CMSIS-RTOS V2 Ejemplos

Release 1.0

Mariano Ruiz

Oct 24, 2025

CONTENTS

El repositorio contiene ejemplos básicos para entender el funcionamiento de la API CMSIS-RTOS V2 utilizando el sistema operativo RTX version 5. Los ejemplos están implementados para el STM32F429 y utilizan mínimamente los periféricos del microcontrolador y hacer hincapié en los conceptos de manejo del Sistema Operativo.

Note

Uso de los ejemplos

Los ejemplos se han implementado para la asignatura **Sistemas Basados en Microprocesador** de la **(ETSI Sistemas de Telecomunicación) Universidad Politécnica de Madrid** y se pueden ejecutar utilizando el simulador del microprocesador incluido en el entorno de ARM keil Microvision o bien el hardware.

CHAPTER

ONE

DOCUMENTACIÓN EN OTROS IDIOMAS

Traducción a inglés - English translation

CHAPTER

TWO

DESCARGA DEL CÓDIGO

Para descargar el código puede utilizar un cliente de git en su ordenador o bien descargar el repositorio completo (formato **zip**). Las instrucciones para clonar el repositorio son:

Note

Descarga del código

```
$ git clone https://github.com/mruizglz/SBM-rtos.git
```

CHAPTER

THREE

LISTADO DE EJEMPLOS INCLUIDOS

Table 3.1: Ejemplos incluidos

Carpeta	Objetivos
ejemplothreads	Aprender el manejo básico de creación de threads. Uso de la misma función con parámetros parea crear multiples threads
ejemplothreads-flags	Sincronización de threads usando flags
ejemplothreads-queues	Intercambio de datos entre threads usando colas
ejemplothreads-timers	Gestion de timers “software”

CHAPTER

FOUR

CONFIGURACIÓN DEL PROYECTO DE KEIL

4.1 Uso del simulador

ARM Keil Microvision dispone de opciones para configurar donde se ejecutará la aplicación (Icono *Options for Target*). Seleccione **Debug** y active el uso del simulador (**Use Simulator**). Es necesario que configure el fichero de inicialización (**Initialization File**) para que cargue un script de configuración del microcontrolador. En este caso, seleccione el fichero `simulator.ini` que se encuentra en cada una de las carpetas de ejmplo. Por ejemplo en la carpeta `.\ejemplothreads` del repositorio encontrará el fichero **simulador.ini** con este contenido:

```
MAP 0x40000000,0x400FFFFF read write
MAP 0xE0000000,0xE00FFFFF read write
```

El significado de estas instrucciones es habilitar para el simulador las operaciones de lectura/escritura en las zonas de memoria donde se encuentran los periféricos.

> **Tip**
>
> Cuando se usa el simulador de Keil las operaciones de escritura y lectura de los periféricos no tienen ningún efecto y por tanto no podrá simular el comportamiento hardware de los mismos Todas las operaciones de la capa HAL que actúan sobre periféricos no tendrán ningún efecto. Por ejemplo, si en el código se configura un pin como salida y luego se escribe un valor alto en el mismo, no podrá ver ningún cambio en el estado del pin.

Como podrá ver en el código del programa `main.c` existe compilación condicional para incluir o no el código de configuración del RCC para usar un reloj externo (HSE). Si utiliza el simulador debe desactivar esta opción y usar el reloj interno (HSI) que es el que utiliza el simulador. La pestaña **C/C++(AC6)** permite añadir en `define` etiquetas. Incluya `SIMULATOR` si quiere utilizar el simulator.

4.2 Uso del hardware

Si dispone de una placa con el microcontrolador STM32F429 puede ejecutar el código directamente en el hardware. En este caso debe configurar las opciones del proyecto para que utilice el ST-Link en lugar del simulador.

No defina la variable `SIMULATOR` en las opciones de compilación para que el circuito de RCC se configure adecuadamente.

4.3 Depuración de aplicaciones usando CMSIS-RTOS V2

La depuración de las aplicaciones se debe realizar combinando el uso de puntos de ruptura y de la aplicación `RTX RTOS view` disponible en el menu `View->Watch Windows->RTX RTOS`. Esta permite ver el estado en el que se encuentran los diferentes objetos del sistema operativo cuando el procesador pausa su ejecución. Herramientas com-

plementarias para entender el funcionamiento de una aplicación son: `Logyc Analyzer`, `Performance Analyzer`, `System Analyzer`, `Event Recorder`. `Event Statistics` y `Symbols Window`

4.4 Symbols Window

La opción **Symbols Window** permite visualizar y explorar todos los símbolos definidos en el proyecto, incluyendo variables globales, variables estáticas, funciones y direcciones de registros . Esta ventana es útil para depuración y análisis en tiempo real.

- Muestra una lista jerárquica de todos los símbolos disponibles en el programa cargado.
- Permite buscar y filtrar símbolos por nombre.
- Muestra la dirección y el valor actual de cada símbolo durante la sesión de depuración.
- Facilita el arrastre de variables a otras ventanas de análisis, como el Watch Window o el Logic Analyzer.
- Permite examinar variables optimizadas si están disponibles en la tabla de símbolos.
- Si un símbolo no aparece, verifique la configuración de optimización del compilador y el ámbito de la variable.

Para utilizarlo:

1. Iniciar una sesión de depuración.
2. Abrir la ventana desde el menú: *View* → *Symbol Window*.
3. Buscar el símbolo deseado utilizando el campo de filtro.
4. Arrastrar el símbolo a la ventana de Watch o `Logic Analyzer` para su monitorización.

4.5 Logic Analyzer

Permite visualizar la evolución temporal de variables que sean globales a la aplicación, el contenido de posiciones de memoria, etc. Se puede configurar el rango de valores y es muy apropiado para comparar visualmente la evolución de la aplicación software a través del seguimiento de variables. Para agregar señales al `Logic Analyzer` puede arrastrarlas de la ventana de símbolos o escribir el nombre de la misma.

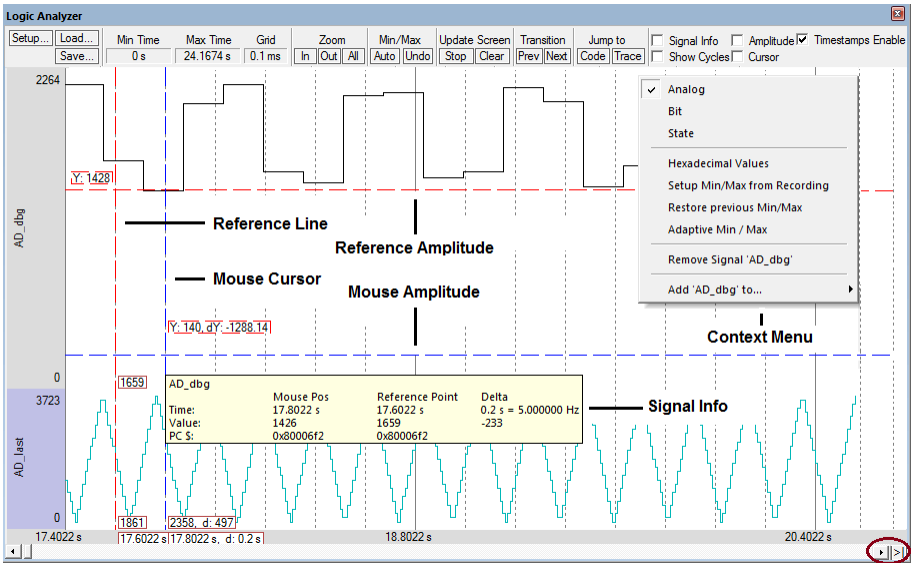


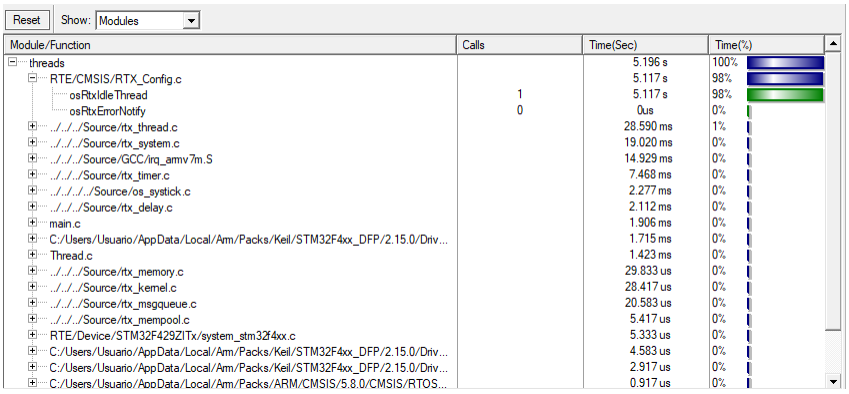
Fig. 4.1: Analizador lógico de ARM Keil Microvision.

4.6 Performance Analyzer

> **Warning**
>
> Solo esta disponible en Simulación porque el ST-LINK no lo soporta

Permite conocer el porcentaje de tiempo utilizado por cada porción del código de nuestra aplicación. Para utilizarlo:

1. Iniciar una sesión de depuración.
2. Abrir la ventana desde el menú: *View* → *Analysis Windows* → *Performance analyzer*.
3. Se muestra una lista de las diferentes secciones de código.



CHAPTER

FIVE

CÓDIGO EN MAIN.C DE LOS EJEMPLOS

El código del programa principal está detallado a continuación y es el mismo o muy similar para todos los ejemplos:

```
#include "main.h"

#ifdef _RTE_
#include "RTE_Components.h"             // Component selection
#endif
#ifdef RTE_CMSIS_RTOS2                  // when RTE component CMSIS RTOS2 is␣
↪used
#include "cmsis_os2.h"                  // ::CMSIS:RTOS2
#endif

#ifdef RTE_CMSIS_RTOS2_RTX5
/**
  * Override default HAL_GetTick function
  */
uint32_t HAL_GetTick (void) {
  static uint32_t ticks = 0U;
        uint32_t i;

  if (osKernelGetState () == osKernelRunning) {
    return ((uint32_t)osKernelGetTickCount ());
  }

  /* If Kernel is not running wait approximately 1 ms then increment
    and return auxiliary tick counter value */
  for (i = (SystemCoreClock >> 14U); i > 0U; i--) {
    __NOP(); __NOP(); __NOP(); __NOP(); __NOP(); __NOP();
    __NOP(); __NOP(); __NOP(); __NOP(); __NOP(); __NOP();
  }
  return ++ticks;
}

/**
  * Override default HAL_InitTick function
  */
HAL_StatusTypeDef HAL_InitTick(uint32_t TickPriority) {

  UNUSED(TickPriority);

```

(continues on next page)

(continued from previous page)

```
  return HAL_OK;
}
#endif
#include "Timer.h"
/** @addtogroup STM32F4xx_HAL_Examples
  * @{
  */

/** @addtogroup Templates
  * @{
  */

/* Private typedef -----------------------------------------------------------*/
/* Private define ------------------------------------------------------------*/
/* Private macro -------------------------------------------------------------*/
/* Private variables ---------------------------------------------------------*/
/* Private function prototypes -----------------------------------------------*/
static void SystemClock_Config(void);
static void Error_Handler(void);

/* Private functions ---------------------------------------------------------*/
/**
  * @brief  Main program
  * @param  None
  * @retval None
  */

int main(void)
{

  /* STM32F4xx HAL library initialization:
       - Configure the Flash prefetch, Flash preread and Buffer caches
       - Systick timer is configured by default as source of time base, but user
             can eventually implement his proper time base source (a general purpose
             timer for example or other time source), keeping in mind that Time base
             duration should be kept 1ms since PPP_TIMEOUT_VALUEs are defined and
             handled in milliseconds basis.
       - Low Level Initialization
     */
  HAL_Init();

  /* Configure the system clock to 168 MHz */
  SystemClock_Config();
```

(continues on next page)

(continued from previous page)

```
  SystemCoreClockUpdate();

  /* Add your application code here
    */

#ifdef RTE_CMSIS_RTOS2
  /* Initialize CMSIS-RTOS2 */
  osKernelInitialize ();

  /* Create thread functions that start executing,
  Example: osThreadNew(app_main, NULL, NULL); */
  Init_Threads();
  /* Start thread execution */
  osKernelStart();
#endif

  /* Infinite loop */
  while (1)
  {
  }
}

/**
  * @brief  System Clock Configuration
  *         The system Clock is configured as follow :
  *            System Clock source            = PLL (HSE)
  *            SYSCLK(Hz)                     = 168000000
  *            HCLK(Hz)                       = 168000000
  *            AHB Prescaler                  = 1
  *            APB1 Prescaler                 = 4
  *            APB2 Prescaler                 = 2
  *            HSE Frequency(Hz)              = 8000000
  *            PLL_M                          = 25
  *            PLL_N                          = 336
  *            PLL_P                          = 2
  *            PLL_Q                          = 7
  *            VDD(V)                         = 3.3
  *            Main regulator output voltage  = Scale1 mode
  *            Flash Latency(WS)              = 5
  * @param  None
  * @retval None
  */
static void SystemClock_Config(void)
{
  RCC_ClkInitTypeDef RCC_ClkInitStruct;
  RCC_OscInitTypeDef RCC_OscInitStruct;

  /* Enable Power Control clock */
  __HAL_RCC_PWR_CLK_ENABLE();

  /* The voltage scaling allows optimizing the power consumption when the␣
↪device is
```

(continues on next page)

(continued from previous page)

```
132     clocked below the maximum system frequency, to update the voltage scaling␣
    ↪value
133     regarding system frequency refer to product datasheet.  */
134   __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);
135
136   /* Enable HSE Oscillator and activate PLL with HSE as source */
137   RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
138   RCC_OscInitStruct.HSEState = RCC_HSE_ON;
139   RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
140   RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
141   RCC_OscInitStruct.PLL.PLLM = 4;
142   RCC_OscInitStruct.PLL.PLLN = 168;
143   RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
144   RCC_OscInitStruct.PLL.PLLQ = 7;
145   if(HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
146   {
147     /* Initialization Error */
148     Error_Handler();
149   }
150
151   /* Select PLL as system clock source and configure the HCLK, PCLK1 and PCLK2
152     clocks dividers */
153   RCC_ClkInitStruct.ClockType = (RCC_CLOCKTYPE_SYSCLK | RCC_CLOCKTYPE_HCLK |␣
    ↪RCC_CLOCKTYPE_PCLK1 | RCC_CLOCKTYPE_PCLK2);
154   RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
155   RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
156   RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV4;
157   RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV2;
158   if(HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_5) != HAL_OK)
159   {
160     /* Initialization Error */
161     Error_Handler();
162   }
163
164   /* STM32F405x/407x/415x/417x Revision Z devices: prefetch is supported */
165   if (HAL_GetREVID() == 0x1001)
166   {
167     /* Enable the Flash prefetch */
168     __HAL_FLASH_PREFETCH_BUFFER_ENABLE();
169   }
170 }
171
172 /**
173   * @brief  This function is executed in case of error occurrence.
174   * @param  None
175   * @retval None
176   */
177 static void Error_Handler(void)
178 {
179   /* User may add here some code to deal with this error */
180   while(1)
181   {
```

(continues on next page)

(continued from previous page)

```
182   }
183 }
```

5.1 Algunos detalles importantes

1. Al utilizar el sistema operativo CMSIS-RTOS V2 se ha definido RTE_CMSIS_RTOS2_RTX5. Esto supone incluir dos funciones que anulan las definidas anteriormente. Estas funciones son `HAL_GetTick` y `HAL_InitTick`. La primera tiene dos comportamientos diferentes. Si el sistema operativo esta en ejecución devuelve el valor retornado por `osKernelGetTickCount` que indica el número de ticks que han pasado desde que se arranco el SO. * Si **no** se ha arrancado el SO la función produce un retardo de aproximadamente 1ms e incrementa la variable estática *ticks*. `Hal_GetTick` se utiliza por las librerías HAL de STM para controlar timeouts en la gestión de los periféricos. La segunda función, `HAL_InitTick`, es usada por la capa HAL para programar un timer HW que proporcione una interrupción cada 1ms. Por defecto este timer es el SysTick timer. Cuando no se usa el sistema operativo esta función realiza las operaciones del código descrito en *stm32f4xx_hal.c* Al usar el SO esta función se substituye por la definida en el `main.c`, que no realiza ninguna operación. Es el código del sistema operativo quien se encarga de inicializar el SysTick (os_systick.c).

2. Despues del código de inicialización de la libraría HAL y de la configuración del RCC del micro (que por cierto utiliza la función HAL_GetTick) se procede a ejecutar la siguiente porción de código:

```
#ifdef RTE_CMSIS_RTOS2
  /* Initialize CMSIS-RTOS2 */
  osKernelInitialize ();

  /* Create thread functions that start executing,
  Example: osThreadNew(app_main, NULL, NULL); */
  Init_Threads();
  /* Start thread execution */
  osKernelStart();
#endif

  /* Infinite loop */
  while (1)
  {
  }
```

La función `osKernelInitialize` inicializa el Sistema Operativo. La función `init_Threads` contiene el código de creación de los recursos necesarios en este ejemplo, y `osKernelStart` comienza la ejecución de los diferentes objetos del SO sin retornar. Eso quiere decir que la porción del código while es código muerto.

3. Toda la inicialización del hardware se debe hacer antes de lanzar la ejecución del SO. Puede incluirse en el código específico de cada thread, en funciones que se ejecuten entre osKernelInitialize y osKernelStart, o en funciones que se incluyan antes de llamar a osKernelInitialize. En cualquier caso debe tener cuidado con el mecanismo de retardo que utiliza en cada parte de la aplicación.

> **Note**
>
> No se recomienda el uso de la función HAL_Delay cuando se esta ejecutando el SO porque la función se puede quedar bloqueada.

CHAPTER

SIX

USO BÁSICO DE UN THREAD EN CMSIS RTOS V2

Esta sección describe el funcionamiento de un programa (**ejemplothread**) en C que utiliza CMSIS RTOS v2 y la biblioteca HAL de STM32 para controlar un LED mediante un hilo.

6.1 Descripción General de ejemplothread

El programa crea un hilo que maneja un LED conectado al pin PB0 del microcontrolador STM32F4.El hilo alterna el estado del LED con una frecuencia configurable, utilizando funciones del sistema operativo en tiempo real (RTOS) y la biblioteca HAL para la configuración y manipulación de los pines GPIO.

6.2 Estructura de Datos

Se define una estructura llamada `mygpio_pin` que encapsula toda la información necesaria para controlar un LED:

- `GPIO_InitTypeDef pin`: configuración del pin (modo, velocidad, tipo de salida).
- `GPIO_TypeDef *port`: puerto GPIO al que pertenece el pin.
- `int delay`: retardo en ms entre cada cambio de estado del LED.
- `uint8_t counter`: contador que se alterna en cada iteración del hilo.

Esta estructura permite pasar todos los parámetros necesarios a la función del hilo de forma organizada.

6.3 Inicialización de los hilos

La función `Init_Thread` realiza las siguientes tareas:

1. Habilita el reloj del puerto GPIOB.
2. Configura `mygpio_pin` para el pin PB0.
3. Crea un hilo con `osThreadNew`, que ejecuta la función `Thread` pasándole una estructura `mygpio_pin` para el pin PB0.

6.4 Función del hilo

La función `Thread` realiza lo siguiente:

1. Inicializa el pin GPIO usando `HAL_GPIO_Init`.
2. Entra en un bucle infinito donde: - Alterna el valor del contador con ~`counter`. - Cambia el estado del pin con `HAL_GPIO_TogglePin`. - Espera el tiempo definido en `delay` usando `osDelay`.

Esto provoca que el LED conectado al pin correspondiente parpadee con una frecuencia que es configurable.

6.5 Uso de HAL y CMSIS RTOS

- **HAL (Hardware Abstraction Layer)**: se utiliza para configurar e inicializar los pines GPIO de forma sencilla y portable.
- **CMSIS RTOS v2**: proporciona las funciones para crear y gestionar hilos, como `osThreadNew` y `osDelay`.

6.6 Código Fuente

```
#include "cmsis_os2.h"
#include "stm32f4xx_hal.h"
#include <stdlib.h>

osThreadId_t tid_Thread;

GPIO_InitTypeDef led_ld1 = {
    .Pin = GPIO_PIN_0,
    .Mode = GPIO_MODE_OUTPUT_PP,
    .Pull = GPIO_NOPULL,
    .Speed = GPIO_SPEED_FREQ_LOW
};



typedef struct {
    GPIO_InitTypeDef pin;
    GPIO_TypeDef *port;
    int delay;
    uint8_t counter;
} mygpio_pin;

mygpio_pin pinB0;


int Init_Thread(void) {
    __HAL_RCC_GPIOB_CLK_ENABLE();

    pinB0.pin = led_ld1;
    pinB0.port = GPIOB;
    pinB0.delay = 15;
    pinB0.counter = 1;
    tid_Thread = osThreadNew(Thread, (void *)&pinB0, NULL);
    if (tid_Thread == NULL) return -1;

    return 0;
}

void Thread(void *argument) {
    mygpio_pin *gpio = (mygpio_pin *)argument;
    HAL_GPIO_Init(gpio->port, &(gpio->pin));
    while (1) {
        gpio->counter++;
        HAL_GPIO_TogglePin(gpio->port, gpio->pin.Pin);
```

(continues on next page)

(continued from previous page)

```
        osDelay(gpio->delay);
    }
}
```

6.7 Dependencias

- Librería HAL de STM32.
- CMSIS RTOS v2.

6.8 Preguntas y respuestas sobre ejemplothread

Esta sección contiene una serie de preguntas con sus respectivas respuestas sobre el funcionamiento del código que utiliza CMSIS RTOS v2 para controlar LEDs en una placa STM32.

6.8.1 ¿Qué hace este código?

Este código crea un hilo (thread) que controla un LED conectado al pin PB0 de una placa STM32F429. El hilo alterna el estado del LED (encendido/apagado) con una frecuencia determinada utilizando funciones del sistema operativo en tiempo real CMSIS RTOS v2. Dentro del código del Thread se realiza un casting al tipo de estructura que se utiliza en el ejemplo

6.8.2 ¿Qué es la estructura *mygpio_pin*?

Es una estructura de datos que encapsula la información necesaria para controlar un pin GPIO en este ejemplo:

- `pin`: configuración del pin (tipo, velocidad, modo).
- `port`: puerto GPIO al que pertenece el pin (por ejemplo, GPIOB).
- `delay`: retardo en ms entre cada cambio de estado (toggle).
- `counter`: variable auxiliar que cuenta la cantidad de veces que se ha realizado el toggle.

6.8.3 ¿Cómo se inicializa el hilo?

La función `Init_Thread()` habilita el reloj del puerto GPIOB, rellena los parámetros de la estructura y crea un hilo con la función `osThreadNew()`, pasando como argumento la estructura `mygpio_pin` correspondiente a cada LED.

6.8.4 ¿Qué hace la función *Thread()*?

La función `Thread(void *argument)` se encarga de:

1. Inicializar el pin GPIO usando `HAL_GPIO_Init`.
2. Ejecutar un bucle infinito donde: - Se incrementa el valor de `counter`. - Se cambia el estado del LED con `HAL_GPIO_TogglePin`. - Se espera el tiempo definido en `delay` usando `osDelay`.

6.8.5 ¿Qué significa *osDelay()*?

Es una función del RTOS que suspende la ejecución del hilo actual durante un número determinado de ms. Esto permite que otros hilos se ejecuten mientras tanto. `osDelay` tiene como parámetro el número de ticks que la tarea estará bloqueada. El número de ticks por segundo se define en el archivo `RTX_Config.h` (parámetro `Kernel Tick Frequency [Hz]`). En este ejemplo se ha configurado a 1000, por lo que un tick equivale a 1 ms.

6.8.6 ¿Qué pasa si *osThreadNew()* devuelve NULL?

Significa que no se pudo crear el hilo. En ese caso, la función `Init_Thread()` devuelve -1 como señal de error. Si el programa principal que llama a esta función no comprueba el retorno no hay ningún control de errores.

6.8.7 ¿Qué ficheros de cabecera se utilizan?

- `cmsis_os2.h`: para funciones del sistema operativo en tiempo real.
- `stm32f4xx_hal.h`: para funciones de acceso a hardware (HAL).
- `stdlib.h`: para funciones estándar de C que en este caso no se están incluyendo en el código.

6.8.8 Determine la carga de la CPU en esta aplicación

Para determinar la carga que supone la ejecución del thread para la CPU se puede utilizar la utilidad de `Performance Analyzer` en modo simulación. La carga de CPU obtenida es insignificante. Si se cambia en la estructura de datos el campo `delay` por 0 la carga del Thread pasa a ser del 19%.

CHAPTER

SEVEN

USO BÁSICO DE THREADS EN CMSIS RTOS V2

Esta sección describe el funcionamiento de un programa en C que utiliza CMSIS RTOS v2 y la biblioteca HAL de STM32 para controlar dos LEDs mediante hilos concurrentes.

7.1 Descripción General

El programa crea dos hilos que controlan dos LEDs conectados a los pines PB0 y PB7 del microcontrolador STM32F429. Cada hilo alterna el estado de su LED con una frecuencia distinta, utilizando funciones del sistema operativo en tiempo real (RTOS) y la biblioteca HAL para la configuración y manipulación de los pines GPIO.

7.2 Estructura mygpio_pin

Se define una estructura llamada `mygpio_pin` que encapsula toda la información necesaria para controlar un LED:

- `GPIO_InitTypeDef pin`: configuración del pin (modo, velocidad, tipo de salida).
- `GPIO_TypeDef *port`: puerto GPIO al que pertenece el pin.
- `int delay`: retardo en ms entre cada cambio de estado del LED.
- `uint8_t counter`: contador que se incrementa en cada iteración del hilo.

Esta estructura permite pasar todos los parámetros necesarios a la función del hilo de forma organizada.

7.3 Inicialización de los threads

La función `Init_Thread` realiza las siguientes tareas:

1. Habilita el reloj del puerto GPIOB.
2. Configura dos instancias de `mygpio_pin` para los pines PB0 y PB7.
3. Crea dos hilos con `osThreadNew`, cada uno ejecutando la función `Thread` con una instancia diferente de `mygpio_pin`.

Cada hilo se ejecuta de forma independiente y controla su propio LED.

7.4 Thread()

La función `Thread` realiza lo siguiente:

1. Inicializa el pin GPIO usando `HAL_GPIO_Init`.
2. Entra en un bucle infinito donde: - Alterna el valor del contador con `~counter`. - Cambia el estado del pin con `HAL_GPIO_TogglePin`. - Espera el tiempo definido en `delay` usando `osDelay`.

Esto provoca que el LED conectado al pin correspondiente parpadee con una frecuencia determinada.

7.5 HAL y CMSIS RTOS

- **HAL (Hardware Abstraction Layer)**: se utiliza para configurar e inicializar los pines GPIO de forma sencilla y portable.
- **CMSIS RTOS v2**: proporciona las funciones para crear y gestionar hilos, como `osThreadNew` y `osDelay`.

7.6 Código

```
#include "cmsis_os2.h"
#include "stm32f4xx_hal.h"
#include <stdlib.h>

osThreadId_t tid_Thread;

GPIO_InitTypeDef led_ld1 = {
    .Pin = GPIO_PIN_0,
    .Mode = GPIO_MODE_OUTPUT_PP,
    .Pull = GPIO_NOPULL,
    .Speed = GPIO_SPEED_FREQ_LOW
};

GPIO_InitTypeDef led_ld2 = {
    .Pin = GPIO_PIN_7,
    .Mode = GPIO_MODE_OUTPUT_PP,
    .Pull = GPIO_NOPULL,
    .Speed = GPIO_SPEED_FREQ_LOW
};

typedef struct {
    GPIO_InitTypeDef pin;
    GPIO_TypeDef *port;
    int delay;
    uint8_t counter;
} mygpio_pin;

mygpio_pin pinB0;
mygpio_pin pinB7;

int Init_Thread(void) {
    __HAL_RCC_GPIOB_CLK_ENABLE();

    pinB0.pin = led_ld1;
    pinB0.port = GPIOB;
    pinB0.delay = 15;
    pinB0.counter = 1;
    tid_Thread = osThreadNew(Thread, (void *)&pinB0, NULL);
    if (tid_Thread == NULL) return -1;

    pinB7.pin = led_ld2;
```

(continues on next page)

(continued from previous page)

```
    pinB7.port = GPIOB;
    pinB7.delay = 10;
    pinB7.counter = 0;
    tid_Thread = osThreadNew(Thread, (void *)&pinB7, NULL);
    if (tid_Thread == NULL) return -1;

    return 0;
}

void Thread(void *argument) {
    mygpio_pin *gpio = (mygpio_pin *)argument;
    HAL_GPIO_Init(gpio->port, &(gpio->pin));
    while (1) {
        gpio->counter++;
        HAL_GPIO_TogglePin(gpio->port, gpio->pin.Pin);
        osDelay(gpio->delay);
    }
}
```

7.7 Dependencias software

- Librería HAL de STM32.
- CMSIS RTOS v2.

7.8 Preguntas y respuestas sobre ejemplothreads

Esta sección contiene una serie de preguntas con sus respectivas respuestas sobre el funcionamiento del código que utiliza CMSIS RTOS v2 para controlar LEDs en una placa STM32.

7.8.1 ¿Qué función hace este código?

Este código crea dos hilos (threads) que controlan dos LEDs conectados a los pines PB0 y PB7 de una placa STM32F4. Cada hilo alterna el estado del LED (encendido/apagado) con una frecuencia determinada utilizando funciones del sistema operativo en tiempo real CMSIS RTOS v2. Es importante entender que el mismo código (funcion Thread) es ejecutado por dos hilos diferentes, cada uno con sus propios parámetros, que se reciben en el argumento de la función. Es de tipo `void` para poder pasar cualquier tipo de estructura como argumento. Dentro del código del Thread se realiza un casting al tipo de estructura que se utiliza en el ejemplo

7.8.2 ¿Qué función tiene *mygpio_pin*?

Es una estructura de datos que encapsula la información necesaria para controlar un pin GPIO en este ejemplo:

- `pin`: configuración del pin (tipo, velocidad, modo).
- `port`: puerto GPIO al que pertenece el pin (por ejemplo, GPIOB).
- `delay`: retardo en ms entre cada cambio de estado (toggle).
- `counter`: variable auxiliar que cuenta la cantidad de veces que se ha realizado el toggle.

7.8.3 ¿Cómo se inicializan los hilos?

La función `Init_Thread()` habilita el reloj del puerto GPIOB, configura los parámetros de cada LED y crea dos hilos con `osThreadNew()`, pasando como argumento la estructura `mygpio_pin` correspondiente a cada LED.

7.8.4 ¿Qué función tieneº *Thread()*?

La función `Thread(void *argument)` es ejecutada por cada hilo. Dentro de ella:

1. Se inicializa el pin GPIO usando `HAL_GPIO_Init`.
2. Se entra en un bucle infinito donde: - Se alterna el valor de `counter`. - Se cambia el estado del LED con `HAL_GPIO_TogglePin`. - Se espera el tiempo definido en `delay` usando `osDelay`.

7.8.5 ¿Se ejecutan los hilos al mismo tiempo?

CMSIS RTOS v2 permite la ejecución concurrente, que no simultanea, de múltiples hilos. El scheduler del sistema operativo se encarga de asignar tiempo de CPU a cada hilo según su estado y prioridad.

7.8.6 ¿Qué función tiene `osDelay()`

Es una función del RTOS que suspende la ejecución del hilo actual durante un número determinado de ticks. Esto permite que otros hilos se ejecuten mientras tanto. `osDelay` tiene como parametro el número de ticks que la tarea estará bloqueada. El número de ticks por segundo se define en el archivo `RTX_Config.h` (parámetro `Kernel Tick Frequency [Hz]`). En este ejemplo se ha configurado a 1000, por lo que un tick equivale a 1 ms.

7.8.7 ¿Qué pasa si *osThreadNew()* retorna NULL?

Significa que no se pudo crear el hilo. En ese caso, la función `Init_Thread()` devuelve *-1* como señal de error.

7.8.8 ¿Qué includes se utilizan?

- `cmsis_os2.h`: para funciones del sistema operativo en tiempo real.
- `stm32f4xx_hal.h`: para funciones de acceso a hardware (HAL).
- `stdlib.h`: para funciones estándar de C.

7.8.9 ¿Cuanto vale el valor del tick es esta aplicación?

El fichero de configuración del sistema operativo tal y como indica la figura tiene configurado un tick de 1ms.

7.8.10 ¿Que es el thread Idle? ¿Qué tamaño de stack tiene? ¿Y otro thread? ¿Que tamaño de stack usa?

El thread idle esta definido en el fichero RTX_Config.c y es un thread que se ejecuta cuando el sistema operativo no tiene ninguna otro thread que ejecutar. Tiene un tamaño de `stack` de 512 bytes. Cualquier otro thread se configura para tener un tamaño de stack de 3072 bytes (3KBytes). Una reflexión interesante es cuantos threads se pueden crear en una aplicación.

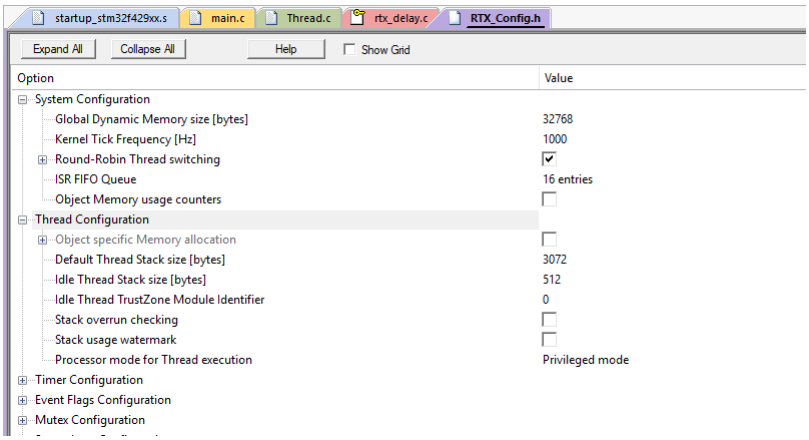


Fig. 7.1: Configuración del sistema operativo.

CHAPTER

EIGHT

USO BÁSICO DE THREADS Y COLAS EN CMSIS RTOS V2

Este documento describe el funcionamiento de un programa en C que utiliza CMSIS RTOS v2 y la biblioteca HAL de STM32 para controlar dos LEDs mediante hilos concurrentes que se comunican con colas.

8.1 Descripción General

El programa crea dos hilos, denominados `Producer` y `Consumer`. El hilo `Producer` se encarga de introducir datos en la cola que tiene el identificador `id_MsgQueue`. El numero total de mensajes que se introducen en la cola en cada iteración del bucle `while` es 32 con un tiempo de retardo entre operaciones de escritura que es variable El `Consumer` se encarga de extraer los datos de cola y de actuar sobre los leds en función del valor leido de la cola.

8.2 Estructuras de Datos

Se define una estructura llamada `mygpio_pin` que encapsula toda la información necesaria para controlar un LED:

- `GPIO_InitTypeDef pin`: configuración del pin (modo, velocidad, tipo de salida).
- `GPIO_TypeDef *port`: puerto GPIO al que pertenece el pin.

Esta estructura permite pasar todos los parámetros necesarios a la función del hilo de forma organizada.

8.3 Creación Hilos

La función `Init_Thread` realiza las siguientes operaciones:

1. Crea una cola con `osMessageQueueNew` para almacenar hasta 16 mensajes cada uno de tamaño un `uint8_t`, es decir, un solo byte.
2. Crea un hilo `Consumer` con `osThreadNew`, ejecutando la función `Consumer`.
3. Crea un hilo `Producer` con `osThreadNew`, ejecutando la función `Producer`.

8.4 Producer

La función `Producer(void *argument)` realiza las siguientes operaciones: 1. De manera continua en un bucle infinito inserta en cola en cada iteración del bucle while 32 mensajes de 1 byte con el valor de la variable index. 2. Los mensajes se introducen con un retardo que varia desde 100ms hasta 400ms 3. La función osMessageQueuePut introduce los mensajes con timeout 0, lo cúal implica que si no hay sitio en la cola el mensaje no se podrá guardar.

8.5 Consumer

La función `Consumer(void *argument)` realiza las siguientes operaciones: 1. De manera continua en un bucle infinito extrae de la cola los mensajes introducidos por el hilo `Producer`. 2. Si se saca un valor de la cola se procede a encender o apagar los leds en función del valor leido. 3. La variable errors_or_timeout cuenta el número de veces que no se ha podido leer un mensaje de la cola, ya sea por timeout o porque la cola está vacía.

8.6 HAL-CMSIS RTOS

- **HAL (Hardware Abstraction Layer)**: se utiliza para configurar e inicializar los pines GPIO de forma sencilla y portable.
- **CMSIS RTOS v2**: proporciona las funciones para crear y gestionar hilos, como `osThreadNew` y `osDelay`, y las funciones para gestionar las colas.

8.7 Fuente

```
#include "cmsis_os2.h"                          // CMSIS RTOS header file
#include "stm32f4xx_hal.h"
#include <string.h>
#include <stdlib.h>


osThreadId_t tid_Thread;                        // thread id
osMessageQueueId_t id_MsgQueue;
int Init_Thread (void);
void Producer (void *argument);                   // thread function producing data
void Consumer (void *argument);                   // thread function consuming data
int qsize=0;
uint16_t h=0;
uint8_t i=0;

typedef struct  {
        GPIO_InitTypeDef pin;
                GPIO_TypeDef *port;
} mygpio_pin;

mygpio_pin pinB0;
mygpio_pin pinB7;

int Init_Thread (void) {

        id_MsgQueue = osMessageQueueNew(16, sizeof(uint8_t), NULL);


tid_Thread = osThreadNew(Producer, NULL, NULL);
if (tid_Thread == NULL) {
        return(-1);
}

        tid_Thread = osThreadNew(Consumer, NULL, NULL);
if (tid_Thread == NULL) {
```

(continues on next page)

(continued from previous page)

```
        return(-1);
}

return(0);
}

void Producer (void *argument) {
        uint8_t index=0;
        osStatus_t status;
while (1) {
                for( h=1; h<5; h++){
                        for( i=0; i< 8; i++){
                                status=osMessageQueuePut(id_MsgQueue, &index, 0U, 0U);
                                index++;
                                osDelay(h*100);
                        }
                }
        }
}
void Consumer (void *argument) {
        uint8_t val=0;
        osStatus_t status;
        int errors_or_timeouts=0;
        GPIO_InitTypeDef led_ld1 = {
                .Pin = GPIO_PIN_0,
                .Mode = GPIO_MODE_OUTPUT_PP,
                .Pull = GPIO_NOPULL,
                .Speed = GPIO_SPEED_FREQ_LOW
        };
        GPIO_InitTypeDef led_ld2 = {
                .Pin = GPIO_PIN_7,
                .Mode = GPIO_MODE_OUTPUT_PP,
                .Pull = GPIO_NOPULL,
                .Speed = GPIO_SPEED_FREQ_LOW
        };
        __HAL_RCC_GPIOB_CLK_ENABLE();

        HAL_GPIO_Init(GPIOB, &led_ld1);

        HAL_GPIO_Init(GPIOB, &led_ld2);


while (1) {
        qsize=osMessageQueueGetCount (id_MsgQueue);
                status = osMessageQueueGet(id_MsgQueue, &val, NULL, 10U);   // wait for␣
↪message
                if (status == osOK){
                        HAL_GPIO_WritePin(GPIOB,led_ld1.Pin,(GPIO_PinState) val&0x01);
                        HAL_GPIO_WritePin(GPIOB,led_ld2.Pin,(GPIO_PinState)(val&0x02)>>
↪1);

                }
```

(continues on next page)

(continued from previous page)

```
                else {
                        errors_or_timeouts++;
                }
                osDelay(250); //This delay is to simulate an operation that needs 101ms␣
↪to complete

}
}
```

8.8 Dependencias de software

- Librería HAL de STM32.
- CMSIS RTOS v2.

8.9 Preguntas y respuestas sobre ejemplothreads-queues

Esta sección contiene una serie de preguntas con sus respectivas respuestas sobre el funcionamiento del código que utiliza CMSIS RTOS v2 para controlar LEDs en una placa STM32.

8.9.1 ¿Cuál es el propósito de la cola de mensajes *id_MsgQueue* en esta aplicación?

La cola de mensajes *id_MsgQueue* actúa como un canal de comunicación y sincronización entre los hilos *Producer* y *Consumer*. Permite que el hilo productor envíe datos (índices) al consumidor de forma segura y sincronizada. Al definir una cola con capacidad para 16 elementos de tipo *uint8_t*, se establece un buffer temporal que desacopla la producción y el consumo de datos.

8.9.2 ¿Qué función cumple el bucle anidado en el hilo *Producer*?

El bucle anidado en *Producer* genera una secuencia de valores que se colocan en la cola de mensajes. El bucle externo recorre *h* de 1 a 4, y el interno recorre *i* de 0 a 7. En cada iteración, se coloca un valor en la cola (*index*) y se incrementa. El retardo *osDelay(h*100)* introduce una variabilidad en el tiempo entre envíos, oscilando entre 100 ms y 400 ms. Esto simula diferentes tasas de producción de datos.

8.9.3 ¿Cuanto tiempo tarda en llenarse la cola de mensajes *id_MsgQueue*?

En la cola se introducen 32 mensajes en cada ciclo completo de los bucles anidados (8 mensajes por cada uno de los 4 valores de *h*) pero el Thread Consumer extrae mensajes cada 250ms en el caso de que existan. Por tanto la cola nunca llega a llenarse. Intente calcular cual sería el numero máximo de mensajes que se pueden acumular en la cola.

8.9.4 ¿cuanto vale la variable errors_or_timeouts despues de 1 minuto de ejecución del código?

Vale 0 porque no se produce dicha condición nunca.

> **Note**
>
> Challenge: Modifique el código del hilo `Producer` para que la variable errors_or_timeouts no valga cero.

CHAPTER

NINE

USO BÁSICO DE THREADS Y FLAGS EN CMSIS RTOS V2

Este documento describe el funcionamiento de un programa en C que utiliza CMSIS RTOS v2 y la biblioteca HAL de STM32 para controlar dos LEDs mediante hilos concurrentes que se sincronizan con flags.

9.1 Descripción

El programa crea dos hilos, denominados `Producer` y `Consumer`. El hilo `Producer` es un bucle infinito que se encarga de activar flags para el thread `Consumer`. Este en función de los flags activados ejecuta unas acciones u otras.

Se define una estructura llamada `mygpio_pin` que encapsula toda la información necesaria para controlar un LED:

- `GPIO_InitTypeDef pin`: configuración del pin (modo, velocidad, tipo de salida).
- `GPIO_TypeDef *port`: puerto GPIO al que pertenece el pin.

Esta estructura permite pasar todos los parámetros necesarios a la función del hilo de forma organizada.

La función `Init_Thread` realiza las siguientes operaciones:

1. Crea un hilo `Consumer` con `osThreadNew`, ejecutando la función `Consumer`. 3. Crea un hilo `Producer` con `osThreadNew`, ejecutando la función `Producer`.

9.2 Función del hilo Producer

La función `Producer(void *argument)` realiza las siguientes operaciones: 1. De manera continua en un bucle infinito señaliza flags en el thread `Consumer`. 2. Los flags activados son el 0x0001 y en 0x0002.

9.3 Función del hilo Consumer

La función `Consumer(void *argument)` realiza las siguientes operaciones: 1. Inticializa dos pines del GPIO en el puerto B. 2. Espera de manera infinita (`osWaitForEver`) a que cualquiera (`osFlagsWaitAny`) de los flags (`0` o `1`, en hexadecimal `0x03`) se activen. 3. Si el flag activado es el `0` se hace un toggle en el pin 0 del GPIOB. Si el activado es el 1 se hace el toggle en el pin 7 del GPIOB. 4. Si se produce otra condición se incrementa la variable `errors`.

9.4 HAL y CMSIS RTOS

- **HAL (Hardware Abstraction Layer)**: se utiliza para configurar e inicializar los pines GPIO de forma sencilla y portable.
- **CMSIS RTOS v2**: proporciona las funciones para crear y gestionar hilos, como `osThreadNew` y `osDelay`, y la funciones de gestion de los flags.

9.5 Código aplicación

```
#include "cmsis_os2.h"                          // CMSIS RTOS header file
#include "stm32f4xx_hal.h"
#include <string.h>
#include <stdlib.h>


osThreadId_t tid_Thread_producer;                       // thread id
osThreadId_t tid_Thread_consumer;
int Init_Thread (void);
void Producer (void *argument);                  // thread function producing data
void Consumer (void *argument);                  // thread function consuming data
int qsize=0;
uint8_t a=0;
uint8_t b=0;

typedef struct  {
        GPIO_InitTypeDef pin;
                GPIO_TypeDef *port;
} mygpio_pin;

mygpio_pin pinB0;
mygpio_pin pinB7;

int Init_Thread (void) {




tid_Thread_producer = osThreadNew(Producer, NULL, NULL);
if (tid_Thread_producer == NULL) {
        return(-1);
}

        tid_Thread_consumer = osThreadNew(Consumer, NULL, NULL);
if (tid_Thread_consumer == NULL) {
        return(-1);
}

return(0);
}

void Producer (void *argument) {

        uint32_t status;
while (1) {


        status= osThreadFlagsSet(tid_Thread_consumer,0x0001);
        osDelay(1000);
        status= osThreadFlagsSet(tid_Thread_consumer,0x0002);
        osDelay(1000);
```

(continues on next page)

(continued from previous page)

```
        }
}
void Consumer (void *argument) {
        uint8_t val=0;
        uint32_t status;
        int errors=0;
        GPIO_InitTypeDef led_ld1 = {
                .Pin = GPIO_PIN_0,
                .Mode = GPIO_MODE_OUTPUT_PP,
                .Pull = GPIO_NOPULL,
                .Speed = GPIO_SPEED_FREQ_LOW
        };
        GPIO_InitTypeDef led_ld2 = {
                .Pin = GPIO_PIN_7,
                .Mode = GPIO_MODE_OUTPUT_PP,
                .Pull = GPIO_NOPULL,
                .Speed = GPIO_SPEED_FREQ_LOW
        };
        __HAL_RCC_GPIOB_CLK_ENABLE();

        HAL_GPIO_Init(GPIOB, &led_ld1);

        HAL_GPIO_Init(GPIOB, &led_ld2);


while (1) {
        status=osThreadFlagsWait(0x3,osFlagsWaitAny,osWaitForever);
                switch (status){
                        case 1:
                                HAL_GPIO_TogglePin(GPIOB,led_ld1.Pin);
                                a=!a;
                                break;
                case 2:
                                HAL_GPIO_TogglePin(GPIOB,led_ld2.Pin);
                        b=!b;
                                break;
                default:errors++;
                                break;
                }


        }
}
```

9.6 Dependencias de software del ejemplo

- Librería HAL de STM32.
- CMSIS RTOS v2.

9.7 Preguntas y respuestas sobre ejemplothreads-flags

Esta sección contiene una serie de preguntas con sus respectivas respuestas sobre el funcionamiento del código que utiliza CMSIS RTOS v2 para controlar LEDs en una placa STM32.

9.7.1 Se modifica el código del Producer para que envíe ambas señales (0x0001 y 0x0002) de forma casi simultánea, seguido de un delay de 1 segundo:

```
void Producer (void *argument) {
        uint32_t status;
        while (1) {
                status = osThreadFlagsSet(tid_Thread_consumer, 0x0001);
                status = osThreadFlagsSet(tid_Thread_consumer, 0x0002);
                osDelay(1000);
        }
}
```

Analice el comportamiento resultante del sistema y responda:

1. ¿Qué valor tendría la variable status en el Consumer después de osThreadFlagsWait?
2. ¿Cómo afecta esta modificación al parpadeo de los LEDs?

1. Valor de status: La variable status en el Consumer tendría el valor 0x0003 (0x0001 | 0x0002), ya que los flags se acumulan en el sistema CMSIS-RTOS cuando se envían antes de que el thread destino los procese.
2. Efecto en los LEDs: Los LEDs dejarían de parpadear por completo. El switch statement en el Consumer solo maneja explícitamente los casos 1 (0x0001) y 2 (0x0002). Al recibir el valor combinado 3, la ejecución cae en el caso default, donde solo se incrementa la variable errors sin ejecutar ninguna operación de toggle en los GPIOs.

CHAPTER

TEN

USO BÁSICO DE THREADS Y SOFTWARE TIMERS EN CMSIS RTOS V2

Esta sección describe el funcionamiento de un programa en C que utiliza CMSIS RTOS v2 y la biblioteca HAL de STM32 para controlar dos LEDs mediante un hilo y timers software.

10.1 Descripción General

El programa crea un único hilo denominado `Timers`. Este hilo se encarga de configurar los pines B0 y B7 como salida para excitar los LEDs LD1 y LD2. Ademas crea un timer one-shot y otro periodico. El timer one-shot se inicia para que al cabo de 10 segundos se active y en su callback se encienda el led LD1 y se inicie el timer periódico. El timer periódico hace que el led LD2 parpadee cada 500ms.

Se define una estructura llamada `mygpio_pin` que encapsula toda la información necesaria para controlar un LED:

- `GPIO_InitTypeDef pin`: configuración del pin (modo, velocidad, tipo de salida).
- `GPIO_TypeDef *port`: puerto GPIO al que pertenece el pin.

La función `Init_Thread` realiza las siguientes operaciones:

1. Crea un hilo `Timers` con `osThreadNew`.

10.2 Función del Hilo `Timers`

La función `Timers(void *arg)` realiza las siguientes operaciones:

1. Configura los pines GPIO para los LEDs LD1 y LD2.
2. Ejecuta un bucle infinito que en cada iteración expera 1 segundo.

10.3 Uso de HAL y CMSIS RTOS

- **HAL (Hardware Abstraction Layer)**: se utiliza para configurar e inicializar los pines GPIO de forma sencilla y portable.
- **CMSIS RTOS v2**: proporciona las funciones para crear y gestionar hilos, como `osThreadNew` y `osDelay`, y funciones específicas para gestionar timers.

10.4 Código específico

```
#include "cmsis_os2.h"                          // CMSIS RTOS header file
#include "stm32f4xx_hal.h"
#include <string.h>
#include <stdlib.h>


void Init_Threads (void);
void Timers (void*);
GPIO_InitTypeDef led_ld1 = {
                .Pin = GPIO_PIN_0,
                .Mode = GPIO_MODE_OUTPUT_PP,
                .Pull = GPIO_NOPULL,
                .Speed = GPIO_SPEED_FREQ_LOW
        };
        GPIO_InitTypeDef led_ld2 = {
                .Pin = GPIO_PIN_7,
                .Mode = GPIO_MODE_OUTPUT_PP,
                .Pull = GPIO_NOPULL,
                .Speed = GPIO_SPEED_FREQ_LOW
        };


typedef struct  {
        GPIO_InitTypeDef pin;
                GPIO_TypeDef *port;
} mygpio_pin;

mygpio_pin pinB0;
mygpio_pin pinB7;
void Timer1_Callback_1(void *arg);
void Timer1_Callback_2(void *arg);
osTimerId_t timsoft2 ;
void Init_Threads(void){
        osThreadId_t tid_Thread = osThreadNew(Timers, NULL, NULL);
}
void Timers (void* arg) {


        __HAL_RCC_GPIOB_CLK_ENABLE();

        HAL_GPIO_Init(GPIOB, &led_ld1);

        HAL_GPIO_Init(GPIOB, &led_ld2);
        HAL_GPIO_WritePin(GPIOB, led_ld1.Pin, GPIO_PIN_RESET);
        HAL_GPIO_WritePin(GPIOB, led_ld2.Pin, GPIO_PIN_RESET);

        osTimerId_t timsoft1 = osTimerNew(Timer1_Callback_1, osTimerOnce, NULL, NULL);

        osTimerStart(timsoft1,10000);
        timsoft2 = osTimerNew(Timer1_Callback_2, osTimerPeriodic, NULL, NULL);


while(1){
```

(continues on next page)

(continued from previous page)

```
                osDelay(1000);
}
}
void Timer1_Callback_1(void *arg){

                        HAL_GPIO_TogglePin(GPIOB,led_ld1.Pin);
                        osTimerStart(timsoft2, 500);

}

void Timer1_Callback_2(void *arg){

                        HAL_GPIO_TogglePin(GPIOB,led_ld2.Pin);

}
```

10.5 Dependencia

- Librería HAL de STM32.
- CMSIS RTOS v2.

10.6 Preguntas y respuestas sobre ejemplothreads-timers

Esta sección contiene una serie de preguntas con sus respectivas respuestas sobre el funcionamiento del código que utiliza CMSIS RTOS v2 para controlar LEDs en una placa STM32.

10.6.1 ¿Cual es la diferencia fundamental entre un timer periódico otro one-shot?

El timer one-shot dispara la función de callback una sola vez. Es importante indicar que el tiempo empieza a contar desde que el timer es arrancado. Un timer periódico por contra ejecuta la función de callback multiples veces. Es importante hacer notar que la función de arrancar un timer no se puede llamar desde una rutina de atención a la interrupción. La función de arrancar un timer se puede llamar de manera reiterada reiniciando la cuenta de tiempo del mismo.

10.6.2 Los ficheros RTX_config.h y RTX_config.c son generados automáticamente por el entorno de desarrollo. ¿Se pueden modificar?

Sí, se pueden modificar. Estos ficheros contienen configuraciones específicas del sistema operativo en tiempo real (RTOS) RTX, como el número máximo de hilos, la prioridad de los hilos, el tamaño de la pila, entre otros parámetros. Modificar estos archivos permite ajustar el comportamiento del RTOS según las necesidades específicas de la aplicación.

10.6.3 Si se fija un punto de ruptura en la línea 47, ¿qué se espera ver en el `Watch Windows->RTX RTOS`?

1. El hilo en estado running. Además no es el único hilo porque aparece el hilo `osRtxIdleThread` y `osRtxTimerThread`.
2. Se visualiza una cola que es utiliza por el sistema operativo para gestionar eventos internos.

Note

Challenge: Investigue el mecanismo para poder poner su propio código en el thread `osRtxIdleThread`.

Warning

No utilice las funciones de manejo de timers software desde rutinas de atención a la interrupción.