

# BUS DE COMMUNICATION

## Support de Travaux Pratiques



## Contacts

**Dimitri Boudier** – Responsable du module – TP

[dimitri.boudier@ensicaen.fr](mailto:dimitri.boudier@ensicaen.fr)

## Ressources

Toutes les ressources (supports CM, TP et outils) sont sur la page Moodle du cours :

<https://foad.ensicaen.fr/course/view.php?id=213>



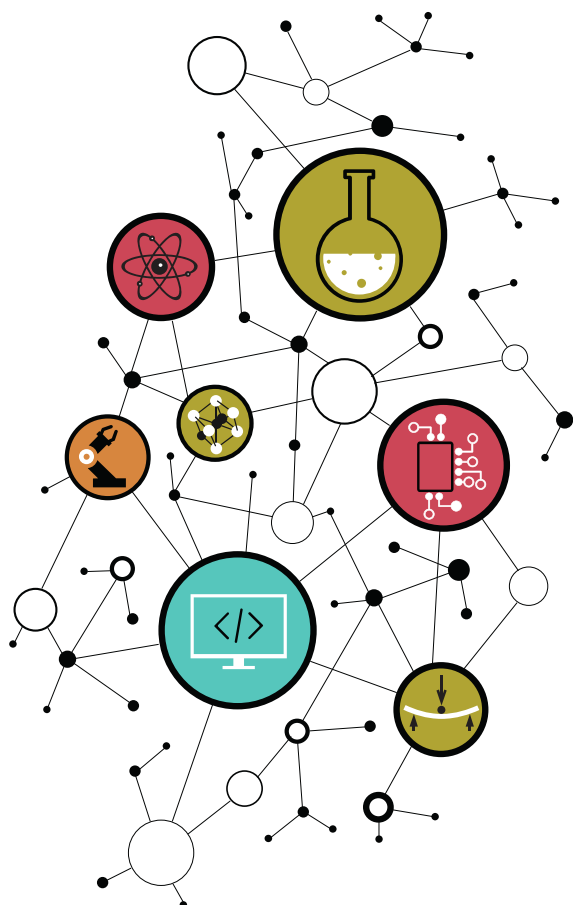
Except where otherwise noted, this work is licensed under <https://creativecommons.org/licenses/by-nc-sa/4.0/>

Contacts.....	2
Ressources.....	2
<b>Partie 1 Prélude.....</b>	<b>5</b>
I. Mise en contexte.....	6
I.1. Capteurs et systèmes connectés.....	6
I.2. Organisation du travail.....	7
II. Besoins en matériels et logiciels.....	8
II.1. En bref.....	8
II.2. Présentation du matériel.....	9
III. Prise en main de l'environnement.....	12
III.1. Création du projet.....	12
III.2. Étude du projet généré.....	13
III.3. Manipuler les GPIOs avec la HAL.....	14
III.4. Utiliser les interruptions.....	16
<b>Partie 2 Liaison Série Asynchrone / UART.....</b>	<b>21</b>
I. Présentation.....	22
II. Spécifications.....	23
III. Implémentation sur STM32.....	24
III.1. Création du projet.....	24
III.2. Étude du projet généré.....	25
III.3. UART en transmission.....	26
III.4. UART en réception.....	29
III.5. Bridge UART (pour les plus avancés).....	32
IV. Synthèse.....	35
<b>Partie 3 Bus SPI.....</b>	<b>37</b>
I. Spécifications.....	38
II. Constitution d'une trame.....	39
III. Application : Light click.....	40
III.1. Étude du capteur.....	40
III.2. Configuration du périphérique SPI.....	41
III.3. Driver light_click.....	42
IV. Application : Accel 2 click.....	47
IV.1. Étude du capteur.....	47
IV.2. Configuration du périphérique SPI.....	48
IV.3. Driver accel_2_click.....	49
V. Synthèse.....	57
<b>Partie 4 Bus I²C.....</b>	<b>59</b>
I. Description.....	60
II. Application : Weather click.....	61
II.1. Étude du capteur.....	61
II.2. Configuration du périphérique I²C.....	62
II.3. Driver weather_click.....	62
II.4. Synthèse.....	73
<b>Partie 5 Glossaire.....</b>	<b>75</b>



### PARTIE 1

# PRÉLUDE



## I. Mise en contexte

Ce document et les ressources associées ont été rédigés dans le cadre de l'enseignement « Capteurs et Systèmes Connectés » dispensé en 3A GPSE-IPC.

Toutefois cette trame a également été pensée pour être utilisée en dehors du cadre des TP, notamment pour celles et ceux qui souhaitent approfondir leur compréhension des systèmes embarqués (coucou les SATE et les FISA). Les informations sur le matériel nécessaire sont données par la suite (II. Besoins en matériels et logiciels page 8), avec une partie du matériel disponible en prêt et/ou libre service directement en salle A203.

### I.1. Capteurs et systèmes connectés

Ce sujet de TP est la première étape de l'enseignement « Capteurs et Systèmes Connectés ». Celui-ci a pour but d'effectuer une introduction aux capteurs communicants d'une part et aux systèmes connectés d'autre part.

Le terme « capteur communicant » désigne dans ce document tout élément de mesure qui est en capacité d'échanger des informations avec un processeur. Dans les systèmes modernes cela se fait grâce à une interface numérique, même s'il subsiste des interfaces analogiques. Ces interfaces numériques d'échange sont des **bus de communication** dont l'usage est extrêmement répandu (si ce n'est systématique) dans le domaine des systèmes embarqués.

Un bus de communication est, comme son nom l'indique, un système de communication entre composants d'un système. Quand on l'emploie dans le cadre des systèmes embarqués le bus de communication est fréquemment implémenté sur un **PCB (Printed Circuit Board, Circuit Imprimé)**, voire sur quelques PCB périphériques attachés au PCB principal du système. Le micro-contrôleur (ou MCU pour *MicroController Unit*) principal du système est alors généralement qualifié de maître (ou contrôleur) et il pilote le bus de communication. Il communique avec d'autres composants (capteurs, actionneurs, voire d'autres MCU dédiés) qui sont eux qualifiés d'esclaves (ou cibles).

Nous étudierons donc avec cet enseignement les bus de communications **UART, SPI et I<sup>2</sup>C**, qui sont de loin les plus utilisés dans les systèmes embarqués. Pour ce faire, nous devons faire communiquer un **micro-contrôleur** avec différents **capteurs**, en développant les **drivers** associés. Il sera alors nécessaire d'étudier de la documentation technique des circuits intégrés (**datasheet**) et aussi de relever et décoder des trames avec un **analyseur logique**.

## I.2. Organisation du travail

En l'état actuel, l'ensemble des ressources de l'archive se veut être suffisant pour suivre cet enseignement de manière autonome. Ainsi, vous trouverez dans l'archive téléchargée les répertoires suivants :




- **datasheets** toutes les documentations techniques des constructeurs
- **lectures** des supports de cours pour plus de détails sur les bus de communication
- **tutorials** des documents de prise en main de certains matériels et logiciels
- **workspace** l'espace de travail dans lequel les projets STM32 seront créés
  - **ensicaen\_drivers** les ébauches de fichiers pilotes des capteurs (à compléter)

Ce document permet l'étude des bus de communication au travers de trois capteurs. Pour tout élève voulant suivre ce cours sur son temps libre, il suffit de demander le matériel à l'enseignant.

Nous pouvons mettre à disposition d'autres capteurs (voir la liste dans le répertoire **datasheets/**) si vous voulez explorer d'autres horizons, ou approfondir vos expériences de terme de développement de driver. Notez que développer un driver en partant de zéro est très formateur, puisqu'il faut être capable d'extraire les informations utiles d'une *datasheet* parfois longue et complexe.

### I.2.a. Modalité d'évaluation

Pour ceux suivants l'enseignement « Capteurs et Systèmes Connectés », la note du module est fixée par une note de contrôle continu de TP, qui se fera au travers d'un compte-rendu.




Pour cela vous devez répondre aux questions  de ce document, fournir les captures d'écran (terminal série ou analyseur logique) demandées , et compléter les fichiers sources .

Les réponses peuvent être renseignée sur ce document (la version éditable est fournie) ou bien sur un document à part (papier ou numérique) réalisé par vos soins. La seule contrainte est que l'intitulé des questions y soit recopié (texte intégral de la question, idéalement dans l'ordre). Les captures d'écran doivent idéalement être directement intégrées au document de réponses (papier ou numérique).

Les fichiers sources sont à rendre à part (version numérique, ne pas oublier de renseigner les champs **@author** en début de fichier).

Des zones de dépôt apparaîtront sur l'espace Moodle de l'enseignement, en fonction de l'avancement du TP.

### I.2.b. Notation

-  Question attendant une réponse écrite (résultant d'une analyse des ressources)
-  Répondre en programmant / travaillant sur machine
-  Répondre en relevant une capture d'écran du terminal série ou d'un chronogramme (oscilloscope, analyseur logique, ...)

## II. Besoins en matériels et logiciels

### II.1. En bref

Vous trouverez sur cette page une courte liste des matériels et logiciels nécessaires pour suivre ce document. Une présentation détaillée de ces items est présente sur les pages suivantes.

Le matériel utilisé avec ce sujet de TP est le suivant :

- une carte NUCLEO-L073RZ
  - n'importe quelle carte NUCLEO (-L746RG, -WL55JC1, ...) est compatible.
- un *Arduino UNO click shield* ;
- des *MIKROE Click Board* :
  - Dans ce document : *USB UART click* ; *Light click* ; *Accel 2 click* ; *Weather click*
  - Dans les salles de TP : *Air Quality click* ; *Pollution click* ; *Barometer click*, ...
- un analyseur logique IKALogic :
  - Mais tout autre analyseur logique ou oscilloscope intégrant une fonctionnalité de décodage de trame (UART, SPI et I<sup>2</sup>C) est utilisable.

Voici les logiciels associés au matériel de la première liste :

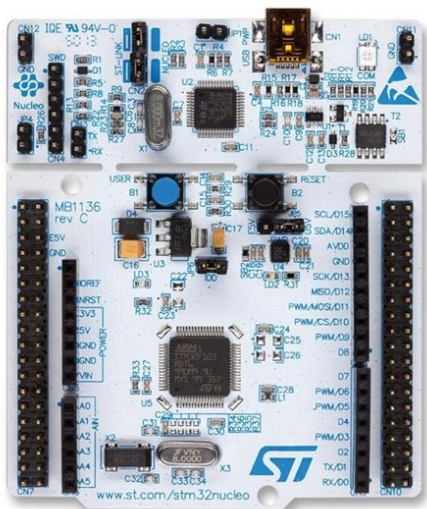
- STM32CubeIDE (validé avec version 1.11.0) :
  - IDE utilisé pour programmer et *debugger* les cartes NUCLEO
  - <https://www.st.com/en/development-tools/stm32cubeide.html>
  - n'importe quel autre IDE compatible (VS Code avec extension, Keil, ...) est accepté
- ScanaStudio
  - Logiciel associé à l'analyseur Ikalogic
  - <https://ikalogic.com/scanastudio/>
  - Si vous utilisez un autre analyseur logique ou oscilloscope, à vous d'utiliser le logiciel correspondant (s'il y a).
  - *Note : c'est le seul composant optionnel de cette liste*
- Un terminal série
  - N'importe lequel fait l'affaire
  - Tera Term (Windows), PuTTY (Windows/Linux), GTKTerm (Linux), minicom (Linux), ...

Pour rappel, un document de prise en main de ces logiciels est à disposition dans le répertoire [tutorials/](#) de l'archive de TP.

## II.2. Présentation du matériel

### II.2.a. Carte STMicroelectronics NUCLEO

L'entreprise franco-italienne STMicroelectronics est un des leaders mondiaux sur le marché du semi-conducteur et plus précisément des micro-contrôleurs 32-bit basé sur cœur ARM. Pour permettre aux développeurs de prendre en main leur matériel, ST propose différents modèles de cartes d'évaluation de leur MCU. Leur gamme principale se nomme **NUCLEO**.



À titre d'exemple, la carte **NUCLEO-L073RZ**<sup>1</sup> utilisée dans ce document contient sur son PCB le MCU cible (STM32L073RZ), une sonde de programmation et *debug*, un bouton poussoir et une LED.

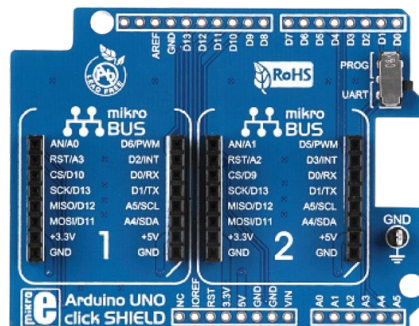
Les cartes NUCLEO sont aussi munies de connecteurs afin d'accéder aux différentes broches du MCU et de faciliter le processus de prototypage : on retrouve sur notre carte un connecteur au format Arduino et un connecteur au format Morpho (2 x 38 broches, propre à ST).

Dernière fonctionnalité intéressante, les cartes NUCLEO possèdent également une interface de communication série via leur port USB, appelée STLink VCP (*Virtual Comm Port*). Pratique pour *debugger*, sans aucun besoin de matériel supplémentaire.

Pour plus d'informations, consultez le tutoriel de l'archive de TP : [tutorials/tutoriel\\_nucleol073rz](https://www.st.com/en/evaluation-tools/nucleo-l073rz.html).

Le processeur embarqué sur la carte NUCLEO-L073RZ est le micro-contrôleur **STM32L073RZ**<sup>2</sup> de STMicroelectronics. C'est un MCU 32-bits basé sur un cœur ARM Cortex-M0+, avec pour principale caractéristique d'être ultra-basse consommation (le 'L' de la série STM32L signifie « *Low-power* »). Parmi ses nombreux périphériques, citons ici l'USART, le SPI et l'I<sup>2</sup>C : ce sont ceux avec lesquels nous travaillerons à travers ce document.

Comme décrit plus haut, les cartes NUCLEO sont toutes équipées d'un connecteur Arduino. Ainsi nous pourrions ajouter un **MIKROE Arduino UNO click shield**, qui à son tour accueillera les *Click boards* utilisés au fil de ce document. Ces *Click boards* sont présentées dans la page qui suit.



<sup>1</sup> <https://www.st.com/en/evaluation-tools/nucleo-l073rz.html>

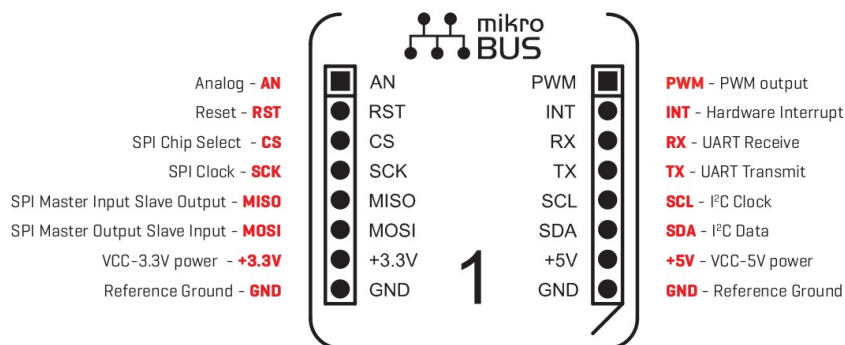
<sup>2</sup> <https://www.st.com/en/microcontrollers-microprocessors/stm32l073rz.html>

### II.2.b. MIKROE Click board™

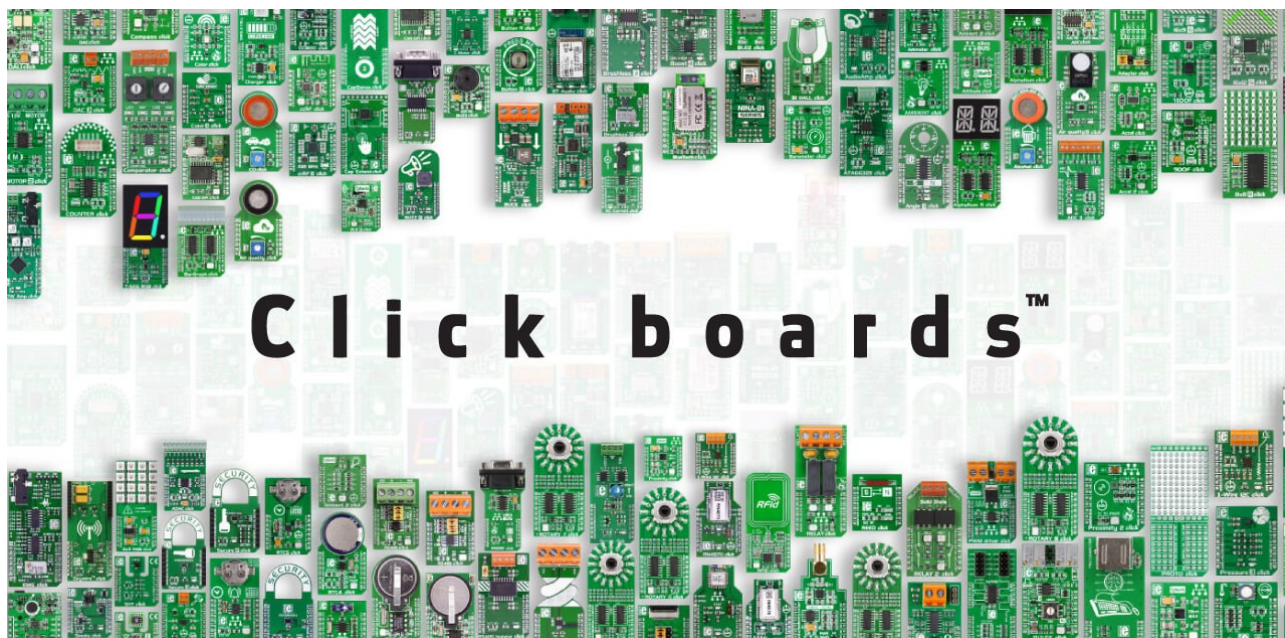


MikroElektronika<sup>3</sup> (abrégé en MIKROE) est une entreprise serbe de conception hardware et software.

La firme a notamment développé le standard ouvert mikroBUS™, dont l'objectif est de définir un format de connecteur pour lequel les broches sont imposées. De part le choix des signaux intégrés à ce format (GND, +3.3V, +5V, UART, SPI, I<sup>2</sup>C, analogique, PWM), le standard offre une grande variété d'usages possibles.



Le standard mikroBUS™ a permis à MIKROE de répandre son produit phare, la gamme *Click board™*. Ces cartes, toutes au même format, embarquent chacune différents composants (capteurs, actionneurs, alimentation, ...). En septembre 2023, plus de 1500 *Click boards* sont actifs, on retrouve également plusieurs cartes d'évaluation (comme la Microchip Curiosity HPC) qui embarquent des emplacement mikroBUS™ afin d'accueillir des *Click boards*.



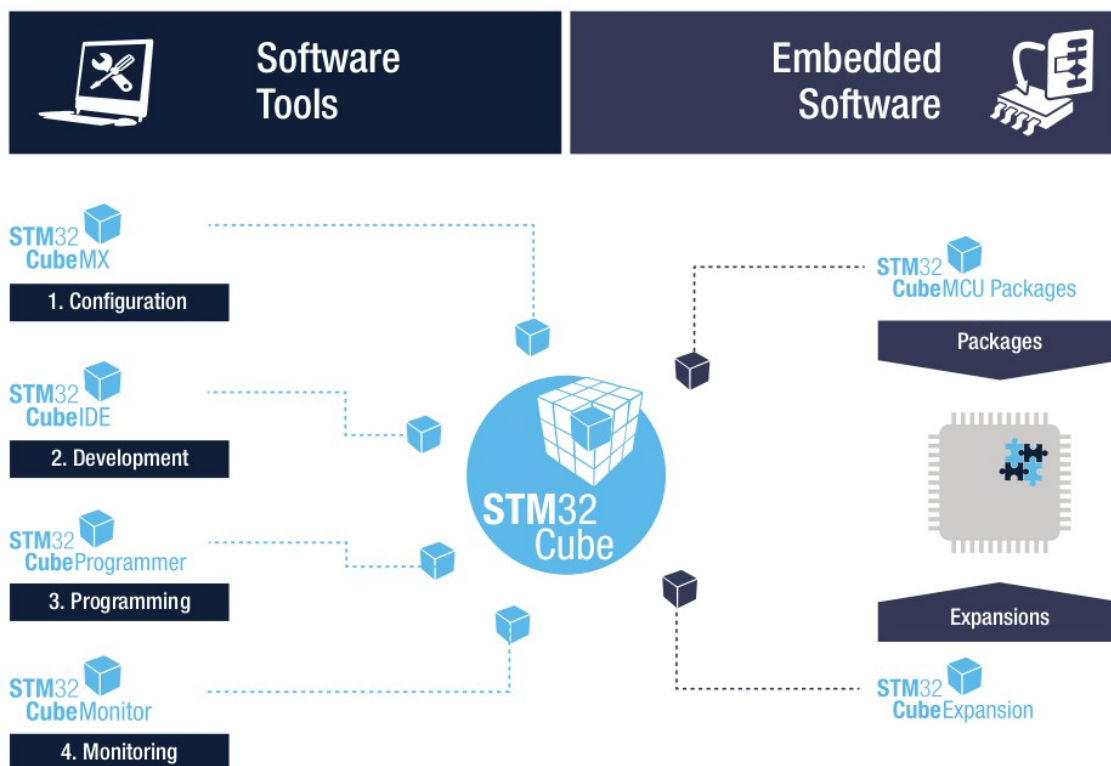
MIKROE fournit les schémas, documentations techniques des composants, ainsi que les drivers des *Click boards*. Pour chaque carte, vous pouvez donc retrouver l'ensemble des ressources sur le site du fabricant. Pour les capteurs utilisés dans ce document, les *datasheets* se trouvent dans le répertoire [connectivity/datasheets/](https://www.mikroe.com/about).

<sup>3</sup> <https://www.mikroe.com/about>

### II.2.c. Environnement de développement

Pour nos développements sur STM32, nous utiliserons la suite logicielle proposée par STMicroelectronics : **STM32Cube**<sup>4</sup>. Il s'agit d'un écosystème complet qui propose plusieurs logiciels, même si nous nous concentrerons sur les deux principaux :

- **STM32CubeIDE**, un IDE ;
- **STM32CubeMX**, un outil de configuration et de génération de code.



L'environnement de développement intégré (qu'on appellera désormais **IDE** pour *Integrated Development Environment*) s'appelle donc **STM32CubeIDE**. Il est construit à partir de l'IDE Eclipse, aka le plus gros projet d'IDE libre et multi-plateforme. Son fonctionnement s'articule autour de *perspectives* (des vues) correspondant à des usages spécifiques (par ex : *edit, debug*).

Si on décidait de partir sur de la programmation du micro-contrôleur à l'étage registre (comme en TP MCU de première année), il faudrait ajouter quelques heures à la formation tant les Cortex-M sont complexes (en comparaison à des PIC18). Nous utiliserons donc **STM32CubeMX**, qui permet de configurer graphiquement le MCU et ses périphériques pour une utilisation en quelques minutes. Dans un contexte d'entreprise, l'objectif de ce genre d'outils est de réduire le *Time-to-market* (temps de développement) des solutions logicielles embarquées. Même s'il vous a été caché, un équivalent existe chez Microchip et il s'appelle MCC (*MPLAB Code Configurator*).

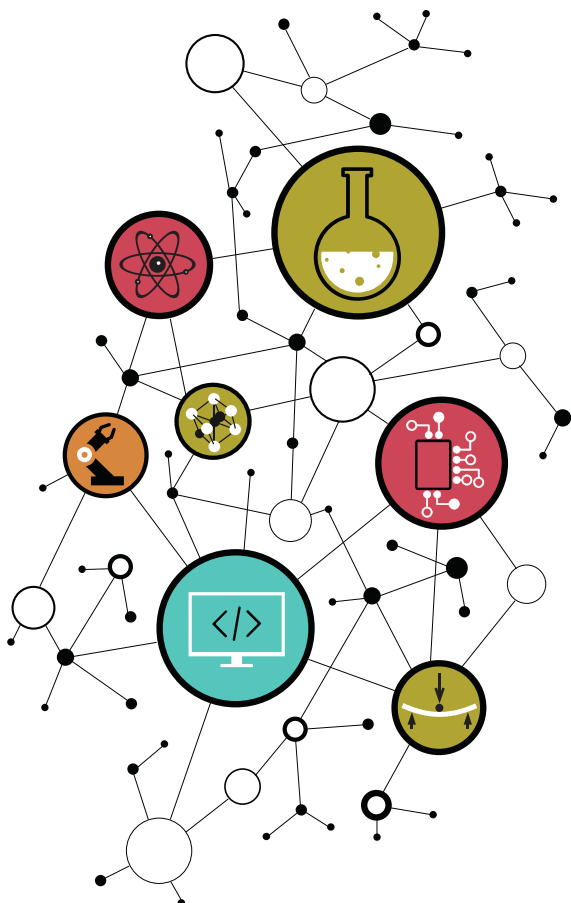
STM32Cube étant *cross-platform*, les TP peuvent être réalisés sous Windows et/ou Linux. Pour plus d'informations sur l'utilisation de ces logiciels (version à télécharger notamment), se référer à l'annexe [tutoriel\\_stm32cubeide.pdf](#).

4 <https://www.st.com/en/ecosystems/stm32cube.html>



PARTIE 2

# LIAISON SÉRIE ASYNCHRONE / UART

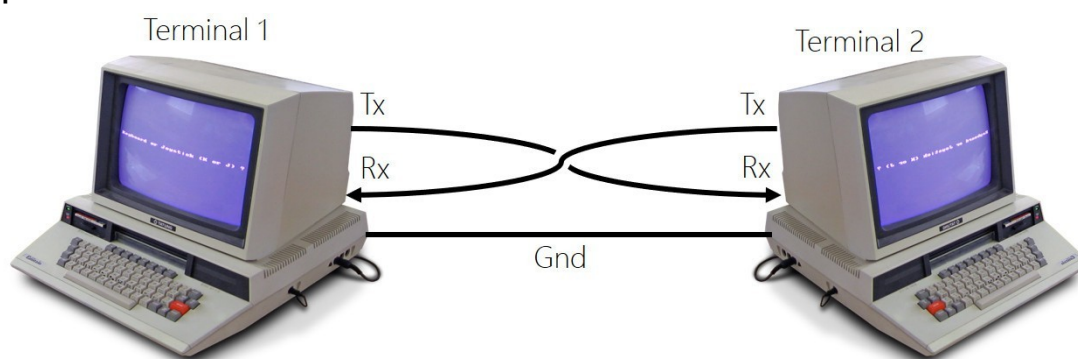


### I. Présentation

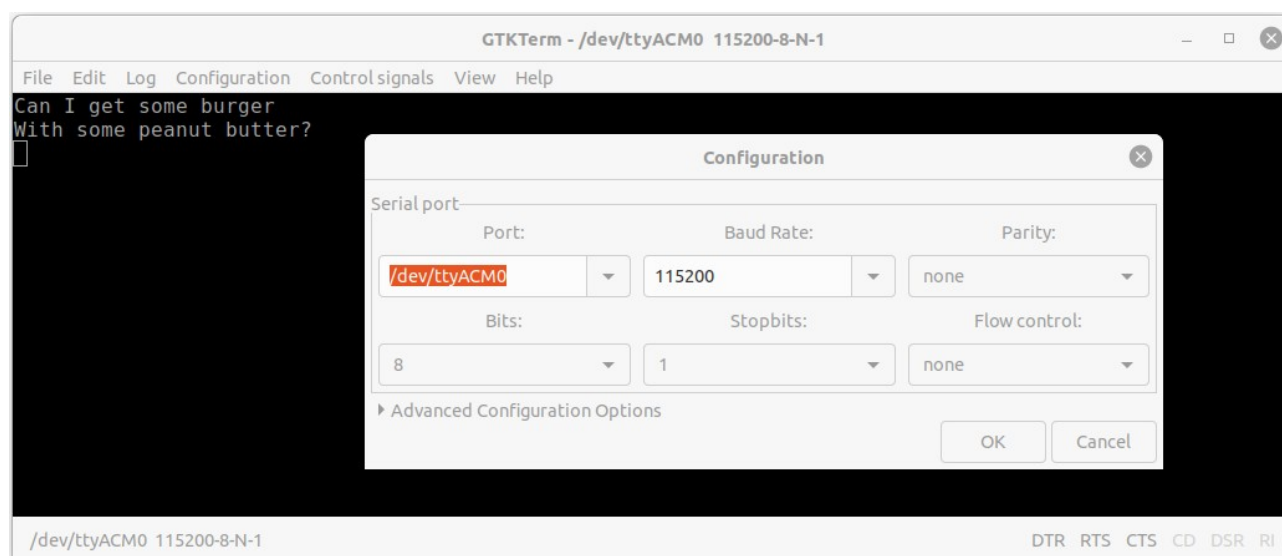
L'**UART** est un **Universal Asynchronous Receiver-Transmitter**. Il s'agit donc d'un composant ou d'un périphérique, et non d'un protocole de communication malgré les abus de langages parfois rencontrés. La communication utilisée par ces UART est une **liaison série asynchrone**.

Aujourd'hui disparu du monde des ordinateurs (l'USB apparu en 1996 l'a progressivement remplacé), l'UART existe toujours dans le monde de l'embarqué. En effet sa simplicité de fabrication et d'utilisation permettent de créer rapidement une interface de communication.

Le composant UART est constitué de deux broches : une broche de transmission appelée **Tx** et une broche de réception appelée **Rx**. Pour connecter deux composants UART, il faut évidemment croiser les câbles ( $Tx_1 \rightarrow Rx_2$  et  $Rx_1 \leftarrow Tx_2$ ). Le mode de transmission utilisé est donc un **full-duplex**.



Il est à noter que sa topologie diffère des autres protocoles usuellement rencontrés dans l'embarqué : il s'agit d'une **communication point-à-point (point-to-point)** et non d'un bus de communication. De fait l'UART est plutôt utilisé pour relier un système à un ordinateur et offrir une interface de *debug* ou configuration du-dit système depuis un terminal série sur ordinateur. Cela rejoint encore une fois l'avantage de simplicité, puisque depuis un simple logiciel (Tera Term, PuTTY, GTKTerm, ...) des données peuvent être échangées sous forme de caractères entre ces deux éléments.



## II. Spécifications

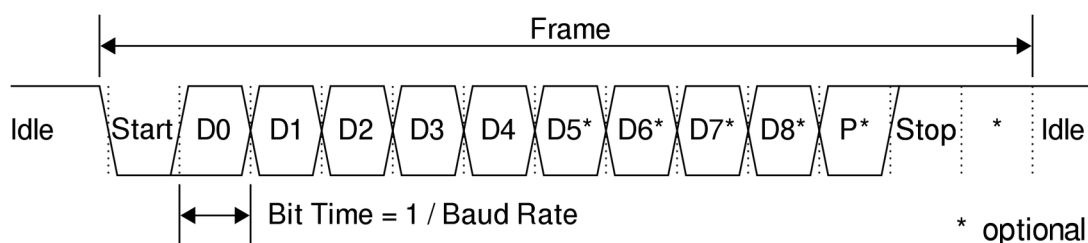
D'un point de vue matériel, un périphérique UART dispose de deux lignes :

- 1 broche **Tx** pour la transmission (reliée à la broche Rx de son homologue) ;
- 1 broche **Rx** pour la réception (reliée à la broche Tx de son homologue).

L'état de repos (*idle state*) de ces lignes est imposé par le protocole de la liaison série asynchrone. En l'absence de donnée échangée, les lignes sont au niveau haut logique. Notons que les UART de systèmes embarqués utilisent les niveaux de tension TTL, à savoir 0 V pour un niveau bas logique et +5 V pour un niveau haut logique.

Le protocole fixe également la structure des trames échangées, même si elle est assez flexible. Une trame est constituée de plusieurs champs :

- 1 bit de **start** (niveau bas logique, pour contraster avec l'état de repos) ;
- 5 à 9 bits de **données** (charge utile ou *payload*), LSB<sup>5</sup> en premier ;
- 1 bit de **parité** optionnel, servant à la détection d'erreur ;
- 1 ou 2 bits de **stop** (niveau haut logique).



La configuration la plus répandue est la 115200-8N1 :


- 115200 est le *baudrate* (les valeurs usuelles sont 9600 et 115200 baud) ;
- 8 bits de données (valeur comprise entre 5 et 9) ;
- *No parity bit* (N = *No parity bit* – O = *Odd parity bit* – E = *Even parity bit*) ;
- 1 bit de stop (1 ou 2).


D'autres standards de liaison série asynchrone existent, comme par exemple les standards RS-232 et RS-485. Il n'est pas utile de les préciser ici car ils sont davantage utilisés dans l'industrie que dans l'embarqué. Notons simplement qu'ils suivent assez fidèlement ce qui a déjà été décrit, mise à part la couche la plus basse du modèle OSI (couche Physique), spécifiant les niveaux de tension, codage, connecteur, paire différentielle, longueur des câbles, ...

5 Ce document utilise la notation **LSb/MSb** pour *Least/Most Significant bit*, et **LSB/MSB** pour *Least/Most Significant Byte*.

### III. Implémentation sur STM32



#### III.1. Création du projet


 En vous aidant de l'annexe [tutoriel\\_stm32cubeide.pdf](#) créez un projet sous STM32CubeIDE en suivant les informations ci-dessous.

 Vous aurez besoin des informations suivantes lors de la **création** du projet :

- **Board selector** : celle que vous avez en TP (voir au dos, référence NUCLEO-xxxxxx)
- **Project Name** : VOTRENOM\_uart ;
- **Project Location** : sélectionnez le répertoire `connectivity/workspace/uart/` ;
- **Initialize all peripherals with their default Mode ?** : Yes.

 Vous aurez besoin des informations suivantes lors de la **configuration** du MCU :

- Connectivity
  - → USART2
    - → Parameter Settings
      - 115 200 baud ; 8 bits de données ; pas de bit de parité ; 1 bit de stop
    - → NVIC Settings
      - USART2 Interrupt : Enable
    - → GPIO Settings
      - Relevez les broches utilisées et leur fonction dans le périphérique USART2
        -  USART2\_TX = \_\_\_\_\_
        -  USART2\_RX = \_\_\_\_\_

Une fois la configuration effectuée : **Project → Generate Code** pour générer le code correspondant aux configurations des périphériques (ou l'icône ).

Et voilà ! Les fonctions d'utilisation de l'UART sont prêtes en quelques minutes grâce au configurateur de code d'initialisation STM32CubeMX.

Pour vous donner une métrique, le *reference manual* du STM32L0x3 comporte 1040 pages, et le chapitre « USART/UART » en compte 67. Le périphérique UART utilise une douzaine de registres. À titre de comparaison, vous aviez développé en première année les drivers du périphérique EUSART pour le PIC18F27K40. La datasheet de ce MCU fait 818 pages, la partie EUSART fait 35 pages et ce périphérique utilise 6 registres. Pour une complexité apparente deux fois plus faible que pour le STM32, vous aviez passé environ 10 heures de TP au développement des drivers de ce seul périphérique pour le PIC18 ! Et le ARM Cortex-M0+ est un des plus simples MCU de ARM !

### III.2. Étude du projet généré

Le volet de gauche de l'IDE affiche le *Project Explorer*.

📄 Ouvrez le fichier `main.c` puis parcourez la fonction `main()`. On remarque l'appel à quelques fonctions d'initialisation, et une boucle `while(1)` pour l'instant vide.

📄 Maintenez la touche `Ctrl` et cliquez sur l'appel à la fonction `MX_USART2_UART_Init()`. Vous devez arriver dans le corps de la fonction, définie dans le fichier `main.c`. Tout comme la fonction d'initialisation des GPIO, celle-ci a été créée quand vous avez paramétré le périphérique USART2, dans le deuxième phase de la création de projet. Vous pouvez d'ailleurs remarquer que les paramètres que vous avez configurés apparaissent dans les premières lignes de cette fonction. Juste après ces paramètres, on remarque l'appel à une fonction de la HAL : `HAL_UART_Init()`.

📄 Effectuez un `Ctrl+Clic` sur l'appel à la fonction `HAL_UART_Init()`. Cette fois un nouveau fichier s'ouvre : `stm32l0xx_hal_uart.c`. Il s'agit d'un fichier de la HAL, contenant toutes les fonctions de configuration et d'utilisation du périphérique UART. Parmi celles-ci, vous utiliserez :


- `HAL_UART_Transmit()` : fonction d'envoi de données, bloquante ;
- `HAL_UART_Transmit_IT()` : idem, mais non-bloquante (utilise les interruptions) ;
  - Lorsque la transmission est terminée, la fonction `HAL_UART_TxCpltCallback()` est automatiquement appelée par la routine d'interruption (ISR). Pour effectuer un traitement particulier, il faut redéfinir cette *callback*.
- `HAL_UART_Receive()` : fonction de réception de données, bloquante ;
- `HAL_UART_Receive_IT()` : idem, mais non bloquante (utilise les interruptions) ;
  - Lorsque la réception est effectuée, la fonction `HAL_UART_RxCpltCallback()` est automatiquement appelée par la routine d'interruption (ISR). Pour effectuer un traitement particulier, il faut redéfinir cette *callback*.

### III.3. UART en transmission


Avec tous ces éléments en main, vous allez pouvoir transmettre un message via UART d'ici quelques minutes.




#### !/ RAPPEL !/

Vous devez écrire votre code entre deux balises de commentaires délimitées par `BEGIN` et `END` (par exemple `/* USER CODE BEGIN WHILE */` et `/* USER CODE END WHILE */`).

 Dans la boucle `while(1)` de la fonction `main()`, vous allez envoyer la chaîne de caractères `"Hi\r\n"` via le périphérique USART2.

 Quels sont les arguments à fournir à la fonction `HAL_UART_Transmit()` ?

 Envoyez cette chaîne de caractères (attention à bien écrire dans les zones balisées) et ajoutez une temporisation de 2 secondes avec la fonction `HAL_Delay()` définie dans le fichier `stm32l0xx_hal.c`.

 Compilez  et téléversez  sur la cible. S'il n'y a pas d'erreur à la compilation et au téléversement, passez à la suite pour valider le fonctionnement.

### III.3.a. STLink Virtual COM Port

La carte NUCLEO embarque certes le micro-contrôleur cible pour nos applications (le STM32WL55JC), mais elle est également équipée d'un autre MCU appelé STLink. Ce dernier est un intermédiaire entre l'ordinateur et le MCU cible, et fait principalement office de sonde de programmation/*debug*.

Or le STLink dispose d'une autre fonctionnalité : le **STLink Virtual COM PORT (VCP)**. Il est capable d'émuler d'un côté une liaison série à travers la communication USB avec un ordinateur. De l'autre côté, le périphérique USART2 du MCU cible est physiquement relié au STLink. Ainsi il est possible d'établir une communication directe entre un ordinateur et la cible en utilisant l'UART2 de cette dernière. Cela ne nécessite aucun composant supplémentaire (en comparaison au module FTDI UART/USB utilisé en 1ère année) et s'avère donc être un outil pratique, qui plus est utile pour un travail en dehors des séances.

Cependant, à cause de son lien avec le STLink VCP, l'UART2 n'est pas physiquement relié aux connecteurs Morpho (PA2/PA3) et Arduino (D0/D1) de la carte Nucleo. C'est un léger soucis que nous réglerons plus tard.

### III.3.b. Vérification par terminal série

🖥 Ouvrez le terminal série de votre choix (Teraterm, PuTTY, GTKTerm, ...). Configurez-le de sorte à échanger avec le STLink VCP (pensez aux paramètres de débit, de parité, ...).

🖨 Vous devriez voir apparaître votre texte sur le terminal. Confirmez avec une capture d'écran.



```

Welcome to minicom 2.7.1

OPTIONS: I18n
Compiled on Dec 23 2019, 02:06:26.
Port /dev/ttyACM0, 18:25:39

Press CTRL-A Z for help on special keys

Hi
Hi
[ ]

CTRL-A Z for help | 115200 8N1 | NOR | Minicom 2.7.1 | VT102 | Offline | ttyACM0
  
```

Ne passez pas à la suite tant que ceci n'est pas fonctionnel ! En effet l'UART et le terminal constituent un moyen très pratique de debug et de test d'un programme. En développement logiciel embarqué, c'est l'équivalent d'un `printf("ici");` en C.

### III.3.c. Analyse de la trame avec analyseur logique

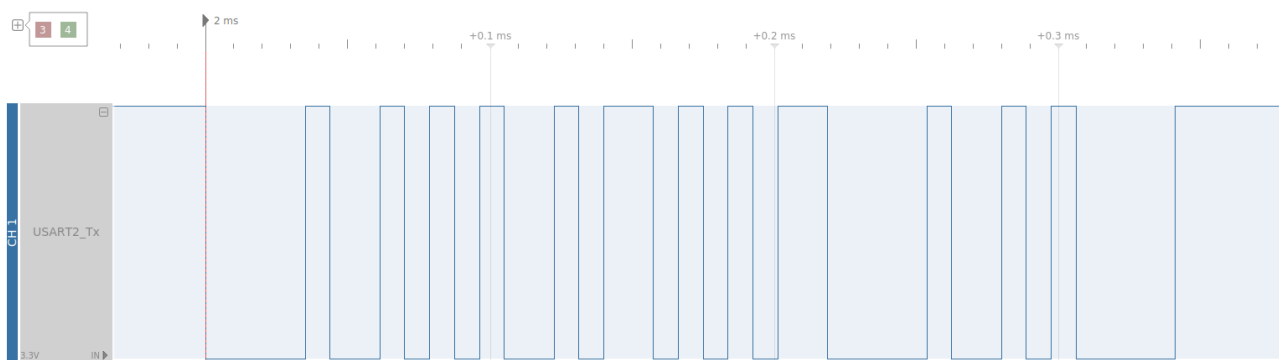
Nous allons maintenant analyser les trames à l'aide d'un analyseur logique. Pour ce faire il faudrait pouvoir relever physiquement les signaux Rx et Tx. Or nous avons vu que les broches associées au périphérique USART2 ne sont pas reliées aux connecteurs mais au MCU STLink. Il faut donc trouver un autre moyen d'accéder électriquement à ces signaux.

✎ Dans la partie III.1 Création du projet page 16, vous avez relevé les noms des broches Rx et Tx du périphérique USART2. Rappelez-les ici :

✎ Ouvrez le schéma électrique de la carte NUCLEO, disponible dans le répertoire **datasheets** du TP. Retrouvez le micro-contrôleur cible, les broches Rx et Tx de l'USART2 puis suivez les fils jusqu'à rencontrer un connecteur qui soit facilement accessible avec des sondes. Listez ci-dessous les différents labels depuis les broches du MCU cible jusqu'aux broches du STLink.

📁 Une fois que vous avez identifié le connecteur à partir duquel les signaux Rx et Tx vont pouvoir être analysés, effectuez une capture avec l'analyseur logique. Vous avez à votre disposition un document tutoriel d'utilisation de l'analyseur IKALogic dans le répertoire **tutorials**.

✎ Voici une capture de trame réalisée à l'analyseur logique. Il s'agit du signal que vous êtes censé relever. Procédez à l'analyse de la trame, bit par bit, afin de retrouver le message initial.



📁 Ce travail peut s'avérer fastidieux, surtout avec une grande quantité de données à décoder. Pour s'affranchir de cela, utilisez dans ScanaStudio l'outil de décodage de trame ainsi que la vue HexView (aidez-vous du tutoriel si besoin). Ne poursuivez que si cette étape est validée.

### III.4. UART en réception

#### III.4.a. Réception bloquante

Pour recevoir des données avec le périphérique USART2, le plus simple est d'utiliser la fonction de réception `HAL_UART_Receive()`. Lisez le fichier `stm32l0xx_hal_uart.c` dans lequel est définie cette fonction.

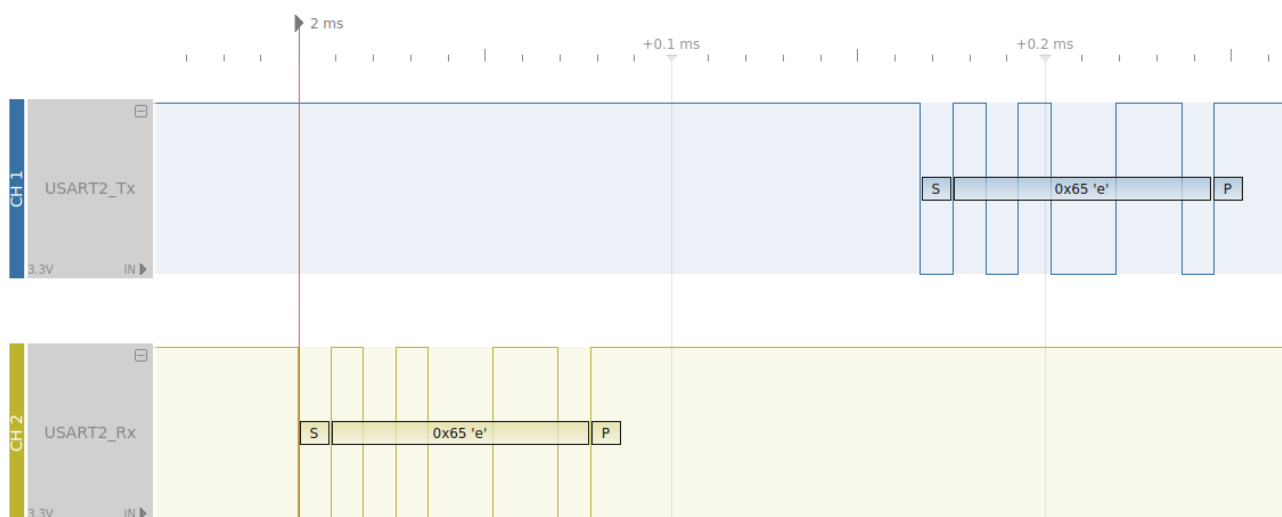
✎ Quels sont les arguments à fournir à la fonction `HAL_UART_Receive()` ?

🖥 Dans la fonction `main()`, après initialisation du périphérique UART mais avant la boucle `while(1)`, envoyez la chaîne de caractère `"\r\nApplication starting\r\n"`. Celle-ci vous permettra de vérifier que votre programme est bien lancé.

🖥 Dans la boucle `while(1)` maintenant, procédez à la réception d'un caractère puis renvoyez-le aussitôt. Il s'agit d'une fonctionnalité « écho » qui vous permettra de renvoyer au terminal série de l'ordinateur le caractère qu'il vient d'émettre.

🖥 Validez le fonctionnement avec votre terminal.

🖥 Validez le fonctionnement avec une capture à l'analyseur logique. Vous devriez obtenir quelque chose de similaire à la figure ci-dessous.



### III.4.b. Réception non-bloquante

Bien que la fonction `HAL_UART_Receive()` utilisée précédemment soit simple d'utilisation, elle utilise une scrutation active (*polling* en anglais). Autrement dit il s'agit d'une fonction bloquante : elle monopolise le CPU du processeur dans l'attente d'une réception et empêche l'exécution de toute autre portion du programme.

Pour libérer le CPU et effectuer d'autres instructions en attendant l'arrivée éventuelle d'une donnée, nous allons utiliser la fonction `HAL_UART_Receive_IT()` qui elle emploie le mécanisme d'interruption.

Le mécanisme d'interruption ayant déjà été abordé dans le document [introduction\\_aux\\_stm32.pdf](#) (retournez voir si besoin), nous nous contentons ici d'un bref rappel. Les interruptions d'un processeur ARM sont gérées par le NVIC (*Nested Vector Interrupt Controller*). Lorsqu'un périphérique détecte un évènement, il peut déclencher une requête d'interruption (IRQ, *Interrupt Request*) si le NVIC est configuré. Une routine d'interruption (ISR) se déclenche alors pour traiter l'évènement. Toutefois l'ISR est transparente pour le développeur. En effet STM32CubeMX met à disposition des fonctions de *callback*, appelées depuis les ISR, de sorte à ce que le développeur ne manipule que de simples fonctions de la HAL.

Sur STM32, la réception d'un caractère par UART suit évidemment cette logique. Mais rassurez-vous, le générateur de code STM32CubeMX a tout préparé quand vous avez demandé à utiliser les interruptions avec le périphérique USART2, lors de la création du projet.

La fonction `HAL_UART_Receive_IT()` est non-bloquante : son appel ne fait « que » configurer une interruption qui se déclenchera à la réception de caractères, puis la fonction se termine et la suite du programme s'exécute.

Quand un ou plusieurs caractères sont reçus sur l'USART désigné, alors le CPU met en pause l'exécution du code courant pour exécuter la routine d'interruption à la place<sup>6</sup>. Cette ISR (que nous ne modifierons pas) appelle la fonction de *callback* `HAL_UART_RxCpltCallback()`. Il s'agit d'une fonction que **vous devez définir** pour y insérer le traitement que vous voulez réaliser.

🔍 Observez d'ailleurs la déclaration de cette *callback* dans le fichier `stm32l0xx_hal_uart.c`. Que signifie le qualificateur `__weak` dans le prototype ?

<sup>6</sup> Fonction `UART_RxISR_8BIT()` définie dans le fichier `stm32l0xx_hal_uart.c`.

Vous allez réaliser un programme d'écho (tout caractère reçu par UART est immédiatement retransmis) en utilisant les interruptions. Mais d'abord vous mettrez en place un compteur, afin d'illustrer le travail de fond réalisé par le MCU (le comptage) et le travail effectué de manière occasionnelle (traitement par interruption d'un caractère reçu).

1. Envoyez un message sur l'UART uniquement au démarrage de l'application
2. Envoyez un message sur l'UART chaque seconde.
  - Ce message est la valeur d'un compteur qui s'incrémente à chaque secondes
  - Utilisez la fonction `sprintf()` pour créer une chaîne de caractère à partir d'un nombre (ou de n'importe quelle variable) ;
  - Transmettez cette chaîne de caractère avec l'UART ;
3. Confirmez que les étapes 1 et 2 fonctionnent avant d'aller plus loin.
4. Avant le `while(1)`, appelez la fonction `HAL_UART_Receive_IT()` pour activer l'interruption pour la réception d'un caractère ;
  - Vous devrez définir un caractère en variable globale (pour l'utiliser avec la callback).
5. Entre les balises `/* USER CODE BEGIN 0 */` et `/* USER CODE BEGIN 0 */`, définissez la callback `HAL_UART_RxCpltCallback()`.
  - Elle doit immédiatement renvoyer le caractère reçu sur l'UART
  - Elle doit ensuite relancer une nouvelle lecture par interruption, afin de traiter les futurs caractères reçus.
6. Compilez, téléversez, observez le résultat.

Validez avec une capture d'écran de votre terminal série.



```

8

Application starting...
0
1
2
hello 3
4

Application starting...
0
1
again 2
3
bye 4
5
6
7
  
```

*Exemple de résultat visible sur terminal série.*



### III.5. Bridge UART (pour les plus avancés)

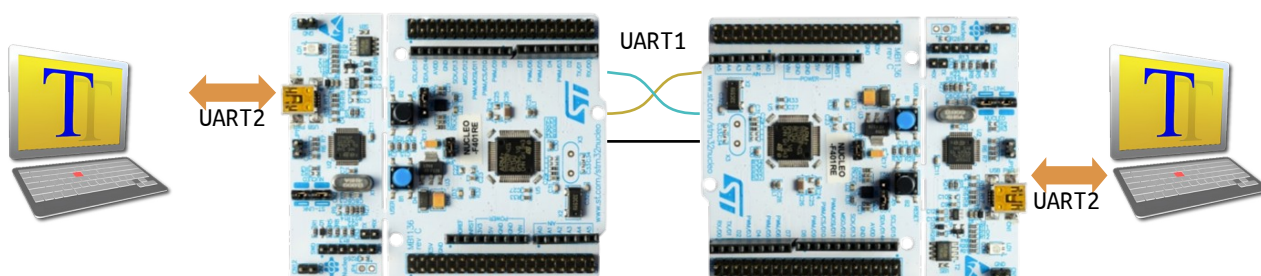
Vous allez dans cette partie créer un bridge UART. Dans le langage des bus de communication, un bridge est une interface qui ne fait que retransmettre un message venant d'une interface vers une autre, avec généralement une conversion de protocole entre les interfaces. Autrement dit, cela change la forme du message sans en changer le contenu.

Dans le cadre de cet exercice, aucune conversion de protocole ne sera effectuée. L'application bridge redirigera simplement le messages de l'UART2 vers l'UART1 d'un même MCU, et inversement.

#### III.5.a. Matériel

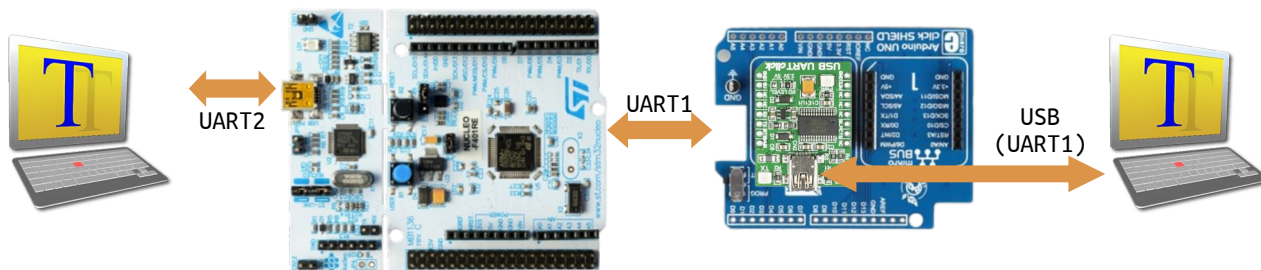
D'un point de vue matériel, vous avez le choix entre deux solutions.

La première solution nécessite plus de matériel, mais permet de mieux comprendre les connexions. En complément à votre NUCLEO, il faudra utiliser une deuxième NUCLEO (identique ou non) avec exactement les mêmes configurations de périphériques. Il faudra alors relier entre eux les périphériques UART1 de chaque NUCLEO (ne pas oublier la masse). Les deux NUCLEO sont reliées à un ou deux ordinateurs (mais bien deux terminaux série) via leur STLink VCP (UART2).



La deuxième solution est plus compacte puisqu'une seule NUCLEO suffit. Il faut néanmoins ajouter un autre convertisseur USB-UART afin de voir les messages de l'UART1. Pour cela vous pouvez réutiliser le USB UART click (vu en Systèmes Embarqués), en l'apposant sur un Arduino UNO click shield. Ainsi la NUCLEO sera reliée à un terminal série via son STLink VCP (UART2) et à un autre terminal série via le USB UART click (UART1).

Cependant les signaux Tx et Rx du périphérique USART1 ne sont pas directement routés sur les Click Board du shield. Il faut donc relier physiquement, à l'aide de fils, les broches de l'USART1 côté NUCLEO avec les broches de l'UART côté shield. Un peu de recherche avec le schéma électrique est nécessaire pour y parvenir ...





### III.5.b. Firmware

📁 Pour cette partie, créez un nouveau projet avec les propriétés suivantes :

- **Board selector** : celle que vous avez en TP (voir au dos, référence NUCLEO-xxxxxxx)
- **Project Name** : VOTRENOM\_uart\_bridge ;
- **Project Location** : sélectionnez le répertoire `connectivity/workspace/uart_bridge/` ;
- **Initialize all peripherals with their default Mode ?** : Yes.

📁 Vous aurez besoin des informations suivantes lors de la **configuration** du MCU :

- Connectivity → USART2 **et** USART1
  - → Parameter Settings
    - Baud Rate = 115 200 baud
    - Bits de données = 8 bits
    - Bit de parité : Non
    - Bit de stop : 1 bit
  - → NVIC Settings
    - USART1/2 Interrupt : Enable (ou pas ! Cf cahier des charges ci-dessous)
  - → GPIO Settings
    - Relevez les broches utilisées et leur fonction dans le périphérique UART1
      - ✂ USART1\_TX = \_\_\_\_\_ USART2\_TX = \_\_\_\_\_
      - ✂ USART1\_RX = \_\_\_\_\_ USART2\_RX = \_\_\_\_\_

📁 Une fois la configuration effectuée : **Project → Generate Code** pour générer le code correspondant aux configurations des périphériques (ou l'icône 🏠).

Le cahier des charges de votre application est en apparence assez simple :

- tout caractère reçu sur l'UART1 doit être renvoyé sur l'UART2 (bridge)
- tout caractère reçu sur l'UART2 doit être renvoyé sur l'UART1 (bridge)
- tout caractère reçu sur l'UART2 doit être renvoyé sur l'UART2 (echo)
- uniquement si vous utilisez une NUCLEO et un USB UART click :
  - tout caractère reçu sur l'UART1 doit être renvoyé sur l'UART1 (echo)



🖥 En jouant sur avec les différentes fonctions de réception (réception bloquante par *polling* ou réception non-bloquante par interruption), répondez à ce cahier des charges.

🔧 Testez en lançant deux terminaux série et après avoir préparé le matériel.

✍ Tracez un diagramme de séquence illustrant les données échangées, en partant d'un message issu d'un des deux terminaux.



### IV. Synthèse

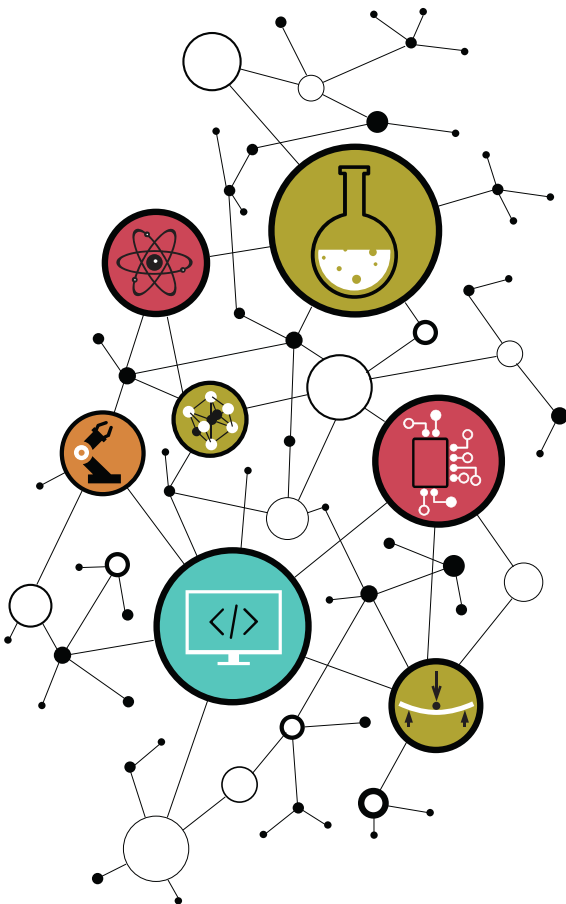
✎ En reprenant les différents points de cette partie, effectuez ici une synthèse de ce qui a été vu. Cette synthèse pourra être consultée lors des futurs développements.

1. Concernant la couche physique (fils, signaux, tensions, niveau logiques, ...)
2. Concernant la couche protocole (nombre de bits, codage, start, stop, parité, repos, ...)
3. Concernant la couche firmware (configuration du périphérique, fonctions utilisées)
4. Concernant la couche applicative (terminal série)



## PARTIE 3

# BUS SPI



### I. Spécifications

Le **SPI (Serial Peripheral Interface)** est un bus de communication série initialement proposé par Motorola au début des années 1980. Cependant il ne s'agit ni d'un standard ni d'une norme, et il existe donc de nombreuses variations autour du SPI. Toutefois nous aborderons à travers ce document la version Motorola, étant *de facto* la « version » la plus répandue.

Le SPI est un bus de communication **série**, **synchrone** et **full-duplex**. Il se base sur une topologie **maître-esclaves**, le maître étant à l'origine de toute communication, les esclaves ne pouvant que répondre à une demande du premier.

La terminologie maître-esclave est considérée comme inacceptable par certains acteurs du marché, dû à sa connotation Historique<sup>7</sup>. Aussi ces acteurs ont décidé de changer la terminologie du protocole SPI, en supprimant les termes *Master* et *Slave*. Cependant, le SPI n'est pas un standard géré par un organisme dédié, et les acteurs du marché font évoluer ces termes avec leur propre terminologie. On retrouve par exemple des *Main*, *Controller*, ... pour désigner *Master* et des *Sub*, *Peripheral*, *Chip*, *Target*, ... pour désigner *Slave*.

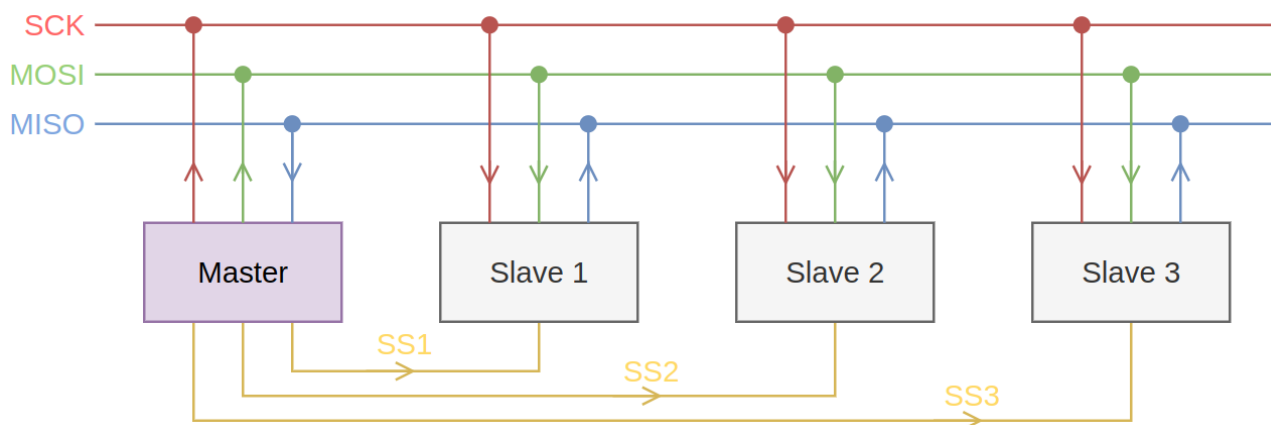
Ainsi il a été choisi pour ce document d'utiliser la terminologie historique, car (1) toutes les ressources (sites web, datasheets, ...) n'ont pas évolué (et si elles ont évolué, ce n'est pas toujours dans la même direction), car (2) tout le monde connaît cette terminologie et car (3), à titre plus pédagogique, cela permet une différenciation avec la terminologie utilisée dans le protocole I<sup>2</sup>C (partie suivante).

Pour fonctionner, le protocole SPI utilise trois signaux communs à tous les acteurs du bus :

- **SCK** : Serial Clock ou parfois SCL, SCK
- **MOSI** : Master Out, Slave In ou parfois Main Out, Sub In
  - SDO (Serial Data Out), PICO (Peripheral In, Controller Out), COTI (Controller Out, Target In), ...
- **MISO** : Master In, Slave Out ou parfois Main In, Sub Out
  - SDI (Serial Data In), POI (Peripheral Out, Controller In), CITO (Controller In, Target Out), ...

De plus le maître dispose d'une connexion pour chaque esclave :

- **$\overline{SS}$**  : Slave Select ou parfois nSS (not Slave-Select),  $\overline{CS}$  (Chip Select), CE (Chip Enable)



<sup>7</sup> [https://en.wikipedia.org/wiki/Master/slave\\_\(technology\)](https://en.wikipedia.org/wiki/Master/slave_(technology))

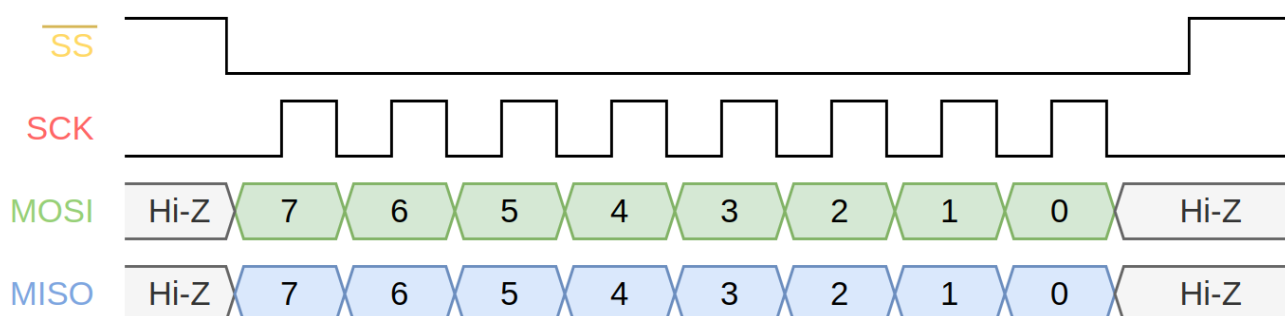
## II. Constitution d'une trame

SPI n'étant pas un standard uniformisé, chaque acteur du marché y va de sa particularité. Cette page décrit ce qui ressemble à un fonctionnement « standard », même si vous pourriez rencontrer des contre-exemples selon les composants manipulés.

Pour échanger avec un esclave, le maître doit d'abord sélectionner celui-ci en imposant généralement un niveau logique bas à la ligne  $\overline{SS}$  concernée.

Ensuite le maître produit un signal d'horloge, d'une durée de généralement huit coups. Chaque coup d'horloge permet au maître et à l'esclave de produire un bit, en commençant généralement par le bit de poids fort.

Dans l'exemple ci-dessous, les bits sont produits à chaque front descendant d'horloge, et ils sont lus à chaque front montant (ce qui laisse le temps aux lignes MOSI et MISO de se stabiliser).



Le niveau de repos de la ligne d'horloge et l'instant d'échantillonnage (lecture) des bits sont deux paramètres généralement ajustables, respectivement nommés **CPOL** et **CPHA**. Un état de repos (*idle state*) au niveau bas est noté CPOL = '0', tandis qu'un état de repos au niveau haut est noté CPOL = '1'. La valeur CPHA = '0' indique que l'échantillonnage du premier bit se fera sur le premier front d'horloge, alors que la valeur CPHA = '1' indique que l'échantillonnage sera fait au deuxième front d'horloge. Le mode illustré ici est le « Mode (0, 0) » ou « Mode 0 »<sup>8</sup>.

Il existe d'autres variantes qui ne seront pas abordées ici, notamment le *three-wire SPI*. Ressemblant fortement au SPI « classique » (parfois appelé *four-wire SPI*), celui-ci ne dispose que d'une seule ligne d'échange de données (SISO ou MIMO), mais elle est bidirectionnelle. Ainsi le *three-wire SPI* est une version *half-duplex* du SPI, rencontrée dans des applications encore plus basse consommation.

Vous l'aurez compris, plusieurs paramètres évolueront selon le concepteur du contrôleur (par exemple un périphérique de MCU) ou du composant esclave.

D'ailleurs généralement, les périphériques externes (capteurs, actionneurs) sont dans un mode de fonctionnement qui est figé à la fabrication. C'est alors au MCU de s'adapter (via la configuration de son périphérique SPI) pour assurer la communication.

<sup>8</sup> L'article Wikipedia explique bien les différents modes : [https://en.wikipedia.org/wiki/Serial\\_Peripheral\\_Interface#Clock\\_polarity\\_and\\_phase](https://en.wikipedia.org/wiki/Serial_Peripheral_Interface#Clock_polarity_and_phase)

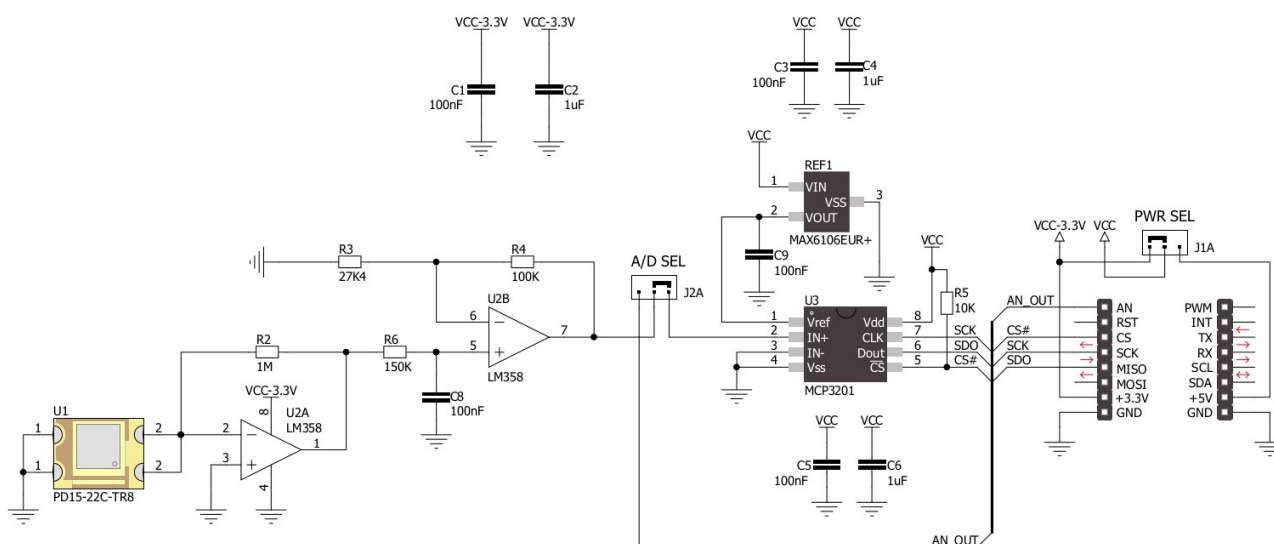
### III. Application : Light click

Le premier périphérique avec lequel nous allons communiquer est un Click board™ de l'entreprise MIKROE : le Light click<sup>9</sup>. Il s'agit d'une carte fille embarquant un capteur de luminosité.

#### III.1. Étude du capteur

Pour chaque carte que nous utiliserons, vous pourrez trouver les documents utiles (schéma, datasheets, ...) dans le répertoire [connectivity/datasheets/](#) de l'archive de TP.

À partir des documents mis à disposition, indiquez sur le schéma ci-dessous les blocs fonctionnels qui constituent cette carte, ainsi que leur fonction.



Repérez et indiquez les broches du Click board qui serviront à la communication SPI avec le micro-contrôleur.

À l'aide du schéma de l'Arduino UNO click shield et du schéma de la carte NUCLEO, complétez le tableau suivant pour arriver à « suivre » les signaux SPI depuis l'ADC jusqu'au MCU (attention, cela dépend de l'emplacement utilisé sur le Arduino shield : on choisira le slot #1).

	MCP3201	Click board	Arduino shield	STM32
Ligne de données	Dout	MISO		
Ligne d'horloge	CLK	SCK		
Ligne de sélection (slot #1 du shield)	$\overline{CS}$	CS		

<sup>9</sup> <https://www.mikroe.com/light-click>

### III.2. Configuration du périphérique SPI

Nous avons vu que le MCP3201 est un ADC avec une interface SPI. Afin d'échanger avec le Light click, il faut donc configurer le périphérique SPI sur notre micro-contrôleur.

📖 Dans STM32Cube IDE, créez un nouveau projet STM32 (reprenez l'annexe si besoin).

📖 Vous aurez besoin des informations suivantes lors de la **création** du projet :

- **Board selector** : celle que vous avez en TP (voir au dos, référence NUCLEO-xxxxxx)
- **Project Name** : VOTRENOM\_spi ;
- **Project Location** : sélectionnez le répertoire `connectivity/workspace/spi/` ;
- **Initialize all peripherals with their default Mode ?** : Yes.

📖 Vous aurez besoin des informations suivantes lors de la **configuration** du projet :

- Connectivity → SPI1
  - Mode : Full-Duplex Master
  - Hardware NSS<sup>10</sup> Signal : Disable
  - Basic parameters : Motorola // 8 Bits // MSB First
  - Clock parameters : Prescaler = 2 // Clock polarity = Low // Clock Phase = 1 Edge
  - GPIO settings : utilisez vos réponses à la question page précédente pour rediriger les signaux SPI1\_SCK // SPI1\_MISO // SPI1\_MOSI vers les broches que vous avez repérées.
- System Core → GPIO
  - utilisez vos réponses pour rediriger le signal CS/SS vers la broche correspondante
    - Sur la vue composant : clic gauche sur la broche → **GPIO\_Output**
    - Sur le volet « GPIO Mode and Configuration », P<sub>??</sub> configuration :
      - GPIO Output level = High
      - ...
      - User Label = **SS\_SLOT1**

📖 Générez le code (🔧) et vérifiez que la fonction **main()** contient désormais l'appel à la fonction **MX\_SPI1\_Init()**.

Le configurateur de code d'initialisation STM32CubeMX montre une nouvelle fois son intérêt, puisque le périphérique SPI est prêt à l'emploi en quelques minutes. Pour vous donner une métrique complémentaire à celle de l'EUSART, le chapitre « SPI/I2S » du *reference manual* compte 50 pages (sur 1040). Le périphérique SPI utilise neuf registres.

<sup>10</sup> NSS = Not Slave Select =  $\overline{SS}$ , or also NCS = Not Chip Select =  $\overline{CS}$

### III.3. Driver `light_click`

Certes le périphérique SPI est configuré et prêt à l'emploi, mais il faut maintenant développer les fonctions qui permettront de communiquer avec le capteur via protocole SPI. Une ébauche de driver a été rédigée pour ce TP. Vous allez l'importer dans le projet puis le compléter.

📁 Les fichiers drivers se trouvent dans le répertoire `workspace/ensicaen_drivers/`. Copiez-collez les fichiers `light_click.c` et `light_click.h` dans les répertoires du projet `.../CM4/Core/Src/` et `.../CM4/Core/Inc/` respectivement. Depuis l'explorateur de projet de l'IDE, vous allez sûrement devoir rafraîchir ces deux dossiers (F5) pour voir apparaître les nouveaux fichiers.

En ouvrant et lisant le *header*, on se rend compte que ce composant est « relativement » simple à utiliser : il y a une fonction d'initialisation et deux fonctions de lecture. En effet, l'ADC MCP3201 n'est pas configurable et ne fait que envoyer par SPI la donnée acquise. Il y a également une structure nommée `LIGHT_handle_t`. Pour simplifier, un *handle* est un objet qui contient les informations (état, configuration) d'une ressource.

En lisant le fichier C maintenant, on retrouve la définition des trois fonctions vues précédemment. Mais celles-ci sont vides, ce sera à vous d'en rédiger le contenu.

#### III.3.a. Initialisation du handle

Lisez la documentation (commentaires Doxygen) du fichier `light_click.h`.

🔪 À quoi sert le champ `hspi` de la structure `LIGHT_handle_t` ? Quelles broches (ou quels signaux) permet-il de manipuler ?

🔪 À quoi servent les champs `SS_Port` et `SS_Pin` de la structure `LIGHT_handle_t` ? Quelle broche (ou quel signal) permettent-il de manipuler ?

🔪 Quel est le rôle de la fonction `LIGHT_init()` ? Quels paramètres doit-on lui fournir ?

✎ Parcourez les fichiers `main.c` et `main.h` pour retrouver le *handle* du périphérique SPI et la définition du port et de numéro de la broche SS. Comment se nomment-ils ?

✎ Dans le fichier `light_click.c`, retrouvez le nom du *handle* du périphérique Light click.

📖 Complétez la définition de la fonction `LIGHT_init()` en suivant les commentaires et les réponses précédentes.

📖 Dans la fonction `main()`, rédigez le code ci-dessous afin d'appeler la fonction d'initialisation du Light click. Rappelez-vous que la documentation dans le fichier `light_click.h` vous donne des informations sur les paramètres attendus par cette fonction.

```
/* USER CODE BEGIN 2 */
HAL_UART_Transmit(&huart2, (uint8_t*)"App initialization...\r\n",
                  strlen("App initialization...\r\n"), HAL_MAX_DELAY);

LIGHT_init( /** @TODO */ );

HAL_UART_Transmit(&huart2, (uint8_t*)"App running...\r\n",
                  strlen("App running...\r\n"), HAL_MAX_DELAY);
/* USER CODE END 2 */
```

📖 Compilez. S'il existe des erreurs ou des warnings, corrigez votre code jusqu'à les faire disparaître.

Notez qu'il ne sert à rien d'exécuter le programme puisque nous n'avons pas encore écrit les fonctions de lecture de données depuis le Light click.

### III.3.b. Fonction de lecture depuis le périphérique

Passons maintenant à la rédaction de `LIGHT_readBrightnessRaw()`, la plus simple des deux fonctions de lecture.

✎ Lisez la section « 5.0 Serial communications » de la datasheet du MCP3201. Quels signaux sont nécessaires et quelles sont les conditions à remplir pour déclencher une lecture de données ?

✎ Indiquez le nombre de coups d'horloge nécessaires pour l'obtention d'une donnée complète. Que se passe-t-il si le signal d'horloge continue à battre après réception d'une donnée ?

✎ Comment, en pratique, allons-nous procéder pour lire une donnée complète sachant qu'un périphérique SPI standard ne sait donner des coups d'horloge que 8 coups par 8 coups<sup>11</sup> ? La réponse est dans la section « 6.1 Using the MCP3201 Device with Microcontroller SPI Ports ».

✎ Quelle(s) opération(s) doit-on effectuer sur les données reçues pour les transformer en un nombre sur 12 bits ?

<sup>11</sup> Il est en réalité possible de configurer ce périphérique SPI pour utiliser des trames de 4 à 16 bits. Nous gardons une taille de 8 bits par soucis de généralité (et de compatibilité avec les prochains Click boards).

✍ Comment se nomme la fonction de réception bloquante du périphérique SPI ? Quels sont les paramètres à lui transmettre ?

💻 Grâce aux réponses précédentes, complétez la fonction `LIGHT_readBrightnessRaw()` de sorte à déclencher une lecture de deux octets, puis convertir les données obtenues en un nombre 12 bits, dont la valeur sera finalement retournée via `*brightness_raw`.

💻 Dans la boucle `while(1)` du `main()`, intégrez le code ci-dessous.

```
HAL_Delay(1000);

// LIGHT CLICK VALIDATION
LIGHT_readBrightnessRaw( &brightness_raw );
sprintf( uart_out, "LIGHT\t Raw brightness = 0x%04X = %5d\n\r", brightness_raw,
brightness_raw );
HAL_UART_Transmit( &huart2, (uint8_t*)uart_out, strlen(uart_out), HAL_MAX_DELAY );
```

💻 Compilez, corrigez les éventuelles erreurs, et téléversez le programme sur cible.

💻 Ouvrez un terminal série et observez le résultat. La valeur fournie par le capteur semble-t-elle cohérente ?

Il est fortement probable que le capteur vous renvoie uniquement des `0x0000`. Si c'est le cas, cela signifie que l'esclave (le Light click) ne répond pas. Vérifions à l'aide d'un analyseur logique.

### III.3.c. Analyse de l'échange

☞ Prenez l'analyseur logique et relevez les 4 signaux liés au périphérique SPI. Vous avez noté le numéro de ses broches et vous avez à votre disposition le *schematic* de la carte NUCLEO.

✎ Normalement, vous devriez observer deux fois huit coups d'horloge : la « demande » est donc bien transmise du maître à l'esclave, mais ce dernier ne semble pas réagir. Quel signal faut-il manipuler pour indiquer à l'esclave que le maître s'adresse à lui ?

✎ Ce signal est contrôlé par une broche configurée en GPIO, précisément en `GPIO_Output`. Quelle fonction devez-vous utiliser pour piloter une GPIO ? (Vous l'avez utilisée pour la LED).

☞ Complétez alors la définition de `LIGHT_readBrightnessRaw()` de sorte à activer l'esclave avant la demande de lecture, et à le désactiver une fois la lecture terminée.

☞ Compilez, téléversez sur cible et confirmez avec l'analyseur logique que l'esclave répond désormais aux sollicitations du maître. Sauvegardez cette capture.

☞ Confirmez également avec le terminal série que la valeur de luminosité s'affiche et varie.

### III.3.d. Fonction de lecture en format flottant

En prenant en compte tous les composants du Light click, on pourrait remonter à une grandeur physique plus parlante qu'un entier compris entre 0 et 4095. Mais pour garder les choses simples, nous allons simplement demander une image de l'irradiance en pourcentage. Ceci sera réalisé par la fonction `LIGHT_readBrightnessPercentage()`.

📖 Complétez la définition de la fonction `LIGHT_readBrightnessPercentage()` de sorte à ce qu'elle réponde à son cahier des charges (indiqué dans le *header* du Light click).

📖 Puis dans la boucle principale du `main()`, ajoutez les lignes suivantes.

```
LIGHT_readBrightnessPercentage( &brightness_percent );
sprintf( uart_out, "LIGHT\t Brightness percentage = %.2f%\n\r", brightness_percent );
HAL_UART_Transmit( &huart2, (uint8_t*)uart_out, strlen(uart_out), HAL_MAX_DELAY );
```

Note : en compilant, votre chaîne de compilation vous indiquera ce warning.

« *The float formatting support is not enabled, check your MCU Settings from "Project Properties > C/C++ Build > Settings > Tool Settings", or add manually "-u \_printf\_float" in linker flags.* »

Suivez cette consigne pour supprimer le warning et surtout permettre l'affichage de nombres à virgule flottante.

📖 Validez le fonctionnement avec un terminal série.

Félicitations ! Vous avez franchi une étape fondamentale dans le monde du développement embarqué, en ayant développé un driver pour un capteur communiquant par bus SPI !

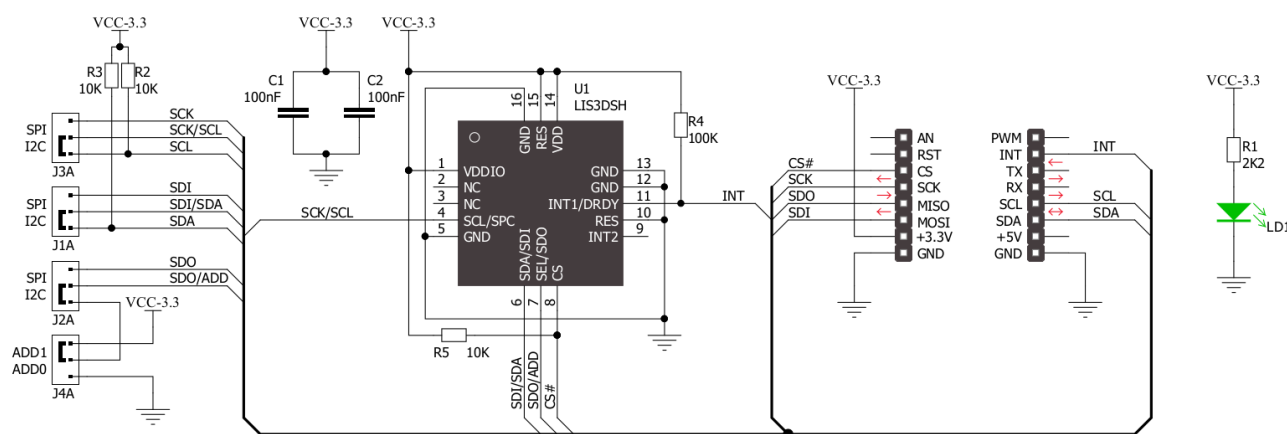
### IV. Application : Accel 2 click

Nous allons maintenant utiliser un autre Click board™ : le Accel 2 click<sup>12</sup>. Comme vous vous en doutez il s'agit d'un accéléromètre. Nous communiquerons toujours en SPI, mais vous allez voir que ce composant est un peu plus complexe que le Light click.

#### IV.1. Étude du capteur

Ici l'étude du schéma n'a que peu d'intérêt puisque le Accel 2 click ne contient qu'un seul circuit intégré (IC), l'accéléromètre 3-axes LIS3DSH de STMicroelectronics. Nous pouvons toutefois noter la présence de quatre cavaliers (*jumpers*) sur la gauche du schéma.

À quels composants ces *jumpers* correspondent-ils physiquement sur la carte ? Quelle est leur fonction ?



En plaçant le Accel 2 click sur l'emplacement n°2 du shield, indiquez à quelles broches du MCU sont reliés les signaux MOSI, MISO, SCK et CS.

	LIS3DSH	Click board	Arduino shield	STM32
Données (STM → LIS)		MOSI		
Données (STM ← LIS)		MISO		
Horloge		SCK		
Sélection du CI (slot #2)		CS		

<sup>12</sup> <https://www.mikroe.com/accel-2-click>

### IV.2. Configuration du périphérique SPI

Vous avez pu constater que le périphérique SPI n'a pas besoin d'être reconfiguré puisqu'il utilise les mêmes broches que le Light Click : c'est d'ailleurs l'intérêt d'un bus de communication.

**Les broches MOSI, MISO et SCK sont ici communes au maître et à tous les esclaves.**

En revanche, le signal CS étant unique pour chaque esclave, il reste à configurer la GPIO qui pilotera le signal CS de l'Accel 2 click.

 Reprenez le projet STM32CubeIDE `VOTRENOM_spi`.

 Ouvrez le fichier `VOTRENOM_spi.ioc` afin de configurer la GPIO correspondant à CS.

- System Core → GPIO
  - Sur la vue composant : clic gauche sur la broche → `GPIO_Output`
  - Sur le volet « GPIO Mode and Configuration », P?? configuration :
    - GPIO Output level = High
    - ...
    - User Label = `SS_SLOT2`

 Générez le code (  ).

Vous pourrez constater que la fonction `MX_GPIO_Init()` du `main.c` contient désormais l'initialisation de la GPIO.

### IV.3. Driver accel\_2\_click

Abordons maintenant le driver de l'Accel 2 click.

Les fichiers drivers se trouvent dans le répertoire `workspace/ensicaen_drivers/`. Copiez-collez les fichiers `accel_2_click.c` et `accel_2_click.h` dans les répertoires du projet `.../CM4/Core/Src/` et `.../CM4/Core/Inc/` respectivement. Depuis l'explorateur de projet de l'IDE, vous allez sûrement devoir rafraîchir ces deux dossiers (F5) pour voir apparaître les nouveaux fichiers.

À première vue, le *header* est bien plus complet que celui du Light click. Mais à y regarder de plus près, la première partie ne contient que des macro-constantes (nous y reviendrons). Quant à ce qui suit, cela ressemble d'assez près au driver précédent : on retrouve une structure (`ACCEL_2_handle_t`, qui contient les informations de l'Accel 2 click après initialisation), une fonction d'initialisation, quelques fonctions de lecture et une fonction d'écriture.

```
void ACCEL_2_init(SPI_HandleTypeDef *hspi, GPIO_TypeDef *SS_Port, uint16_t SS_Pin);

void ACCEL_2_writeReg(uint8_t reg_addr, uint8_t reg_value);
void ACCEL_2_readReg(uint8_t reg_addr, uint8_t* reg_value, uint8_t n_bytes);

void ACCEL_2_getSensitivity(float* sensitivity);
void ACCEL_2_getAccelX(float* x_accel);
void ACCEL_2_getAccelY(float* y_value);
void ACCEL_2_getAccelZ(float* z_value);
void ACCEL_2_getAccelXYZ(float* x_accel, float* y_accel, float* z_accel);
```

Dans le fichier C, certaines définitions de fonctions restent à compléter. Nous les aborderons étape par étape.

#### IV.3.a. Initialisation du handle

Parcourez les fichiers `main.c` et `main.h` pour retrouver le handle du périphérique SPI et la définition du port et de numéro de la broche SS de l'Accel 2 click. Comment se nomment-ils ?

Débutez la définition de la fonction `ACCEL_2_init()`, (commentaire `@TODO (1)`) en initialisant le *handle* avec les paramètres fournis à la fonction.

Dans la fonction `main()`, trouvez l'endroit le plus approprié pour appeler la fonction d'initialisation de l'Accel 2 click. Compilez et éliminez toute source d'erreur ou de warning.

Il n'y a pas lieu d'exécuter le programme sur cible puisque les fonctions de lecture et d'écriture n'ont pas encore été rédigées.

### IV.3.b. Fonction de lecture depuis le périphérique

✎ D'après le chapitre « 5 Digital interfaces » de la datasheet du LIS3DSH, trois modes de communication sont disponibles. Quels sont-ils ?

✎ La table 8 de la datasheet montre que la dénomination des broches est différente de ce qu'on voit habituellement en SPI (CS, SCK, MOSI, MISO). En effet, bien que standardisés, les noms des broches changent parfois selon le fondeur. Ne pas hésiter à noter ici la traduction si besoin.

✎ Comme toute documentation qui se respecte, celle-ci explique comment accéder à une donnée stockée dans le composant. Retrouvez ces informations et retranscrivez-les ici.

📖 À partir de votre précédente réponse, complétez la définition de `ACCEL_2_readReg()`. Aidez-vous du cartouche Doxygen qui précède la déclaration de cette fonction (dans le `.h`).

📖 Dans le `while(1)` du `main()`, ajoutez les lignes suivantes. Compilez et ajustez le code de sorte à corriger les erreurs et warnings.

```
ACCEL_2_readReg( ACCEL_2_REG_WHO_AM_I, &reply, 1 );
sprintf( uart_out, "ACCEL\t Who am I = 0x%02X\n\r", reply );
HAL_UART_Transmit( &huart2, (uint8_t*)uart_out, strlen(uart_out), HAL_MAX_DELAY );
```

✎ Que fait cette portion de code ? Quelle valeur est censée s'afficher sur le terminal série ?

📖 Confirmez avec un terminal série.

### IV.3.c. Modes SPI

☞ Effectuez une capture de cet échange avec un analyseur logique. Comparez la capture avec la « Figure 8. SPI read protocol » de la datasheet. Vous devriez constater des différences. Expliquons-les grâce aux questions suivantes.

✎ D'après la datasheet, quel devrait être l'état de repos de la ligne SCLK ? Qu'observez-vous sur votre capture ?

✎ D'après la datasheet, à quel instant les lignes MOSI et MISO devraient lire chaque bit ? Qu'observez-vous sur votre capture ?

✎ D'après la datasheet, à quel instant les lignes MOSI et MISO devraient écrire chaque bit ? Qu'observez-vous sur votre capture ?

Ces informations correspondent aux **modes (CPOL, CPHA)**. Le paramètre CPOL (*Clock POLarity*) indique le niveau de repos de la ligne d'horloge ('0' = niveau bas, '1' = niveau haut). Le paramètre CPHA (*Clock PHAse*) indique si la donnée (sur MOSI et MISO) est échantillonnée au premier front (valeur '0') ou au second front ('1') d'horloge.

Ainsi, les modes (0, 0) et (1, 1) sont compatibles en ce qui concerne l'échange de données puisque dans les deux cas les bits sont produits sur front descendant d'horloge et sont lus sur front montant. Seul l'état de repos du signal d'horloge change. Similairement, les modes (0, 1) et (1, 0) sont compatibles entre eux.

✎ À quel mode correspond la section « 5.2 SPI bus interface » de la datasheet du LIS3DSH ?

✎ Dans quel mode a été configuré notre périphérique SPI1 ? Réponse dans le fichier [.ioc](#).

Conclusion : configuré ainsi, le périphérique SPI de notre MCU est bien compatible avec le LIS3DSH, même si ce n'est pas exactement le mode demandé par l'accéléromètre.

D'ailleurs si on revient à l'ADC du Light click, la datasheet du MCP3201 indique : « *SPI Mode 0,0 (clock idles low) and SPI Mode 1,1 (clock idles high) are both compatible with the MCP3201* », comme en témoignent les figures 6-1 et 6-2 présentant ces deux modes.

### IV.3.d. Fonction d'écriture vers le périphérique

Le LIS3DSH contient de nombreux registres, dont certains contiennent les paramètres de configuration du capteur. Nous aurons donc à modifier ceux-ci afin d'opérer dans le mode de fonctionnement qui nous intéresse.

✍ Retrouvez dans la documentation les informations indiquant comment modifier une donnée stockée dans le composant, et retranscrivez-les ici.

📄 Complétez la définition de la fonction `ACCEL_2_writeReg()`.

📄 Validez en ajoutant le code suivant au `while(1)` du `main()`.

```
ACCEL_2_writeReg(ACCEL_2_REG_CTRL_REG4, 0x97);
ACCEL_2_readReg( ACCEL_2_REG_CTRL_REG4, &reply, 1);
sprintf( uart_out, "ACCEL\t CTRL_REG4 = 0x%02X\n\r", reply );
HAL_UART_Transmit( &huart2, (uint8_t*)uart_out, strlen(uart_out), HAL_MAX_DELAY );

ACCEL_2_writeReg(ACCEL_2_REG_CTRL_REG4, 0x00);
ACCEL_2_readReg( ACCEL_2_REG_CTRL_REG4, &reply, 1);
sprintf( uart_out, "ACCEL\t CTRL_REG4 = 0x%02X\n\r", reply );
HAL_UART_Transmit( &huart2, (uint8_t*)uart_out, strlen(uart_out), HAL_MAX_DELAY );
```

✍ Quelles sont les opérations réalisées par ces lignes ? (Note : il n'est pas demandé de déterminer le rôle du registre `CTRL_REG4` : ice sera fait plus tard).

📄 Confirmez sur terminal série que votre fonction `ACCEL_2_writeReg()` est fonctionnelle.

📄 Relevez avec analyseur logique une demande d'écriture et la lecture correspondante.

### IV.3.e. Configuration du LIS3DSH

Maintenant que les fonctions de communication SPI entre le MCU et l'Accel 2 click sont opérationnelles, nous allons pouvoir configurer le LIS3DSH de sorte à répondre à nos besoins. C'est ici la principale différence avec le précédent capteur (le Light click) qui n'était pas configurable.

Observez le chapitre « 6 Register mapping » de la *datasheet*. Vous constaterez que tous les registres sont listés ici, avec leur adresse, leur mode d'accès (r, w, r/w), leur description et éventuellement leur valeur par défaut (i.e. à la mise sous tension). Vous avez peut-être remarqué que ces registres ont été listés dans le fichier `accel_2_click.h` à l'aide de macro-constantes qui imposent leur adresse, de sorte à pouvoir les utiliser depuis le driver.

🖥️ Complétez la fonction d'initialisation `ACCEL_2_init()` à l'aide des commentaires (@TODO (2) à @TODO (3)) et de la documentation technique. Attention, aucun *magic number* ne doit apparaître : vous devez utiliser les macro-constantes définies dans le fichier `accel_2_click.h`.

🖥️ Compilez, téléversez.

🔧 Validez avec le terminal série que le programme se bloque si le capteur n'est pas présent au démarrage de l'application.

Quand il découvre la *datasheet* d'un nouveau capteur, le développeur expérimenté cherchera toujours les registres nommés `CTRL`, `CFG`, `SETTINGS`, ou autre nom similaire. Ceux-ci permettent de cibler rapidement les paramètres réglables du composant. Dans ces registres le développeur cherche ensuite les bits *enable* ou *power*, afin d'activer quelques fonctionnalités de base. Avec quelques allers-retours dans la *datasheet*, cela permet d'avoir une configuration qui fait le strict minimum.

Dans un second temps, une lecture approfondie de la documentation est nécessaire pour configurer un composant de sorte à répondre précisément au cahier des charges.

### IV.3.f. Lecture des valeurs d'accélération

Les fonctions de lecture d'accélération sur chaque axe indépendamment et sur les trois axes simultanément ont été rédigées dans le fichier `accel_2_click.c`.

✎ Prenez par exemple la fonction `ACCEL_2_getAccelZ()` et expliquez son fonctionnement. Vous pouvez vous aider de la datasheet, notamment de la table 3

📄 Rédigez dans le `while(1)` du `main()` un code assurant une lecture période de la valeur d'accélération sur l'axe z, et l'affichant sur le terminal série. Vous pouvez vous aider de ce qui a été fait pour le capteur de luminosité.

🔧 Compilez, téléversez et observez le résultat sur terminal série. Confirmez que la valeur d'accélération évolue entre +1.0 g et -1.0 g selon l'orientation de la carte.

📄 Une fois ceci validé, reproduisez la démarche pour les trois axes de l'accéléromètre.

🔧 Compilez, téléversez et observez le résultat sur terminal série. Confirmez que la valeur d'accélération évolue entre +1.0 g et -1.0 g selon l'orientation de la carte.



### IV.3.g. Ajustement : pleine-échelle

Avec tout ce qui a été fait précédemment, vous avez pu déployer un driver remplissant des opérations rudimentaires mais fonctionnelles. Le LIS3DSH offre de nombreuses fonctionnalités (notamment des machines d'état qui pourraient être utilisées pour des applications de casque VR, de souris gamer, ...). Il n'est pas dans les objectifs de cette séquence pédagogique de faire la démonstration du plein potentiel de ce capteur. En revanche, nous pouvons encore peaufiner un détail.

📖 Épurez le code du `while(1)` du `main()` :

- modifiez la période de la boucle à 100 ms (au lieu de 1 s) ;
- commentez ou supprimez toutes les autres lignes, sauf la lecture et l'affichage de l'accélération sur l'axe z.

🔧 Validez sur terminal série que vous obtenez bien la valeur d'accélération.

🔧 Secouez la cible, vous devriez voir sur le terminal série que la valeur d'accélération sature.

🔍 Quel registre et quels bits permettent de régler la sensibilité du capteur ? Quelles valeurs ces bits doivent-ils prendre pour obtenir la plus grande plage de mesure ?

📖 Modifiez la définition de la fonction `ACCEL_2_init()` en imposant une nouvelle valeur de pleine-échelle au capteur.

🔧 Compilez, secouez, validez.

Sacrebleu, la valeur d'accélération ne correspond plus ! En effet la fonction `ACCEL_2_getSensitivity()` retourne toujours la même valeur de sensibilité, or celle-ci dépend de la pleine-échelle de mesure. Au lieu de faire une fonction valable pour une seule sensibilité (et donc pour une seule pleine-échelle), il faut faire évoluer son code de sorte à rendre celle-ci adaptable.

📖 Complétez la fonction `ACCEL_2_getSensitivity()`, de sorte à ce que :

- la valeur de pleine-échelle actuellement configurée dans le LIS3DSH soit récupérée ;
- cette valeur soit testée et que la valeur de sensibilité équivalente soit retournée par la fonction.

🔧 Compilez, téléversez.

🔧 Secouez et validez que la valeur d'accélération dépasse maintenant  $\pm 2 g$ .



## V. Synthèse

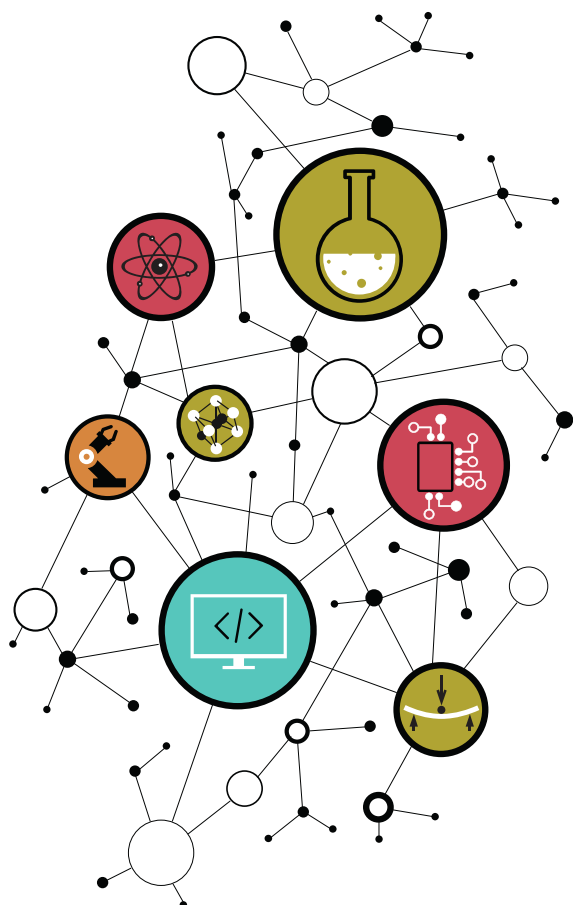
✎ En reprenant les différents points de cette partie, effectuez ici une synthèse de ce qui a été vu. Cette synthèse pourra être consultée lors des futurs développements.

1. Concernant la couche physique (fils, signaux, tensions, niveau logiques, ...)
2. Concernant la couche protocole (nombre de bits, qui gère quelle ligne, ...)
3. Concernant la couche firmware (configuration du périphérique, fonctions utilisées)
4. Concernant le côté capteur (mêmes paramètres ?, accès aux registres, ...)



## PARTIE 4

# BUS I<sup>2</sup>C



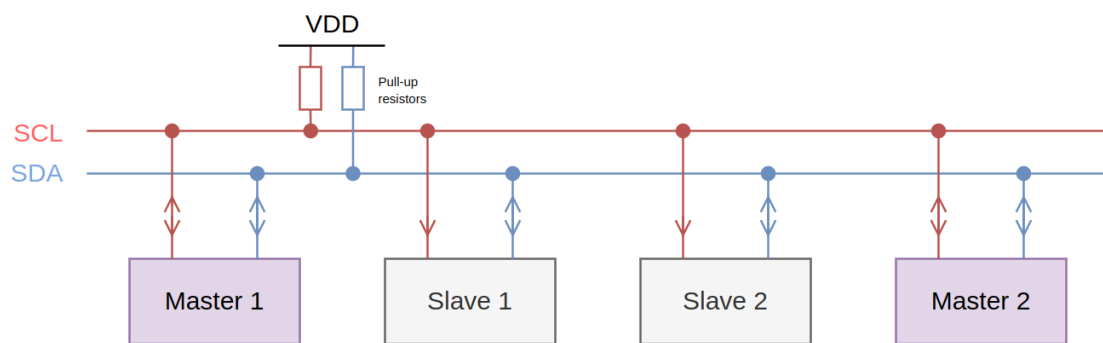
### I. Description

Développé en 1992 par Philips Semiconductors (aujourd'hui NXP), le bus I<sup>2</sup>C est un bus de communication **série, synchrone, half-duplex** et basé sur une topologie **maître-esclaves**. Contrairement au SPI, l'I<sup>2</sup>C offre la possibilité d'avoir plusieurs maîtres sur le bus.

Le bus I<sup>2</sup>C est constitué de deux lignes bidirectionnelles :

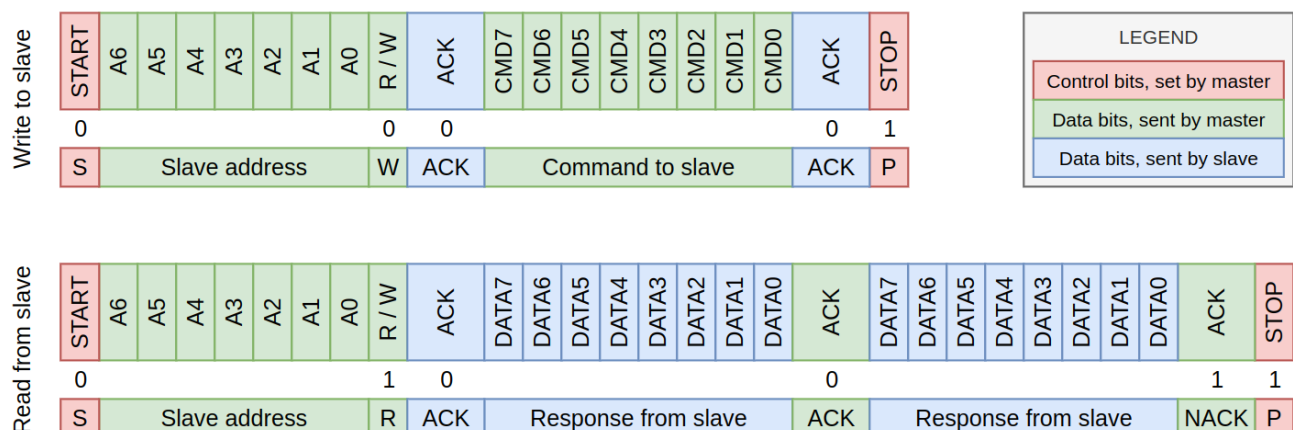
- **SCL** : Serial Clock, ligne manipulée uniquement par les nœuds maîtres ;
- **SDA** : Serial Data, manipulée tantôt par un nœud maître, tantôt par un nœud esclave.

Ces deux lignes sont tirées vers un niveau logique haut de par deux résistances de *pull-up*.



Le principal avantage de l'I<sup>2</sup>C face au SPI est son nombre réduit de fils : **seulement deux fils sont nécessaires** (alors que le SPI nécessite  $3+n$  fils,  $n$  étant le nombre d'esclaves). En effet, la sélection de l'esclave en I<sup>2</sup>C ne se fait pas par le biais d'une ligne électrique, mais par un adressage des cibles. Ceci nous amène au défaut de l'I<sup>2</sup>C : il est **plus lent que le SPI** (en débit utile).

Pour communiquer avec un esclave, le maître envoie d'abord un bit de start ('0'), 7 bits d'adresse désignant l'esclave et un bit de mode d'accès (Read/Write), puis laisse l'esclave acquitter ('0'). Ensuite, en fonction du mode d'accès (Read/Write), la ligne SDA est manipulée soit par le maître soit par l'esclave adressé. Dans tous les cas, c'est toujours le maître qui décide d'arrêter l'échange avec un bit de stop (aussi, en cas de lecture, le maître n'acquitte pas la dernière trame).

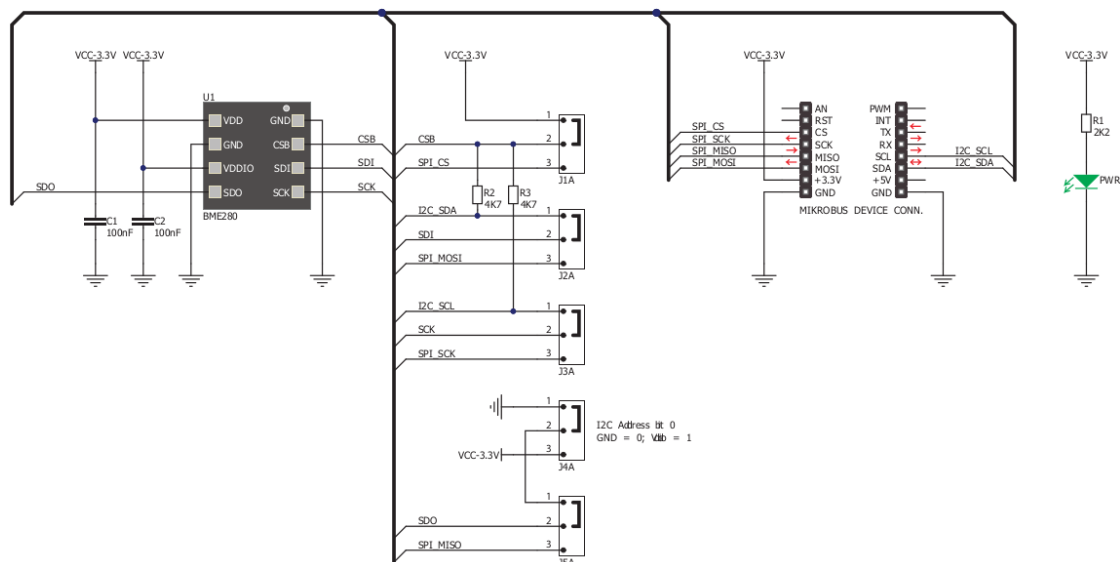


## II. Application : Weather click

Nous allons étudier le protocole I<sup>2</sup>C au travers d'un nouveau Click Board : le Weather click.

### II.1. Étude du capteur

Ici encore, le click board n'est qu'une simple interface de connectique pour le capteur embarqué (un Bosch Sensortec BME280), comme en témoigne son schéma très simple.



🔪 Quelles sont les grandeurs mesurées par le capteur BME280 (voir sa datasheet) ?

🔪 Sur cette carte, quels sont les signaux nécessaires à une communication I<sup>2</sup>C ?

🔪 À l'aide du schéma de l'Arduino UNO click shield et du schéma de la carte NUCLEO, retrouvez sur quelles broches du MCU sont reliées les broches I<sup>2</sup>C du Weather click.

🔪 Trouvez le jumper J4A sur le schéma, et notez la position de ce jumper sur votre carte.

🔪 Quel est le rôle des résistances R2 et R3 ? La section « 6.2 I<sup>2</sup>C interface » de la datasheet du BME280 (capteur du Weather click) peut aider.

### II.2. Configuration du périphérique I<sup>2</sup>C

Comme vous avez pu l'observer, l'interface de communication utilisée avec le Weather click est un bus I<sup>2</sup>C. Configurons ce périphérique sur le micro-contrôleur STM32.

📁 Dans STM32CubeIDE, créez un nouveau projet STM32 (reprenez l'annexe si besoin).

📁 Vous aurez besoin des informations suivantes lors de la **création** du projet :

- **Board selector** : celle que vous avez en TP (voir au dos, référence NUCLEO-xxxxxx)
- **Project Name** : VOTRENOM\_i2c ;
- **Project Location** : sélectionnez le répertoire `connectivity/workspace/i2c/` ;
- **Initialize all peripherals with their default Mode ?** : Yes.

📁 Vous aurez besoin des informations suivantes lors de la **configuration** du projet :

- Sur la vue composant, sélectionnez le mode I<sup>2</sup>C sur les deux broches repérées plus tôt
  - Cela vous permet d'identifier le périphérique I<sup>2</sup>C utilisé (I2C1, I2C2 ou I2C3)
- Connectivity → I2Cx
  - I2C : I2C
  - tout le reste par défaut

📁 Générez le code (🔧) et vérifiez que la fonction `main()` contient désormais l'appel à la fonction `MX_I2Cx_Init()`.

### II.3. Driver weather\_click

Comme pour les Click boards déjà rencontrés dans ce module, il vous faudra développer une partie du driver permettant d'échanger des informations avec le Weather click. Les fichiers drivers se trouvent dans le répertoire `connectivity/workspace/ensicaen_drivers/`.

📁 Copiez et collez les fichiers `weather_click.c` et `weather_click.h` dans les répertoires du projet `.../CM4/Core/Src/` et `.../CM4/Core/Inc/` respectivement. Depuis l'explorateur de projet de l'IDE, vous allez sûrement devoir rafraîchir ces deux dossiers (F5) pour voir apparaître les fichiers.

En parcourant ces deux fichiers, vous devriez être maintenant assez familiers avec leurs contenus. Seulement cette fois-ci, on voit dans le fichier `weather_click.c` des fonctions déclarées et définies avec le mot-clé `static`.

🔪 Que signifie le qualificateur `static` pour une fonction et en C ?

### II.3.a. Initialisation du handle

Lisez la documentation (commentaires Doxygen) du fichier `weather_click.h`.

✎ À quoi sert le champ `hi2c` de la structure `WEATHER_handle_t` ? Quelles broches (ou quels signaux) permet-il de manipuler ?

✎ Quel paramètre doit-on fournir à la fonction `WEATHER_init()` ?

✎ Retrouvez dans le `main.c` la déclaration du *handle* du périphérique I<sup>2</sup>C. Quel est son nom ?

✎ Vous n'avez fait que trouver sa déclaration, mais à quel endroit ce handle est-il initialisé ?

📖 Débutez la définition de la fonction `WEATHER_init()`, (commentaire `@TODO (1)`) en initialisant le *handle* avec le paramètre fourni à la fonction.

📖 Dans la fonction `main()`, appelez la fonction d'initialisation du Weather click. Compilez et éliminez toute source d'erreur ou de warning.

```
/* USER CODE BEGIN 2 */
HAL_UART_Transmit(&huart2, (uint8_t*)"App initialization...\r\n",
                  strlen("\r\nApp initialization...\r\n"), HAL_MAX_DELAY);

WEATHER_init( /** @TODO */ );

HAL_UART_Transmit(&huart2, (uint8_t*)"App running...\r\n",
                  strlen("App running...\r\n"), HAL_MAX_DELAY);
/* USER CODE END 2 */
```

Il n'y a pas lieu d'exécuter le programme sur cible puisque les fonctions de lecture et d'écriture n'ont pas encore été rédigées.

### II.3.b. Fonction de lecture depuis le périphérique

✍ À partir de la datasheet du BME280, retrouvez les informations pour effectuer une lecture des registres du capteur depuis un bus I<sup>2</sup>C. Indiquez ici les étapes.

✍ Quelle est l'adresse de **votre** esclave ?

💻 Définissez la macro-constante `WEATHER_I2C_ADDR` en conséquence.

✍ Dans le fichier driver du périphérique I<sup>2</sup>C fourni par la HAL, retrouvez et indiquez ici les deux fonctions de transmission et réception du périphérique I<sup>2</sup>C.

💻 Avec ces deux fonctions de la HAL et les indications de la datasheet du BME280, complétez la définition de la fonction `WEATHER_readReg()` du fichier `weather_click.c`.

Pour tester la fonction de lecture de registre, nous allons lire la valeur du registre `id`.

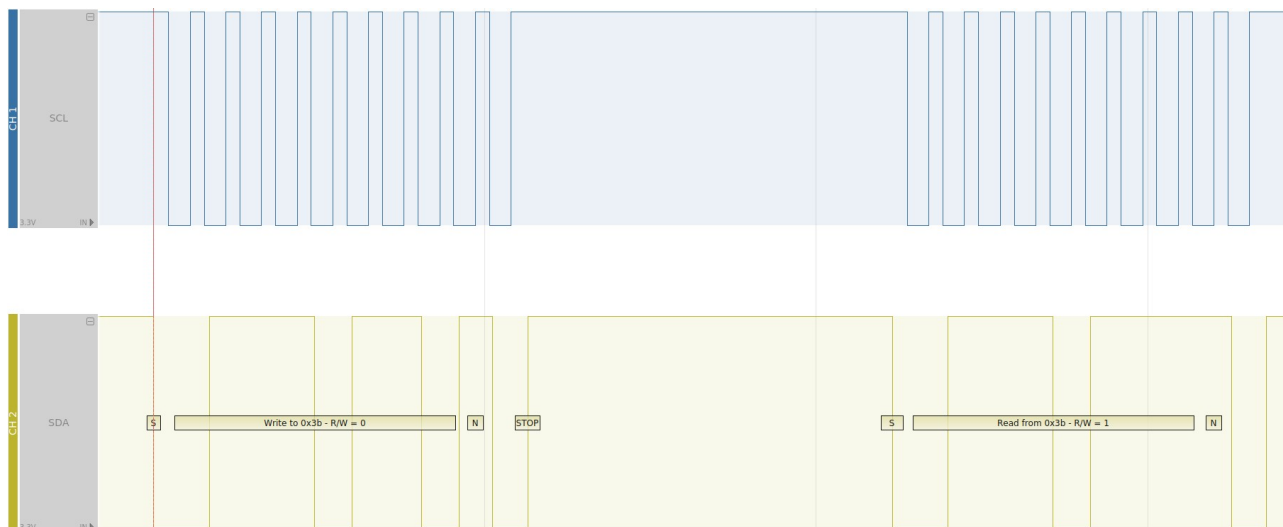
✍ Quelle est l'adresse de ce registre `id` ? À quoi sert-il ? Quelle devrait être la valeur lue ?

💻 Testez la fonction `WEATHER_readReg()` dans le `main()` en lisant la valeur du registre `id`.

💻 Affichez-la sur un terminal série.

La valeur lue ne semble pas être correcte (ou si elle l'est, c'est un heureux hasard).

☞ Effectuez une capture avec un analyseur logique pour vérifier les trames échangées, et effectuez un décodage du protocole I<sup>2</sup>C sur ces trames. Voici ce que vous devriez observer pour un composant d'adresse I<sup>2</sup>C égale à 0x76.



☞ Combien d'octets aurait-on dû observer d'après la figure 10 de la datasheet ? Détaillez.

☞ À quoi correspondent les deux octets observés ?

☞ Quelle est l'adresse décodée ici ? Est-ce celle vous avez demandé (celle de votre esclave) ?  
(Note : les bits d'adresse (et de données) sont lus sur front montant de SCL).

☞ Que remarque-t-on si on compare ces bits d'adresse à ceux de la figure 10 de la datasheet ?

☞ À quoi correspond le bit « N » ou « NACK » indiqué par le décodeur ? Est-ce normal ?

✎ Revenons aux fonctions de la HAL. Si vous lisez le cartouche Doxygen de la fonction de transmission I<sup>2</sup>C `HAL_I2C_Master_Transmit()`, qu'est-il indiqué pour le paramètre `DevAddress` ?

✎ Même question pour la fonction `HAL_I2C_Master_Receive()`.

En effet l'adresse donnée par le capteur est sur 7 bits (« *The 7-bit device address is 111011x* »). Mais pour correspondre au protocole I<sup>2</sup>C, l'adresse de l'esclave doit occuper les 7 bits de poids fort afin de laisser le bit 0 (LSb) indiquer s'il s'agit d'une écriture (valeur '0') ou d'une lecture ('1').

🔧 Modifiez la définition de la macro-constante `WEATHER_I2C_ADDR` pour décaler les bits d'un rang vers les poids forts.

🔧 Effectuez de nouveau un test en affichant la valeur lue sur le terminal série.

🔧 Validez que la valeur affichée sur le terminal est bien celle attendue avant de poursuivre.

🔧 Effectuez une relevé à l'analyseur logique.

🔧 À l'aide du décodeur de trame et de la figure 10 de la datasheet, indiquez le rôle de chaque bit présent sur la ligne SDA, en précisant qui en est à l'origine (maître ou esclave). Soyez aussi complet que possible.

Ce que vous devez constater est que la ligne SDA peut être manipulée par le maître et l'esclave, mais que chacun a un slot temporel fixé (il ne peuvent pas manipuler la ligne en même temps). Cela signifie que le protocole I<sup>2</sup>C est **half-duplex**.

Plusieurs composants peuvent utiliser la même ligne SDA car celle-ci est par défaut tirée vers l'état logique haut grâce à une résistance de pull-up, et toutes les sorties SDA des circuits intégrés sont en open-drain.

✎ Comment un composant écrit-il un '0' logique ? Un '1' logique ?

✎ Quel est le niveau logique d'un acquittement ? Pourquoi ce n'est pas le contraire ?

### II.3.c. Fonction d'initialisation

Vous venez de valider que vous êtes capable de lire un registre du BM2280, un registre contenant l'identifiant du composant qui plus est.

📖 Complétez la fonction d'initialisation `WEATHER_init()` (commentaire `@TODO (2)`) de sorte à ce qu'elle effectue une lecture de ce registre d'identification. Si la valeur récupérée n'est pas celle attendue, on restera bloqué dans un boucle infinie.

🔧 Compilez et testez. Si votre composant est branché vous devriez voir vos deux messages d'initialisation sur le terminal série. S'il n'est pas branché, vous ne devriez voir que le premier message.

Validez ce point avant de passer à la suite.

### II.3.d. Fonction d'écriture vers le périphériques

🔧 À partir de la datasheet du BME280, retrouvez les informations pour effectuer une écriture dans les registres du capteur depuis un bus I<sup>2</sup>C. Indiquez ici les étapes.

📖 Avec ces deux fonctions de la HAL et les indications de la datasheet du BME280, complétez la définition de la fonction `WEATHER_writeReg()` du fichier `weather_click.c`.

📖 Pour valider votre fonction, dé-commentez la suite et fin de la fonction d'initialisation `WEATHER_init()`. Pour chaque registre de configuration, on vient écrire une valeur par I<sup>2</sup>C, puis on effectue une lecture du même registre pour valider que le contenu a bien été modifié (et donc que votre fonction `WEATHER_writeReg()` est correcte).

🔧 Compilez et testez. Comme pour la lecture du registre `id`, vous devriez voir vos deux messages d'initialisation sur le terminal série si votre capteur est branché. S'il ne l'est pas, vous ne devriez voir que le premier message.

### II.3.e. Analyse d'une trame d'écriture

☞ Une fois les fonctions d'initialisation et d'écriture du BME280 validées, effectuez une capture à l'analyseur logique. Trouvez une trame réalisant l'écriture d'une valeur dans un registre.

☞ À l'aide du décodeur de trame et de la figure 9 de la datasheet, indiquez le rôle de chaque bit présent sur la ligne SDA, en précisant qui en est à l'origine (maître ou esclave). Soyez aussi complet que possible.

### II.3.f. Lecture des grandeurs mesurées

Maintenant que la fonction d'initialisation est prête (ainsi que les fonctions permettant de lire et d'écrire un registre du Weather click), vous allez pouvoir récupérer les valeurs des grandeurs d'humidité, de température et de pression. Les fonctions d'accès à ces valeurs vous sont fournies, vous n'avez pas (pour l'instant) à comprendre leur contenu, mais comment vous en servir.

☞ Nettoyez la boucle `while(1)` du `main()`. Développez un code permettant de lire et d'afficher chaque seconde les trois grandeurs mesurées par le capteur.

☞ Compilez et validez sur terminal série.



### II.3.g. Comprendre le driver

Cette partie permet de comprendre comment ont été rédigées les fonctions d'acquisition des trois grandeurs mesurées par le BME280. Cela met en lumière comment lire une documentation technique et comment en extraire les informations utiles.

Afin de maîtriser les différents paramètres pouvant impacter la lecture de la température, de la pression et de l'humidité, une lecture de la datasheet est nécessaire. En effet ce composant est en réalité plus complexe qu'il n'y paraît. Ainsi la lecture des parties citées ci-dessous est conseillée pour répondre aux questions qui suivent :

- 3.4 Measurement flow (jusqu'au 3.4.3 inclus)
- 4. Data readout
- 5.4 Register description (pour les registres que vous lirez)

✎ Quel est l'intérêt du sur-échantillonnage ?

✎ Quel est l'intérêt d'utiliser un filtre (ici à réponse impulsionnelle infinie) ?

✎ Quelle est la résolution des valeurs d'humidité, de pression et de température dans notre cas (pas de filtre, pas de sur-échantillonnage) ?

✎ Quels registres contiennent le résultat des mesures (température, pression et humidité) ?

✎ Le nombre de bits sur lesquels sont stockées les valeurs correspond-il à leur résolution ? Que devrait-on normalement faire si ces nombres ne correspondent pas ?

✎ Après avoir récupéré le contenu des trois registres qui forment la valeur de température, comment reconstruire la valeur de température sur une seule variable de type entier ?

La valeur ainsi construite ne fournit cependant pas la valeur vraie de température. Il est en effet stipulé que chaque élément de mesure est différent (sous-entendu, d'un composant à l'autre). En conséquence, les paramètres de calibration de chaque BME280 ont été déterminés en usine puis stockés en dur dans la mémoire NVM (*Non-Volatile Memory*) du composant. Ces paramètres, accessibles depuis les registres, permettent de « compenser » les valeurs mesurées pour calculer les valeurs vraies.

La datasheet donne le code des fonctions de compensation des trois valeurs (température, pression, humidité) dans la section « 4.2.3 Compensation formulas ». Dans ces extraits de code, les valeurs sont stockées sur des entiers 32-bit en utilisant une représentation en virgule fixe (« *fixed-point arithmetic* »). Bosch Sensortec fournit aussi ces fonctions sur son *repository* GitHub<sup>13</sup> avec des valeurs au format flottant (« *floating-point* »).

Dans le fichier `weather_click.c` les fonctions de compensation ont été ré-écrites, afin de correspondre au reste du driver (prototypes de fonctions, valeurs, structures, ...). Parmi les deux versions proposées, c'est celle effectuant les opérations en virgule fixe qui a été choisie.

✎ Pourquoi devons-nous effectuer les calculs sur des entiers et non sur des flottants ?  
Indice : ceci est lié à notre MCU<sup>14</sup>.

✎ Qu'est-ce qu'un nombre en virgule fixe ? Quels sont les avantages ? Recherchez sur Internet ou auprès de l'enseignant.

<sup>13</sup> [https://github.com/boschsensortec/BME280\\_driver/tree/master](https://github.com/boschsensortec/BME280_driver/tree/master)

<sup>14</sup> Cette question n'est pas forcément juste selon le STM32 utilisé ...

Observez les cartouches Doxygen des trois fonctions de compensation présentes dans le fichier `weather_click.h`. Parmi les informations données, on y trouve le format et l'unité des valeurs retournées `fine_temp`, `fine_press` et `fine_hum`<sup>15</sup>. Observez également le code de ces fonctions dans le fichier `weather_click.c`.

✎ Retrouvez les instructions de conversion de la valeur de température `fine_temp` (format entier 32-bit, exprimé en 0.01 °C) en `temp_degC` au format flottant exprimé en °C.

✎ Retrouvez les instructions de conversion de la valeur d'humidité `fine_hum` (format virgule fixe Q22.10, en % RH) en `hum_rh` au format flottant en %RH (pourcentage d'humidité relative).

✎ Retrouvez les instructions de conversion de la valeur de pression `fine_press` (format virgule fixe Q24.8, en Pa) en `press_hPa` au format flottant en hPa.

<sup>15</sup> Ces informations sont évidemment issues de la datasheet, section « 4.2.3 Compensation formulas ».



### II.4. Synthèse

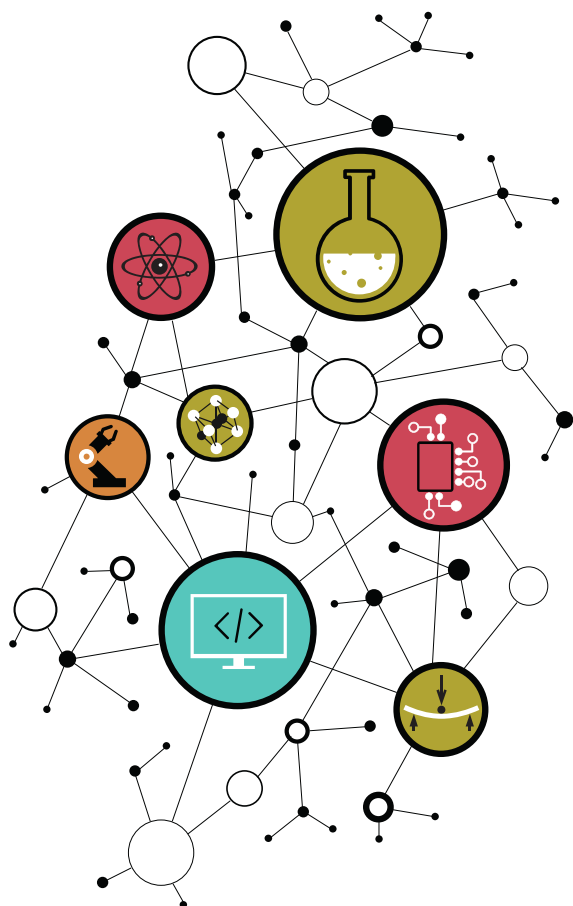
✍ En reprenant les différents points de cette partie, effectuez ici une synthèse de ce qui a été vu. Cette synthèse pourra être consultée lors des futurs développements.

1. Concernant la couche physique (fils, signaux, tensions, niveau logiques, ...)
2. Concernant la couche protocole (nombre de bits, qui gère quelle ligne, ...)
3. Concernant la couche firmware (configuration du périphérique, fonctions utilisées)
4. Concernant le côté capteur (mêmes paramètres ?, accès aux registres, ...)



## PARTIE 5

# GLOSSAIRE



Termes classés par thématique

LSb / MSb LSB / MSB	<i>Least/Most Significant bit</i> <i>Least/Most Significant Byte</i>
<b>MCU</b> CPU Périphérique GPIO	<b>Microcontroller Unit</b> <i>Central Processing Unit</i> Composant matériel interne au MCU, conçu pour une fonction précise <i>General Purpose Input/Output</i> , périphérique pour manipuler les broches
<b>STM</b> STM32 STM32CubeIDE STM32CubeMX HAL STLink VCP	<b>STMicroelectronics</b> , société franco-italienne de semi-conducteurs Gamme de MCU 32-bits de ST, basée sur des CPU ARM Cortex-M cores Environnement de développement intégré pour processeurs ST Configurateur graphique de code d'initialisation pour processeurs ST <i>Hardware Abstraction Layer</i> , fonctions d'utilisation du matériel <i>STLink Virtual Com Port</i> , port COM virtuel accessible via l'USB
NVIC IF IRQ ISR Callback	<i>Nested Vector Interrupt Controller</i> , contrôleur d'interruption des CPU ARM <i>Interrupt Flag</i> , indicateur d'occurrence d'un évènement <i>Interrupt Request</i> , demande au CPU d'interrompre le programme <i>Interrupt Service Routine</i> , fonction exécutée lors d'une interruption Fonction appelée par l'ISR et dont la définition est laissée au développeur
<b>UART</b> USART Tx Rx	<b>Universal Asynchronous Receiver-Transmitter</b> , périphérique de comm série <i>Universal Synchronous-Asynchronous Receiver-Transmitter</i> <i>Transmit line</i> <i>Receive line</i>
<b>SPI</b> MISO MOSI $\overline{SS}$ $\overline{CS}$	<b>Serial Peripheral Interface</b> bus série synchrone bidirectionnel full-duplex, développé par Motorola <i>Master Main In, Slave Sub Out</i> <i>Master Main Out, Slave Sub In</i> <i>Slave Select</i> , ligne de sélection de l'esclave, active à l'état bas <i>Chip Select</i> , ligne de sélection de l'esclave, active à l'état bas
<b>I<sup>2</sup>C</b> SDA SCL ACK NACK	<b>Inter-Integrated Circuit</b> bus série synchrone bidirectionnel half-duplex, développé par Philips <i>Serial Data Line</i> <i>Serial Clock Line</i> <i>Acknowledgment</i> <i>Non-Acknowledgment</i>





