PULSE WIDTH MODULATION (PWM)
La PWM è una tecnica per generare un segnale pseudo-analogico a partire da un segnale digitale attraverso la modulazione della larghezza di un'onda quadra. In questo caso useremo la tecnica PWM per regolare la luminosità dei led. Perchè il flicker dei led sia invisibile all'occhio umano la frequenza deve essere almeno 100 Hz. Sviluppare la seguente applicazione (PWM dimmer):

- La luminosità dei led deve essere controllata dal PWM in maniera che vari continuamente con risoluzione di 8 bit nel range di luminosità 0% - 100%.
- La pressione di B1 deve avere una funzione duplice: se premuto e rilasciato prima di un secondo, deve commutare lo stato dei led da accesi a spenti e viceversa, mentre se tenuto premuto per più di un secondo quando i led sono nello stato spento, fanno entrare il sistema nello stato di configurazione.
- Lo stato di configurazione permane fintanto che il pulsante rimane premuto, e in tale modalità la luminosità dei led varia ciclicamente a partire da quella corrente nel range 10% - 100% (esempio: dalla luminosità minima aumenta fino al 100%, poi diminuisce fino alla luminosità minima, poi torna ad aumentare fino a 100%, ecc.). La variazione è continua e relativamente lenta (il ciclo completo deve durare non più di un paio di secondi). Il rilascio del pulsante fa uscire il sistema dallo stato di configurazione e fa entrare nello stato di led acceso, con il led alla luminosità che aveva quando è uscito dallo stato di configurazione. Tutte le volte che il sistema rientra nello stato di led acceso i led sono alla luminosità che è stata impostata dall'ultimo stato di configurazione.

Utilizziamo TIM6 e TIM7; utilizzeremo TIM6 per il PWM, mentre TIM7 sarà utilizzato per il timing del pulsante e per il timing della variazione ciclica della luminosità nello stato di configurazione. Se la frequenza deve essere almeno 100 Hz, e la risoluzione almeno di 8 bit, significa che il periodo deve essere divisibile almeno per $2^8 = 256$ se vogliamo avere tutti i possibili duty cycle. Quindi TIM6 deve essere commutato con una frequenza di almeno $100 \cdot 256 \approx 26$ KHz. Usando la solita formula, e tenendo conto che TIM6 è di default clockato a 96 MHz: $26 \cdot 10^3$ Hz = $96 \cdot 10^6$ Hz /((Prescaler + 1) (Period + 1)), ossia, (Prescaler + 1) (Period + 1) $\approx 4 \cdot 10^3$. Se imponiamo Period = 1 (per avere la frequenza più alta, il periodo deve essere il minimo) otteniamo Prescaler = $4 \cdot 10^3 / 2 - 1 = 1999$. Impostato pertanto TIM6 con questo parametro, ed abilitato l'interrupt di TIM6, possiamo scrivere il codice del PWM come segue:

main.c:
```
...
/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */

#define PWM_MAX 256

/* USER CODE END PD */
...
/* Private variables ---------------------------------------------------------*/
...
/* USER CODE BEGIN PV */

volatile int PWM_high_semiperiod;
volatile int PWM_status_high;

/* USER CODE END PV */
...

int main(void)
{
...
  /* USER CODE BEGIN 2 */
  PWM_high_semiperiod = PWM_MAX;
```

```
  PWM_status_high = 1;
  HAL_TIM_Base_Start_IT(&htim6);
  HAL_GPIO_WritePin(GPIOB, LD1_Pin|LD3_Pin|LD2_Pin, GPIO_PIN_SET);
  /* USER CODE END 2 */
...
}
...

/* USER CODE BEGIN 4 */

void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {
  if (htim->Instance == TIM6) {
    if (PWM_high_semiperiod > 0 && PWM_high_semiperiod < PWM_MAX) {
      PWM_status_high = 1 - PWM_status_high;
    } else if (PWM_high_semiperiod == 0) {
      PWM_status_high = 0;
    } else { /* PWM_high_semiperiod == PWM_MAX */
      PWM_status_high = 1;
    }
    HAL_GPIO_WritePin(GPIOB, LD1_Pin|LD3_Pin|LD2_Pin, (PWM_status_high ?
GPIO_PIN_SET : GPIO_PIN_RESET));
    htim->Instance->ARR = (PWM_status_high ? PWM_high_semiperiod : (PWM_MAX -
PWM_high_semiperiod));
  }
}

/* USER CODE END 4 */
```

Questo esempio aiuta a capire il funzionamento del parametro auto-reload preload del timer. Se impostiamo auto-reload preload a true per il timer vediamo che, se proviamo a modificare l'inizializzazione nel main di PWM_high_semiperiod per vedere il dimming, i LED rimangono sempre a luminosità massima. Questo probabilmente perché se attiviamo l'auto-reload preload il valore del periodo del timer rimane lo stesso fino allo scadere del periodo. Ma allo scadere del periodo impostiamo di nuovo il valore del periodo, che non viene caricato (ancora) fino alla fine del periodo: risultato, in questo modo il periodo del timer non cambia mai. Pertanto dobbiamo impostare il parametro auto-reload preload a true (dovrebbe esserlo di default).
Ahimè anche in questo modo non funziona a dovere. Se facciamo andare tutto ad una frequenza percepibile, ad esempio con il debugging step-by-step mettendo un breakpoint nella callback del timer, tutto sembra funzionare come dovuto e i LED si accendono e spengono alternativamente. Se facciamo andare tutto alla frequenza necessaria al dimming (> 100 Hz) si verifica un interessante effetto: se proviamo a regolare la luminosità dei LED modificando il valore di inizializzazione di PWM_high_semiperiod, ci accorgiamo che il dimming è invertito: i valori alti di PWM_high_semiperiod rendono la luminosità bassa, mentre i valori bassi rendono la luminosità alta. L'origine del problema è verosimilmente la mancata sincronizzazione tra l'avvio del timer e il toggle dei LED: il primo dovrebbe avvenire solo dopo che il secondo è stato completato. A tale scopo modifichiamo il codice per stoppare il timer quando termina un semiperiodo, e lo facciamo esplicitamente ripartire per il semiperiodo successivo solo dopo il toggle dei LED. Per tale motivo occorre attivare l'One Pulse Mode per TIM6 nel device configuration tool, e modificare il codice della callback come segue:

main.c:

```
/* USER CODE BEGIN 4 */

void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {
  if (htim->Instance == TIM6) {
    if (PWM_high_semiperiod > 0 && PWM_high_semiperiod < PWM_MAX) {
      PWM_status_high = 1 - PWM_status_high;
    } else if (PWM_high_semiperiod == 0) {
      PWM_status_high = 0;
```

```
    } else { /* PWM_high_semiperiod == PWM_MAX */
      PWM_status_high = 1;
    }
    HAL_GPIO_WritePin(GPIOB, LD1_Pin|LD3_Pin|LD2_Pin, (PWM_status_high ?
GPIO_PIN_SET : GPIO_PIN_RESET));
    htim->Instance->ARR = (PWM_status_high ? PWM_high_semiperiod : (PWM_MAX –
PWM_high_semiperiod));
    HAL_TIM_Base_Start_IT(htim);
  }
}

/* USER CODE END 4 */
```

Inoltre occorre anche aggiungere ulteriore inizializzazione al codice di inizializzazione del timer. Infatti il codice generato da CubeMX permette di attivare il timer solo una volta (modalità TIM_OPMODE_SINGLE), mentre a noi serve poterlo attivare molte volte (modalità TIM_OPMODE_REPETITIVE):
main.c:

```
static void MX_TIM6_Init(void)
{
...
  /* USER CODE BEGIN TIM6_Init 2 */
  if (HAL_TIM_OnePulse_Init(&htim6, TIM_OPMODE_REPETITIVE) != HAL_OK
  {
      Error_Handler();
  }
  /* USER CODE END TIM6_Init 2 */
}
```

Per quanto riguarda la gestione del pulsante, utilizziamo un solo timer (TIM7) sia per individuare la modalità di configurazione, sia per, una volta entrati in modalità di configurazione, far variare il valore del duty cycle. Occorre pertanto configurare il timer con due diversi valori di costanti di tempo: nel primo caso (individuazione della modalità di configurazione) occorre che il timer vada in overflow dopo un secondo, nel secondo caso (cambiamento del valore del duty cycle), dal momento che il ciclo deve durare due secondi e che la risoluzione è 256 da min a max, il timer deve andare in overflow dopo 2 sec / (2 · 256) ≈ 4 msec (notare che il denominatore ha un fattore 2 perché un ciclo va da min a max e da max a min). Cerchiamo nelle due configurazioni di TIM7 di tener fisso un parametro, ad esempio Prescaler, e di far variare solo Period; abbiamo pertanto:

$$1=\frac{96\cdot 10^6}{(Prescaler+1)\cdot(Period+1)} \quad \text{per il primo caso, e}$$

$$\frac{1}{4\cdot 10^{-3}}=\frac{96\cdot 10^6}{(Prescaler+1)\cdot(Period+1)} \quad \text{per il secondo caso.}$$

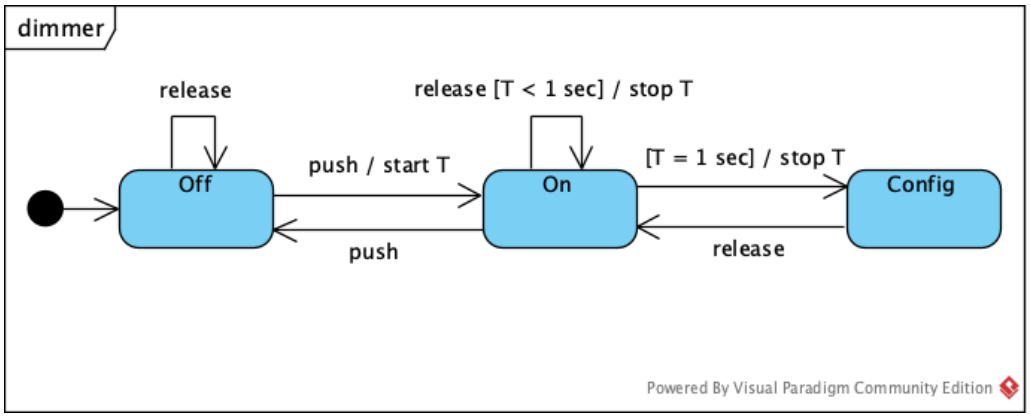
Le due equazioni sono approssimatamente: Prescaler · Period = 10^8 e Prescaler · Period = $4 \cdot 10^5$. Se fissiamo, ad esempio, Prescaler = 10^4, nel primo caso deve essere Period = 10^4, nel secondo caso Period = 40. Anche TIM7 va configurato nel device configuration tool impostando il prescaler a 10000 e attivando l'interrupt e l'One Pulse Mode. Inoltre anche nella funzione di inizializzazione di TIM7 va aggiunto il codice per attivare la modalità TIM_OPMODE_REPETITIVE.

La nostra applicazione deve comportarsi come una macchina a stati che possiamo modellizzare come segue:

- Gli stati sono tre: off (LED spenti), on (LED accesi stabili), config (LED accesi con luminosità in variazione per configurazione luminosità);

- Se nello stato off premo il pulsante vado in stato on facendo partire TIM7 per 1 secondo; se nello stato on rilascio il pulsante prima che TIM7 scada, rimango nello stato on e disattivo TIM7;
- Se nello stato on TIM7 va in overflow dopo 1 secondo prima che abbia rilasciato il pulsante, vado in stato config; in stato config il pulsante è premuto e quindi posso solo rilasciarlo, nel qual caso vado in stato on
- Se nello stato on premo il pulsante vado in stato off; se in tale stato lo tengo premuto/lo rilascio non succede nulla, rimango in stato off.

Riportiamo un diagramma UML state machine che sintetizza il tutto:



Una implementazione che pone la macchina a stati interamente nelle callback degli interrupt è la seguente:

main.c:

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

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

/* USER CODE END PTD */

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

#define PWM_MAX 256
#define TIM7_1SEC_DELAY 10000
#define TIM7_4MSEC_DELAY 40

/* USER CODE END PD */
...
/* Private variables ---------------------------------------------------------*/
...
/* USER CODE BEGIN PV */

volatile int PWM_high_semiperiod;
volatile bool PWM_status_high;
volatile bool btn_status_down;
volatile app_state current_state;
volatile bool config_direction_down;

/* USER CODE END PV */

/* Private function prototypes -----------------------------------------------*/
```

```
...
/* USER CODE BEGIN PFP */
static void app_init(void);
static void turn_on_LEDs(void);
static void turn_off_LEDs(void);
static void calc_PWM_status(void);
static void PWM_toggle_LEDs(void);
static void start_TIM7_1sec_delay(void);
static void start_TIM7_4msec_delay(void);
static void stop_TIM7(void);
static void calc_config_direction_down(void);
static void calc_PWM_high_semiperiod(void);
/* USER CODE END PFP */
...

int main(void)
{
...
  /* USER CODE BEGIN 2 */
  app_init();
  /* USER CODE END 2 */
...
}
...

/* USER CODE BEGIN 4 */

void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {
  if (htim->Instance == TIM6) {
    calc_PWM_status();
    PWM_toggle_LEDs();
  } else if (htim->Instance == TIM7) {
    if (current_state == on) {
      current_state = config;
      start_TIM7_4msec_delay();
    } else if (current_state == config) {
      calc_config_direction_down();
      calc_PWM_high_semiperiod();
      start_TIM7_4msec_delay();
    } else { /* current state == off */
      /* Error_Handler(); */
    }
  }
}

void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
  if (GPIO_Pin == USER_Btn_Pin) {
    btn_status_down = !btn_status_down;
    if (btn_status_down) {
      if (current_state == off) {
        current_state = on;
        turn_on_LEDs();
        start_TIM7_1sec_delay();
      } else if (current_state == on) {
        current_state = off;
        turn_off_LEDs();
      } else { /* current_state == config */
        /* Error_Handler(); */
      }
    } else {
      if (current_state == on || current_state == config) {
        stop_TIM7();
        current_state = on; /* exit from config state */
      } else { /* current_state == off */
```

```
        /* do nothing */
      }
    }
  }
}

/* here follow the definitions of all the auxiliary functions */

static void app_init(void) {
  PWM_high_semiperiod = PWM_MAX;
  PWM_status_high = true;
  btn_status_down = false;
  current_state = on;
  config_direction_down = true; /* not necessary */
  turn_on_LEDs();
}

static void turn_on_LEDs(void) {
  HAL_GPIO_WritePin(GPIOB, LD1_Pin|LD3_Pin|LD2_Pin, GPIO_PIN_SET);
  HAL_TIM_Base_Start_IT(&htim6);
}

static void turn_off_LEDs(void) {
  HAL_GPIO_WritePin(GPIOB, LD1_Pin|LD3_Pin|LD2_Pin, GPIO_PIN_RESET);
  HAL_TIM_Base_Stop_IT(&htim6);
}

static void calc_PWM_status(void) {
  if (PWM_high_semiperiod > 0 && PWM_high_semiperiod < PWM_MAX) {
    PWM_status_high = !PWM_status_high;
  } else if (PWM_high_semiperiod == 0) {
    PWM_status_high = false;
  } else { /* PWM_high_semiperiod == PWM_MAX */
    PWM_status_high = true;
  }
}

static void PWM_toggle_LEDs(void) {
  HAL_GPIO_WritePin(GPIOB, LD1_Pin|LD3_Pin|LD2_Pin, (PWM_status_high ?
GPIO_PIN_SET : GPIO_PIN_RESET));
  htim6.Instance->ARR = (PWM_status_high ? PWM_high_semiperiod : (PWM_MAX -
PWM_high_semiperiod));
  HAL_TIM_Base_Start_IT(&htim6);
}

static void start_TIM7_1sec_delay(void) {
  htim7.Instance->ARR = TIM7_1SEC_DELAY;
  HAL_TIM_Base_Start_IT(&htim7);
}

static void start_TIM7_4msec_delay(void) {
  htim7.Instance->ARR = TIM7_4MSEC_DELAY;
  HAL_TIM_Base_Start_IT(&htim7);
}

static void stop_TIM7(void) {
  HAL_TIM_Base_Stop_IT(&htim7);
}

static void calc_config_direction_down(void) {
  if (PWM_high_semiperiod == PWM_MAX) {
    config_direction_down = true;
  } else if (PWM_high_semiperiod == 0) {
    config_direction_down = false;
```

```
  } /* else, config direction stays the same */
}

static void calc_PWM_high_semiperiod(void) {
  PWM_high_semiperiod = (config_direction_down ? PWM_high_semiperiod - 1 :
PWM_high_semiperiod + 1);
}

/* USER CODE END 4 */
```

Se proviamo ad eseguire questo codice, il comportamento è molto instabile. Quando premiamo il pulsante con i led spenti questi a volte non si accendono, a volte si accendono quando rilasciamo il pulsante, a volte si accendono per un istante e si spengono subito dopo, a volte se entriamo in modalità configurazione non ne usciamo al rilascio del pulsante…
Facendo un po' di sperimentazione con il pulsante si può comprendere che una possibile sorgente del problema è lo switch bounce. In effetti se facciamo un semplice progetto con solo attivo l'interrupt del pulsante in modalità IT_RISING_FALLING, e con il seguente codice per la callback dell'interrupt del pulsante:

main.c:

```
...
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
  static volatile int counter = 0;
  if (GPIO_Pin == USER_Btn_Pin) {
    ++counter;
  }
}
```

quindi lanciamo il codice in modalità debug, premiamo il pulsante per un certo numero di volte, ad esempio 20, ed inseriamo un breakpoint per esaminare il valore di counter, non otteniamo che esso ha come valore 40, come ci aspetteremmo, ma ha in realtà un valore decisamente più alto, in alcuni casi anche più del doppio del valore atteso. Nell'esempio in cui il pulsante era usato per fare il toggle dei led questo effetto non si notava, perché in quel caso lo stato della luce dei led dipende solo dal livello dell'input, e quindi quando il livello si aggiusta dopo i bounce, anche la luce si aggiusta sul valore corretto. In questo caso, invece, lo stato dipende dal conteggio (per la precisione, dalla parità) delle transizioni di livello, e pertanto l'introduzione di transizioni spurie si nota.
Sospendiamo pertanto l'esercizio e discutiamo come possiamo effettuare il debounce del pulsante.