Pasta Roberto 869247

Progetto del corso di Sistemi Embedded 2022/2023

Capitolo 1: Breve introduzione al progetto

Il progetto assegnato consiste nel predisporre un sottosistema di acquisizione dati per il controllo del volo di un drone ad ala rotante. Il drone integra una scheda Nucleo F767ZI per il controllo e l'acquisizione ed elaborazione dei dati e lo schield sensoristico X-NUCLEO-IKS01A2, composto da:

- LSM6DSL: accelerometro e giroscopio;
- LSM303AGR: accelerometro e magnetometro;
- LPS22HB: pressione barometrica.

Il software deve essere sviluppato partendo da una base già data, utilizzando il sistema operativo Real-Time $FreeRTOS$ con scheduling Rate Monotonic.

Capitolo 2: Configurazione iniziale della scheda Nucleo

- La componente $FreeRTOS$ è stata attivata già nel codice di base da cui sono partito e impostato il Timbase su TIM1;
- Ho selezionato l'opzione <Generate peripheral initialization as a pair of '.c/.h' files per peripheral> nella sezione Project Manager, così da spostare le componenti di $FreeRTOS$ nel file $freertos.c$ per rendere il codice più funzionale e comprensibile;
- Ho impostato i due pin per la comunicazione I2C (SDA e SCL) sui pin PB8 e PB9 e ho attivato nella sezione Connectivity il modulo I2C1 ad una frequenza di 400 kHZ (Fast Mode);
- Ho attivato, sempre nella sezione Connectivity, USART3 sui pin PD8 e PD9 per stampare i valori sulla seriale.

Capitolo 3: Componenti aggiuntive del progetto

Driver per i tre sensori:

- **lps22hb_reg.h, lps22hb_reg.c**: Driver per il sensore LPS22HB;
- **lsm303agr_reg.h, lsm303agr_reg.c**: Driver per il sensore LSM303AGR:
- **lsm6dsl_reg.h, lsm6dsl_reg.c**: Driver per il sensore LSM6DSL;

Questi sei file li ho presi dalla cartella GitHub di STMicroelectronics condivisa su Moodle dal prof.

Interfacce per i driver dei tre sensori:

- **lps22hb_interface.h, lps22hb_interface.c**: Interfaccia per il sensore LPS22HB;
- **lsm303agr_interface.h, lsm303agr_interface.c**: Interfaccia per il sensore LSM303AGR:
- **lsm6dsl_interface.h, lsm6dsl_interface.c**: Interfaccia per il sensore LSM6DSL;

Nel caso di questi tre file ho preso spunto dalla sezione examples nella cartella GitHub soprariportata. In particolare, creo delle funzioni di $Init()$ e di $Read()$ che successivamente chiamo per inizializzare e leggere dai sensori.

Header per le variabili globali che contengono il valore letto dai sensori:

- **global_variables.h**

Questo file header contiene le sei variabili globali che serviranno per raccogliere e inviare i dati dei sensori. Per poter salvare in maniera più efficace i dati tridimensionali ho definito una nuova struttura $data3xFloat$ che è formata da tre variabili float, precisamente una variabile che rappresenta la x, una la y e una la z. Da notare che le variabili le ho dichiarate tutte $extern$ per poter essere utilizzate a livello globale.

Header per le variabili temporali delle frequenze:

- **delays.h**

All'interno di questo file header sono definite le costanti di tempo utilizzate nel file $freertos.c$ per impostare il valore della frequenza con il quale ogni task deve essere chiamato (utilizzando $osDelay()$).

Header per la dichiarazione dei sensori a livello globale:

- **bin_semaphores.h**

Questo file header contiene la dichiarazione dei sette semafori binari che vengono usati per ottenere i lock sulle variabili globali e sulla comunicazione I2C. Di base $FreeRTOS$ li dichiara in automatico nel file $freertos.c$, ma in questo caso ho deciso di spostarli per poterli utilizzare anche nel file $main.c$ e nei file di interfaccia dei sensori dove si trovano le funzioni che consumano e acquisiscono i dati delle variabili globali e poter diminuire la regione critica al minimo.

Capitolo 4: Tabella dei Task e discussione del WCET

Nome del task	Priorità	Frequenza	Ruolo
startTaskControlMotor (già implementato)	osPriorityNormal6	500 HZ	Consuma in input il valore dell'accelerazione angolare
startTaskAttitude (già implementato)	osPriorityNormal3	100 HZ	Consuma in input il valore dell'accelerazione lineare e quello del campo magnetico
startTaskAltitude (già implementato)	osPriorityNormal1	40 HZ	Consuma in input il valore della pressione barometrica
getPressure	osPriorityNormal2	50 HZ	Acquisisce la pressione barometrica dal sensore LPS22HB
getMagnetometer	osPriorityNormal4	125 HZ	Acquisisce il flusso del campo magnetico dal sensore LSM6DSL
getAccMean	osPriorityNormal5	125 HZ	Acquisisce il valore dei due accelerometri lineari e ne calcola la media
getGyroscope	osPriorityNormal7	1000 HZ	Acquisisce l'accelerazione angolare dal sensore LSM6DSL
displayData	osPriorityNormal	0,5 HZ	Trasmette su USART i valori letti dai sensori

I primi tre e l'ultimo task della tabella hanno una frequenza fissata dal progetto, per tutti gli altri ho deciso la loro frequenza in base a quella dei primi, ad esempio avendo $\boldsymbol{startTaskAltitude}$ (il task che consuma il dato della pressione barometrica) una frequenza di 40 HZ ho impostato la frequenza di $\boldsymbol{getPressure}$ (il task che acquisisce la pressione barometrica) ad un valore leggermente superiore: 50 HZ per fare in modo che il dato consumato sia sempre nuovo, e così anche per gli altri task di lettura dal sensore.

Le priorità dei task le ho decise seguendo la regola di Rate Monotonic: task con una frequenza più elevata devono avere anche una priorità più alta.

Le frequenze dei task sono definite in tick (file delays.h), la conversione in Hertz è calcolata con la seguente formula:

$$\frac{\boldsymbol{R}\ \left[\frac{\boldsymbol{tick}}{\boldsymbol{s}}\right]}{\boldsymbol{t_i}[\boldsymbol{tick}]} = \boldsymbol{f_i}[\boldsymbol{HZ}]$$

Dove $\boldsymbol{R}$ è pari al valore del tick rate (1000) e $\boldsymbol{t_i}$ è il periodo del task i.

Per dimostrare che il progetto dello scheduling permette di rispettare i requisiti real-time, occorre calcolare il WCET della comunicazione dal bus I2C, mentre, come stabilito nella traccia del progetto, tutti gli altri tempi sono trascurabili. Le informazioni sulla comunicazione I2C tra shield e microcontrollore contenute nel datasheet mostrano che nel mio caso vengono trasferiti 50 bit (in realtà sono 47 bit, ma arrotondo in eccesso per avere più margine di errore) per ogni lettura dal sensore; considerando di avere impostato la velocita del bus a 400kHZ, posso concludere che il WCET di ogni lettura dati è

$\frac{50\ bit}{400\ kHZ} = 0{,}000125$ secondi.

Il test di schedulabilità dell'algoritmo Rate Monotonic si può fare applicando la formula dell'hyperbolic bound in modo seguente:

$$\prod_{i=0}^{n}(U_i + 1) \leq 2$$

Perciò:

per il task $\boldsymbol{startTaskControlMotor}$: $(500Hz * 90usec) + 1 = 1{,}045$

per il task $\boldsymbol{startTaskAttitude}$: $(100\ Hz\ *\ 120usec)\ +\ 1\ =\ 1{,}012$

per il task $\boldsymbol{startTaskAltitude}$: $(40\ Hz\ *\ 60usec)\ +\ 1\ =\ 1{,}0024$

per il task $\boldsymbol{getPressure}$: $(50\ Hz\ *\ 125usec)\ +\ 1\ =\ 1{,}00625$

per il task $\boldsymbol{getMagnetometer}$: $(125\ Hz\ *\ 125usec)\ +\ 1\ =\ 1{,}015$

per il task $\boldsymbol{getAccMean}$: $(125\ Hz\ *\ 125usec)\ +\ 1\ =\ 1{,}015$

per il task $\boldsymbol{getGyroscope}$: $(1000\ Hz\ *\ 125usec)\ +\ 1\ =\ 1{,}125$

Nel progetto, sono presenti regioni critiche date dai semafori e queste potrebbero aumentare il WCET di alcuni task in caso di semaforo bloccato, ma considerando che nelle funzioni di $osSemaphoreAcquire()$ ho impostato 0 come valore di attesa del semaforo questo caso non si verifica.

Per quanto riguarda il task $\boldsymbol{displayData}$, questo ha un impatto trascurabile sulla proprietà di schedulabilità del sistema avendo una frequenza molto più bassa rispetto a tutti gli altri task.

La formula si riduce a $1{,}236 \leq 2$ che è verificato; perciò, posso concludere che l'applicazione è schedulabile con l'algoritmo Rate Monotonic.

Per quanto riguarda il task $displayData$ mi rimane una considerazione da fare: inizializzandolo come tutti gli altri task, dopo poco tempo che facevo partire il debug, l'applicazione non stampava più. Più tardi mi sono reso conto che le operazioni di $HAL\_UART\_Transmit()$ nel task hanno bisogno di più memoria per eseguire, per questo nella pagina di configurazione dei task in $FreeRTOS$ ho aumentato lo stack size di $displayData$ da 128 a 1000 Words, questo valore è stato scelto perché abbastanza grande da far funzionare correttamente la stampa.

Come si vede, nonostante $Display$ occupi più memoria rispetto agli altri task, rimane della memoria disponibile per eventuali task aggiuntivi.

Summary	
HEAP STILL AVAILABLE	6200 Bytes
TOTAL HEAP USED	9160 Bytes
Total amount for tasks	8544 Bytes
Total amount for queues	0 Bytes
Total amount for timers	0 Bytes
Total amount for mutexes and semaphores	616 Bytes
Total amount for events	0 Bytes
FreeRTOS tasks	
Idle task (FreeRTOS internal)	0 Bytes
Timer service task (FreeRTOS internal)	0 Bytes
tControlMotor	632 Bytes
tAttitude	632 Bytes
tAltitude	632 Bytes
LBS22HB_pressio	632 Bytes
LSM303AGR_magne	632 Bytes
Media_Accellero	632 Bytes
LSM6DSL_girosco	632 Bytes
Display	4120 Bytes

Capitolo 5: Spiegazione del codice

L'inizializzazione delle periferiche presenti sul bus I2C avviene nel $main.c$ chiamando le rispettive funzioni di $init()$:

```
115   LSM303AGR_init();
116   LSM6DSL_init();
117   LPS22HB_init();
```

Subito dopo avviene l'inizializzazione degli oggetti usati da $FreeRTOS$:

```
134   osKernelInitialize();
135   MX_FREERTOS_Init();
```

La funzione $osKernelStart()$ inizializza lo scheduling.

Il codice eseguibile di ogni task nel file $freertos.c$ è sviluppato in modo che:

1. avviene l'acquisizione del semaforo riferito alla comunicazione I2C dal task;
2. viene chiamata la funzione $dataRead()$ per leggere il valore del sensore;
3. finita la lettura viene rilasciato il lock sul bus;
4. viene messo in pausa il task per il periodo di tempo stabilito.

```c
void getPressure(void *argument)
{
  /* USER CODE BEGIN getPressure */
  /* Infinite loop */
  for(;;)
  {
   if(osSemaphoreAcquire(binSemaphoreI2CHandle, 0)==osOK){

       LPS22HB_dataRead();

       osSemaphoreRelease(binSemaphoreI2CHandle);
      }

    osDelay(TASK_PRESSIONE_PERIOD);
  }
  /* USER CODE END getPressure */
}
```

Nelle funzioni di $dataRead()$ e $sensorRead()$ ho utilizzato i semafori per ottenere il lock delle variabili globali in fase di scrittura e lettura del valore del sensore. La funzione $osSemaphoreAcquire()$ restituisce il valore 0 (NB $osOK$ = 0) quando ottiene il semaforo a tempo 0.

```c
void LSM6DSL_dataReadGyro() {
    /* Read output only if new value is available */
    lsm6dsl_reg_t reg;
    lsm6dsl_status_reg_get(&lsm6dsl_dev_ctx, &reg.status_reg);
    int16_t temp[3];  //variabile temporanea
    if (reg.status_reg.gda) {
        lsm6dsl_angular_rate_raw_get(&lsm6dsl_dev_ctx, temp);
        if(osSemaphoreAcquire(binSemaphoreGyrHandle, 0)==osOK){
            gyroscope.x = lsm6dsl_from_fs2000dps_to_mdps(temp[0]);
            gyroscope.y = lsm6dsl_from_fs2000dps_to_mdps(temp[1]);
            gyroscope.z = lsm6dsl_from_fs2000dps_to_mdps(temp[2]);
            osSemaphoreRelease(binSemaphoreGyrHandle);
        }
    }
}
```

Capitolo 5: Esempio di funzionamento

Qui riporto una stampa dei dati letti dai sensori mentre la scheda nucleo era appoggiata sul tavolo di lavoro:

```
dati velocita' angolare        x=0.00 y=0.00 z=0.00
dati accellerometro 1          x=-1.30 y=-1.80 z=100.40
dati accellerometro 2          x=0.20 y=0.00 z=100.87
media dati accellerometri      x=-0.55 y=-0.90 z=100.63
dati flusso magnetico          x=87.00 y=-166.50 z=-427.50
dati pressione                 p=1002.06
```

I dati mostrano sull'asse z dei due accelerometri il valore 100, considerando che è espresso in $\frac{m}{s^2}$ il suo valore equivalente nel vettore accelerazione di gravità è circa 10, il che più o meno rispecchia il valore della forza di gravità. I valori letti dal giroscopio mostrano una velocità angolare nulla in quanto la scheda era ferma sul tavolo.