**COMUNICAZIONE SERIALE SINCRONA: I2C**

Il bus I2C è un bus seriale sincrono molto semplice; utilizza solo 3 linee, di cui 1 è la massa e 2 sono le linee di comunicazione, che vengono chiamate:

- SCL: synchronous clock, e
- SDA: synchronous data.

Sul bus possono essere collegati al massimo 128 dispositivi. Quando un dispositivo vuole comunicare con un altro dispositivo assume il ruolo di master; l'altro dispositivo con il quale comunica assume il ruolo di slave. Il master può voler trasmettere o ricevere, e quindi possiamo avere le combinazioni:

- Master transmitter, slave receiver, oppure
- Master receiver, slave transmitter.

Quando nessuno trasmette le due linee sono entrambe alte; esse infatti sono collegate a resistenze di pull-up che portano la tensione al valore dell'alimentazione.
Quando il master vuole comunicare abbassa la linea SDA, e dopo un po' di tempo abbassa anche la linea SCK. Questa particolare sequenza di fronti di discesa (prima SDA e poi SCK) indica a tutti i dispositivi sul bus l'inizio della comunicazione (segnale START, spesso indicato con ST nei diagrammi). Il segnale di STOP (spesso indicato con SP nei diagrammi), che notifica la fine della comunicazione, è complementare allo START e parte da una situazione in cui entrambe le linee SCK sono basse e la linea SCK viene alzata prima della linea SDA, arrivando di nuovo alla situazione in cui entrambe le linee sono alte. Una comunicazione che avviene tra START e STOP viene anche detta *transazione* I2C, e coinvolge un ben determinato master e un ben determinato slave che detengono il bus per tutta la durata della transazione stessa: quando termina la transazione altre unità possono cercare di accedere al bus in qualità di master e stabilire una nuova transazione con un altro slave.
In una transazione si trasmettono le informazioni come sequenze di bit. Questi sono messi sulla linea SDA dal trasmittente ma solamente negli istanti di tempo in cui SCK è basso: in tale condizione SDA può essere commutata, ad esempio da 0 a 1, dal trasmittente. Quando SCK è alto SDA non può essere commutata, ma deve rimanere costante perché possa essere letta dal ricevente. Quindi il trasmittente gestisce entrambe le linee SDA e SCK, e per trasmettere un bit deve abbassare SCK, commutare eventualmente SDA in funzione del bit che deve trasmettere (se è diverso da quello precedente trasmesso, altrimenti SDA non va commutata), alzare di nuovo SCK ed aspettare perché il dato venga letto dal ricevente, per poi ritornare ad abbassare SCK per iniziare a trasmettere il bit successivo. La frequenza di commutazione di SCK (del clock) determina quindi la velocità di trasmissione. Notare inoltre che solo nelle fasi di START e STOP il segnale SDA commuta quando SCK è alto: durante la trasmissione dei bit di informazione SDA commuta solo quando SCK è basso. Questo permette di distinguere i segnali START/STOP dalla trasmissione dei bit di informazione.
La trasmissione delle informazioni segue un protocollo fisso. Dopo lo START il master spedisce un *address frame*, ossia una sequenza di 7 bit (indicata con SUB nei diagrammi) che rappresenta l'indirizzo del destinatario (lo slave), e quindi un ottavo bit (indicato con R/W nei diagrammi), che indica se il master è transmitter (W) o receiver (R). In risposta lo slave invia un bit di ackowledgment positivo (indicato con ACK, e pari a 1) o negativo (indicato con NACK, pari a 0); il secondo caso si può verificare se durante la trasmissione c'è stata una collisione sul bus, o il receiver ha avuto un errore di timing o un overflow del buffer di ricezione. Quindi il transmitter (che, ricordiamo, può essere il master o lo slave) invia un certo numero di *data frame*, ossia di sequenze di 8 bit; a ciascun data frame il receiver risponde con un ACK/NACK, in maniera diversa se il master e trasmitter o receiver: se il master è trasmitter, lo slave deve mandare un ACK per ogni

byte che il master gli invia, se il master è receiver esso manda ACK allo slave fino a quando desidera ricevere dati dallo slave: quando non desidera più ricevere dati manda NACK. Infine la comunicazione termina con il master che porta le linee in condizione di STOP. Il numero e il contenuto dei data frame scambiati tra trasmitter e receiver dipendono dal protocollo applicativo che viaggia su di esso e che di solito è implementato sullo slave: ad esempio, se il master è un microcontrollore, lo slave è una memoria FLASH, e l'operazione è R, il master deve seguire il protocollo I2C della FLASH (di solito dettagliato nel datasheet di questa) per trasmettere l'indirizzo della memoria che vuole leggere: se l'indirizzo di lettura è, ad esempio, di 32 bit, occorrerà che il master invii alla FLASH quattro data frame di 8 bit, nell'ordine corretto che si aspetta lo slave. Spesso occorre concatenare due operazioni di lettura/scrittura di seguito; consideriamo ancora l'esempio del microcontrollore (master) che vuole leggere da un certo indirizzo di memoria di una FLASH (slave). Il protocollo per leggere da una memoria FLASH collegata ad un bus I2C di solito è strutturato in questo modo: il microcontrollore deve effettuare una prima transazione di tipo master trasmitter (W) per spedire alla FLASH l'indirizzo della cella di memoria da leggere. Terminata questa transazione, il microcontrollore deve subito dopo effettuare una seconda transazione I2C di tipo master receiver (R) per ricevere dalla FLASH il valore corrispondente letto dalla memoria. Siccome questa sequenza di due transazioni successive (W per spedire un comando o un indirizzo, seguito da R per leggere il corrispondente dato prodotto in output) è molto comune, è prevista nel protocollo I2C la possibilità di concatenare due transazioni per formarne una sola "eliminando" lo STOP intermedio. Se quindi una transazione è terminata da un segnale di START anziché da uno STOP, questo secondo segnale di START viene interpretato come la ripartenza di una successiva transazione concatenata, ossia come un RESTART (indicato con SR nei diagrammi). Concatenare due transazioni ha diversi vantaggi, non ultimo il fatto che eliminando lo STOP intermedio il bus non viene rilasciato tra una transazione e l'altra, cosa che potrebbe introdurre tempi di attesa molto lunghi se per sorte tra le due transazioni un altro master prendesse il bus, o addirittura inconsistenze se tale nuovo master volesse iniziare a comunicare con lo stesso slave.

Il protocollo I2C è così semplice che può essere facilmente implementato in bit-banging sul GPIO, ma noi abbiamo la comodità di avere nel microcontrollore delle periferiche dedicate che lo implementano, scaricando la CPU dal compito, e quindi sfruttiamo tale comodità. Per sperimentare colleghiamo lo shield dei sensori, che ha codice identificativo X-NUCLEO-IKS01A2, alla scheda Nucleo. Tale shield comunica con la scheda Nucleo utilizzando il bus I2C. Sulla scheda Nucleo gli slot dedicati all'I2C sono nel connettore Zio di destra, nella fila più esterna (più a destra): sono i due slot più in alto. Se guardiamo lo user manual della scheda Nucleo, sezione 6.13 (figura 11) vediamo che i contatti Arduino sono segnati in rosa. Quelli che hanno nome Dxx sono digitali, quelli che hanno nome Axx sono analogici. I contatti D14 e D15 corrispondono sulle schede Arduino a SDA e SCL rispettivamente per l'I2C, ma non in tutte le schede Arduino (grazie al cielo, in quelle più recenti, e quindi anche in quelle supportate dal nostro shield). Nella figura si nota che D14 corrisponde al GPIO PB9, e D15 al GPIO PB8, che vanno pertanto attivati. Per ulteriore conferma possiamo vedere nell'user manual dello shield IKS01A2 (UM2121) che i connettori del I2C stanno (tabella 6) su CN5, pin 7, 9 e 10 – notare che si intende CN5 dello shield IKS01A2, non della Nucleo. Il CN5 si può facilmente trovare sulla scheda; i pin non sono numerati, ma se li contiamo dal basso verso l'alto il settimo, nono e decimo corrispondono, sulla Nucleo, a GND, PB9 e PB8. Se nel configuration tool attiviamo i pin PB8 e PB9 si può notare che, tra le varie periferiche che si possono associare ai pin ci sono ben due periferiche I2C, l'I2C1 e l'I2C4. In entrambi i casi il pin PB8 può essere associato solo all'SCL di tali periferiche (il che è corretto, dal momento che il pin PB8 esce su D15). Similmente per il pin PB9 (in tal caso posso selezionare solo l'SDA di I2C1 o I2C4). Vedremo che la configurazione automatica che faremo successivamente li imposterà entrambi ad I2C1. Notiamo poi che i sensori montati sullo shield IKS01A2 sarebbero anche in grado di comunicare con il protocollo SPI (basta leggere i datasheet di tali sensori), ma lo shield utilizza solo il protocollo I2C, probabilmente per ridurre il numero delle linee necessarie per connettersi con tutti i sensori (con SPI ogni sensore dovrebbe avere delle linee dedicate).

Un parametro importante è la velocità di comunicazione del protocollo I2C. La scheda IKS01A2 ospita quattro diversi sensori, ognuno dei quali è collegato allo stesso bus I2C; concentriamoci solo sulla comunicazione con il sensore HTS221 (temperatura e umidità). Se andiamo nel datasheet del sensore notiamo che il sensore può operare solamente a tre frequenze di output (output data rate, ODR): 1 Hz, 7 Hz e 12.5 Hz. D'altra parte le velocità massime per il protocollo I2C sono:

- Standard mode: 100 kbit/sec;
- Fast mode: 400 kbit/sec;
- Fast mode plus: 1 Mbit/sec.

(C'è anche un high speed mode a 3.4 Mbit/sec, ma è un po' particolare e non è supportato dalle periferiche del microcontrollore). Dal datasheet dello shield e del sensore si può vedere che la velocità di comunicazione I2C che essi supportano è almeno fast mode, ma anche standard mode (in velocità massima) dovrebbe essere più che sufficiente per sostenere gli ODR molto bassi del sensore. Quando vi sono più unità I2C collegate allo stesso bus, la velocità di comunicazione deve essere la velocità più bassa tra quella supportata dalle unità collegate al bus. Dal momento che i quattro sensori sullo shield sono tutti collegati allo stesso bus I2C, bisogna verificare qual è il più lento dei quattro e impostare la comunicazione I2C a quella velocità. In realtà si può vedere dai datasheet dei sensori che tutti supportano almeno fast mode, e quindi fast mode dovrebbe essere ok. Inoltre, secondo gli esempi forniti con il software pack, la scheda IKS01A2 supporta esplicitamente fast mode, e quindi possiamo assumere questa come la velocità migliore per la comunicazione.

Riguardo all'interazione con i sensori, il problema è quali sono i comandi che vanno inviati ad essi per avviarli, tararli, leggere i dati – dati che poi vanno convertiti da valori digitali prodotti dall'ADC nella corrispondenza grandezza fisica. il tutto è estremamente complicato. Per semplificare l'operatività con i sensori esistono delle librerie ST che permettono di interagire in maniera facile con i sensori, ed anche con gli shield X-NUCLEO, prodotti dalla ST, tra il quale il nostro shield di laboratorio X-NUCLEO-IKS01A2. Creiamo pertanto un nuovo progetto, ed attiviamo questa libreria: aprite il configuratore CubeMX e selezionate il menu Software Packs > Select Components. Scegliete la riga STMicroelectronics.X-CUBE-MEMS1, aprite l'elenco a spunta sulla sinistra, e nella lunga lista di elementi trovate la riga "Board extension IKS01A2", e selezionate la checkbox corrispondente, quindi premete il pulsante Ok. Ora, in fondo all'elenco Categories sulla sinistra, oltre a System Core, Analog … Middleware, in fondo troverete anche Software Packs. Aprite Software Packs e vedrete un solo elemento, di nome STMicroelectronics.X-CUBE-MEMS1. Cliccandolo sulla destra troverete una check box "Board extension IKS01A2": selezionate la check box, e nella finestra sottostante Configuration, nel tab Parameter Settings, lasciate "IPs or Components" su "I2C:I2C", e per "Found Solutions" selezionate "I2C1" (unica selezione possibile). Sulla destra, la colonna "BSP API" dovrebbe riportare "BSP_BUS_DRIVER". Salvate la configurazione e generate il codice. Come si può vedere, non si può scegliere né la periferica I2C (è per forza I2C1) né la velocità di comunicazione I2C, che comunque viene impostata in maniera da sostenere l'ODR dei sensori. Non si può scegliere nemmeno l'ODR dei sensori – ma questo, come vedremo, si può impostare attraverso API.

Le nuove API per utilizzare i sensori sono piuttosto semplici. Sul sito del corso trovate un link al github dell'intero software pack X-CUBE-MEMS1, che contiene documentazione ed esempi di utilizzo. Le API IKS01A2_ENV_SENSOR_XXXXXX permettono di interagire con i sensori ambientali (temperatura, umidità, pressione), e le API IKS01A2_MOTION_SENSOR_XXXXXX permettono di interagire con i sensori di movimento (giroscopio, accelerometro, magnetometro). Le prime sono dichiarate nell'header file iks01a2_env_sensors.h, le seconde in iks01a2_motion_sensors.h.

Ogni sensore può avere più di una funzionalità: ad esempio il sensore HTS221 ha due funzionalità: rilevamento temperatura, e rilevamento umidità relativa. Una coppia sensore + funzionalità va inizializzata con l'API IKS01A2_XXX_SENSOR_Init: ad esempio, è possibile inizializzare la funzione di rilevamento temperatura del sensore HTS221, e non inizializzare quella di rilevamento umidità. La coppia sensore + funzionalità va poi abilitata con l'API IKS01A2_XXX_SENSOR_Enable. A questo punto si può usare l'API IKS01A2_XXX_SENSOR_GetValue per leggere un valore da un certo sensore + funzionalità. Tale valore è direttamente convertito nella grandezza fisica corrispondente, ad esempio gradi Celsius per la temperatura. Un semplicissimo progetto che sfrutta tali API per effettuare un'acquisizione periodica del dato di temperatura con frequenza di 1 Hz è il seguente:

main.c

```
...
/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include <stdio.h>
#include "iks01a2_env_sensors.h"
/* USER CODE END Includes */
...
int main(void)
{
...

  /* USER CODE BEGIN 2 */
  IKS01A2_ENV_SENSOR_Init(IKS01A2_HTS221_0, ENV_TEMPERATURE);
  IKS01A2_ENV_SENSOR_Enable(IKS01A2_HTS221_0, ENV_TEMPERATURE);
  /* USER CODE END 2 */

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

    /* USER CODE BEGIN 3 */
      float temperature;
      int res = IKS01A2_ENV_SENSOR_GetValue(IKS01A2_HTS221_0, ENV_TEMPERATURE,
&temperature);
      if (res == BSP_ERROR_NONE) {
            printf("T = %f C\r\n", temperature);
      } else {
            printf("ERROR\r\n");
      }
      HAL_Delay(1000);
  }
  /* USER CODE END 3 */
}
...
/* USER CODE BEGIN 4 */
int _write(int file, char *ptr, int len) {
      for (int i = 0; i < len; ++i) {
            ITM_SendChar(*ptr++);
      }
      return len;
}
/* USER CODE END 4 */
...
```

Notare che dobbiamo mettere un ritardo di 1 secondo tra una lettura del sensore e la successiva in maniera da far corrispondere la frequenza di lettura con l'ODR: altrimenti perderemmo valori (se la

prima fosse minore della seconda) o leggeremmo lo stesso valore più volte (se la prima fosse maggiore della seconda).

Non ci accontenteremo di utilizzare queste librerie perché, come vedremo, la loro semplicità implica anche forti limitazioni. Studiamo allora come queste sono implementate: questo ci permetterà di capire come la comunicazione I2C è usata per interagire con i sensori, e da qui potremo scrivere delle nostre funzioni di comunicazione più flessibili e potenti. La struttura delle librerie, in verità, è decisamente intricata. Si può semplificare il tutto considerando i seguenti files:

- Drivers/BSP/IKS01A2/iks01a2_env_sensors.c e Drivers/BSP/IKS01A2/iks01a2_motion_sensors.c: nel codice delle funzioni HTS221_0_Probe(uint32_t), LPS22HB_0_Probe(uint32_t), LSM6DSL_0_Probe(uint32_t), LSM303AGR_ACC_0_Probe(uint32_t) e LSM303AGR_MAG_0_Probe(uint32_t), funzioni che vengono invocate in fase di inizializzazione dei drivers, possiamo vedere il codice seguente, che imposta per ciascun driver le funzioni utilizzate per leggere/scrivere via I2C nei registri dei sensori:

  ```
  io_ctx.Init        = IKS01A2_I2C_Init;
  io_ctx.DeInit      = IKS01A2_I2C_DeInit;
  io_ctx.ReadReg     = IKS01A2_I2C_ReadReg;
  io_ctx.WriteReg    = IKS01A2_I2C_WriteReg;
  io_ctx.GetTick     = IKS01A2_GetTick;
  ```

- X-CUBE-MEMS1/Target/iks01a2_conf.h definisce IKS01A2_I2C_Init/DeInit/… come alias di BSP_I2C_Init/DeInit/…
- Core/Src/stm32f7xx_nucleo_bus.c definisce le funzioni BSP_I2C_Init/DeInit/…
- Le effettive operazioni si trovano in Drivers/BSP/Components; ad esempio, Drivers/BSP/Components/hts221/hts221.c definisce la funzione HTS221_TEMP_GetTemperature che legge la temperatura dal sensore. Questa a sua volta chiama funzioni di più basso livello che effettivamente leggono i registri del sensore, definite in Drivers/BSP/Components/hts221/hts221_reg.c.

Consideriamo ad esempio HTS221_TEMP_GetTemperature; il suo codice è:

hts221.c:

```
int32_t HTS221_TEMP_GetTemperature(HTS221_Object_t *pObj, float *Value)
{
  hts221_axis1bit16_t data_raw_temperature;
  lin_t lin_temp;

  if (hts221_temp_adc_point_0_get(&(pObj->Ctx), &lin_temp.x0) != HTS221_OK)
  {
    return HTS221_ERROR;
  }

  if (hts221_temp_deg_point_0_get(&(pObj->Ctx), &lin_temp.y0) != HTS221_OK)
  {
    return HTS221_ERROR;
  }

  if (hts221_temp_adc_point_1_get(&(pObj->Ctx), &lin_temp.x1) != HTS221_OK)
  {
    return HTS221_ERROR;
  }

  if (hts221_temp_deg_point_1_get(&(pObj->Ctx), &lin_temp.y1) != HTS221_OK)
```

```
  {
    return HTS221_ERROR;
  }

  (void)memset(&data_raw_temperature.i16bit, 0x00, sizeof(int16_t));
  if (hts221_temperature_raw_get(&(pObj->Ctx), &data_raw_temperature.i16bit) !=
HTS221_OK)
  {
    return HTS221_ERROR;
  }

  *Value = Linear_Interpolation(&lin_temp, (float)data_raw_temperature.i16bit);

  return HTS221_OK;
}
```

Tutte le funzioni hts221_temp_XXX, e hts221_temperature_raw_get, sono definite in hts221_reg.h. Esse invocano la funzione hts221_read_reg, che utilizza la comunicazione I2C per leggere i registri nella memoria interna del sensore. Vedremo dopo come questo viene realizzato.

Studiando il codice sorgente delle librerie ci si può fare un'idea di quale sia la giusta sequenza di funzionamento dell'interazione tra microcontrollore e sensore, cosa che non sempre emerge in maniera chiarissima dal datasheet del sensore stesso. Ad esempio, leggendo il codice delle funzioni HTS221_Initialize, HTS221_Enable (in Drivers/BSP/Components/hts221/hts221.c), LPS22HB_Initialize, LPS22HB_Enable (in Drivers/BSP/Components/lps22hb/lps22hb.c), ecc. possiamo notare che i sensori vengono impostati a questi ODR:

- HTS221: 1 Hz;
- LPS22HB: 25 Hz;
- LSM303AGR: 100 Hz (sia magnetometro che accelerometro);
- LSM6DSL: 104 Hz.

Questi non sono gli ODR massimi, dal momento che HTS221 può operare fino a 12.5 Hz (sia temperatura che umidità relativa), LPS22HB fino a 75 Hz, LSM303AGR fino a 100 Hz per il magnetometro e fino a 1.34 KHz per l'accelerometro in modalità normale o alta risoluzione, e 5.38 KHz per l'accelerometro in modalità low-power, e infine LSM6DSL fino a 6.66 KHz (sia giroscopio che accelerometro). Abbiamo un API per impostare l'ODR, IKS01A2_XXX_SENSOR_SetOutputDataRate, ma se diamo un'occhiata al datasheet dei sensori vediamo che questi hanno altre possibili configurazioni, molte di più di quelle accessibili dalle API. Ad esempio, per HTS221 è possibile, tramite il registro AV_CONF, impostare il numero di campioni di cui il sensore fa la media, e tramite il registro CTRL_REG2 si può avviare un ciclo di riscaldamento per far recuperare il sensore in caso di condensazione. Per utilizzare tali configurazioni occorre manipolare direttamente i registri dei sensori.

La manipolazione diretta dei registri dei sensori è possibile, dal momento che i datasheet riportano quali sono i loro indirizzi e che scopo hanno i loro bit, ma è molto complicata, perché ad esempio i datasheet non riportano le sequenze corrette di operazioni. Fortunatamente le funzioni in Drivers/BSP/Components/<Sensore>/<Sensore>_reg.h formano un'altra API, di basso livello, che espongono tutte le possibili configurazioni dei sensori accessibili attraverso i loro registri, in maniera semplificata dal momento che nascondono all'utente quali registri specificamente manipolano. Ad esempio in Drivers/BSP/Components/hts221/hts221_reg.h abbiamo un insieme di funzioni C che, attraverso la lettura/scrittura diretta dei registri dell'HTS221, permettono di configurare il sensore di temperatura e umidità in ogni possibile modo, e di leggere da esso tutte le possibili informazioni che produce. Le API di basso livello astraggono rispetto a come può avvenire la comunicazione con il sensore (attraverso I2C oppure SPI), in maniera da essere indipendenti

dalla specifica board. Questo però implica che per utilizzarle bisogna configurare un'opportuna struttura dati (detta il "contesto") che si occuperà di memorizzare tutto ciò che serve per effettuare concretamente la comunicazione. Il contesto contiene:

- un puntatore ad una funzione che effettua la lettura di un certo registro del sensore, passato come parametro (occorre implementare tale funzione attraverso una lettura sincrona del registro attraverso I2C o SPI);
- un puntatore ad una funzione che effettua la scrittura di un certo registro del sensore, passato come parametro (anche in questo caso occorre implementare la funzione attraverso una scrittura sincrona attraverso I2C o SPI);
- un puntatore allo handle I2C o SPI che viene usato dalle due funzioni di cui sopra per comunicare con il sensore. Tale puntatore viene passato come (primo) parametro alle funzioni dei due punti precedenti quando esse vengono invocate.

Il tipo "contesto" definito in hts221_reg.h è il tipo stmdev_ctx_t, definito come:

hts221_reg.h:

```
typedef struct
{
  /** Component mandatory fields **/
  stmdev_write_ptr  write_reg;
  stmdev_read_ptr   read_reg;
  /** Customizable optional pointer **/
  void *handle;
} stmdev_ctx_t;
```

dove i tipi "puntatore a funzione" stmdev_write_ptr e stmdev_read_ptr sono definiti:

hts221_reg.h:

```
typedef int32_t (*stmdev_write_ptr)(void *, uint8_t, uint8_t *, uint16_t);
typedef int32_t (*stmdev_read_ptr)(void *, uint8_t, uint8_t *, uint16_t);
```

dove il primo parametro è lo handle contenuto nel contesto stesso, il secondo parametro è l'indirizzo del registro, il terzo parametro è un puntatore a un buffer per i dati di lettura/scrittura, e il quarto contiene la dimensione del buffer in byte, e quindi determina anche il numero di byte da trasferire. Nel caso in cui tale, ultimo numero sia maggiore di 1, e quindi debba – ad esempio – leggere due bytes, viene effettuata una unica comunicazione I2C che legge/scrive tutti i byte in sequenza. In tal caso occorre segnalare al sensore se i byte vanno tutti letti/scritti dallo/sullo stesso indirizzo del registro, oppure se occorre incrementare il valore dell'indirizzo del registro mano a mano che si leggono/scrivono i byte successivi. Per comunicare questa informazione al sensore HTS221 si utilizza il bit più significativo dell'indirizzo del registro (gli indirizzi dei registri dell'HTS221 sono a 7 bit): se questo bit è impostato a zero, allora non vi è autoincremento dell'indirizzo, e tutti i dati vanno letti/scritti allo stesso indirizzo, mentre se il bit è impostato a 1 vi è autoincremento. Ad esempio, il valore di temperatura è contenuto in due registri di 8 bit i cui indirizzi sono consecutivi, e quindi si può leggere il valore di temperatura con una sola comunicazione I2C utilizzando l'autoincremento: per farlo bisogna impostare a 1 il bit più significativo del secondo parametro delle funzioni di lettura/scrittura registri (almeno, se si usano le funzioni dello HAL per la comunicazione, questa cosa va fatta).
La funzione hts221_read_reg, precedentemente citata come la funzione che viene utilizzata per effettuare tutte le letture dei registri, viene implementata semplicemente "redirezionando" la chiamata sulla corrispondente funzione del contesto, e similmente la funzione di scrittura hts221_write_reg:

hts221_reg.h:

```
...
int32_t hts221_read_reg(stmdev_ctx_t *ctx, uint8_t reg, uint8_t *data,
                        uint16_t len)
{
  int32_t ret;

  ret = ctx->read_reg(ctx->handle, reg, data, len);

  return ret;
}
...
int32_t hts221_write_reg(stmdev_ctx_t *ctx, uint8_t reg,
                         uint8_t *data,
                         uint16_t len)
{
  int32_t ret;

  ret = ctx->write_reg(ctx->handle, reg, data, len);

  return ret;
}
```

Anche gli altri <Sensore>_reg.h definiscono simili contesti, tutti similmente strutturati, e implementano similmente le funzioni di lettura e scrittura.

Come possiamo implementare le funzioni read_reg e write_reg del contesto? Molto semplicemente, possiamo utilizzare due funzioni dell'HAL che sono perfettamente adatte allo scopo, ossia HAL_I2C_Mem_Read e HAL_I2C_Mem_Write. Esse sono concepite per effettuare una comunicazione I2C nel caso in cui lo scopo sia leggere/scrivere la memoria dello slave. Tali funzioni sono sincrone, e quindi il processore rimane bloccato fino alla fine della comunicazione: notare (attenzione!) che HAL_I2C_Mem_Write aspetta solo che la comunicazione con lo slave termini, non che lo slave termini la scrittura in memoria. Nel caso in cui dovessimo scrivere codice che deve aspettare anche il completamento dell'operazione sulla memoria, è possibile usare un'altra API, HAL_I2C_IsDeviceReady. Possiamo a questo punto reimplementare il codice che stampa la temperatura letta dal sensore HTS221 con frequenza 1 Hz utilizzando solo le funzioni di hts221_reg.h, in questo modo:

main.c:

```
/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include <stdio.h>
#include <stm32f7xx_nucleo_bus.h>
#include <hts221_reg.h>
/* USER CODE END Includes */
...
/* USER CODE BEGIN PV */
static stmdev_ctx_t ctx0;
static float t_x0, t_y0, t_x1, t_y1;
/* USER CODE END PV */
...
/* USER CODE BEGIN PFP */
static void hts221_init(void);
static int32_t my_read_reg(void *handle, uint8_t reg, uint8_t *data, uint16_t
len);
static int32_t my_write_reg(void *handle, uint8_t reg, uint8_t *data, uint16_t
len);
```

```
static float linear_interpolation(float x0, float y0, float x1, float y1, float
coeff);
/* USER CODE END PFP */
...
int main(void)
{
  ...
  /* USER CODE BEGIN 2 */
  BSP_I2C1_Init();
  hts221_init();
  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    /* USER CODE END WHILE */
    /* USER CODE BEGIN 3 */
    int16_t temp_raw = 0;
    int32_t res = hts221_temperature_raw_get(&ctx0, &temp_raw);
    if (res == HAL_OK) {
      float temperature = linear_interpolation(t_x0, t_y0, t_x1, t_y1, (float)
temp_raw);
      printf("T = %f C\r\n", temperature);
    } else {
      printf("ERROR\r\n");
    }

    HAL_Delay(1000);

  }
  /* USER CODE END 3 */
}
...
/* USER CODE BEGIN 4 */
...
static void hts221_init(void) {
  /* set the context for low-level sensor library */
  ctx0.handle = &hi2c1;
  ctx0.read_reg = my_read_reg;
  ctx0.write_reg = my_write_reg;

  /* block update of registers after reading the lower/upper part of
   * temperature data, until the complementary part of the data is
   * read
   */
  if (hts221_block_data_update_set(&ctx0, PROPERTY_ENABLE) != HAL_OK) {
    Error_Handler();
  }

  /* set the output data rate (ODR) to 1 Hz) */
  if (hts221_data_rate_set(&ctx0, HTS221_ODR_1Hz) != HAL_OK) {
    Error_Handler();
  }

  /* turn on the sensor */
  if (hts221_power_on_set(&ctx0, PROPERTY_ENABLE) != HAL_OK) {
    Error_Handler();
  }

  /* get the temperature calibration data */
  if (hts221_temp_adc_point_0_get(&ctx0, &t_x0) != HAL_OK) {
    Error_Handler();
  }
```

```
  if (hts221_temp_deg_point_0_get(&ctx0, &t_y0) != HAL_OK) {
    Error_Handler();
  }
  if (hts221_temp_adc_point_1_get(&ctx0, &t_x1) != HAL_OK) {
    Error_Handler();
  }
  if (hts221_temp_deg_point_1_get(&ctx0, &t_y1) != HAL_OK) {
    Error_Handler();
  }
}

static int32_t my_read_reg(void *handle, uint8_t reg, uint8_t *data, uint16_t
len) {
  /* need to set the most significant bit of register to allow address
   * auto increment with multiple commands mode
   */
  return HAL_I2C_Mem_Read((I2C_HandleTypeDef *) handle, HTS221_I2C_ADDRESS, (reg
| 0x80U), I2C_MEMADD_SIZE_8BIT, data, len, 100);
}

static int32_t my_write_reg(void *handle, uint8_t reg, uint8_t *data, uint16_t
len) {
  /* need to set the most significant bit of register to allow address
   * auto increment with multiple commands mode
   */
  return HAL_I2C_Mem_Write((I2C_HandleTypeDef *) handle, HTS221_I2C_ADDRESS,
(reg | 0x80U), I2C_MEMADD_SIZE_8BIT, data, len, 100);
}

static float linear_interpolation(float x0, float y0, float x1, float y1, float
coeff) {
  return (((y1 - y0) * coeff) + ((x1 * y0) - (x0 * y1))) / (x1 - x0);
}
```

Alcuni commenti:

- la funzione BSP_I2C1_Init inizializza l'I2C1, quello usato per comunicare con il sensore di temperatura. È contenuta in stm32f7xx_nucleo_bus.h, header file che viene creato quando includiamo l' X-CUBE-MEMS1 software pack; tale header non viene incluso automaticamente nel main, quindi dobbiamo includerlo manualmente;
- hts221_temperature_raw_get legge il dato dei registri TEMP_OUT_L/TEMP_OUT_H del sensore, che contengono l'output dell'ADC: dobbiamo pertanto effettuare noi la conversione di tale dato nella grandezza fisica corrispondente (gradi Celsius) utilizzando i dati di calibrazione, secondo quanto riportato nel datasheet; questo veniva fatto in maniera trasparente dalla funzione HTS221_TEMP_GetTemperature (in hts221.c), che ritornava direttamente la grandezza fisica, e pertanto abbiamo copiato da essa il codice di conversione (una semplice interpolazione lineare basata sui punti di calibrazione);
- il codice di hts221_init è stato copiato in gran parte da HTS221_Initialize e in minor parte da HTS221_TEMP_GetTemperature (entrambe in hts221.c); riguardo a quest'ultima funzione, notare che HTS221_TEMP_GetTemperature, ogni volta che viene invocata, rilegge i dati di calibrazione per l'interpolazione lineare, cosa del tutto inutile dal momento che questi sono costanti: pertanto noi introduciamo un piccolo miglioramento e li leggiamo una volta soltanto, in fase di inizializzazione;
- le funzioni my_read_reg e my_write_reg, quando invocano HAL_I2C_Mem_Read/Write devono impostare il bit più significativo dell'indirizzo del registro che vanno a scrivere ad 1, in modo da abilitare l'autoincremento dell'indirizzo: questo perché diverse funzioni di hts221_reg.c (ad esempio, hts221_temp_adc_point_0_get e tutte le altre funzioni che

leggono i dati di calibrazioni) leggono due byte in sequenza nella memoria del sensore, e tali funzioni non impostano loro il bit ad 1; questa cosa si può dedurre dal fatto che le funzioni di contesto read_reg e write_reg utilizzate in hts221.c, ReadRegWrap, e WriteRegWrap, impostato sempre il bit più significativo ad 1 se la comunicazione è I2C.

Le API contenute negli header files Drivers/BSP/Components/<Sensore>/<Sensore>_reg.h, comunque di livello abbastanza basso, richiedono che le funzioni read_reg e write_reg che noi dobbiamo iniettare nel contesto siano delle funzioni di comunicazione sincrona. In altre parole, tali API non sono concepite per funzionare in maniera asincrona. Dal momento che le chiamate sincrone bloccano il microcontrollore fino alla fine della comunicazione I2C, che di regola la comunicazione tra chip e chip è lenta, e che nell'HAL esistono delle funzioni asincrone HAL_I2C_Mem_Read_IT e HAL_I2C_Mem_Read_DMA (con le corrispondenti write), potremmo trovarci nella situazione di dover realizzare almeno una parte della comunicazione in maniera asincrona in maniera da liberare il processore per altri compiti. In tal caso nemmeno le API di basso livello sono utili: dobbiamo andare allora a livello ancora più basso e implementare noi delle API asincrone. A tale scopo è utile analizzare il codice sorgente di hts221_reg.c, cosa che ci permette di capire quali sono le vere e proprie letture/scritture dei registri del sensore che permettono di effettuare le diverse operazioni. Supponiamo ad esempio che sia la lettura della temperatura, operazione periodica che potrebbe in futuro essere effettuata a frequenza molto più elevata di 1 Hz, la candidata ad essere implementata in maniera asincrona. Il codice di lettura della temperatura in hts221_reg.c è il seguente:

hts221_reg.c:

```
int32_t hts221_temperature_raw_get(stmdev_ctx_t *ctx, int16_t *val)
{
  uint8_t buff[2];
  int32_t ret;

  ret = hts221_read_reg(ctx, HTS221_TEMP_OUT_L, buff, 2);
  *val = (int16_t)buff[1];
  *val = (*val * 256) + (int16_t)buff[0];

  return ret;
}
```

Come si può vedere, questa funzione leggere due registri ad 8 bit consecutivi nella memoria del sensore, a partire dall'indirizzo HTS221_TEMP_OUT_L (l'indirizzo del registro che contiene il byte meno significativo del valore della temperatura). Questo perché l'indirizzo successivo a HTS221_TEMP_OUT_L è quello del registro che contiene il byte più significativo del valore della temperatura. Letti i due byte, questi sono concatenati nel giusto ordine per formare un valore con segno a 16 bit della temperatura letta dall'ADC del sensore.

Creiamo quindi un progetto che sfrutta la comunicazione I2C asincrona per leggere la temperatura, ed effettua tutte le altre operazioni non periodiche (ad esempio, acquisizione dei dati di calibrazione) in maniera sincrona, come nel progetto precedente. A tale scopo evitiamo di importare il software pack X-CUBE-MEMS1 dall'IDE, che ha un controllo sulla configurazione dell'I2C un po' troppo stretto. Nel progetto che creeremo configureremo l'I2C manualmente: selezioniamo il configuratore ed attiviamo i pin PB8 e PB9, assegnando loro SDA e SDC della periferica I2C1. Notiamo però che i due pin non sono colorati di verde, ma di giallo: quindi la configurazione non è ancora corretta. Questo perché la periferica I2C1 non è attiva. Andiamo allora su Connectivity > I2C1 e attiviamo la periferica selezionando I2C nel drop-down Mode (le modalità SMBus non ci interessano): vedremo che i pin diventano verdi. Riguardo alla velocità di comunicazione, sarà sufficiente impostarla ad un valore sufficientemente maggiore rispetto a 12.5 Hz (velocità massima del sensore di temperatura) per essere in grado di servire tutti i casi d'uso.Abbiamo detto che sia la

scheda IKS01A2 che i sensori montati su di essa supportano fast mode: quindi impostiamo l'I2C1 come segue:

- Parameters settings, Timing configuration, I2C speed mode: fast mode
- Parameters settings, Timing configuration, I2C speed frequency: 400 kbit/sec (già selezionato di default).

Nell'esempio che seguirà useremo la comunicazione asincrona basata sul DMA per acquisire i dati di temperatura: quindi dobbiamo abilitare il DMA per la periferica I2C1 attraverso il configuratore CubeMX. Selezioniamo il tab DMA Settings, clicchiamo il pulsante Add, e sulla drop down box che esce fuori selezioniamo "I2C1_RX": questo abilita il DMA in ricezione (dalla periferica I2C1 verso la memoria). Viene aggiunta una riga alla tabella dei canali DMA; cliccando la riga si possono modificare i parametri per la comunicazione DMA, ma quelli di default vanno bene (mode normal, increment address memory selezionato, data width byte per tutto). Infine selezioniamo il tab NVIC Settings e abilitiamo I2C1 event interrupt, in modo che sia generato un interrupt alla fine del trasferimento DMA così che lo HAL possa invocare la giusta callback.

Ora occupiamoci del codice. Installiamo manualmente i driver di basso livello (ci servono ancora per le comunicazioni sincrone) copiandoli dal progetto precedente: creiamo dentro al nuovo progetto, nel folder Drivers, il sottofolder Drivers>BSP>Components>HTS221 ed aggiungiamoci i file hts221_reg.c e  hts221_reg.h presi dall'analogo folder del progetto precedente. Poi clicchiamo sul nuovo folder HTS221 con il tasto destro del mouse e selezioniamo dal menu pop-up "Add/remove include path". Infine implementiamo la solita applicazione di lettura del sensore di temperatura con frequenza di 1 Hz aggiungendo il seguente codice:

main.c

```
/* USER CODE BEGIN Includes */
#include <stdio.h>
#include "hts221_reg.h"

/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */
typedef enum { false = 0, true } bool;
/* USER CODE END PTD */
...
/* USER CODE BEGIN PV */
static stmdev_ctx_t ctx0;
static float t_x0, t_y0, t_x1, t_y1;
static volatile bool raw_val_available;
static bool next_cycle;
/* USER CODE END PV */
...
/* USER CODE BEGIN PFP */
static void hts221_init(void);
static int32_t my_read_reg(void *handle, uint8_t reg, uint8_t *data, uint16_t
len);
static int32_t my_write_reg(void *handle, uint8_t reg, uint8_t *data, uint16_t
len);
static float linear_interpolation(float x0, float y0, float x1, float y1, float
coeff);
static int32_t hts221_temperature_raw_get_async(void *handle, uint8_t *destBuf);
/* USER CODE END PFP */
...
int main(void)
{
```

```
...
  /* USER CODE BEGIN 2 */
  hts221_init();
  raw_val_available = false;
  next_cycle = true;
  uint8_t buf[2];
  /* USER CODE END 2 */
  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
    if (raw_val_available) {
      raw_val_available = false;
      uint16_t raw_val = (int16_t) buf[1];
      raw_val = (raw_val * 256) + (int16_t) buf[0];
      float temperature = linear_interpolation(t_x0, t_y0, t_x1, t_y1, (float)
raw_val);
      printf("T = %f C\r\n", temperature);
      next_cycle = true;
    }

    if (next_cycle) {
      HAL_Delay(1000);
      int32_t res = hts221_temperature_raw_get_async(&hi2c1, buf);
      if (res != HAL_OK) {
          printf("ERROR\r\n");
      }
    }
  }
  /* USER CODE END 3 */
}
...
/* USER CODE BEGIN 4 */
int _write(int file, char *ptr, int len) {
  ...
}

static void hts221_init(void) {
  ...
}

static int32_t my_read_reg(void *handle, uint8_t reg, uint8_t *data, uint16_t
len) {
  ...
}

static int32_t my_write_reg(void *handle, uint8_t reg, uint8_t *data, uint16_t
len) {
  ...
}

static float linear_interpolation(float x0, float y0, float x1, float y1, float
coeff) {
  ...
}

static int32_t hts221_temperature_raw_get_async(void *handle, uint8_t *destBuf)
{
  return HAL_I2C_Mem_Read_DMA((I2C_HandleTypeDef *) handle, HTS221_I2C_ADDRESS,
(HTS221_TEMP_OUT_L | 0x80U), I2C_MEMADD_SIZE_8BIT, destBuf, 2);
}
```

```
void HAL_I2C_MemRxCpltCallback(I2C_HandleTypeDef *hi2c) {
  if (hi2c == &hi2c1) {
    raw_val_available = true;
  }
}
/* USER CODE END 4 */
...
```

La funzione hts221_temperature_raw_get_async effettua un trasferimento asincrono del valore di temperatura dal sensore al buffer buf. È simile nella sua signature alle funzioni in hts221_reg.h, ed è banalmente implementata attraverso l'invocazione diretta della funzione HAL_I2C_Mem_Read_DMA. Quando il trasferimento DMA termina lo HAL invoca la callback HAL_I2C_MemRxCpltCallback, che attiva il task di stampa (l'architettura è la solita round robin con interrupt). Le funzioni _write, hts221_init, my_read_reg, my_write_reg, e linear_interpolation sono identiche a quelle del progetto precedente. Dal configuratore CubeMX si può configurare il DMA della periferica I2C perché funzioni in modalità ciclica, ma non è chiaro come si possa in tal caso impostare la frequenza delle letture ad 1 Hz.