DMA

Ora modifichiamo l'esempio precedente utilizzando la lettura multicanale per acquisire anche il dato della lettura di V_{refint} e correggere il valore di temperatura con il fattore di compensazione per l'alimentazione. Saremo particolarmente "paranoici" e ricalcoleremo il fattore di compensazione ad ogni acquisizione, anche se in teoria dovrebbe restare stabile se l'alimentazione resta stabile. Questo ci permetterà di sperimentare il trasferimento in modalità DMA dei dati dalla periferica ADC1 alla memoria dell'applicazione.

Cose da fare in più rispetto al progetto precedente:

- Analog > ADC1: attivare Vrefint channel;
- Sotto DMA Settings: Add DMA, con DMA Request impostato a ADC1; cliccata la riga che è stata aggiunta compare un pannello DMA Request Settings; impostare:
  - Mode = Circular
  - Lasciare i vari data width su Half Word
- Sotto Parameter Settings > ADC_Settings: Scan Conversion Mode = Enabled, Continuous Conversion Mode = Enabled, DMA Continuous Request = Enabled
- Sotto Parameter Settings > ADC_Regular_ConversionMode: Number of conversions = 2:
  - Rank 1: Channel = Channel Vrefint, Sampling time = 480 cycles
  - Rank 2: Channel = Channel Temperature, Sampling time = 480 cycles

main.c:

```
...
/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */
#define TS_CAL1_ADDR ((uint16_t *) 0x1ff0f44c)
#define TS_CAL2_ADDR ((uint16_t *) 0x1ff0f44e)
#define VREFIN_CAL_ADDR ((uint16_t *) 0x1ff0f44a)
/* USER CODE END PD */
...
/* Private variables ---------------------------------------------------------*/
...
/* USER CODE BEGIN PV */
static volatile uint16_t raw_vals[10];
static volatile raw_vals_available;
/* USER CODE END PV */
...
int main(void)
{
  ...
  /* USER CODE BEGIN 2 */
  uint16_t ts_cal1 = *TS_CAL1_ADDR;
  uint16_t ts_cal2 = *TS_CAL2_ADDR;
  uint16_t delta_ts_cal = ts_cal2 - ts_cal1;
  uint16_t vrefin_cal = *VREFIN_CAL_ADDR;
  raw_vals_available = false;
  HAL_ADC_Start_DMA(&hadc1, (uint32_t *) raw_vals, 10);
  /* USER CODE END 2 */

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

    /* USER CODE BEGIN 3 */
    if (raw_vals_available) {
      raw_vals_available = false;
```

```
      uint16_t val_vrefint[5];
      uint16_t val_temp_raw[5];
      for (int i = 0; i < 10; i += 2) {
        val_vrefint[i / 2] = raw_vals[i];
        val_temp_raw[i / 2] = raw_vals[i + 1];
      }
      for (int i = 0; i < 5; ++i) {
        uint16_t val_temp = val_temp_raw[i] * vrefin_cal / val_vrefint[i];
        float temperature = ((float) ((((int) val_temp) - ts_cal1) * 80)) /
delta_ts_cal + 30;
        printf("t = %f C\r\n", temperature);
      }
    }
  }
  /* USER CODE END 3 */
}
...

/* USER CODE BEGIN 4 */
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef *hadc) {
  if (hadc->Instance == ADC1) {
    raw_vals_available = true;
  }
}
/* USER CODE END 4 */
```

Notare che nella chiamata ad HAL_ADC_Start_DMA la dimensione del buffer (qui 10) stabilisce quanti elementi possono essere letti. Dal momento che abbiamo impostato il data width ad half word (ossia, a 16 bit, più che sufficienti dal momento che l'ADC ha una risoluzione di 12 bit), il nostro array raw_vals deve essere dichiarato come un array di 10 elementi di tipo uint_16. Se ogni ciclo legge da due canali un valore a 16 bit, il buffer può contenere i valori di 5 cicli (quindi, 5 valori di temperatura e 5 valori di REFINT).

Un'altra osservazione importante è che il buffer va di regola allocato come variabile globale: infatti, se viene dichiarato come variabile locale di una qualche funzione, bisogna stare attenti che, nel tempo in cui il trasferimento DMA viene attivato, non si ritorni da tale funzione, e che quindi il frame della funzione non venga deallocato, e con esso il buffer.

Il codice così strutturato non è ottimale. Avere una routine che legge il buffer DMA mentre il DMA è ancora attivo, e quindi può scrivere nel buffer, potrebbe generare corse critiche, a maggior ragione se l'architettura è round robin con interrupt, e quindi può esserci un ritardo tra l'interrupt e l'attivazione del task di calcolo e stampa temperature; in questa particolare applicazione, dove abbiamo un solo task, non dovrebbe esserci significativo ritardo, ma all'aumentare dei task il ritardo potrebbe aumentare significativamente. Per tale motivo nel codice precedentemente proposto viene copiato il contenuto del buffer nelle variabili locali val_vrefint e val_temp_raw, che sono usate per l'elaborazione. Dal momento che il tempo di campionamento è di circa 40 µsec, il ciclo di copia dovrebbe auspicabilmente completare prima di questo tempo, dopo il quale il buffer inizia ad essere sovrascritto (questo nell'ipotesi che il task venga attivato in tempo zero dopo l'interrupt, altrimenti il ciclo di copia ha ancora meno tempo). Il WCET del task deve essere minore del periodo degli interrupt dell'ADC: dal momento che l'ADC effettua 10 coppie di campionamenti ogni 40 µsec, il periodo degli interrupt dell'ADC è di circa 200 µsec.

Un'implementazione migliore si può ottenere utilizzando il doppio buffering, ossia far utilizzare al DMA due buffer a turno. Mentre il DMA scrive su un buffer, il task che consuma i dati legge l'altro. Al ciclo successivo il DMA può utilizzare il buffer che il task che consuma i dati ha finito di usare, e il task che consuma i dati può utilizzare i nuovi dati che il DMA ha caricato nell'altro buffer. Per effettuare il doppio buffering lo HAL mette a disposizione la callback

HAL_ADC_ConvHalfCpltCallback, che viene triggerata quando la conversione è arrivata a riempire metà del buffer passato con la HAL_ADC_Start_DMA. In tal modo possiamo passare alla HAL_ADC_Start_DMA un buffer di dimensione doppia, e stamparne metà ogni volta che viene chiamata HAL_ADC_ConvHalfCpltCallback o HAL_ADC_ConvCpltCallback. Il codice seguente implementa il tutto sempre con un architettura round robin con interrupt.

main.c:

```
...
/* USER CODE BEGIN PV */
static uint16_t ts_cal1;
static uint16_t delta_ts_cal;
static uint16_t vrefin_cal;
static volatile uint16_t raw_vals[20];
static volatile raw_vals_available_low;
static volatile raw_vals_available_high;
/* USER CODE END PV */
...
/* USER CODE BEGIN PFP */
static void convert_and_display(bool);
/* USER CODE END PFP */
...
int main(void)
{
  ...
  /* USER CODE BEGIN 2 */
  ts_cal1 = *TS_CAL1_ADDR;
  uint16_t ts_cal2 = *TS_CAL2_ADDR;
  delta_ts_cal = ts_cal2 - ts_cal1;
  vrefin_cal = *VREFIN_CAL_ADDR;
  raw_vals_available_low = false;
  raw_vals_available_high = false;
  HAL_ADC_Start_DMA(&hadc1, (uint32_t *) raw_vals, 20);
  /* USER CODE END 2 */

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

    /* USER CODE BEGIN 3 */
    if (raw_vals_available_low) {
      raw_vals_available_low = false;
      convert_and_display(true);
    }

    if (raw_vals_available_high) {
      raw_vals_available_high = false;
      convert_and_display(false);
    }
  }
  /* USER CODE END 3 */
}
...

/* USER CODE BEGIN 4 */
void convert_and_display(bool low) {
  for (int i = (low ? 0 : 10); i < (low ? 10 : 20); i += 2) {
    uint16_t val_vrefint = raw_vals[i];
    uint16_t val_temp_raw = raw_vals[i + 1];
    uint16_t val_temp = val_temp_raw * vrefin_cal / val_vrefint;
    float temperature = ((float) ((((int) val_temp) - ts_cal1) * 80)) /
delta_ts_cal + 30;
```

```
    printf("t = %f C\r\n", temperature);
  }
}

void HAL_ADC_ConvHalfCpltCallback(ADC_HandleTypeDef *hadc) {
  if (hadc->Instance == ADC1) {
    raw_vals_available_low = true;
  }
}

void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef *hadc) {
  if (hadc->Instance == ADC1) {
    raw_vals_available_high = true;
  }
}
...
```

Notare che stavolta non vi è più bisogno di fare una copia di sicurezza locale dei dati del buffer, dal momento che mentre il task elabora metà del buffer il DMA scrive sull'altra metà, e quindi la metà del buffer usata dal task ha i dati stabili per tutto il tempo che il DMA ci mette a riempire la seconda metà del buffer, tempo che è sempre 200 µsec. Ovviamente il tutto funziona correttamente se il tempo di calcolo temperatura e stampa di metà del buffer (WCET del task) è minore del tempo che la lettura ADC e il trasferimento DMA ci mettono a riempire l'altra metà del buffer, ossia 200 µsec. Stavolta il task risparmia il tempo necessario per la copia locale del buffer che utilizza, e quindi il suo WCET scende.