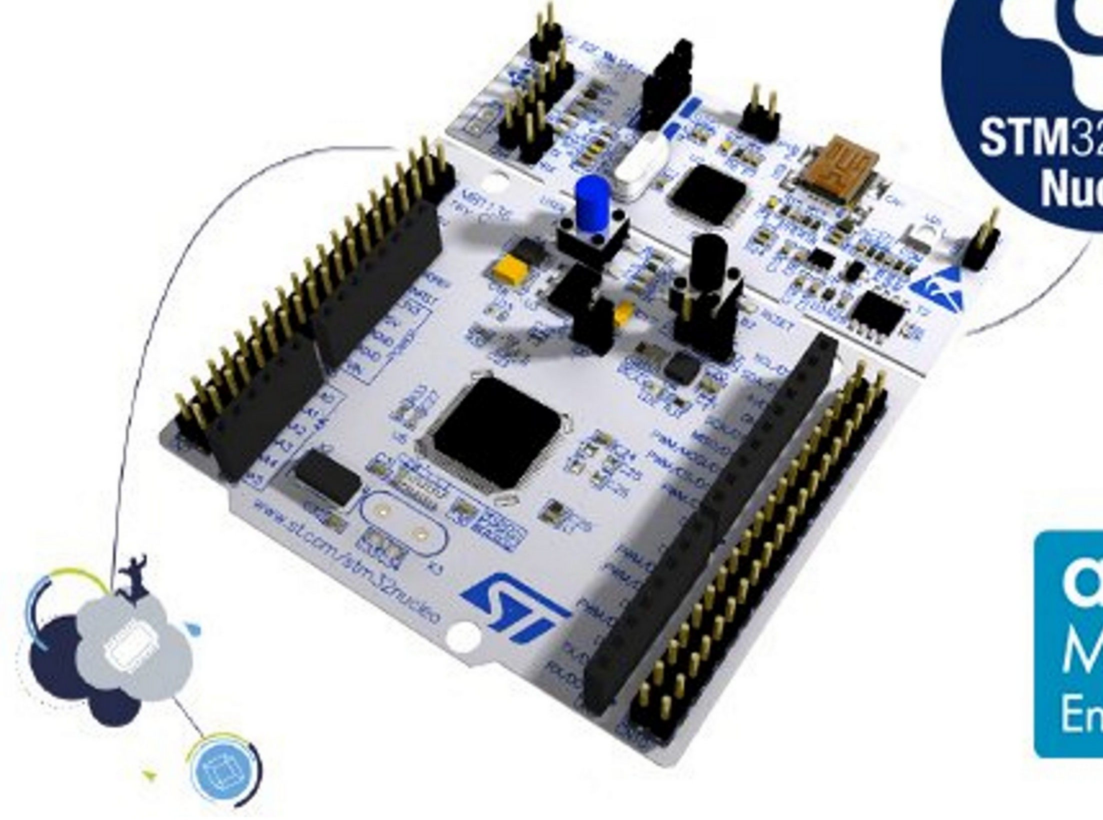
INTRODUCTION AUX STM32

Prise en main du matériel



GPSE / IPC-SATE-ECSE
dimitri.boudier@ensicaen.fr

Contacts

Dimitri Boudier – Auteur du document – Encadrant de TP
dimitri.boudier@ensicaen.fr

Hugo Descoubes – Encadrant de TP
hugo.descoubes@ensicaen.fr

Oumaima Assou – Encadrante de TP
oumaima.assou231@ensicaen.fr

Ressources

Ce document fait office d'introduction pour plusieurs cours. Vous pouvez trouver les cours correspondants sur les pages Moodle suivantes :

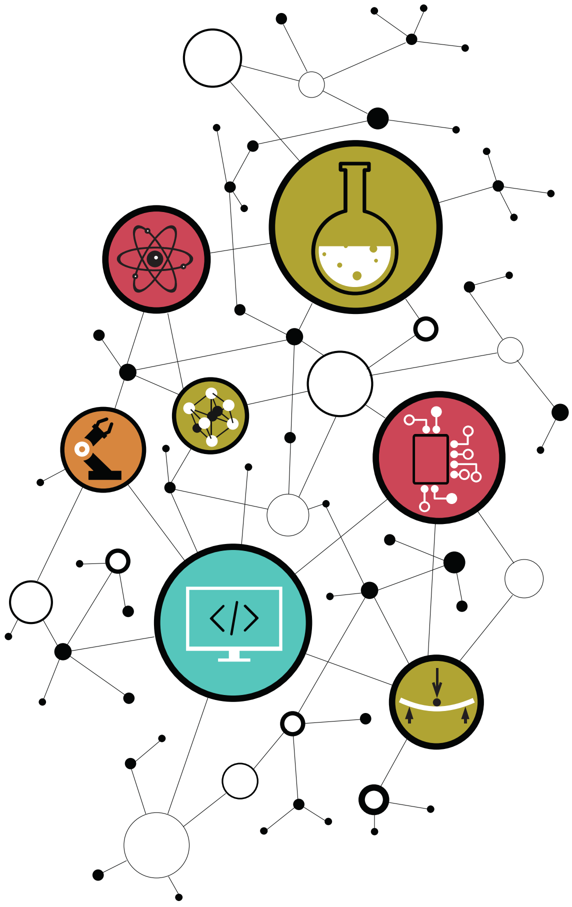
2A SATE – Systèmes Temps-Réel
https://foad.ensicaen.fr/course/view.php?id=118

2A ECSE – Systèmes Temps-Réel
https://foad.ensicaen.fr/course/view.php?id=840

3A IPC – Capteurs & Systèmes Connectés
https://foad.ensicaen.fr/course/view.php?id=213

CHAPITRE 1

PRÉLUDE



I. Mise en contexte

Ce document et les ressources associées ont été rédigés afin de proposer une prise en main simple et en autonomie des STM32 Nucleo Boards. En effet ces cartes sont utilisées dans différents enseignements (systèmes temps-réel, bus de communication, LoRa, …) et les principes de bases vus dans ce document sont nécessaires au suivi de ces enseignements.

I.1. Besoins matériels et logiciels

Afin de répondre au critère d'autonomie, le matériel et les logiciels nécessaires sont réduits au strict minimum : une carte Nucleo, un IDE, un terminal série. Les cartes Nucleo étant accessibles en prêt, les exercices de ce document sont entièrement réalisables à la maison.

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

- une carte NUCLEO-L073RZ
  - n'importe quelle carte NUCLEO (-L746RG, -F411RE, -L053R8, -WL55JC1, …) est compatible.
- Un câble USB-A – mini-USB

Voici les logiciels associés au matériel de la première liste :

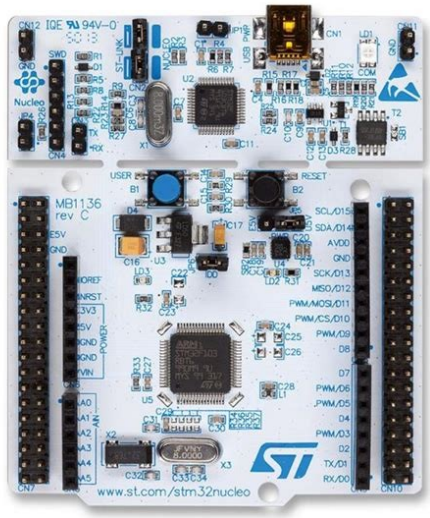
- STM32CubeIDE (validé avec version 1.14.0) :
  - IDE utilisé pour programmer et debugger les cartes NUCLEO
  - https://www.st.com/en/development-tools/stm32cubeide.html
  - n'importe quel autre IDE compatible (VS Code avec extension, Keil, …) est accepté, mais le support enseignant n'est pas garanti !
- STM32CubeMX (validé avec version 6.7.0)
  - Normalement installé avec STM32CUbeIDE, sans besoin de l'ajouter vous-même
- Un terminal série
  - N'importe lequel fait l'affaire
  - Tera Term (Windows), PuTTY (Windows/Linux), GTKTerm (Linux), minicom (Linux), …

Pour information, les pages Moodle des différents cours proposent des tutoriels de prise en main de ces logiciels.

II. Présentation du matériel

II.1. Carte STMicroelectronics NUCLEO

L’entreprise franco-italienne STMicroelectonics 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**.

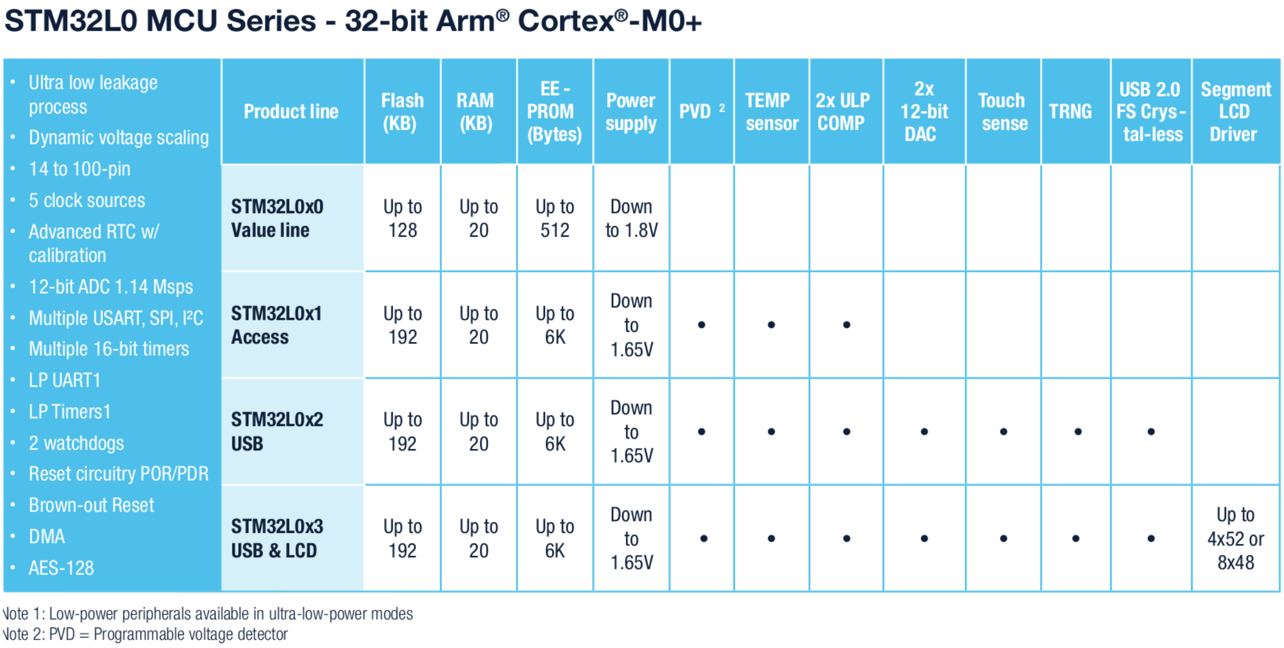


À titre d’exemple, la carte **NUCLEO-L073RZ**[1] utilisée dans ce document contient sur son PCB le MCU cible (STM32L073RZ), une sonde de programmation et debug, un bouton poussoir et une LED.

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.

Le processeur embarqué sur la carte NUCLEO-L073RZ est le micro-contrôleur **STM32L073RZ**[2] 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 les GPIO et l’USART : ce sont ceux avec lesquels nous travaillerons à travers ce document.



STM32L0 MCU Series - 32-bit Arm® Cortex®-M0+

	Product line	Flash (KB)	RAM (KB)	EE-PROM (Bytes)	Power supply	PVD ^{2}	TEMP sensor	2x ULP COMP	2x 12-bit DAC	Touch sense	TRNG	USB 2.0 FS Crys-tal-less	Segment LCD Driver
• Ultra low leakage process • Dynamic voltage scaling • 14 to 100-pin • 5 clock sources • Advanced RTC w/ calibration • 12-bit ADC 1.14 Msps • Multiple USART, SPI, I^{2}C • Multiple 16-bit timers • LP UART1 • LP Timers1 • 2 watchdogs • Reset circuitry POR/PDR • Brown-out Reset • DMA • AES-128	**STM32L0x0 Value line**	Up to 128	Up to 20	Up to 512	Down to 1.8V								
	STM32L0x1 Access	Up to 192	Up to 20	Up to 6K	Down to 1.65V	•	•	•					
	STM32L0x2 USB	Up to 192	Up to 20	Up to 6K	Down to 1.65V	•	•	•	•	•	•	•	
	STM32L0x3 USB & LCD	Up to 192	Up to 20	Up to 6K	Down to 1.65V	•	•	•	•	•	•	•	Up to 4x52 or 8x48

Note 1: Low-power peripherals available in ultra-low-power modes
Note 2: PVD = Programmable voltage detector

---

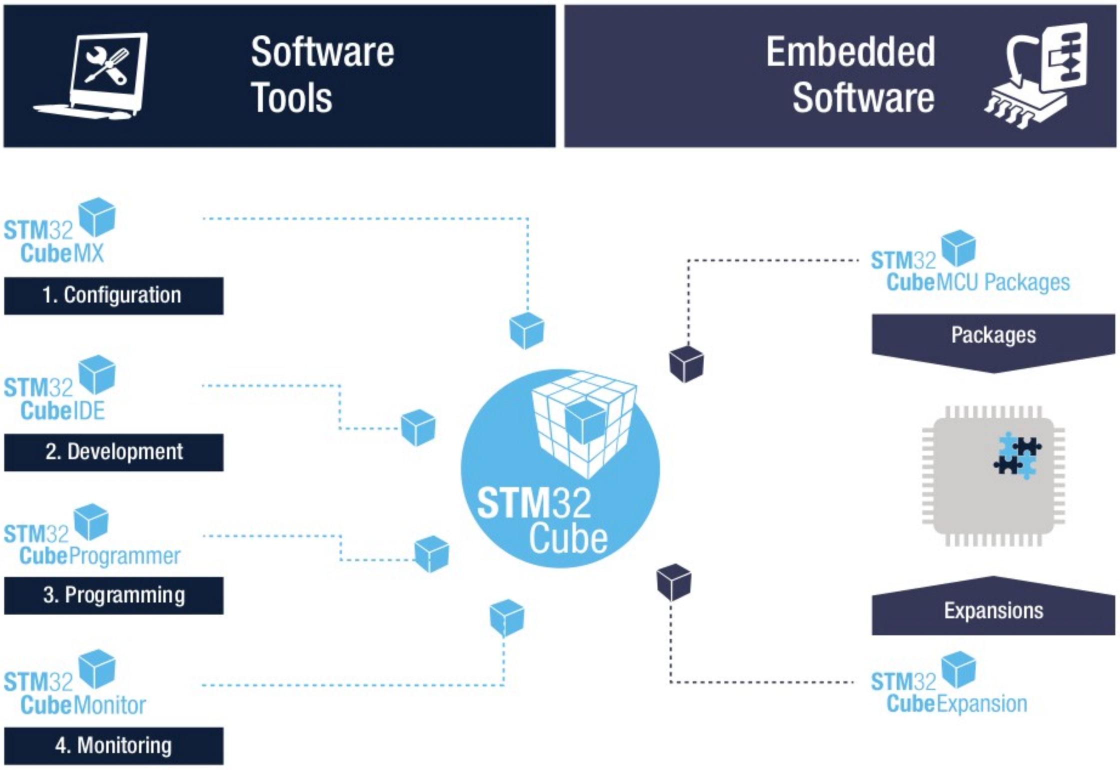
[1] https://www.st.com/en/evaluation-tools/nucleo-l073rz.html

[2] https://www.st.com/en/microcontrollers-microprocessors/stm32l073rz.html

II.2. Environnement de développement

Pour nos développements sur STM32, nous utiliserons la suite logicielle proposée par STMicroelectonics : **STM32Cube**[3]. 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*).

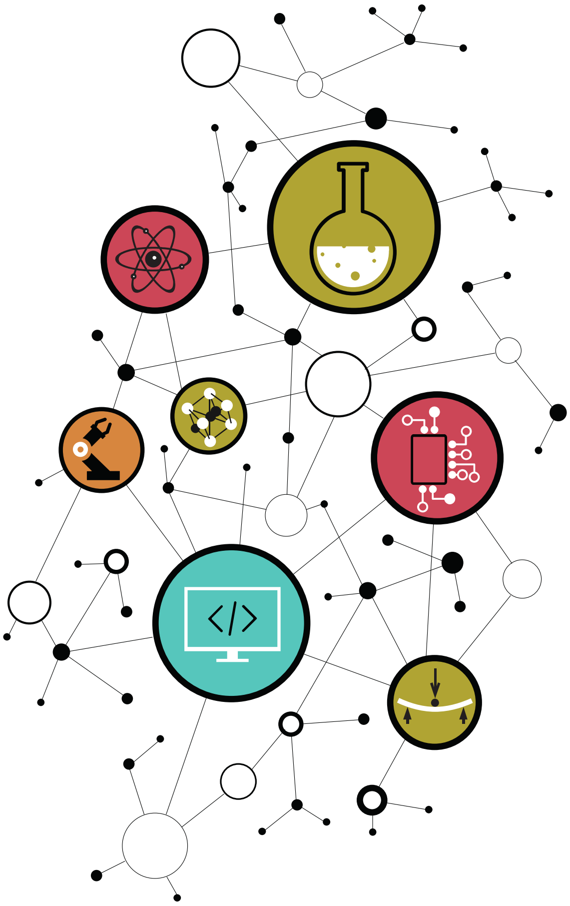
Si on décidait de partir sur de la programmation du micro-contrôleur à l'étage registre (comme en TP MCU de première année), il faudrait ajouter quelques heures à la formation tant les Cortex-M sont complexes (en comparaison à des PIC18). Nous utiliserons donc **STM32CubeMX**, qui permet de configurer 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*).

---

[3] https://www.st.com/en/ecosystems/stm32cube.html

CHAPITRE 2

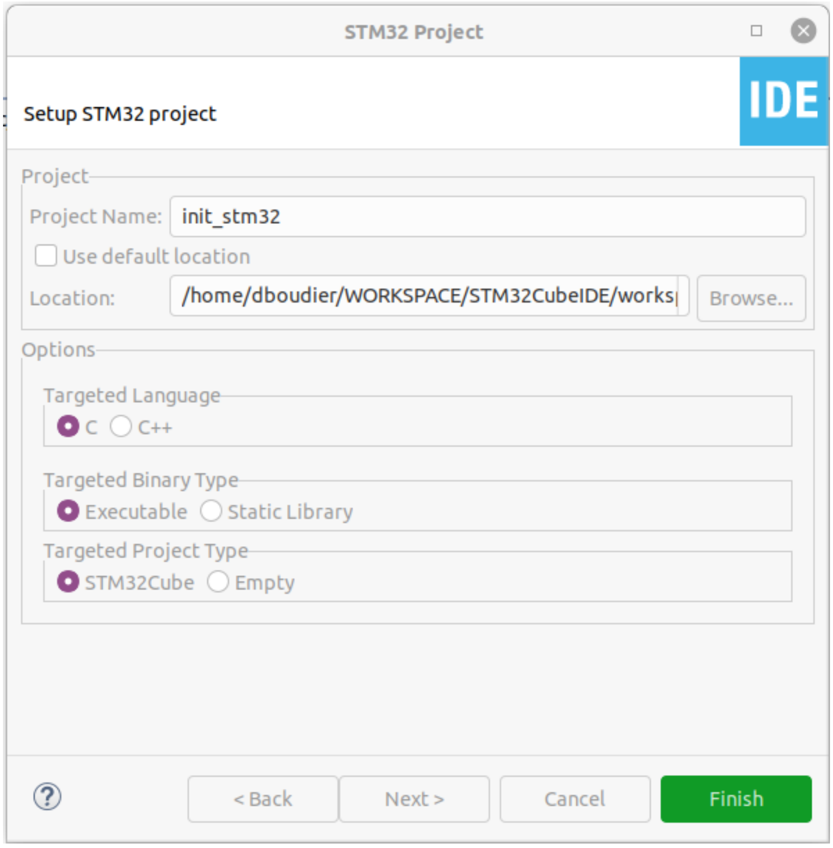
PROJET STM32



I. Création du projet

💻 Ouvrez le logiciel STM32CubeIDE et procédez à la création d'un nouveau projet :

1. Dans **STM32CubeIDE** (pas STM32CubeMX) : `File → New → STM32 Project`.
2. Dans l'onglet « `Board Selector` » (attention, pas l'onglet `MCU/MPU Selector`), retrouver avec la barre de recherche la carte de TP (la référence est sur une étiquette de la carte), la sélectionner et cliquer sur « `Next` ».
   - *Notez qu'il est possible de sélectionner des projets exemples fournis par STMicroelectronics*
3. `Project Name`: nommez le projet `intro_VOTRENOM`
   - Attention : le nom du projet ne doit contenir ni accent, ni espace, ni caractère spécial.
4. `Location`: choisir un emplacement judicieux[4]
   - Si vous êtes en TP, prenez le répertoire `tp/workspace/intro` de l'archive de TP.
   - Attention : le ***chemin complet*** du projet ne doit contenir ni accent, ni espace, ni caractère spécial.
5. Laisser les options (`C` / `Executable` / `STM32Cube`) telles quelles et cliquer sur `Finish`.
6. « `Initalize all peripherals with their default Mode ?` » → « `Yes` »



STM32 Project

Setup STM32 project

Project

Project Name: init_stm32

☐ Use default location

Location: /home/dboudier/WORKSPACE/STM32CubeIDE/works

Options

Targeted Language

● C ○ C++

Targeted Binary Type

● Executable ○ Static Library

Targeted Project Type

● STM32Cube ○ Empty

---

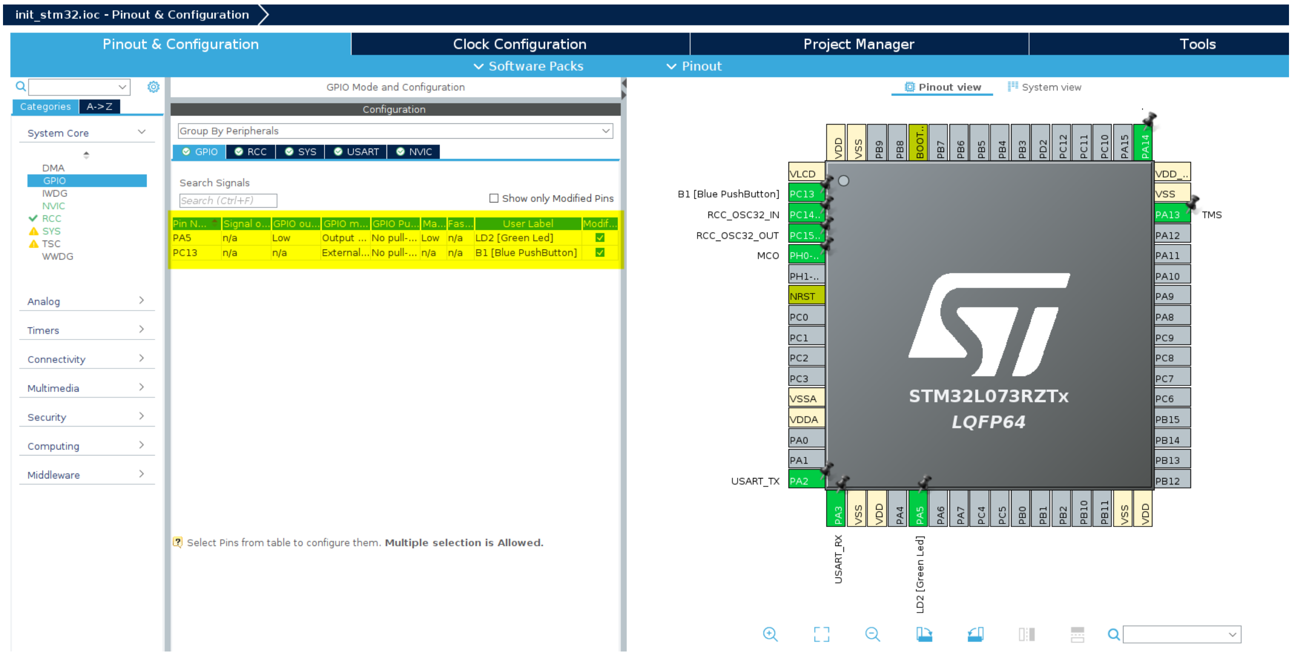
[4] Sous Windows, choisir un emplacement sur le disque `Z:` (et non dans `Mes Documents`).

II. Configuration du processeur

Après avoir créé le projet, une fenêtre de configuration du micro-contrôleur s’ouvre. Il s’agit de l’outil **STM32CubeMX** (le configurateur de code d’initialisation). Avec le choix précédent (à la création du projet), certains périphériques ont ici été configurés par défaut. C’est par exemple le cas du périphérique `System Core/GPIO`, et du périphérique `Connectivity/USART2`.

✎ Dans l’onglet de configuration des GPIOs, retrouvez le numéro de broche et le `GPIO Mode` des broches suivantes (notez qu’on retrouve également ces broches sur le `Pinout view`, à droite) :

- `LD2 [Green LED]` :
- `B1 [Blue PushButton]` :



Pin N...	Signal o...	GPIO ou...	GPIO m...	GPIO Pu...	Ma...	Fas...	User Label	Modif...
PA5	n/a	Low	Output ...	No pull-...	Low	n/a	LD2 [Green Led]	☑
PC13	n/a	n/a	External...	No pull-...	n/a	n/a	B1 [Blue PushButton]	☑

Select Pins from table to configure them. **Multiple selection is Allowed.**

STM32CubeMX a donc préparé la configuration des périphériques. Le développeur a tout de même cette interface graphique à disposition pour modifier la configuration du micro-contrôleur selon ses exigences. Le tout est sauvegardé dans un fichier `*.ioc`, que nous pourrons rouvrir ultérieurement.

De notre côté, nous n’avons pas plus de besoins. Il nous reste à produire le code correspondant à cette configuration.

💻 Cliquez sur `Project → Generate Code` (ou l’icône).

Et voilà ! Les GPIO associées au bouton poussoir et à la LED sont prêtes à être utilisées. Leur initialisation a été effectuée, et des fonctions d’utilisation ont été écrites et sont prêtes à l’emploi par le développeur (vous). Nous abordons tout cela page suivante.

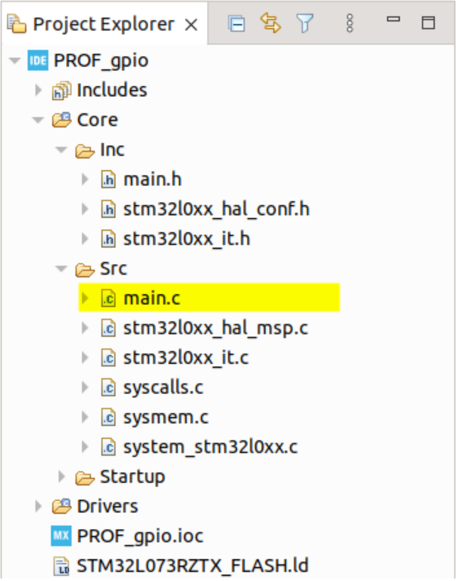
III. Étude du projet généré

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

L'arborescence du projet paraît remplie de prime abord, mais elle est assez simple à comprendre :

- `Core/Src/` : répertoire contenant les fichiers sources (en C et asm) propres à l'application (donc propres à un cahier des charges) ;
- `Core/Inc/` : idem, mais pour les *headers* ;
- `Drivers/STM32L0xx_HAL_Driver/Src/` et `Inc/` : répertoires contenant les sources et headers propres à la HAL (*Hardware Abstraction Layer*) des **MCU STM32**. Autrement dit, ce sont les « pilotes » (ou BSP) fournis par CubeMX pour utiliser les périphériques des MCU STM32 ;
- un fichier `.ioc`, qui ouvre la fenêtre de configuration observée précédemment (CubeMX).

Plus d'informations sont disponibles dans l'annexe `tutoriel_stm32cubeide.pdf`.



🖳 Ouvrez le fichier `Core\Src\main.c`, vous remarquerez qu'il a déjà été pré-rempli. C'est notamment la conséquence du configurateur de code utilisé page précédente.

🖳 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_GPIO_Init()`. Ce `ctrl+clic` vous amène à la définition de la fonction. Celle-ci est définie dans le fichier `main.c`. Elle a été créée quand vous étiez dans la partie configurateur de projet. Vous pouvez d'ailleurs remarquer que les instructions de cette fonction correspondent aux paramètres de l'interface graphique. Juste après ces paramètres, on remarque l'appel à une fonction de la HAL : `HAL_GPIO_Init()`.

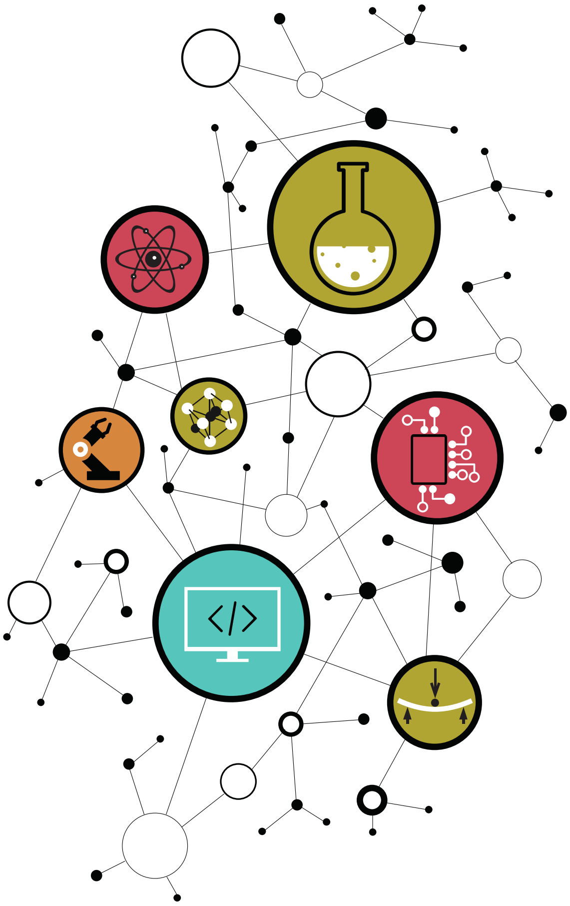
🖳 Effectuez un `Ctrl+Clic` sur l'appel à la fonction `HAL_GPIO_Init()`. Cette fois un nouveau fichier s'ouvre : `stm32l0xx_hal_gpio.c`. Il s'agit d'un fichier de la HAL, contenant toutes les fonctions de configuration et d'utilisation du périphérique GPIO. Parmi celles-ci, vous utiliserez :

- `HAL_GPIO_Init(…)` : fonction de configuration de la broche en GPIO ;
- `HAL_GPIO_ReadPin(…)` : lit le niveau logique de la broche indiquée ;
- `HAL_GPIO_WritePin(…)` : impose un niveau logique à la broche indiquée.

En bref, la HAL (*Hardware Abstraction Layer*) est un ensemble de fonctions d'utilisation des périphériques du MCU, fournies par STMicroelectonics. Pour une explication approfondie, référez-vous au document `tutoriel_stm32cubeide.pdf`.

CHAPITRE 3

PÉRIPHÉRIQUE GPIO



I. Manipuler les GPIOs avec la HAL

Les cartes NUCLEO-64 (famille dont la NUCLEO-LR073 fait partie) intègrent un bouton poussoir et une LED. Vous avez pu constater dans la partie II. Configuration du processeur que ces GPIO ont été configurées, et sont utilisables grâce aux fonctions observées précédemment.

/!\ ATTENTION /!

Vous avez pu remarquer qu'avec la création du projet, vous n'avez pas eu à écrire le contenu de la fonction `main()`. C'est l'outil STM32CubeMX qui s'en est chargé à votre place, vous permettant ainsi de gagner du temps.

Vous avez évidemment le droit d'ajouter du code à celui généré, mais vous devez respecter les espaces que vous impose CubeMX. Ceux-ci se caractérisent par deux balises sous forme de commentaire (par ex. : `/* USER CODE BEGIN WHILE */` et `/* USER CODE END WHILE */`).

Ainsi, le code que vous allez ajouter à partir de maintenant doit forcément être compris entre deux balises de commentaire `BEGIN` et `END`.

Tout code compris en dehors de ces balises se trouvera supprimé si vous faites de nouveau appel à CubeMX (pour ajouter un périphérique par exemple, ce qu'on fera à l'avenir). Tout code compris entre ces balises sera conservé. C'est une règle très simple, mais très stricte.

💻 Dans la boucle `while(1){...}` de la fonction `main()`, écrivez le code suivant.

Attention : écrivez entre les balises dédiées !

```
if( HAL_GPIO_ReadPin(B1_GPIO_Port, B1_Pin) == GPIO_PIN_RESET )
{
    HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_SET);
}
else
{
    HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);
}
```

Notez que les constantes utilisées ici (`B1_GPIO_Port`, `LD2_Pin`, …) ont été définies dans le fichier `main.h`. C'est encore STM32CubeMX qui s'est chargé de préparer ces constantes pour nous.

💻 Compilez (Project → Build All) et lancez l'exécution du programme (Run → Run).

Vous aurez vite deviné comment le programme fonctionne : si le bouton poussoir est pressé, la LED est allumée, sinon la LED est éteinte.

II. Encore d'autres fonction de la HAL

Avec la HAL (*Hardware Abstraction Layer*), STMicroelectonics gâte ses utilisateurs en fournissant un grand nombre de fonctions d'utilisation de ses périphériques. En voici encore deux plutôt utiles.

La fonction `HAL_GPIO_TogglePin(…)` est définie dans le fichier `stm32l0xx_hal_gpio.c`.

✎ En vous aidant du bloc de commentaires, indiquer ce que cette fonction permet de faire.

La fonction `HAL_Delay(…)` est définie dans le fichier `stm32l0xx_hal.c`.

✎ En vous aidant du bloc de commentaire, indiquer ce que cette fonction permet de faire.

💻 À l'aide des deux fonctions découvertes, faîtes clignoter la LED en changeant son niveau logique toutes les deux secondes.

💻 Validez visuellement.

En quelques minutes, vous avez pris en main une carte de développement (pas si simple) intégrant un MCU STM32 (plutôt complexe). Vous avez pu configurer deux GPIO pour les manipuler en entrée et en sortie. L'aide apportée par l'IDE (et plus précisément par CubeMX) fut ici très précieuse en terme de temps de développement !

OPTIONAL

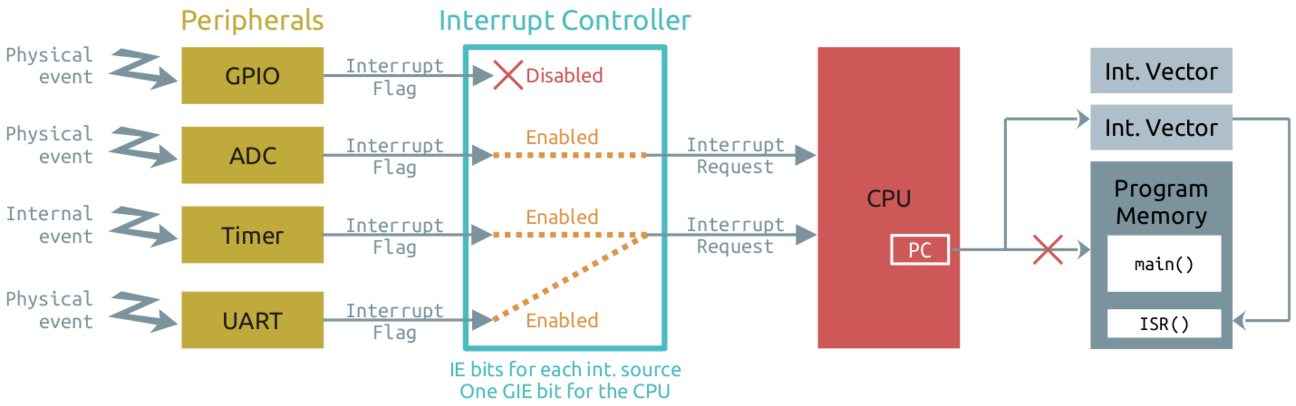
III. Utiliser les interruptions

III.1. Rappels sur les interruptions

Comme tout micro-contrôleur qui se respecte, les STM32 utilisent un mécanisme d'interruption.

Pour rappel, le principe des interruptions est de laisser les périphériques « capter » un évènement, auquel cas ceux-ci lèvent un **IF (*Interrupt Flag*)**. Si ces évènement sont autorisés à interrompre le programme en cours d'exécution, un IF devient une **IRQ (*Interrupt Request*)**, signal électrique relié au CPU. Lorsqu'une IRQ arrive au CPU, ce dernier met en pause le programme en cours pour aller exécuter une portion de code présente dans une zone mémoire précise appelée ***Interrupt Vector***.

Généralement, ce vecteur ne contient qu'un simple appel à une fonction chargée du traitement de l'évènement qui vient de se produire : l'**ISR (*Interrupt Service Routine*)**.



Pour les cœurs ARM (comme celui qui équipe les STM32), le contrôleur d'interruption s'appelle **NVIC (*Nested Vector Interrupt Controller*)**.

Les GPIO configurées en entrée peuvent déclencher une interruption suite un changement de niveau logique. Ceci se configure grâce au **EXTI (*External Interrupts*) *Controller***, un sous-étage du NVIC. Dans cette section nous allons configurer l'entrée du bouton poussoir de sorte à ce qu'elle déclenche une interruption.

III.2. Préparer les interruptions

Ouvrez le fichier `*.ioc` de votre projet. Ceci rouvre l'interface de configuration graphique du micro-contrôleur.

💻 Sur la vue du composant « *Pinout view* », cliquez sur la broche du bouton poussoir.

✎ Dans quel mode est-elle configurée ? Vous noterez que ce n'est pas `GPIO_Input`.

💻 Sur l'onglet de gauche : *System Core* → *GPIO*, cliquez sur cette broche dans le tableau.

La broche est en mode `External Interrupt Mode with Falling edge trigger detection`. En d'autres termes, la broche permet de déclencher une interruption externe (sous-entendu externe au circuit intégré du MCU) lorsque qu'elle détecte un front descendant. Dans notre cas, cela permet de déclencher une interruption lors d'un appui sur le bouton poussoir.

Autorisons les broches EXTI (*External Interrupts*) à déclencher des interruptions.

💻 Dans `System Core → NVIC` (et non dans `System Core → GPIO`, attention!), cochez la ligne `EXTI line 4 to 15 interrupts`.

Dans les cœur ARM, le niveau de priorité le plus élevé est le '0'. Nous demandons que les EXTI soient de priorité plus faible que le `HAL_Delay()` (qui lui est de niveau '0').

💻 Toujours dans `System Core → NVIC`, affectez la valeur '1' au champ `Preemption Priority`.

💻 Cliquez sur `Project → Generate Code` (ou l'icône ⚙) pour générer le code associé.

Logiquement le configurateur de code STM32CubeMX vient de modifier le code existant afin d'y intégrer les paramètres que vous avez tout juste demandés. Allons vérifier les changements obtenus.

Dans la fonction `MX_GPIO_Init()` du fichier `main.c`, ces instructions viennent d'être ajoutées :

```
/* EXTI interrupt init*/
HAL_NVIC_SetPriority(EXTI4_15_IRQn, 1, 0);
HAL_NVIC_EnableIRQ(EXTI4_15_IRQn);
```

✎ À quoi correspondent-elles ?

OPTIONAL

III.3. Utiliser les interruptions

Dans le fichier `stm32l0xx_hal_gpio.c`, la fonction `HAL_GPIO_EXTI_IRQHandler()` correspond à la routine d'interruption (ISR) de l'interruption déclenchée par un changement d'état sur une GPIO. C'est elle qui est déclenchée lorsqu'une requête d'interruption (IRQ) est faîte. En regardant le corps de cette fonction, on remarque qu'elle *clear* l'interruption (aquittement) puis qu'elle appelle une autre fonction : `HAL_GPIO_EXTI_Callback()`.

En effet, c'est la fonction `HAL_GPIO_EXTI_Callback()` qui sera celle utilisable par le développeur (vous) pour traiter l'évènement en question. Cette fonction est d'ailleurs déclarée quelques lignes plus loin :

`__weak void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)`

Le mot-clé `__weak` laisse à l'utilisateur la possibilité de redéfinir la fonction ailleurs, ce qui serait autrement impossible en C. Ainsi nous serons en mesure de redéfinir cette fonction pour l'utiliser à notre guise.

💻 Dans le fichier `main.c`, copiez-collez le code suivant entre les balises indiquées.

```
/* USER CODE BEGIN 0 */
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    if(GPIO_Pin == B1_Pin) {
        for( int i = 0 ; i < 10 ; i++ ) {
            HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
            HAL_Delay(50);
        }
    }
}
/* USER CODE END 0 */
```

Vous remarquerez qu'il s'agit donc d'une redéfinition de la fonction de callback (rendue possible grâce au mot-clé `__weak`), afin de l'adapter à nos besoins.

✎ Que fait cette fonction ?

💻 Programmez le MCU cible et validez visuellement le fonctionnement.

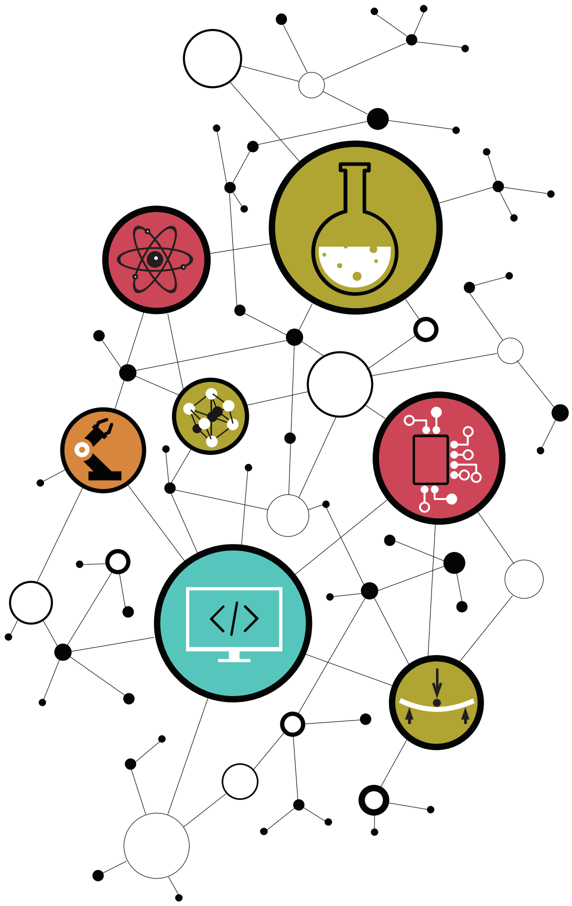
Vous devez constater que l'application de base (clignotement toutes les deux secondes) est toujours exécutée, mais qu'elle est interrompue par une autre portion de code déclenchée par l'appui sur le bouton poussoir.

Bilan

Le mécanisme d'interruption, chose assez obscure étudiée en première année, a ici été repassé en quelques minutes. Certes cette configuration automatique n'a pas forcément aidé à mieux comprendre le mécanisme, mais l'abstraction apportée par CubeMX permet encore une fois de gagner un temps précieux pour le développement d'une application.

CHAPITRE 4

PÉRIPHÉRIQUE UART



I. Configuration du périphérique UART

Le périphérique **UART** (pour ***Universal Asynchronous Receiver-Transmitter***) est un composant matériel permettant d'assurer une **liaison série asynchrone**. Malgré sa disparition chez le grand public, il reste très répandu dans les systèmes embarqués afin de proposer une interface de debug simple matériellement et logiciellement parlant. C'est aussi la raison pour laquelle nous utiliserons ce périphérique pendant les TP.

💻 Reprenez le projet déjà créé et ouvrez le fichier de configuration `*.ioc`.

✎ Parcourez le menu de configuration du périphérique UART (`Connectivity → USART2`) et retrouvez les paramètres utilisés par défaut :

- Baud rate :
- Taille de la donnée :
- Nombre de bits de stop :
- Bit de parité :
- Broche du signal Tx :
- Broche su signal Rx :

Nous n'avons rien modifié, il n'y a pas besoin de re-générer le code. Vous pouvez tout simplement fermer le fichier `*.ioc`.

> Pour rappel, tous ces paramètres ont été imposés en moins de deux minutes par STM32CubeMX lorsque vous avez créé le projet.
>
> 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 encore, le ARM Cortex-M0+ est un des plus simples MCU de ARM !

II. Étude du projet généré

Au chapitre précédent, parmi les fichiers générés par STM32CubeMX, nous nous étions concentrés sur les fichiers liés aux GPIOs. Maintenant abordons les fichiers liés à l'UART.

💻 Ouvrez le fichier `main.c` puis parcourez la fonction `main()`. On remarque l'appel à quelques fonctions d'initialisation, et une boucle `while(1)` (initialement vide, mais qui a été remplie).

💻 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 relevés 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. UART en transmission

III.1. Fonction de transmission

Observons la définition de la fonction de transmission, dans le fichier `stm32l0xx_hal_uart.c`.

```
HAL_StatusTypeDef HAL_UART_Transmit(
   UART_HandleTypeDef *huart, const uint8_t *pData, uint16_t Size, uint32_t Timeout)
```

La fonction est de type `HAL_StatusTypeDef`. Il s'agit d'une redéfinition de type (`typedef`) qui représente un code d'erreur de la fonction. Ceci permet ainsi d'indiquer si la fonction s'est bien exécutée ou non. À titre d'exemple, la fonction peut renvoyer les valeurs `HAL_OK`, `HAL_BUSY`, `HAL_TIMEOUT`, …

Le premier argument est un `UART_HandleTypeDef*`. Là encore il s'agit d'une redéfintion de type (typedef) qui contient le handle d'un UART. Pour faire simple, un handle permet de désigner une ressource (ici un périphérique USART) puisque plusieurs peuvent utiliser cette même fonction. Dans le `main.c`, on retrouve le handle associé à l'USART2 :

```
/* Private variables ---------------------------------------------------------*/
UART_HandleTypeDef huart2;
/* USER CODE BEGIN PV */
```

Le deuxième argument est un pointeur vers une chaîne de caractères. Il s'agit donc du message que l'on veut transmettre, soit sous forme d'une chaîne de caractères, soit sous forme d'un tableau de caractères.

Le troisième argument est la quantité de données à envoyer. En reprenant la chaîne de caractères, on pourrait simplement utiliser la fonction `strlen(la_chaine)` ici.

Enfin, le dernier argument correspond au temps accordé à la fonction pour effectuer la transmission. Si celle-ci dépasse ce temps, alors la fonction s'arrête en cours d'envoi (et retourne la valeur `HAL_TIMEOUT`). Si on ne souhaite pas que la fonction soit limitée dans le temps, on peut utiliser la macro `HAL_MAX_DELAY`.

💻 Effacez le contenu de la boucle `while(1)` de la fonction `main().`

💻 Écrivez maintenant le code suivant à l'intérieur de la boucle `while(1)` :

```
  /* USER CODE BEGIN WHILE */
  while (1)
  {
     HAL_UART_Transmit( &huart2, (uint8_t*)"Hi\r\n", 4, HAL_MAX_DELAY );
     HAL_Delay(2000);
    /* USER CODE END WHILE */
```

💻 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.2. STLink Virtual COM Port

La carte NUCLEO embarque certes le micro-contrôleur cible pour nos applications, 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.

III.3. Vérification par terminal série

💻 Ouvrez le terminal série de votre choix (Teraterm, PuTTY, GTKTerm, …). Configurez-le de sorte à échanger avec le STLink VCP (vous avez relevé les paramètres de votre périphérique UART).

🗗 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.

IV. UART en réception

IV.1. 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()` ? Vous noterez qu'ils sont très (très) proches de ceux utilisés pour la fonction de transmission.

💻 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.

```
/* USER CODE BEGIN Includes */
#include <string.h>                  // strlen()
/* USER CODE END Includes */

...
...

  /* USER CODE BEGIN 2 */
  char* init_text = "\r\n### Application starting ###\r\n";
  HAL_UART_Transmit( &huart2, (uint8_t*)init_text, strlen(init_text), HAL_MAX_DELAY );
  /* USER CODE END 2 */
```

💻 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.

```
  /* USER CODE BEGIN WHILE */
  char data;
  while (1)
  {
     HAL_UART_Receive(  &huart2, (uint8_t*)&data, 1, HAL_MAX_DELAY );
     HAL_UART_Transmit( &huart2, (uint8_t*)&data, 1, HAL_MAX_DELAY );
    /* USER CODE END WHILE */
```

🗗 Validez le fonctionnement avec votre terminal.

OPTIONAL

IV.2. Réception non-bloquante (interruption)

IV.2.a. Principe des interruptions avec l'USART

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.

💻 Ouvrez le fichier de configuration `*.ioc` afin d'activer les interruptions pour le périphérique USART2 :

- Connectivity → USART2 → NVIC Settings → USART2 global tnterrupt : Enabled (checkbox)

💻 Cliquez sur `Project → Generate Code` (ou l'icône) pour générer le code associé.

Le mécanisme d'interruption ayant déjà été abordé dans la partie III.1 Rappels sur les interruptions page 20 (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.

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[5]. 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.

---

[5] Fonction `UART_RxISR_8BIT()` définie dans le fichier `stm32l0xx_hal_uart.c`.

IV.2.b. Mise en œuvre

Vous allez maintenant réaliser le même programme d'echo que précédemment (tout caractère reçu est immédiatement renvoyé), mais en n'utilisant que le mécanisme des interruptions.

💻 Videz le contenu de la boucle `while(1)` du `main()`.

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

💻 Dans le `main.c` (avant la fonction `main()`), déclarez une variable globale qui contiendra la donnée reçue (c'est pour l'instant le seul moyen de partager une donnée avec une fonction interruption).

```
/* USER CODE BEGIN PV */
char data;
/* USER CODE END PV */
```

💻 Dans la fonction `main()`, envoyez un message d'initialisation puis lancez la lecture d'un caractère par interruption. La donnée reçue sera stockée dans la variable préalablement déclarée.

```
  /* USER CODE BEGIN 2 */
  char init_text = "\r\n### Application starting ###\r\n";
  HAL_UART_Transmit( &huart2, (uint8_t*)init_text, strlen(init_text), HAL_MAX_DELAY );

  HAL_UART_Receive_IT( &huart2, (uint8_t*)&data, 1 );
  /* USER CODE END 2 */
```

💻 Avant la fonction `main()`, re-définissez la fonction de callback `HAL_UART_RxCpltCallback()`, laquelle se déclenche à chaque réception de caractère (si la fonction `HAL_UART_Receive_IT()` a été appelée avant, évidemment). Dans cette fonction, le caractère tout juste reçu doit être re-transmis à l'UART, puis une nouvelle réception par interruption doit être lancée.

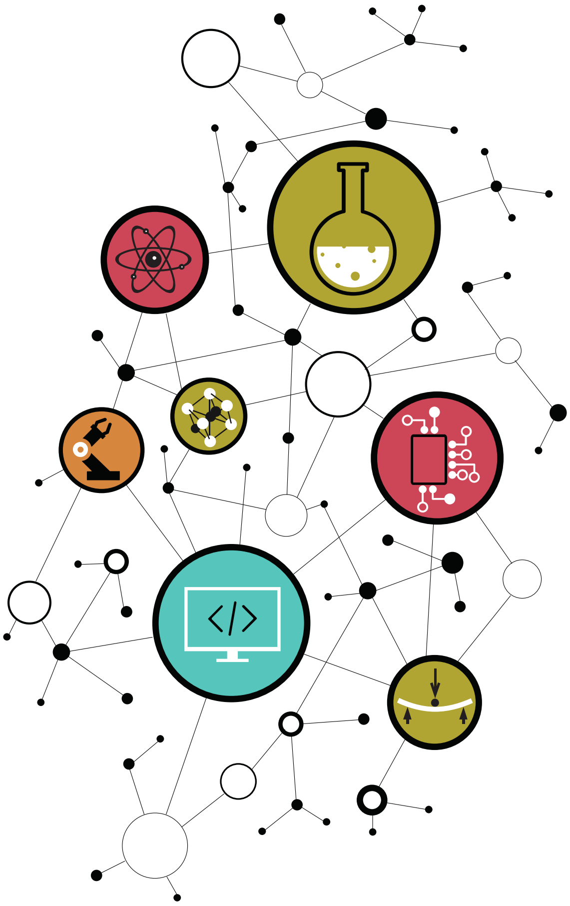
```
/* USER CODE BEGIN 0 */
void HAL_UART_RxCpltCallback( UART_HandleTypeDef *huart )
{
   HAL_UART_Transmit( &huart2, (uint8_t*)&data, 1, HAL_MAX_DELAY );
   HAL_UART_Receive_IT( &huart2, (uint8_t*)&data, 1 );
}
/* USER CODE END 0 */
```

💻 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.

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

CHAPITRE 5

GLOSSAIRE



Termes classés par thématique

LSb / MSb	*Least/Most Significant bit*
LSB / MSB	*Least/Most Significant Byte*

MCU	***Microcontroller Unit***
CPU	*Central Processing Unit*
Périphérique	Composant matériel interne au MCU, conçu pour une fonction précise
GPIO	*General Purpose Input/Output*, périphérique pour manipuler les broches

STM	***STMicroelectronics***, société franco-italienne de semi-conducteurs
STM32	Gamme de MCU 32-bits de ST, basée sur des CPU ARM Cortex-M cores
STM32CubeIDE	Environnement de développement intégré pour processeurs ST
STM32CubeMX	Configurateur graphique de code d'initialisation pour processeurs ST
HAL	*Hardware Abstraction Layer*, fonctions d'utilisation du matériel
STLink VCP	*STLink Virtual Com Port,* port COM virtuel accessible via l'USB

NVIC	*Nested Vector Interrupt Controller*, contrôleur d'interruption des CPU ARM
IF	*Interrupt Flag*, indicateur d'occurrence d'un évènement
IRQ	I*nterrupt Request*, demande au CPU d'interrompre le programme
ISR	*Interrupt Service Routine*, fonction exécutée lors d'une interruption
Callback	Fonction appelée par l'ISR et dont la définition est laissée au développeur

UART	***Universal Asynchronous Receiver-Transmitter**,* périphérique de comm série
USART	*Universal Synchronous-Asynchronous Receiver-Transmitter*
Tx	*Transmit line*
Rx	*Receive line*