto está muy cerca de un ciclo de actividad del 50% a todos los efectos prácticos. Esta demostración demostró que la generación PWM estaba funcionando como se esperaba.
- La Tabla 10-1 se proporciona como una referencia útil que muestra una serie de valores de CCRx que generarán un ciclo de trabajo respectivo dado ese valor. Tenga en cuenta que los pequeños ciclos de trabajo que se muestran al principio de la tabla son importantes para la demostración del servocontrol. Esta tabla se basa en una señal PWM de período de 50 Hz o 20 ms.

CCRx (value)	Duty Cycle (%)	Pulse High Time (ms)
98	5	1
148	7.5	1.5
196	10	2
392	20	4
588	30	6
784	40	8
980	50	10
1176	60	12
1372	70	14
1568	80	16
1764	90	18
1960	100	20

Table 10-1 *CCRx Values and Corresponding Duty Cycles and Pulse High Times*

Demonstration Two

- Esta demostración controla la intensidad o luminiscencia de un único LED rojo, utilizando una salida PWM como se muestra en la demostración anterior. El único parámetro que se va a variar es el valor CCRx asignado en el método principal mediante esta declaración:

```
TIM2->CCR1 = 980; // CCR1 register set for 50% duty cycle
```

- Elegí variar este valor editando el archivo main.c y luego reconstruyéndolo rápidamente. Este proceso fue muy rápido y nada engorroso. También puede configurar una sesión de OpenOCD y ajustar el valor de forma dinámica, pero encontré que el proceso de edición, compilación y descarga es muy conveniente sin la molestia de crear una sesión de telnet. Le sugiero que pruebe ambos enfoques y utilice el que le resulte útil y productivo.

Connecting Channel 1 PWM Output to an LED/Resistor

- La Figura 10-7 muestra las conexiones para la misma combinación de LED/resistencia, que usé anteriormente para la demostración del proyecto en el Capítulo 5.
  - El LED/resistencia se conecta entre PA1 y tierra usando el protoboard Arduino.
- Test Results
  - Observé que la luminiscencia del LED variaba considerablemente con el ciclo de trabajo de la señal PWM aplicada. Debes tener en cuenta una distinción importante entre atenuar una lámpara incandescente y una LED.

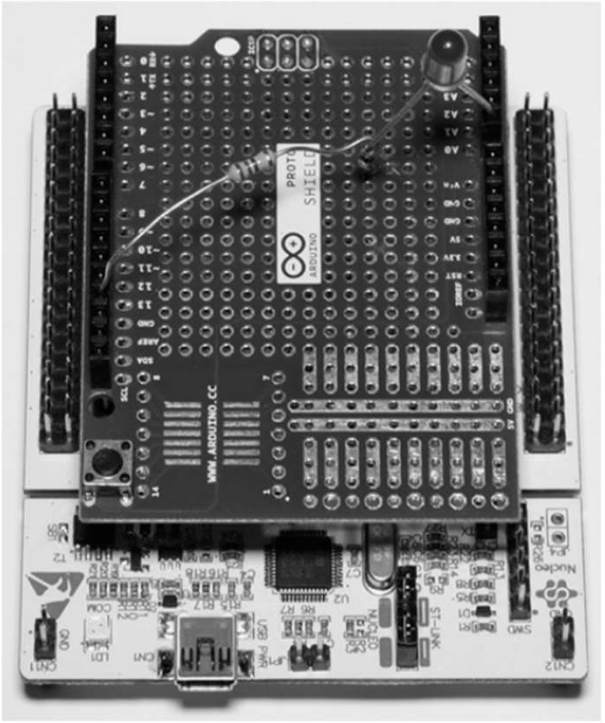


Figure 10-7 *LED/resistor connected on the Arduino protoboard.*

Connecting Channel 1 PWM Output to an LED/Resistor

- La intensidad de la lámpara incandescente corresponde a la tensión promedio que se le aplica, mientras que la luminiscencia del LED depende de la respuesta integradora de su ojo.
- Con esto quiero decir que la tensión aplicada al LED está en un nivel constante, pero el intervalo de tiempo aplicado varía con el ciclo de actividad.
- El ojo humano tenderá a integrar o promediar la intensidad de la luz LED y, por lo tanto, percibirá diferentes intensidades para distintos ciclos de actividad. El ojo también tiene una relación no lineal entre la luminancia real y el brillo percibido, como se muestra en la Figura 10-8.

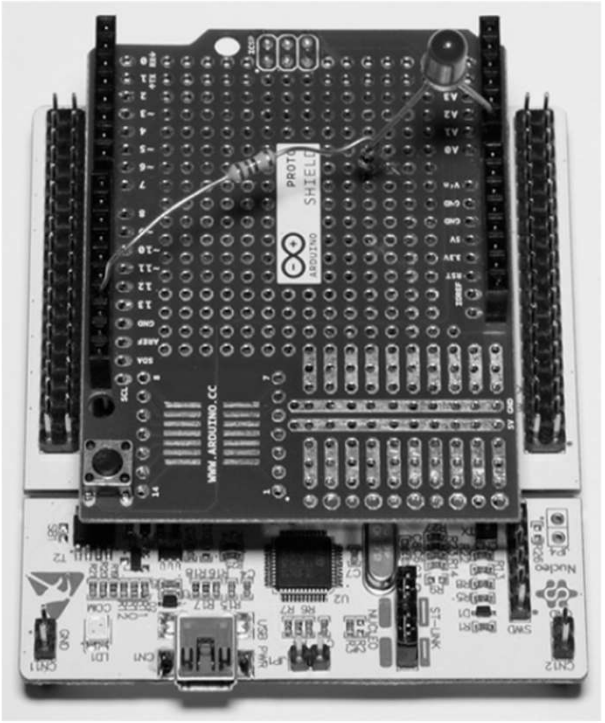


Figure 10-7 *LED/resistor connected on the Arduino protoboard.*

Connecting Channel 1 PWM Output to an LED/Resistor

- Lo que todo esto significa es que el ojo reconocerá más fácilmente diferentes luminiscencias de LED en ciclos de actividad pequeños, pero no percibirá ninguna diferencia significativa a medida que aumenta el ciclo de actividad.
- Hay ecuaciones que modelan bastante bien la relación entre el brillo del ojo y la luminancia y han sido codificadas como la fórmula de luminosidad CIE1931.
- Los fabricantes de LED de luz blanca han modificado sus productos utilizando esta relación para hacer que sus LED comerciales tengan una propiedad de atenuación lineal.

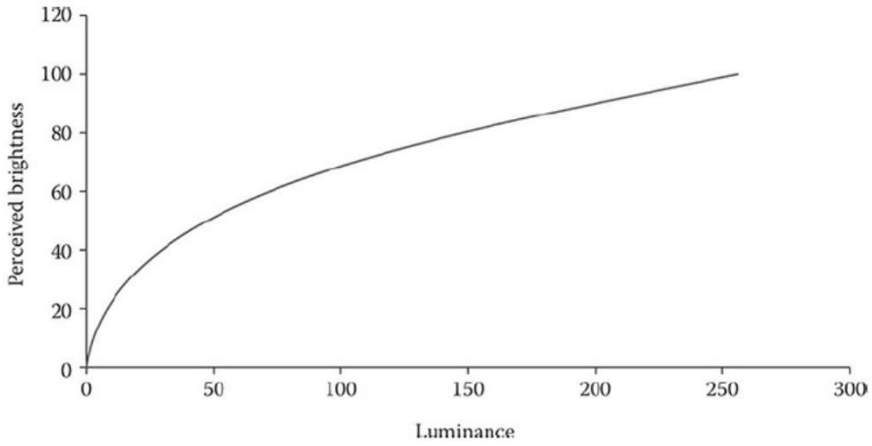


Figure 10-8 *Human eye brightness perception.*

Connecting Channel 1 PWM Output to an LED/Resistor

- No he hecho esto con esta demostración de un solo LED, pero es bastante factible y usted colocaría el código en el bucle permanente main.c.
- Intenté capturar la diferencia de intensidad entre ciclos de trabajo del 100% y el 10%.
- La Figura 10-9 es una fotografía compuesta con el 100% de PWM a la izquierda y el 10% de PWM a la derecha.

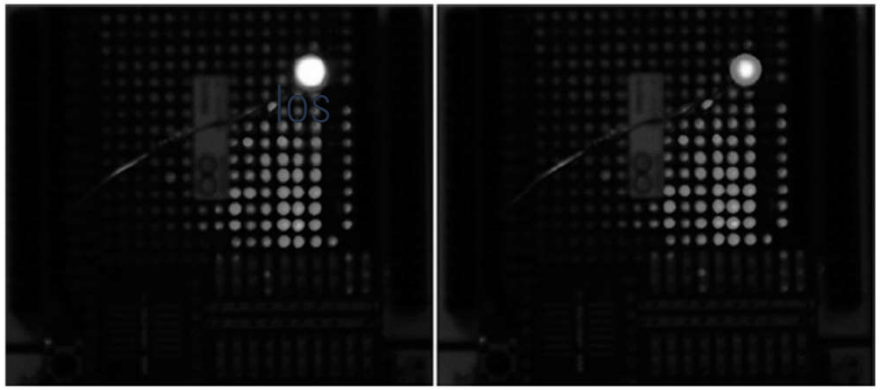


Figure 10-9 *100% and 10% PWM signal-duty cycles applied to a red LED.*

- La velocidad de obturación de la cámara requerida para estas imágenes era bastante larga, lo que tendía a hacer que la imagen del 10% fuera mucho más brillante de lo que realmente era. Supongo que tendrá que duplicar esta demostración y ver usted mismo el ciclo de trabajo real versus la luminancia percibida.

Demonstration Three

- En esta demostración controlaré un LED RGB que tiene tres líneas de entrada, una para cada color.
- El temporizador GP TIM2 se reconfigurará para tener tres salidas PWM conectadas a cada una de los pines del LED.
- Las líneas PWM controlarán tanto el color del LED como la luminiscencia según el ciclo de actividad establecido en cada pin.
- El primer paso es crear un nuevo proyecto CubeMX para manejar las líneas PWM adicionales.
- La Figura 10-10 muestra la vista de Pinout donde configuré TIM2 para salidas de cuatro canales, de los cuales usaré los primeros tres para controlar el LED RGB.

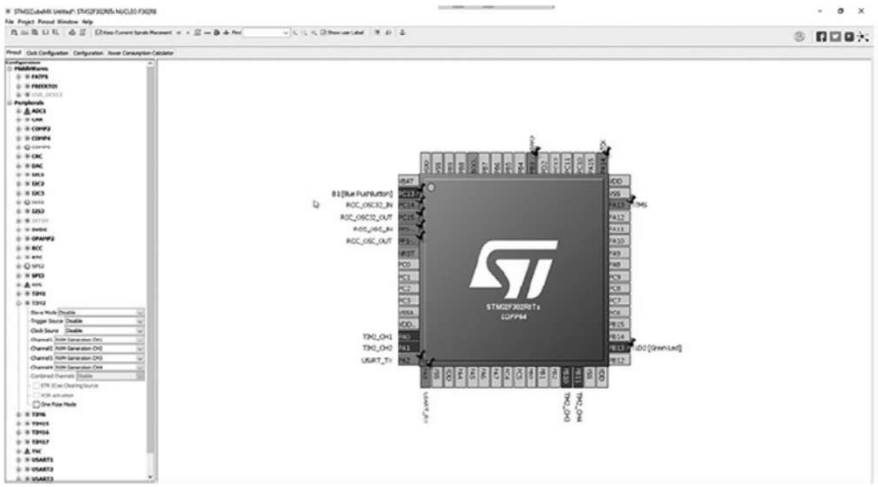


Figure 10-10 *Enabling timer TIM2 with PWM output on channels 1 to 4.*

Connecting Channel PWM Outputs to a RGB LED

- La Figura 10-11 es un esquema que muestra las conexiones entre el LED RGB y los pines GPIO del temporizador conectados en la protoplaca Arduino.
- Tenga en cuenta que cada pin de LED tiene una resistencia limitadora de corriente en serie para proteger tanto el pin GPIO como el LED RGB.
- Las salidas del canal del temporizador TIM2 se encuentran en los siguientes pines:
  - Canal 1—PA0
  - Canal 2—PA1
  - Canal 3—PB10
  - Canal 4—PB11

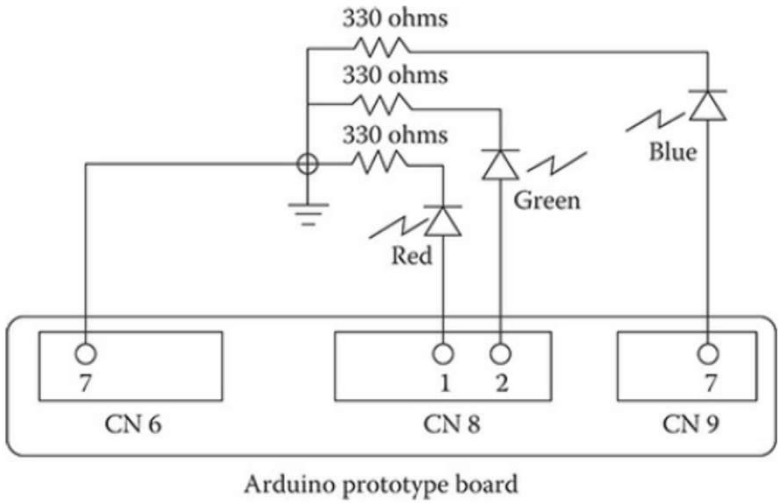


Figure 10-11 *RGB to Arduino protoboard connection schematic.*

Connecting Channel PWM Outputs to a RGB LED

- La Figura 10-12 muestra la configuración física de un LED RGB conectado a una placa de prototipo Arduino.

- Software Modifications
  - Se agregará un método MX_TIM2_Init modificado al archivo main.c en función de las salidas del canal adicional. Este nuevo método se enumera a continuación:

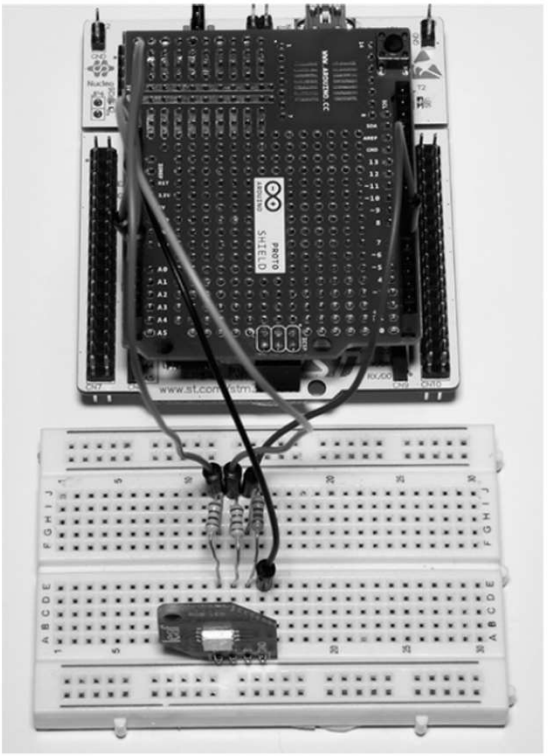


Figure 10-12 *Physical RGB LED connections to an Arduino protoboard.*

```
static void MX_TIM2_Init(void)
{
  TIM_MasterConfigTypeDef sMasterConfig;
  TIM_OC_InitTypeDef sConfigOC;

  htim2.Instance = TIM2;
  htim2.Init.Prescaler = 655;
  htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim2.Init.Period = 1960;
  htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_PWM_Init(&htim2) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig) !=
HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

  sConfigOC.OCMode = TIM_OCMODE_PWM1;
  sConfigOC.Pulse = 0;
  sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
  sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
  if (HAL_TIM_PWM_ConfigChannel(&htim2, &sConfigOC, TIM_CHANNEL_1) !=
HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }
  if (HAL_TIM_PWM_ConfigChannel(&htim2, &sConfigOC, TIM_CHANNEL_2) !=
HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }
  if (HAL_TIM_PWM_ConfigChannel(&htim2, &sConfigOC, TIM_CHANNEL_3) !=
HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }
  if (HAL_TIM_PWM_ConfigChannel(&htim2, &sConfigOC, TIM_CHANNEL_4) !=
HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }
```

uts to a RGB LED

```
  HAL_TIM_MspPostInit(&htim2);
}
```

- También hay un código nuevo que debe ingresarse en el método principal.
- Este nuevo código se enumera a continuación:

```
  /* USER CODE BEGIN 2 */

HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);
HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_2);
HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_3);

TIM2->CCR1 =  980; // Red channel
TIM2->CCR2 =  980; // Green channel
TIM2->CCR3 =  980; // Blue channel

  /* USER CODE END 2 */
```

- Tenga en cuenta que cada uno de los canales RGB se inicializó en un ciclo de actividad del 50%.

Connecting Channel PWM Outputs to a RGB LED

Test Results

- El LED RGB inicialmente mostró una luz blanca brillante, lo cual era de esperar porque todos los componentes LED individuales tenían el mismo ciclo de trabajo.
- Luego reinicié los registros CCRx individuales para probar cada color por separado y confirmé que estaban funcionando correctamente.
- También intenté variar los ciclos de trabajo para crear diferentes colores con poco éxito debido a la forma en que estaba construido el LED RGB. No fusionó la luz de cada LED de color y, por lo tanto, no creó correctamente un color alternativo, lo que puedes hacer si estuvieras mezclando pintura. Creo que habría tenido éxito si hubiera usado un LED RGB más caro con algunas ópticas incorporadas, pero desafortunadamente todo lo que tenía era un LED RGB muy económico.
- En cualquier caso, el objetivo de esta demostración era mostrarle cómo crear y utilizar múltiples salidas PWM, lo cual creo que se logró.

Demonstration Four

- Esta última demostración se centrará en el uso de una única señal PWM para controlar un servomotor de nivel aficionado.
  - Usaré el mismo programa que usé para las dos primeras demostraciones y no se requieren modificaciones de código.
  - Comenzaré la demostración revisando algunos datos básicos sobre cómo se controlan los servos analógicos.
- Analog Servo Control
  - Un servo analógico es esencialmente un motor eléctrico que tiene un circuito electrónico incorporado que convierte anchos de pulso específicos en una rotación proporcional.

Demonstration Four

- Las especificaciones para los servos analógicos de nivel aficionado son bastante “relajadas”, lo que significa que existen tolerancias flexibles que relacionan el ancho del pulso con la rotación física.
- Esto se debe principalmente al uso de componentes de muy bajo costo tanto en los componentes eléctricos como mecánicos que componen el servo.
- La tabla 10-2 detalla el ancho del pulso y la rotación esperada.

Pulse Width (50 Hz) (ms)	Servo Rotation with Respect to the Neutral Position (1.5 ms) (degrees)
1.0	–60
1.5 (neutral)	0
2.0	+60

Table 10-2 *Pulse Width and Servo Rotation*

Demonstration Four

- Un servo analógico no debe moverse de su posición neutral cuando se le envía una señal de pulso repetitiva de 1,5 ms.
- Un tren de pulsos de 1,0 ms hará que gire su eje principal 60° en sentido antihorario si se ve de frente.
- De manera similar, un tren de pulsos de 2,0 ms hará que gire 60° en el sentido de las agujas del reloj.
- Sin embargo, entran en juego tolerancias flexibles y, a menudo, el grado de rotación puede ser de 10° a 15°, más o menos.

Pulse Width (50 Hz) (ms)	Servo Rotation with Respect to the Neutral Position (1.5 ms) (degrees)
1.0	–60
1.5 (neutral)	0
2.0	+60

Table 10-2 *Pulse Width and Servo Rotation*

Demonstration Four

- En todos los casos, la frecuencia del tren de impulsos se establece en un período nominal de 50 Hz o 20 ms.
- La Figura 10-13 muestra el servo aficionado que utilicé para esta demostración. Es un modelo Hitec HS-311, que es un servo confiable y bien construido.
- Este servo requiere bastante corriente para funcionar, un poco más de lo que puede proporcionar la placa de proyecto STM.

Figure 10-13 *Hitec model HS-311 analog servo.*

Demonstration Four

- En consecuencia, utilicé una alimentación separada, como se puede ver en el esquema de conexión de la Figura 10-14.
- La señal de servocontrol de la salida PWM solo requiere varios miliamperes, lo que está dentro de las capacidades del controlador de pin GPIO.
- La Figura 10-15 muestra la forma de señal de la señal de posición neutral de 1,5 ms.
- Es una señal agradable y limpia con tiempos de subida y bajada bruscos, que son importantes para un control servo adecuado.

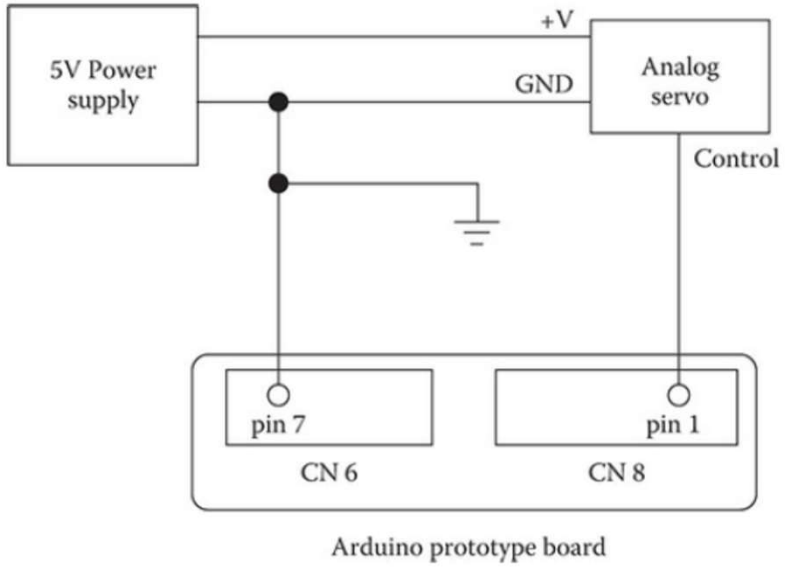


Figure 10-14 *Servo connection schematic.*

Demonstration Four

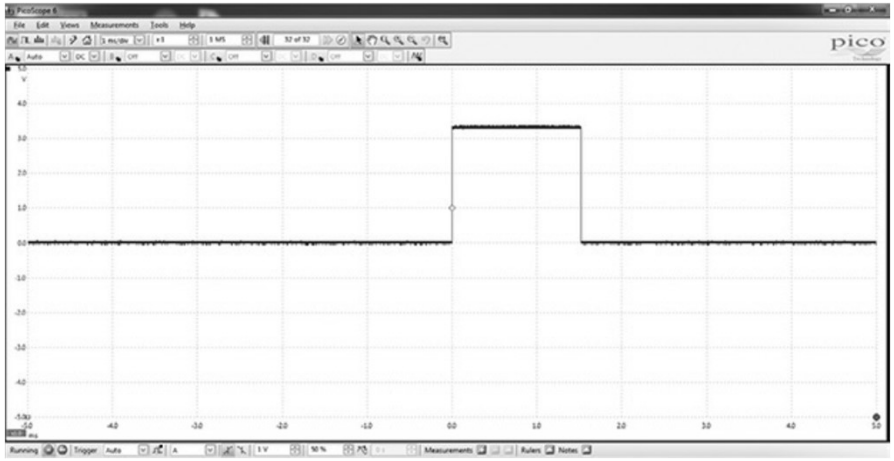


Figure 10-15 *1.5-ms servo control signal.*

- Las formas de onda de 1,0 ms y 2,0 ms son muy similares excepto por el ancho del pulso.
- La Figura 10-16 muestra la configuración física con el servo analógico conectado a una placa sin soldadura, que a su vez está conectada al protoboard Arduino con cables de puente.

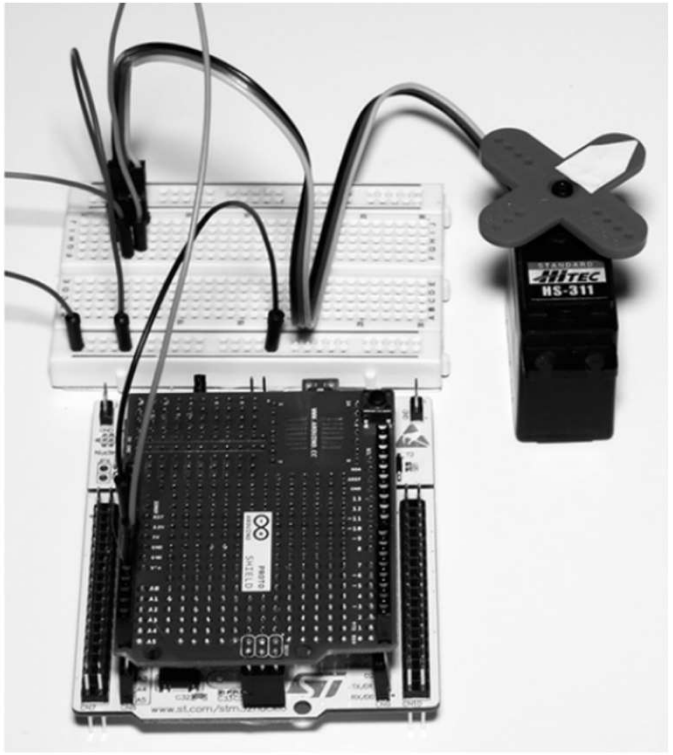


Figure 10-16 *Servo test setup.*

Demonstration Four

Test Results

- Primero cambié el valor de CCR1 a 148, lo que provoca que se genere un ancho de pulso de 1,5 ms. El servo se movió inmediatamente a su posición neutral después de que el programa se descargó en la placa del proyecto. Repetí el paso anterior con un CCR1 igual a 98, que genera un ancho de pulso de 1,0 ms. El servo respondió moviendo su eje principal 60° en sentido antihorario. Finalmente, cambié CCR1 a 196, lo que genera un ancho de pulso de 2,0 ms. El eje principal del servo giró 120° en el sentido de las agujas del reloj como se esperaba.
- El servo funcionó perfectamente y no mostró signos de inquietud o vibración, lo que a veces ocurre al usar servos de nivel aficionado.
- Luego agregué un código funcional y lo coloqué en el bucle eterno para ejercitar continuamente el servo, eliminando así la necesidad de editar, reconstruir y descargar el programa de control.

Demonstration Four

- Adding Functional Test Code
  - El siguiente código C se colocó dentro del archivo main.c para controlar continuamente el servo.
  - Este código provocó que el eje principal del servo oscilara lentamente entre los puntos finales de ±60°.
- Test Results
  - Observé que el eje principal del servo oscilaba lentamente entre los puntos finales como se esperaba después de descargar el código modificado en la placa del proyecto.
  - Esta demostración final concluye el capítulo. Creo que le he proporcionado suficiente experiencia que le permitirá utilizar servos con éxito con una placa STM Nucleo.

```
* USER CODE BEGIN PV */
// Private variables
int pw;
/* USER CODE END PV */


/* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
  /* USER CODE END WHILE */
    pw = 98;
    while(pw < 197)
    {
       TIM2->CCR1 = pw;
       HAL_Delay(20);
          pw++;
    }

    while(pw > 97)
    {
          TIM2->CCR1 = pw;
           HAL_Delay(20);
           pw--;
    }
  /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
```

Summary

- Comencé este capítulo sobre modulación de ancho de pulso (PWM) explicando qué constituye una señal PWM y detallando ciertas propiedades clave de esa señal.
- A esto le siguió una breve discusión sobre cómo el hardware del temporizador STM genera una señal PWM.
- Luego vino una discusión sobre cómo se usa el marco HAL para configurar e inicializar un temporizador de propósito general (GP) para emitir una señal PWM.
- Se mostró una lista completa de códigos C que generaba una señal PWM con un ciclo de trabajo del 50 % y una frecuencia de salida de 50 Hz.

Summary

- A continuación siguió la primera de cuatro demostraciones de PWM. La primera demostración de PWM simplemente generó la forma de onda del ciclo de trabajo del 50% que generó la placa del proyecto después de que el código se creó y descargó en la placa.
- La segunda demostración utilizó una forma de onda de ciclo de trabajo variable para atenuar e iluminar un LED rojo.
- La tercera demostración utilizó formas de onda PWM multicanal para controlar tanto el color como la intensidad de un LED RGB.
- La cuarta y última demostración ilustró cómo controlar un servo analógico estándar utilizando señales PWM.

Referencias

- Programming with STM32: Getting Started with the Nucleo Board and C/C++ 1st Edición - Donal Norris (Author)
- Nucleo Boards Programming with the STM32CubeIDE, Hands-on in more than 50 projects - Dogan Ibrahim (Author)
- STM32 Arm Programming for Embedded Systems, Using C Language with STM32 Nucleo - Muhammad Ali Mazidi (Author), Shujen Chen (Author), Eshragh Ghaemi (Author)

Manos a la obra con el . . .

. . . Proyecto Intermedio

. . . un enfoque centrado en la **práctica propia de la carrera** más que en el desarrollo teórico disciplinar, con eje en la **participación de las y los estudiantes**


Las y los estudiantes preguntarán:
¿en qué lío nos metimos?

¡Muchas gracias!

¿Preguntas?

. . .

Consultas a: jcruz@fi.uba.ar