IMPLEMENTAZIONE PWM DIMMER CON OS REAL TIME

Vedere le slides per il pattern “OS real time”. L’idea è avere delle ISR (interrupt service routines) che comunicano con i task dell’OS utilizzando delle primitive di comunicazione interprocesso (nelle slides vengono utilizzati dei segnali). I task sono “pseudo-cooperativi”, nel senso che sono strutturati come task ciclici che si bloccano sull’elemento di comunicazione condiviso con l’ISR corrispondente, e quindi eseguono la loro computazione. Con l’architettura basata su OS realtime sorge il problema di coordinare e sincronizzare la comunicazione tra i task se essi devono condividere informazioni tra di loro, cosa che si verifica nel caso il sistema operativo usi la preemption. Di solito però i sistemi operativi mettono a disposizioni primitive per garantire tale coordinazione/sincronizzazione, come lock mutex, semafori, mailbox...

Nel nostro caso tutto è facilitato dal device configuration tool, che permette di configurare non solo il dispositivo, ma anche l’inclusione nel progetto software di un certo numero di librerie di middleware, incluso il sistema operativo real time open source FreeRTOS. Creiamo un nuovo progetto, al solito (sezione System Core) sotto GPIO per PC13 (pulsante) attiviamo la modalità rising/falling, sotto NVIC attiviamo l’interrupt per EXTI[15:10], sotto (sezione Timers) TIM6 e TIM7 configuriamo i timers con i corretti prescalers, con gli interrupt attivati e in one pulse mode. Infine andiamo nella sezione Middleware e selezioniamo FREERTOS. Nel tab Config Parameters selezioniamo come API le CMSIS v2, nel tab Task and Queues creiamo tre tasks e tre code.

Tasks:

- tim6Task, entry function: StartTIM6Task
- tim7Task, entry function: StartTIM7Task
- buttonTask, entry function StartButtonTask

I tre task hanno tutti e tre le stesse caratteristiche (priorità osPriorityNormal, stack 128…)

Code:

- tim6_queue
- tim7_queue
- button_queue

Le tre code hanno le stesse caratteristiche; le sovradimensioniamo a 16 e le diamo allocazione dinamica per nessun particolare motivo. L’item size è di tipo bool. Usiamo le code anziché i segnali per far comunicare le ISR con i task semplicemente perché il device configuration tool ci permette di crearle con facilità, ma l’uso delle code è ovviamente eccessivo.

Dal momento che la sorgente di default (interna) per il SysTick non va bene con FreeRTOS, occorre selezionare un timer come sorgente per il SysTick. Occorre quindi andare nella sezione System Core e selezionare SYS, quindi scegliere come Timebase Source un timer qualunque, ad esempio TIM1.

A questo punto il codice generato sarà circa così:

main.c:

```
...
/* Includes ------------------------------------------------------------------*/
#include "main.h"
#include "cmsis_os.h"
...
/* Private variables ---------------------------------------------------------*/
...
/* Definitions for tim6Task */
osThreadId_t tim6TaskHandle;
const osThreadAttr_t tim6Task_attributes = { ,,,
```

```
/* Definitions for tim7Task */
osThreadId_t tim7TaskHandle;
const osThreadAttr_t tim7Task_attributes = { ,,,
/* Definitions for buttonTask */
osThreadId_t buttonTaskHandle;
const osThreadAttr_t buttonTask_attributes = { ...
/* Definitions for tim6_queue */
osMessageQueueId_t tim6_queueHandle;
const osMessageQueueAttr_t tim6_queue_attributes = { ...
/* Definitions for tim7_queue */
osMessageQueueId_t tim7_queueHandle;
const osMessageQueueAttr_t tim7_queue_attributes = { ...
/* Definitions for button_queue */
osMessageQueueId_t button_queueHandle;
const osMessageQueueAttr_t button_queue_attributes = { ...
...
/* Private function prototypes -----------------------------------------------*/
void StartTIM6Task(void *argument);
void StartTIM7Task(void *argument);
void StartButtonTask(void *argument);

int main(void)
{
  ...
  /* Init scheduler */
  osKernelInitialize();


  ...
  /* Create the queue(s) */
  /* creation of tim6_queue */
  tim6_queueHandle = osMessageQueueNew (16, sizeof(bool),
&tim6_queue_attributes);

  /* creation of tim7_queue */
  tim7_queueHandle = osMessageQueueNew (16, sizeof(bool),
&tim7_queue_attributes);

  /* creation of button_queue */
  button_queueHandle = osMessageQueueNew (16, sizeof(bool),
&button_queue_attributes);

  ...

  /* Create the thread(s) */
  /* creation of tim6Task */
  tim6TaskHandle = osThreadNew(StartTIM6Task, NULL, &tim6Task_attributes);

  /* creation of tim7Task */
  tim7TaskHandle = osThreadNew(StartTIM7Task, NULL, &tim7Task_attributes);

  /* creation of buttonTask */
  buttonTaskHandle = osThreadNew(StartButtonTask, NULL, &buttonTask_attributes);

  ...

  /* Start scheduler */
  osKernelStart();

  /* We should never get here as control is now taken by the scheduler */
  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    /* USER CODE END WHILE */
```

```
    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
}
...
/* USER CODE BEGIN Header_StartTIM6Task */
/**
  * @brief  Function implementing the tim6Task thread.
  * @param  argument: Not used
  * @retval None
  */
/* USER CODE END Header_StartTIM6Task */
void StartTIM6Task(void *argument)
{
  /* USER CODE BEGIN StartTIM6Task */
  /* Infinite loop */
  for(;;)
  {
    osDelay(1);
  }
  /* USER CODE END StartTIM6Task */
}
...
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
  /* USER CODE BEGIN Callback 0 */

  /* USER CODE END Callback 0 */
  if (htim->Instance == TIM1) {
    HAL_IncTick();
  }
}
```

(notare la callback dell'interrupt di TIM1 che incrementa il systick dell'HAL).
Riguardo alla struttura dell'applicazione, l'idea è che la callback di una interruzione segnali l'avvenuta interruzione al task corrispondente inserendo un messaggio (un booleano, in realtà il contenuto del messaggio non ha alcuna importanza) nella corrispondente coda. Due importanti osservazioni:

1. Dal momento che l'uso di coda + scheduling "disaccoppia" temporalmente in maniera importante l'occorrenza dell'interruzione dalla sua gestione effettuata nel task, è meglio che il codice di debounce sia trasferito nel gestore dell'interruzione. In tal modo gli eventi inseriti nella coda button_queue sono già filtrati dal bouncing.
2. Per ragioni simili a quelle viste nella sezione precedente, occorre evitare le corse critiche sulle interruzioni quando si disattivano i timer. Un possibile modo potrebbe essere svuotare le code dopo aver disattivato i timer, soluzione del tutto analoga a quella utilizzata nell'architettura round-robin con interrupt per TIM7 (ossia, resettare il valore di tim7_it dopo la disattivazione del timer). Purtroppo questa soluzione, nel caso dell'architettura OS realtime, non funziona correttamente perché il task che disattiva il timer e quello che reagisce all'interrupt del timer, che nel caso dell'architettura round-robin erano eseguiti run-to-completion e in sequenza, e che quindi erano implicitamente sincronizzati, in questa architettura eseguono in maniera concorrente. Può pertanto capitare che, nell'intervallo di tempo in cui la routine di interrupt ha riempito la coda e prima che il task che disattiva il timer la svuoti, lo scheduler attivi il task che era in attesa sulla coda. Questo nell'architettura round-robin con interrupt non poteva avvenire perché il task che legge il flag dell'interrupt esegue sicuramente dopo il task che lo resetta. Pertanto utilizziamo la soluzione di arricchire

la macchina a stati della nostra applicazione, distinguendo lo stato on_preconfig dallo stato on_postconfig, così da poter filtrare gli interrupt di TIM7 in eccesso.

Il codice è il seguente:

main.c:

```
...
/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */

typedef enum { on_preconfig, on_postconfig, off, config } app_state;
typedef enum { false = 0, true = 1 } bool;

/* USER CODE END PTD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */

#define PWM_MAX 255
#define TIM7_1SEC_DELAY 10000
#define TIM7_4MSEC_DELAY 40

/* USER CODE END PD */
...
/* USER CODE BEGIN PV */

static uint8_t PWM_high_semiperiod;
static bool PWM_status_high;
static app_state current_state;
static bool config_direction_down;
static uint32_t lastDebouncedButtonMillis;

/* USER CODE END PV */
...
/* USER CODE BEGIN PFP */

static void app_init();
static void turn_on_LEDs();
static void turn_off_LEDs();
static void PWM_toggle_LEDs();
static void start_TIM7_1sec_delay();
static void start_TIM7_4msec_delay();
static void stop_TIM7();
static void calc_config_direction_down();
static void calc_PWM_status();
static void calc_PWM_high_semiperiod();

/* USER CODE END PFP */
...
int main(void)
{
  ...
  button_queueHandle = osMessageQueueNew (16, sizeof(bool),
&button_queue_attributes);
  ...
  app_init(); /* this must go after creation of queues because it turns on LEDs
*/
  ...
  tim6TaskHandle = osThreadNew(StartTIM6Task, NULL, &tim6Task_attributes);
  ...
}
...
```

```
/* USER CODE BEGIN 4 */
static void app_init() {
  PWM_high_semiperiod = PWM_MAX;
  PWM_status_high = true;
  current_state = off;
  config_direction_down = true;
  lastDebouncedButtonMillis = HAL_GetTick();
  turn_off_LEDs();
}
...

void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
  if (GPIO_Pin == USER_Btn_Pin) {
    uint32_t currentMillis = HAL_GetTick();
    if (currentMillis - lastDebouncedButtonMillis > 20) {
      lastDebouncedButtonMillis = currentMillis;
      bool msg = true;
      osMessageQueuePut(button_queueHandle, &msg, 0U, 0U);
    }
  }
}
/* USER CODE END 4 */

/* USER CODE BEGIN Header_StartTIM6Task */
/**
  * @brief  Function implementing the tim6Task thread.
  * @param  argument: Not used
  * @retval None
  */
/* USER CODE END Header_StartTIM6Task */
void StartTIM6Task(void *argument)
{
  /* USER CODE BEGIN 5 */
  /* Infinite loop */
  for(;;)
  {
    /* waits */
    bool msg;
    osMessageQueueGet(tim6_queueHandle, &msg, 0U, osWaitForever);

    if (current_state == on_preconfig || current_state == on_postconfig ||
        current_state == config) {
      calc_PWM_status();
      PWM_toggle_LEDs();
    }
  }
  /* USER CODE END 5 */
}

/* USER CODE BEGIN Header_StartTIM7Task */
/**
* @brief Function implementing the tim7Task thread.
* @param argument: Not used
* @retval None
*/
/* USER CODE END Header_StartTIM7Task */
void StartTIM7Task(void *argument)
{
  /* USER CODE BEGIN StartTIM7Task */
  /* Infinite loop */
  for(;;)
  {
    /* waits */
    bool msg;
```

```
    osMessageQueueGet(tim7_queueHandle, &msg, 0U, osWaitForever);

    if (current_state == on_preconfig) {
      current_state = config;
      start_TIM7_4msec_delay();
    } else if (current_state == config) {
      calc_config_direction_down();
      calc_PWM_high_semiperiod();
      start_TIM7_4msec_delay();
    }
  }
  /* USER CODE END StartTIM7Task */
}

/* USER CODE BEGIN Header_StartButtonTask */
/**
* @brief Function implementing the buttonTask thread.
* @param argument: Not used
* @retval None
*/
/* USER CODE END Header_StartButtonTask */
void StartButtonTask(void *argument)
{
  /* USER CODE BEGIN StartButtonTask */
  /* Infinite loop */
  static bool button_status_down = false;
  for(;;)
  {
    /* waits */
    bool msg;
    osMessageQueueGet(button_queueHandle, &msg, 0U, osWaitForever);

    button_status_down = !button_status_down;
    if (button_status_down) {
      if (current_state == off) {
        current_state = on_preconfig;
        turn_on_LEDs();
        start_TIM7_1sec_delay();
      } else if (current_state == on_postconfig) {
        current_state = off;
        turn_off_LEDs();
      } /* else error */
    } else { /* !button_status_down */
      if (current_state == on_preconfig || current_state == config) {
        current_state = on_postconfig;
        stop_TIM7();
      } /* else do nothing */
    }
  }
  /* USER CODE END StartButtonTask */
}

 /**
  * @brief  Period elapsed callback in non blocking mode
  * @note   This function is called  when TIM1 interrupt took place, inside
  * HAL_TIM_IRQHandler(). It makes a direct call to HAL_IncTick() to increment
  * a global variable "uwTick" used as application time base.
  * @param  htim : TIM handle
  * @retval None
  */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
  /* USER CODE BEGIN Callback 0 */
  /* USER CODE END Callback 0 */
```

```
  if (htim->Instance == TIM1) {
    HAL_IncTick();
  }
  /* USER CODE BEGIN Callback 1 */
  else if (htim->Instance == TIM6) {
    bool msg = true;
    osMessageQueuePut(tim6_queueHandle, &msg, 0U, 0U);
  } else if (htim->Instance == TIM7) {
    bool msg = true;
    osMessageQueuePut(tim7_queueHandle, &msg, 0U, 0U);
  }
  /* USER CODE END Callback 1 */
}
...
```

L’applicazione funziona e appare stabile nel suo comportamento, ma lo sweep della luminosità dei led ha un range meno ampio della soluzione che non faceva uso dell’OS. Non lasciamoci però ingannare dall’apparente stabilità dell’applicazione dal momento che questa ora è costituita da tre task concorrenti (più due ISR, anche loro concorrenti) e che i task concorrenti condividono delle variabili senza sincronizzarsi. Le variabili condivise sono:

- PWM_high_semiperiod: condivisa tra tim6Task (che effettua il PWM e quindi accede a tale variabile in sola lettura) e tim7Task (che varia il semiperiodo del PWM durante la configurazione, e quindi accede a tale variabile in lettura/scrittura);
- current_state: condivisa tra tim7Task e buttonTask, che causano i cambiamenti di stato e quindi accedono entrambi a tale variabile in lettura/scrittura.

Le altre variabili non sono condivise: PWM_status_high è usata solo da tim6Task, config_direction_down è usata solo da tim7Task, e lastDebouncedButtonMillis solo da HAL_GPIO_EXTI_Callback. Non conviene dichiararle static nelle funzioni, le prime due perché ci sono delle funzioni ausiliarie che le calcolano, la terza perché deve essere inizializzata dopo l’inizializzazione dell’HAL, quindi occorre controllare esattamente la sua inizializzazione attraverso l’invocazione di app_init().
La variabile condivisa PWM_high_semiperiod non è troppo problematica: se tim7Task scrive tale variabile anche mentre tim6Task la sta leggendo, nel caso peggiore si creano situazioni transienti che si risolvono nel tempo di un semiperiodo, e pertanto non dovrebbero nemmeno essere in grado di produrre effetti visibili sulla luminosità dei led. Potrebbe pertanto essere sufficiente dichiarare tale variabile volatile. Più problematica potrebbe essere la condivisione senza sincronizzazione della variabile current_state tra tim7Task e buttonTask, dal momento che tale condivisione potrebbe generare stati scorretti a causa di corse critiche tra i due blocchi di codice che implementano le transizioni di stato. Consideriamo ad esempio questa possibile situazione:

- L’applicazione è in stato on_preconfig, e l’utente rilascia il pulsante un secondo dopo averlo premuto: sia TIM7 che il pulsante generano un interrupt, e i task buttonTask e tim7Task diventano ready;
- Lo scheduler manda in esecuzione buttonTask che legge lo stato corrente (on_preconfig);
- Prima che buttonTask possa modificare alcunché avviene una preemption (i due task hanno la stessa priorità ed eseguono in round robin), e lo scheduler esegue tim7Task, che legge lo stato corrente (on_preconfig);
- A questo punto possiamo andare in uno stato incoerente dovuto ad una sequenza di scheduling arbitraria; ad esempio, tim7Task imposta current_state a configure – preemption – buttonTask imposta current_state a on_postconfig e disattiva TIM7 – preemption – tim7Task riattiva TIM7.

Nell’esempio sopra riportato l’applicazione a causa della corsa critica finisce dopo il rilascio del pulsante in stato on_postconfig, con i led che stanno facendo sweeping della luminosità. Tale situazione è improbabile ma non impossibile. Il modo più semplice per evitarla è proteggere la variabile current_state con un mutex, rendendo mutuamente esclusive le due transizioni di stato in tim7Task e buttonTask. A tale scopo andare nel device configuration tool, selezionare Middleware > FREERTOS e il tab Mutexes, ed aggiungere un mutex di nome mutex_state.

Codice generato:

main.c:

```
...
/* Private variables ---------------------------------------------------------*/
...
/* Definitions for mutex_state */
osMutexId_t mutex_stateHandle;
const osMutexAttr_t mutex_state_attributes = { ,,,
...
int main(void)
{
  ...
  /* Init scheduler */
  osKernelInitialize();
  /* Create the mutex(es) */
  /* creation of mutex_state */
  mutex_stateHandle = osMutexNew(&mutex_state_attributes);
  ...
}
...
```

Codice da aggiungere:

main.c:

```
...
/* USER CODE BEGIN Header_StartTIM7Task */
/**
* @brief Function implementing the tim7Task thread.
* @param argument: Not used
* @retval None
*/
/* USER CODE END Header_StartTIM7Task */
void StartTIM7Task(void *argument)
{
  /* USER CODE BEGIN StartTIM7Task */
  /* Infinite loop */
  for(;;)
  {
    /* waits */
    bool msg;
    osMessageQueueGet(tim7_queueHandle, &msg, 0U, osWaitForever);

    osMutexAcquire(mutex_stateHandle, osWaitForever);
    if (current_state == on) {
      ...
    }
    osMutexRelease(mutex_stateHandle);
  }
  /* USER CODE END StartTIM7Task */
}

/* USER CODE BEGIN Header_StartButtonTask */
/**
```

```
* @brief Function implementing the buttonTask thread.
* @param argument: Not used
* @retval None
*/
/* USER CODE END Header_StartButtonTask */
void StartButtonTask(void *argument)
{
  /* USER CODE BEGIN StartButtonTask */
  /* Infinite loop */
  static bool button_status_down = false;
  for(;;)
  {
    /* waits */
    bool msg;
    osMessageQueueGet(button_queueHandle, &msg, 0U, osWaitForever);


    button_status_down = !button_status_down;
    if (button_status_down) {
      osMutexAcquire(mutex_stateHandle, osWaitForever);
      ...
      osMutexRelease(mutex_stateHandle);
    }
  }
  /* USER CODE END StartButtonTask */
}
```

Un’altra osservazione interessante riguarda la velocità relativa tra routine di interrupt e task, velocità che condiziona la capacità dei task di elaborare gli interrupt. L’aspetto è particolarmente critico, ad esempio, per tim6Task, che riceve una notifica di un interrupt di TIM6 e che deve “riarmare” TIM6 per generare il successivo interrupt, ed effettuare il toggle dei led per implementare il PWM. Non può permettersi di perdere nemmeno un interrupt, pena il blocco completo del meccanismo del PWM (usando una coda a profondità sufficiente possiamo cercare di ovviare in parte ai problemi di jitter nella velocità di gestione dell’interrupt). Non è però possibile mettere tim6Task a una priorità più alta degli altri due task: essendo gli interrupt di TIM6 a frequenza elevata lo scheduling continuo di tim6Task impedirebbe agli altri task di eseguire. Notare che tim6Task è relativamente breve, ma invoca delle funzioni dello HAL (il toggle dei pin GPIO) che potrebbero essere lente. Una possibile idea di modifica all’architettura potrebbe essere lo spostamento del codice che “riarma” TIM6 nella ISR, lasciando il codice lento (toggle dei pin GPIO) nel tim6Task. Ciò però reintroduce il problema della mancata sincronizzazione tra overflow del timer e switch dei led che aveva causato, nella nostra primissima implementazione del PWM, l’errata variazione di luminosità. In generale è meglio che gli interrupt a bassa frequenza, e generati dall’interazione utente, siano serviti con priorità rispetto a quelli ad alta frequenza non generati dall’interazione utente. Quindi possiamo provare a dare le seguenti priorità ai task (andare nel device configuration tool, Middleware > FREERTOS, tab Tasks and Queues, doppio click sul task e modificare nel popup che viene visualizzato):

- buttonTask: osPriorityAboveNormal
- tim7Task: osPriorityNormal
- tim6Task: osPriorityLow

Vediamo infine un meccanismo di sincronizzazione specifico di FreeRTOS che, usato al posto delle code, permette di accelerare la reattività di un task al verificarsi di un evento di interrupt. Tale meccanismo è detto di notifica diretta ai task. Ogni task ha un array di (contatori di) eventi che possono essere notificati da altri task in maniera diretta, e un task può mettersi in attesa sull’arrivo dell’evento. Non solo: esiste una funzione di notifica specifica per le ISR, e che permette di

effettuare, successivamente, un cambio di contesto diretto al task notificato, se questo ha una priorità più alta del task corrente, che la ISR ha interrotto. In tal modo è “come se” il task svegliato dalla ISR eseguisse nella ISR – in realtà, esegue nel suo proprio contesto, ma immediatamente dopo la conclusione della ISR. Nel nostro caso, dal momento che stiamo usando le notifiche come un semaforo contatore di eventi “leggero”, possiamo utilizzare le API di FreeRTOS ulTaskNotifyTake(), per mettersi in attesa su un evento, e vTaskNotifyGiveFromISR(), per segnalare un evento, entrambe definite in task.h. La macro per effettuare il context switch al task notificato è portYIELD_FROM_ISR(), definita in un header file importato attraverso FreeRTOS.h. Entrambi gli header files sono importati attraverso cmsis_os.h, quindi non dobbiamo preoccuparci di aggiungere nulla. Si possono pertanto eliminare le code nella pagina FREERTOS del device configuration tool, e poi modificare così il codice:

main.c:

```
/* USER CODE BEGIN 4 */
...

void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
  if (GPIO_Pin == USER_Btn_Pin) {
    uint32_t currentMillis = HAL_GetTick();
    if (currentMillis - lastDebouncedButtonMillis > 20) {
      lastDebouncedButtonMillis = currentMillis;
      BaseType_t pxHigherPriorityTaskWoken;
      vTaskNotifyGiveFromISR(buttonTaskHandle, &pxHigherPriorityTaskWoken);
      portYIELD_FROM_ISR(pxHigherPriorityTaskWoken);
    }
  }
}
/* USER CODE END 4 */
...

/* USER CODE BEGIN Header_StartTIM6Task */
/**
  * @brief  Function implementing the tim6Task thread.
  * @param  argument: Not used
  * @retval None
  */
/* USER CODE END Header_StartTIM6Task */
void StartTIM6Task(void *argument)
{
  /* USER CODE BEGIN 5 */
  /* Infinite loop */
  for(;;)
  {
    /* waits */
    ulTaskNotifyTake(pdFALSE, osWaitForever);
    ...
  }
  /* USER CODE END 5 */
}

/* USER CODE BEGIN Header_StartTIM7Task */
/**
* @brief Function implementing the tim7Task thread.
* @param argument: Not used
* @retval None
*/
/* USER CODE END Header_StartTIM7Task */
void StartTIM7Task(void *argument)
{
  /* USER CODE BEGIN StartTIM7Task */
  /* Infinite loop */
```

```
  for(;;)
  {
    /* waits */
    ulTaskNotifyTake(pdFALSE, osWaitForever);
    ...
  }
  /* USER CODE END StartTIM7Task */
}

/* USER CODE BEGIN Header_StartButtonTask */
/**
* @brief Function implementing the buttonTask thread.
* @param argument: Not used
* @retval None
*/
/* USER CODE END Header_StartButtonTask */
void StartButtonTask(void *argument)
{
  /* USER CODE BEGIN StartButtonTask */
  /* Infinite loop */
  static bool button_status_down = false;
  for(;;)
  {
    /* waits */
    ulTaskNotifyTake(pdFALSE, osWaitForever);
    ...
  }
  /* USER CODE END StartButtonTask */
}

 /**
  * @brief  Period elapsed callback in non blocking mode
  * @note   This function is called  when TIM1 interrupt took place, inside
  * HAL_TIM_IRQHandler(). It makes a direct call to HAL_IncTick() to increment
  * a global variable "uwTick" used as application time base.
  * @param  htim : TIM handle
  * @retval None
  */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
  /* USER CODE BEGIN Callback 0 */
  /* USER CODE END Callback 0 */
  if (htim->Instance == TIM1) {
    HAL_IncTick();
  }
  /* USER CODE BEGIN Callback 1 */
  else if (htim->Instance == TIM6) {
    BaseType_t pxHigherPriorityTaskWoken;
    vTaskNotifyGiveFromISR(tim6TaskHandle, &pxHigherPriorityTaskWoken);
    portYIELD_FROM_ISR(pxHigherPriorityTaskWoken);
  } else if (htim->Instance == TIM7) {
    BaseType_t pxHigherPriorityTaskWoken;
    vTaskNotifyGiveFromISR(tim7TaskHandle, &pxHigherPriorityTaskWoken);
    portYIELD_FROM_ISR(pxHigherPriorityTaskWoken);
  }
  /* USER CODE END Callback 1 */
}
...
```