7

TEMPORIZADORES

Los temporizadores son uno de los periféricos más importantes en un microcontrolador. Su función principal es generar eventos periódicos, medir intervalos de tiempo y sincronizar tareas. Un temporizador consta de una fuente de reloj (interna o externa), registros, divisores de frecuencia, selectores de reloj y lógica de interrupción.

En capítulos anteriores se ha empleado *vTaskDelay( )* para introducir pausas. Aunque es un método sencillo, resulta ineficiente ya que la CPU queda bloqueada y no puede atender otras tareas. Además, gestionar múltiples temporizaciones mediante *delays* complica el código y afecta al rendimiento. El uso adecuado de los temporizadores evita desperdiciar ciclos de CPU y aporta precisión a la hora de gestionar eventos sincronizados, por lo que es la solución idónea en sistemas embebidos y de tiempo real.

Una buena gestión del *timer* ayuda a optimizar los recursos del microcontrolador y hace que el programa sea más eficiente. Sin duda, el uso de temporizadores y su interrupción asociada será la solución preferida para gestionar eventos dependientes del tiempo.

Según la documentación oficial [3], el ESP32-S3 dispone de:

- *Timers* de propósito general (GPT), agrupados en Grupo ‘0’ y Grupo ‘1’.
- *Watchdog* asociados a cada grupo.
- *System Timer* (ESP-Timer), un temporizador global independiente.

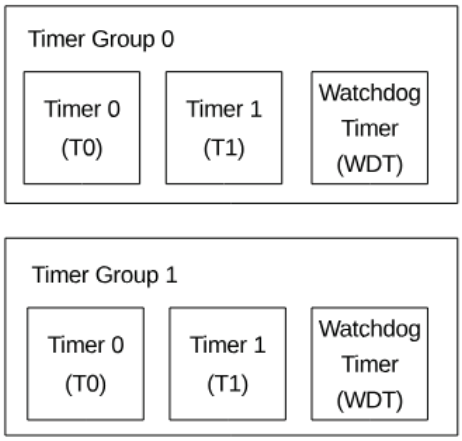


Imagen 20. Timers dentro de Grupo 0 y Grupo 1

Características principales de los *Timers* de Propósito General (GPT):

- El ESP32-S3 dispone de cuatro *timers* de propósito general (dos por grupo), de 54 bits, programables en incremento o decremento.
- Preescalado de 16 bits, rango divisor 2 a 65536.
- Lectura del contador en tiempo real.
- Posibilidad de parar y reanudar el contaje.
- Alarmas configurables.
- *Autoreload* automático o manual.
- Generación de interrupciones.

Propiedades del *System Timer* (o ESP_Timer):

- Dos contadores de 52 bits y tres comparadores.
- Frecuencia fija: 16 MHz.
- Tres fuentes de interrupción independientes.
- Modos de alarma: *Target* (52 bits) y Periódica (26 bits).
- Permite mantener referencia temporal al salir de modos de bajo consumo (*Deep-Sleep* / Light-Sleep).
- Puede detenerse en modos de depuración (OCD/JTAG).

7.1 CONFIGURACIÓN DE LOS GPT TIMER

Para utilizar los *timers* genéricos del SDK, se debe incluir la biblioteca "gptimer.h":

```
#include "driver/gptimer.h"
```

El *timer* se gestiona con un *handler*: *gptimer_handle_t* y se configura con la estructura *gptimer_config_t:*

```
#include "driver/gptimer.h"
gptimer_handle_t gp_timer = NULL;
gptimer_config_t gp_timer_config =
{
   .clk_src       = [GPTIMER_CLK_SRC_APB|GPTIMER_CLK_SRC_XTAL|GPTIMER_CLK_SRC_
DEFAULT]
   .direction     = [GPTIMER_COUNT_UP| GPTIMER_COUNT_DOWN],
   .resolution_hz = 1 * 1000 * 1000,      // 1 MHz, 1 tick = 1µs
   .priority      = [0|1|2|3],            // Orden creciente
};
```

Para crear el *timer* se usa la función:

```
esp_err_t gptimer_new_timer(const gptimer_config_t *config,
                            gptimer_handle_t *ret_timer);
```

Los *timers* se asocian a alarmas que son las que definen funciones de interrupción. Para crear una alarma se usa la estructura gptimer_alarm_config_t:

```
gptimer_alarm_config_t alarm_config; = {
.reload_count               = 0,      // Reiniciar el contador al valor inicial
.alarm_count                = 500000, // valor de la alarma en µs.
.flags.auto_reload_on_alarm = true,   // Auto-reload para repetir la alarma
   };
```

Se asocia la alarma y el *timer*:

```
esp_err_t gptimer_set_alarm_action(gptimer_handle_t timer,
                                   const gptimer_alarm_config_t *config);
```

A continuación, se define el *callback* que será llamado cuando se produzca el desbordamiento de la alarma:

```
gptimer_event_callbacks_t mycallbackAlarm = {
.on_alarm = timer_callback,          // Configurar el callback
};
```

Hay que usar *gptimer_register_event_callbacks* para asociar el *callback* al *timer*:

```
esp_err_t gptimer_register_event_callbacks (gptimer_handle_t timer,
                                    const gptimer_event_callbacks_t *cbs,
                                    void *user_data)
```

Por último, se habilita e inicia el *timer*.

```
esp_err_t gptimer_enable(gptimer_handle_t timer);
esp_err_t gptimer_start(gptimer_handle_t timer);
```

Nota

Para usar gptimer.h, hay que añadir el recurso *driver* en el *makefile "REQUIRES esp_driver_gptimer"*:

```
idf_component_register(SRCS "main.c"
                       PRIV_REQUIRES esp_driver_gptimer
                          INCLUDE_DIRS ".")
```

A continuación, se describe a bajo nivel (nivel de registro) cómo se configura la frecuencia del reloj que controla el *timer*. Aunque el compilador calcula automáticamente el divisor adecuado mediante la HAL, es importante comprender la arquitectura para identificar configuraciones inviables. En la siguiente sección se analizará un problema particular que puede ocurrir si la configuración del reloj no se realiza con cuidado.

Cada *timer* se configura de acuerdo con el siguiente esquema [3]:

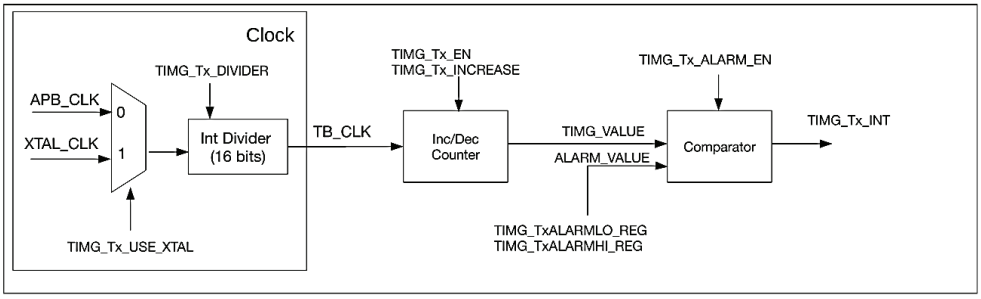


Imagen 21. Reloj interno del Timer GPT

Cada GPT Timer se incrementa a partir de dos fuentes de señal de reloj, APB_CLK (del bus de reloj, fija a 80 MHz) y XTAL_CLK (del cristal externo a 40 MHz). Se selecciona cada una de ellas con el campo *.clk_src* de la estructura *gptimer_config_t.*

A continuación, la frecuencia (APB_CLK o XTAL_CLK) se divide por un preescalado de 16 bits para generar la frecuencia base del *timer*: TB_CLK = CLK_SRC / prescaler.

Hay que tener en cuenta que el preescalado más grande es 65.535 (2^{16}-1) y el más pequeño 2. El *timer* o reloj tiene 54 bits de capacidad y se puede configurar en incremento o decremento de acuerdo al campo *.direction* en la estructura *gptimer_config_t.*

Se genera una alarma cuando el conteo del temporizador sea igual al valor de la alarma definido en los registros TIMG_Tx_ALARMLO_REG (32 bits) y TIMG_Tx_ALARMHI_REG (22 bits).

Para que la alarma esté activa, se tiene que habilitar con TIMG_Tx_ALARM_EN.

El divisor se configura en el registro TIMG_TxCONFIG_REG [3].

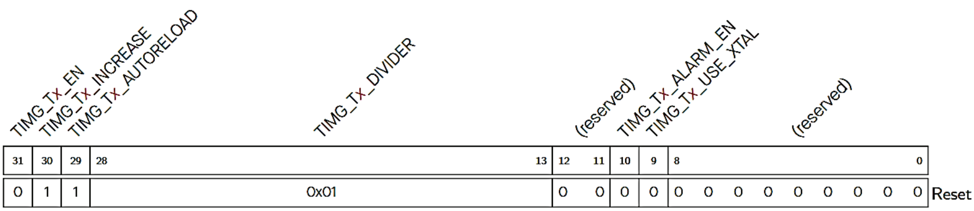


Imagen 22. Registro TIMG_TxCONFIG_REG

Como máximo, se podrán crear y utilizar 4 *timers* GPT.

Ejemplo 7. Señal temporizada por medio de un *Timer* genérico GPT.

El código que se muestra a continuación hace parpadear un LED conectado en GPIO4 con una frecuencia de 1 Hz (periodo T = 1s.). Además, se incluye el código de una función *print_config_timers* que muestra la configuración de cada grupo de temporizadores genéricos (GPT) del ESP32-S3.

```
#include <stdio.h>
#include “esp_system.h”
#include “soc/clk_tree_defs.h”
#include “esp_clk_tree.h”
#include “soc/timer_group_reg.h”
#include “soc/timer_group_struct.h”
#include “driver/gpio.h”
#include “driver/gptimer.h”

#define LED4 4

volatile int Flag_timer = 0;  //marcador de interrupción para su gestión en
main_app

//------------------------------------------------------------------------------
----
static uint32_t get_freq_cpu_hz(void)
{
    uint32_t freq = 0;
    esp_clk_tree_src_get_freq_hz(SOC_CPU_CLK_SRC_PLL,
                         ESP_CLK_TREE_SRC_FREQ_PRECISION_EXACT,
                         &freq);
    return freq;
}
//------------------------------------------------------------------------------
---
static uint32_t get_freq_apb_hz(void)
{
    uint32_t freq = 0;
    esp_clk_tree_src_get_freq_hz(SOC_MOD_CLK_APB,
                         ESP_CLK_TREE_SRC_FREQ_PRECISION_EXACT,
                         &freq);
    return freq;
}
//------------------------------------------------------------------------------
---
static uint32_t get_freq_xtal_hz(void)
{
    uint32_t freq = 0;
    esp_clk_tree_src_get_freq_hz(SOC_MOD_CLK_XTAL,
                                 ESP_CLK_TREE_SRC_FREQ_PRECISION_EXACT,
```

```
                                    &freq);
    return freq;
}


//------------------------------------------------------------------------
----
static uint32_t timer_base_hz(bool use_xtal, uint32_t f_xtal, uint32_t f_apb)
{
    return use_xtal ? f_xtal : f_apb;
}
//------------------------------------------------------------------------
----
static uint32_t timer_freq_hz(uint32_t divider, bool use_xtal,
 uint32_t freq_xtal, uint32_t freq_apb)
{
    if (divider == 0)  return 0;
    return timer_base_hz(use_xtal, freq_xtal, freq_apb) / divider;
}

//------ Función para imprimir la configuración de los GPT  ----------------
----
void print_config_timers(void)
{
    uint32_t freq_cpu  = get_freq_cpu_hz();
    uint32_t freq_apb  = get_freq_apb_hz();
    uint32_t freq_xtal = get_freq_xtal_hz();

    printf("CPU  Freq = %lu MHz\n", freq_cpu  / 1000000);
    printf("APB  Freq = %lu MHz\n", freq_apb  / 1000000);
    printf("XTAL Freq = %lu MHz\n", freq_xtal / 1000000);

    // ----- Timer Group 0 -----
    uint32_t t0cfg_g0 = REG_READ(TIMG_T0CONFIG_REG(0));
    uint32_t t1cfg_g0 = REG_READ(TIMG_T1CONFIG_REG(0));

    uint32_t div_t0_g0 = REG_GET_FIELD(TIMG_T0CONFIG_REG(0), TIMG_T0_DIVIDER);
    uint32_t div_t1_g0 = REG_GET_FIELD(TIMG_T1CONFIG_REG(0), TIMG_T1_DIVIDER);

    bool use_xtal_t0_g0 = REG_GET_FIELD(TIMG_T0CONFIG_REG(0), TIMG_T0_USE_XTAL);
    bool use_xtal_t1_g0 = REG_GET_FIELD(TIMG_T1CONFIG_REG(0), TIMG_T1_USE_XTAL);
```

```
    // ----- Timer Group 1 -----
    uint32_t t0cfg_g1 = REG_READ(TIMG_T0CONFIG_REG(1));
    uint32_t t1cfg_g1 = REG_READ(TIMG_T1CONFIG_REG(1));

    uint32_t div_t0_g1 = REG_GET_FIELD(TIMG_T0CONFIG_REG(1), TIMG_T0_DIVIDER);
    uint32_t div_t1_g1 = REG_GET_FIELD(TIMG_T1CONFIG_REG(1), TIMG_T1_DIVIDER);

    bool use_xtal_t0_g1 = REG_GET_FIELD(TIMG_T0CONFIG_REG(1), TIMG_T0_USE_XTAL);
    bool use_xtal_t1_g1 = REG_GET_FIELD(TIMG_T1CONFIG_REG(1), TIMG_T1_USE_XTAL);

    // ----- LOG de salida -----
    printf(«TIMG0.T0: CFG=0x%08lX DIV=%lu USE_XTAL=%d f_timer=%lu Hz\n»,
             t0cfg_g0, div_t0_g0, use_xtal_t0_g0,
             timer_freq_hz(div_t0_g0, use_xtal_t0_g0, freq_xtal, freq_apb));

    printf(«TIMG0.T1: CFG=0x%08lX  DIV=%lu  USE_XTAL=%d f_timer=%lu Hz\n»,
             t1cfg_g0, div_t1_g0,
             use_xtal_t1_g0,
             timer_freq_hz(div_t1_g0, use_xtal_t1_g0, freq_xtal, freq_apb));

    printf("TIMG1.T0: CFG=0x%08lX DIV=%lu USE_XTAL=%d f_timer=%lu Hz\n",
             t0cfg_g1, div_t0_g1,
             use_xtal_t0_g1,
             timer_freq_hz(div_t0_g1, use_xtal_t0_g1, freq_xtal, freq_apb));

    printf("TIMG1.T1: CFG=0x%08lX DIV=%lu USE_XTAL=%d f_timer=%lu Hz\n",
             t1cfg_g1, div_t1_g1,
             use_xtal_t1_g1,
             timer_freq_hz(div_t1_g1, use_xtal_t1_g1, freq_xtal, freq_apb));
}
//---------------------------------------------------------------------
//----------------- ISR del timer -------------------------------------
//---------------------------------------------------------------------
bool IRAM_ATTR GPT_TIMER_ISR(gptimer_handle_t timer,
                             const gptimer_alarm_event_data_t *event,
                             void *user_data)
{
    Flag_timer = 1;  // MARCA EL EVENTO!!
    return true;     // Return true para que el temporizador vuelva a reiniciar
}
```

```
// -----------------app_main-----------------------------------------
void app_main(void)
{
    // Configurar GPIO4
    gpio_set_direction(LED4, GPIO_MODE_INPUT_OUTPUT);

    // Handler
    gptimer_handle_t gptimer = NULL;

    // Configurar el temporizador
    gptimer_config_t timer_config = {
        .clk_src       = GPTIMER_CLK_SRC_DEFAULT,  // Clk por defecto (APB 80
MHz)
        .direction     = GPTIMER_COUNT_UP,         // Contar hacia arriba
        .resolution_hz = 1000*1000,                // 1 MHz (1us por tick)
    };

    gptimer_new_timer(&timer_config, &gptimer);

    // Configurar la alarma (en µs)
    gptimer_alarm_config_t alarm_config = {
        .reload_count = 0,                   // Reiniciar el contador al valor
inicial
        .alarm_count  = 500000,              // 0,5 segundos = 500000 µs
        .flags.auto_reload_on_alarm = true,  // Auto-reload para repetir la
alarma
    };

    gptimer_set_alarm_action(gptimer, &alarm_config);

    // Registrar el callback para la interrupción
    gptimer_event_callbacks_t cbs = {
        .on_alarm = GPT_TIMER_ISR,           // Configurar el callback
    };
    gptimer_register_event_callbacks(gptimer, &cbs, NULL);

    // Habilita e inicia el temporizador
    gptimer_enable(gptimer);
    gptimer_start(gptimer);
```

```
    printf("GPTimer started!\n");

    // Muestra la configuración de los 4 GPT Timers.
    print_config_timers();

    while (1)
    {
        // GESTIONA EL EVENTO temporal dentro de while(1)
        if (Flag_timer)
        {
            Flag_timer = 0;                              // Borra el flag

            printf("Timer\n");
            gpio_set_level(LED4, !gpio_get_level(LED4)); // conmuta GPIO4
        }
    }
}
```

7.2 ASPECTOS A TENER EN CUENTA EN LA CONFIGURACIÓN DEL GPT TIMER

Se muestran a continuación varios casos de configuración del GPT para diferentes temporizaciones analizando cómo se configuran internamente los registros de preescalado a través de la capa HAL.

Caso 1

Timer '0' que opere a 1 MHz. (`.resolution_hz = 1000000`)

El compilador va a seleccionar por defecto APB_CLK = 80 MHz.

Así pues, la frecuencia base del *timer* = $\frac{APB\_CLK}{Pre\text{-}escalado}$

Que resulta en Pre – escalado = $\frac{APB\_CLK}{frec.\ base} = \frac{80e6}{1e6} = 80$

El programa reporta:

```
CPU  Freq  = 160 MHz
APB  Freq  = 80 MHz
XTAL Freq = 40 MHz
TIMG0.T0: CFG=0xE00A0400  DIV=80  USE_XTAL=0  f_timer=1000000 Hz
```

```
TIMG0.T1: CFG=0x60002000  DIV=1  USE_XTAL=0  f_timer=80000000 Hz
TIMG1.T0: CFG=0x00000000  DIV=0  USE_XTAL=0  f_timer=0 Hz
TIMG1.T1: CFG=0x00000000  DIV=0  USE_XTAL=0  f_timer=0 Hz
```

El Timer '0' del banco Grupo '0', usará un divisor de 80, usando APB_CLK (USE_XTAL='0') a 80 MHz y la frecuencia base del *timer* obtenida es de 1.000.000 Hz = 1 MHz.

Caso 2

Timer '0' que opere a 100 kHz. (`.resolution_hz = 100000`). Se obtendrá un divisor de 800 respecto de la frecuencia de APB_CLK. El programa reporta:

```
CPU  Freq = 160 MHz
APB  Freq = 80 MHz
XTAL Freq = 40 MHz
TIMG0.T0: CFG=0xE0640400  DIV=800  USE_XTAL=0  f_timer=100000 Hz
TIMG0.T1: CFG=0x60002000  DIV=   USE_XTAL=0  f_timer=80000000 Hz
TIMG1.T0: CFG=0x00000000  DIV=0  USE_XTAL=0  f_timer=0 Hz
TIMG1.T1: CFG=0x00000000  DIV=0  USE_XTAL=0  f_timer=0 Hz
```

De nuevo, coherente con los cálculos.

Caso 3

Timer '0' que opere a 1 kHz. (`.resolution_hz = 1.000`). Se necesitaría un divisor de 80.000 respecto de la frecuencia de APB_CLK. Este valor excede 65.535, por lo que se producirá una excepción y reiniciará el microcontrolador dando lugar a un error crítico. Para que el temporizador funcione correctamente, es necesario cambiar la fuente del reloj (`.clk_src = GPTIMER_CLK_SRC_XTAL`). Ahora ese valor sí es alcanzable ya que la frecuencia es la mitad y el divisor será de 40.000.

```
CPU  Freq = 160 MHz
APB  Freq = 80 MHz
XTAL Freq = 40 MHz
TIMG0.T0: CFG=0xF3880600  DIV=40000  USE_XTAL=1  f_timer=1000 Hz
TIMG0.T1: CFG=0x60002000  DIV=1  USE_XTAL=0  f_timer=80000000 Hz
TIMG1.T0: CFG=0x00000000  DIV=0   USE_XTAL=0  f_timer=0 Hz
TIMG1.T1: CFG=0x00000000  DIV=    USE_XTAL=0  f_timer=0 Hz
```

Caso 4

Como puede deducirse, con la configuración de reloj que incorpora por defecto la placa de desarrollo ESP32-S3-DevKitC-1 hay un límite inferior de la resolución definido por el máximo del preescalado y de XTAL_CLK que no se debe sobrepasar. Este valor será: $\frac{40e6}{65535} < f \rightarrow$ (`.resolution_hz > 610 Hz`). En caso de configurar la resolución por debajo de este valor, el microcontrolador produciría una excepción y se reiniciaría de forma continua. Se obtiene un mensaje de error similar a este:

```
Assert failed: timer_ll_set_clock_prescale … (divider >= 2 &&  divider <= 65535)
```

Nota

Normalmente no será necesario utilizar relojes con una resolución tan baja.

7.3 CONFIGURACIÓN DEL SYSTEM TIMER

System Timer es el *timer* del sistema. Es también de alta resolución dado que emplea 52 bits. No se puede reiniciar, pero si evaluar incrementos de tiempo. La resolución es fija de valor 1µs. Permite medir tiempos y generar interrupciones *software* o *callback.*

ESP-Timer para medida de tiempos

Se incluirá la cabecera "esp_timer.h".

```
#include "esp_timer.h"
```

Ya se puede calcular el tiempo en µs entre dos instantes t1 y t2 con *esp_timer_get_time*:

```
int64_t esp_timer_get_time(void);
```

Por ejemplo:

```
int64_t t1 = esp_timer_get_time();   // Tiempo t1 en µs.
… //Tarea
int64_t t2 = esp_timer_get_time();   // Tiempo t2 en µs.
//Tiempo Transcurrido: dT = t2-t1;
```

Configurar el *callback* a través de ESP_Timer

En primer lugar, se necesita un *handler* para manejar el temporizador.

```
esp_timer_handler_t myTimer_ESP;
```

A continuación, se crea la estructura de configuración del *timer esp_timer_create_args_t*, donde se indica la función de *callback*:

```
const esp_timer_create_args_t periodic_timer_args =
{
  .callback = &periodic_timer_callback,       //callback
  .name = "periodic"                          //nombre
};
```

Por último, crea e inicia la alarma asociada al *callback*:

```
esp_err_t esp_timer_create(const esp_timer_create_args_t *create_args,
        esp_timer_handle_t *out_handle);
esp_err_t esp_timer_start_periodic(esp_timer_handle_t timer, uint64_t
period);
```

ⓘ Notas

1) El *callback* se debe pasar como argumento (void *arg).

2) Para poder usar esp_timer.h, hay que añadir el recurso *esp_timer* en el makefile:

```
idf_component_register(SRCS ${srcs}
                       PRIV_REQUIRES esp_timer
                       INCLUDE_DIRS ${includes})
```

Ejemplo 8. Señal temporizada por medio de System Timer y *callbacks*.

El código que se muestra a continuación hace parpadear un LED conectado en GPIO4 con una frecuencia de 1 segundo usando el Timer del Sistema ESP_Timer.

```
#include <stdio.h>
#include "driver/gpio.h"
#include "esp_timer.h"

#define LED4 4
```

```
int state_led_1 = 0;
int cont = 0;

esp_timer_handle_t myESP_Timer;

void ESP_TimerCallback (void* arg);

//----------app_main ------------------------------------------------------------
void app_main(void)
{
    gpio_set_direction(LED4, GPIO_MODE_OUTPUT);

    const esp_timer_create_args_t My_ESP_Timer_Configuration =
    {
        .callback = &ESP_TimerCallback,              //función callback
        .name     = "ISR ESP_Timer "                 //nombre
    };

    //configura y arranca el callback asociado al System Timer
    esp_timer_create(&My_ESP_Timer_Configuration, &myESP_Timer);
    esp_timer_start_periodic(myESP_Timer, 500000);   //500000 µs = 0,5s

    while(1)
    {
         //Bucle principal
    }
}
//----------callback asociado al System Timer---------------------------------
void ESP_TimerCallback (void* arg)
{
    printf("ESP Timer: %d\n",cont++);

    gpio_set_level(LED4,state_led_1);
    state_led_1 = !state_led_1;
}
```

7.4 EJERCICIOS

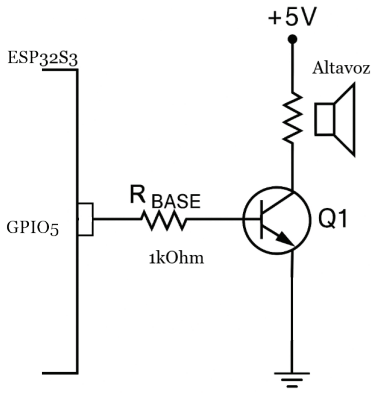
- Escribe un programa que haga parpadear un LED conectado a GPIO4 a 2 Hz. En cada pulsación de BOOT, la frecuencia de parpadeo se incrementará +1 Hz. Cuando llegue a 15 Hz, en la siguiente pulsación de BOOT, volverá a parpadear a 2 Hz.

ⓘ Nota

Observar que para generar una señal cuadrada de frecuencia f, el periodo de la interrupción debe ser la mitad del periodo de dicha frecuencia f. En la primera mitad la señal estará a '1' y en la segunda mitad a '0'. (T = 1/ f).

- Escribe un programa que saque a través de GPIO5 una frecuencia equivalente a la de las notas musicales. Cuando se pulsa BOOT, se cambia de nota. Conectar un pequeño altavoz a la salida utilizando un transistor bipolar:

Do	261.63 Hz
Re	293.66 Hz
Mi	329.63 Hz
Fa	349.23 Hz
Sol	392.00 Hz
La	440.00 Hz
Si	493.88 Hz



- Se desea medir la frecuencia de una señal digital externa conectada al pin GPIO8.

  La señal es un tren de pulsos cuadrado entre 10 Hz y 2 kHz. Escribe un programa que calcule la frecuencia.

  - Usa interrupciones (flanco ascendente) asociadas a GPIO8.
  - En cada interrupción toma el tiempo transcurrido con *esp_timer_get_time*( ).
  - Calcula el periodo instantáneo entre dos flancos consecutivos.
  - Muestra la frecuencia promedio en el terminal serie cada 500 ms.

8

PWM

La modulación PWM (*pulse-width modulation*) es una técnica que permite codificar información mediante la variación del ancho de pulso de una señal que conmuta entre dos estados '0' y '1'.

En una señal PWM se distinguen:

- *Ton*: tiempo que la señal está a '1' (anchura del pulso).
- *Toff*: tiempo que la señal está a '0'.
- *T* = *Ton*+*Toff*: periodo de la señal PWM (constante).
- Ciclo de trabajo (%) = $\text{duty cycle (\%)} = 100\frac{Ton}{Ton+Toff} = 100\frac{Ton}{T}$
  Es el porcentaje de tiempo que la señal está a '1' sobre el periodo total.

La frecuencia base de la señal: $\text{f (Hz)} = \frac{1}{T} = \frac{1}{Ton+Toff}$

En la imagen se muestra la forma de onda para diferentes ciclos de trabajo con una frecuencia base de 1 kHz.

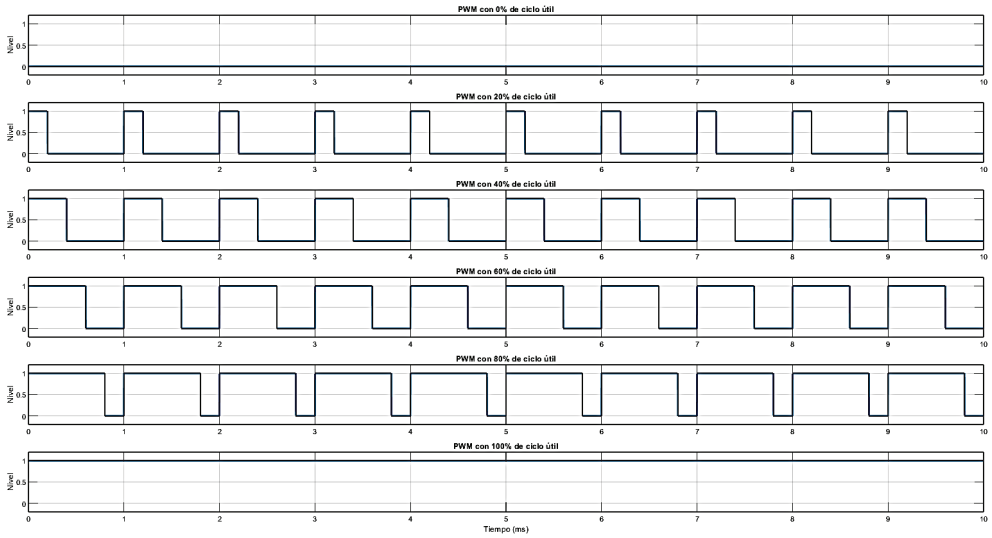


Imagen 23. Señal PWM con diferentes anchos de pulso

Se puede utilizar una señal PWM para conmutar LED u otros dispositivos de doble estado (on-off) y conseguir una señal pseudo analógica, es decir que aparenta ser de naturaleza analógica, aunque en realidad es digital.

Una señal PWM puede generarse de dos formas:

- Por medio de temporizadores (PWM *software*). Se utiliza un *timer* para generar interrupciones periódicas y el programa conmuta manualmente el pin en los instantes Ton/Toff. Es una solución flexible, pero consume CPU y no garantiza precisión a frecuencias elevadas.
- Por medio de *hardware* (PWM *hardware*), que resulta más eficiente. Utiliza un *timer* interno que proporciona la frecuencia base. El ciclo de trabajo se ajusta en función de la resolución del temporizador.

8.1 CONFIGURACIÓN DEL PWM

Para configurar el PWM-*hardware* habrá que incluir la biblioteca “driver/ledc.h”.

En primer lugar, se debe crear y configurar el canal por medio de una estructura de tipo *ledc_channel_config*. Por ejemplo:

```
ledc_channel_config_t ledc_channel =
{
   .speed_mode    = [LEDC_HIGH_SPEED_MODE | LEDC_LOW_SPEED_MODE | LEDC_SPEED_
MODE_MAX],
      .channel       = [LEDC_CHANNEL_0|_1|…|_7],
      .timer_sel     = [LEDC_TIMER_0|_1|_2|_3],
      .intr_type     = LEDC_INTR_DISABLE,
      .gpio_num      = LEDC_OUTPUT_IO,
      .duty          = 0,
};
```

A continuación, se inicializa el canal que se ha configurado:

```
esp_err_t ledc_channel_config (ledc_channel_config_t *ledc_conf);
```

El siguiente paso consiste en configurar el *timer* que se va a emplear para generar la frecuencia base del PWM, para ello se utiliza la estructura ledc_timer_config_t:

```
ledc_timer_config_t ledc_timer =
{
   .speed_mode       = [LEDC_HIGH_SPEED_MODE|LEDC_LOW_SPEED_MODE|LEDC_SPEED_
MODE_MAX],
      .duty_resolution  = LEDC_DUTY_RES,
      .timer_num        = [LEDC_TIMER_0|1|2|3],
      .freq_hz          = LEDC_FREQUENCY,  // en Hz, por ejemplo 4000 = 4 kHz
      .clk_cfg          = LEDC_AUTO_CLK,
};
```

Seguidamente, se configura el *timer* asociado a la PWM:

```
esp_err_t ledc_channel_config(const ledc_channel_config_t *ledc_conf);
```

Por último, para cambiar el ciclo de trabajo, se deberá llamar a estas dos funciones:

```
esp_err_t ledc_set_duty(ledc_mode_t speed_mode,
                        ledc_channel_t channel,
                        uint32_t duty_cycle);
esp_err_t ledc_update_duty(ledc_mode_t speed_mode, ledc_channel_t channel);
```

Nota

El ciclo de trabajo *duty_cycle* es relativo a la resolución (*res*) de acuerdo con la expresión $duty\_cycle \cdot 2^{res} - 1$. Por ejemplo, para una resolución de 12 bits, un ciclo de trabajo del 100% corresponde con un contaje de $1 \cdot 2^{12} - 1 = 4095$, el 0% corresponde con un contaje de 0 y el 50% corresponde con un contaje de $0{,}5 \cdot 2^{12} - 1 = 2047$.

Ejemplo 9. Programa que genera PWM de forma *hardware*.

El siguiente ejemplo, basado en la solución de Espressif para control de LED: \v5.5.1\esp-idf\examples\peripherals\ledc\ledc_basic\main\ledc_basic_example_main.c, se genera una señal PWM en GPIO6 cuyo ancho de pulso se incrementa progresivamente cada 200ms haciendo un efecto de iluminación creciente.

```
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "driver/gpio.h"
#include "driver/ledc.h"

#define LED6 6

#define LEDC_MODE           LEDC_LOW_SPEED_MODE

void app_main(void)
{
    // configura salida digital
    gpio_set_direction(LED6,GPIO_MODE_OUTPUT);

    // configura y habilita el canal PWM
    ledc_channel_config_t ledc_channel = {
        .speed_mode       = LEDC_MODE,
        .channel          = LEDC_CHANNEL_0,
        .timer_sel        = LEDC_TIMER_0,
        .intr_type        = LEDC_INTR_DISABLE,
        .gpio_num         = LED6,
        .duty             = 0,
    };
    ledc_channel_config(&ledc_channel);
```

```
    // configura y habilita el timer asociado al canal
     ledc_timer_config_t ledc_timer = {
         .speed_mode       = LEDC_MODE,
         .duty_resolution  = LEDC_TIMER_10_BIT, // 10 bits de resolución 0..1023
         .timer_num        = LEDC_TIMER_0,
         .freq_hz          = 8000,              // 8 kHz de frecuencia
         .clk_cfg          = LEDC_AUTO_CLK
     };
     ledc_timer_config(&ledc_timer);

     uint32_t dutycycle = 0;

     while (1)
     {
         // actualiza el ancho de pulso
         ledc_set_duty(LEDC_MODE, LEDC_CHANNEL_0, dutycycle);
         ledc_update_duty(LEDC_MODE, LEDC_CHANNEL_0);

         dutycycle += 32;          // Incremento gradual en 32 escalones
         if (dutycycle>1023)       // 10 bits de resolución -> max = 1023
             dutycycle = 0;

         // delay de 200ms
         vTaskDelay(pdMS_TO_TICKS(200));
     }
 }
```

8.2 EJERCICIOS

- Escribe un programa que haga encenderse un LED conectado en GPIO4. La intensidad lumínica se irá incrementando de forma lineal desde completamente apagado a intensidad máxima. Una vez alcanzado el máximo, iniciará de nuevo la secuencia desde 0. La secuencia tendrá un periodo de 5 s.
- Usando un potenciómetro externo y el ADC como se vio en el CAPÍTULO 6, escribe un programa que varíe la intensidad luminosa de un LED en función del valor de tensión del potenciómetro.

- Escribe un programa que controle dos LED mediante dos canales PWM independientes. Cuando el primer LED aumenta su luminosidad, el segundo debe disminuirla en la misma proporción y viceversa, generando un efecto "balancín" o *fade inverso*.

- Conecta un transistor bipolar NPN (por ejemplo, 1N2222A) a una salida digital a través de una resistencia limitadora de base (Rb = 1kΩ). Conecta el emisor a GND. En el colector conecta un motor DC de pequeña potencia a una fuente externa de 5V. Pon un diodo en antiparalelo con el motor DC. Escribe un programa que haga variar la velocidad del motor en función del ancho de pulso en la salida digital. Para definir la velocidad de giro usa un potenciómetro en otra GPIO, de forma que, si la lectura del ADC es 0V, el motor esté parado y si la lectura del ADC es 3,3V, la velocidad del motor sea máxima y varíe linealmente en función de la tensión del ADC.