MASB | STM32
Comunicación serie IV

#stm32 #c #biomedical #microcontroladores #programacion

Albert Álvarez Carulla
hello@thealbert.dev
https://thealbert.dev/

5 de abril de 2022

Índice

1. Comunicación serie - Parte IV (STM32)

Ya hemos visto en qué consiste el I2C y cómo operar con él mediante la librería `Wire`. Ahora vamos a ver cómo hacer lo mismo con STM32CubeIDE. ¿Diferencias? Que a parte de utilizar el modo *blocking* o *polling*, también utilizaremos interrupciones para gestionar la comunicación I2C y liberar la CPU. En esta parte de la práctica no detallaremos aspectos del funcionamiento del BME280 que ya hemos visto en la primera parte (habilitar las diferentes medidas, las fórmulas de compensación, poner el sensor en modo "Normal", etc.), ni cómo operar los bytes para unirlos o desplazarlos (que se hace exactamente igual que en Arduino ya que es una característica propia del lenguaje). Si tenéis dudas o hay algo de lo que no os acordéis, no dudéis en echar un vistazo rápido a la parte anterior. ¡Vamos a ello!

1.1. Objetivos

- Comunicación I2C mediante las HAL de STM32F4 en modo *blocking.*
- Comunicación I2C mediante las HAL de STM32F4 mediante interrupciones.

1.2. Procedimiento

1.2.1. Modo *blocking*

Modo *blocking* es lo mismo que *polling.* (¿Que qué es *pooling*? Práctica 1 por favor…) Es decir, realiza una operación de escritura o lectura a través del bus I2C y queda a la espera hasta que haya finalizado la operación. Esto es lo que hemos hecho en Arduino. Vamos a hacer los mismo en STM32CubeIDE. Lo haremos para leer el ID del sensor BM280 y comprobar que está todo bien conectado. Una vez hecho esto, veremos cómo operar el I2C con interrupciones.

Figura 1: Las interrupciones son el camino.

1.2.1.1. Creación y configuración del proyecto Creamos en nuestra rama un proyecto en STM32CubeIDE con el nombre `masb-p08` en la carpeta de siempre. Nos vamos a la herramienta gráfica de configuración de las HAL (`masb-p08.ioc`) y nos vamos a la pestaña I2C1 dentro de *Connectivity*.

> En un microcontrolador, es muy común que haya más de un módulo/periférico repetido. Este es el caso del I2C, del cual existen dos. Trabajaremos con el número 1.

Habilitamos el I2C seleccionando I2C en el desplegable y, en este caso, no hace falta modificar nada más. Algunos parámetros importantes que podríamos modificar son la frecuencia de operación del reloj (SCL) o la dirección del microcontrolador como esclavo. Pero puesto que no necesitamos que el programar vaya super rápido ni operaremos en microcontrolador en modo esclavo (recordemos, en esta práctica el microcontrolador es el maestro), no modificaremos ni la frecuencia ni la dirección. Los *datasheets* de los sensores/componentes siempre indican el rango de frecuencias a los que pueden funcionar.

La configuración del I2C nos quedaría del siguiente modo. Fijaros cómo se nos configura automáticamente los pines PB7 y PB6 como SDA y SCL automáticamente. **¡Eso está mal!** Los periféricos también pueden salir por diferentes pines. Los que escoge la herramienta de configuración por defecto están mal y debemos de indicar los pines PB9 y PB8 como SDA y SCL, respectivamente.

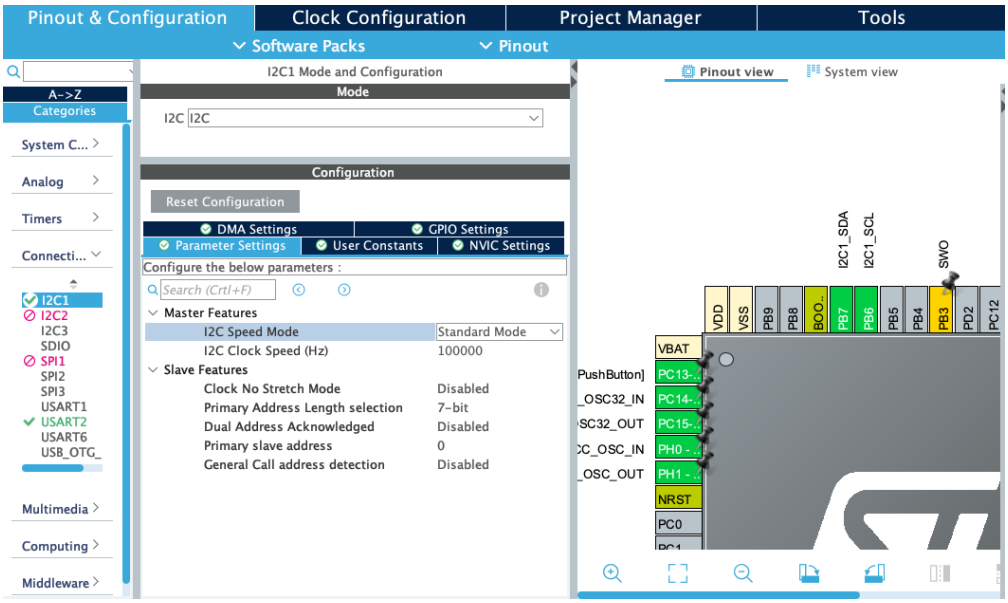

Figura 2: Configuración I2C.

Enviaremos mensajes al terminal serie, así que habilitad/configurad la UART para ello.

Le damos a guardar y generamos el código.

1.2.1.2. Petición del ID Vamos a ver si está todo bien conectado pidiendo al sensor su ID tal y como ya hicimos en Arduino. Acordemonos que esto es pedir el contenido del registro `0xD0` y que nos debe de devolver `0x60`. Vamos a hacerlos directamente en el archivo `main.c`.

```
...
/* USER CODE BEGIN PD */
#define BME280_ADDRESS 0x77
#define BME280_REG_ID 0xD0
/* USER CODE END PD */

...

/* USER CODE BEGIN PV */
uint8_t I2C_TxBuffer[32] = {0}, I2C_RxBuffer[32] = {0};
uint8_t UART_TxBuffer[64] = {0}, UART_RxBuffer[64] = {0};
/* USER CODE END PV */

...

/* USER CODE BEGIN 2 */
  I2C_TxBuffer[0] = BME280_REG_ID;
  HAL_I2C_Master_Transmit(&hi2c1, BME280_ADDRESS << 1, I2C_TxBuffer, 1,
      100);
  HAL_I2C_Master_Receive(&hi2c1, BME280_ADDRESS << 1, I2C_RxBuffer, 1,
     100);

  uint16_t UART_TxBufferElements = 0;

  if (I2C_RxBuffer[0] == 0x60) {
      UART_TxBufferElements = sprintf(UART_TxBuffer, "BME280 connected!
         ");
  } else {
      UART_TxBufferElements = sprintf(UART_TxBuffer, "No BME280 found
         ...");
  }
  HAL_UART_Transmit(&huart2, UART_TxBuffer, UART_TxBufferElements, 100)
      ;
  /* USER CODE END 2 */
...
```

Diferencias respecto Arduino. Al igual que con otros periféricos, cuando utilizamos las HAL debemos de indicar primeramente el puntero a la estructura que contiene la configuración del periférico (en este caso, `hi2c1`). Luego pasamos la dirección del esclavo, pero fijaros que desplazamos los bits una posición hacia la izquierda. ¿Por qué? Acordemonos que los 7 bits más significativos del primer byte son la dirección del esclavo y que el bit menos significativo indica si la operación es de lectura o escritura. En Arduino, este desplazamiento nos lo hacia por detrás la librería Wire. Aquí tenemos que hacerlo nosotros. El bit de operación lectura/escritura si que nos lo pone automáticamente.

> También podemos en la macro `BME280_ADDRESS` almacenar directamente el valor `0xEE`, que corresponde al valor `0x77` desplazado un bit hacia la izquierda. Pero lo hemos hecho de este modo para dejar claro que el desplazamiento debemos de hacerlo nosotros.

También indicamos a la hora de transmitir cuántos bytes transmitiremos (en Arduino hacíamos una instrucción independiente de escritura para cada byte). Y por último indicamos un *timeout*. Es decir, si en ese tiempo no se ha llevado a cabo la operación, se salta la instrucción. Esto no lo hicimos en Arduino y, si se diera el caso de un fallo en la comunicación por no estar conectado el sensor o cualquier otro motivo, el programa se quedaría enganchado en esos *while loops* que teníamos en el código.

Si todo está correcto, ejecutamos el código y deberíamos de ver en CoolTerm un "BME280 connected!".

En este pequeño ejemplo, las instrucciones detienen la ejecución del código hasta que la operación de lectura/escritura tenga lugar o hasta que haya transcurrido el tiempo indicado en el parámetro *timeout*. Inaceptable. Vamos a ver cómo hacerlo con interrupciones para que la CPU puede dedicarse hacer otras cosas más importantes.

1.2.2. Modo interrupciones

Lo primero que haremos es irnos al archivo `masb-p08.ioc` y habilitar las interrupciones del I2C desde la pestaña NVIC Settings.

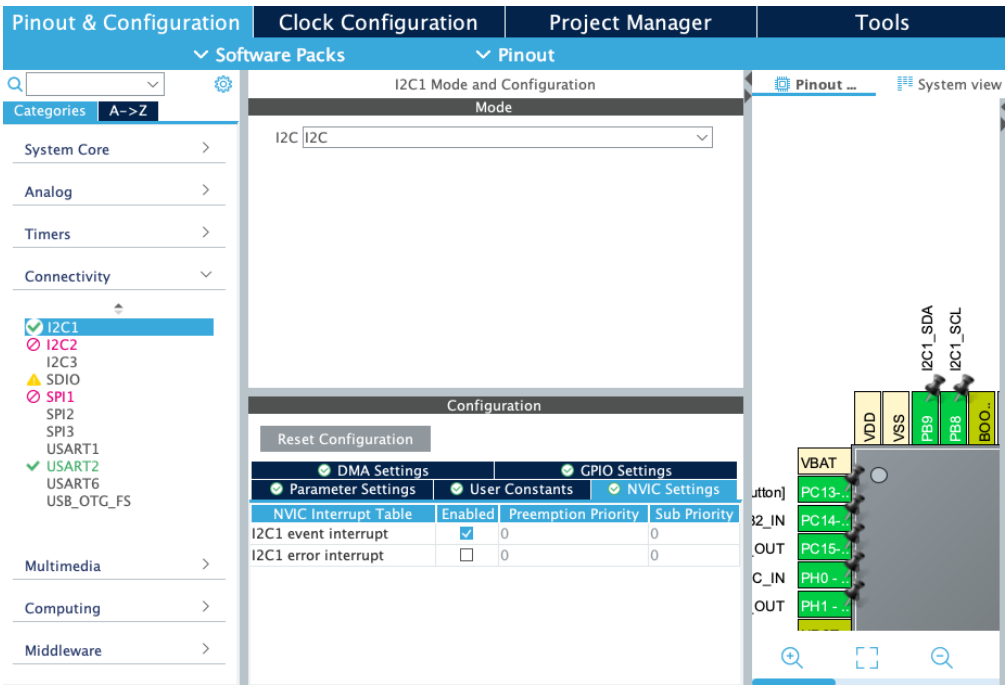

Figura 3: Configuración I2C.

Guardamos y generamos el código.

Nos vamos al archivo `main.c` y añadimos la función *callback* de la interrupción.

```
...
/* USER CODE BEGIN 4 */
void HAL_I2C_MasterRxCpltCallback (I2C_HandleTypeDef * hi2c) {

    uint16_t UART_TxBufferElements = 0;

    if (I2C_RxBuffer[0] == 0x60) {
        UART_TxBufferElements = sprintf(UART_TxBuffer, "BME280
            connected!");
    } else {
        UART_TxBufferElements = sprintf(UART_TxBuffer, "No BME280 found
            ...");
    }
    HAL_UART_Transmit(&huart2, UART_TxBuffer, UART_TxBufferElements,
        100);

}
/* USER CODE END 4 */
...
```

Hay muchos *callbacks* para el I2C. En este caso utilizamos el *callback* para atender la interrupción que tiene lugar al recibir los bytes que solicitábamos a través del I2C. Ahora en la función `main` tendríamos

únicamente:

```
...
/* USER CODE BEGIN 2 */
  I2C_TxBuffer[0] = BME280_REG_ID;
  HAL_I2C_Master_Transmit(&hi2c1, BME280_ADDRESS << 1, I2C_TxBuffer, 1,
      100);
  HAL_I2C_Master_Receive_IT(&hi2c1, BME280_ADDRESS << 1, I2C_RxBuffer,
     1);
/* USER CODE END 2 */
...
```

Comprobamos que funcione y… ¡tachán! Tenemos interrupciones funcionando. *Easy peasy lemon squeezy!*

Pregunta para nota… ¿Os habéis fijado que la transmisión seguimos haciéndola sin interrupciones? ¿Os imagináis por qué? Ya os lo digo yo, va. Si hacemos que ambas operaciones sean mediante interrupciones, tal y como tenemos ahora el código empezaría la transmisión y, sin haberle dado tiempo a acabar, empezaríamos después inmediatamente la recepción. Estaríamos machacando la operación de escritura o no se daría directamente la de lectura. Por ello, hacemos la operación de escritura de manera *blocking* para que se quede el código allí a la espera de que se haya realizado una transmisión y, una vez hecha, iniciamos la recepción. Esto lo hacemos aquí para no complicar más el proyecto con código adicional, pero la escritura y lectura mediante pueden convivir sin problemas y simplemente deberíamos de añadir código que compruebe que se ha hecho la transmisión antes de realizar una recepción.

1.3. Reto

Aquí acaba la parte guiada… ¿Os lo veis venir, no…? Pues sí. Hay que hacer la parte anterior (activar sensor, habilitar medidas y obtener las mediciones de temperatura, humedad y presión), pero en STM32CubeIDE. Algo importante: no quiero que me odiéis, así que **hacedlo con las funciones *blocking*** (sin interrupciones). De este modo, la migración de Arduino a STM32CubeIDE es casi directa (si sois un poco “pillos”, es cuestión de, **literal**, 5 minutos…).

1.4. Evaluación

1.4.1. Entregables

Estos son los elementos que deberán de estar disponibles para el profesorado de cara a vuestra evaluación.

- ☐ **Commits** Se os deja a vuestro criterio cuándo realizar los *commits*. No hay número mínimo/máximo y se evaluará el uso adecuado del control de versiones (creación de ramas, *commits* en el momento adecuado, mensajes descriptivos de los cambios, …).
- ☐ **Reto**
- ☐ ~~**Informe**~~ No hay informe.

1.4.2. Pull Request

Finalizado el informe, acordaos de hacer el *push* pertinente y cread un *Pull Request* (PR) de vuestra rama a la `master`. Acordaos de ponerme como *Reviewer*. No hagáis *merge*.

1.4.3. Rúbrica

La rúbrica que utilizaremos para la evaluación la podéis encontrar en el CampusVirtual. Os recomendamos que le echéis un vistazo para que sepáis exactamente qué se evaluará y qué se os pide.

1.5. Conclusiones

Pues ya hemos visto cómo utilizar el I2C en STM32CubeIDE. El I2C es lo mismo tanto en Arduino como en STM32CubeIDE, lo único que cambia son las librerías de STM32, que nos requieren algún paso adicional para hacerlas funcionar, pero que nos ofrecen una mayor flexibilidad. Esto habilita la implementación de proyectos más avanzados que puedan operar en regímenes de bajo consumo o en aplicaciones de alta computación.

¡Vamos que esta ha sido la última práctica! Ya tenemos todo para hacer el proyecto.