



Tutoriels STM32F0

Périphériques standards

I²C (Inter-Integrated Circuit)

Arouna DARGA – enseignant chercheur Sorbonne Université
arouna.darga@sorbonne-universite.fr

Table des matières

1. Introduction.....	3
2. Mise place du projet pour le tutoriel	5
3. Configuration d'un périphérique I2C.....	5
4. Premiers essais	7
5. Transaction de lecture.....	12

1. Introduction

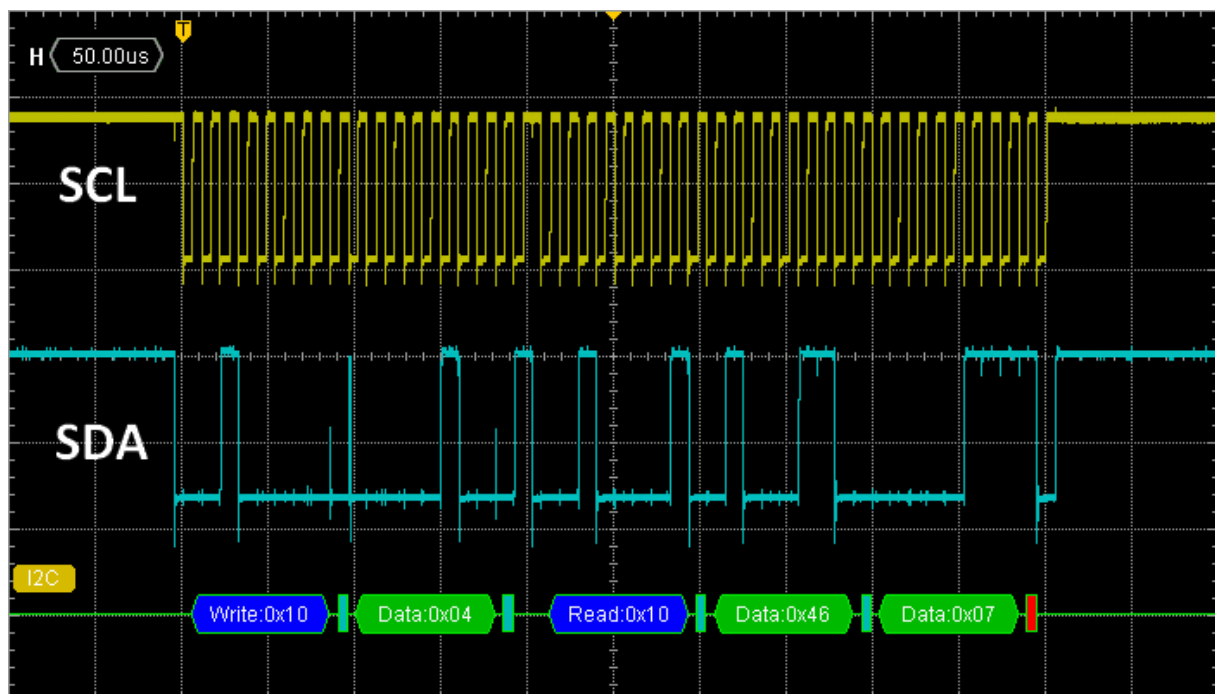


I²C (Inter-Integrated Circuit), qui se prononce I-2-C, est un bus série synchrone, multi-maître, multi-esclaves, à commutation par paquets, à terminaison unique, inventé en 1982 par Philips Semiconductors (aujourd'hui NXP Semiconductors). Il est largement utilisé pour relier des circuits périphériques peu rapides à des processeurs et à des microcontrôleurs dans le cadre de communications intra-carte à courtes distances. I²C s'écrit également I2C (prononcé I-two-C) ou IIC (prononcé I-I-C). (<https://en.wikipedia.org/wiki/I%C2%B2C>)

I2C est un bus à 2 fils (3 en réalité, si vous considérez GND) :

- Un pour le signal d'horloge → SCL
- Un pour le signal des données → SDA

Avant d'entrer dans les détails, il convient de regarder la capture d'écran ci-dessous, qui représente le STM32 en train de lire un résultat de 16 bits provenant d'un capteur de lumière ambiante. Comme vous le remarquez peut-être déjà, le processus n'est pas tout à fait simple... Les données attendues se trouvent vers la fin de la transaction (0x0746) et représentent un certain nombre d'unités de lumière (lux). Près de 50 cycles d'horloge ont été nécessaires pour l'ensemble du transfert. C'est 500µs en considérant une fréquence d'horloge de 100kHz.



Une seule transaction I2C implique un **dispositif maître** et un **dispositif esclave**. Dans ce qui suit, le dispositif **maître est le microcontrôleur**, et le **dispositif esclave est un capteur externe**. Les deux peuvent prendre le contrôle de la ligne SDA, mais **pas en même temps**. La ligne SCL est toujours piloté par le maître.

Comme le maître et l'esclave peuvent tous deux prendre le contrôle de la ligne **SDA**, il pourrait y avoir un risque de situations conflictuelles où les deux tentent de tirer ou de pousser des niveaux opposés en même temps. Un tel cas entraînerait de graves dommages pour le côté le plus faible. Ce risque est éliminé par l'utilisation de sorties à **Open Drain** pour toutes les lignes **SCL** et **SDA** du bus **I2C**. Le "1" logique est donc obtenu par des résistances de tirage vers le haut externes. Le "1" est l'état par défaut, lorsqu'aucun dispositif ne place un zéro sur le bus. Le "0" gagne toujours, si au moins un dispositif le signale.

Vous pouvez consulter les spécifications et le [manuel d'utilisation du bus I2C](#) (Rév. 6 - 4 avril 2014). Il ne compte que 64 pages !

Pour la formation sur le bus I2C et SPI (vue plus tard), nous avons besoin de matériels qui ne sont pas directement disponibles sur les cartes Nucleo®, c'est-à-dire de certains composants I2C/SPI avec lesquels nous pouvons communiquer. Vous pouvez connecter n'importe quel appareil I2C au STM32 tant que les niveaux numériques sont de 0V et 3,3V. Pour ces tutoriels, un ensemble de capteurs environnementaux a été retenu pour le plaisir de mesurer quelque chose que nous pouvons ressentir :

- Le [VEML7700](#) de Vishay Semiconductors est un capteur de lumière ambiante I2C
- Le [LPS25HB](#) de STMicroelectronics est un capteur de pression absolue (baromètre) I2C/SPI. Nous utiliserons l'interface SPI dans le prochain tutoriel

Pour simplifier l'apprentissage nous utiliserons des modules à base de ces capteurs

L'image ci-dessous présente le module à base du capteur [VEML7700](#). Ce module est fabriqué et vendu par la société Adafruit. Une page est consacrée à ce module. [Page adafruit du module VEML7700](#).



Les lignes de communication I2C/SPI des capteurs seront reliées au MCU STM32F072 par les pins donner dans le tableau ci-dessous. Il est possible de changer de pins.

Périphérique	Ligne	Nucleo-STM32F072
I2C1	SCL	PB8 (AF1)
	SDA	PB9 (AF1)
SPI1	SCK	PB3 (AF0)
	MISO	PB4 (AF0)
	MOSI	PB5 (AF0)
GPIO	CS	PC7
	INT	PB6

2. Mise place du projet pour le tutoriel

Pour ce tutoriel, nous **n'allons pas utiliser le mécanisme d'interruptions**. Afin d'éviter des erreurs de compilation, Il faut donc un projet dédié à ce tutoriel. Pour cela, il faut cloner le projet « **my_project** » qui ne contient pas d'exemple d'interruption, et nommer le projet cloné par « **Tutoriels_I2C_SPI** ». Pour la procédure de clonage suivre celle décrite dans le tutoriel « **Tutoriels STM32F0-Environnement de développement v05022021.pdf** » (à partir de la page 30), disponible sur moodle.

3. Configuration d'un périphérique I2C

Le code ci-dessous est une configuration de base pour le périphérique **I2C1**. Il laisse la plupart des paramètres à leurs valeurs par défaut. Vous devez ajouter cette fonction à votre fichier **bsp.c** et déclarer le prototype de la fonction dans **bsp.h**.

Les principales caractéristiques de la fonction init sont les suivantes :

- Configurer les broches **PB8** et **PB9** pour qu'elles fonctionnent en mode lignes **SCL** et **SDA** à travers leurs fonctions alternatives **I2C1 (AF1)**
- Assurez-vous que l'architecture électronique des sorties **PB8** et **PB9** sont configurer en **Open Drain** (important)
- Configurer les horloges du périphérique I2C1. Cela se fait en sélectionnant l'horloge principale I2C (à partir de RCC), puis en réglant le prescaler de l'horloge, puis en programmant le nombre de cycles pour les états de l'horloge (haut et bas).

Dans la configuration ci-dessous, nous choisissons :

- SYSCCLK = 48MHz pour l'horloge principale I2C
- Prescaler = /4 pour obtenir une horloge de 12MHz
- Un délai de 60 cycles (=5µs) pour l'état haut de l'horloge
- Un délai de 60 cycles (= 5µs) pour l'état bas de l'horloge

La période d'horloge résultante est de 10µs, ce qui correspond à une fréquence de bus de 100kHz. Dans la spécification I2C, 100 kHz (c'est-à-dire 100 kbit/s) est désigné comme le "mode standard". C'est le mode le plus lent que vous trouverez adapté à la plupart des dispositifs I2C disponibles. Cependant, il n'y a pas de limite à l'utilisation d'horloges encore plus lentes. Vous pouvez faire fonctionner l'I2C à 1 Hz si vous le souhaitez... Parfois, cela facilite le débogage et le test.

```

/*
 * BSP_I2C1_Init()
 * Initialise le peripherique I2C1
 * SCL -> PB8
 * SDA -> PB9
 */
void BSP_I2C1_Init()
{
    // configuration des pins pour I2C1
    // SCL -> PB8
    // SDA -> PB9

    // Activer horloge de GPIOB
    RCC->AHBENR |= RCC_AHBENR_GPIOBEN;

    // Configurer les Pins PB8 et PB9 en mode AF
    GPIOB->MODER &= ~(GPIO_MODER_MODER8 | GPIO_MODER_MODER9);
    GPIOB->MODER |= (0x02 <<16U) | (0x02 <<18U);

    // Choix de AF1 (I2C1) pour les pins PB8 et PB9
    GPIOB->AFR[1] &= ~(0x000000FF);
    GPIOB->AFR[1] |= 0x00000011;

    // Choix de la techno Open-Drain pour les pins PB8 et PB9
    GPIOB->OTYPER |= GPIO_OTYPER_OT_8 | GPIO_OTYPER_OT_9;

    // Selectionner SYSCLK comme horloge I2C1 (48MHz)
    RCC->CFGR3 |= RCC_CFGR3_I2C1SW;

    // Activer horloge du peripherique I2C1
    RCC->APB1ENR |= RCC_APB1ENR_I2C1EN;

    // Assurons-nous que I2C1 est desactiver avec de le configurer
    I2C1->CR1 &= ~I2C_CR1_PE;

    // Reinitialisation de la configuration I2C1 aux valeurs par défaut
    I2C1->CR1 = 0x00000000;
    I2C1->CR2 = 0x00000000;
    I2C1->TIMINGR = 0x00000000;

    // Configurer le timing pour 100kHz, 50% rapport clycique
    utilisation
    I2C1->TIMINGR |= ((4 -1) <<I2C_TIMINGR_PRESC_Pos); // Clock
prescaler /4 -> 12MHz
    I2C1->TIMINGR |= (60 <<I2C_TIMINGR_SCLH_Pos); // High half-
period = 5µs
    I2C1->TIMINGR |= (60 <<I2C_TIMINGR_SCLL_Pos); // Low half-
period = 5µs

    // Activer I2C1
    I2C1->CR1 |= I2C_CR1_PE;
}

```

4. Premiers essais

Expérimentons le petit code ci-dessous dans la fonction **main()**. Comme vous pouvez le voir, **I2C** est tout simplement initialisé, puis nous lançons un événement **START** pour tester.

```
#include "stm32f0xx.h"
#include "bsp.h"
#include "delay.h"
#include "main.h"

#include <math.h>

static void SystemClock_Config(void);

int main(void)
{
    // Configurer horloge du systeme pour 48MHz a partir de la source
    8MHz HSE
    SystemClock_Config();

    // Initialiser la LED et le bouton bleu
    BSP_LED_Init();
    BSP_PB_Init();

    // Initialiser la console de debogage
    BSP_Console_Init();
    mon_printf("\r\n La est Console Ready!\r\n");
    mon_printf("SYSCLK = %d Hz\r\n", SystemCoreClock);

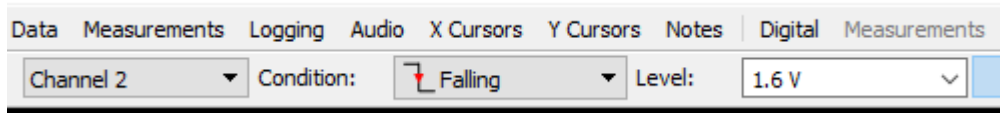
    // Initialiser le périphérique I2C1
    BSP_I2C1_Init();

    // Demarrer la transaction I2C
    I2C1->CR2 |= I2C_CR2_START; // <-- Mettre un Point arret ici

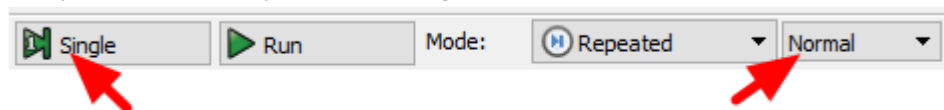
    while(1)
    {
    }
}
```

Afin de capturer la transaction I2C, vous devrez effectuer les opérations suivantes très soigneusement :

1. Compilez 🛠️, téléversez le code dans le MCU et démarrez une session de débogage 🐞.
2. Configurez un oscilloscope afin d'observer les signaux **SCL** et **SDA** (canaux 1 (fil orange) et 2 (fil bleu)).
3. Configurez le déclenchement de l'oscilloscope sur le front descendant de SDA (canal 2 (Bleu)).



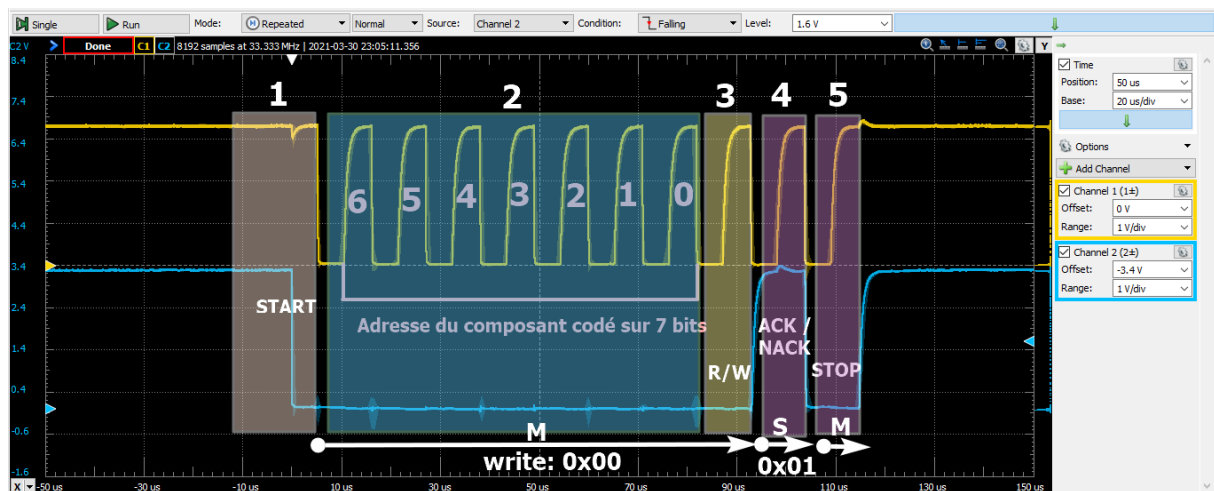
4. Placez un point d'arrêt juste avant que la transaction I2C ne soit lancée (c'est-à-dire devant "I2C1->CR2 |= I2C_CR2_START").
5. Exécutez 🏃 le programme jusqu'à cette ligne, assurez-vous que le point d'arrêt est atteint.
6. Puis passez l'oscilloscope en mode Single shot



7. Reprenez l'exécution du programme 🏃. Ceci exécutera la séquence I2C et sera ensuite piégé dans la boucle while(1).

Si vous ne voyez rien sur l'oscilloscope, mettre en pause 🛑, puis réinitialisez le programme 🔄, revoyez les réglages de votre oscilloscope et répétez les étapes 5, 6 et 7. Vous devez vraiment vous sentir à l'aise avec ce qui précède, car nous allons beaucoup l'utiliser dans ce tutoriel.

Si vous réussissez, vous capturerez le début de la transaction **I2C** comme indiqué ci-dessous.



Ce que nous avons (de gauche à droite) :

1. Comme spécifié par la norme **I2C**, la transaction commence par un **SDA** à l'état bas alors que **SCL** est à l'état haut (**condition START**). Les deux lignes **SCL** et **SDA** sont contrôlées par le maître (STM32).
2. Une adresse de dispositif de **7 bits** est ensuite transmise par le maître sur **SDA** à chaque front montant de **SCL** (0b0000000 ici).
3. Au **8ème front montant** de **SCL**, le maître spécifie si la transaction en cours est un **READ (1)** ou un **WRITE (0)**. Nous avons ici un **WRITE**. Les modes maître en écriture et maître en lecture sont appelés maître émetteur (écriture) et maître récepteur (lecture). En pratique, cela indique simplement qui prend le contrôle de la ligne SDA après le prochain accusé de réception : le maître (écriture) ou l'esclave (lecture).
4. Sur le front descendant **du 8ème cycle SCL**, le **maître passe en mode écoute**, lisant un accusé de réception fourni par l'esclave sur le **9ème front montant SCL**. En raison des résistances de tirage (pull-up), un '1' ici signifie qu'aucun dispositif n'accuse effectivement réception de la transaction.
5. Puisqu'aucun dispositif n'a acquitté, le maître prend en charge la **ligne SDA** sur le front descendant du **9ème cycle SCL** et arrête la transaction par un **SDA montant** alors que **SCL est élevé** (condition **STOP**). Cela termine la transaction proprement.

La transaction précédente n'a pas fait l'objet d'un accusé de réception pour une raison simple : il n'y a en fait aucun périphérique esclave à l'adresse 0x00 sur le bus. L'adresse esclave est une caractéristique intégrée pour un modèle de périphérique donné. Elle est fournie par le fabricant, et donc spécifiée dans la fiche technique du dispositif. En supposant que nous voulons adresser notre capteur de lumière, la fiche technique du VEML7700 spécifie à la page 4 que l'adresse I2C 7-bit de ce capteur est 0x10.

Modifiez le fichier main comme ci-dessous :

```
#include "stm32f0xx.h"
#include "bsp.h"
#include "delay.h"
#include "main.h"

#include <math.h>

static void SystemClock_Config(void);

int main(void)
{
    // Configurer horloge du systeme pour 48MHz a partir de la source
    8MHz HSE
    SystemClock_Config();

    // Initialiser la LED et le bouton bleu
    BSP_LED_Init();
    BSP_PB_Init();

    // Initialiser la console de debogage
    BSP_Console_Init();
    mon_printf("\r\n La est Console Ready!\r\n");
    mon_printf("SYSCLK = %d Hz\r\n", SystemCoreClock);

    // Initialiser le périphérique I2C1
    BSP_I2C1_Init();

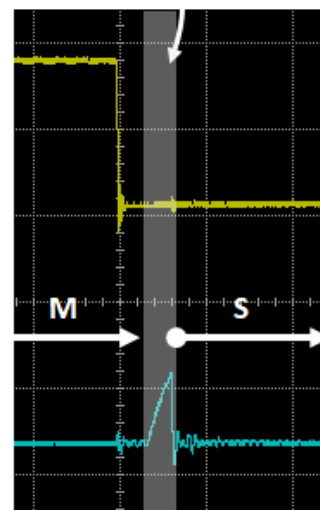
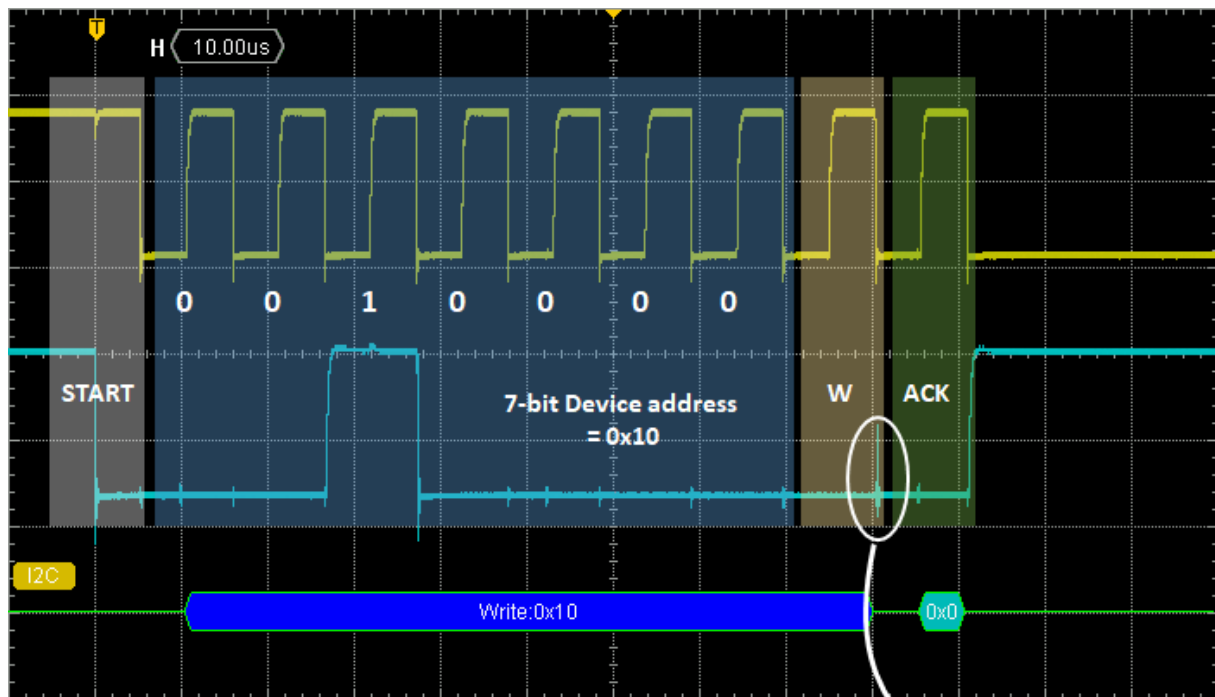
    // Adresse du composant VEML7700 (SLAVE) est 0x10
    I2C1->CR2 &= ~I2C_CR2_SADD_Msk;
    I2C1->CR2 |= ((0x10 <<1U) <<I2C_CR2_SADD_Pos);

    // Demarrer la transaction I2C
    I2C1->CR2 |= I2C_CR2_START; // <-- Mettre un Point arret ici

    while(1)
    {

    }
}
```

Répétez les étapes ci-dessus pour capturer le début de la transaction **I2C**. Si vous manquez la capture, il est possible que vous devriez réessayer, ou même débrancher/rebrancher la carte Nucleo® pour réinitialiser la machine d'état I2C esclave, car le code ci-dessus laisse le décodeur **I2C esclave** dans un état d'attente " non terminé ", d'où il pourrait ne pas être en mesure de redémarrer correctement.



Cette fois, une adresse de périphérique existante est transmise, et le bit d'accusé de réception attendu '0' est présent sur le **9ème cycle SCL**. Il n'y a donc pas d'arrêt généré par le maître et la transaction est prête pour l'étape suivante du protocole.

Obtenir un accusé de réception du composant Slave est une très bonne nouvelle. Cela signifie "OK, mon capteur répond !". Il est d'ailleurs intéressant de zoomer autour du front descendant de **SCL du 8ème cycle**. C'est là que **SDA passe du contrôle maître au contrôle esclave**. Le petit **pic sur SDA** est dû à courte durée séparant la libération du maître de la prise en charge de l'esclave, tirant l'accusé de réception '0'. Ce petit pic vous indique que votre composant esclave est là, et qu'il répond ! Cela valide en quelque sorte votre matériel, vos paramètres d'E/S, la vitesse de communication, ... Très cool en effet !

	Sauvegarder votre code main et les captures d'écrans montrant vos essais
--	--

5. Transaction de lecture - READ

Un dispositif esclave I2C ou SPI, quelle que soit sa fonction (un capteur, un écran, une mémoire, un convertisseur, une extension d'E/S...), est toujours vu par le maître comme un ensemble de registres (emplacements mémoire) dans lesquels vous pouvez écrire ou lire. Chacun de ces emplacements de mémoire possède une adresse et contient des données. Dans le cas d'un capteur, vous devrez écrire des données dans certains registres pour configurer les options ou activer la mesure. Ensuite, vous lirez les résultats dans d'autres registres. À peu près comme vous le feriez avec n'importe quel périphérique interne en fait. Celui-ci est uniquement externe.

La plupart des périphériques I2C/SPI possèdent un registre contenant un numéro d'identification appelé "Device ID", ou "Who Am I". Lorsque l'on développe une communication avec un nouveau périphérique, c'est toujours un bon début d'essayer de lire ce registre car on peut facilement vérifier si la valeur renvoyée est bonne ou non. Malheureusement, le VEML7700 ne possède pas un tel registre. Le registre à l'adresse 0x00 est un registre de configuration que nous pouvons essayer de lire pour commencer.

La page 4 de la fiche technique du VEML7700 spécifie le protocole 'Read'. Il est conforme à la norme I2C :



Le début de la transaction a déjà été couvert jusqu'au premier accusé de réception de l'esclave. L'étape suivante consiste à envoyer ce que l'on appelle le 'Command code', qui correspond essentiellement à l'adresse du registre que vous voulez lire (0x00).

Avant d'aller plus loin, vous devez vous familiariser avec le périphérique I2C du STM32 et les différents drapeaux d'état qu'il utilise pour contrôler la machine d'état de la transaction.

25.7.7 Interrupt and status register (I2C_ISR)

Address offset: 0x18

Reset value: 0x0000 0001

Access: No wait states

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	ADDCODE[6:0]							DIR
								r							r
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BUSY	Res.	ALERT	TIME OUT	PEC ERR	OVR	ARLO	BERR	TCR	TC	STOPF	NACKF	ADDR	RXNE	TXIS	TXE
r		r	r	r	r	r	r	r	r	r	r	r	r	rs	rs

Bien que décrit dans le manuel de référence, le comportement des drapeaux I2C n'est pas toujours facile à comprendre. Pour rendre les choses plus visuelles, nous pouvons utiliser des GPIOs de réserve pour rapporter l'état en temps réel de certains drapeaux que nous voulons surveiller.

Le premier drapeau qui nous intéresse est **TXIS**. Le manuel de référence indique que **TXIS** est activé lorsque le périphérique I2C attend un nouvel octet à envoyer. Rappelez-vous que notre prochaine étape est d'envoyer le 'Command code' 0x00. Donc **TXIS** semble être le candidat parfait.

Commençons par écrire une petite fonction qui attend qu'un ou plusieurs drapeaux se déclenchent, tout en signalant l'état des drapeaux sur GPIO en temps réel :

```
static uint8_t wait_for_flags()
{
    uint8_t    exit = 0;

    while(exit == 0)
    {
        // TXIS -> PA0
        // Quitter attente si TXIS passe a '1'
        if ((I2C1->ISR & I2C_ISR_TXIS) != 0)
        {
            GPIOA->BSRR = GPIO_BSRR_BS_0;    // mise a '1' de PA0
            exit = 1;
        }
        else GPIOA->BSRR = GPIO_BSRR_BR_0;    // mise a '0' de PA0

        // Ajouter autres sortie de drapeaux
        // ...
    }

    return exit;
}
```

Ajoutez cette fonction au fichiers main.c

Évidemment, pour que cette fonction fonctionne, vous devrez configurer les GPIOs (PA0 pour l'instant) en mode Output Push-Pull. Pour cela, je vous suggère d'ajouter une nouvelle fonction BSP appelée "**BSP_DBG_Pins_Init()**" dans votre fichier **bsp.c**. N'oubliez pas d'appeler cette fonction dans la partie **init** de **main()**.

Maintenant, modifiez main() pour surveiller le drapeau TXIS après le démarrage de la transaction :

```
#include "stm32f0xx.h"
#include "bsp.h"
#include "delay.h"
#include "main.h"

#include <math.h>

static void SystemClock_Config(void);
static uint8_t wait_for_flags();

int main(void)
{
    // Configurer horloge du systeme pour 48MHz a partir de la source
    8MHz HSE
    SystemClock_Config();

    // Initialiser la LED et le bouton bleu
    BSP_LED_Init();
    BSP_PB_Init();
    // Initialiser les pins de debug
    BSP_DBG_Pins_Init();

    // Initialiser la console de debogage
    BSP_Console_Init();
    mon_printf("\r\n La est Console Ready!\r\n");
    mon_printf("SYSCLK = %d Hz\r\n", SystemCoreClock);

    // Initialiser le périphérique I2C1
    BSP_I2C1_Init();

    // Adresse du composant VEML7700 (SLAVE) est 0x10
    I2C1->CR2 &= ~I2C_CR2_SADD_Msk;
    I2C1->CR2 |= ((0x10 <<1U) <<I2C_CR2_SADD_Pos);

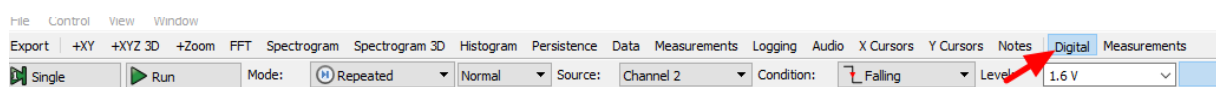
    // Demarrer la transaction I2C
    I2C1->CR2 |= I2C_CR2_START; // <-- Mettre un Point arret ici
    wait_for_flags();

    while(1)
    {

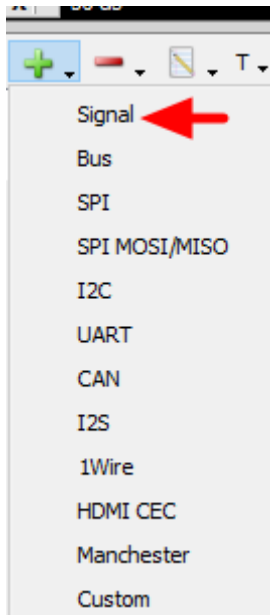
    }
}
```

Connectez l'entrée sortie numérique N° 2 du boîtier analog discovery à la patte PA0, puis essayez de capturer le début de la transaction I2C :

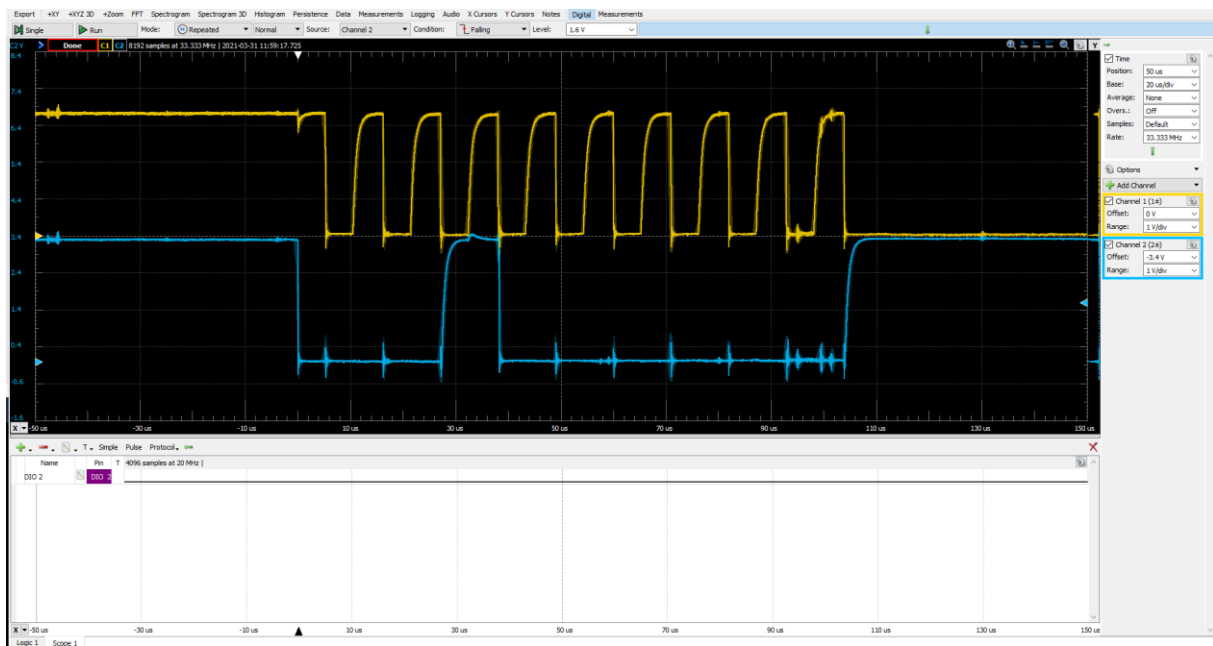
Cliquez sur digital :



Puis rajouter un signal  et choisir DIO 2



Vous obtenez l'image ci-dessous



Comme vous pouvez le voir, le drapeau **TXIS** n'est pas activé après l'envoi de l'adresse. C'est parce que **TXIS** ne vous dit pas vraiment que vous pouvez envoyer un nouvel octet. Il vous dit que vous DEVEZ envoyer un octet. Comment sait-il que nous devons ? Nous devons le lui dire.

Cela se fait en remplissant le champ **NBYTES** dans le registre **CR2**. En examinant l'ensemble du protocole de lecture, nous pouvons constater qu'une autre condition **START** se produit juste après l'envoi du '**Command code**'. Par conséquent, le nombre d'octets que nous voulons transférer pour l'instant n'est qu'un **seul**.

Essayons le code ci-dessous

```
#include "stm32f0xx.h"
#include "bsp.h"
#include "delay.h"
#include "main.h"

#include <math.h>

static void SystemClock_Config(void);
static uint8_t wait_for_flags();

int main(void)
{
    // Configurer horloge du systeme pour 48MHz a partir de la source
    8MHz HSE
    SystemClock_Config();

    // Initialiser la LED et le bouton bleu
    BSP_LED_Init();
    BSP_PB_Init();
    // Initialiser les pins de debug
    BSP_DBG_Pins_Init();

    // Initialiser la console de debogage
    BSP_Console_Init();
    mon_printf("\r\n La est Console Ready!\r\n");
    mon_printf("SYSCLK = %d Hz\r\n", SystemCoreClock);

    // Initialiser le périphérique I2C1
    BSP_I2C1_Init();

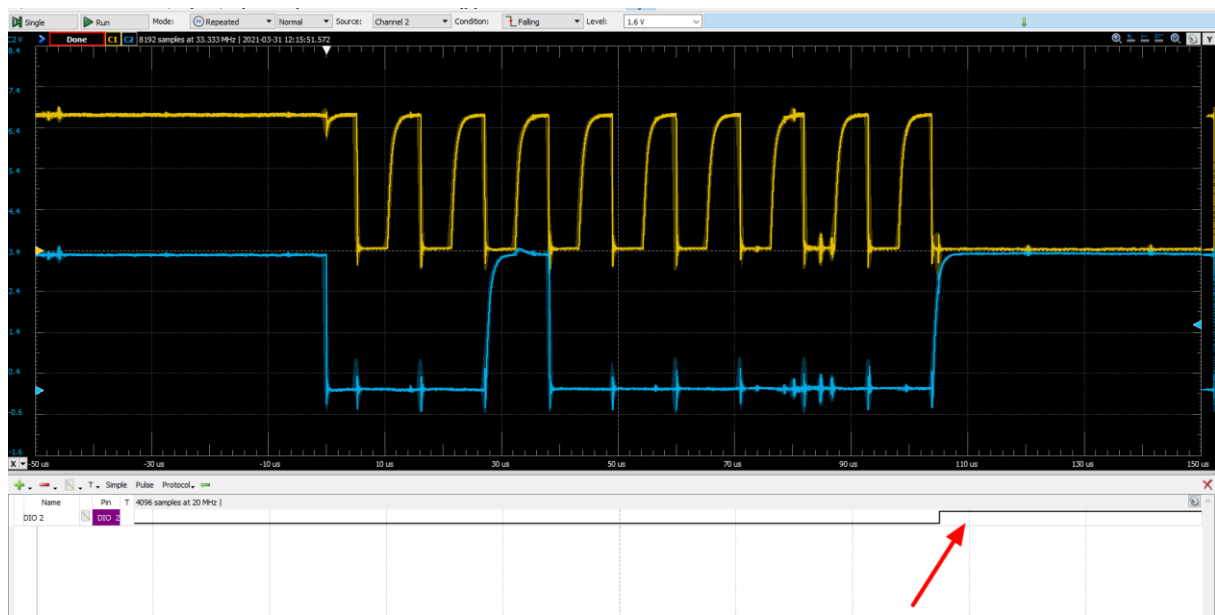
    // Adresse du composant VEML7700 (SLAVE) est 0x10
    I2C1->CR2 &= ~I2C_CR2_SADD_Msk;
    I2C1->CR2 |= ((0x10 <<1U) <<I2C_CR2_SADD_Pos);

    // Transfert avec NBYTES=1, pas de AUTOEND
    I2C1->CR2 &= ~I2C_CR2_NBYTES;
    I2C1->CR2 |= (1 <<16U);
    I2C1->CR2 &= ~I2C_CR2_AUTOEND;

    // Demarrer la transaction I2C
    I2C1->CR2 |= I2C_CR2_START; // <-- Mettre un Point arret ici
    wait_for_flags();

    while(1)
    {
    }
}
```

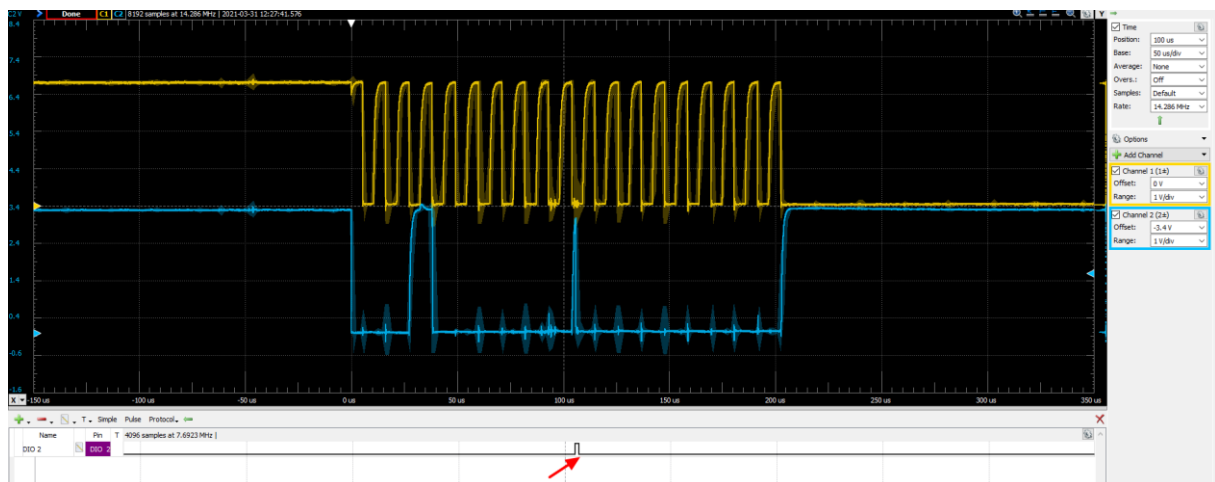

Encore une fois, enregistrez le début de la transaction :



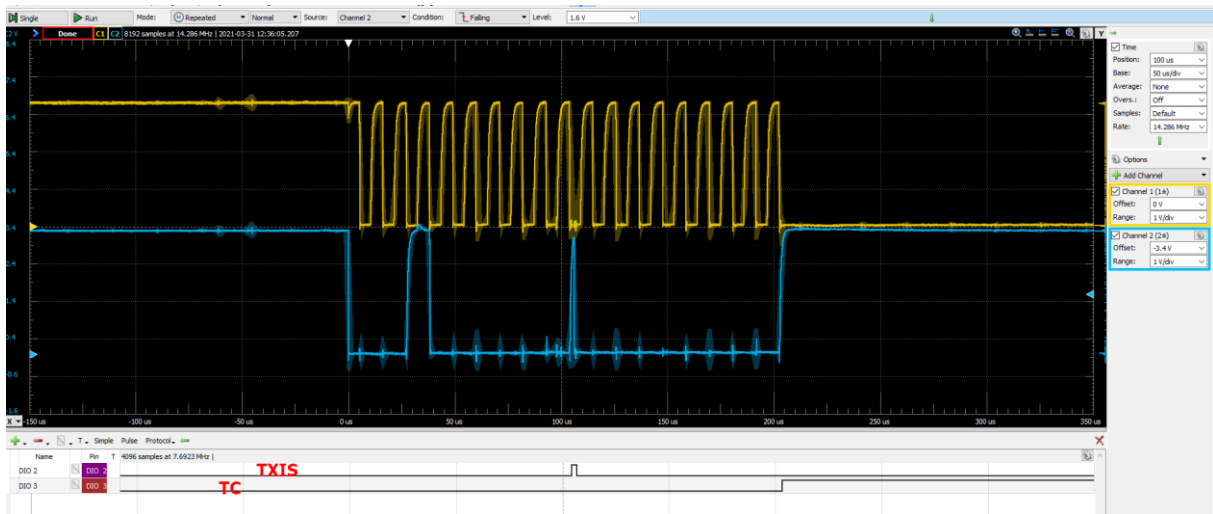
Cette fois, TXIS est activé après l'envoi de l'adresse et l'accusé de réception. C'est l'événement dont nous avons besoin pour passer à l'étape suivante, l'envoi du 'Command code' 0x00 :

Ajoutons donc à notre fonction main, l'envoi de la commande :

Capturez la transaction.



Ajouter un test pour le drapeau TC (Transfer Complete) dans la fonction **wait_for_flag()**, avec une **surveillance en direct sur PA1**, et une sortie lorsqu'il est activé. Connectez l'entrée sortie numérique N° 3 du boîtier analog discovery à la patte PA1, puis sondez SCL, SDA, PA0 et PA1 avec le boîtier analog discovery. Vous devriez obtenir ceci :



Nous avons réussi à faire fonctionner 2 phases :

- La condition START, suivie d'une adresse de périphérique acknowledged en mode WRITE.
- La mise à '1' de TXIS, indiquant qu'une donnée doit maintenant être envoyée à l'esclave. Notez que l'écriture dans le registre de données TXDR efface automatiquement TXIS.
- Après l'envoi de l'octet de commande, TXIS à l'état bas (aucune autre donnée à envoyer, car NBYTES vaut 1). TC passe à '1' pour signaler que le transfert en cours est maintenant terminé.

De retour sur le protocole de lecture, nous devons générer à nouveau une condition START et envoyer à nouveau l'adresse du périphérique.

Puisque nous entrons maintenant dans la partie lecture de la transaction, surveillons un autre drapeau : **RXNE** (RX Not Empty), qui signifie qu'une **donnée a été reçue**. Ajoutez un nouveau test à la fonction **wait_for_flag()** avec un moniteur **RXNE sur PA1** et une sortie sur état haut. Vous devez conserver les tests sur TXIS et TC pour la sortie, mais il faut supprimer la partie surveillance.

```
// RXNE-> PA1
// Quitter attente si RXNE passe a '1'
if ((I2C1->ISR & I2C_ISR_RXNE) != 0)
{
    GPIOA->BSRR = GPIO_BSRR_BS_1; // mise a '1' de PA1
    exit = 1;
}
else GPIOA->BSRR = GPIO_BSRR_BR_1; // mise a '0' de PA1
```

Puis modifiez **main()** afin de procéder au redémarrage du bus I2C en mode READ :

```
int main(void)
{
    // Configurer horloge du systeme pour 48MHz a partir de la source
    8MHz HSE
    SystemClock_Config();

    // Initialiser la LED et le bouton bleu
    BSP_LED_Init();
    BSP_PB_Init();
    // Initialiser les pins de debug
    BSP_DBG_Pins_Init();

    // Initialiser la console de debogage
    BSP_Console_Init();
    mon_printf("\r\n La est Console Ready!\r\n");
    mon_printf("SYSCLK = %d Hz\r\n", SystemCoreClock);

    // Initialiser le périphérique I2C1
    BSP_I2C1_Init();

    // Adresse du composant VEML7700 (SLAVE) est 0x10
    I2C1->CR2 &= ~I2C_CR2_SADD_Msk;
    I2C1->CR2 |= ((0x10 <<1U) <<I2C_CR2_SADD_Pos);

    // Transfert avec NBYTES=1, pas de AUTOEND
    I2C1->CR2 &= ~I2C_CR2_NBYTES;
    I2C1->CR2 |= (1 <<16U);
    I2C1->CR2 &= ~I2C_CR2_AUTOEND;

    // Demarrer la transaction I2C
    I2C1->CR2 |= I2C_CR2_START; // <-- Mettre un Point arret ici
    wait_for_flags();           // attente de TXIS

    //TXIS a été activé, passez à l'envoi du code de commande.
    I2C1->TXDR = 0x00;
    wait_for_flags();           // attente de TC

    // Mettre l'I2C en mode lecture
    I2C1->CR2 |= I2C_CR2_RD_WRN;

    // Relancer la transaction
    I2C1->CR2 |= I2C_CR2_START;

    wait_for_flags();           // attente RXNE

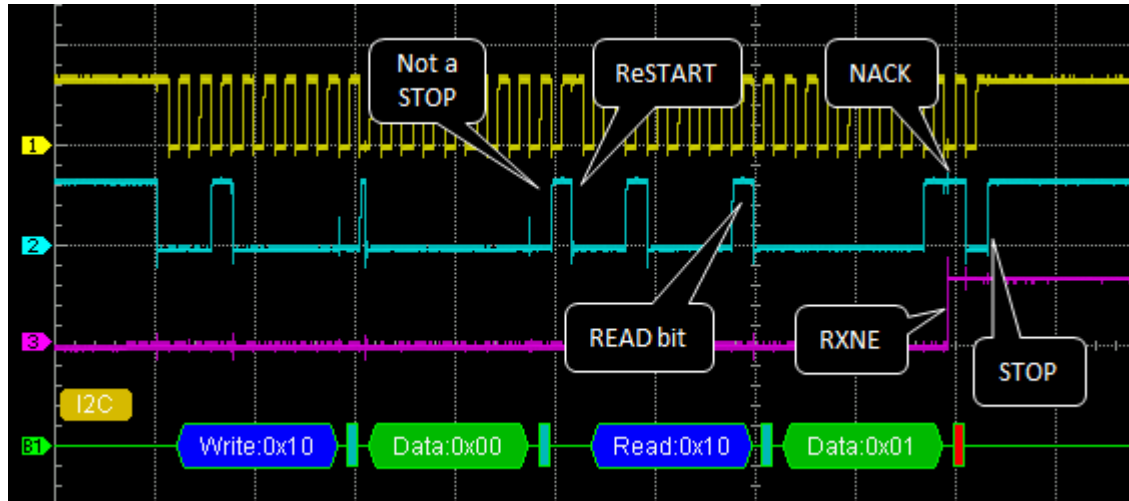
    // Generer une condition STOP
    I2C1->CR2 |= I2C_CR2_STOP;

    while(1)
    {

    }
}
```

La génération de la condition STOP à la fin n'est là que pour clore la transaction proprement. Ne pas le faire peut laisser le VEML7700 dans un état inachevé dont vous ne pouvez pas facilement vous remettre.

Capturez la transaction et le drapeau RXNE sur PA1. Voici ce que vous devez attendre :



Il y a plusieurs choses à commenter ici :

- Rappelez-vous que la condition d'arrêt est obtenue par une montée de SDA alors que SCL est à l'état haut. En regardant ce qui se passe juste après le deuxième accusé de réception, vous pouvez voir que SDA monte alors que SCL est bas, ne créant donc pas de condition d'arrêt avant le redémarrage (SDA descend alors que SCL est haut). C'est ce dont nous avons besoin, et la raison pour laquelle nous avons défini AUTOEND=0 auparavant. Avec AUTOEND=1, un STOP se produirait ici, interrompant tout le processus de lecture.
- Après le redémarrage, le maître envoie à nouveau l'adresse du dispositif (0x10), mais avec le bit R/W à 1, et l'esclave accuse réception. A partir de ce moment, SDA est sous le contrôle de l'esclave. SCL est toujours piloté par le maître.
- Comme le maître joue maintenant un rôle de récepteur, il produit toujours des cycles SCL et lit l'octet de l'esclave (0x01 ici). A la fin **du 8ème cycle SCL**, RXNE s'élève, indiquant qu'une donnée a été reçue.
- Au 9ème cycle SCL, le maître n'a pas accusé réception. Un signal de non-accusé de réception (NACK) informe ici l'esclave que le maître a fini de lire les octets. L'esclave relâche alors le contrôle SDA au maître.
- La condition STOP est finalement générée par le maître

La transaction ci-dessus est totalement valide. La seule chose est que le **VEML7700** a des **registres de 16 bits**, donc une action de lecture doit être préparée pour accueillir **2 octets au lieu d'un seul**... C'est là que le **NBYTES** entre en jeu, encore une fois.

Modifiez le code du fichier **main.c** avec le code ci-dessous :

```
static void SystemClock_Config(void);
static uint8_t wait_for_flags();

int main(void)
{
    // Configurer horloge du systeme pour 48MHz a partir de la source
    8MHz HSE
    SystemClock_Config();

    // Initialiser la LED et le bouton bleu
    BSP_LED_Init();
    BSP_PB_Init();
    // Initialiser les pins de debug
    BSP_DBG_Pins_Init();
    // Initialiser la console de debogage
    BSP_Console_Init();
    mon_printf("\r\n La est Console Ready!\r\n");
    mon_printf("SYSCLK = %d Hz\r\n", SystemCoreClock);
    // Initialiser le périphérique I2C1
    BSP_I2C1_Init();
    // Adresse du composant VEML7700 (SLAVE) est 0x10
    I2C1->CR2 &= ~I2C_CR2_SADD_Msk;
    I2C1->CR2 |= ((0x10 <<1U) <<I2C_CR2_SADD_Pos);

    // Transfert avec NBYTES=1, pas de AUTOEND
    I2C1->CR2 &= ~I2C_CR2_NBYTES;
    I2C1->CR2 |= (1 <<16U);
    I2C1->CR2 &= ~I2C_CR2_AUTOEND;

    // Demarrer la transaction I2C
    I2C1->CR2 |= I2C_CR2_START; // <-- Mettre un Point arret ici
    wait_for_flags();           // attente de TXIS

    //TXIS a été activé, passez à l'envoi du code de commande.
    I2C1->TXDR = 0x00;
    wait_for_flags();           // attente de TC

    // Mettre l'I2C en mode lecture
    I2C1->CR2 |= I2C_CR2_RD_WRN;
    // Relancer la transaction
    I2C1->CR2 |= I2C_CR2_START;
    wait_for_flags();           // attente RXNE

    // Lecture du premier octet (LSB)
    rx_data = I2C1->RXDR;
    wait_for_flags();           // attente RXNE

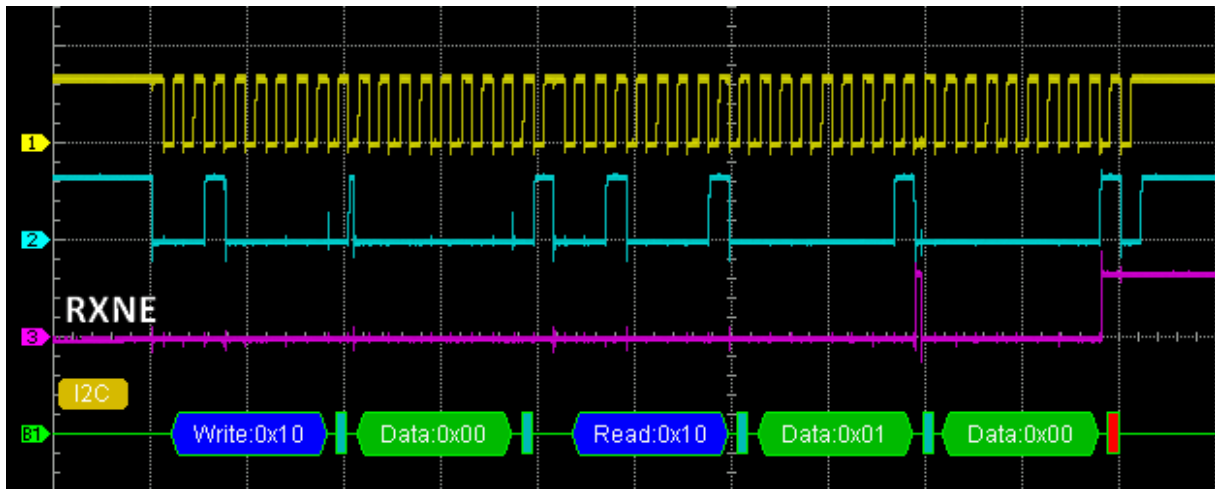
    // Lecture du deuxième octet (MSB)
    rx_data += (I2C1->RXDR <<8U);

    // Generer une condition STOP
    I2C1->CR2 |= I2C_CR2_STOP;

    while(1)
    {

    }
}
```

Pour faire fonctionner ce qui précède, vous devez déclarer **rx_data** comme un **uint16_t**. Et enfin, nous avons réussi à faire fonctionner la transaction de lecture complète, avec la valeur du registre à 0x00 étant 0x0001 :



Quelques commentaires :

- Parce que NBYTES est maintenant 2 pendant la phase de lecture, le premier octet entrant (0x01) est maintenant acquitté par le maître, disant à l'esclave "continuez, j'en veux plus !".
- Après la réception du premier octet, RXNE s'élève. Il produit une sortie de notre fonction **wait_for_flags()** suivie d'une lecture du registre RXDR.
- La lecture de RXDR efface automatiquement le drapeau RXNE. Lorsque le nombre total d'octets (2) a été lu, le NACK est envoyé, et la condition STOP est générée par le logiciel.

Au cas où vous vous poseriez la question, 0x0001 dans le registre 0x00 du VEML7700 signifie simplement que le capteur est effectivement arrêté (voir la fiche technique page 5). Effacer le bit [0], en d'autres termes, écrire la valeur 0x0000 dans ce registre, remettrait le capteur sous tension.

Avant d'explorer la transaction **WRITE**, transformons la transaction READ ci-dessus en une fonction de lecture plus compacte et générique. Ajoutez ce qui suit à votre **bsp.c/bsp.h** :

```
uint8_t BSP_I2C1_Read( uint8_t device_address,
                        uint8_t register_address,
                        uint8_t *buffer,
                        uint8_t nbytes )
{
    uint32_t timeout;    // Delai d'attente du drapeau
    uint8_t n;           // variable Compteur

    // Définir adresse du composant slave
    I2C1->CR2 &= ~I2C_CR2_SADD_Msk;
    I2C1->CR2 |= ((device_address <<1U) <<I2C_CR2_SADD_Pos);

    // Mettre le peripherique I2C en mode Write
    I2C1->CR2 &= ~I2C_CR2_RD_WRN;

    // Transfert avec NBYTES=1, pas de AUTOEND
    I2C1->CR2 &= ~I2C_CR2_NBYTES;
    I2C1->CR2 |= (1 <<16U);
    I2C1->CR2 &= ~I2C_CR2_AUTOEND;

    // Demarrer la transaction I2C
```

```

I2C1->CR2 |= I2C_CR2_START;

// Attendre que TXIS passe a '1' avec un timeout
timeout = 100000;
while (((I2C1->ISR) & I2C_ISR_TXIS) != I2C_ISR_TXIS)
{
    timeout--;
    if (timeout == 0) return 1;
}

// Envoyer Adresse du composant (SLAVE)
I2C1->TXDR = register_address;

// Attendre que TC passe a '1' avec un timeout
timeout = 100000;
while (((I2C1->ISR) & I2C_ISR_TC) != I2C_ISR_TC)
{
    timeout--;
    if (timeout == 0) return 2;
}

// Mettre le peripherique I2C en mode Write
I2C1->CR2 |= I2C_CR2_RD_WRN;

// Transfert avec NBYTES=1, pas de AUTOEND
I2C1->CR2 &= ~I2C_CR2_NBYTES;
I2C1->CR2 |= (nbytes << 16U);
I2C1->CR2 &= ~I2C_CR2_AUTOEND;

// Re-Demarrer la transaction I2C
I2C1->CR2 |= I2C_CR2_START;

// Lecture des data
n = nbytes; // nombre octet

while (n>0)
{
    // Attendre que RXNE passe a '1' avec un timeout
    timeout = 100000;
    while (((I2C1->ISR) & I2C_ISR_RXNE) != I2C_ISR_RXNE)
    {
        timeout--;
        if (timeout == 0) return 3;
    }

    // Enregister les data dans un buffer
    *buffer = I2C1->RXDR;
    buffer++;
    n--;
}

// Generere une condition STOP
I2C1->CR2 |= I2C_CR2_STOP;

// Attendre que STOPF passe a '1' avec un timeout
timeout = 100000;
while (((I2C1->ISR) & I2C_ISR_STOPF) != I2C_ISR_STOPF)
{
    timeout--;
    if (timeout == 0) return 4;
}

```

```

    // Renvoi 0 si tout est ok
    return 0;
}

```

A partir de maintenant, vous pouvez réaliser des transactions de lecture I2C facilement. Exemple de code de test.

```

static void SystemClock_Config(void);
static uint8_t wait_for_flags();
uint16_t rx_data[2];

int main(void)
{
    // Configurer horloge du systeme pour 48MHz a partir de la source
    8MHz HSE
    SystemClock_Config();

    // Initialiser la LED et le bouton bleu
    BSP_LED_Init();
    BSP_PB_Init();
    // Initialiser les pins de debug
    BSP_DBG_Pins_Init();

    // Initialiser la console de debogage
    BSP_Console_Init();
    mon_printf("\r\n La est Console Ready!\r\n");
    mon_printf("SYSCLK = %d Hz\r\n", SystemCoreClock);

    // Initialiser le périphérique I2C1
    BSP_I2C1_Init();

    //Lire 2 octets dans le registre VEML7700 @0x00
    BSP_I2C1_Read(0x10, 0x00, rx_data, 2);

    // envoi des data aux pc
    my_printf( "configuration actuelle du VEML7700 = 0x%04x\r\n",
               (uint16_t)(rx_data[1] << 8U | rx_data[0]) );

    while(1)
    {

    }
}

```

```

COM5 - PuTTY
La est Console Ready!
SYSCLK = 48000000 Hz
configuration actuelle du VEML7700 = 0x0001

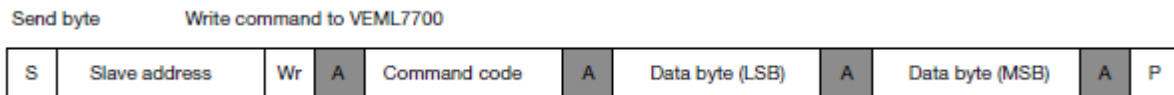
```



Sauvegarder votre code main et les captures d'écrans montrant vos essais

6. Transaction d'écriture – WRITE

La transaction WRITE telle que fournie dans la fiche technique du VEML7700 est présentée ci-dessous. Encore une fois, elle est conforme à la norme I2C :



Par rapport à la séquence READ, l'écriture est légèrement plus simple car il n'y a pas de re-START impliqué. Le maître continue à contrôler SDA pendant toute la durée du processus, en écoutant simplement les accusés de réception après le transfert de chaque octet.

Nous sommes intéressés de tester et de surveiller certains drapeaux : **TXIS**, **TC**, et **STOPF**. Modifiez la fonction **wait_for_flags()** en conséquence. Notez que vous devez effacer manuellement le drapeau **STOPF** avant d'entrer dans la boucle de test :

```
/*
 * Attendre et notifier les drapeaux etats du bus I2C
 * fonction aide pour debug I2C
 */

static uint8_t wait_for_flags(void)
{
    uint8_t exit = 0;
    // Reset des drapeaux STOPF et NACKF
    I2C1->ICR |= I2C_ICR_STOPCF;
    I2C1->ICR |= I2C_ICR_NACKCF;

    while(exit == 0)
    {
        // TXIS -> PA0
        // Quitter attente si TXIS passe a '1'
        if ((I2C1->ISR & I2C_ISR_TXIS) != 0)
        {
            GPIOA->BSRR = GPIO_BSRR_BS_0; // mise a '1' de PA0
            exit = 1;
        }
        else GPIOA->BSRR = GPIO_BSRR_BR_0; // mise a '0' de PA0

        // TC -> PA4
        // Quitter attente si TC passe a '1'
        if ((I2C1->ISR & I2C_ISR_TC) != 0)
        {
            GPIOA->BSRR = GPIO_BSRR_BS_4; // mise a '1' de PA4
            exit = 3;
        }
        else GPIOA->BSRR = GPIO_BSRR_BR_4; // mise a '0' de PA4

        // STOPF-> PC1
        // Quitter attente si STOPF passe a '1'
        if ((I2C1->ISR & I2C_ISR_STOPF) != 0)
        {
            GPIOC->BSRR = GPIO_BSRR_BS_1; // mise a '1' de PC1
            exit = 2;
        }
        else GPIOC->BSRR = GPIO_BSRR_BR_1; // mise a '0' de PC1
    }
}
```

```

        // Ajouter autres sortie de drapeaux
        // ...
    }

    return exit;
}

```

Puis exécutez la séquence ci-dessous. Vous pouvez placer ce code dans **main ()**, après l'appel à la fonction **BSP_I2C1_Read()**; que vous venez de construire. Il s'agit d'une tentative d'écriture de **0x0000** dans le registre **@0x00**, afin d'allumer le capteur.

```

// Définir l'adresse du composant VEML7700
I2C1->CR2 &= ~I2C_CR2_SADD_Msk;
I2C1->CR2 |= ((0x10 <<1U) <<I2C_CR2_SADD_Pos);

// Mettre I2C en mode Write mode
I2C1->CR2 &= ~I2C_CR2_RD_WRN;

// Transfert NBYTES = 3, pas de AUTOEND
I2C1->CR2 &= ~I2C_CR2_NBYTES;
I2C1->CR2 |= (3 <<16U);
I2C1->CR2 &= ~I2C_CR2_AUTOEND;

// Demarrer la transaction I2C
I2C1->CR2 |= I2C_CR2_START;

wait_for_flags();           // TXIS #1

// Envoyer adresse du registre
I2C1->TXDR = 0x00;

wait_for_flags();           // attente TXIS #2

// Envoyer le premier octet (LSB)
I2C1->TXDR = 0x00;

wait_for_flags();           // attente TXIS #3

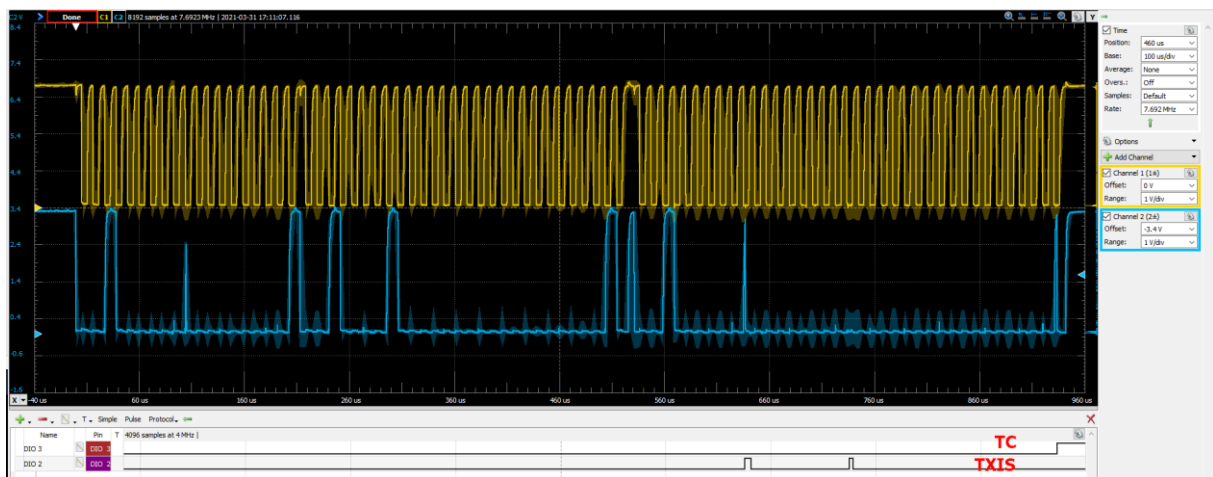
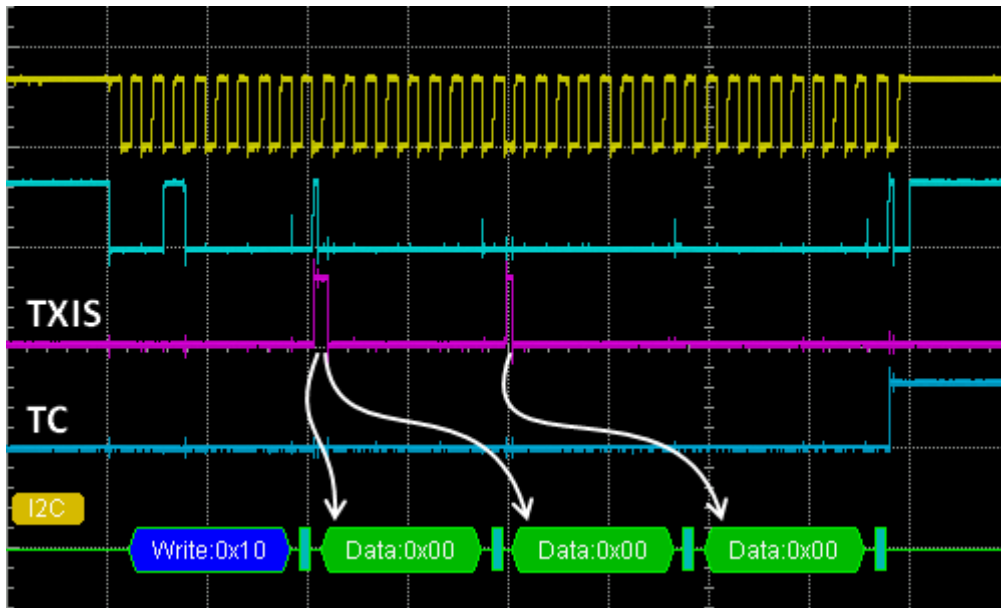
// Envoyer le deuxième octet (MSB)
I2C1->TXDR = 0x00;

wait_for_flags();           // attente de TC

// Générer une condition STOP
I2C1->CR2 |= I2C_CR2_STOP;

while(1)
{
}

```



L'oscilloscope montre que tout fonctionne comme prévu. Pourtant, nous ne pouvons voir que 2 événements TXIS. En fait, il y en a 3...

Premièrement, rappelez-vous que TXIS est activé dès que le registre TXDR est prêt à accepter un nouvel octet à transmettre.

Ensuite, lorsque vous chargez l'adresse du registre slave dans **TXDR** la première fois, ces données sont immédiatement déplacées dans le registre à décalage interne du périphérique **I2C** pour que la transmission série sur **SDA** commence. Ce processus vide le **TXDR** assez instantanément, le rendant prêt pour le prochain chargement. Vous pouvez donc recharger **TXDR** avec les premières données tout de suite. Mais à ce moment, le registre à décalage est occupé à envoyer l'adresse du registre esclave, de sorte que **TXIS** est maintenu bas jusqu'à ce que le registre à décalage soit disponible. Une fois que l'adresse de l'esclave a été envoyée sur **SDA**, elle est rechargée avec la valeur réelle de **TXDR**, qui est le premier octet de données. Ce faisant, le **TXDR** est purgé et le **TXIS** remonte.

Ce n'est qu'alors que vous pouvez charger **TXDR** avec la deuxième donnée.

En fait, vous chargez la $n^{\text{ième}}$ donnée dans **TXDR** pendant que la $(n-1)^{\text{ième}}$ donnée est transmise sur **SDA**.

Essayons quelque chose d'un peu différent, en utilisant la fonction AUTOEND du périphérique I2C :

```
// Définir l'adresse du composant VEML7700
I2C1->CR2 &= ~I2C_CR2_SADD_Msk;
I2C1->CR2 |= ((0x10 <<1U) <<I2C_CR2_SADD_Pos);

// Mettre I2C en mode Write mode
I2C1->CR2 &= ~I2C_CR2_RD_WRN;

// Transfert NBYTES = 3, avec AUTOEND
I2C1->CR2 &= ~I2C_CR2_NBYTES;
I2C1->CR2 |= (3 <<16U);
I2C1->CR2 |= I2C_CR2_AUTOEND;

// Demarrer la transaction I2C
I2C1->CR2 |= I2C_CR2_START;

wait_for_flags();           // TXIS #1

// Envoyer adresse du registre
I2C1->TXDR = 0x00;

wait_for_flags();           // attente TXIS #2

// Envoyer le premier octet (LSB)
I2C1->TXDR = 0x00;

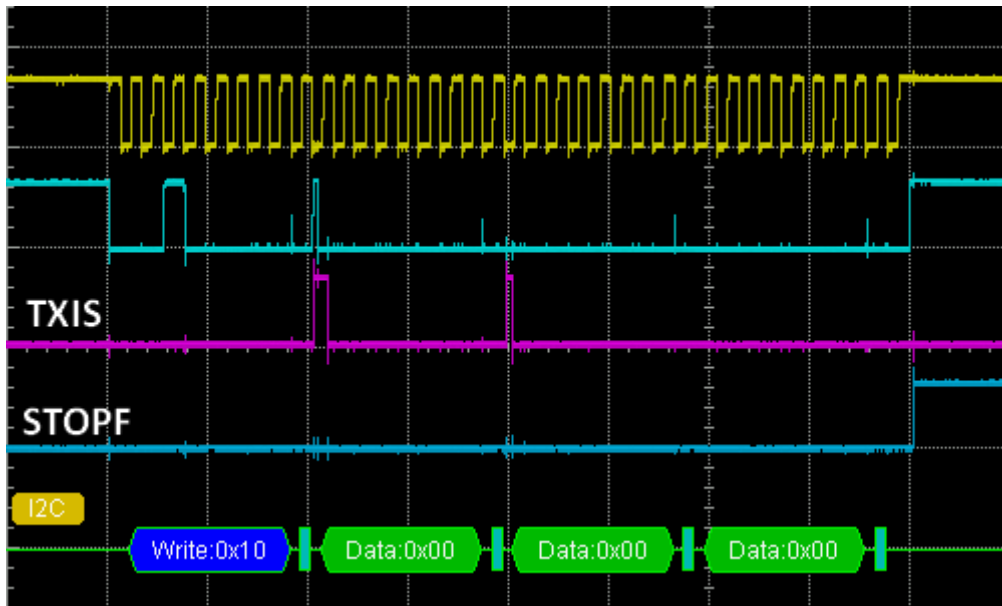
wait_for_flags();           // attente TXIS #3

// Envoyer le deuxième octet (MSB)
I2C1->TXDR = 0x00;

wait_for_flags();           // attente de TC

while(1)
{
}
```

Vous obtenez à peu près la même séquence que ci-dessus. La seule différence est que la condition END est maintenant générée automatiquement après le transfert de 3 octets. Si vous regardez le drapeau TC, rien ne se passe après la fin du transfert. Dans ce cas, l'indicateur STOPF est activé à la place :



Enfin, vous pouvez intégrer le code ci-dessus dans une fonction plus générique dans la bibliothèque périphérique bsp.c/bsp.h :

```

/*
 * BSP_I2C1_Write( )
 * Fonction ecrtiure I2C1
 * SCL -> PB8
 * SDA -> PB9
 */
uint8_t BSP_I2C1_Write( uint8_t device_address,
                        uint8_t register_address,
                        uint8_t *buffer, uint8_t nbytes )
{
    uint32_t timeout;    // Delai d'attente du drapeau
    uint8_t n;           // compteur

    // Set device address
    I2C1->CR2 &= ~I2C_CR2_SADD_Msk;
    I2C1->CR2 |= ((device_address <<1U) <<I2C_CR2_SADD_Pos);

    // Set I2C in Write mode
    I2C1->CR2 &= ~I2C_CR2_RD_WRN;

    // Transfer NBYTES, with AUTOEND
    I2C1->CR2 &= ~I2C_CR2_NBYTES;
    I2C1->CR2 |= ((nbytes+1) <<16U);
    I2C1->CR2 |= I2C_CR2_AUTOEND;

    // Clear STOPF flag
    I2C1->ICR |= I2C_ICR_STOPCF;

    // Start I2C transaction
    I2C1->CR2 |= I2C_CR2_START;

    // Wait for TXIS with timeout
    timeout = 100000;
    while (((I2C1->ISR) & I2C_ISR_TXIS) != I2C_ISR_TXIS)
    {
        timeout--;
        if (timeout == 0) return 1;
    }

    // Send register address
    I2C1->TXDR = register_address;

    n = nbytes;

    while(n>0)
    {
        // Wait for TXIS with timeout
        timeout = 100000;
        while (((I2C1->ISR) & I2C_ISR_TXIS) != I2C_ISR_TXIS)
        {
            timeout--;
            if (timeout == 0) return 2;
        }

        // Send data
        I2C1->TXDR = *buffer;
        buffer++;
        n--;
    }
}

```

```
}

// Wait for STOPF with timeout
timeout = 100000;
while (((I2C1->ISR) & I2C_ISR_STOPF) != I2C_ISR_STOPF)
{
    timeout--;
    if (timeout == 0) return 3;
}

// Return success
return 0;
}
```



Sauvegarder votre code main et les captures d'écrans montrant vos essais

7. Application simple

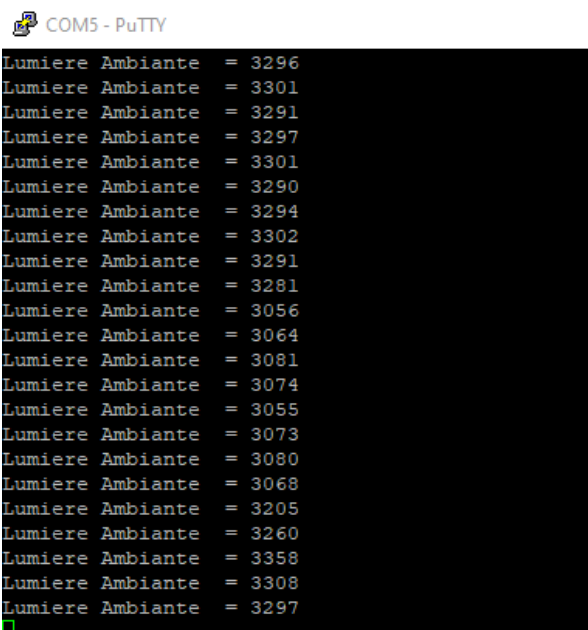
Nous disposons enfin d'un ensemble simple de fonctions pour lire et écrire sur les capteurs I2C. Les fonctions de votre en-tête bsp.h :

```
*
*  fonctions I2C1
*/

void BSP_I2C1_Init(void);
uint8_t BSP_I2C1_Read( uint8_t device_address,
                        uint8_t register_address,
                        uint8_t *buffer,
                        uint8_t nbytes );

uint8_t BSP_I2C1_Write( uint8_t device_address,
                        uint8_t register_address,
                        uint8_t *buffer, uint8_t nbytes );
```

Réaliser une application qui mesure l'intensité lumineuse en Lux et l'envoi au PC pour afficher sur la console comme ci-dessous :



```
COM5 - PuTTY
Lumiere Ambiante = 3296
Lumiere Ambiante = 3301
Lumiere Ambiante = 3291
Lumiere Ambiante = 3297
Lumiere Ambiante = 3301
Lumiere Ambiante = 3290
Lumiere Ambiante = 3294
Lumiere Ambiante = 3302
Lumiere Ambiante = 3291
Lumiere Ambiante = 3281
Lumiere Ambiante = 3056
Lumiere Ambiante = 3064
Lumiere Ambiante = 3081
Lumiere Ambiante = 3074
Lumiere Ambiante = 3055
Lumiere Ambiante = 3073
Lumiere Ambiante = 3080
Lumiere Ambiante = 3068
Lumiere Ambiante = 3205
Lumiere Ambiante = 3260
Lumiere Ambiante = 3358
Lumiere Ambiante = 3308
Lumiere Ambiante = 3297
```

La valeur de l'intensité lumineuse est dans le registre @l'adresse 0x04.



Sauvegarder votre code main et les captures d'écrans montrant vos essais