

ESAME SISTEMI EMBEDDED

Francesco Ambrogio Marinoni Matr.869276

Il progetto fonda le sue radici nell'acquisizione di dati da un sistema software di controllo di volo per un drone che integra una scheda **Nucleo F767ZI** e uno shield di espansione **X-NUCLEO-IKS01A2** (il sistema stesso è eseguito dalla **MCU** della scheda nucleo).

Il software fa uso del sistema operativo **FreeRTOS** con scheduling **rate monotonic** e a noi è stato chiesto di implementare dei task che potessero raccogliere i dati dai sensori dello shield **X-NUCLEO-IKS01A2**:

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

e poi inviare i dati recuperati da questi sensori alle 3 task precedentemente fornite dal professor Braione che, come detto nella traccia:

- Sottosistema di controllo motori: controlla i motori per stabilizzare il volo del drone in condizioni stazionarie e per far reagire il drone ai comandi del pilota in maniera che il drone si sposti nella direzione che il pilota ha indicato.
- Sottosistema di calcolo dell'assetto: calcola l'orientamento assoluto del drone rispetto al riferimento North-East-Down (NED) utilizzando lo stimatore di Mahony.
- Sottosistema di calcolo dell'altitudine: calcola l'altitudine del drone rispetto al livello del mare.

e comporre, inoltre, un ultimo task finalizzato alla stampa dei dati con l'ordine sottocitato (come affermato nella traccia):

- Prima riga: le tre componenti del vettore tridimensionale delle velocità angolari del giroscopio; le velocità angolari vanno visualizzate con due cifre decimali;
- Riga successiva: le tre componenti del vettore tridimensionale delle accelerazioni del primo accelerometro (dato prodotto dal sensore LSM6DSL); le accelerazioni vanno visualizzate con due cifre decimali;
- Riga successiva: le tre componenti del vettore tridimensionale delle accelerazioni del secondo accelerometro (dato prodotto dal sensore LSM303AGR); le accelerazioni vanno visualizzate con due cifre decimali;
- Riga successiva: le tre componenti del vettore tridimensionale delle accelerazioni (media dei vettori prodotti dai due accelerometri); le accelerazioni vanno visualizzate con due cifre decimali;
- Riga successiva: le tre componenti del vettore tridimensionale delle densità di flusso magnetico del magnetometro; le densità di flusso magnetico vanno visualizzate con due cifre decimali;

- Riga successiva: la pressione del sensore di pressione barometrica; la pressione va visualizzate con una cifra decimale;
- Riga successiva: riga bianca (per separare la successiva stampa).

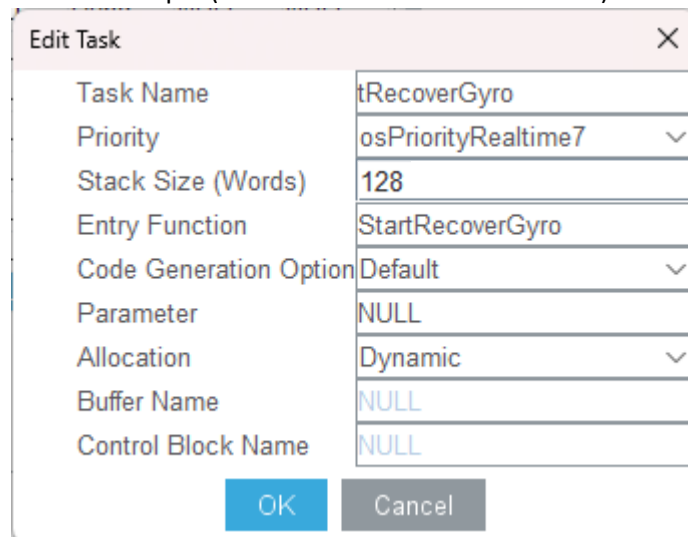
Utilizzando l'ide **STM32CubeIDE** con **FreeRTOS** abilitato, abbiamo la possibilità di creare delle task di cui ci viene fornito automaticamente, a livello di codice, una interfaccia vuota contenente un comando `osDelay()` e il `for` che compone la task.

Allora seguendo la traccia e componendo i task si può iniziare a ragionare sull'ordine di priorità che avranno i task stessi che, nel caso del mio progetto, saranno:

- I task di recupero dei dati.
- I task di update dei dati.
- Stampa (ultima come affermata dalla traccia stessa del progetto).

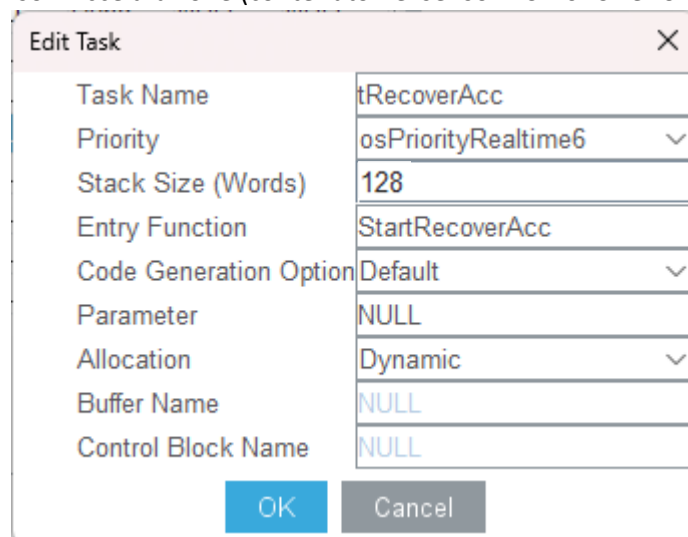
Partendo quindi dalla traccia e seguendo l'idea dello scheduling **Rate Monotonic** possiamo affermare che i task saranno ordinati più nello specifico:

1. Recupero dati dal sensore Giroscopio (contenuto nel sensore LSM6DSL)



Edit Task	
Task Name	tRecoverGyro
Priority	osPriorityRealtime7
Stack Size (Words)	128
Entry Function	StartRecoverGyro
Code Generation Option	Default
Parameter	NULL
Allocation	Dynamic
Buffer Name	NULL
Control Block Name	NULL
<div>OK Cancel</div>	

2. Recupero dati dai sensori Accelrazione (contenuto nei sensori LSM6DSL e LSM303AGR)



Edit Task	
Task Name	tRecoverAcc
Priority	osPriorityRealtime6
Stack Size (Words)	128
Entry Function	StartRecoverAcc
Code Generation Option	Default
Parameter	NULL
Allocation	Dynamic
Buffer Name	NULL
Control Block Name	NULL
<div>OK Cancel</div>	

3. Recupero dati dal sensore Pressione (contenuto nel sensore LPS22HB)

Edit Task		✕
Task Name	tRecoverPress	
Priority	osPriorityRealtime5 ▾	
Stack Size (Words)	128	
Entry Function	StartRecoverPress	
Code Generation Option	Default ▾	
Parameter	NULL	
Allocation	Dynamic ▾	
Buffer Name	NULL	
Control Block Name	NULL	
<input type="button" value="OK"/>		<input type="button" value="Cancel"/>

4. Recupero dati dal sensore Magnetometro (contenuto nel sensore LSM303AGR)

Edit Task		✕
Task Name	tRecoverMag	
Priority	osPriorityRealtime4 ▾	
Stack Size (Words)	128	
Entry Function	StartRecoverMag	
Code Generation Option	Default ▾	
Parameter	NULL	
Allocation	Dynamic ▾	
Buffer Name	NULL	
Control Block Name	NULL	
<input type="button" value="OK"/>		<input type="button" value="Cancel"/>

5. Poi il task di controllo motori fornito dal professor Braione all'interno della traccia

Edit Task		✕
Task Name	tControlMotor	
Priority	osPriorityRealtime3 ▾	
Stack Size (Words)	128	
Entry Function	startTaskControlMotor	
Code Generation Option	Default ▾	
Parameter	NULL	
Allocation	Dynamic ▾	
Buffer Name	NULL	
Control Block Name	NULL	
<input type="button" value="OK"/>		<input type="button" value="Cancel"/>

6. Poi il task di calcolo dell'assetto fornito dal professor Braione all'interno della traccia

The 'Edit Task' dialog box is shown with the following fields:

Field	Value
Task Name	tAttitude
Priority	osPriorityRealtime2
Stack Size (Words)	128
Entry Function	startTaskAttitude
Code Generation Option	Default
Parameter	NULL
Allocation	Dynamic
Buffer Name	NULL
Control Block Name	NULL

Buttons: OK, Cancel

7. Poi il task di calcolo di calcolo dell'altitudine dal professor Braione all'interno della traccia

The 'Edit Task' dialog box is shown with the following fields:

Field	Value
Task Name	tAltitude
Priority	osPriorityRealtime1
Stack Size (Words)	128
Entry Function	startTaskAltitude
Code Generation Option	Default
Parameter	NULL
Allocation	Dynamic
Buffer Name	NULL
Control Block Name	NULL

Buttons: OK, Cancel

8. E infine, il task di stampa dei dati raccolti

The 'Edit Task' dialog box is shown with the following fields:

Field	Value
Task Name	PrintData
Priority	osPriorityRealtime
Stack Size (Words)	128
Entry Function	StartPrintData
Code Generation Option	Default
Parameter	NULL
Allocation	Dynamic
Buffer Name	NULL
Control Block Name	NULL

Buttons: OK, Cancel

Si ottiene questo ordine di priorità poiché, come definito dall'algoritmo Rate Monotonic, le task si ordinano in modo che quella avente il periodo più breve o la frequenza più alta abbia la priorità maggiore; quindi seguendo le tre categorie dette in precedenza:

- Recupero dei dati:
 1. Giroscopio 500 Hz o 0.002 sec (per calcolare da Hertz a secondi si deve fare 1/frequenza in Hertz)
 2. Accelerometri 100 Hz o 0.01 sec
 3. Barometro 20 Hz o 0.05 sec
 4. Magnetometro 10 Hz o 0.1 sec
- update dei dati:
 1. controllo motori 500 Hz o 0.002 sec
 2. calcolo dell'assetto 100 Hz o 0.01 sec
 3. calcolo dell'altitudine 40 Hz o 0.025 sec
- Stampa 0.5 Hz o 2 sec

Quindi, create le task, settiamo gli `osDelay()` con il periodo in millisecondi pertanto, prendendo i secondi precedenti e moltiplicandoli per mille.

A questo punto però dobbiamo calcolare se è possibile svolgere lo scheduling con Rate Monotonic poiché quest'ultimo ha la possibilità di calcolare la garanzia di successo quindi, dovremo attuare:

$$U_b = \sum \frac{w_{cet}}{P} =$$

Avremo bisogno del periodo e del W_{cet} fornitoci dalla traccia

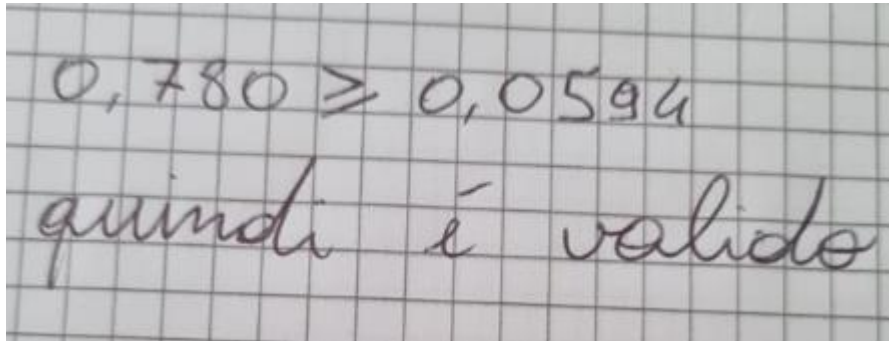
$$\begin{aligned}
 U_b &= \sum \frac{w_{cet}}{P} = \frac{90 \mu s}{0,002 s} + \frac{120 \mu s}{0,01} + \frac{60 \mu s}{0,025 s} = \\
 &= \frac{90 \mu s}{2000 \mu s} + \frac{120 \mu s}{10000 \mu s} + \frac{60 \mu s}{25000 \mu s} = \\
 &= 0,045 + 0,012 + 0,0024 = 0,0594
 \end{aligned}$$

Di seguito, confrontiamo il risultato di uscita con la tabella dei valori del Rate Monotonic:

n	U_{lub}
1	1.000
2	0.828
3	0.780
4	0.757
5	0.743

n	U_{lub}
6	0.735
7	0.729
8	0.724
9	0.721
10	0.718

Infine avendo tre task:



Arrivati a questo punto, iniziamo a comporre il passaggio dei dati tra la raccolta dati stessa, l'update e la stampa per i vari sensori:

- Delle struct per l'interazione col sensore fornite dall'api dei sensori stessi, tra i quali:
 - i. **LSM6DSL_Acc**
 - ii. **LSM303AGR_Acc**
 - iii. **LSM303AGR_Mag**
 - iv. **LSM6DSL_Gyro**

Le prime tre sono di tipo **IKS01A2_MOTION_SENSOR_Axes_t** poiché queste struct contengono i valori tridimensionali delle computazioni dei sensori non raw, invece, l'ultima è di tipo **IKS01A2_MOTION_SENSOR_AxesRaw_t** poiché per il giroscopio usiamo i dati raw ma, la motivazione della scelta, sarà giustificata più avanti nella relazione in quanto legata alle funzioni.

- Degli array o valori per trasportare i dati dai sensori alle funzioni update tra cui:
 - i. **VectAcc_LSM303AGR**
 - ii. **VectAcc_LSM6DSL**
 - iii. **VectPress**
 - iv. **VectMag**
 - v. **VectGyro**

Il numero **i**, **ii**, **iv**, **v** sono tutti degli array di **float** composti da tre slot in quanto salvano i dati delle strutture tridimensionali invece, il **iii** è un normale **float** poiché le variabili ambientali sono unidimensionali come la temperatura o, nel nostro caso, la pressione.

Identificate le struct e gli array da utilizzare, estraiamo dal datasheet dello shield **X-NUCLEO-IKS01A2** e dalla documentazione fornitaci dall'ide le funzioni di estrazioni dati da inserire all'interno dei task di raccolta dati stessi che:

- Per gli accelerometri, il magnetometro e il giroscopio, essendo sensori di movimento, si userà per i primi 2

**IKS01A2_MOTION_SENSOR_GetAxes (uint32_t Instance, uint32_t Function,
IKS01A2_MOTION_SENSOR_Axes_t *Axes)**

E per il giroscopio

**IKS01A2_MOTION_SENSOR_GetAxesRaw (uint32_t Instance, uint32_t Function,
IKS01A2_MOTION_SENSOR_AxesRaw_t *Axes)**

Per il giroscopio ho scelto di raccogliere i dati in formato raw a causa di un problema derivante dall'utilizzo del codice poiché, quando si utilizzava la debug-mode con il getAxes, il codice andava in Hardfault come errore anche se la motivazione però non è certa e, di conseguenza, ho estratto i dati in formato raw.

- Per il barometro, essendo un sensore ambientale, si utilizzerà
IKS01A2_ENV_SENSOR_GetValue (uint32_t Instance, uint32_t Function, float *Value)

Per queste funzioni, in tutti i casi, viene chiesto un **instance** che si modifica a seconda del sensore e, dunque, per le varie casistiche utilizzeremo:

- **IKS01A2_LSM6DSL_0**
- **IKS01A2_LSM303AGR_ACC_0**
- **IKS01A2_LSM303AGR_MAG_0**
- **IKS01A2_LPS22HB_0**

Inoltre, a seconda dei sensori specifici per ogni sensore integrato nella scheda, utilizzeremo come **function**:

1. per i sensori di movimento:
 - **MOTION_GYRO** e/o **MOTION_ACCELERO** per **LSM6DSL**
 - **MOTION_ACCELERO** per **LSM303AGR**
 - **MOTION_MAGNETO** per **LSM303AGR**
2. Per quello ambientale:
 - **ENV_PRESSURE** per **LPS22HB**

Nell'ultimo slot che sia l'axes o che sia il value utilizzavo le strutture citate a pagina 5 e dopo aver preso quelle stesse strutture ed aver sovrascritto la X, Y, Z (coordinate tridimensionali) negli array specifici, citati sempre a pagina 5.

Dopo aver attuato l'acquisizione dei dati, è stato necessario modificare le funzioni **sensorGyroRead**, **sensorAccRead**, **sensorMagRead**, **sensorBaroRead** in cui erano stati posti dei valori fittizi di update quindi, ho aggiunto che **gyro->gyroADC**, **acc->accADC**, **mag->magADC**, **baro->baroADC** debbano prendere i dati direttamente dagli array che noi aggiorniamo nella raccolta dei dati stessa.

Visto che in questo punto possiamo constatare che c'è un uso conteso degli array e, soprattutto per recuperare determinati dati dobbiamo utilizzare più volte gli stessi sensori, allora è necessario introdurre il concetto di semafori binari e di mutua esclusione.

Un semaforo binario è un tipo di dato astratto utilizzato per sincronizzare più task o attività del sistema operativo.

I semafori binari, più nello specifico, sono chiamati così perché si basano sull'utilizzo di valori 1 o 0.

La mutua esclusione è invece un problema che sorge quando più di un processo o task vogliono accedere ad una o più variabili comuni che, se si sovrappongessero temporalmente, potrebbero dare origine a fault o a deadlock e stalli.

Per prevenire il problema sopracitato uno degli approcci consiste nell'utilizzo dei semafori binari e, in questo progetto ne ho creati 6, uno per ogni casistica di sovrapposizione per le varie possibili chiamate del codice:

- **Sem_I2C**
- **Sem_LSM6DSL_Acc**
- **Sem_LSM303AGR_Acc**
- **Sem_Mag**
- **Sem_Gyro**
- **Sem_Press**

Il **Sem_I2C** è per le chiamate di Get dei dati poiché esse utilizzano tutte la porta I2C1 per recuperare i dati stessi quindi, non potendo tutte leggere allo stesso tempo, hanno bisogno di un semaforo che gestisca che la porta sia occupata o meno.

I semafori successivi sono finalizzati alla possibilità di lettura e scrittura dei dati poiché noi li salviamo dalla struct negli array e poi dobbiamo passarli all'update per inserirli nelle funzioni forniteci dal Professor Braione quindi, potrebbe accadere che, senza un semaforo, si stia scrivendo nell'array e, prima che concluda di scrivere, i dati vengano passati per eseguire l'update ma che quegli stessi siano già obsoleti.

Questi semafori serviranno anche successivamente per attuare la stampa poiché la stampa stessa utilizzerà gli array sopracitati per leggere le informazioni da stampare ma, di questo, ne parleremo successivamente.

Un semaforo viene acquisito con la funzione, se non occupato:

osSemaphoreAcquire (osSemaphoreId_t semaphore_id, uint32_t timeout)

Nel **semaphore_id** dovremo inserire **osSemaphoreId_t** che nel caso di **STM32CubeIDE** viene generata in automatico dall'**ioc** appena si creerà nel reparto di **FreeRTOS** un semaforo.

Un esempio per il caso del semaforo **Sem_I2C** sarà **Sem_I2CHandle**.

Il **timeout** come enunciato nella documentazione di CMSIS-RTOS:

"The parameter *timeout* specifies how long the system waits to acquire the token. While the system waits, the thread that is calling this function is put into the BLOCKED state. The parameter timeout can have the following values:

- when *timeout* is 0, the function returns instantly.
- when *timeout* is set to `osWaitForever` the function will wait for an infinite time until the semaphore becomes available.
- all other values specify a time in kernel ticks for a timeout."

Ho scelto di utilizzare timeout 0 poiché esso col metodo "try and error" funzionava in quanto il tempo di computazione delle funzioni era breve quindi avrebbe permesso di svolgere il tutto senza problemi di stalling.

Dopo aver acquisito il semaforo, per rilasciarlo, si utilizza il comando:

osSemaphoreRelease (osSemaphoreId_t semaphore_id)

Ovviamente nel **semaphore_id** dovremo inserire il semaforo che stavamo utilizzando per rilasciarlo e dare la possibilità ad altri task di richiedere la variabile bloccata.

Infine tratteremo della Stampa.

La stampa è un task che si struttura in più printf che si compongono nell'ordine citato a conclusione di pagina 1.

Per ottenere la stampa si utilizzano una serie di printline gestite dai semafori, come precedentemente affermato, poiché dovendo leggere i dati che sono presenti all'interno dell'array questa azione deve essere eseguita dopo che i dati siano già stati aggiornati e salvati e, di conseguenza, che le variabili siano libere.

```
void StartPrintData(void *argument)
{
    /* USER CODE BEGIN StartPrintData */
    /* Infinite loop */
    for(;;)
    {
        if(osSemaphoreAcquire(Sem_GyroHandle, 0) == osOK) {
            printf("Gyroscope tridimensional data x = %.2f, y = %.2f, z = %.2f in rad*sec^-1\n", VectGyro[0], VectGyro[1], VectGyro[2]);
            osSemaphoreRelease(Sem_GyroHandle);
        }
        if(osSemaphoreAcquire(Sem_LSM6DSL_AccHandle, 0) == osOK) {
            printf("LSM6DSL Accelerometer tridimensional data x = %.2f, y = %.2f, z = %.2f in m*sec^-2\n", VectAcc_LSM6DSL[0], VectAcc_LSM6DSL[1], VectAcc_LSM6DSL[2]);
            osSemaphoreRelease(Sem_LSM6DSL_AccHandle);
        }
        if(osSemaphoreAcquire(Sem_LSM303AGR_AccHandle, 0) == osOK) {
            printf("LSM303AGR Accelerometer tridimensional data x = %.2f, y = %.2f, z = %.2f in m*sec^-2\n", VectAcc_LSM303AGR[0], VectAcc_LSM303AGR[1], VectAcc_LSM303AGR[2]);
            osSemaphoreRelease(Sem_LSM303AGR_AccHandle);
        }
        if(osSemaphoreAcquire(Sem_LSM303AGR_AccHandle, 0) == osOK) {
            if(osSemaphoreAcquire(Sem_LSM6DSL_AccHandle, 0) == osOK) {
                float variable_append0 = (VectAcc_LSM303AGR[0] + VectAcc_LSM6DSL[0]) / 2;
                float variable_append1 = (VectAcc_LSM303AGR[1] + VectAcc_LSM6DSL[1]) / 2;
                float variable_append2 = (VectAcc_LSM303AGR[2] + VectAcc_LSM6DSL[2]) / 2;
                osSemaphoreRelease(Sem_LSM6DSL_AccHandle);
                printf("Mean Accelerometer tridimensional data x = %.2f, y = %.2f, z = %.2f in m*sec^-2\n", variable_append0, variable_append1, variable_append2);
            }
            osSemaphoreRelease(Sem_LSM303AGR_AccHandle);
        }
        if(osSemaphoreAcquire(Sem_MagHandle, 0) == osOK) {
            printf("Magnetometer tridimensional data x = %.2f, y = %.2f, z = %.2f in uT\n", VectMag[0], VectMag[1], VectMag[2]);
            osSemaphoreRelease(Sem_MagHandle);
        }
        if(osSemaphoreAcquire(Sem_PressHandle, 0) == osOK) {
            printf("Barometer data = %.1f in hPa\n", VectPress);
            osSemaphoreRelease(Sem_PressHandle);
        }
        osDelay(2000);
    }
    /* USER CODE END StartPrintData */
}
```

Inoltre, per poter eseguire la stampa, il codice ha bisogno della funzione `_write` che si compone:

```
int _write(int file, char *ptr, int len) {
    for (int i = 0; i < len; ++i) {
        ITM_SendChar(*ptr++);
    }
    return len;
}
```

Poiché questa funzione è utilizzata per scrivere dati su un dispositivo di output, noi la abbiamo scritta per poter inserire i dati stessi sulla **SWM ITM Data Console**.

Dopo aver composto la stampa e aver visualizzato i risultati si è potuto notare che essi non fossero conformi alle unità di misura richieste quindi, è stata attuata una conversione, partendo dall'unità di misura specificata sul datasheet e trasformandola in quelle segnalate sulla traccia.

Per l'accelerometro ho dovuto convertirlo da mg a metri al secondo quadrato e, per farlo, è bastato svolgere:

$$(\text{dato recepito dal sensore convertito in float}) / 1000 * 9.81$$

Per il magnetometro il sistema restituisce i dati in Gauss e deve essere convertito in μT e questo è possibile tramite:

$$(\text{dato recepito dal sensore}) * 100$$

Per il barometro la conversione non è necessaria poiché già restituisce i dati in hPa.

Per il giroscopio abbiamo dovuto trasformare da dps (o 100 gradi/sec) in rad/sec e questo è stato possibile tramite:

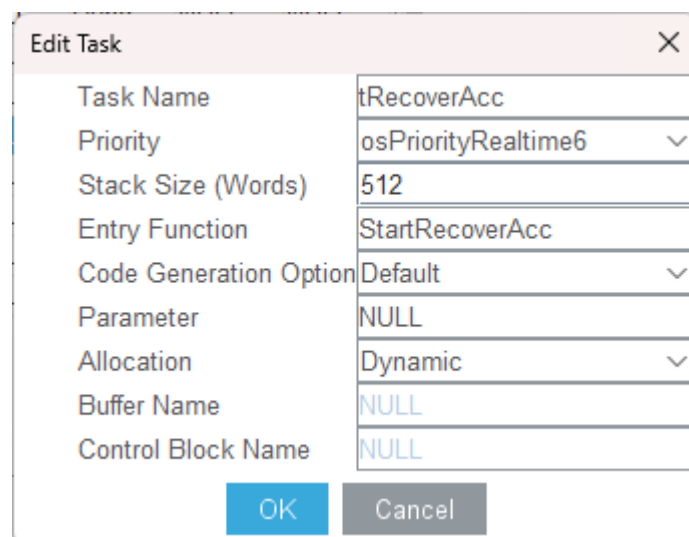
$$(\text{dato recepito dal sensore convertito in float}) * 100 * 0.0175$$

Così facendo otterremo come stampa:

```
Gyroscope tridimensional data x = -19.25, y = -7.00, z = 54.25 in rad*sec^-1
LSM6DSL Accelerometer tridimensional data x = 0.06, y = -0.13, z = 10.12 in m*sec^-2
LSM303AGR Accelerometer tridimensional data x = 0.07, y = -0.19, z = 9.63 in m*sec^-2
Mean Accelerometer tridimensional data x = 0.06, y = -0.16, z = 9.88 in m*sec^-2
Magnetometer tridimensional data x = 40500.00, y = -23200.00, z = -48000.00 in MicroT
Barometer data = 994.8 in hPa
```

Dunque, dopo aver testato la stampa con la **printf** come richiesto dalla traccia, dobbiamo utilizzare per stampare **UART (Universal Synchronous-Asynchronous Receiver/Transmitter)** il quale è un dispositivo hardware, che nel progetto è utilizzato in modo asincrono, per gestire comunicazioni seriali.

Visto che dovremo trasportare della memoria i dati, la prima cosa da fare è modificare lo stack delle funzioni di recupero dati stessi e di stampa per come erano state create con **FreeRTOS**:



Edit Task	
Task Name	tRecoverAcc
Priority	osPriorityRealtime6
Stack Size (Words)	512
Entry Function	StartRecoverAcc
Code Generation Option	Default
Parameter	NULL
Allocation	Dynamic
Buffer Name	NULL
Control Block Name	NULL
<div>OK Cancel</div>	

Edit Task

×

Task Name	tRecoverPress
Priority	osPriorityRealtime5
Stack Size (Words)	512
Entry Function	StartRecoverPress
Code Generation Option	Default
Parameter	NULL
Allocation	Dynamic
Buffer Name	NULL
Control Block Name	NULL

OK

Cancel

Edit Task

×

Task Name	tRecoverMag
Priority	osPriorityRealtime4
Stack Size (Words)	512
Entry Function	StartRecoverMag
Code Generation Option	Default
Parameter	NULL
Allocation	Dynamic
Buffer Name	NULL
Control Block Name	NULL

OK

Cancel

Edit Task

×

Task Name	tRecoverGyro
Priority	osPriorityRealtime7
Stack Size (Words)	512
Entry Function	StartRecoverGyro
Code Generation Option	Default
Parameter	NULL
Allocation	Dynamic
Buffer Name	NULL
Control Block Name	NULL

OK

Cancel

Task Name	PrintData
Priority	osPriorityRealtime
Stack Size (Words)	512
Entry Function	StartPrintData
Code Generation Option	Default
Parameter	NULL
Allocation	Dynamic
Buffer Name	NULL
Control Block Name	NULL

OK Cancel

Dopo aver fatto ciò, dovremo creare degli array **uint8_t** per incasellare la stringa da trasmettere a **UART**, in ordine:

- **MSG_Gyro[150]**
- **MSG_LSM6DSL_Acc[150]**
- **MSG_LSM303AGR_Acc[150]**
- **MSG_Mean_Acc[150]**
 - **MSG_Mag[150]**
 - **MSG_Press[150]**

Dopo averlo attuato sostituiremo alla **printf** una **sprintf** (sempre all'interno dell'if del semaforo) e aggiungeremo la **transmit** della **UART**.

I due comandi saranno:

- **int sprintf(char *str, const char *format, ...)**

la **str** è il nostro **MSG** specifico per casistica e invece il **format** conterrà la stringa da stampare a schermo.

- **HAL_StatusTypeDef HAL_UART_Transmit(UART_HandleTypeDef *huart, const uint8_t *pData, uint16_t Size, uint32_t Timeout)**

Il **huart** è la reference alla porta **UART** attiva che, nel nostro caso, è **huart3**.

Il **pData** contiene il messaggio da trasmettere attraverso **UART**.

Il **Size** è la size del messaggio stesso che noi abbiamo estratto tramite il comando **sizeof** e il **timeout** che ho impostato arbitrariamente a 100 dopo un testing di tipo **"try and error"**.

```

if(osSemaphoreAcquire(Sem_GyroHandle, 0) == osOK) {
    sprintf(MSG_Gyro, "Gyroscope %f,%f,%f data x = %.2f, y = %.2f, z = %.2f in rad*sec^-1\r\n", VectGyro[0], VectGyro[1], VectGyro[2]);
    HAL_UART_Transmit(&huart3, MSG_Gyro, sizeof(MSG_Gyro), 100);
    osSemaphoreRelease(Sem_GyroHandle);
}

if(osSemaphoreAcquire(Sem_LSM6DSL_AccHandle, 0) == osOK) {
    sprintf(MSG_LSM6DSL_Acc, "LSM6DSL Accelerometer %f,%f,%f data x = %.2f, y = %.2f, z = %.2f in m*sec^-2\r\n", VectAcc_LSM6DSL[0], VectAcc_LSM6DSL[1], VectAcc_LSM6DSL[2]);
    HAL_UART_Transmit(&huart3, MSG_LSM6DSL_Acc, sizeof(MSG_LSM6DSL_Acc), 100);
    osSemaphoreRelease(Sem_LSM6DSL_AccHandle);
}

if(osSemaphoreAcquire(Sem_LSM303AGR_AccHandle, 0) == osOK) {
    sprintf(MSG_LSM303AGR_Acc, "LSM303AGR Accelerometer %f,%f,%f data x = %.2f, y = %.2f, z = %.2f in m*sec^-2\r\n", VectAcc_LSM303AGR[0], VectAcc_LSM303AGR[1], VectAcc_LSM303AGR[2]);
    HAL_UART_Transmit(&huart3, MSG_LSM303AGR_Acc, sizeof(MSG_LSM303AGR_Acc), 100);
    osSemaphoreRelease(Sem_LSM303AGR_AccHandle);
}

if(osSemaphoreAcquire(Sem_LSM303AGR_AccHandle, 0) == osOK) {
    if(osSemaphoreAcquire(Sem_LSM6DSL_AccHandle, 0) == osOK) {
        float variable_append0 = (VectAcc_LSM303AGR[0] + VectAcc_LSM6DSL[0])/2;
        float variable_append1 = (VectAcc_LSM303AGR[1] + VectAcc_LSM6DSL[1])/2;
        float variable_append2 = (VectAcc_LSM303AGR[2] + VectAcc_LSM6DSL[2])/2;
        sprintf(MSG_Mean_Acc, "Mean Accelerometer %f,%f,%f data x = %.2f, y = %.2f, z = %.2f in m*sec^-2\r\n", variable_append0, variable_append1, variable_append2);
        HAL_UART_Transmit(&huart3, MSG_Mean_Acc, sizeof(MSG_Mean_Acc), 100);
        osSemaphoreRelease(Sem_LSM6DSL_AccHandle);
    }
    osSemaphoreRelease(Sem_LSM303AGR_AccHandle);
}

if(osSemaphoreAcquire(Sem_MagHandle, 0) == osOK) {
    sprintf(MSG_Mag, "Magnetometer %f,%f,%f data x = %.2f, y = %.2f, z = %.2f in MicroT\r\n", VectMag[0], VectMag[1], VectMag[2]);
    HAL_UART_Transmit(&huart3, MSG_Mag, sizeof(MSG_Mag), 100);
    osSemaphoreRelease(Sem_MagHandle);
}

if(osSemaphoreAcquire(Sem_PressHandle, 0) == osOK) {
    sprintf(MSG_Press, "Barometer data = %.1f in hPa\r\n", VectPress);
    HAL_UART_Transmit(&huart3, MSG_Press, sizeof(MSG_Press), 100);
    osSemaphoreRelease(Sem_PressHandle);
}

osDelay(2000);

```

Inoltre, per il funzionamento di tutto questo abbiamo dovuto in primis togliere la parte di codice `_write` perché finalizzata alla comunicazione con la console **ITM** e, tramite l'ide **STM32CubeIDE**, abbiamo dovuto attivare la **Command Shell Console** per ricevere dalla porta **COM7** (trovata nella gestione dispositivi del mio computer) il risultato della stampa.

L'immagine della stampa sarà:

```

Gyroscope tridimensional data x = -22.75, y = -3.50, z = 50.75 in rad*sec^-1
LSM6DSL Accelerometer tridimensional data x = -3.03, y = -0.40, z = 9.67 in m*sec^-2
LSM303AGR Accelerometer tridimensional data x = -2.83, y = 0.00, z = 9.18 in m*sec^-2
Mean Accelerometer tridimensional data x = -2.93, y = -0.20, z = 9.43 in m*sec^-2
Magnetometer tridimensional data x = 33100.00, y = 6400.00, z = -37500.00 in MicroT
Barometer data = 993.0 in hPa

```