TD n°7 - le bus I2C

L'objectif du TD est d'étudier une interface générique d'accès au bus I2C. Cette interface a pour but de permettre l'échange d'octets entre le microcontrôleur (qui est le maître du bus) et un disposif esclave accroché sur le bus. Le labo se concentrera sur l'utilisation de cette interface pour dialoguer avec deux capteurs.

L'interface est définie dans les fichiers `lib/i2c.h` et `lib/i2c.c` de la manière suivante :

```
/* i2c interface initialization in master mode */
int i2c_master_init(I2C_t *i2c);

/* i2c_write : write n bytes from buf to slave identified by addr */
int i2c_write(I2C_t *i2c, uint8_t addr, uint8_t* buf, uint32_t n);

/* i2c_read : read n bytes from slave identified by addr to buf */
int i2c_read(I2C_t *i2c, uint8_t addr, uint8_t* buf, uint32_t n);

/* i2c_write_read : write nwr bytes from buf to slave identified by addr,
 *                  then read nrd bytes to buf */
int i2c_write_read(I2C_t *i2c, uint8_t addr, uint8_t* buf, uint32_t nwr, uint32_t nrd);
```

Du point de vue de l'interface de programmation, les coupleurs I2C du STM32F411, présentent une interface de registres constituée de

– 2 registres de contrôle : `CR1` et `CR2` + 1 registre de configuration d'horloge `CCR`,

– 2 registres d'état : `SR1` et `SR2` (état + flags d'interruption),

– 1 registre d'entrée/sortie `DR` permettant la réception d'un octet en provenance du bus I2C ou l'envoi d'un octet vers le bus I2C.

1. Initialisation de l'interface I2C : i2c_master_init

a. Initialisations spécifiques au coupleur `_I2C1`.

   i. Ce coupleur est sur le bus APB1. Indiquer le bit à configurer dans le registre `_RCC->APB1ENR` pour valider l'horloge dans le coupleur `_I2C1`.

   ii. Il est accroché sur le bus APB1. Retrouver les broches utilisées pour les lignes SDA et SCL à partir du schéma. La configuration est a écrire dans le fichier `include/config.h`.

b. L'initialisation du coupleur consiste essentiellement à configurer l'horloge nécessaire à la génération des trames du bus I2C. La fréquence du bus APB1 est de 42 MHz. Le code est le suivant :

```
// peripheral input clock frequency
        i2c->CR2 = (sysclks.apb1_freq/1000000);

        // clock control register config
        i2c->CCR = (1<<15) | (sysclks.apb1_freq/1200000);
```

```
// configure the rise time register (from ST Cube library)
// TRISE = risetime / Tapb1 = 300e-9 * apb1_freq = 300 * apb1_freq_MHz * 1e6 / 1e9
i2c->TRISE =(((sysclks.apb1_freq/1000000) * 300) / 1000) + 1;

// analog noise filter on, digital filter off
i2c->FLTR = i2c->FLTR & (~0x1F);
```

Montrer que la configuration choisie dans le registre `CCR` permet de faire fonctionner la liaison I2C en mode “Fast” (horloge SCL à 400 kHz).

L'initialisation se termine par la configuration du NVIC (irqn=31 pour la gestion des évènements issus de l'interface I2C, et irqn=32 pour la gestion des erreurs), et la mise en route de l'interface (bit 0 du registre `CR1`).

2. Transfert de données : maître émetteur et esclave récepteur : i2c_write

a. La fonction `i2c_write` permet d'initier une trame de communication avec un esclave dont l'adresse est fournie. On donne également en paramètre un tableau d'octets à envoyer à l'esclave.

```
int i2c_write(I2C_t *i2c, uint8_t addr, uint8_t* buf, uint32_t n)
{
    I2CContext *ctx = i2c_get_context(i2c);

    if (ctx) {
        ctx->status = I2C_BUSY;

        ctx->address            = addr;
        ctx->buffer             = buf;
        ctx->n_to_read          = 0;
        ctx->n_to_write         = n;
        ctx->n_wr = ctx->n_rd   = 0;
        ctx->op                 = I2C_WRITE;

        i2c->CR2 |= I2C_IT_ERR | I2C_IT_EVT | I2C_IT_BUF;
        i2c->CR1 |= I2C_CR1_START;

        while (ctx->status == I2C_BUSY) ;          // wait for the transaction to be done

        return ctx->status;
    }

    return I2C_ERROR;
}
```

La structure `I2CContext` initialisée est destinée à accompagner le déroulement de la trame de communication I2C entre le maître et l'esclave. La trame démarre dès que le bit de start est positionné à 1.

Identifier le rôle des bits `I2C_IT_EVT`, `I2C_IT_BUF`, `I2C_IT_ERR` et `I2C_CR1_START`.

b. A partir de la génération de la condition de start, le déroulement de la trame est rythmé par la génération d'évènements (interruptions) qui permettent à notre code de reprendre la main pour réaliser les actions suivantes. La séquence d'évènements générés et d'actions à réaliser est explicitée sur la figure 164 p 480.

   i. On garde une trace de l'évènement qui a provoqué la demande d'interruption dans le registre d'état `SR1`. Identifier les bits `SB`, `ADDR`, `TxE` et `BTF` et expliciter leur rôle en rapport avec le déroulement de

la trame.

ii. Expliquer qui positionne le bit A (Acknowledge).

iii. Expliquer l'importance de la partie "cleared by …" dans les commentaires relatifs aux évènements.

iv. Expliquer pourquoi un évènement EV8 sera reçu immédiatement après avoir traité l'EV8_1, alors que les évènements EV8 sont espacés dans le temps.

v. Expliquer la différence entre les évènements EV8 et EV8_2.

c. Compléter le code de la routine d'interruption suivante qui permet de traiter l'écriture de *n* octets sur le bus I2C.

```
void event_handler(I2CContext *ctx) {
    I2C_t *i2c = ctx->i2c;
    uint32_t sr1 = i2c->SR1;

    if (sr1 & I2C_SR1_SB) {                    // EV5:



    } else if (sr1 & I2C_SR1_ADDR) {           // EV6:



    } else if (sr1 & I2C_SR1_TxE) {            // EV8:
        if (ctx->n_wr < ctx->n_to_write) {     //   EV8 or EV8_1



        } else if ((ctx->n_wr == ctx->n_to_write) && (sr1 & I2C_SR1_BTF)) {     // EV8_2



        }
    }
}
```

3. Transfert de données : maître récepteur et esclave émetteur : i2c_read

Le principe de la lecture est identique. La trame est donnée sur la figure 165 p 482.

a. Analyse de la trame

   i. Indiquer le rôle du bit `RxNE` du registre d'état `SR1`.

   ii. Expliquer qui positionne le bit `A` (Acknowledge), et à quelle valeur au cours du déroulement de la trame.

   iii. Indiquer comment le maître peut positionner le bit `A` dans une trame. A quel moment doit-il le faire ?

b. Compléter le code de la routine d'interruption suivante qui permet de traiter la lecture de *n* octets sur le bus I2C.

```
void event_handler(I2CContext *ctx) {
        I2C_t *i2c = ctx->i2c;
        uint32_t sr1 = i2c->SR1;

        if (sr1 & I2C_SR1_SB) {                 // EV5:



        } else if (sr1 & I2C_SR1_ADDR) {        // EV6:
                if (ctx->op==I2C_READ) {




                }
        } else if (sr1 & I2C_SR1_RxNE) {        // EV7:




        }
}
```

4. mode combiné : i2c_write_read

Ce mode de fonctionnement combine les deux modes précédents. On démarre par une écriture. Lorsque tous les octets à écrire ont été transmis sur le bus, au lieu d'arrêter la communication, on réinitialise une trame de transmission (en lecture cette fois) en imposant via le registre `CR1` une nouvelle condition de start, ce qui permet d'enchaîner les deux trames avec l'assurance de ne pas perdre la maîtrise du bus.