SWITCH DEBOUNCE

La serie di articoli che inizia con https://www.eejournal.com/article/ultimate-guide-to-switch-debounce-part-1/ è una possibile guida al problema e alle possibili soluzioni, ma introduce quasi esclusivamente soluzioni hardware. L'unica (alla data odierna) soluzione software proposta in quella serie di articoli è una soluzione "base": una volta individuato un fronte (di salita, di discesa) vengono rifiutati tutti gli altri fronti per un periodo di tempo sufficiente alla stabilizzazione del segnale, periodo di tempo individuato sperimentalmente in circa 20 msec. Possiamo cercare di fare lo stesso: individuato un interrupt del pulsante, rifiutiamo tutti gli interrupt successivi per un periodo di 20 msec.

Una prima implementazione basata sugli interrupt, dove usiamo TIM6 come timer per attendere 20 msec (dalla formula si ricava ad es. Prescaler = 1000 e Period = 2000):

main.c:

```
/* Private variables ---------------------------------------------------------*/
...
/* USER CODE BEGIN PV */

volatile bool active;

/* USER CODE END PV */
...
int main(void)
{
...
  /* USER CODE BEGIN 2 */
  active = true;
  /* USER CODE END 2 */
...
}
...
/* USER CODE BEGIN 4 */

void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {
  if (htim->Instance == TIM6) {
    active = true;
  }
}

void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
  static volatile int count = 0;
  if (GPIO_Pin == USER_Btn_Pin) {
    if (active) {
      active = false;
      HAL_TIM_Base_Start_IT(&htim6);
      ++count;
      HAL_GPIO_TogglePin(GPIOB, LD1_Pin|LD3_Pin|LD2_Pin);
    }
  }
}

/* USER CODE END 4 */
```

Come al solito la variabile count è usata per contare quanti interrupt vengono generati – in questo caso quanti interrupt filtrati. Notare alcuni accorgimenti: la variabile active in HAL_GPIO_EXTI_Callback viene impostata a false subito dopo il check (è la prima istruzione

dell’if) e solo dopo viene tutto il resto. Ciononostante il codice non è perfettamente stabile: il debouncing funziona per tutti gli interrupt eccetto il primo!

Per rendere il tutto più stabile cerchiamo di semplificare radicalmente la routine di interrupt: essa fa troppe cose, e questo può dare i problemi di sincronizzazione sulla variabile condivisa active dovuti ad invocazioni multiple della callback. Pertanto “svuotiamo” completamente la routine di interrupt ed usiamo una architettura “round robin con interrupt”, dove le routine di interrupt semplicemente segnalano l’occorrenza di eventi, ma la loro gestione viene fatta nel main ciclico, e quindi in maniera sincrona. Questo dovrebbe eliminare i problemi di sincronizzazione e rendere tutto funzionante. Infine eliminiamo il timer, dal momento che a questo punto si può utilizzare la funzione dell’HAL che ritorna i tick correnti.

Il codice:

```
/* Private variables ---------------------------------------------------------*/
...
/* USER CODE BEGIN PV */

volatile bool button_it;

/* USER CODE END PV */
...
int main(void)
{
  /* USER CODE BEGIN 1 */
  uint32_t lastDebouncedButtonMillis;
  int count = 0;
  /* USER CODE END 1 */
...
  /* USER CODE BEGIN 2 */
  lastDebouncedButtonMillis = HAL_GetTick();
  button_it = false;
  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  HAL_GPIO_TogglePin(GPIOB, LD1_Pin|LD3_Pin|LD2_Pin);
  while (1)
  {
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
    if (button_it) {
      button_it = false;
      uint32_t currentMillis = HAL_GetTick();
      if (currentMillis - lastDebouncedButtonMillis > 20) {
        lastDebouncedButtonMillis = currentMillis;
        HAL_GPIO_TogglePin(GPIOB, LD1_Pin|LD3_Pin|LD2_Pin);
        ++count;
      }
    }
  }
  /* USER CODE END 3 */
}
...
/* USER CODE BEGIN 4 */


void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
  button_it = true;
}
```

```
/* USER CODE END 4 */
```

Stavolta come si può notare il toggle del led è perfetto, e il conteggio degli interrupt pure (esattamente il doppio del numero dei click).

INTEGRAZIONE DEBOUNCE CON PWM DIMMER

A questo punto possiamo integrare la soluzione di debounce con la soluzione dell'applicazione PWM dimmer in un'architettura "round robin con interrupt". Per dettagli sull'architettura vedere le slides.

main.c:

```
...
/* Private variables ---------------------------------------------------------*/
...
/* USER CODE BEGIN PV */

static volatile bool tim6_it;
static volatile bool tim7_it;
static volatile bool button_it;
static uint8_t PWM_high_semiperiod;
static bool PWM_status_high;
static bool button_status_down;
static app_state current_state;
static bool config_direction_down;

/* 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)
{
  /* USER CODE BEGIN 1 */
  uint32_t lastDebouncedButtonMillis;
  /* USER CODE END 1 */
...
  /* USER CODE BEGIN 2 */
  app_init();
  lastDebouncedButtonMillis = HAL_GetTick();
  /* USER CODE END 2 */

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

    /* USER CODE BEGIN 3 */
```

```
    /* task 1 */
    if (tim6_it) {
      tim6_it = false;

      calc_PWM_status();
      PWM_toggle_LEDs();
    }

    /* task 2 */
    if (tim7_it) {
      tim7_it = false;

      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();
      }
    }

    /* task 3 */
    if (button_it) {
      button_it = false;
      uint32_t currentMillis = HAL_GetTick();
      if (currentMillis - lastDebouncedButtonMillis > 20) {
        lastDebouncedButtonMillis = currentMillis;

        button_status_down = !button_status_down;
        if (button_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 error */
        } else { /* !button_status_down */
          if (current_state == on || current_state == config) {
            current_state = on;
            stop_TIM7();
          } /* else do nothing */
        }
      }
    }
  }
  /* USER CODE END 3 */
}
...
/* USER CODE BEGIN 4 */

void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {
  if (htim->Instance == TIM6) {
    tim6_it = true;
  } else if (htim->Instance == TIM7) {
    tim7_it = true;
  }
}

void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
  button_it = true;
```

```
}

static void app_init() {
  tim6_it = false;
  tim7_it = false;
  button_it = false;
  PWM_high_semiperiod = PWM_MAX;
  PWM_status_high = true;
  button_status_down = false;
  current_state = on;
  config_direction_down = true;
  turn_on_LEDs();
}
...
/* USER CODE END 4 */
```

Se però proviamo, vediamo che l'applicazione non funziona correttamente. Ad esempio, di tanto in tanto cliccando il bottone i led non si spengono. Perché non funziona? Se analizziamo il codice possiamo notare che c'è una corsa critica che può causare il non spegnimento dei led al click del pulsante:

- Clicchiamo il pulsante, generiamo l'interrupt, attiviamo button_it; il main arriva al task 3, supponiamo current_state == on e button_status_down == true; viene invocata turn_off_LEDs e TIM6 viene disattivato;
- Prima però che TIM6 venga disattivato però questo genera un interrupt; tim6_it viene impostato a true;
- Successivamente il main arriva al task 1, che è attivo, e PWM_toggle_LEDs riattiva TIM6, e quindi i LED si riattivano.

Il problema è che, avendo spostato il codice calc_PWM_status(); PWM_toggle_LEDs(); dalla routine di interrupt di TIM6, lo abbiamo desincronizzato con TIM6, e quindi disattivare TIM6 non è più sufficiente per spegnere i led! La soluzione è introdurre una guardia sul codice del task 1 basata sullo stato: l'interrupt del timer 6 va considerato solo se lo stato corrente è on oppure config, altrimenti si deve ignorare. Possiamo pertanto rimediare la situazione modificando il codice in questo modo:

main.c:

```
...
...
int main(void)
{
...
  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */

    /* task 1 */
    if (tim6_it) {
      tim6_it = false;

      if (current_state == on || current_state == config) {
        calc_PWM_status();
        PWM_toggle_LEDs();
      } /* else, do nothing */
```

```
    }
```

Esiste un’altra simile corsa critica relativa a TIM7, anch’essa dovuta al fatto che abbiamo tolto il codice dello stop di TIM7 dalla routine di interrupt di TIM7 e la abbiamo messa nel task 3. Quando il task 3 rileva il sollevamento del pulsante nello stato di configurazione, stoppa TIM7 per terminare lo sweeping della luminosità dei led. Se però prima dello stop di TIM7 questo genera un interrupt, quando il task 2 viene successivamente attivato esso fa ritornare il sistema in stato di configurazione e riattiva TIM7. Questa corsa critica è stata individuata con code review, ma in pratica l’effetto indesiderato non sembra verificarsi a runtime.
Possibili soluzioni:

- Usare due timer distinti, uno per l’attesa di 1 secondo e uno per l’attesa di 4 millisecondi; in tal modo si può ignorare l’interrupt dello sweeping quando si è in stato on, senza confonderlo con l’interrupt di attivazione dello stato di configurazione;
- Distinguere lo stato on nel quale entriamo dallo stato off (chiamiamolo on_preconfig) dallo stato on nel quale entriamo quando entriamo dallo stato config (chiamiamolo on_postconfig). In tal modo se siamo in on_postconfig possiamo filtrare gli interrupt di TIM7 utilizzando una guardia sullo stato, come fatto per TIM6. Notare che bisogna aggiungere una transizione: se sono in on_preconfig e sollevo il pulsante prima di 1 secondo, devo passare in on_postconfig.
- La soluzione più semplice però è modificare il codice del task 3 mettendo, dopo aver invocato turn_off_LEDs(), un’istruzione che resetta (ossia mette a false) la variabile tim7_it. In tal modo, se anche prima di aver invocato turn_of_LEDs(), il timer 7 ha generato un evento, questo viene scartato prima che il task 2 abbia la possibilità di ritornare attivo.

Per ora ci accontentiamo di questa terza soluzione, la più semplice:

main.c:

```
...
...
int main(void)
{
...
  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
    ...

    /* task 3 */
    if (button_it) {
      button_it = false;

      uint32_t currentMillis = HAL_GetTick();
      if (currentMillis - lastDebouncedButtonMillis > 20) {
        lastDebouncedButtonMillis = currentMillis;

        button_status_down = !button_status_down;
        if (button_status_down) {
          ...
        } else { /* !button_status_down */
          if (current_state == on || current_state == config) {
            current_state = on;
            stop_TIM7();
            tim7_it = false;
```

```
          } /* else do nothing */
        }
      ...
  }
```

C’è un altro effetto indesiderato, che emerge saltuariamente se premiamo il pulsante un po’ di volte: si “inverte” lo stato in funzione della pressione del pulsante, ossia quando i led sono off e premiamo il pulsante, i led non si accendono. I led si accendono quando il pulsante ritorna su, e dopo un secondo che il pulsante resta su il sistema entra in modalità di configurazione. Se a questo punto premiamo il pulsante il sistema esce dalla modalità di configurazione e torna in modalità on, se rilasciamo il pulsante entra in modalità off, se premiamo il pulsante resta in modalità off, e così via. Non è chiara la sorgente dell’effetto, ma probabilmente è un artefatto dovuto ad uno switch debouncing imperfetto. Si accettano proposte sulla possibile sorgente, e su modi per eliminarlo.