

# CHAPITRE 4

## Version 5.0 CubeMX

# SYSTÈME ARM STM32F407

(ARCHITECTURE ARM)  
(STM32CUBEMX)  
(PÉRIPHÉRIQUES)  
(SYSTEME DE DEVELOPPEMENT)

4.1	INTRODUCTION AU PROCESSEUR ARM .....	3
4.1.1	L'architecture ARM .....	3
4.1.2	La gamme de produit ST.....	4
4.1.3	Le résumé du processeur STM32F407.....	5
4.1.4	Le cœur et les registres .....	6
4.1.5	L'adressage de la mémoire.....	8
4.1.6	Le pipeline .....	9
4.1.7	Les interruptions et la gestion des exceptions.....	10
4.1.8	Les modes d'alimentation .....	11
4.1.9	Le CMSIS.....	13
4.1.10	Le Schéma bloc du STM32F407 .....	15
4.2	LE STM32CUBEMX.....	16
4.2.1	Les caractéristiques de base.....	16
4.2.2	Une description du produit .....	16
4.2.3	Le STM32CubeMx... un logiciel d'intégration.....	18
4.2.3.1	La couche matérielle.....	18
4.2.3.2	La couche logicielle de bas niveau .....	19
4.2.3.3	La couche logicielle de gestion des périphériques complexes.....	19
4.2.3.4	La couche application.....	21
4.2.4	Les pilotes HAL .....	21
4.2.4.1	Les fichiers « DRIVERS » (pilotes de périphérique) .....	22
4.2.4.2	Les fichiers « USER » (réservés aux utilisateurs) .....	23
4.2.5	Les structures de données associées aux librairies HAL .....	24
4.2.5.1	La structure de données des gestionnaires de périphériques .....	24
4.2.6	Les règles d'utilisation des librairies HAL .....	25
4.2.7	Exemple d'utilisation, (un premier projet...clignotement d'une del) .....	26
4.2.8	Acronymes et définitions.....	33
4.3	LES PÉRIPHÉRIQUES DU MCU.....	34
4.3.1	Les GPIO .....	37
4.3.2	Les INT externes .....	40
4.3.3	Les timers.....	43
4.3.4	Les PWM.....	45
4.3.5	Les ADC.....	47
4.3.6	Les USART .....	49
4.3.7	La communication I2C .....	51
4.3.8	La communication SPI .....	53
4.3.9	La communication CAN .....	55
4.3.10	La communication USB (USB DEVICE).....	58
4.3.11	Clef USB en disque (USB HOST) .....	61
4.3.12	Les fonctions I/O et la carte multi-I/O de 5e session .....	66
4.3.13	L'écran graphique KS0108 .....	68
4.4	INTRODUCTION AU SYSTÈME DE DÉVELOPPEMENT .....	69
4.4.1	Les schémas .....	70
4.4.2	Les couches de la plaquette de circuit.....	64
4.4.3	Les connecteurs.....	67
4.4.4	Les fonctions .....	72

## 4.1 INTRODUCTION AU PROCESSEUR ARM

---

### 4.1.1 L'architecture ARM

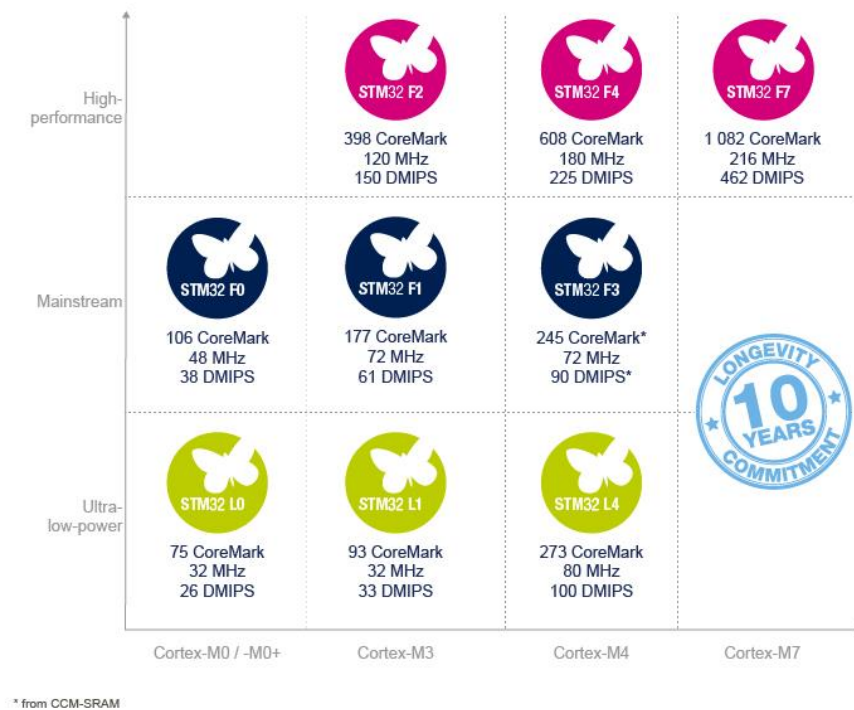
Les architectures ARM sont des architectures matérielles RISC 32 bits (ARMv1 à ARMv7) et 64 bits (ARMv8) créées par ARM Ltd depuis 1990. Elles ont été initialement développées par la société britannique Acorn Computers, qui les utilisa à partir de 1987 dans sa gamme d'ordinateurs 32 bits Archimedes. ARM signifiait alors « Acorn Risc Machine ». Ultérieurement, la division création de microprocesseurs d'Acorn fut détachée de la société mère et devint la société « Advanced Risc Machine limited », se positionnant avec une offre indépendante pour le marché de l'électronique embarquée.

Dotés d'une architecture relativement plus simple que d'autres familles de processeurs, et bénéficiant d'une faible consommation, les processeurs ARM sont devenus dominants dans le domaine de l'informatique embarquée, en particulier la téléphonie mobile et les tablettes.

Ces processeurs sont fabriqués sous licence par un grand nombre de constructeurs.

Aujourd'hui, ARM est surtout connu pour ses SoC (System on Chip), intégrant sur une seule puce, MCU (microprocesseur), GPU (Graphic Processor Unit), DSP (Digital Signal Processor), FPU (Floating Point Unit), et contrôleur de périphériques. Les SoC sont présents dans la majorité des smartphones et tablettes. ARM propose des architectures qui sont vendues sous licence de propriété intellectuelle aux concepteurs. Ils offrent différentes options dans lesquelles les constructeurs peuvent prendre ce qui les intéresse pour compléter avec leurs options propres ou de concepteurs tiers. ARM propose ainsi, pour les SoC les plus récents, les microprocesseurs Cortex (Cortex-A (Appliance), Cortex-M (Microcontrôleur), Cortex-R (Real Time), ainsi que les divers autres composants nécessaires à la composition du SoC complet. Certains constructeurs préfèrent produire leur propre processeur graphique, d'autres, préfèrent prendre dans certains cas un processeur graphique de prestataire tiers ou d'ARM selon les modèles, et d'autres, modifient certains composants du microprocesseur en mélangeant plusieurs architectures processeur ARM.

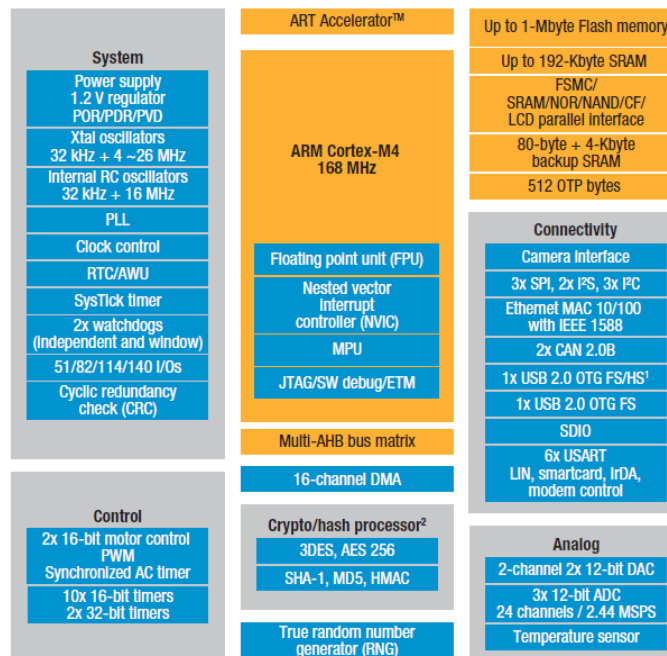
### 4.1.2 La gamme de produit ST



La définition des codes de produits se fait comme suit.

STM32	Product type	Series	Line	Pine count	Memory
	F : Foundation	0 : Entry	XX	A : 80	4 : 16
	L : Low power	1 : Mainstream		B : 208	6 : 32
		2 : Hi Perf		C : 48	8 : 64
		3 : Analog/DSP		E : 24	B : 128
		4 : Hi Perf/DSP		F : 20	E : 512
		7 : Very Hi Perf		G : 28	G : 1024
				I : 176	I : 2048
				K : 32	
				M : 80	
				N : 216	
				R : 64	
				V : 100	
				Z : 144	

### 4.1.3 Le résumé du processeur STM32F407



Cœur :

- Un cœur ARM Cortex-M4F cadencé à 168 MHz.

Mémoire :

- La RAM statique de 192 KB est composée, entre autre, de 112 KB de SRAM1, de 16 KB de SRAM2, de 64 KB de CCM et de 4 KB pouvant être alimentée par batterie.
- La flash consiste en des agencements de 512 / 1024 / 2048 KB dont 30 KB pour le système de démarrage, 512 bytes de programmation unique (OTP) et de 16 octets pour des options.
- Tous les circuits ont un code identifiant de 96 bits gravé en usine.

Périphériques :

- Les périphériques intégrés dans les circuits de la famille STM32F4 sont : le USB 2.0 OTG HS et FS, deux CAN 2.0B, un SPI + deux SPI ou un full-duplex I<sup>2</sup>S, trois I<sup>2</sup>C, quatre USART, deux UART, SDIO pour cartes SD/MMC, douze timers 16-bit, deux timers 32-bit, deux watchdog timers, un capteur de température interne, trois ADC, deux DAC, 51 à 140 GPIO, seize DMA, un RTC, Les circuits les plus gros offrent aussi la possibilité d'expansion mémoire parallèle sur bus de 8 ou 16 bits.
- Le STM32F4x7 offre l'Ethernet MAC et une interface pour camera.

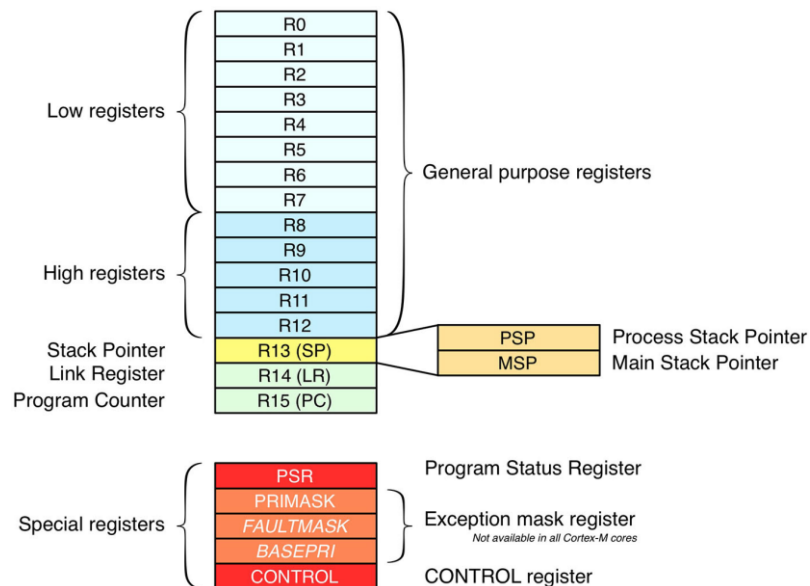
Tension d'opération va de 1,8 à 3,6 volts.

Architecture de 32 bits, Harvard (bus d'instruction distinct du bus de données) et pipeline d'instruction à trois niveaux.

#### 4.1.4 Le cœur et les registres

À l'image de toutes les architectures RISC, les processeurs Cortex-M sont de type « load/store » ce qui signifie qu'ils effectuent des opérations de chargement et stockage sur les registres du CPU. Ce sont donc des instructions grandement utilisées pour transférer des données entre les registres du CPU et les emplacements de mémoire.

La figure suivante montre les principaux registres du Cortex-M. Certains d'entre eux ne sont disponibles que dans les séries plus performantes : M3, M4 et M7. Les registres R0 à R12 sont des registres à usage général et peuvent être employés comme opérandes dans les instructions ARM. Certains registres à usage général peuvent être utilisés par le compilateur comme registres à fonctions spéciales. Le registre R13 est utilisé comme pointeur de pile (SP). Les modifications apportées au contenu de ce registre dépendent du mode du CPU (privilegié - non privilégié). Ces modes sont généralement utilisés par les « Real Time Operating Systems » (RTOS) pour faire le changement de contexte.



Par exemple, considérons le code « C » suivant utilisant les variables locales 'A', 'B' et 'C'.

```
...
uint8_t a,b,c;
a = 3;
b = 2;
c = a * b;
...
```

Le compilateur générerait ce code assembleur :

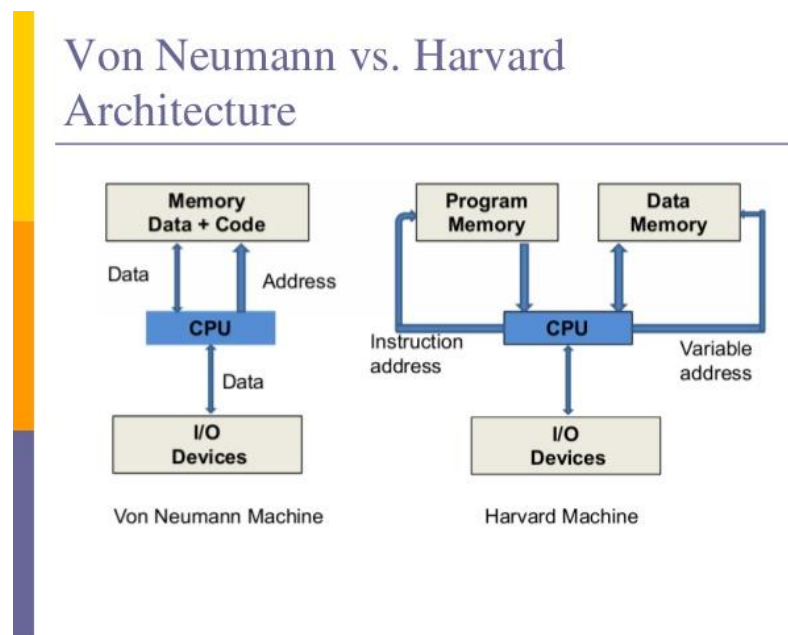
```

1 movs r3, #3 ;      déplace "3" dans le registre r3
2 strb r3, [r7, #7] ; emmagasine le contenu de r3 in "a"
3 movs r3, #2 ;      déplace "2" dans le registre r3
4 strb r3, [r7, #6] ; emmagasine le contenu de r3 in "b"
5 ldrb r2, [r7, #7] ; charge le contenu de "a" in r2
6 ldrb r3, [r7, #6] ; charge le contenu de "b" in r3
7 smulbb r3, r2, r3 ; multiplie "a" et "b" et emmagasine dans r3
8 strb r3, [r7, #5] ; emmagasine le résultat dans "c"

```

Comme nous pouvons le voir, toutes les opérations impliquent l'utilisation d'un registre. Les instructions aux lignes 1-2 permettent de placer la valeur immédiate 3 dans le registre r3 et stockent son contenu (la valeur 3) à l'intérieur de l'emplacement mémoire indiqué par le registre r7 plus un décalage de 7 (ce qui représente l'emplacement en mémoire de la variable 'a'). On retrouve le même scénario aux lignes 3 et 4 alors que la variable 'b' est placée en mémoire à l'adresse pointée par le registre r7 et un décalage de 6. Ensuite, les lignes 5-7 permettent de charger le contenu des variables a et b dans les registres r2 et r3 avant d'effectuer une multiplication dont le résultat est retourné dans le registre r3. Enfin, la ligne 8 effectue l'opération de stockage à l'emplacement mémoire de la variable 'c'.

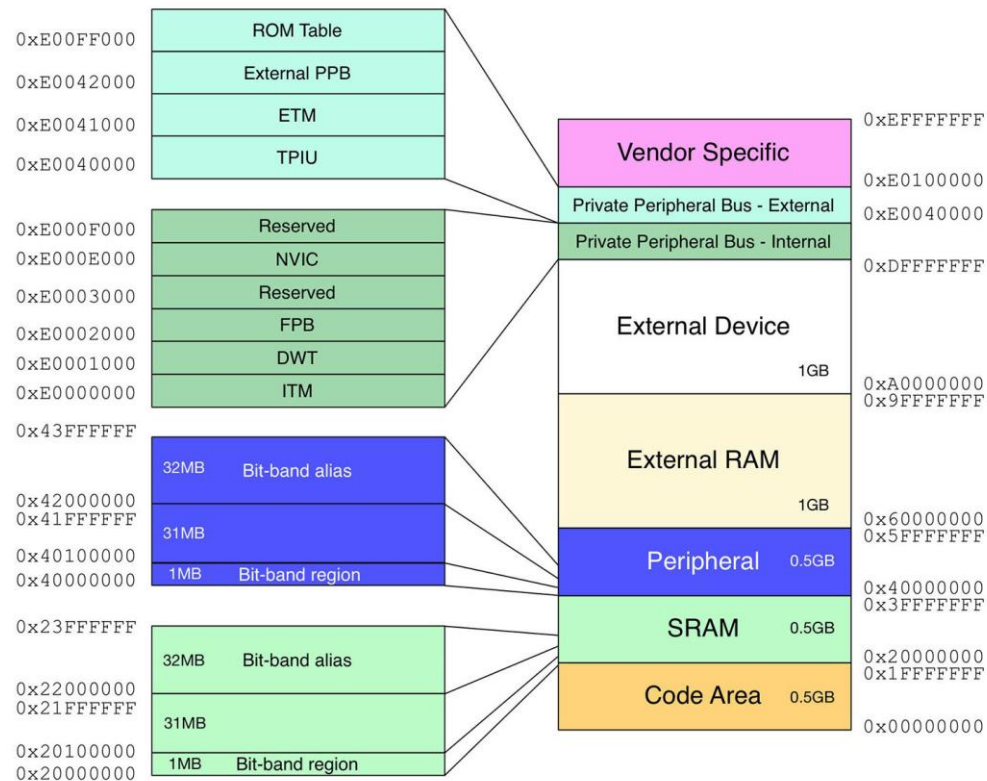
Mentionnons que le cœur ARM possède une architecture Harvard qui, contrairement à l'architecture Von Neumann, permet la lecture d'instruction en parallèle à la lecture et à l'écriture en mémoire de donnée.



Cette séparation des bus d'instruction (I-Bus) et du système (S-Bus) est très avantageuse et permet d'accélérer le traitement des processus du cœur Cortex-M4.

#### 4.1.5 L'adressage de la mémoire

ARM définit un espace d'adressage mémoire standard à tous les cœurs Cortex-M, ce qui permet la portabilité du code entre les différents fabricants de puces de silicium. L'espace d'adressage est de 4 Go et il est organisé en plusieurs sous-régions avec des fonctionnalités logiques différentes. L'agencement de mémoire d'un processeur Cortex-M ressemble à ceci :



Tous les Cortex-M ont une zone de mémoire code qui débute à l'adresse 0x0000 0000. Cette zone comprend également le pointeur vers le début de la pile (généralement placée dans la SRAM) et la table des vecteurs d'interruptions. La position de la zone de code est standardisée pour tous les fournisseurs Cortex-M, quoique l'architecture de base soit suffisamment souple pour permettre aux fabricants d'organiser ce domaine de bien des manières différentes, à l'image des besoins qu'ils ont. Pour tous les appareils STM32 la zone à partir de l'adresse 0x0800 0000 est liée à la mémoire interne MCU FLASH et elle représente la zone où le code de programme réside.

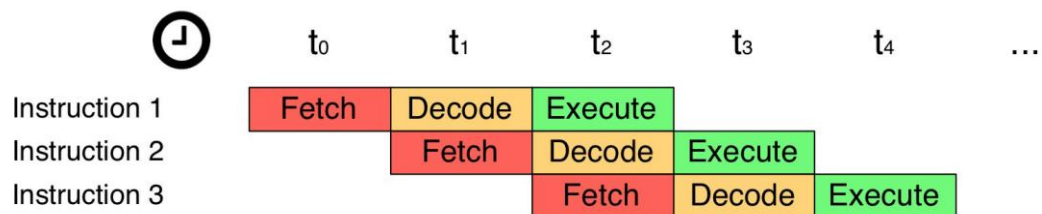
Suivent, dans l'ordre, l'espace mémoire SRAM interne, l'espace pour les périphériques, pour la RAM externe, pour les dispositifs externes, pour les périphériques privés interne et externe et une zone réservée pour le fabricant du circuit SoC.



#### 4.1.6 Le pipeline

Chaque fois que nous parlons de l'exécution d'une instruction, nous supposons une série de détails non négligeables. Avant qu'une instruction ne soit traitée, le CPU doit aller chercher cette instruction en mémoire (FETCH), la décoder (DECODE) et l'exécuter (EXECUTE). Ce procédé consomme du temps (nombre de cycles processeur) qui dépend de l'organisation et du type de mémoire et de l'architecture centrale du processeur.

Les processeurs modernes ont introduit un moyen de paralléliser ces opérations pour augmenter le débit d'instruction (le nombre d'instructions qui peuvent être exécutées dans une unité de temps). Le cycle d'instruction de base est divisé en une série d'étapes comme si les instructions voyageaient le long d'un pipeline. Plutôt que de traiter chaque séquence d'instructions une à la fois, c'est-à-dire, lire, décoder et exécuter l'instruction avant d'en traiter une autre, l'instruction est subdivisée en séquence afin que les différentes étapes puissent être exécutées en parallèle.



Tous les microcontrôleurs basés sur le Cortex-M présentent une forme de pipelining. Le plus commun est le pipeline en 3 niveaux. Cette forme de pipeline est soutenue par les Cortex-M0, M3 et M4. Les Cortex-M0+ fournissent un pipeline à 2 niveaux car, même si le pipelining permet une économie de temps, il présente un coût énergétique. Le pipeline est donc minimisé car les MCU à noyau Cortex-M0+ sont dédiés à des applications de faible puissance. Les noyaux Cortex-M7 fournissent un pipeline à 6 niveaux.

#### 4.1.7 Les interruptions et la gestion des exceptions

Le cortex-M possède un puissant gestionnaire des interruptions / exceptions. Ces dernières sont des événements asynchrones qui modifient le déroulement du programme. Quand une interruption ou une exception se produit, le CPU suspend l'exécution de la tâche en cours, enregistre son contexte (sur son pointeur de pile) et commence l'exécution d'une routine conçue pour gérer l'événement. Cette routine est appelée gestionnaire d'exceptions en cas d'exceptions et routine de service d'interruption (ISR) dans le cas d'une interruption. Après que l'exception ou d'interruption ait été traitée, le CPU reprend le contexte précédent et exécute le flux d'instructions abandonné à l'arrivée de l'évènement.

Les interruptions sont généralement générées à partir de périphériques intégrés (par exemple une minuterie) ou à partir des entrées externes (par exemple, un commutateur tactile relié à un GPIO). Dans certains cas, elles peuvent être déclenchées par le logiciel. Les exceptions sont, quant à elles, liées à l'exécution du logiciel et le CPU lui-même peut générer une exception. Ces événements pourraient être une faute (par exemple une tentative d'accéder à un emplacement non valide de mémoire) ou un événement généré par le système d'exploitation.

Chaque interruption / exception a un numéro qui l'identifie de façon unique. Le tableau suivant montre les interruptions / exceptions fixes, communes à tous les cœurs Cortex-M. Le numéro reflète la position de la routine du gestionnaire NVIC dans la table vectorielle, où l'adresse effective de la routine est stockée. Par exemple, la position 15 contient l'adresse mémoire de la zone de code contenant le gestionnaire de l'exception sysTick, interruption générée lorsque le compteur atteint zéro.

No.	Type	Priorité	Description
1	Reset	-3 (Plus élevée)	Reset
2	NMI	-2	Exception Non Masquable
3	Hard Fault	-1	Toutes les exceptions matérielles
4	MemManageFault	Programmable	Exception du à la gestion mémoire
5	BusFault	Programmable	Erreur matérielle du au bus
6	UsageFault	Programmable	Erreur du au programme
7-10	Reserved	Pas utilisée	—
11	SVC	Programmable	Appel système
12	DebugMonitor	Programmable	Moniteur de mise au point
13	Reserved	Pas utilisée	—
14	PendSV	Programmable	Requête de service d'exception
15	SYSTICK	Programmable	Timer interne cortex M4

Il est possible d'attribuer à chaque interruption / exception un niveau de priorité qui définit l'ordre de traitement à respecter lors d'évènements simultanés. Par contre, il est impossible de le faire pour les trois premiers types d'interruptions. Plus le nombre de priorités est élevé, plus la priorité est faible. Par exemple, supposons que nous ayons deux routines d'interruption liées aux entrées externes A et B et que l'on attribue une priorité plus élevée (nombre inférieur) à l'entrée A. Si l'interruption liée à A arrive lorsque le processeur est au service de l'interruption de l'entrée B, l'exécution de B est suspendue, ce qui permet à la routine de service d'interruption de priorité plus élevée, dans ce cas-ci la A, de s'exécuter immédiatement.

Les interruptions / exceptions sont traitées par une unité dédiée appelée « Vectored Interrupt Controller » (NVIC). Le NVIC présente les caractéristiques suivantes:

- **exception flexible / gestion d'interruption** : Le NVIC est capable de traiter à la fois des signaux d'interruption ou des demandes provenant de périphériques et des exceptions provenant du processeur central. Toutes les interruptions / exceptions peuvent être activées / désactivées par logiciel.
- **exception imbriquée / support d'interruption** : NVIC permet l'attribution de niveaux de priorité aux exceptions et interruptions (sauf pour les trois premiers types d'exception), ce qui donne la possibilité de classer les interruptions selon les besoins de l'utilisateur.
- **vecteur exception / entrée d'interruption** : NVIC localise automatiquement la position du gestionnaire d'exception lié à une exception / interruption, sans utiliser un code supplémentaire.
- **interruption masquable** : les développeurs sont libres de suspendre l'exécution de tous les gestionnaires d'exceptions (sauf pour le NMI), ou de suspendre certains d'entre eux sur la base du niveau de priorité, grâce à un ensemble de registres dédiés. Ceci permet l'exécution des tâches essentielles d'une manière sûre.
- **latence d'interruption** : une caractéristique intéressante du NVIC est le temps de latence de traitement d'interruption, qui est égale de 12 cycles pour tous les Cortex-M3 et M4, de 15 cycles pour Cortex-M0 et de 16 cycles pour Cortex-M0+.
- **relocalisation des gestionnaires d'exception** : les gestionnaires d'exception peuvent être déplacés en mémoire FLASH, ainsi que dans tout autre emplacement mémoire (à l'exception de la mémoire en lecture seule). Cela offre un grand degré de flexibilité pour les applications avancées.

Les processeurs Cortex-M peuvent fournir une minuterie système, également connue sous le nom sysTick. Tous les dispositifs STM32 fournissent un sysTick basé sur un timer de 24 bits utilisé pour fournir une tique du système pour les systèmes d'opération à temps réel (RTOS), comme FreeRTOS. Le sysTick est utilisé pour générer des interruptions périodiques à des tâches planifiées. Les programmeurs peuvent définir la fréquence de mise à jour de la minuterie sysTick par le contrôle des registres. Le sysTick timer est également utilisé par le STM32 HAL pour générer des retards précis, même sans l'utilisation d'un RTOS.

#### 4.1.8 Les modes d'alimentation

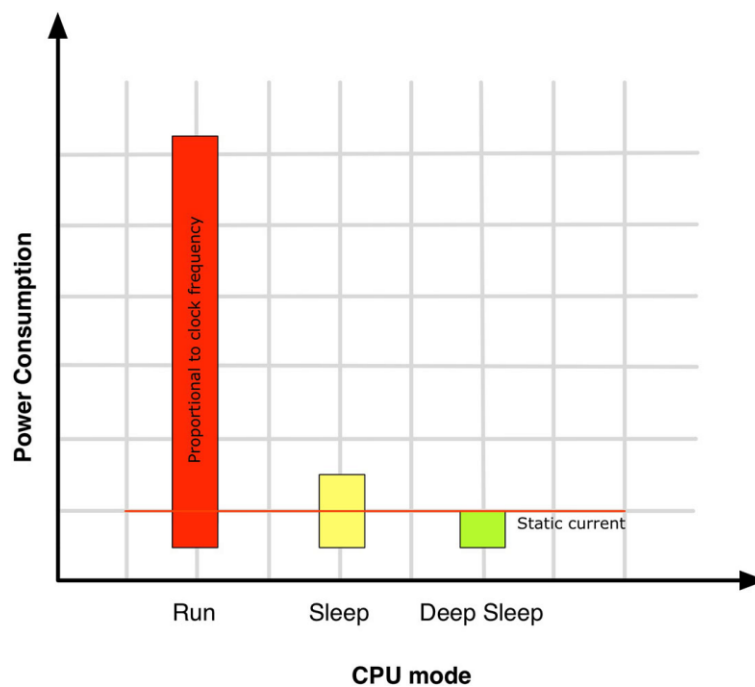
Le défi actuel dans l'industrie de l'électronique, en particulier en ce qui a trait à la conception d'équipement mobile, est en lien avec la gestion d'énergie. Réduire la puissance au niveau minimum est l'objectif principal de tous les concepteurs de matériel et des programmeurs impliqués dans le développement de dispositifs alimentés par batterie. Les processeurs de la famille Cortex-M offrent plusieurs niveaux de gestion de l'alimentation, que l'on peut diviser en deux groupes principaux : les caractéristiques intrinsèques et les modes de puissance définis par l'utilisateur.

Les caractéristiques intrinsèques font référence aux fonctionnalités natives liées à la consommation d'énergie définies lors de la conception à la fois le noyau Cortex-M et de l'ensemble du MCU. Par exemple, le Cortex-M0 possède seulement deux étages de pipeline afin de réduire la consommation d'énergie liée aux traitements des instructions. Un autre élément important lié à la gestion de l'alimentation se situe au niveau de la densité du code. Ainsi, le jeu d'instruction Thumb-2 offre la possibilité de réduire l'espace mémoire et par le fait même, la consommation de courant du dispositif SoC.

Traditionnellement, les processeurs Cortex-M offrent plusieurs modes d'alimentation programmable à l'utilisateur. Ce dernier n'a qu'à configurer le SoC à travers le contrôle de registres de commande de système (SCR). Le premier mode est le mode « RUN » où le MCU est cadencé à ses pleines capacités. En mode « RUN » la consommation d'énergie dépend de la fréquence d'horloge et des périphériques utilisés.

Le mode veille est le premier mode disponible pour réduire la consommation d'énergie. Dans ce mode, la plupart des fonctionnalités sont suspendues, la fréquence du processeur est abaissée et ses activités sont réduites à celles nécessaires au réveil.

En mode sommeil profond tous les signaux d'horloge sont arrêtés et le CPU a besoin d'un événement extérieur pour sortir de cet état.



#### 4.1.9 Le CMSIS

L'un des principaux avantages de la plate-forme ARM (à la fois pour les fournisseurs de silicium et les développeurs d'applications) est l'existence d'un ensemble complet d'outils de développement (compilateurs, bibliothèques d'exécution, débogueurs, etc.) réutilisables entre plusieurs fournisseurs.

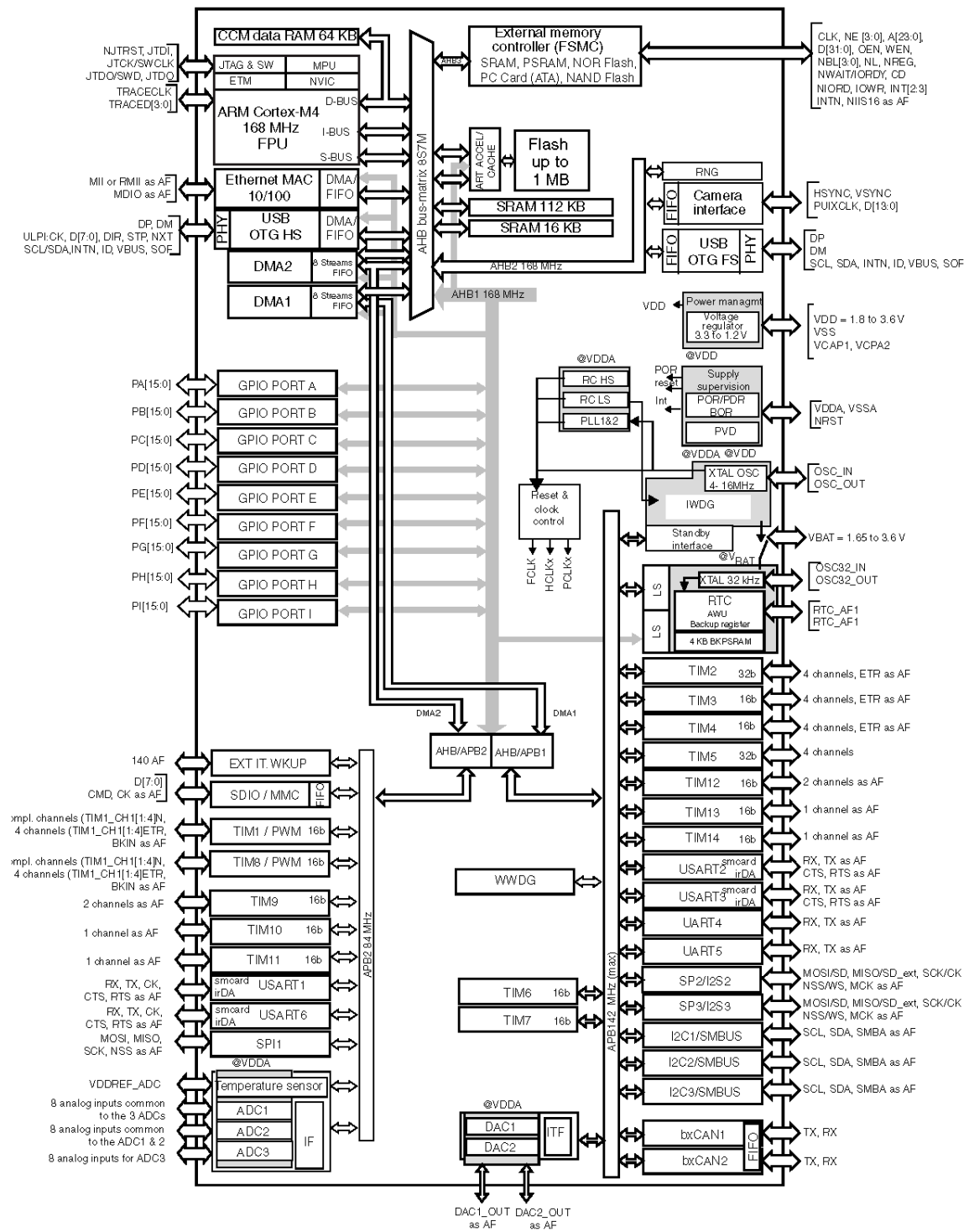
ARM travaille aussi activement sur une façon de normaliser les infrastructures logicielles entre les fournisseurs MCU. Le « Cortex Microcontroller Software Interface Standard » (CMSIS) est une couche d'abstraction matérielle « Hardware Abstraction Layer » produit par un fournisseur indépendant pour la série de processeurs Cortex-M. Le CMSIS se compose des éléments suivants:

- **CMSIS-CORE** : API pour le processeur central Cortex-M et ses périphériques. Elle fournit une interface standardisée pour Cortex-M0, M3, M4 et M7.
- **CMSIS-Driver** : définit les interfaces des pilotes de périphériques génériques pour middleware afin de les rendre réutilisables à travers les dispositifs pris en charge. L'API est indépendante du RTOS et gère des périphériques de microcontrôleur grâce à des applications middleware qui implémente, par exemple, des piles de communication, des systèmes de fichiers, ou des interfaces utilisateur graphiques.
- **CMSIS-DSP** : librairie de fonctions pour le DSP avec plus de 60 fonctions pour les différents types de données : virgule fixe et simple précision en virgule flottante (32 bits). La librairie est disponible pour le Cortex-M0, le Cortex-M3, le Cortex-M4 et le Cortex-M7. La mise en œuvre du Cortex-M4 est optimisée pour le jeu d'instructions SIMD.
- **CMSIS-RTOS API** : API commune pour les systèmes d'exploitation temps réel. Elle fournit une interface de programmation standardisée qui est portable à plusieurs RTOS et permet donc des modèles de logiciels, middleware, des bibliothèques et d'autres composants qui peuvent fonctionner à travers les systèmes RTOS supportés.
- **CMSIS-Pack** : décrit, avec un fichier de description XML (nommé PDSC), une collection de fichiers (appelé pack logiciel) qui comprend la source, l'entête, les fichiers de la librairie, la documentation, les algorithmes de programmation Flash, le code source des modèles et des exemples de projets. Les outils de développement et les infrastructures Web utilisent le fichier PDSC pour extraire des paramètres de l'appareil, des composants logiciels et des configurations de certaines cartes d'évaluation.
- **CMSIS-SVD** : System View Description (SVD) pour périphériques. Décrit les périphériques d'un dispositif dans un fichier XML et peut être utilisé pour dépanner le périphérique dans le débogueur.
- **CMSIS-DAP** : Debug Port d'accès. Firmwares normalisées pour une unité de débogage qui se connecte au « CoreSight Debug Accès Port ». CMSIS-DAP est distribué sous emballage séparé et est adapté à l'intégration sur les cartes d'évaluation.

Cependant, cette initiative d'ARM est toujours en évolution et le soutien à tous les composants de ST n'est pas encore parfait. La librairie HAL officielle de la compagnie ST est le principal moyen de développer des applications pour la plateforme STM32 car elle seule répond à toutes les particularités de ses propres MCU. En outre, il est tout à fait clair que l'objectif principal des fournisseurs de silicium est de conserver leurs clients afin d'éviter leur migration vers d'autres plates-formes de microcontrôleurs (même si elle est fondée sur le même noyau Cortex ARM). Donc, nous sommes vraiment loin d'avoir une couche complète et portable qui fonctionne sur tous les ARM MCU de base sur le marché.

La compagnie ST, comme bien d'autres fournisseurs de microcontrôleur, offre un logiciel générateur de code pour aider ses utilisateurs dans le développement d'applications dédiées aux microcontrôleurs de toutes les familles du STM32. Le STM32CubeMX fait appel à la librairie HAL mise au point spécifiquement par ST Microelectronics.

#### 4.1.10 Le Schéma bloc du STM32F407



## 4.2 LE STM32CubeMX

### 4.2.1 Les caractéristiques de base

- Le STM32F4CubeMX est un logiciel intégré, cohérent et complet qui offre à l'utilisateur un générateur de code libre de tous droits d'auteur.
- Ce logiciel offre une portabilité maximisée entre tous les processeurs de la famille STM32.
- Il renferme des centaines d'exemples pour une meilleure compréhension.
- Il intègre des bibliothèques « Hardware Abstract Layer » (HAL) de haute qualité utilisant CodeSonar®, un outil d'analyse statique.
- Il offre des applications « middleware » comprenant, entre autres, des bibliothèques spécifiques à l'emploi du USB « Host » et « Device » et à la pile TCP / IP.
- Il propose un mécanisme de mise à jour qui peut être activé par l'utilisateur afin d'être informé des nouveautés.

### 4.2.2 Une description du produit

Le STMCube™ est une initiative originale de la compagnie STMicroelectronics. Il permet de faciliter la vie des développeurs en réduisant les efforts de développement, le temps et le coût. STM32Cube couvre tous les processeurs de la famille STM32.

Le STM32Cube comprend le STM32CubeMX, un outil de configuration graphique qui permet de générer le code d'initialisation « C » en utilisant des assistants graphiques.

Il comprend également la plate-forme STM32CubeF4 qui intègre le STM32Cube HAL (un logiciel embarqué STM32 de couche d'abstraction, assurant la portabilité maximisée à travers tous les circuits de la famille STM32), en plus d'un ensemble cohérent de composants middleware (RTOS, USB, TCP / IP et graphiques). Tous les utilitaires logiciels embarqués sont livrés avec des exemples complets.

Le STM32CubeF4 rassemble, en un seul paquet, tous les composants logiciels embarqués génériques nécessaires au développement d'une application sur les microcontrôleurs STM32F4. Suite à l'initiative STM32Cube, cet ensemble de composants est très portable, non seulement au sein de la série STM32F4 mais aussi à d'autres séries STM32.

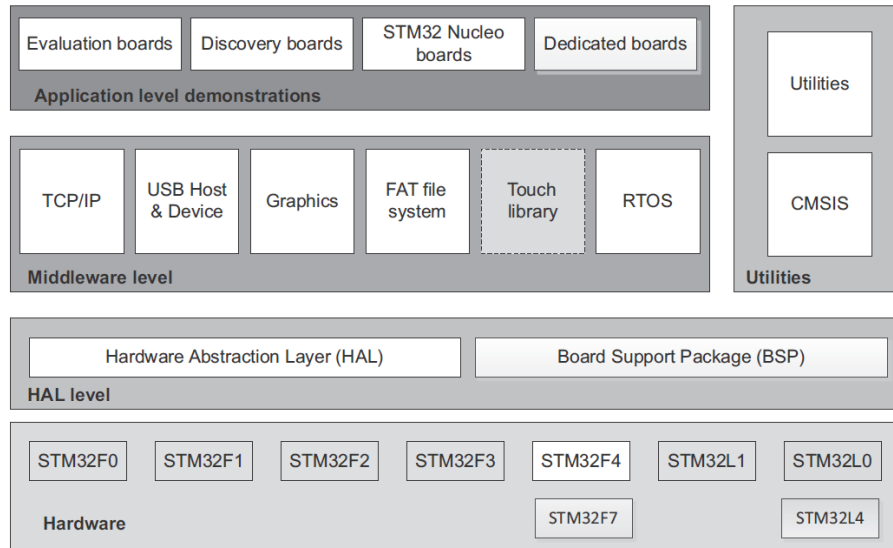
Le STM32CubeF4 est entièrement compatible avec le générateur de code STM32CubeMX qui permet la génération du code d'initialisation. L'ensemble logiciel comprend une couche d'abstraction matérielle de bas niveau (HAL) qui couvre le



matériel du microcontrôleur ainsi qu'une vaste panoplie d'exemples touchant presque tous les périphériques embarqués du SoC.

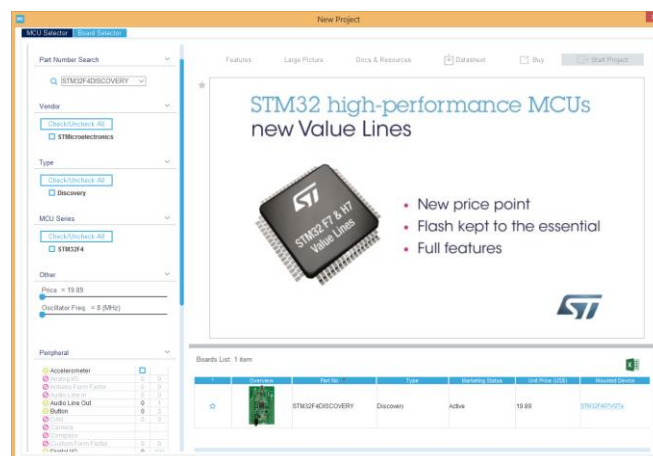
### 4.2.3 Le STM32CubeMx... un logiciel d'intégration

Le STM32CubeMX est un outil de développement d'application permettant d'intégrer les quatre couches nécessaires à la conception d'un produit composé d'électronique programmable : la couche matérielle, la couche logicielle de bas niveau, la couche logicielle de gestion des périphériques complexe et la couche application.



#### 4.2.3.1 La couche matérielle

La couche matérielle est très simple à définir à l'intérieur du logiciel STM32CubeMX. Elle fait référence au « Microcontroller unit » (MCU) de la famille STM32 choisie pour l'application à créer. Elle est fréquemment accompagnée d'un circuit d'évaluation tel les circuits « Discovery » ou « Nucleo », conçu et produit par la compagnie STMicroelectronics. Lors de la création d'un projet avec le STM32CubeMX, l'utilisateur doit connaître le matériel avec lequel il réalisera son développement. La première étape consistera à informer le logiciel du choix du microcontrôleur (MCU) ou du circuit de développement (BOARD) employé.



#### 4.2.3.2 La couche logicielle de bas niveau

Cette couche se découpe en deux éléments distincts : Le « Board Support Package » (BSP) et les librairies « Hardware Abstract Layer » (HAL).

Le **BSP** offre un ensemble de « Application Programming Interface » (API) liées aux composants matériels intégrés sur les plaquettes de circuit d'évaluation. Dans les systèmes embarqués, le BSP est une implémentation de code de soutien spécifique (logiciel) pour une carte donnée conforme à un système d'exploitation donné. Il contient le support de dispositif minimal pour charger le système d'exploitation et des pilotes de périphériques de tous les éléments intégrés à la carte. Sur une carte telles les plaquettes de développement « Discovery », il fait plus référence aux définitions des registres internes du MCU et au matériel intégré à la carte. Si une carte contient des boutons poussoirs, des diodes électroluminescentes indicatrices ou tous autres éléments d'entrées sorties, le BSP en fera la définition complète (position en mémoire, broches de port employées...). Le travail du concepteur sera simplifié par cet élément logiciel de la couche de bas niveau.

En outre, le BSP doit effectuer les opérations suivantes :

- Initialisation du processeur,
- Initialisation du bus,
- Initialisation du contrôleur d'interruption,
- Initialisation de l'horloge,
- Réinitialisez les paramètres de la RAM,
- Charger et exécuter bootloader de la mémoire flash

La **HAL** est une couche logicielle qui fournit les pilotes de bas niveau ainsi qu'une méthode pour interfacier le matériel aux couches supérieures (les applications, les bibliothèques et les piles). Par exemple, pour les périphériques de communication (I2C, UART ...), il fournit des API permettant d'initialiser et de configurer le périphérique, de gérer le transfert de données par polling, interruption ou canal DMA, et de gérer les erreurs de communication qui peuvent survenir pendant la communication. Les pilotes des librairies HAL sont divisés en deux catégories :

- API générique qui fournit des fonctions communes et génériques à toute la série STM32,
- API d'extension qui fournit des fonctions spécifiques et personnalisées pour une famille spécifique ou un numéro de pièce spécifique.

#### 4.2.3.3 La couche logicielle de gestion des périphériques complexes

Les composants **Middleware** sont des bibliothèques couvrant la gestion USB, le STemWin (fonctionnalités graphique), la libjpeg (traitement des fichiers JPEG), le FreeRTOS (Système d'opération à temps réel), la FATFS (gestionnaire de fichier), la lwIP (une pile IP légère) et le PolarSSL (gestionnaire du protocole SSL). Les interactions horizontales entre les composants de cette couche sont faites directement en appelant les fonctionnalités API alors que les interactions verticales sur les pilotes de bas niveau se font à travers des appels spécifiques et des macros statiques mises en œuvre dans l'interface du système de bibliothèque d'appels.

Les principales caractéristiques de chaque composant intermédiaire sont les suivantes:

### **Libraries « host and device » USB**

- Plusieurs classes USB pris en charge (Mass-Storage, HID, CDC, DFU, AUDIO, MTP).
- Une prise en charge multifonction du transfert de paquets : permet d'envoyer de grandes quantités de données sans partitionnement de ces données.
- Des fichiers de configuration pour modifier le noyau sans modifier le code de la bibliothèque (lecture seule).
- Des structures de données alignées sur 32 bits pour gérer le transfert à base de DMA.
- Une prise en charge de plusieurs instances de base de OTG USB de niveau utilisateur par le biais de fichiers de configuration.
- Un fonctionnement avec plus d'un périphérique USB « host and device ».
- Une possibilité d'utiliser un RTOS en fonctionnement autonome.
- Un lien avec le pilote de bas niveau qui se fait à travers une couche d'abstraction, ce qui permet d'éviter toute dépendance entre la bibliothèque et les pilotes de bas niveau.

### **STemWin pile graphique**

- Une solution de qualité professionnelle pour le développement de l'interface graphique basée sur la solution EMWIN par SEGGER.
- Les pilotes d'affichage optimisés.
- Un outil logiciel pour la génération de code et l'édition bitmap (STemWin Builder ...)

### **libjpeg**

- Une mise en œuvre en langage C pour l'encodage et le décodage des images JPEG.

### **FreeRTOS**

- Une couche de compatibilité CMSIS.
- Des opérations de Tickless en mode de faible puissance.
- Une intégration simple avec tous les modules de middleware STM32Cube.

### **Le système de fichiers FAT**

- Les noms de fichier longs supportés.
- Un support multi-drive.
- Une possibilité d'utiliser un RTOS en fonctionnement autonome.
- Des exemples avec microSD et USB.

### **LwIP pile TCP / IP**

- Une possibilité d'utiliser un RTOS en fonctionnement autonome.

#### 4.2.3.4 La couche application

Cette couche n'est pas réellement supportée par les outils de développement du STM32CubeMX. C'est le concepteur qui en a la responsabilité et lui seul prendra en charge cette couche. Elle est généralement conçue, réalisée et dépannée après que tous les périphériques soient fonctionnels tant du point de vue de l'électronique que de la programmation des pilotes qui accompagnent ces périphériques.

### 4.2.4 Les pilotes HAL

Les pilotes HAL ont été conçus pour offrir un ensemble d'API capables d'interagir facilement avec les couches supérieures de l'application. Chaque pilote se compose d'un ensemble de fonctions couvrant les caractéristiques les plus courantes du périphérique. Le développement de chaque pilote est associé à une API commune qui normalise la structure du pilote, les fonctions et les noms de paramètres.

Les pilotes HAL se composent d'un ensemble de modules de pilote, chaque module étant relié au périphérique. Cependant, dans certains cas, le module est relié à un mode de fonctionnement du périphérique. A titre d'exemple, plusieurs modules existent pour l'unité USART périphérique : Module pilote UART, module pilote USART, module pilote de carte à puce et module pilote IrDA.

Les principales caractéristiques des librairies HAL sont les suivantes :

- Un ensemble d'API portable entre les familles de circuits STM32 et couvrant les caractéristiques communes des périphériques, ainsi que des API d'extension pour les caractéristiques spécifiques aux périphériques.
- Trois modèles de programmation : polling, interruption et DMA.
- Conformés RTOS :
- API entièrement réentrants.
- l'utilisation systématique des délais d'attente (TimeOut) en mode polling.
- Le soutien multi-instance permettant des appels d'API simultanés pour plusieurs instances d'un même groupe de périphériques. (USART1, USART2 ...)
- Toutes les API HAL mettent en œuvre un mécanisme de rappel des fonctions utilisateur :
- Les événements d'interruption associés aux périphériques.
- Les événements d'erreur.
- Un mécanisme de verrouillage de l'objet : un accès sécurisé du matériel pour éviter des accès multiples parasites aux ressources partagées.
- Un Timeout utilisé pour tous les processus : le délai d'attente peut être un simple compteur ou une base de temps.

## 4.2.4.1 Les fichiers « DRIVERS » (pilotes de périphérique)

Les noms des fichiers « Drivers » se définissent ainsi :

Fichier	Description
<i>stm32f4xx_hal_ppp.c</i>	Module principal du fichier de pilote de périphérique. Il comprend les API qui sont communes à tous les appareils STM32. Ex: <i>stm32f4xx_hal_adc.c</i> , <i>stm32f4xx_hal_irda.c</i> , ...
<i>stm32f4xx_hal_ppp.h</i>	Fichier d'en-tête du fichier C du module principal. Il comprend des données communes, le «handle», l'énumération de la structure, la définition des instructions, des macros, ainsi que le générique API. Ex: <i>stm32f4xx_hal_adc.h</i> , <i>stm32f4xx_hal_irda.h</i> , ...
<i>stm32f4xx_hal_ppp_ex.c</i>	Fichier d'extension d'un pilote de périphérique / module. Il comprend les API spécifiques pour un numéro de pièce ou une famille, ainsi que les API nouvellement définis qui remplacent les API génériques par défaut si le processus interne est mis en œuvre de manière différente. Ex: <i>stm32f4xx_hal_adc_ex.c</i> , <i>stm32f4xx_hal_dma_ex.c</i> , ...
<i>stm32f4xx_hal_ppp_ex.h</i>	Fichier d'en-tête du fichier extension C. Il comprend les données spécifiques et les structures de dénombrement, les définitions, les états et macros, ainsi que les API spécifiques des dispositifs. Ex: <i>stm32f4xx_hal_adc_ex.h</i> , <i>stm32f4xx_hal_dma_ex.h</i> , ...
<i>stm32f4xx_ll_ppp.c</i>	Pilote périphérique de bas niveau qui peut être consulté à partir d'un ou de plusieurs pilotes HAL. Il offre un ensemble d'API et de services utilisés par le pilote supérieur. Du point de vue de l'utilisateur, les pilotes de bas niveau ne sont pas accessibles directement. Ex : <i>stm32f4xx_ll_fsmc.c</i> offre un ensemble d'API utilisée par : <i>stm32f4xx_hal_sdram.c</i> , <i>stm32f4xx_hal_sram.c</i> , <i>stm32f4xx_hal_nor.c</i> , <i>stm32f4xx_hal_nand.c</i> , ...
<i>stm32f4xx_ll_ppp.h</i>	Fichier d'en-tête du fichier de bas niveau C. Il est inclus dans le fichier d'en-tête du pilote HAL. Ex : <i>stm32f4xx_ll_fsmc.h</i> , <i>stm32f4xx_ll_usb.h</i> , ...
<i>stm32f4xx_hal.c</i>	Ce fichier est utilisé pour l'initialisation HAL et il contient, entre autres, le délai de base pour les API sysTick.
<i>stm32f4xx_hal.h</i>	Fichier d'en-tête du fichier <i>stm32f4xx_hal.c</i> .
<i>stm32f4xx_hal_msp_template.c</i>	Fichier modèle à copier dans le dossier de l'application utilisateur. Il contient l'initialisation et la dé-initialisation des périphériques utilisés dans l'application de l'utilisateur.
<i>stm32f4xx_hal_conf_template.h</i>	Fichier d'entête du modèle permettant de personnaliser les pilotes pour une application donnée.
<i>stm32f4xx_hal_def.h</i>	Ressources communes pour le HAL telles les définitions, les déclarations communes, les énumérations, les structures et les macros.
<i>system_stm32f4xx.c</i>	Ce fichier contient la fonction <code>SystemInit()</code> qui est appelée au démarrage juste après la remise à zéro et avant d'aller au programme principal. Il ne configure pas l'horloge système au démarrage (contrairement à la bibliothèque standard). Cette configuration doit être effectuée en utilisant la couche d'abstraction API dans les fichiers utilisateur. L'appel de ce fichier permet de : <ul style="list-style-type: none"> <li>déplacer la table de vecteur dans la SRAM interne.</li> <li>configurer le périphérique FSMC / FMC (lorsque disponible) pour l'utilisation en mémoire externe SRAM ou SDRAM montés sur la carte d'évaluation.</li> </ul>

Où ppp représente le nom du périphérique. (ex : adc, usb, spi, i2c...)

\*\*\*\*\*  
**\*\* Ces fichiers ne sont généralement pas modifiés par l'utilisateur. \*\***  
 \*\*\*\*\*

#### 4.2.4.2 Les fichiers « USER » (réservés aux utilisateurs)

Fichier	Description
<i>startup_stm32f4xx.s</i>	Fichier spécifique qui contient le gestionnaire des périphériques et les vecteurs d'exception.
<i>stm32f4xx_flash.icf (optional)</i>	Fichier permettant de gérer le linker pour EWARM. Il permet principalement d'adapter la taille des stack / heap pour répondre aux exigences de l'application. De plus, il gère l'emplacement de la mémoire.
<i>stm32f4xx_hal_msp.c</i>	Ce fichier contient l'initialisation et la dé-initialisation du MSP des périphériques utilisés dans l'application de l'utilisateur.
<i>stm32f4xx_hal_conf.h</i>	Ce fichier permet à l'utilisateur de personnaliser les pilotes HAL pour une application spécifique. Il n'est pas obligatoire de modifier cette configuration. L'application peut utiliser la configuration par défaut sans aucune modification.
<i>stm32f4xx_it.c/.h</i>	Ce fichier contient le gestionnaire d'exceptions et les routines d'interruption des périphériques de service et appelle HAL_IncTick () à des intervalles de temps réguliers pour incrémenter une variable locale (déclarée dans <i>stm32f4xx_hal.c</i> ) utilisé comme base de temps. Par défaut, cette fonction est appelée chaque 1ms dans sysTick ISR. Le PPP_IRQHandler () doit appeler HAL_PPP_IRQHandler () si un processus d'interruption est utilisé dans l'application.
<i>main.c/.h</i>	Ce fichier contient le programme principal du projet, notamment : <ul style="list-style-type: none"> <li>• l'appel à <i>hal_init ()</i>,</li> <li>• la mise en œuvre <i>assert_failed ()</i>,</li> <li>• la configuration de l'horloge du système,</li> <li>• l'initialisation des périphériques HAL,</li> <li>• et le code d'application de l'utilisateur.</li> </ul>

Sauf exception, l'utilisateur modifiera uniquement les fichiers du groupe USER dans l'IDE. Il s'agit des fichiers *main.c* et *.h*, *stm32f4xx\_hal\_msp.c* et *.h* ainsi que les fichiers *stm32f4xx\_it.c* et *.h* si des interruptions sont traitées. Le STM32CubeMX se charge de presque tout!

Lorsque l'utilisateur entre du nouveau code dans un fichier (ex : dans le fichier *main.c*), il doit respecter les zones réservées.

```
/* USER CODE BEGIN 1 */
```

```
/* USER CODE END 1 */
```

Ces zones permettent au STM32CubeMX de générer du code par-dessus le code existant sans effacer le code entré par l'utilisateur. Très utile si l'utilisateur désire, par exemple, ajouter une broche I/O dans le STM32CubeMX et générer le code de nouveau pour que la modification soit apparente dans les fichiers du projet.

Le STM32Cube est livré avec des modèles et exemples prêts à l'utilisation et ce pour chaque plaquette de circuit pris en charge. Chaque projet contient les fichiers répertoriés ci-dessus et un projet préconfiguré pour les environnements de programmation (IDE) pris en charge.

Chaque modèle de projet fournit une fonction de boucle principale vide et peut être utilisé comme point de départ pour se familiariser avec les paramètres du projet. Leurs caractéristiques sont les suivantes:

- Il contient les sources de pilotes HAL, CMSIS et BSP qui sont les composants minimaux pour développer un code sur une carte donnée.
- Il contient les chemins d'accès pour tous les composants du projet.
- Il définit le dispositif STM32 pris en charge et permet de configurer les pilotes CMSIS et HAL en conséquence.
- Il fournit les fichiers utilisateurs prêts à employer. Ces derniers sont préconfigurés tels que définis ci-dessous:
- Le HAL est initialisé,
- Le ISR sysTick mis en œuvre pour la fonction HAL\_Delay ()
- L'horloge du système est configurée avec la fréquence maximum du dispositif.

#### 4.2.5 Les structures de données associées aux librairies HAL

Chaque pilote HAL peut contenir les structures de données suivantes:

- structures du «handle» du périphérique,
- structures d'initialisation et de configuration,
- structures de processus spécifiques.

##### 4.2.5.1 La structure de données des gestionnaires de périphériques

Les API ont une architecture générique multi-instance modulaire qui permet de travailler avec plusieurs instances IP simultanées.

PPP\_HandleTypeDef \*handle est la structure principale qui est mise en œuvre pour chacun des pilotes HAL. Elle gère la configuration et les registres du périphérique / module et intègre toutes les structures et les variables nécessaires pour suivre le flux de périphérique.

Le «handle» du périphérique est utilisé dans les buts suivants:

- Un support multi-instance: chaque instance périphérique / module a son propre «handle». Il en résulte que les ressources d'instance sont indépendantes.
- Un processus périphérique intercommunication: le «handle» est utilisé pour gérer les ressources de données partagées entre les routines de traitement.
- Pour le stockage, un «handle» est également utilisé pour gérer les variables globales au sein d'un pilote HAL donné.



#### 4.2.6 Les règles d'utilisation des librairies HAL

Pour les périphériques partagés et de système, aucun « handle » ou instance n'est utilisée. Cette règle s'applique aux périphériques suivants:

- GPIO
- sysTick
- NVIC
- RCC
- FLASH.

Exemple : Le HAL\_GPIO\_Init () ne nécessite que l'adresse du GPIO et ses paramètres de configuration.

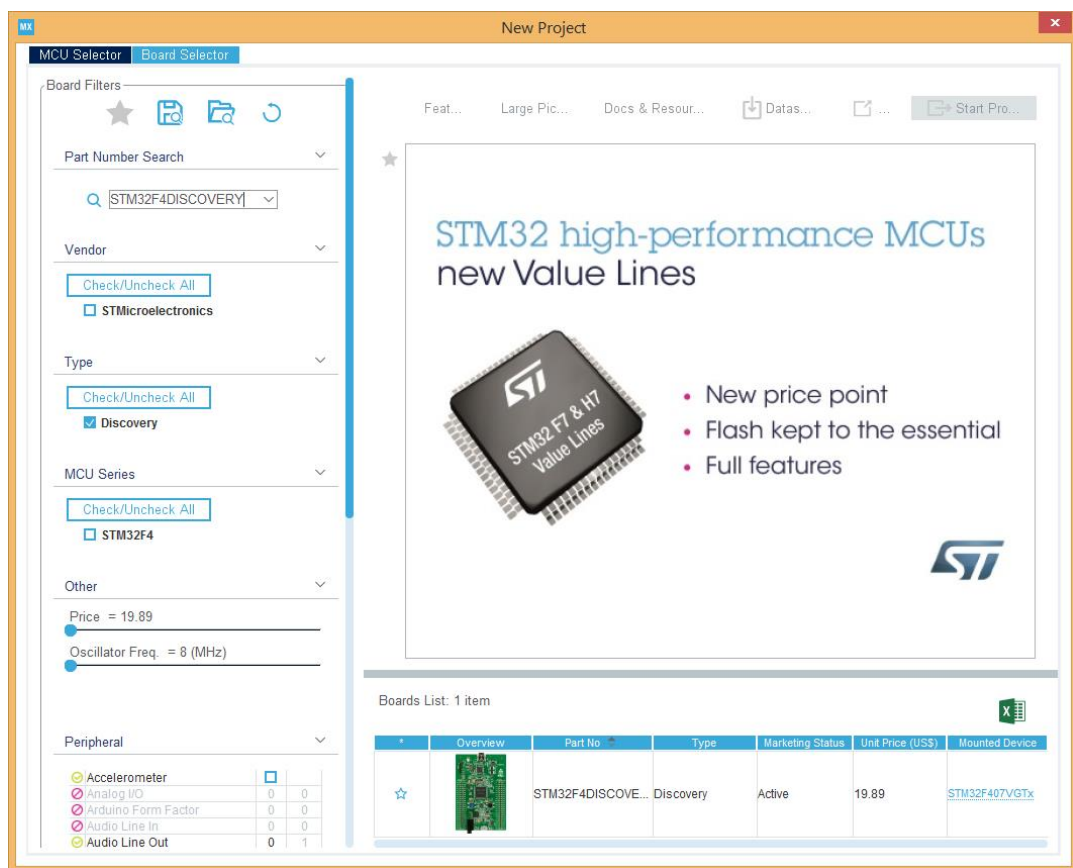
Les périphériques non partagés et n'étant pas directement attachés au système nécessiteront un « handle » qui sera généralement employé comme paramètre dans les fonctions HAL associées.


- Note : pour tout autre détail, faites référence au document UM1725. Le manuel utilisateur des librairies HAL.

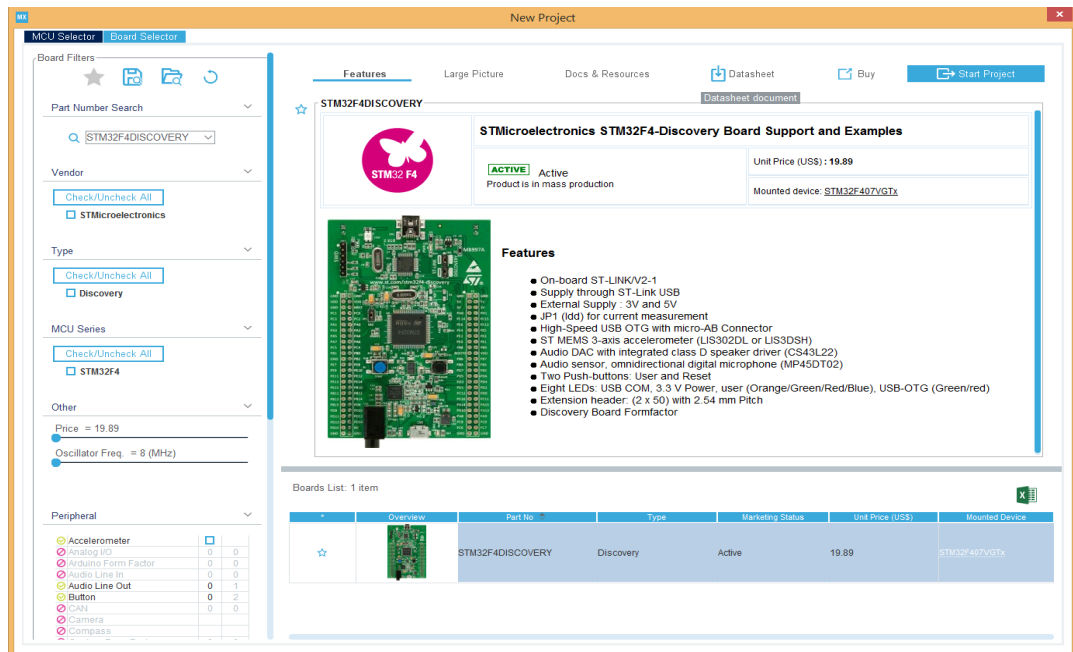
#### 4.2.7 Exemple d'utilisation, (un premier projet...clignotement d'une del)



1. Ouvrez le logiciel .
2. Définissez le matériel avec lequel vous désirez effectuer votre projet. Vous avez la possibilité de sélectionner un microcontrôleur (MCU Selector) ou une plaquette de circuit de développement (Board Selector). Comme le travail se fera autour d'une plaquette de développement « Discovery F407 », on sélectionnera Discovery dans la liste **Type of Board** de l'onglet **Board Selector** et on fera la recherche de STM32F4DISCOVERY dans la liste des **Part Number Search**.



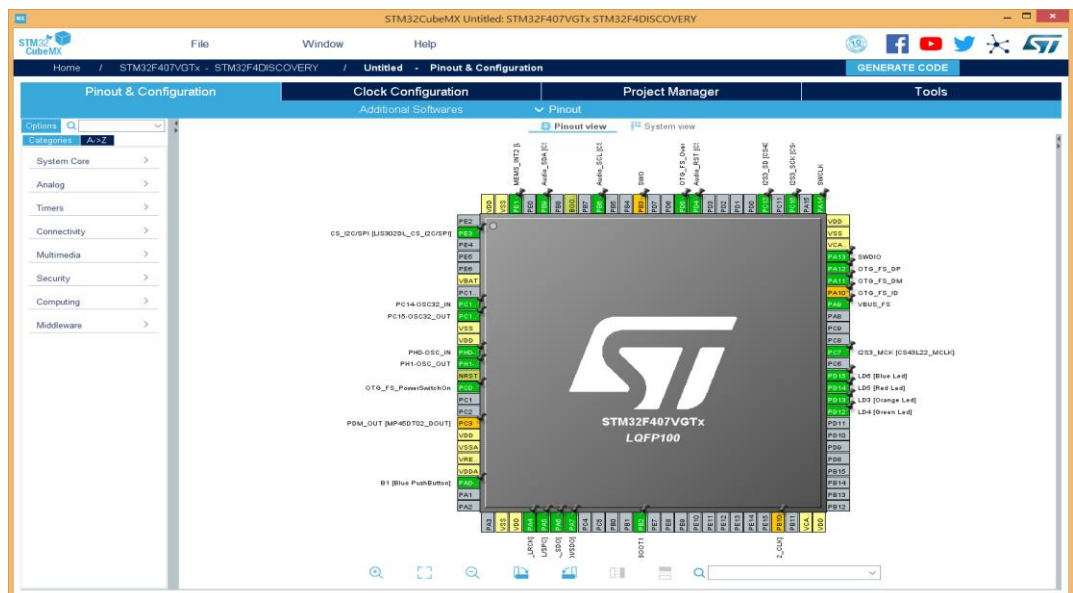
Après avoir appuyé sur STM32F4DISCOVERY dans le bas de la fenêtre au centre gauche, vous devriez avoir l'aperçu suivant et être en mesure de démarrer le projet en appuyant sur le bouton  en haut à droite.



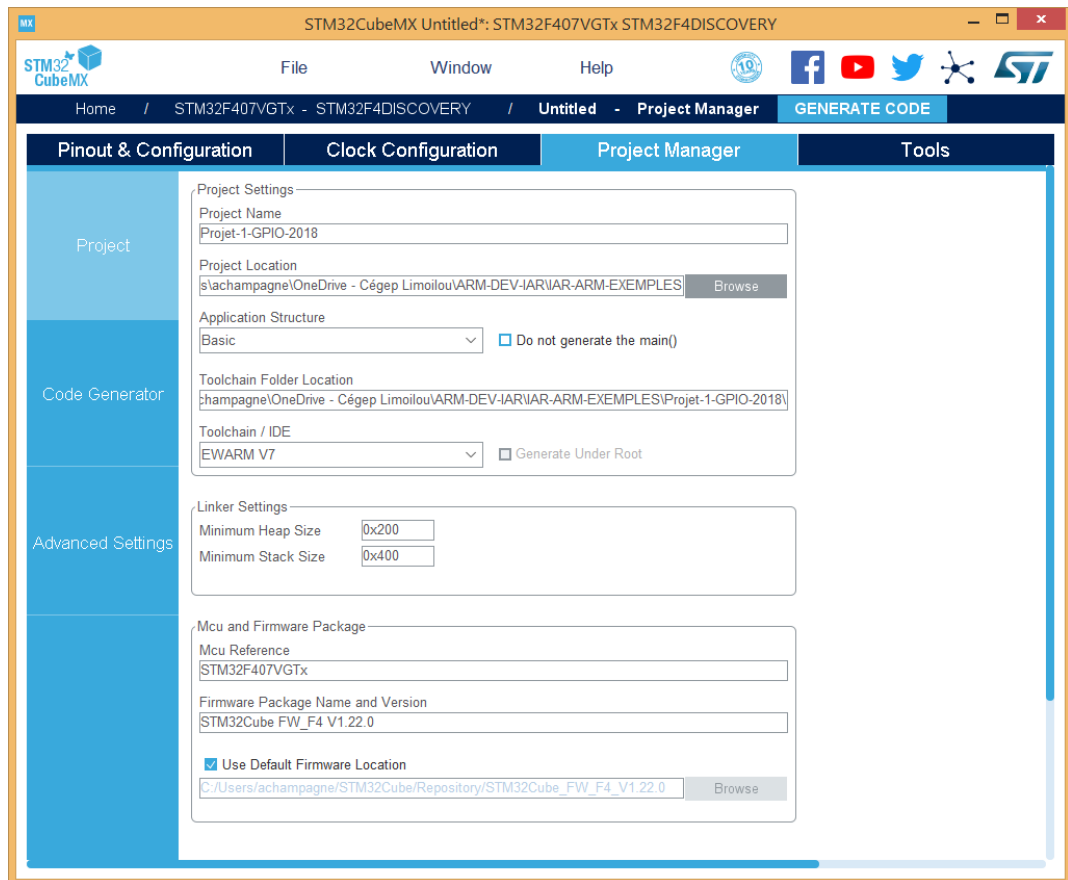
À la question :



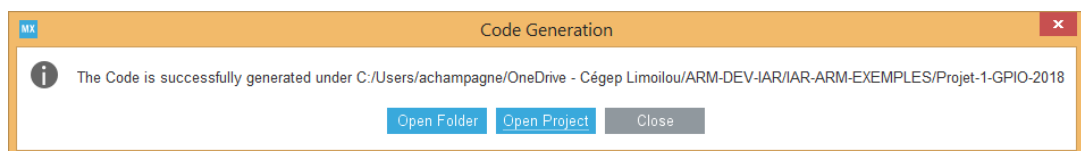
Répondez « Yes » pour ainsi visualiser l'agencement des broches du MCU correspondant à la disposition sur la plaquette Discovery F407. Cette définition matérielle est en lien avec le BSP dont nous avons parlé précédemment.



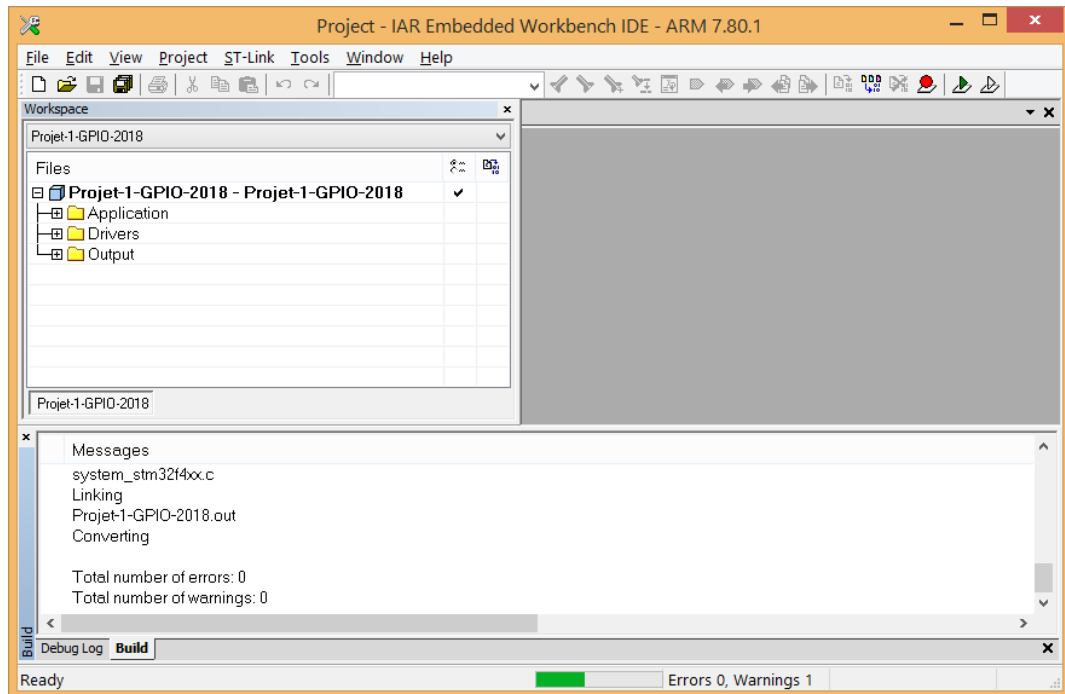
3. Sous l'onglet « **Project Manager** », accordez un nom et une location à votre projet et définissez l'environnement IDE avec lequel vous travaillez. (EWARM V7 pour l'utilisation avec IAR)



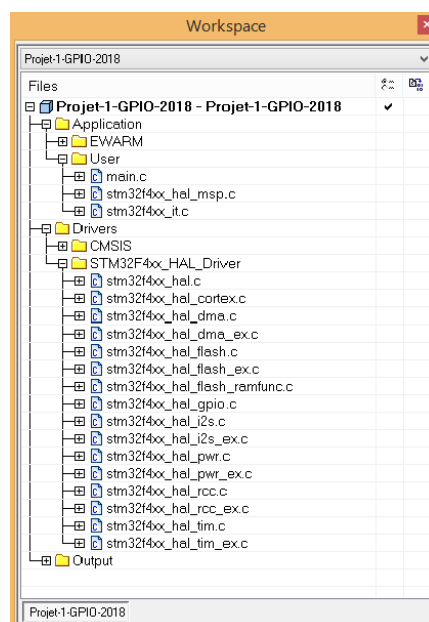
4. Maintenant que la définition de votre projet est complétée, demandez au logiciel de générer le code de base en choisissant la commande **GENERATE CODE** en haut à droite de la fenêtre principale.
5. Appuyez sur le bouton **Open Project** pour visualiser le code à travers l'IDE de IAR.



6. Plusieurs fichiers ont été générés et agencés de manière à créer un projet complet et compatible à la plaquette Discovery F407. Pour le moment, ce projet compilé n'aurait aucune utilité tangible car il ne sert qu'à faire une configuration adéquate du matériel avec lequel nous travaillons.



Si nous regardons de plus près l'environnement de travail (**Workspace**), nous constatons que les groupes **Application**, **Drivers** et **Output** ont été créés. Ces groupes renferment un ensemble de fichiers nécessaires au bon fonctionnement du système. Analysons de plus près le contenu de ces groupes.



Le groupe **APPLICATION** renferme deux groupes : le groupe **EWARM** destiné à la configuration du MCU et le groupe **USER** réservé à l'utilisateur. C'est dans ce groupe que les fichiers pourront être modifiés. Il renferme le **main.c**, bien sûr, ainsi que le fichier réservé à l'écriture des routines d'interruption, **stm32f4xx\_it.c**. Le fichier **stm32f4xx\_hal\_msp.c** quant à lui fera l'objet de modifications uniquement lorsque des besoins spécifiques, telle la modification de la priorité des interruptions/exceptions, se feront sentir. C'est à l'intérieur de ce groupe que l'utilisateur devra intégrer les fichiers .c ou .o, écrit par lui ou un tiers, destinés à des applications spécifiques telles la gestion d'un écran LCD.

Le groupe **DRIVERS** intègre les groupes **CMSIS** et **STM32F4xx\_HAL\_Driver**. L'ensemble des fichiers de la librairie HAL nécessaire au bon fonctionnement du projet se retrouve dans ce groupe. L'application STM32CubeMX se charge d'inclure uniquement les fichiers nécessaires au respect de la définition faite lors de la génération du code.

Le groupe **OUTPUT** contient les fichiers résultant de la compilation. Le fichier .MAP peut représenter un certain intérêt pour l'utilisateur car il renferme des informations intéressantes dont celle de la mémoire utilisée par un projet donnée.

```
6 062 bytes of readonly code memory
  31 bytes of readonly data memory
1 112 bytes of readwrite data memory
```

Faisons une première tentative de compilation de ce projet « vide ». Dans l'environnement de programmation (IDE) de IAR, sous l'onglet « **Project** », sélectionnez la commande « **Rebuild All** ». Si tout se passe bien, vous devriez retrouver le message :

```
Total number of errors: 0
Total number of warnings: 0
```

Ce projet pourrait être téléchargé dans la cible mais, comme nous l'avons mentionné précédemment, il n'aurait aucun effet. Ajoutons donc quelques lignes pour activer une DEL et l'éteindre.

7. Nous avons parlé des bibliothèques HAL, bibliothèques faisant partie intégrante de l'application STM32CubeMX. Ces bibliothèques offrent des API permettant de contrôler l'ensemble des périphériques du MCU. Elles renferment donc des fonctions pour manipuler des bits et par le fait même les leds y étant attachées. Le fichier UM1725.PDF sera employé ici pour trouver la syntaxe à utiliser avec ces fonctions. En effectuant une recherche de HAL\_GPIO, vous trouverez la description des fonctions de l'API associées aux GPIO. En voici la liste :

- HAL\_GPIO\_Init()
- HAL\_GPIO\_DeInit()
- HAL\_GPIO\_ReadPin()
- HAL\_GPIO\_WritePin()
- HAL\_GPIO\_TogglePin()
- HAL\_GPIO\_LockPin()
- HAL\_GPIO\_EXTI\_IRQHandler()
- HAL\_GPIO\_EXTI\_Callback()

8. Assumez que l'initialisation est faite. En effet, la définition de la plaquette « Discovery » donnée au logiciel STM32CubeMX a permis à ce dernier de générer l'initialisation complète assignant les lignes de port employées pour les del en sortie. Cette définition c'est appliquée suite à l'appel de la fonction HAL\_GPIO\_Init(); appel lancé après avoir minutieusement défini tous les éléments de la structure associée. L'appel complet ressemble à ceci :

```
GPIO_InitStruct.Pin = LD4_Pin|LD3_Pin|LD5_Pin|LD6_Pin|Audio_RST_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init(GPIOD, &GPIO_InitStruct);
```

9. Dans la précédente structure, nous retrouvons deux informations très importantes : Le nom de la broche et son groupe associé. Le nom de la broche attachée à la del verte est « LD4\_Pin » et le groupe pour les quatre del est « GPIOD », faisant référence au port D employé.

La fonction HAL\_GPIO\_TogglePin() permettant de changer l'état d'une sortie, nous lui attribuerons les paramètres GPIOD et LD4\_Pin pour modifier l'état de la del verte. La fonction prend la forme suivante :

```
HAL_GPIO_TogglePin(GPIOD, LD4_Pin);
```

Ajoutons à cela un délai de 1000 ms grâce à la fonction HAL\_Delay(1000); et nous aurons un programme complet à intégrer au fichier main.c.

10. Dans le main(), sous le while(1), ajoutez les deux lignes et compilez de nouveau. Prenez soin d'insérer les deux lignes entre les lignes USER CODE ... 3 comme suit :

```
/* USER CODE BEGIN 3 */  
    HAL_GPIO_TogglePin(GPIOD, LD4_Pin);  
    HAL_Delay(1000);  
}  
/* USER CODE END 3 */
```

11. Ne reste qu'à faire l'essai de ce programme en le compilant, avec la commande « **Rebuild All** » du menu « **Project** », et en le transférant à la cible avec la commande « **Download** » de l'onglet « **Projet** ».

Amusez-vous maintenant avec les autres dels. Leur définitions, couleurs et positions sont les suivantes :

Définition	Couleur	Position
LD3_Pin	Orange	Droite
LD4_Pin	Verte	Haut
LD5_Pin	Rouge	Bas
LD6_Pin	Bleue	Gauche



#### 4.2.8 Acronymes et définitions

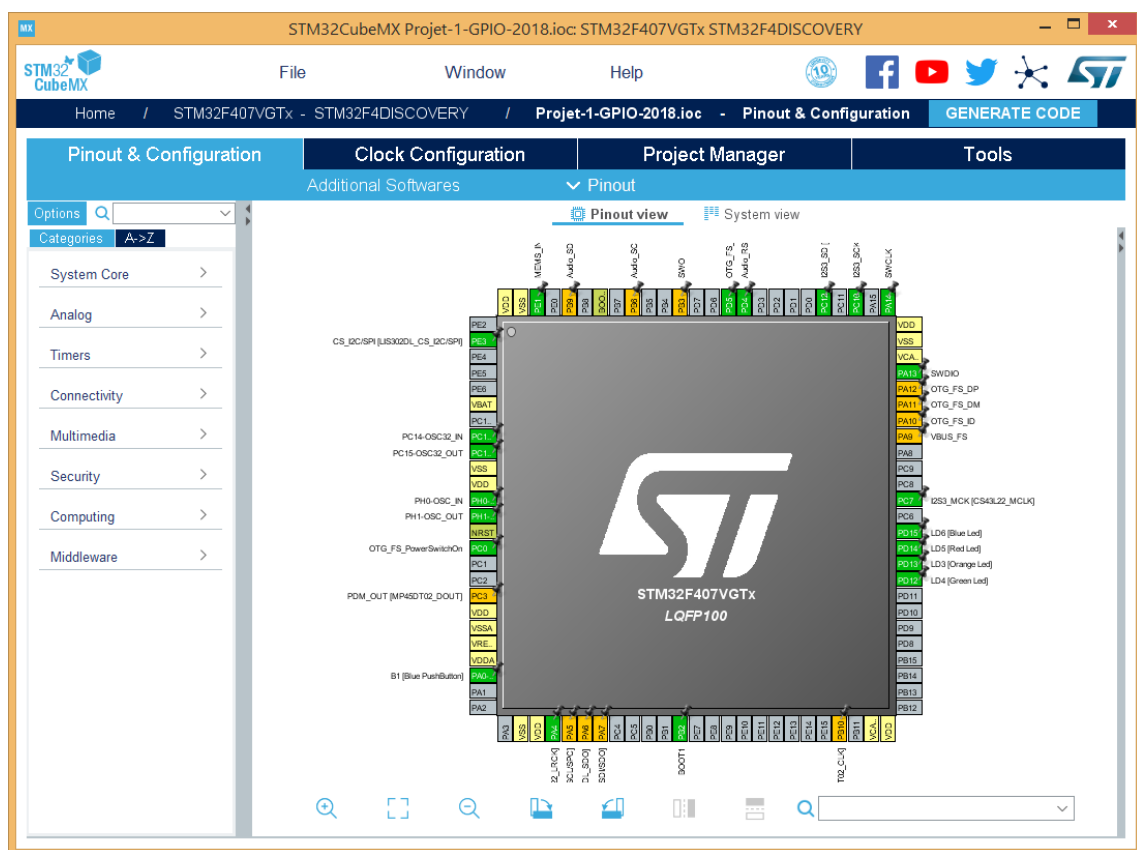
Acronymes	Définition
ADC	Analog-to-digital converter
ANSI	American National Standards Institute
API	Application Programming Interface
BSP	Board Support Package
CAN	Controller area network
CEC	Consumer Electronics Control
CMSIS	Cortex Microcontroller Software Interface Standard
CPU	Central Processing Unit
CRYP	Cryptographic processor unit
CRC	CRC calculation unit
DAC	Digital to analog converter
DCMI	Digital Camera Module Interface
DMA	Direct Memory Access
DMA2D	Chrom-Art Accelerator™ controller
DSI	Display Serial Interface
ETH	Ethernet controller
EXTI	External interrupt/event controller
FLASH	Flash memory
FSMC	Flexible Static Memory controller
FMC	Flexible Memory controller
FMPI2C	Fast-mode Plus inter-integrated circuit
GPIO	General purpose I/Os
HAL	Hardware abstraction layer
HASH	Hash processor
HCD	USB Host Controller Driver
I2C	Inter-integrated circuit
I2S	Inter-integrated sound
IRDA	InfraRed Data Association
IWDG	Independent watchdog
LPTIM	Low-power Timer
LTDC	LCD TFT Display Controller
MSP	MCU Specific Package
NAND	NAND external Flash memory
NOR	NOR external Flash memory
NVIC	Nested Vectored Interrupt Controller
PCCARD	PCCARD external memory
PCD	USB Peripheral Controller Driver
PWR	Power controller
QSPI	QuadSPI Flash memory Interface
RCC	Reset and clock controller
RNG	Random Number Generator
RTC	Real-time clock
SAI	Serial Audio Interface
SD	Secure Digital
SDRAM	SDRAM external memory
SRAM	SRAM external memory
SMARTCARD	Smartcard IC
SPDIFRX	SPDIF-RX Receiver Interface
SPI	Serial Peripheral interface
SysTick	System tick timer
TIM	Advanced-control, general-purpose or basic timer
UART	Universal asynchronous receiver/transmitter
USART	Universal synchronous receiver/transmitter
WWDG	Window watchdog
USB	Universal Serial Bus
PPP	STM32 peripheral or block

## 4.3 LES PÉRIPHÉRIQUES DU MCU

Les MCU de la famille STM32 renferment une quantité importante et diversifiée de périphériques. Non seulement ils sont nombreux mais leurs possibilités d'attachement aux broches sont multiples. Il en résulte une grande complexité lors de l'écriture des routines de configuration.

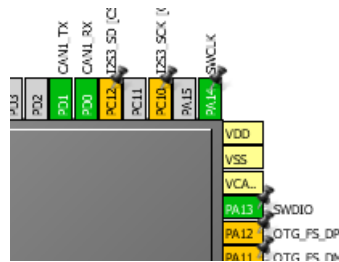
Heureusement, l'utilisation du logiciel STM32CubeMX permettra de simplifier la programmation de bas niveau de ces périphériques. En effet, ce logiciel permet de générer un projet, sous différents « Integrated Development Environment » (IDE) ou « toolchain » (uVision 5, EWARM, TrueStudio...) qui intègre les fichiers de configuration de base des périphériques sélectionnés.

L'apparence du logiciel STM32CubeMX est telle que le démontre l'image suivante :



## PINOUT & CONFIGURATION

La fenêtre de gauche renferme les périphériques du système et la fenêtre de droite le brochage du MCU. À titre d'exemple, l'indication de l'emploi d'une communication de type « Controller Area Network » (CAN), se fait par un simple crochet appliqué à la boîte à cocher du périphérique impliqué. Lorsqu'un périphérique est sélectionné dans la section gauche de la fenêtre, l'illustration du brochage du MCU présent dans la section droite est modifiée afin de respecter cette sélection ajoutée.



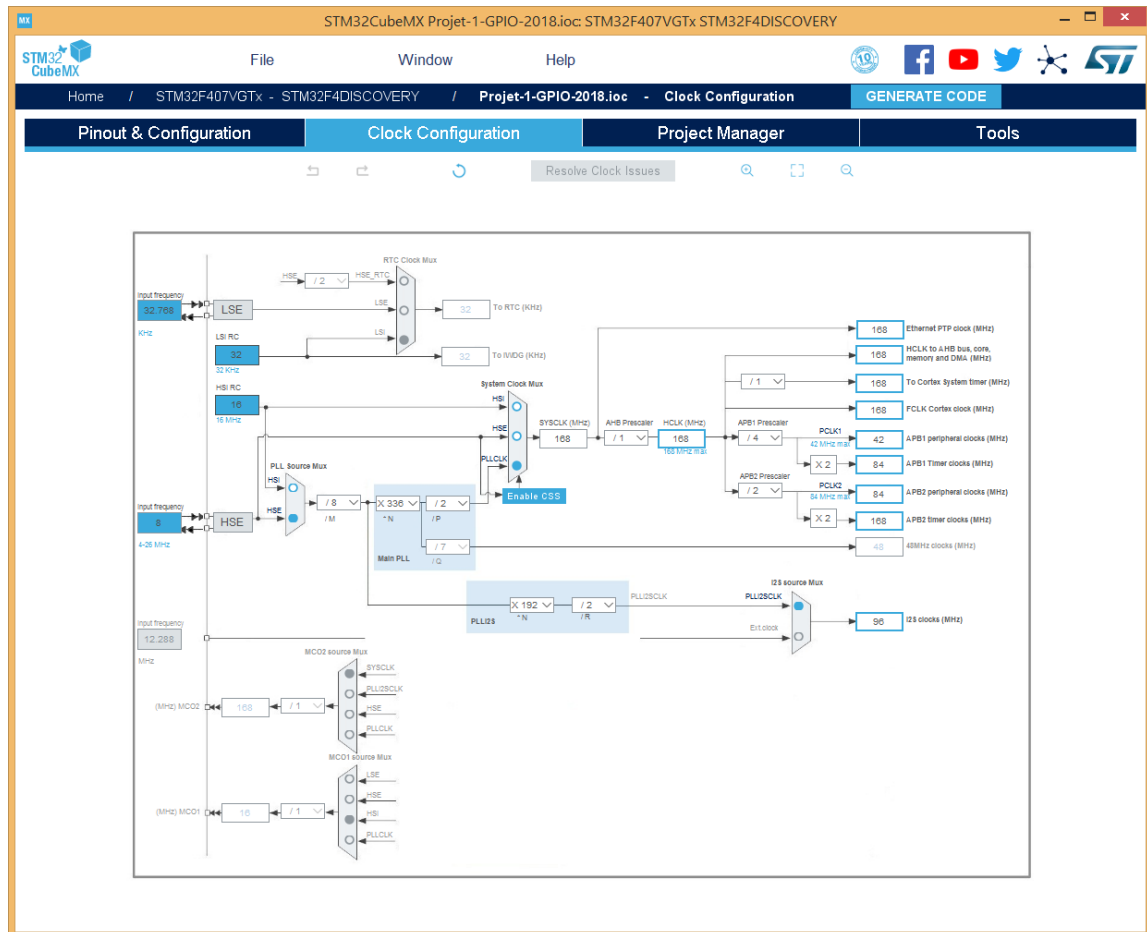
Le classement des périphériques se fait par catégories ou par ordre alphabétique. 8 catégories sont présentes et chacune d'elles possède un ou plusieurs périphériques:

- System Core : DMA, GPIO, NVIC, RCC, SYS et WWDG
- Analog : ADC et DAC
- Timers : RTC et TIMx
- Connectivity : CAN, ETH, FSMC, I2C, SDIO, SPI, UART, USART, USB
- Multimédia : DCMI, I2S
- Security : RNG
- Computing : CRC
- Middleware : FATFS, FREERTOS, LIBJPEG, LWIP, USB

Il arrive qu'un symbole de limitation ⚠️ ou de non-disponibilité 🚫 apparaisse à côté d'un périphérique de la liste. Cela signifie que il y a impossibilité d'employer certaines fonctionnalités (limitation) ou tout simplement tout le périphérique (non-disponibilité). La plupart du temps la cause provient du fait qu'une broche nécessaire au bon fonctionnement du périphérique est employée à une autre fonctionnalité du MCU.

## CLOCK CONFIGURATION

L'onglet « Clock Configuration » permet de modifier les fréquences d'horloges associées au MCU, aux périphériques et aux groupes de périphériques. Les processeurs ARM se distinguent, entre autres, par leur capacité d'économie d'énergie. Cette gestion de l'énergie passe par l'utilisation d'un « Real-time Clock Controller » (RCC) qui cadence ou non les périphériques du système. Ainsi, il est possible de placer en arrêt un périphérique en cessant de lui appliquer l'horloge lui étant associée. Mais cette capacité à limiter la consommation du courant passe par une gestion serrée du RCC et complexifie les manipulations de paramétrage du système. Fort heureusement, le STM32CubeMX facilite le travail de l'utilisateur ici aussi.



## PROJECT MANAGER

L'onglet « Project Manager » permet de spécifier le nom du projet, l'IDE avec lequel on compilera ainsi que les paramètres de « Stack » et de « Heap » du linker.

C'est aussi sous cet onglet que nous pourrions générer le code pour une première fois ou suite à tout changement apporté à notre projet.

## TOOLS

Ce dernier onglet apportera une aide précieuse pour l'estimation de la consommation énergétique du système. Il sera possible, selon les périphériques activés, la température ambiante et le type de batterie sélectionner, le temps de fonctionnement des périphériques, etc. de déterminer la durée de la batterie.

### 4.3.1 Les GPIO

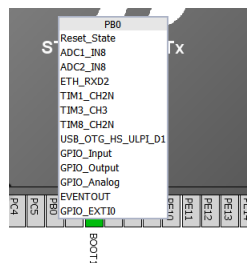
#### *Preamble.*

L'élément le plus simpliste de tous les périphériques d'un MCU est sans aucun doute le port d'entrée-sortie. Malgré cette apparence de simplicité, on devra être prudent et minutieux lors de la configuration des broches d'entrées-sorties car chacune des broches peut se voir attribuer une multitude de fonctions. Voyons l'emploi des GPIO\_Input et GPIO\_Output.

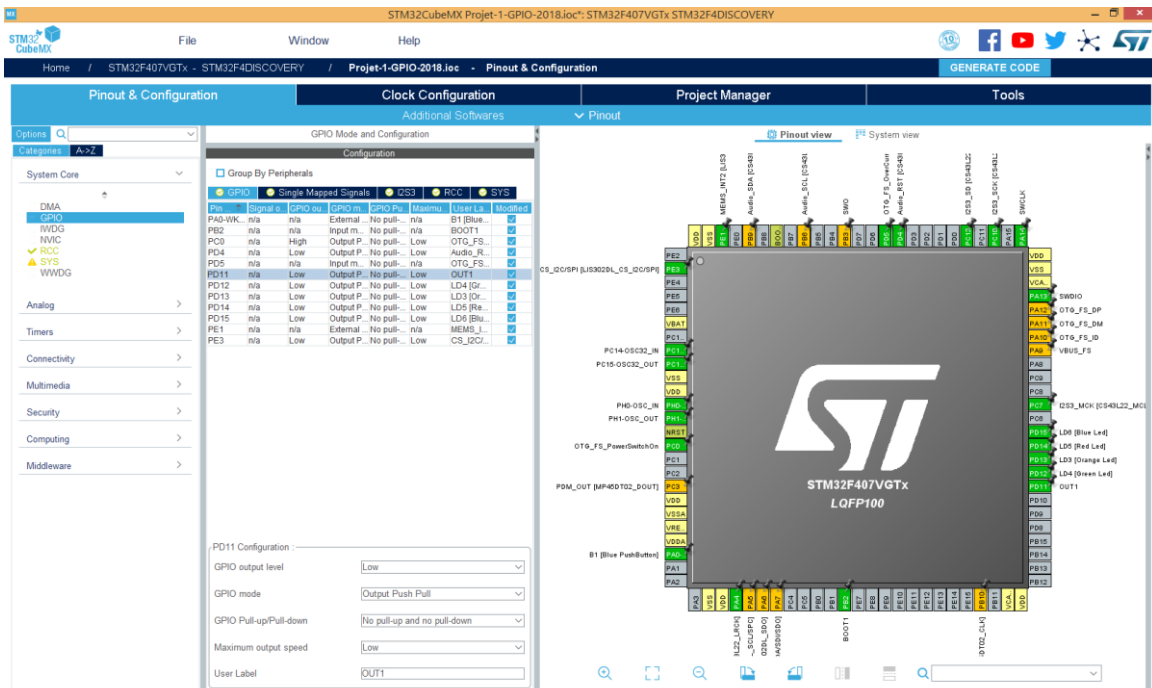
#### *Configuration dans le STM32CubeMX.*

La sélection d'une broche du MCU apparaissant à droite de la fenêtre principale du logiciel STM32CubeMX nous permet de voir les fonctions qui y sont associées.

Dans l'exemple, la représentation des fonctions de la broche PB0 est donnée. Cette broche peut-être utile à certains convertisseurs, à une communication Ethernet, à un canal de timer, à une communication USB, à une entrée d'interruption externe ou encore à un élément de GPIO d'entrée-sortie.



Pour chacun des périphériques, incluant les GPIO, une fenêtre « Mode and Configuration » permet de peaufiner leur configuration. Dans l'exemple suivant, la broche PD11 est configurée en sortie « push pull », sans « pull-up » et son niveau sera à zéro après une remise à zéro du système.



Voici trois exemples de configuration de GPIO :

Configuration en GPIO de sortie avec sortie "Push-Pull" pour activation d'une del.

```
GPIO_InitStruct.Pin = GPIO_PIN_12 | GPIO_PIN_13 | GPIO_PIN_14 | GPIO_PIN_15;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP; GPIO_InitStruct.Pull = GPIO_PULLUP;
GPIO_InitStruct.Speed = GPIO_SPEED_FAST; HAL_GPIO_Init(GPIOD, &GPIO_InitStruct);
```

Configuration de PA0 en EXTI déclenché sur front descendant.

```
GPIO_InitStructure.Mode = GPIO_MODE_IT_FALLING; GPIO_InitStructure.Pull = GPIO_NOPULL;
GPIO_InitStructure.Pin = GPIO_PIN_0; HAL_GPIO_Init(GPIOA, &GPIO_InitStructure);
```

Configuration de la broche Tx du USART3 (PC10 est mappé sur AF7) en fonction alternative.

```
GPIO_InitStruct.Pin = GPIO_PIN_10;
GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
GPIO_InitStruct.Pull = GPIO_PULLUP;
GPIO_InitStruct.Speed = GPIO_SPEED_FAST;
GPIO_InitStruct.Alternate = GPIO_AF7_USART3;
HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
```

Et un exemple d'activation d'horloge pour le groupe de GPIOD :

```
__HAL_RCC_GPIOD_CLK_ENABLE();
```

La beauté dans l'utilisation du STM32CubeMX c'est que lorsque les paramètres de configuration sont bien entrés, toutes les lignes de code permettant de gérer les périphériques sont générées dans un projet compatible avec l'IDE choisi. Il ne reste donc, lorsque toute la configuration est faite, qu'à utiliser les API associées.

Les API GPIO des librairies HAL sont les suivantes:

- HAL\_GPIO\_Init (),
- HAL\_GPIO\_DeInit (),
- HAL\_GPIO\_ReadPin (),
- HAL\_GPIO\_WritePin (),
- HAL\_GPIO\_TogglePin ().

Le tableau ci-dessous décrit la structure d'initialisation des données : GPIO\_InitTypeDef.

Structure field	Description
Pin	Spécifie le GPIO à configurer. Valeur possible : GPIO_PIN_x ou GPIO_PIN_All, où x[0..15] ou une valeur correspondant à l'étiquette appliqué au Cube. EX : si l'étiquette RD est donnée à une broche, son nom pourra être RD_Pin.
Mode	Spécifie le mode d'opération d'une broche sélectionnée : GPIO mode ou EXTI mode. Les valeurs possibles sont : GPIO mode <ul style="list-style-type: none"> <li>• GPIO_MODE_INPUT : Input Floating</li> <li>• GPIO_MODE_OUTPUT_PP : Output Push Pull</li> <li>• GPIO_MODE_OUTPUT_OD : Output Open Drain</li> <li>• GPIO_MODE_AF_PP : Alternate Function Push Pull</li> <li>• GPIO_MODE_AF_OD : Alternate Function Open Drain</li> <li>• GPIO_MODE_ANALOG : Analog mode</li> </ul> EXTI Mode <ul style="list-style-type: none"> <li>• GPIO_MODE_IT_RISING : Rising edge trigger detection</li> <li>• GPIO_MODE_IT_FALLING : Falling edge trigger detection</li> <li>• GPIO_MODE_IT_RISING_FALLING : Rising/Falling edge trigger detection</li> </ul> External Event Mode <ul style="list-style-type: none"> <li>• GPIO_MODE_EVT_RISING : Rising edge trigger detection</li> <li>• GPIO_MODE_EVT_FALLING : Falling edge trigger detection</li> <li>• GPIO_MODE_EVT_RISING_FALLING: Rising/Falling edge trigger detection</li> </ul>
Pull	Spécifie si les Pull-up ou Pull-down interne sont activés ou non. Possible values are: GPIO_NOPULL GPIO_PULLUP GPIO_PULLDOWN
Speed	Spécifie la vitesse. Possible values are: GPIO_SPEED_LOW GPIO_SPEED_MEDIUM GPIO_SPEED_FAST GPIO_SPEED_HIGH
Alternate	Spécifie le périphérique à attacher à une broche. <ul style="list-style-type: none"> <li>• Exemple : GPIO_AFx_PPP, où</li> </ul> AFx : est la fonction alternative PPP : est l'instance périphérique

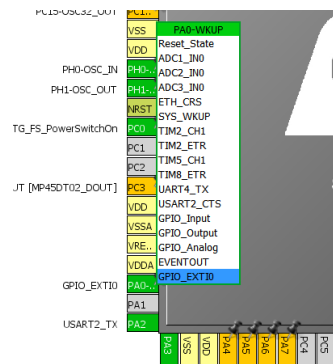
### 4.3.2 Les INT externes

#### *Preamble.*

Le STM32F407 offre la possibilité de gérer 16 entrées d'interruption externes. Toutes les broches de port A, B, C, D, E et H sont disponibles pour l'emploi d'une entrée EXTI. La seule exception est que l'on ne doit pas employer deux broches portant le même numéro. Ainsi, il est possible d'employer PA0, PD1 et PC3 comme éléments EXTI0, EXTI1 et EXTI3 mais il sera impossible d'attribuer une fonction EXTI à toute autre broche étant définie du numéro 0, 1 ou 3, lorsque ces trois broches auront déjà été sélectionnées.

#### *Configuration dans le STM32CubeMX.*

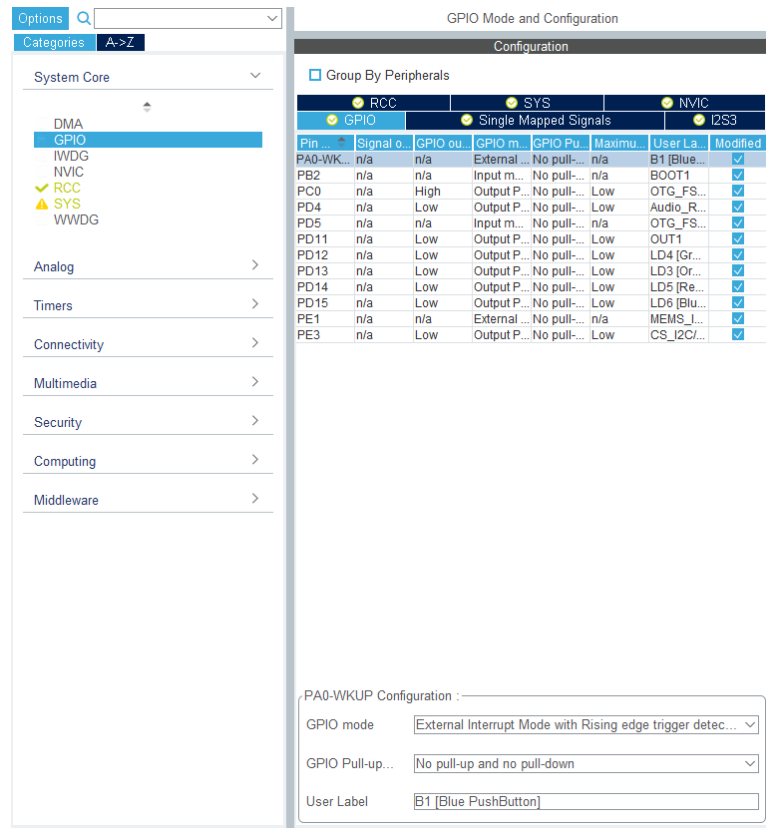
Tout comme dans le cas des GPIO, les entrées d'interruption externe (EXTI) se définissent en paramétrant directement la broche du MCU. Lorsque l'on clique sur le bouton de la souris tout en pointant une broche du MCU, un menu de choix de fonctions apparaît :



En sélectionnant la fonction GPIO\_EXTI0, de la broche PA0, vous indiquez au STM32CubeMX d'utiliser la broche PA0 en entrée permettant de générer une interruption. Mais la configuration ne s'arrête pas là...



Toujours sous l'onglet « Pinout & Configuration », du bloc « System Core », en choisissant le sous bloc GPIO et finalement la broche qui nous intéresse dans la liste, il est possible de spécifier le « GPIO mode » et d'appliquer une « pull-up » ou « pull-down » lorsque requis.



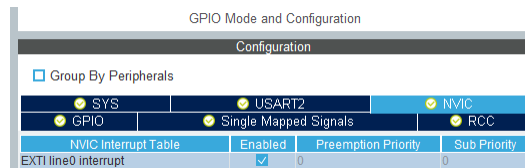
En ce qui a trait au « GPIO mode », on doit comprendre la différence entre les modes « Interrupt » et « Event ». En fait, les interruptions déclenchent l'exécution d'un gestionnaire d'interruption, alors que l'événement ne le fait pas. La résultante est qu'aucun code n'est intégré au fichier « stm32f4xx\_it.c » lorsque le mode « Event » est sélectionné. À l'opposé, lorsque le mode « Interrupt » est sélectionné et que toutes les configurations sont bien faites, le code suivant est intégré au fichier « stm32f4xx\_it.c » :

```
void EXTI0_IRQHandler(void)
{
    /* USER CODE BEGIN EXTI0_IRQn 0 */

    /* USER CODE END EXTI0_IRQn 0 */
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_0);
    /* USER CODE BEGIN EXTI0_IRQn 1 */
    HAL_GPIO_TogglePin(GPIOD, LD4_Pin);    //Led verte.

    /* USER CODE END EXTI0_IRQn 1 */
}
```

Pour une configuration complète ayant pour but l'exécution d'une routine d'interruption déclenchée sur un changement d'état d'une broche d'entrée choisie à cette fin, l'utilisateur doit informer le « NVIC » du déclenchement d'une interruption en cochant la case « Enable » de la fenêtre « NVIC Configuration ». Cette fenêtre se retrouve sous l'onglet configuration, dans le bloc système, en choisissant le sous bloc NVIC.



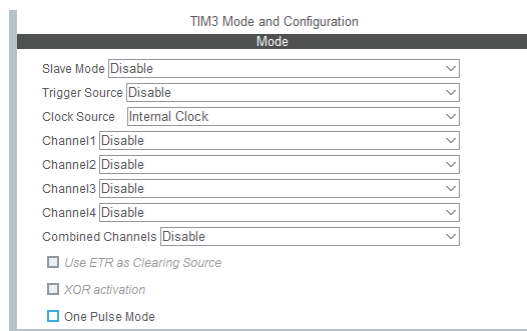
### 4.3.3 Les timers

#### *Preamble.*

Plusieurs « timers » sont disponibles dans le STM32F4. Le MCU renferme, entre autre, 12 « timers » 16 bits et 2 « timers » 32 bits. Ces « timers » offrent la possibilité de générer des interruptions et permettent la génération de signaux tels les PWM.

#### *Configuration dans le STM32CubeMX.*

La démarche permettant de configurer un « timer » ne comporte aucun attachement à une broche du MCU ce qui la rend relativement simple. Il s'agit d'activer la source de l'horloge d'un des « timer » dans la fenêtre des périphériques.



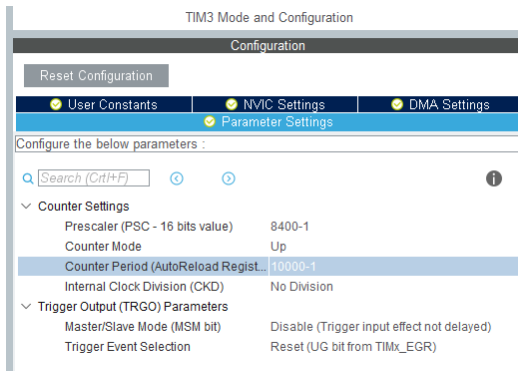
Dans cet exemple, le « timer » 3 est activé par la source « Internal Clock ». Ne restent que deux petites configurations à compléter pour que le « timer » gère une routine d'interruption : soit celle du « timer » 3 proprement dit et celle du « NVIC ».

Si nous regardons à la fenêtre « Clock Configuration » la fréquence d'horloge spécifiée pour les périphériques « APB1 timer », nous remarquons qu'elle est fixée à 84 MHz. Si l'utilisateur désire travailler avec des temporisations de l'ordre du dixième de milliseconde, le « Prescaler » pourrait être ajusté à 8400. Pour atteindre un compte d'une seconde, le « Counter Period » devrait être chargé avec la valeur 10 000 car  $84\,000\,000 / 8\,400 = 10\,000$ . Alors, chacune des unités ajoutées au « Counter Period » représentera 1 / 10 000 de seconde ou 100 usec.

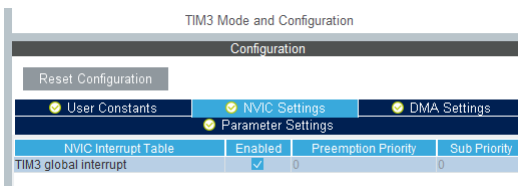
Notez que pour une valeur juste, on doit retirer une unité des nombres que l'on passe en paramètre dans la fenêtre de configuration du « timer ».

Il est important de tenir compte de la valeur maximale possible pour les éléments « Prescaler » et « Counter Period » qui est de 65535, limite imposé par les registres de 16 bits employés.

L'illustration suivante représente la fenêtre de configuration avec les valeurs utilisées dans l'exemple précédent.



Il est nécessaire de configurer adéquatement le « NVIC » pour qu'une routine d'interruption soit ajoutée au fichier « stm32f4xx\_it.c ». Dans ce cas-ci, la case « Enabled » du « timer » 3 doit être activée.



Lors de la génération des fichiers par le STM32CubeMX, le code de la routine d'interruption du « timer » sera intégré au fichier « stm32f4xx\_it.c ». Ce code ressemblera à ceci :

```
void TIM3_IRQHandler(void)
{
    /* USER CODE BEGIN TIM3_IRQn 0 */

    /* USER CODE END TIM3_IRQn 0 */
    HAL_TIM_IRQHandler(&htim3);
    /* USER CODE BEGIN TIM3_IRQn 1 */
    HAL_GPIO_TogglePin(GPIOD, LED3_Pin);    //Ligne ajoutée par l'utilisateur.
    /* USER CODE END TIM3_IRQn 1 */
}
```

C'est la responsabilité de l'utilisateur de compléter la routine en y ajoutant le code qu'il désire voir exécuter lors de l'appel de la routine. Dans l'exemple, une ligne de changement d'état de la broche LED3 est ajoutée.

En dernier lieu, l'utilisateur devra démarrer le « timer » grâce à la fonction HAL de l'API du « timer » suivante :

```
HAL_TIM_Base_Start_IT(&htim3);
```

Les API des « timer » sont nombreuses car une multitude de modes de fonctionnement sont disponibles. La référence UM1725 de ST Microelectronic en offre la liste complète.

#### 4.3.4 Les PWM

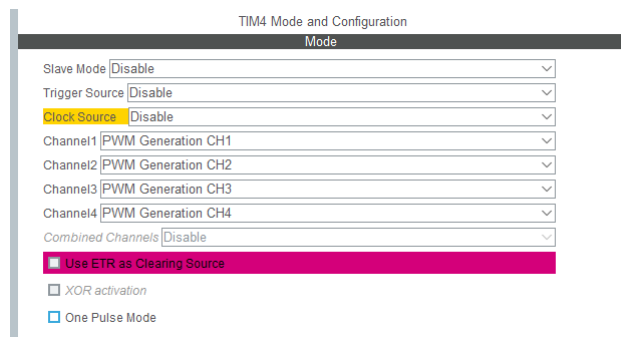
##### *Préambule.*

Il est possible de gérer plusieurs canaux de PWM avec certains des « Timers » intégrés au STM32F4. Les « Timers » 1 (1 canal), 2 (4 canaux), 3 (4 canaux), 4 (4 canaux), 5 (4 canaux), 8 (1 canal), 9 (2 canaux) et 12 (2 canaux) offrent cette possibilité pour un total de 22 canaux.

##### *Configuration dans le STM32CubeMX.*

Pour faire la démonstration d'une configuration de PWM, nous utiliserons le « timer » 4 car les canaux attachés à ce dernier sont reliés aux broches pilotant les quatre delds du circuit. Nous pourrions ainsi visualiser rapidement le fonctionnement du PWM.

La première étape consiste à indiquer au STM32CubeMX que les canaux du « timer » 4 sont attribués au PWM.



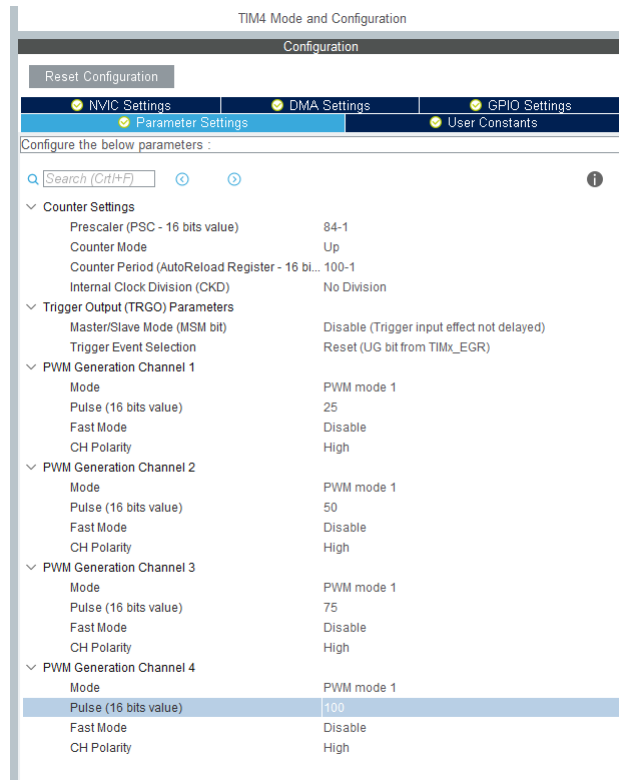
Ensuite, on doit configurer les paramètres du « timer » 4 afin de spécifier la fréquence d'utilisation et le cycle de service de chacun des canaux. La fréquence est déterminée avec le « Prescaler » et le « Counter Period ». La fréquence des PWM sera de :

$$\text{APB 1 Timer clocks} / (\text{Prescaler} * \text{Counter Periode})$$

Pour notre exemple, nous fixons le « Prescaler » à (84-1) et le « Counter Periode » à (100-1). La fréquence du PWM sera donc de 10 000 Hz. (84MHz / 84 / 100)

Ne reste qu'à spécifier le temps actif du cycle de service de chacun des canaux du PWM. Les cycles de services ont été fixés à 25, 50, 75 et 100 %.

La valeur attribuée au « Pulse » de chacun des canaux doit être égale ou inférieure à la valeur du « Counter Periode ».



Une fois le projet généré, les canaux PWM doivent être amorcés. La fonction `HAL_TIM_PWM_Start()`; permet cela. Les paramètres d'entrée de cette fonction sont le « handle » du « timer » ainsi que le canal en cause. Pour le canal 1 du « timer » 4, la ligne de fonction complète serait :

```
HAL_TIM_PWM_Start (&htim4, TIM_CHANNEL_1);
```

Pour modifier le cycle de service en cour d'exécution du programme, il est possible de modifier la valeur du pulse par la commande suivante :

```
TIM4->CCR1 = 50; //Permet de placer à 50%
```

Cette ligne de commande permettrait de placer le cycle de service du canal 1 à 50%.

### 4.3.5 Les ADC

#### *Préambule.*

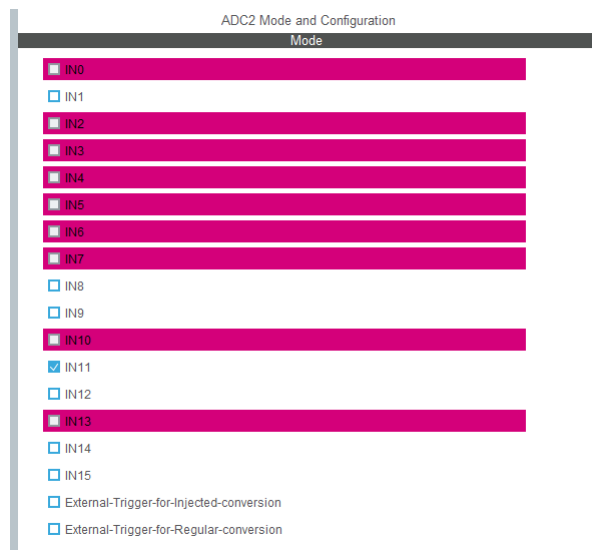
Le STM21F407 offre trois convertisseurs ADC à canaux multiples. Les deux premiers ADC se découpent en 16 canaux multiplexés alors que l'ADC3 n'offre que 8 entrées. Les broches associées aux différents canaux sont les suivantes :

ENTRÉE	ADC1	ADC2	ADC3
IN0	PA0	PA0	PA0
IN1	PA1	PA1	PA1
IN2	PA2	PA2	PA2
IN3	PA3	PA3	PA3
IN4	PA4	PA4	---
IN5	PA5	PA5	---
IN6	PA6	PA6	---
IN7	PA7	PA7	---
IN8	PB0	PB0	---
IN9	PB1	PB1	---
IN10	PC0	PC0	PC0
IN11	PC1	PC1 (Kit)	PC1
IN12	PC2	PC2	PC2 (Kit)
IN13	PC3	PC3	PC3
IN14	PC4	PC4	---
IN15	PC5	PC5	---

Il est aussi possible d'employer des entrées spéciales sur l'ADC1 afin de lire la température du MCU, la tension de référence et la tension à l'entrée VBat.

#### *Configuration dans le STM32CubeMX.*

L'activation d'un canal de lecture ADC se fait à partir de la fenêtre « Mode and Configuration » des périphériques, par le choix du convertisseur et l'attribution d'un canal.



Dans l'exemple illustré ci-haut, le choix s'est arrêté sur le canal IN11 correspondant à la broche PC1 du MCU pour le convertisseur intégré ADC2. La génération du code faite par l'application STM32CubeMX configurera l'ADC2 et vous permettra d'utiliser les fonctions de l'API HAL\_ADC. En mode « Pooling », les fonctions nécessaires à l'usage du convertisseur seront :

- HAL\_ADC\_Start()
- HAL\_ADC\_Stop()
- HAL\_ADC\_PollForConversion()
- HAL\_ADC\_GetValue()

Et le paramètre principal sera le « handle » du convertisseur, soit le hadc2.

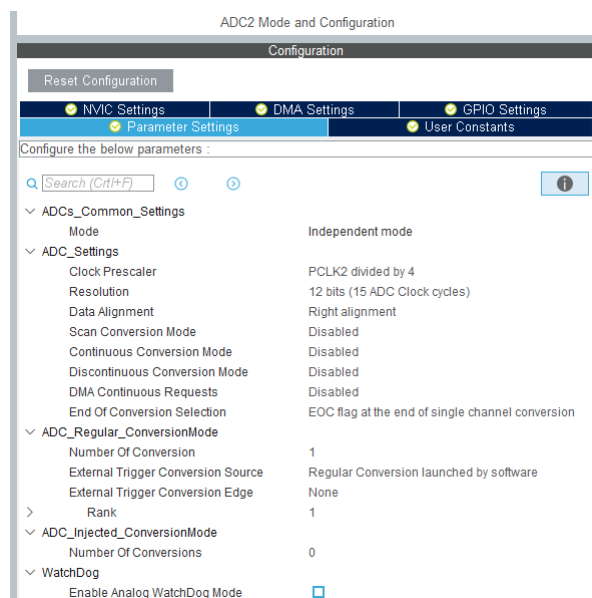
Dans l'exemple, l'utilisation des fonctions de l'API de l'ADC pourrait se faire comme suit.

```
HAL_ADC_Start(&hadc2); //Démarrer la conversion.
HAL_ADC_PollForConversion(&hadc2,100); //Attendre la fin de la conversion.
uiValeurADC2 = HAL_ADC_GetValue(&hadc2); //Lire la donnée.
HAL_ADC_Stop(&hadc2); //Arrêter la conversion.

sprintf(cADC2,"%X",uiValeurADC2);
```

La dernière ligne de ce court programme permet de prendre la valeur du convertisseur et de placer le résultat en valeur hexadécimale dans un tableau de « char » nommé cADC2.

La configuration du canal du convertisseur doit être faite et si la conversion est initiée par le logiciel, on doit le spécifier.





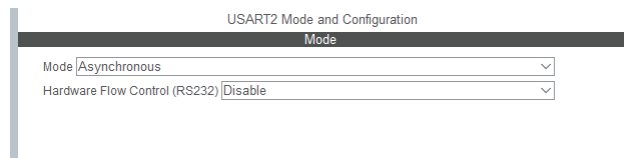
### 4.3.6 Les USART

#### Préambule.

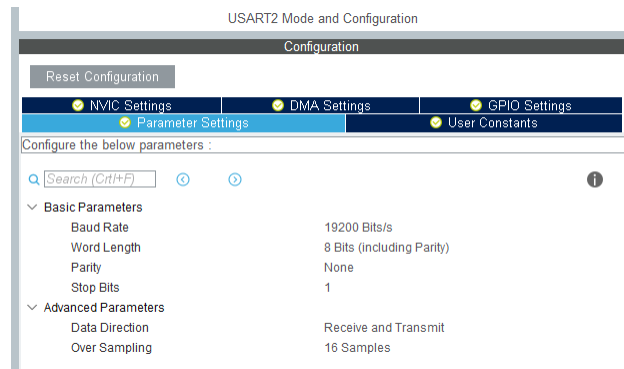
Plusieurs ports de communication de type UART/USART (Universal Synchronous Asynchronous Receiver Transmitter) sont présents dans le MCU : Quatre ports USART et deux ports UART. Le port USART2 est choisi pour l'attachement d'un dispositif « Bluetooth » sur le support STM32F4. Les broches PA2 (TX) et PA3 (RX) composent ce port utilisé sans contrôle de flux.

#### Configuration dans le STM32CubeMX.

Il suffit de spécifier le « Mode » et le « Hardware Flow Control ». (onglet pinout fenêtre de gauche, USART2)



et ensuite spécifier les paramètres du protocole de communication à l'onglet configuration dans le bloc connectivity au sous bloc USART2, pour qu'une configuration complète de ce port soit faite. Dans l'exemple suivant, le protocole de communication est fixé à 19200, n, 8, 1.



Les deux principales fonctions de l'API de communication série sont les suivantes :

- HAL\_UART\_Receive();
- HAL\_UART\_Transmit();

Dans les deux cas, les paramètres d'entrée des fonctions sont :

- L'adresse du handle (huart2),
- un pointeur de type uint8\_t,
- le nombre de caractères à émettre ou à recevoir,
- un timeout. Ce dernier pourra être fixé arbitrairement à 100.

Un exemple d'appel ressemblerait à ceci :

```
HAL_UART_Receive(&huart2, (uint8_t*)ucRxUart, 2, 10);  
HAL_UART_Transmit(&huart2, (uint8_t*)ucTxUart, 2, 10);
```

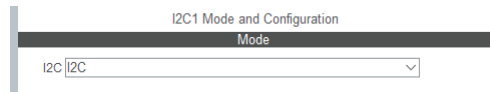
### 4.3.7 La communication I2C

#### *Preamble.*

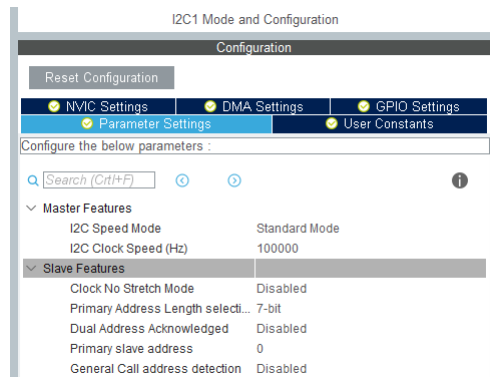
Trois canaux de communication I2C sont disponibles sur le MCU intégré au support de développement et le canal I2C1 est choisi pour l'attachement à deux connecteurs I2C simples, un connecteur pour clavier I2C et un connecteur prévu pour la carte I2C complète. Le signal SDA est piloté par la broche PB9 et le signal SCL par la broche PB6.

#### *Configuration dans le STM32CubeMX.*

La configuration dans le STM32CubeMX se fait aisément en sélectionnant le protocole I2C sous le périphérique I2C1.



Et la configuration pour les dispositifs que nous employons au collège devrait être comme suit :



La génération du code faite par l'application STM32CubeMX configurera l'I2C et vous permettra d'utiliser les fonctions de l'API HAL\_I2C. En mode « Pooling », les fonctions nécessaires à l'usage du canal I2C seront :

Pour les dispositifs courant.

- HAL\_I2C\_Master\_Transmit()
- HAL\_I2C\_Master\_Receive()

Pour les dispositifs mémoire.

- HAL\_I2C\_IsDeviceReady()
- HAL\_I2C\_Mem\_Write ()
- HAL\_I2C\_Mem\_Read ()

Des fonctions maison sont disponibles pour cinq dispositifs électroniques :

- Mémoire : 24C32, (fichier 2432.o et 2432.h)
- RTC : DS1307, (fichier 1307.o et 1307.h)
- I/O : PCF8574, (fichier 8574.o et 8574.h)
- ADC : MAX1236, (fichier 1236.o et 1236.h)
- DAC : DAC6574. (fichier 6574.o et 6574.h)

Pour employer les fonctions de ces fichiers, il suffit de copier les fichiers .h et .o dans les répertoires appropriés de votre projet (.h dans le répertoire « inc » et .o dans le répertoire « obj ») et de spécifier l'inclusion du fichier dans le main.c.

```
"#include 1236.h"
```

### 4.3.8 La communication SPI

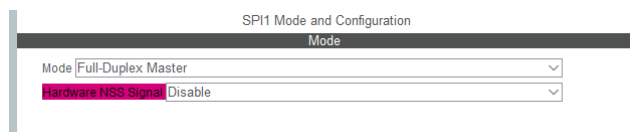
#### *Preamble.*

Trois canaux de communication SPI sont disponibles sur le MCU intégré au support de développement et le canal SPI1 est choisi pour l'attachement à un connecteur prévu pour la carte SPI complète. Le signal MISO est piloté par la broche PA6, le signal MOSI par la broche PA7, le signal SCL par la broche PA5 et les « Chip Select » sont pilotés par :

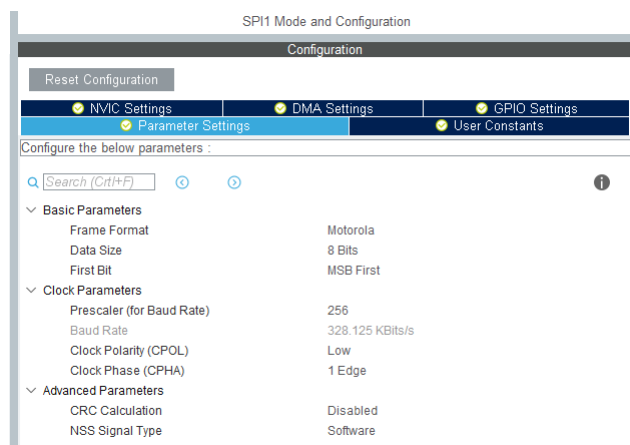
- I/O : MCP23S09, PE5 (SPI\_CS\_3\_Pin)
- ADC : TLV2544, PE4 (SPI\_CS\_2\_Pin)
- DAC : DAC7554, PE6 (SPI\_CS\_4\_Pin)
- INCLINO : LIS3DSH PE3

#### *Configuration dans le STM32CubeMX.*

La configuration dans le STM32CubeMX se fait aisément en sélectionnant le mode « Full-Duplex Master » sous le périphérique SPI1.



Et la configuration ressemblera à ceci :



La génération du code faite par l'application STM32CubeMX configurera le SPI et vous permettra d'utiliser les fonctions de l'API HAL\_SPI. En mode « Pooling », les fonctions nécessaires à l'usage du canal SPI seront :

Pour les dispositifs courant.

- HAL\_SPI\_GetState ()
- HAL\_SPI\_Transmit ()
- HAL\_SPI\_Receive()
- HAL\_SPI\_TransmitReceive()

Des fonctions sont disponibles pour trois dispositifs électroniques :

- I/O : MCP23S09, (fichier 23S09.o et 23S09.h)
- ADC : TLV2544, (fichier 2544.o et 2544.h)
- DAC : DAC7554. (fichier 7554.o et 7554.h)

Pour employer les fonctions de ces fichiers, il suffit de copier les fichiers .h et .o dans les répertoires appropriés de votre projet (.h dans le répertoire « inc » et .o dans le répertoire « obj ») et de spécifier l'inclusion du fichier dans le main.c.

```
#include 23S09.h"  
#include 2544.h"  
#include 7554.h"
```

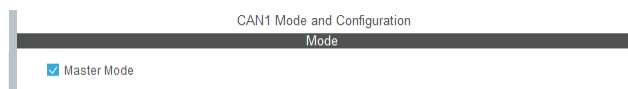
### 4.3.9 La communication CAN

#### Préambule.

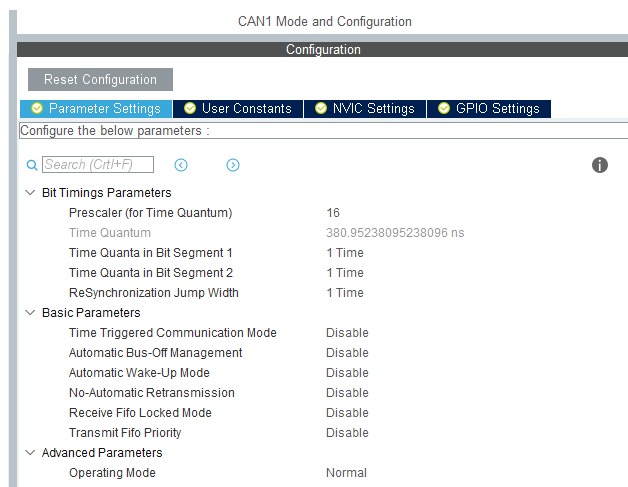
Deux ports CAN sont disponibles sur le MCU STM32F407. Le port CAN1 est choisi sur la carte support STM32F4 et les broches PD0 (CAN1\_RX) et PD1 (CAN1\_TX) y sont attachées.

#### Configuration dans le STM32CubeMX.

La démarche permettant de configurer le CAN Bus est fort simple. Dans un premier temps, comme pour l'ensemble des périphériques du MCU, on active le périphérique dans la liste prévue à cette fin.



Ensuite, on doit valider les paramètres du « Prescaler » et du mode d'opération. (onglet configuration / bloc connectivity / sous bloc CAN1)



Une routine d'initialisation de base (vInit\_CAN\_Bus()) est disponible au fichier CAN\_Bus.o. Ce fichier est accompagné de son fichier de définition CAN\_Bus.h. Pour utiliser la routine d'initialisation, on doit faire la copie des fichiers CAN\_Bus dans les répertoires appropriés de votre projet (.h dans le répertoire « inc » et .o dans le répertoire « obj ») et l'ajouter au projet IAR avec la commande « ADD FILE ». La routine permet, entre autre, de définir un « Filter » pour la communication CAN.

On doit inclure le fichier de définition au fichier main.c :

```
#include "CAN_Bus.h"
```

et lancer la fonction d'initialisation au début de la fonction main() du programme.

```
vInit_CAN_Bus();
```

Notez que la fonction vInit\_CAN\_Bus() doit être lancée après que l'initialisation de tous les périphériques soit faite dans le fichier main.c.

En dernier lieu, on doit préparer la structure de transmission comme suit :

```
/* Préparation de la structure de transmission */
hcan1.pTxMsg->StdId = 0x321;
hcan1.pTxMsg->ExtId = 0x01;
hcan1.pTxMsg->RTR = CAN_RTR_DATA;
hcan1.pTxMsg->IDE = CAN_ID_STD;
hcan1.pTxMsg->DLC = 8; //Data Length Code
```

En mode « Pooling », les fonctions nécessaires à l'usage du canal CAN seront :

Pour les dispositifs courant.

- HAL\_CAN\_GetState ()
- HAL\_CAN\_Transmit ()
- HAL\_CAN\_Receive()

Les données de transmission et de réception sont toujours intégrées à une structure. On y accède de la manière suivante :

```
hcan1.pTxMsg->Data[0] = ucData; //Pour le « buffer » transmission.
aRxBuffer[i] = hcan1.pRxMsg->Data[i]; //Pour le « buffer » réception.
```

Voici un exemple de réception :

```
ucData = HAL_CAN_Receive (&hcan1, CAN_FIFO0, 10);
if (hcan1.pRxMsg->Data[0] != 0)
{
    for (i=0; i<8; i++)
    {
        aRxBuffer[i] = hcan1.pRxMsg->Data[i];
    }
    HAL_UART_Transmit(&huart2, (uint8_t*)aRxBuffer, 8, 10);
    aRxBuffer[0] = 0;

    hcan1.pRxMsg->Data[0] = 0;
```



Et un exemple de transmission :

```
hcan1.pTxMsg->Data[0] = ucData;
hcan1.pTxMsg->Data[1] = 0x31;
hcan1.pTxMsg->Data[2] = 0x32;
hcan1.pTxMsg->Data[3] = 0x33;
hcan1.pTxMsg->Data[4] = 0x34;
hcan1.pTxMsg->Data[5] = 0x35;
hcan1.pTxMsg->Data[6] = 0x36;
hcan1.pTxMsg->Data[7] = 0x37;
HAL_CAN_Transmit(&hcan1, 10);
HAL_CAN_GetState(&hcan1);           //On doit relire l'état pour continuer.
```

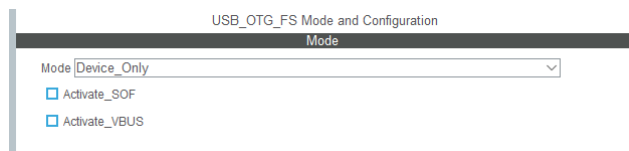
#### 4.3.10 La communication USB (USB DEVICE)

##### *Préambule.*

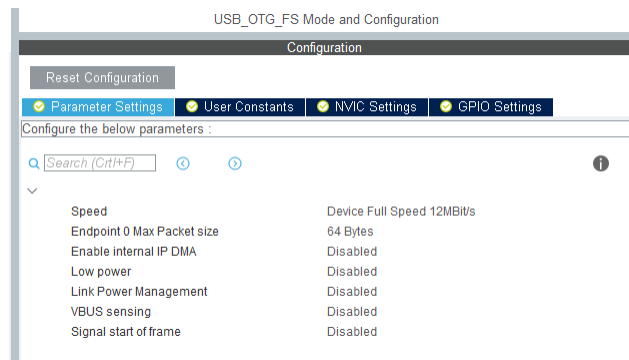
Plusieurs modes de fonctionnement USB sont offerts avec les MCU de la compagnie ST. Il est possible de travailler en « USB OTG », en « Device » ou en « Host » et ce en FS ou en HS. Par contre, l'utilisation de la carte « Discovery » F4 limite au FS car des conflits existent avec d'autres périphériques de la carte.

##### *Configuration dans le STM32CubeMX.*

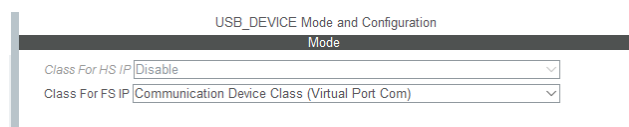
Pour l'utilisation d'un « device » « Virtual Com Port » (VCP), la première étape est d'informer le STM32CubeMx par l'activation du mode « Device\_Only » dans la fenêtre de la configuration des « Peripherals » sous le périphérique USB\_OTG\_FS présent dans la catégorie « Connectivity ».



La configuration quant à elle, sera comme suit :



On doit par la suite activer le mode « VCP » dans la « Class For FS IP » que l'on retrouve sous le périphérique « USB\_DEVICE » dans la catégorie « Middleware ».



Avant d'aller plus loin, l'utilisateur devra modifier à la hausse la grosseur de la « STACK » et du « HEAP » pour permettre le bon fonctionnement du VCP. Cette étape permettra d'accorder suffisamment d'espace mémoire au traitement complexe de la pile du port USB. La modification se fait dans la fenêtre « Project Manager » du logiciel STM32CubeMX.

Les nouvelles valeurs accordées seront de 0x400 pour le « HEAP » et 0x2000 pour la « STACK ». Il ne faut pas hésiter à gonfler ces valeurs lorsque le port n'est pas reconnu par le PC auquel il est branché.

Après la génération des fichiers avec le STM32CubeMX, les fonctions de transmission et de réception USB seront présentes dans le fichier `usbd_cdc_if.c`. On retrouvera aussi des « Buffers » de réception et de transmission associés aux deux fonctions :

```
uint8_t UserRxBufferFS[APP_RX_DATA_SIZE];
uint8_t UserTxBufferFS[APP_TX_DATA_SIZE];
```

La routine de réception s'active de manière autonome lors de la réception d'un caractère sur le port USB. Pour que l'utilisateur soit informé d'une réception de caractère, il devra créer une variable « Flag » et l'activer dans la routine de réception. L'utilisateur devra avoir accès à la variable « Flag » dans la fonction `main()` afin de traiter le caractère reçu. La détection de la présence d'un caractère dans le « Buffer » de réception du VCP se fera par « Polling » du « Flag ». Après la lecture du caractère présent dans le tableau `UserRxBufferFS`, l'utilisateur devra désactiver le « Flag » pour permettre de détecter un nouveau caractère entrant.

La variable `ucFlagRxCDC` est créée dans le fichier `usbd_cdc_if.c` pour les besoins de l'exemple.

```
/* USER CODE BEGIN EXPORTED_VARIABLES */
//Flag pour aviser qu'il y a eu réception d'un caractère.
uint8_t ucFlagRxCDC = 0x00;
/* USER CODE END EXPORTED_VARIABLES */
```

Après coup, on doit gérer l'activation du « flag » en l'ajoutant à la routine de réception. Le résultat ressemblera à ceci :

```
static int8_t CDC_Receive_FS (uint8_t* Buf, uint32_t *Len)
{
    /* USER CODE BEGIN 6 */
    USBDCDC_SetRxBuffer(hUsbDevice_0, &Buf[0]);
    USBDCDC_ReceivePacket(hUsbDevice_0);
    ucFlagRxCDC = 0xFF;           //Flag représentant la réception d'un caractère.
    return (USBD_OK);
    /* USER CODE END 6 */
}
```

Les variables UserRxBufferFS et ucFlagRxCDC devant être partagées entre les fichiers usbd\_cdc\_if.c et main.c, l'utilisateur devra définir ces dernières en variables externes dans le fichier main.c. Cette définition se fait comme suit, dans la fonction main(), avec l'ensemble des autres variables de la fonction main() :

```
extern uint8_t UserRxBufferFS[];
extern uint8_t ucFlagRxCDC;
```

Et bien entendu, on devra inclure le fichier .h, associé au fichier usbd\_cdc\_if.c, dans le main() :

```
#include "usbd_cdc_if.h"
```

Après compilation du programme et branchement du port micro-USB au PC, ce dernier devrait faire la détection du « Device VCP » et ajouter le pilote nécessaire à son fonctionnement.

Pour tester la communication, on doit vérifier si un caractère est reçu en comparant le « Flag » de réception et ensuite transmettre le caractère reçu sur la communication VCP. Le programme ressemblerait à ceci :

```
if (ucFlagRxCDC == 0xFF)
{
    CDC_Transmit_FS(UserRxBufferFS,1);
    UserRxBufferFS[0] = 0x00;
    HAL_Delay(10);
    ucFlagRxCDC = 0x00;
}
```

#### 4.3.11 Clef USB en disque (USB HOST)

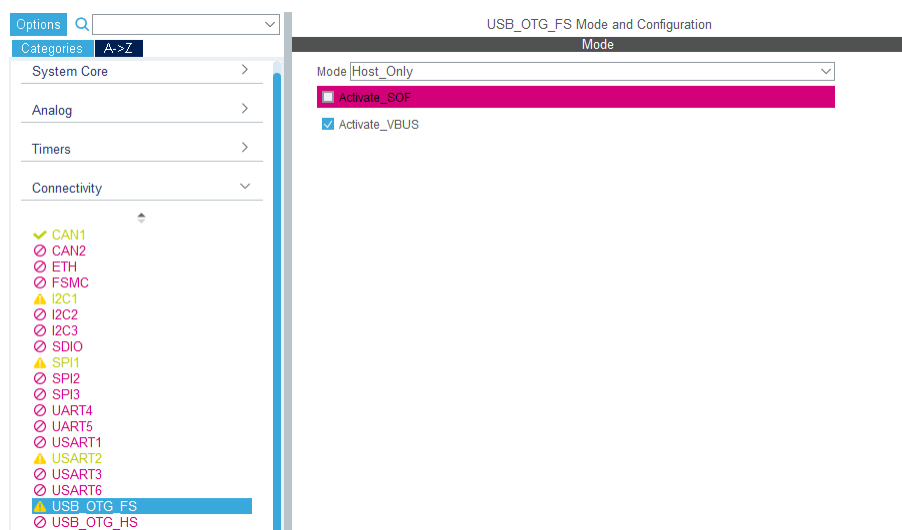
##### *Préambule.*

Le périphérique USB peut être employé sous plusieurs modes. Nous avons vu qu'il peut être employé en dispositif de communication, à l'exemple précédent, mais il peut aussi être employé en dispositif de mémoire de masse. Dans ce cas, le port USB du système ARM sera considéré comme hôte : HOST.

##### *Configuration dans le STM32CubeMX.*

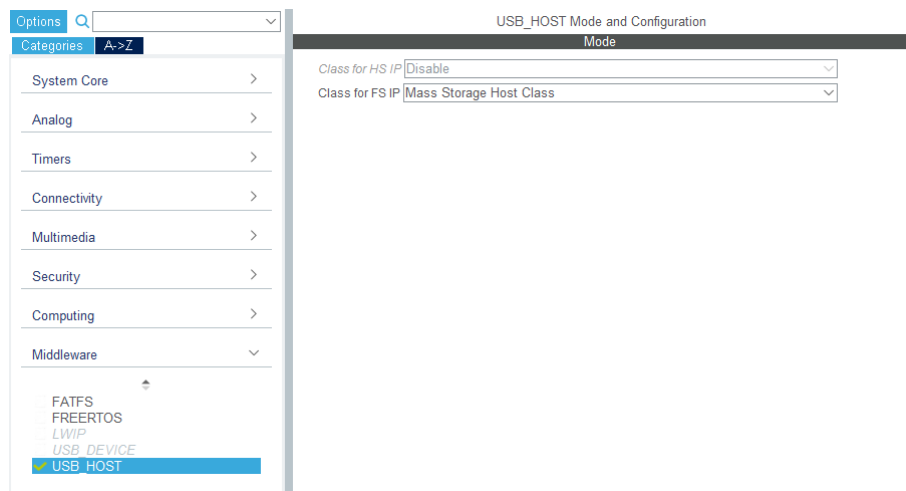
L'approche la plus simple est de démarrer le projet STM32CubeMX avec le fichier SupportSTM32F407.ioc que l'on renomme d'un nom correspondant à l'application désirée. Il est plus sécuritaire de démarrer le projet avec ce fichier .IOC car les broches impliquées dans le traitement des fonctions IO\_BUS y sont bien définies.

L'utilisateur doit désactiver le mode « USB\_DEVICE » pour activer le mode « USB\_HOST » en sélectionnant le périphérique dans la catégorie « Connectivity »...

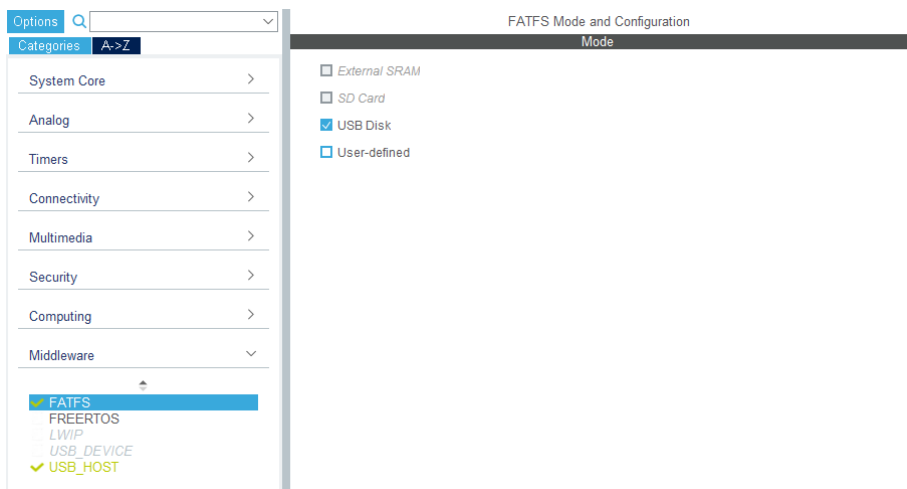


et ensuite activer le « VBUS » en cochant la case prévue à cet effet dans la fenêtre « Mode ».

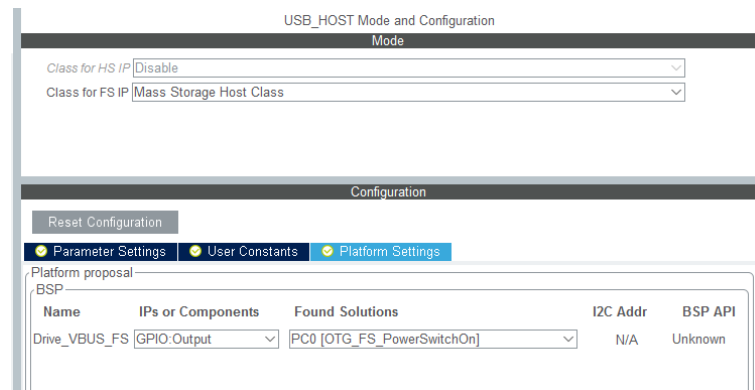
La configuration se poursuit en indiquant que le périphérique « Middleware » USB\_HOST sera employé dans le mode « Mass Storage Host Class ».



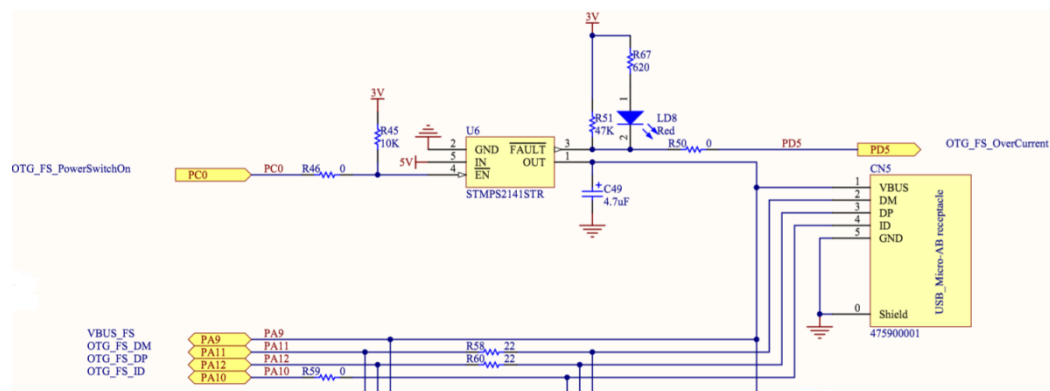
Comme les fonctions de gestion de fichiers « FATFS » devront être employées, l'utilisateur devra activer cet élément de la catégorie « Middleware » en prenant soin d'indiquer que ce sera pour un dispositif « USB Disk ».



En dernier lieu, la configuration du CubeMX sera complétée par l'activation d'un GPIO nécessaire à l'alimentation du USB et ainsi au bon fonctionnement du mode « Mass Storage Host Class ». En appuyant sur « Platform Setting » de la fenêtre de configuration du « USB\_HOST », l'utilisateur aura accès à une nouvelle fenêtre lui permettant de spécifier le GPIO qui activera le VBUS du USB.



Cette configuration est nécessaire pour alimenter le VBUS du dispositif USB tel que présenté sur les plans du système de développement DISCOVERY STM32F4.



Il est maintenant temps de générer le code et de faire un premier test de compilation. Si tout se passe bien, l'utilisateur doit apporter les ajouts suivant au code :

Dans le fichier main.c

Ajouter l'inclusion du fichier « fatfs.h » à la section /\* USER CODE BEGIN Includes \*/

```
#include "fatfs.h"
```

Ajouter les variables globales nécessaires à la section /\* USER CODE BEGIN PV \*/

```
extern ApplicationTypeDef Appli_state; //État de l'application
FIL stMonFichierA; //Structure du fichier A
FATFS stClefUSB; //Structure FATFS de la clef
```

Ajouter une fonction d'erreur du Handler USB à la section `/* USER CODE BEGIN 4 */`

```
void vErreurHandlerUSB(void)
{
    /* Code pour la gestion d'une erreur de traitement FatFs */
    /* On peut implanter ce que l'on veut pour gérer une erreur */
    /* Ici, on allume le LED rouge pour signifier cette erreur */
    HAL_GPIO_WritePin(LD5_GPIO_Port, LD5_Pin, GPIO_PIN_SET);
    while(1);
    /* Fin du code de gestion d'erreur */
}
```

Et sa prédéclaration à la section `/* USER CODE BEGIN PFP */`

```
void vErreurHandlerUSB(void);
```

On doit monter la clef USB comme suit à la section `/* USER CODE BEGIN 2 */`

```
if(f_mount(&stClefUSB, (TCHAR const*)cCheminClefUSB, 0) != FR_OK)
{
    // Erreur d'initialisation
    vErreurHandlerUSB();
}
```

Et finalement, poursuivre avec ce code toujours à la section `/* USER CODE BEGIN 2 */`

```
while (Appli_state!=APPLICATION_READY)
    MX_USB_HOST_Process();
// Création et ouverture d'un nouveau fichier avec permission d'écriture
if(f_open(&stMonFichierA, cNomFichierA, FA_CREATE_ALWAYS | FA_WRITE) != FR_OK)
{
    vErreurHandlerUSB();    // Erreur d'ouverture de fichier 'ESSAI10.TXT'
}
// Écriture de donnée dans un fichier texte
res = f_write(&stMonFichierA, ucTexteEcrit, sizeof(ucTexteEcrit), (void *)&ulBytesEcrits);
if((ulBytesEcrits == 0) || (res != FR_OK))
{
    vErreurHandlerUSB();    // Erreur d'écriture dans le fichier 'STM32.TXT'
}
// Fermeture du fichier
f_close(&stMonFichierA);
// Ouverture du fichier avec permission de lecture seule
if(f_open(&stMonFichierA, cNomFichierA, FA_READ) != FR_OK)
{
    vErreurHandlerUSB();    // Erreur de lecture du fichier STM32.TXT'
}
// Lecture des données d'un fichier texte
res = f_read(&stMonFichierA, ucTexteLu, sizeof(ucTexteLu), (void *)&ulBytesLus);
if((ulBytesLus == 0) || (res != FR_OK))
{
    vErreurHandlerUSB();    // Erreur de lecture de fichier 'STM32.TXT'
}
// Fermeture du fichier texte
f_close(&stMonFichierA);
// Comparaison du nombre de données lues et écrites
if((ulBytesLus != ulBytesEcrits))
{
    vErreurHandlerUSB();    // Données lues != Données écrites
}
// Succès de l'opération
// Ici, on allume le LED vert pour signifier la réussite
HAL_GPIO_WritePin(LD4_GPIO_Port, LD4_Pin, GPIO_PIN_SET);
FATFS_UnLinkDriver(cCheminClefUSB);    // Dissocier la clef USB
```



Le résultat de l'exécution de ce code permettra de créer un fichier sur la clef USB, fichier portant le nom ESSAI10.TXT et dont le contenu est

#### 4.3.12 Les fonctions I/O et la carte multi-I/O de 5e session

##### *Preamble.*

L'utilisation de différents périphériques parallèle n'est pas supportée par le MCU. Il existe une fonctionnalité « Flexible Static Memory Controller » (FSMC) mais cette dernière ne permet pas la gestion de plus d'un périphérique.

L'emploi d'un périphérique parallèle nécessite le contrôle de lignes de données, d'adresses et de contrôle. Sur la carte support STM32F4, on retrouve huit lignes de données (D0 à D7) et les lignes d'adresse sont limitées à cinq (A0 à A4) afin de ne pas trop hypothéquer les ressources du MCU. Ces lignes ne seront donc pas suffisantes pour qu'un décodage d'adresse soit implanté et les « Chip Select » seront contrôlés manuellement. On retrouve quatre CS (CS0 à CS3), le CS0 étant réservé à l'écran et le CS1 à la carte I/O de la cinquième session.

##### *Configuration dans le STM32CubeMX.*

L'approche la plus simple est de démarrer le projet STM32CubeMX avec le fichier SupportSTM32F407.ioc car ce dernier est testé et fonctionnel pour l'ensemble des périphériques intégrés à la carte de développement, dont l'écran. Cela limitera les possibilités d'erreur de noms et de direction des broches impliquées dans le traitement des fonctions IO\_BUS.

Lorsque le projet est généré par le STM32CubeMX, on doit y ajouter les fichiers nécessaires au fonctionnement des périphériques parallèles. Le fichier IO\_BUS.o doit être copié dans le répertoire « obj » et le fichier IO\_BUS.h dans le répertoire « inc » de votre dossier de travail. Vous devez aussi ajouter les fichiers .obj à votre projet IAR grâce à la commande « ADD FILE ». Deux fonctions seront accessibles à l'utilisateur : les fonctions de lecture et d'écriture d'un élément parallèle externe.

Lecture d'un dispositif attaché au CS2, dont le CS est actif à niveau bas, à l'adresse 0x02.

```
ucData = ucReadIO(2, 0, 0x02);
```

Écriture de la valeur 0x55 sur un dispositif attaché au CS2, dont le CS est actif à niveau bas, à l'adresse 0x04.

```
vWriteIO(0x55, 2, 0, 0x04);
```

Exemple de programmes pour le traitement des éléments de la carte I/O :

- Lecture des entrées digitales.

```
ucDataDigi = ucReadIO(CS1, Level_CS1, ADR_D_IN);
```

- Lecture de l'entrée analogique 0.

```
ucReadIO(CS1, Level_CS1, ADR_A_IN_0);
```

```
//Demande de conversion.  
HAL_GPIO_WritePin(GPIOB, CONVERT_Pin, GPIO_PIN_RESET);  
HAL_GPIO_WritePin(GPIOB, CONVERT_Pin, GPIO_PIN_SET);  
ucDataAnalog = ucReadIO(CS1, Level_CS1, ADR_A_IN_0);  
ucEssai = ucReadIO(CS1, Level_CS1, ADR_A_IN_ALL);
```

- Écriture des sorties digitales.

```
vWriteIO(aRxBuffer[0], CS1, Level_CS1, ADR_D_OUT);
```

- Écriture de la sortie analogique 0.

```
ucReadIO(CS1, Level_CS1, ADR_A_IN_0);
```

```
//Demande de conversion.  
HAL_GPIO_WritePin(GPIOB, CONVERT_Pin, GPIO_PIN_RESET);  
HAL_GPIO_WritePin(GPIOB, CONVERT_Pin, GPIO_PIN_SET);  
ucDataAnalog = ucReadIO(CS1, Level_CS1, ADR_A_IN_0);  
ucDataAnalog = ucReadIO(CS1, Level_CS1, ADR_A_IN_ALL);
```

### 4.3.13 L'écran graphique KS0108

#### *Préambule.*

L'utilisation de l'écran KS0108 implique que la configuration des broches D0 à D7, A0 à A4, CS0, ainsi que RD soit faite dans le logiciel STM32CubeMX afin de permettre l'utilisation des fonctions IO\_BUS.

#### *Configuration dans le STM32CubeMX.*

L'emploi de l'écran nécessite l'utilisation des fonctions GLcd et les fonctions GLcd sont dépendantes des fonctions IO\_BUS. On doit donc copier les fichiers IO\_BUS.o et GLcd.o dans le répertoire « obj » et les fichiers IO\_BUS.h et GLcd.h dans le répertoire « inc » de votre dossier de travail. On ajoute aussi le fichier ST.h si on désire afficher le logo de la compagnie ST au démarrage du programme.

On doit spécifier l'inclusion des fichiers précédemment mentionnés avec les lignes de code suivantes à intégrer au fichier main.c :

```
/* USER CODE BEGIN Includes */
#include "IO_BUS.h"
#include "GLcd.h"
#include "ST.h"
/* USER CODE END Includes */
```

L'initialisation de l'écran et l'affichage du logo de ST se fait à travers ces lignes de code. Il est important de désactiver les CS pour ne pas causer de conflit sur le bus des I/O.

```
/* USER CODE BEGIN 2 */

HAL_GPIO_WritePin(GPIOB, CS0_Pin, GPIO_PIN_RESET);
HAL_GPIO_WritePin(GPIOB, CS1_Pin, GPIO_PIN_SET);
HAL_GPIO_WritePin(GPIOB, CS2_Pin, GPIO_PIN_SET);
HAL_GPIO_WritePin(GPIOB, CS3_Pin, GPIO_PIN_SET);

HAL_Delay(100);
vInitGLcd();
vPutBMPLcd(ucST);
HAL_Delay(1000);
vClearGLcd(0x00);

/* USER CODE END 2 */
```

Lorsque ces ajouts sont faits et que l'écran semble fonctionnel, les fonctions de l'écran sont accessibles et prêtes à l'emploi. La liste complète des fonctions est donnée à la section suivante du présent document.

#### 4.4 INTRODUCTION AU SYSTÈME DE DÉVELOPPEMENT

---

La carte « Support STM32 » est prévue pour accueillir une plaquette de développement « Discovery STM32F407 ». Elle offre une multitude de possibilités d'interconnexion dont :

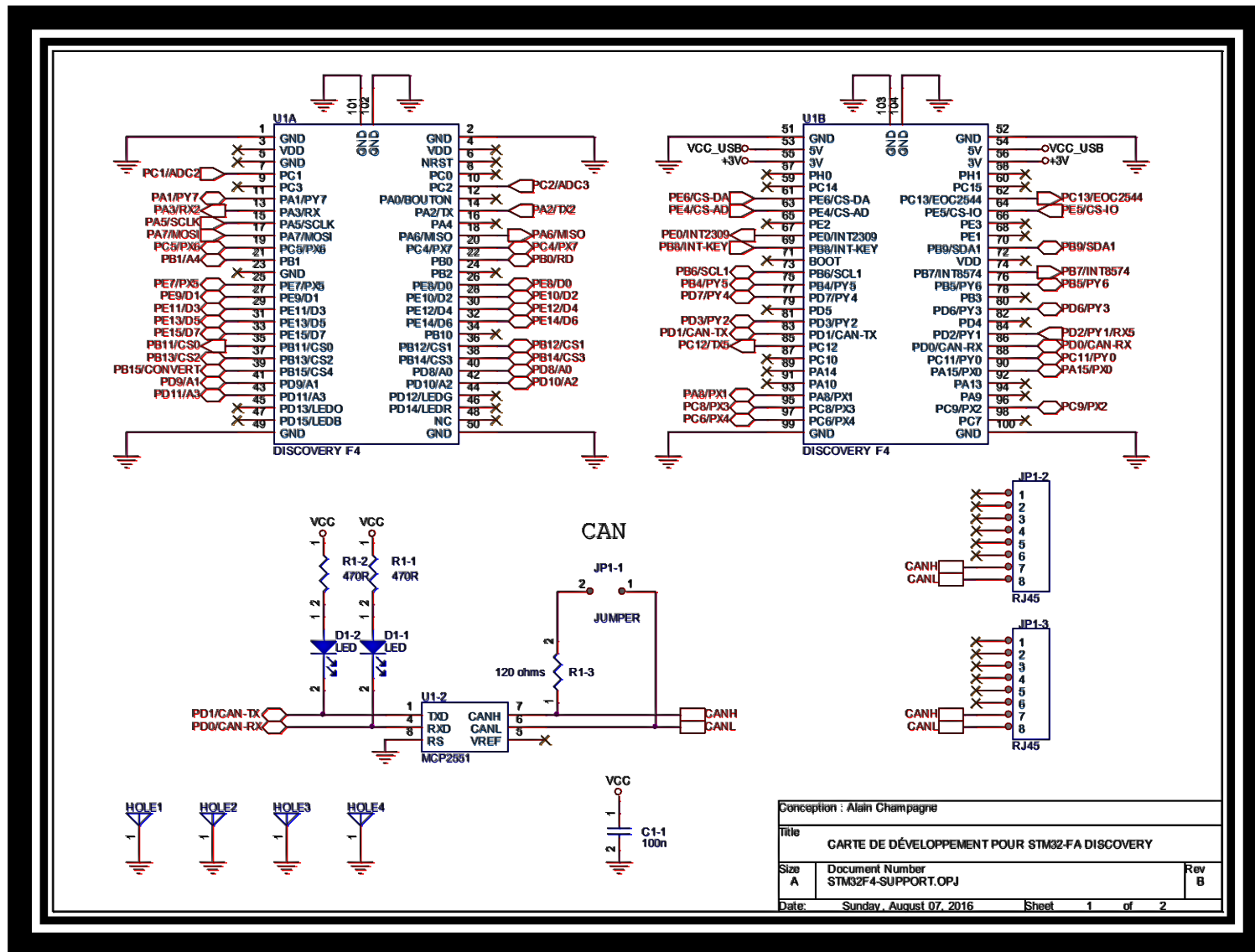
- Un écran graphique de 128 X 64 modèle KS0108,
- Deux entrées analogiques,
- Deux connecteurs I2C comprenant les lignes SDA, SCL, VCC et GND,
- Un connecteur pour clavier I2C,
- Deux connecteurs CAN-Bus,
- Une possibilité d'alimentation externe pour des développements nécessitant plus de courant,
- Deux ports d'extension de 8 bits,
- Les signaux pour le contrôle de bus parallèle 8 bits dont cinq lignes d'adresse et quatre CS,
- Un connecteur I2C pour l'utilisation de la carte I2C multi I/O,
- Un connecteur SPI pour l'utilisation de la carte SPI multi I/O,
- Un lien de communication Bluetooth en mode RS-232,

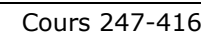
De plus, la plaquette de développement « Discovery STM32F407 » intègre une connexion USB pour la reprogrammation et le déverminage d'applications en plus d'offrir la connexion USB OTG attaché au MCU principale de la carte. Un capteur inclinomètre ainsi qu'un micro et un circuit audio viennent compléter l'ensemble du système.

Joint à ce document, les schémas de la carte, les imprimés des couches de la plaquette de circuit, une explication des possibilités d'utilisation des connecteurs de la carte ainsi qu'une liste de fonctions permettant d'utiliser aisément les périphériques et dispositifs électroniques suivant :

- Écran KS0108,
- Les circuits I2C suivant : Mémoire 24C32, RTC DS1307, I/O PCF8574, D/A DAC6574 et A/D MAX 1236,
- Les circuits SPI suivants : I/O 23S09, D/A DAC7554 et A/D TLV2544,
- La communication CAN-Bus,
- Et le circuit inclinomètre LIS3DSH.

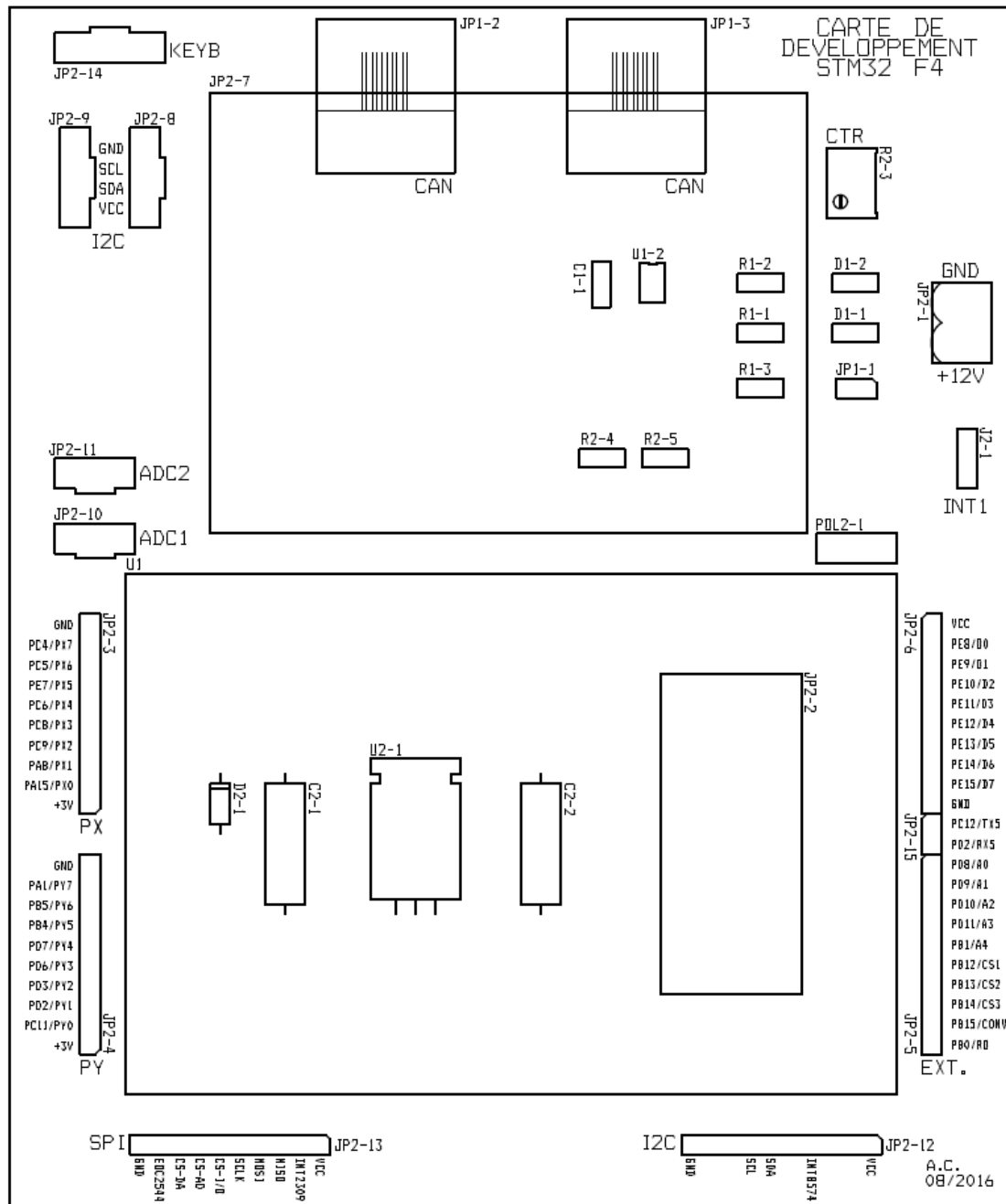
## 4.4.1 Les schémas





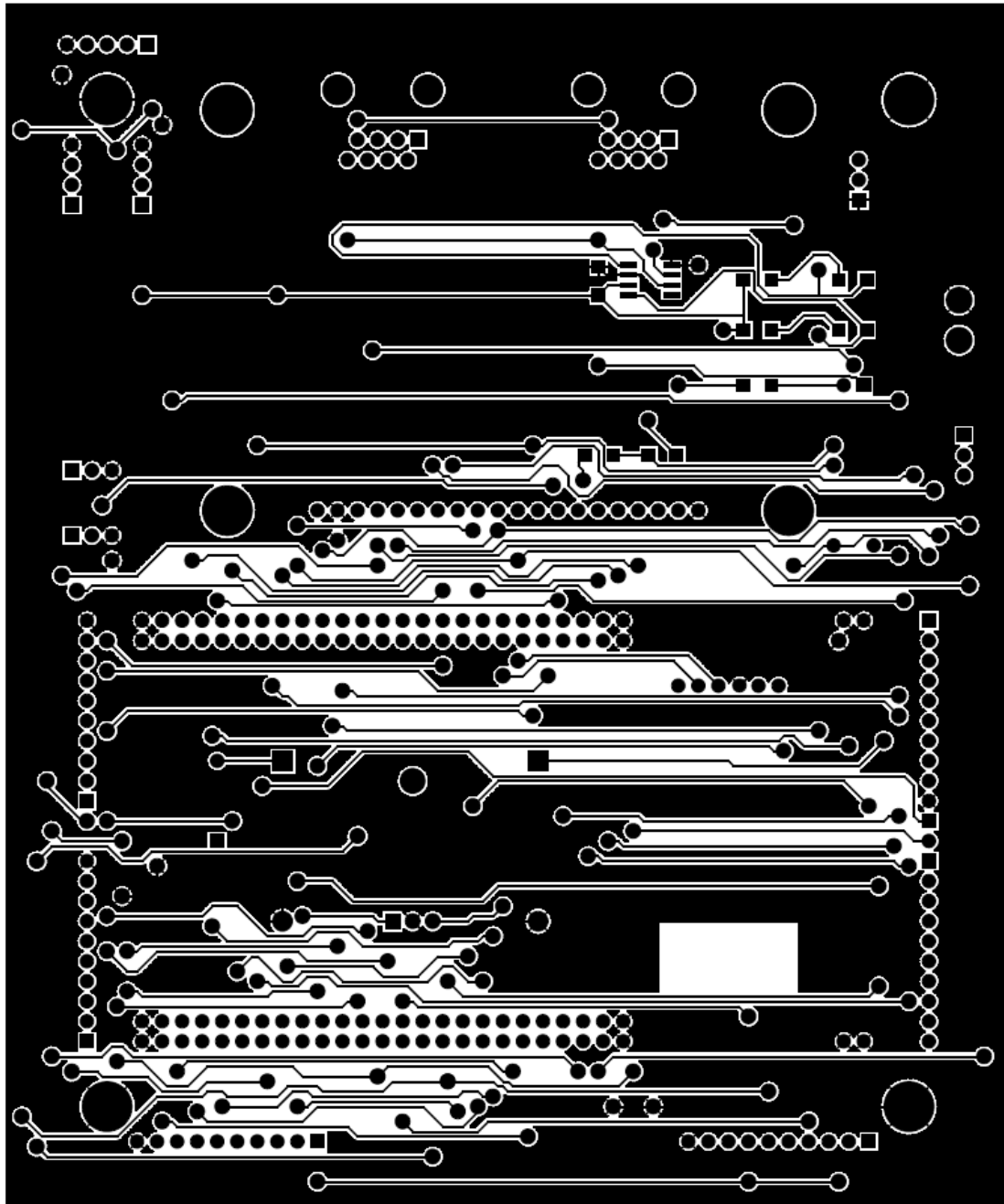
#### 4.4.2 Les couches de la plaquette de circuit

Masque de soie. (Silk screen layer)

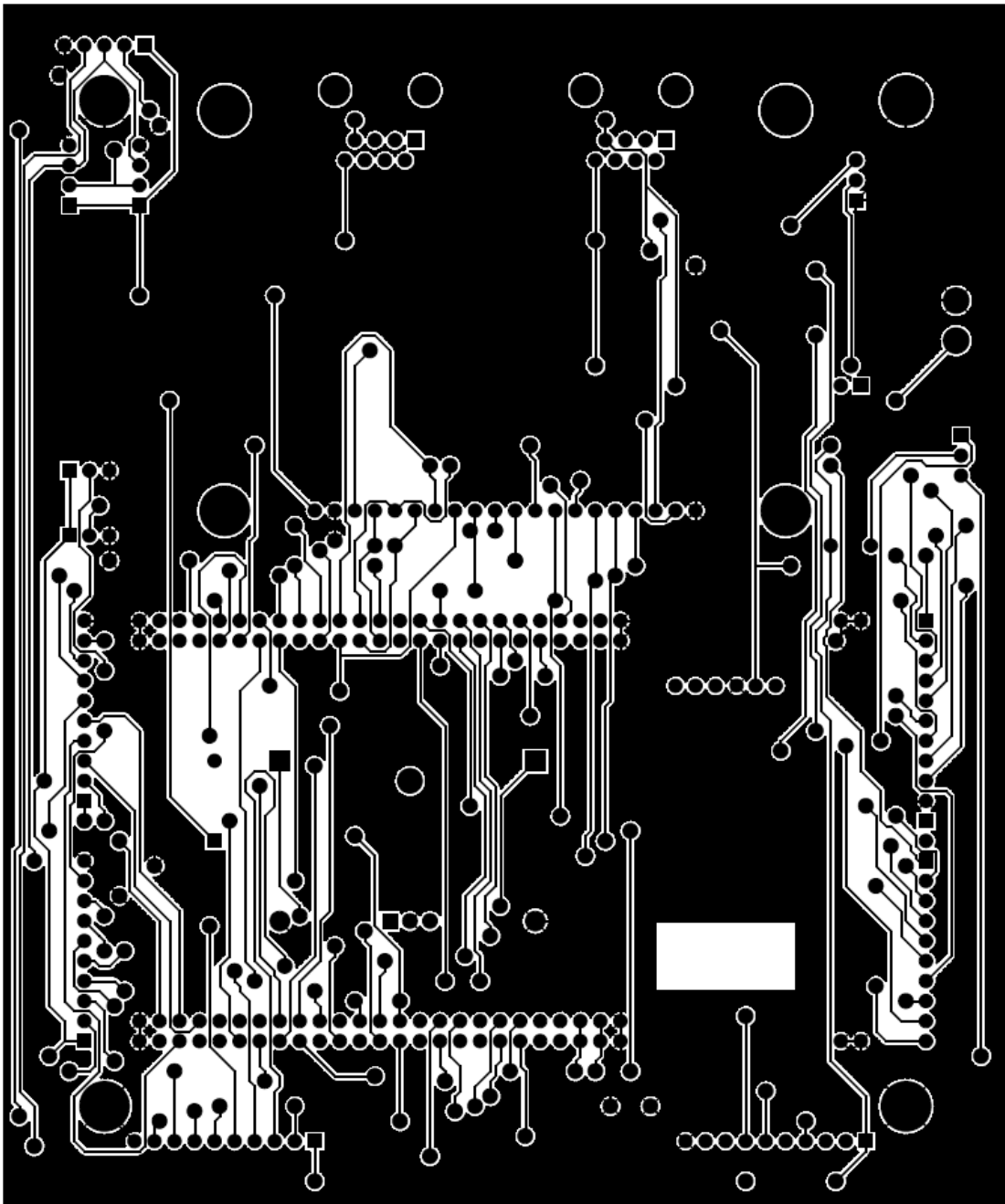




Couche du dessus. (Top layer)

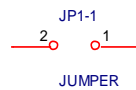


Couche du dessous. (Bottom layer)



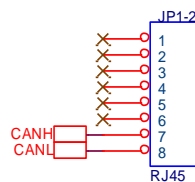
### 4.4.3 Les connecteurs

JP1-1 : Connecteur de résistance de terminaison pour le CAN BUS.



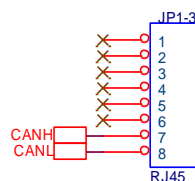
- Le cavalier doit être en position pour l'utilisation du CAN BUS.

JP1-2 : Connecteur 1 de liaison pour le CAN BUS.



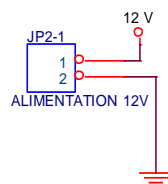
- Le connecteur 1 ou 2 sont employés pour connecter le dispositif sur un bus de type CAN.

JP1-3 : Connecteur 2 de liaison pour le CAN BUS.



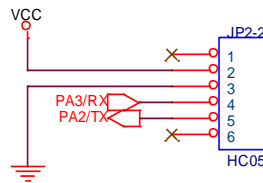
- Le second connecteur de liaison CAN BUS est employé pour faire boucler les connexions sur le bus de type CAN.

JP2-1 : Connecteur d'alimentation externe 12 Volts.



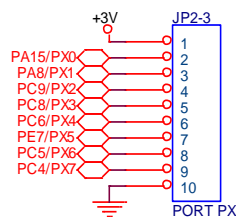
- L'alimentation externe est requise lorsque l'alimentation USB est insuffisante. La carte DISCOVERY doit malgré tout être alimentée par le lien USB.

JP2-2 : Connecteur pour le module Bluetooth HC05.



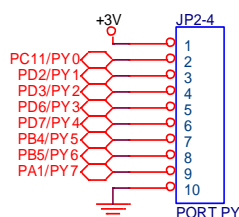
- La communication RS-232 passe par un module Bluetooth HC05. Ce dernier doit être présent pour communiquer en mode RS-232.

JP2-3 : Connecteur du port X avec alimentation 3,3 Volts.



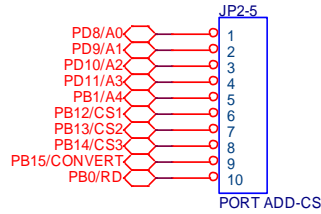
- Un connecteur de 8 bits de port nommé « X » pouvant être employé pour des tests ou des applications. L'alimentation +3,3 Volts et masse sont disponibles sur le connecteur.

JP2-4 : Connecteur du port Y avec alimentation 3,3 Volts.



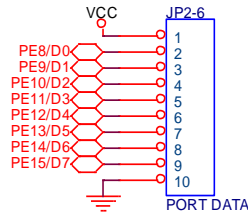
- Un connecteur de 8 bits de port nommé « Y » pouvant être employé pour des tests ou des applications. L'alimentation +3,3 Volts et masse sont disponibles sur le connecteur.

JP2-5 : Connecteur adresses et CS pour le BUS I/O.



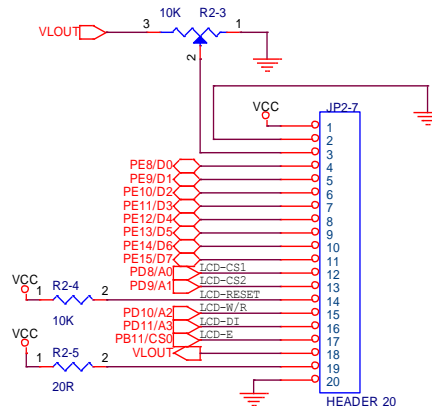
- Ce connecteur est employé pour relier un ou des dispositifs nécessitant un contrôle par bus parallèle. Il est habituellement employé avec le connecteur de données JP2-6.

JP2-6 : Connecteur données pour le BUS I/O.



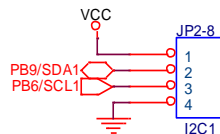
- Ce connecteur est employé pour relier un ou des dispositifs nécessitant un contrôle par bus parallèle. Il est habituellement employé avec le connecteur d'adresses et CS JP2-5.

JP2-7 : Connecteur d'écran graphique KS0108.



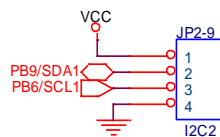
- Le connecteur est employé pour attacher un écran modèle KS0108. Cet écran est de type bus parallèle et il est attaché à l'ensemble des lignes des bus de données, d'adresses et de contrôle.

JP2-8 : Connecteur 1 I2C simple.



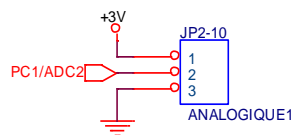
- Le connecteur permet d'attacher un module de communication I2C 4 fils. Les modules 24C32 et DS1307 peuvent s'y relier.

JP2-9 : Connecteur 2 I2C simple.



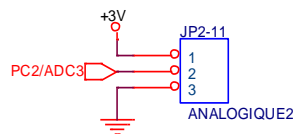
- Le connecteur permet d'attacher un module de communication I2C 4 fils. Les modules 24C32 et DS1307 peuvent s'y relier.

JP2-10 : Connecteur ADC 1.



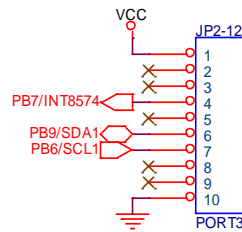
- Le connecteur accepte un dispositif capteur alimenté à 3,3 Volts ou tout simplement un potentiomètre.

JP2-11 : Connecteur ADC 2.



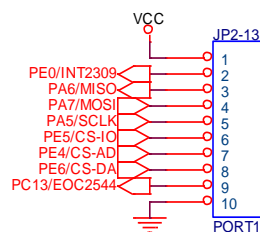
- Le connecteur accepte un dispositif capteur alimenté à 3,3 Volts ou tout simplement un potentiomètre.

JP2-12 : Connecteur carte I2C multifonctions.



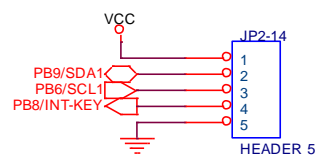
- Le connecteur est prévu pour accueillir la carte multi IO I2C.

JP2-13 : Connecteur carte SPI multifonctions.



- Le connecteur est prévu pour accueillir la carte multi IO SPI.

JP2-14 : Connecteur clavier I2C.



- Le connecteur est prévu pour accueillir un clavier I2C.

#### 4.4.4 Les fonctions

IO-BUS.

ucReadIO() : Fonction de lecture des IO BUS.

```

//*****ucReadIO*****
//  Nom de la fonction : ucReadIO
//  Auteur : Alain Champagne
//  Date de création : 21-12-2015
//  Description : Routine pour lire une donnée d'un device avec BUS
//                parallèle.
//
//  Appel : ucData = ucReadIO(2, 0, 0x04);
//          Permet de lire une donnée sur un dispositif dont le CS est le #2,
//          actif bas, et l'adresse est la 4 (00000100).
//
//  Fonctions appelées : HAL_GPIO_WritePin(), HAL_GPIO_ReadPin
//  Paramètres d'entrée : CSx, Level, ucAdresse
//                      CSx : 0, 1, 2, 3, 4
//                      Level : 0, 1
//                      ucAdresse : 0x00 à 0x1F
//  Paramètres de sortie : uint8_t
//  Variables utilisées : CSx, Level, ucAdresse
//  Equate : Aucun
//  #Define : Aucun
//
//*****

```

vWriteIO() : Fonction d'écriture des IO BUS.

```

//*****vWriteIO*****
//  Nom de la fonction : vWriteIO
//  Auteur : Alain Champagne
//  Date de création : 21-12-2015
//  Description : Routine pour placer les lignes de données en sortie.
//
//  Appel : vWriteIO(0x55, 2, 0, 0x04);
//          Permet d'écrire la donnée 0x55 sur un dispositif dont le CS est
//          le #2, actif bas, et l'adresse est la 4 (00000100).
//
//  Fonctions appelées : HAL_GPIO_WritePin(), HAL_GPIO_ReadPin
//  Paramètres d'entrée : CSx, Level, ucAdresse
//                      CSx : 0, 1, 2, 3, 4
//                      Level : 0, 1
//                      ucAdresse : 0x00 à 0x1F
//  Paramètres de sortie : uint8_t
//  Variables utilisées : CSx, Level, ucAdresse
//  Equate : Aucun
//  #Define : Aucun
//
//*****

```



PORT\_XY.

ucReadPort () : Fonction de lecture des PORTs X et Y.

```

//*****ucReadPort*****
//  Nom de la fonction : ucReadPort
//  Auteur : Alain Champagne
//  Date de création : 04-11-2016
//  Description : Routine pour lire le PORT X ou Y.
//
//  Appel : ucData = ucReadPort('X');
//           Permet de lire les bits de données du PORT X.
//
//  Fonctions appelées : HAL_GPIO_ReadPin().
//  Paramètres d'entrée : ucPort
//  Paramètres de sortie : uint8_t
//  Variables utilisées : ucData, unData, ucLireDataIO
//  Equate : Aucun
//  #Define : Aucun
//
//*****

```

vWritePort () : Fonction d'écriture des PORTs X et Y.

```

//*****vWritePort*****
//  Nom de la fonction : vWritePort
//  Auteur : Alain Champagne
//  Date de création : 04-11-2016
//  Description : Routine pour écrire sur le port X ou Y.
//
//  Appel : vWritePort(0x55, 'X');
//           Permet d'écrire la valeur 0x55 sur le port X.
//
//  Fonctions appelées : HAL_GPIO_WritePin().
//  Paramètres d'entrée : ucEcrireDataPort, ucPort.
//  Paramètres de sortie : uint8_t
//  Variables utilisées : CSx, Level, ucAdresse
//  Equate : Aucun
//  #Define : Aucun
//
//*****

```

GLcd.

**vInitGLcd() : Fonction d'initialisation de l'écran KS0108.**

```

//*****vInitGLcd*****
//  Nom de la fonction : vInitGLcd
//  Auteur : Alain Champagne
//  Date de création : 15-03-2006
//  Date de modification pour le STM32F4 : 23-12-2015
//  Description : Initialisation de l'écran graphique KS0108.
//
//  Appel : vInitGLcd();
//           Initialise l'écran selon des paramètres établis.
//
//  Fonctions appelées : vOutputGLcd.
//  Paramètres d'entrée : Aucun.
//  Paramètres de sortie : Aucun.
//  Variables utilisées : Aucun.
//  Equate : Aucun
//  #Define : Aucun
//
//*****

```

**vClearGLcd() : Fonction d'effacement de l'écran KS0108.**

```

//*****vClearGLcd*****
//  Nom de la fonction : vClearGLcd
//  Auteur : Alain Champagne
//  Date de création : 15-03-2006
//  Date de modification pour le STM32F4 : 23-12-2015
//  Description : Effacer l'écran graphique KS0108.
//
//  Appel : vClearGLcd(0x00);
//           Place tous les pixel à 0, donc efface l'écran.
//
//  Fonctions appelées : vOutputGLcd()
//  Paramètres d'entrée : ucData
//  Paramètres de sortie : Aucun
//  Variables utilisées : i et j
//  Equate : Aucun
//  #Define : ADRINSTRRIGHTW, ADRINSTRLEFTW, ADRDATALEFTW, ADRDATARIGHTW
//
//*****

```

**vPutPixelGLcd() : Fonction d'écriture d'un pixel à l'écran KS0108.**

```
//*****vPutPixelGLcd*****
//  Nom de la fonction : vPutPixelGLcd
//  Auteur : Alain Champagne
//  Date de création : 15-03-2006
//  Date de modification pour le STM32F4 : 23-12-2015
//  Description : Placer un pixel en x et y sur l'écran graphique KS0108.
//                X = 0 a 127 et Y = 0 a 63. Pour placer le pixel, on doit
//                lire la position 8 bits, à l'écran, et masquer le bit à
//                afficher. La lecture doit être faite deux fois (directive
//                du fabricant) et on doit replacer le curseur après
//                chaque lecture.
//
//  Appel : vPutPixelGLcd(10,10);
//          Active un pixel à la position 10, 10.
//
//  Fonctions appelées : vOutputGLcd(),ucInputGLcd()
//  Paramètres d'entrée : ucX et ucY
//  Paramètres de sortie : Aucun
//  Variables utilisées : ucAdresse, ucData, ucDataY, ucDataOld
//  Equate : Aucun
//  #Define : ADRINSTRRIGHTW, ADRINSTRLEFTW
//
//*****
```

**vClearPixelGLcd() : Fonction d'effacement d'un pixel à l'écran KS0108.**

```
//*****vClearPixelGLcd*****
//  Nom de la fonction : vClearPixelGLcd
//  Auteur : Alain Champagne
//  Date de création : 15-03-2006
//  Date de modification pour le STM32F4 : 23-12-2015
//  Description : Effacer un pixel en x et y sur l'écran graphique KS0108.
//                X = 0 a 127 et Y = 0 a 63. Pour effacer le pixel, on doit
//                lire la position 8 bits, à l'écran, et masquer le bit à
//                afficher. La lecture doit être faite deux fois (directive
//                du fabricant) et on doit replacer le curseur après
//                chaque lecture.
//
//  Appel : vClearPixelGLcd(10,10);
//          Désactive un pixel à la position 10, 10.
//
//  Fonctions appelées : vOutputGLcd(), ucInputGLcd().
//  Paramètres d'entrée : ucX et ucY.
//  Paramètres de sortie : Aucun.
//  Variables utilisées : ucAdresse, ucData, ucDataY, ucDataOld.
//  Equate : Aucun
//  #Define : ADRINSTRRIGHTW, ADRINSTRLEFTW
//
//*****
```

**vDrawLineGLcd() : Fonction de traçage d'une ligne à l'écran KS0108.**

```
//*****vDrawLineGLcd*****
//  Nom de la fonction : vDrawLineGLcd
//  Auteur : Alain Champagne
//  Date de création : 20-03-2006
//  Date de modification pour le STM32F4 : 23-12-2015
//  Description : Dessiner une ligne de la coordonnée (X1Y1) avec une
//                direction (H ou V) et une longueur.
//
//  Appel : vDrawLineGLcd(0, 0, 20, 'H');
//          Trace une ligne horizontale de 20 pixel dont l'origine est 0, 0.
//
//  Fonctions appelées : vPutPixelGLcd()
//  Paramètres d'entrée : ucX, ucY, ucLong, ucDir
//  Paramètres de sortie : Aucun
//  Variables utilisées : i
//  Equate : Aucun
//  #Define : ADRINSTRIGHTW, ADRINSTRLEFTW
//
//*****
```

**vDrawRectangleGLcd() : Fonction de traçage d'une rectangle à l'écran KS0108.**

```
//*****vDrawRectangleGLcd*****
//  Nom de la fonction : vDrawRectangleGLcd
//  Auteur : Alain Champagne
//  Date de création : 18-03-2006
//  Date de modification pour le STM32F4 : 23-12-2015
//  Description : Dessiner un rectangle aux coordonnées (X1Y1, X2Y2).
//
//  Appel : vDrawRectangleGLcd(1,1,20,20);
//          Traçage d'un rectangle dont la diagonale est positionnée
//          aux coordonnées X1=1, Y1=1 et X2=20, Y2=20.
//
//  Fonctions appelées : vPutPixelGLcd()
//  Paramètres d'entrée : ucX1, ucY1, ucX2 et ucY2
//  Paramètres de sortie : Aucun
//  Variables utilisées : ucLong et ucHaut, i
//  Equate : Aucun
//  #Define : Aucun
//
//*****
```

**vDrawCircleGLcd() : Fonction de traçage d'un cercle à l'écran KS0108.**

```
//*****vDrawCircleGLcd*****
//  Nom de la fonction : vDrawCircleGLcd
//  Auteur : Alain Champagne
//  Date de création : 18-03-2006
//  Date de modification pour le STM32F4 : 23-12-2015
//  Description : Dessiner un cercle de rayon ucRayon à la coordonnée ucX et
//               ucY.
//
//  Appel : vDrawCircleGLcd(20, 50, 10);
//           Traçage d'un cercle de 10 de rayon dont le centre est situé
//           aux coordonnées X=20, Y=50.
//
//  Fonctions appelées : vPutPixelGLcd()
//  Paramètres d'entrée : ucX, ucY et ucRayon
//  Paramètres de sortie : Aucun
//  Variables utilisées : iSwitch, iX, iY, ucD
//  Equate : Aucun
//  #Define : Aucun
//
//*****
```

**vPutCharGLcd() : Fonction d'écriture d'un caractère à l'écran KS0108.**

```
//*****vPutCharGLcd*****
//  Nom de la fonction : vPutCharGLcd
//  Auteur : Alain Champagne
//  Date de création : 17-03-2006
//  Date de modification pour le STM32F4 : 23-12-2015
//  Description : Écriture d'un caractère a une position Ligne Colonne sur
//               l'écran GLCD. Il est possible d'utiliser les colonnes 0 à
//               15 et les lignes 0 à 7 pour le FONT 7, les colonnes 0 à 19
//               et les lignes 0 à 7 pour le FONT 5 et les colonnes 0 à 31
//               et les lignes 0 à 7 pour le FONT 3.
//
//  Appel : vPutCharGLcd('A', 1, 0, 3);
//           Écrit le caractère A de grosseur de font 3 à la ligne 1
//           et la colonne 0.
//
//  Fonctions appelées : vOutputGLcd()
//  Paramètres d'entrée : ucChar, ucLigne, ucColonne et ucFont
//  Paramètres de sortie : Aucun
//  Variables utilisées : uiAdresse, i, iChar, ucX et ucData
//  Equate : Aucun
//  #Define : Aucun
//
//*****
```

**vPutStringGLcd() : Fonction d'écriture d'une chaîne de caractères à l'écran KS0108.**

```

//*****vPutStringGLcd*****
//  Nom de la fonction : vPutStringGLcd
//  Auteur : Alain Champagne
//  Date de création : 17-03-2006
//  Date de modification pour le STM32F4 : 23-12-2015
//  Description : Écriture d'une ligne de caractères a une position Ligne sur
//                l'écran GLCD. Il est possible d'utiliser les lignes 0 à 7 et
//                les FONT 3 ou 5. 32 caractères de font 3, 20 caractères de
//                font 5 et 16 caractères de font 7 par lignes.
//
//  Appel : vPutStringGLcd(ucTab, 0, 3);
//          Écrit une chaîne de caractères de font 3 provenant d'un
//          tableau ucTab à la ligne 0.
//
//  Fonctions appelées : vPutCharGLcd()
//  Paramètres d'entrée : ucChar, ucLigne et ucFont
//  Paramètres de sortie : Aucun
//  Variables utilisées : uiAdresse, i
//  Equate : Aucun
//  #Define : Aucun
//
//*****

```

**vPutBMPGLcd() : Fonction d'affichage d'un BMP à l'écran KS0108.**

```

//*****vPutBMPGLcd*****
//  Nom de la fonction : vPutBMPGLcd
//  Auteur : Alain Champagne
//  Date de création : 15-03-2006
//  Date de modification pour le STM32F4 : 23-12-2015
//  Description : Affiche un BMP l'écran graphique KS0108.
//
//  Appel : vPutBMPGLcd(ucTab[]);
//          Affiche un fichier BMP de 128 X 64 à l'écran.
//
//  Fonctions appelées : vOutputGLcd()
//  Paramètres d'entrée : Aucun
//  Paramètres de sortie : Aucun
//  Variables utilisées : i et j
//  Equate : Aucun
//  #Define : ADRDATALEFTW, ADRDATARIGHTW
//
//*****

```

## 24C32

v2432WriteByte() : Fonction d'écriture d'un octet en mémoire I2C.

```

//*****v2432WriteByte*****
//  Nom de la fonction : v2432WriteByte
//  Auteur : Alain Champagne
//  Date de création : 03-01-2016
//  Description : Routine de transmission d'un octet de donnée
//                provenant de la mémoire I2C.
//
//  Appel : v2432WriteByte(0x55, 0x0140);
//           Placer la valeur 0x55 à l'adresse 0x0140.
//
//  Fonctions appelées :      HAL_I2C_IsDeviceReady(), HAL_I2C_Mem_Write()
//
//  Paramètres d'entrée :    uint8_t ucData, uint16_t uiAdresse
//  Paramètres de sortie :   Aucun
//  Variables utilisées :    ucTableau[]
//  Equate :                 Aucun.
//  #Define :                 ECRIREI2C24C32, LIREI2C24C32
//
//*****

```

uc2432ReadByte() : Fonction de lecture d'un octet en mémoire I2C.

```

//*****uc2432ReadByte*****
//  Nom de la fonction : uc2432ReadByte
//  Auteur : Alain Champagne
//  Date de création : 03-01-2016
//  Description : Routine de réception d'un octet de donnée
//                provenant de la mémoire I2C. On doit passer en paramètre
//                la variable iAdresse.
//
//  Appel : ucData = uc2432ReadByte(0x0140);
//           Lire la valeur en mémoire I2C à l'adresse 0x0140.
//
//  Fonctions appelées : HAL_I2C_IsDeviceReady(), HAL_I2C_Mem_Read().
//
//  Paramètres d'entrée :    uint16_t uiAdresse
//  Paramètres de sortie :   ucData
//  Variables utilisées :    ucTableau[1], ucData
//  Equate :                 Aucun
//  #Define :                 ECRIREI2C24C32, LIREI2C24C32
//
//*****

```

**uc2432Write() : Fonction d'écriture d'un bloc d'octet en mémoire I2C.**

```

//*****v2432Write*****
//  Nom de la fonction : v2432Write
//  Auteur : Alain Champagne
//  Date de création : 03-01-2016
//  Description : Routine de transmission d'une page de données char
//                  vers le circuit de mémoire I2C. Les paramètres attendus
//                  sont le tableau d'entrée, l'adresse en mémoire où les
//                  données doivent être inscrites et la grosseur du tableau
//                  qui ne doit pas dépasser 32 octets. On doit démarrer
//                  l'écriture sur une adresse dont les 5 dernier bits sont
//                  à 0. Ex: 0x0000, 0x0020, 0x0040, 0x0200...
//
//  Appel : v2432Write(ucTableau, 0x0100, 0x20);
//          Écrire 32 octets à l'adresse 0x0100 en mémoire I2C. Les données
//          proviennent d'un tableau de uint8_t.
//
//  Fonctions appelées : HAL_I2C_IsDeviceReady, HAL_I2C_Mem_Write
//  Paramètres d'entrée : char ucTableau[], uint16_t uiAdresse,
//                        uint8_t ucSize
//  Paramètres de sortie : Aucun
//  Variables utilisées : Aucune
//  Equate : Aucun
//  #Define : ECRIREI2C24C32, LIREI2C24C32
//
//*****

```

**v2432Read() : Fonction d'écriture d'un bloc d'octet en mémoire I2C.**

```

//*****v2432Read*****
//  Nom de la fonction : v2432Read
//  Auteur : Alain Champagne
//  Date de création : 03-01-2016
//  Description : Routine de réception d'une page de données char pro-
//                  venant du circuit de mémoire I2C. Les paramètres attendus
//                  sont le tableau d'entrée, l'adresse en mémoire où les
//                  données doivent être inscrites et la grosseur du tableau
//                  qui ne doit pas dépasser 32 octets. On doit démarrer
//                  la lecture sur une adresse dont les 5 dernier bits sont
//                  à 0. Ex: 0x0000, 0x0020, 0x0040, 0x0200...
//
//  Appel : v2432Read(ucTableau, 0x0100, 0x20);
//          Lecture de 32 octets à l'adresse 0x0100 en mémoire I2C. Les
//          données sont retournées dans un tableau de uint8_t.
//
//  Fonctions appelées : HAL_I2C_IsDeviceReady, HAL_I2C_Mem_Read
//  Paramètres d'entrée : char ucTableau[], uint16_t uiAdresse,
//                        uint8_t ucSize
//  Paramètres de sortie : Retourne les valeurs dans le tableau d'entrée
//  Variables utilisées : Aucune
//  Equate : Aucun
//  #Define : ECRIREI2C24C32, LIREI2C24C32
//
//*****

```



## DS1307

## v1307WriteByte() : Fonction d'écriture d'un octet en mémoire I2C.

```

//*****v1307WriteByte*****
//  Nom de la fonction : v1307WriteByte
//  Auteur : Alain Champagne
//  Date de création : 03-01-2016
//  Description : Initialisation complète du circuit DS1307 avec 8 octets
//                  permettant d'inscrire : (00)les secondes x 1 1 1 1 1 1 1 1
//
//  Appel : v1307WriteByte(0x55, 0x0010);
//            Placer la valeur 0x55 à l'adresse 0x0010.
//
//  Fonctions appelées : HAL_I2C_Master_Transmit()
//  Paramètres d'entrée : uint8_t ucData, uint8_t ucAdresse
//  Paramètres de sortie : Aucun
//  Variables utilisées : ucTableau[]
//  Equate : Aucun
//  #Define : ECRIREI2C1307
//
//*****

```

## uc1307ReadByte() : Fonction de lecture d'un octet en mémoire I2C.

```

//*****uc1307ReadByte*****
//  Nom de la fonction : uc1307ReadByte
//  Auteur : Alain Champagne
//  Date de création : 03-01-2016
//  Description : Routine de réception d'un octet de donnée
//                  provenant du circuit RTC DS1307.
//
//  Appel : ucData = uc1307ReadByte(0x0010);
//            Lire la valeur en mémoire I2C à l'adresse 0x0010.
//
//  Fonctions appelées : HAL_I2C_Master_Transmit(), HAL_I2C_Master_Receive()
//  Paramètres d'entrée : *ptr, uiAdd.
//  Paramètres de sortie : Retourne les valeurs dans le tableau d'entrée
//  Variables utilisées : i
//  Equate : Aucun
//  #Define : Aucun
//
//*****

```

uc1307WritePage() : Fonction d'écriture d'un bloc de 9 octets dans le DS1307.

```

//*****v1307WritePage*****
//  Nom de la fonction : v1307WritePage
//  Auteur : Alain Champagne
//  Date de création : 03-01-2016
//  Description : Initialisation complète du circuit DS1307 avec 9 octets
//                permettant d'inscrire : (00) Premier octet séparateur
//
//                (01)les secondes x 1 1 1 1 1 1 1 1
//                00-05 00-09
//                (02)les minutes x 1 1 1 1 1 1 1 1
//                00-05 00-09
//                (03)les heures x x 1 1 1 1 1 1
//                00-02 00-09
//                (04)la journée x x x x x 1 1 1
//                01-07
//                (05)la date x x 1 1 1 1 1 1
//                00-03 00-09
//                (06)le mois x x x 1 1 1 1 1
//                00-01 00-09
//                (07)l'année 1 1 1 1 1 1 1 1
//                00-09 00-09
//
//                et l'octet de configuration : bit 0 : RS0
//                bit 1 : RS1
//                bit 4 : SQWE
//                bit 7 : OUT
// Appel :
//      char ucInitDS1307[9]={0x00,0x00,0x07,0x72,0x02,0x05,0x01,0x16,0x93};
//      v1307WritePage(ucInitDS1307, 0x0000);
//
//      Écrire les valeurs de 9 octets à l'adresse 0x0000 du DS1307.
//      Cette fonction peut être employée pour initialiser le DS1307.
//
// Fonctions appelées : HAL_I2C_Master_Transmit
// Paramètres d'entrée : char ucTableau[], uint8_t ucAdresse
// Paramètres de sortie : Aucun
// Variables utilisées : ucAddDS1307[]
// Equate : Aucun
// #Define : ECRIREI2C1307
//
//*****

```

**v1307ReadPage() : Fonction de lecture d'un bloc d'octet en mémoire I2C.**

```

//*****v1307ReadPage*****
//  Nom de la fonction : v1307ReadPage
//  Auteur : Alain Champagne
//  Date de création : 03-01-2016
//  Description : Routine de lecture de la configuration du circuit DS1307.
//                La routine doit recevoir un tableau de 8 octets en
//                entrée pour retourner les 8 octets de configuration.
//                Pour plus de manoeuvrabilité, on passe aussi le
//                paramètre d'adresse afin de permettre la lecture d'un
//                groupe de 8 octets ailleurs en mémoire.
//                Pour lire les octets de configuration :
//                v1307ReadPage(ucLireInitDS1307, 0x00) où
//                ucLireInitDS1307 est un tableau de 8 octets.
//                Il est aussi possible de lire des données des adresses
//                0x08 - 0x3F sur le DS1307.
//
//  Appel : v1307ReadPage(ucTableau, 0x0000);
//                Lire les valeurs de 8 octets à l'adresse 0x0000 du DS1307.
//                Cette fonction peut être employée pour initialiser le DS1307.
//
//  Fonctions appelées : HAL_I2C_Master_Transmit, HAL_I2C_Master_Receive
//  Paramètres d'entrée : ucTableau[], ucAdresse
//  Paramètres de sortie : ucData
//  Variables utilisées : ucAddDS1307[1]
//  Equate : Aucun
//  #Define : ECRIREI2C1307, LIREI2C1307
//
//*****

```

**MAX1236****i1236ReadChanel() : Fonction de lecture d'un canal du convertisseur MAX1236.**

```

//*****i1236ReadChanel*****
//  Nom de la fonction : i1236ReadChanel(ucChanel)
//  Auteur : Alain Champagne
//  Date de création : 04 avril 2006
//  Modifiée pour le KIT ARM : 03-01-2016
//  Description : Routine de lecture de la donnée du convertisseur MAX1236.
//                On passe comme paramètre le no du chanel du convertisseur
//                en code ascii (0, 1, 2 ou 3).
//
//  Appel : i1236ReadChanel('0');
//                Lire la valeur de l'entrée 0 du convertisseur MAX1236.
//
//  Fonctions appelées : HAL_I2C_Master_Transmit(), HAL_I2C_Master_Receive()
//  Paramètres d'entrées : ucChanel
//  Paramètres de sortie : unsigned int
//  Variables utilisées : iAnalog, ucTableau
//  Equate : Aucun
//  #Define : CONFIGURE01236, SETUP01236, ECRIREI2C1236, LIREI2C1236
//
//*****

```

## DAC6574

v6574WriteInt() : Fonction d'écriture sur un canal du convertisseur DAC6574.

```

//*****v6574WriteInt*****
//  Nom de la fonction : v6574WriteInt
//  Auteur : Alain Champagne
//  Date de création : 04-01-2016
//  Description : Routine de transmission d'un INT de donnée
//                sur le canal d'un convertisseur DA.
//
//  Appel : v6574WriteInt('0', 0x3000);
//          Écrire la valeur 0x3000 sur le convertisseur du DAC6574.
//
//  Fonctions appelées : HAL_I2C_Master_Transmit()
//  Paramètres d'entrée : uint8_t ucChanel, uint16_t uiData
//  Paramètres de sortie : Aucun
//  Variables utilisées : ucTableau[]
//  Equate : Aucun
//  #Define : CONFIGUREx6574, ECRIREI2C6574
//
//*****

```

## PCF8574

V8574WriteByte() : Fonction d'écriture sur le PCF8574.

```

//*****v8574WriteByte*****
//  Nom de la fonction : v8574WriteByte
//  Auteur : Alain Champagne
//  Date de création : 03-01-2016
//  Description : Routine de transmission d'un octet de donnée
//                vers le circuit IO PCF8574.
//
//  Appel : v8574WriteByte(0x55);
//          Écrire la valeur 0x55 sur le dispositif PCF8574.
//
//  Fonctions appelées : HAL_I2C_Master_Transmit
//  Paramètres d'entrée : uint8_t ucData
//  Paramètres de sortie : Aucun
//  Variables utilisées : ucTableau[]
//  Equate : Aucun
//  #Define : ECRIREI2C8574
//
//*****

```

**V8574ReadByte() : Fonction de lecture sur le PCF8574.**

```

//*****uc8574ReadByte*****
//  Nom de la fonction : uc8574ReadByte
//  Auteur : Alain Champagne
//  Date de création : 03-01-2016
//  Description : Routine de réception d'un octet de donnée
//                provenant du circuit IO PCF8574.
//
//  Appel : ucData = uc8574ReadByte();
//           Lire la valeur sur le dispositif PCF8574.
//
//  Fonctions appelées : HAL_I2C_Master_Receive
//  Paramètres d'entrée : Aucun
//  Paramètres de sortie : uint8_t ucData
//  Variables utilisées : ucTableau, ucData
//  Equate : Aucun
//  #Define : LIREI2C8574
//
//*****

```

**23S09****v23S09Read() : Fonction de lecture sur le 23S09.**

```

//*****v23S09Read*****
//  Nom de la fonction : v23S09Read
//  Auteur : Alain Champagne
//  Date de création : 20-01-2016
//  Description : Routine de lecture d'un registre du circuit MCP23S09.
//                Les paramètres sont un tableau dans lequel sera retourné
//                la valeur lue et l'adresse du registre à lire. La
//                définition de tous les registres du MCP23S09 est faite
//                dans le fichier 23S09.h.
//
//  Appel : v23S09Read(ucBufferIn, SPIGPIOA);
//           Lire la valeur sur le dispositif 23S09.
//
//  Fonctions appelées : HAL_SPI_GetState, HAL_SPI_Transmit,
//                      HAL_SPI_Receive
//  Paramètres d'entrée : char ucBufferIn[], uint8_t ucAddRegister
//  Paramètres de sortie : Un retour se fait par un pointeur sur tableau
//  Variables utilisées : ucBuffer2309Out[]
//  Equate : Aucun
//  #Define : HAL_SPI_STATE_RESET, HAL_SPI_STATE_BUSY_TX
//           HAL_SPI_STATE_BUSY_RX, HAL_SPI_STATE_READY
//           SPI23S09Read, CS_SPI23S09_L, CS_SPI23S09_H.
//
//*****

```

**v23S09Write() : Fonction d'écriture du 23S09.**

```

//*****v23S09Write*****
//  Nom de la fonction : v23S09Write
//  Auteur : Alain Champagne
//  Date de création : 20-01-2016
//  Description : Routine d'écriture d'un registre du circuit MCP23S09.
//                Les paramètres sont l'adresse du registre et la valeur
//                à écrire. La définition de tous les registres du
//                MCP23S09 est faite dans le fichier 23S09.h.
//
//  Appel : v23S09Write(SPIIODIRA, 0xFF);
//           Place les lignes en entrée sur le 23S09.
//
//  Fonctions appelées : HAL_SPI_GetState, HAL_SPI_Transmit
//  Paramètres d'entrée : uint8_t ucValue, uint8_t ucAddRegister
//  Paramètres de sortie : Un retour se fait par un pointeur sur tableau.
//  Variables utilisées : ucBufferOut[]
//  Equate : Aucun
//  #Define : HAL_SPI_STATE_RESET, HAL_SPI_STATE_BUSY_TX
//            HAL_SPI_STATE_READY, SPI23S09ECRIRE,
//            CS_SPI23S09_L, CS_SPI23S09_H
//
//*****

```

**TLV2544****v2544Write() : Fonction d'écriture du convertisseur ADC TLV2544.**

```

//*****v2544Write*****
//  Nom de la fonction : v2544Write
//  Auteur : Alain Champagne
//  Date de création : 22-01-2016
//  Description : Routine d'écriture de la donnée de configuration du
//                convertisseur TLV2544ID. On passe comme paramètre
//                l'entier de 16 bits de configuration.
//  Appel : v2544Write(ECRIRECFRSPi2544);
//           Configuration du TLV2544.
//
//  Fonctions appelées : HAL_SPI_Transmit
//  Paramètres d'entrée : uiConfiguration
//  Paramètres de sortie : Aucun
//  Variables utilisées : ucBuffer2544Out
//  Equate : Aucun
//  #Define : CS_SPI2544_L, CS_SPI2544_H
//
//*****

```

ui2544Read() : Fonction de lecture du convertisseur ADC TLV2544.

```

//*****ui2544Read*****
//  Nom de la fonction : ui2544Read
//  Auteur : Alain Champagne
//  Date de création : 22-01-2016
//  Description : Routine de lecture de la donnée du convertisseur
//                TLV2544ID. On passe comme paramètre le no du chanel du
//                convertisseur grâce à un #Define : CHANEL0, CHANEL1,
//                CHANEL2, CHANEL4.
//
//  Appel : ui2544Canal = ui2544Read(CHANEL0);
//          ui2544Canal = ui2544Read(CHANEL0);
//          On appelle deux fois la fonction pour une conversion correcte.
//
//  Fonctions appelées : HAL_SPI_TransmitReceive
//  Paramètres d'entrée : ucChanel
//  Paramètres de sortie : uint16_t
//  Variables utilisées : ucBuffer2544Out, ucBuffer2544In, uiConversion
//  Equate : Aucun
//  #Define : CS_SPI2544_L, CS_SPI2544_H
//
//*****

```

DAC7554

v7554Write() : Fonction d'écriture du convertisseur DAC DAC7554.

```

//*****v7554Write*****
//  Nom de la fonction : v7554Write
//  Auteur : Alain Champagne
//  Date de création : 21-01-2016
//  Description : Routine d'écriture d'une valeur numérique de 12 bits
//                sur un des canaux du convertisseur DAC. Les paramètres
//                attendus sont le numéro du canal ('0', '1', '2' ou '3')
//                et la valeur de conversion sur 12 bits.
//
//  Appel : v7554Write('0', 0x0400);
//          Demander une conversion de 0x0400 sur le chanel 0.
//
//  Fonctions appelées : HAL_SPI_GetState, HAL_SPI_Transmit
//  Paramètres d'entrée : uint8_t ucValue, uint8_t ucAddRegister
//  Paramètres de sortie : Aucun
//  Variables utilisées : ucBufferOut[]
//  Equate : Aucun
//  #Define : HAL_SPI_STATE_RESET, HAL_SPI_STATE_BUSY_TX
//            HAL_SPI_STATE_READY, SPI7554CHANEL0,
//            SPI7554CHANEL1, SPI7554CHANEL2, SPI7554CHANEL3,
//            CS_SPI7554_L, CS_SPI7554_H
//
//*****

```

## LIS3DSH

ucLIS3DSHLire() : Fonction de lecture d'un registre 8 bits du LIS3DSH.

```
//*****ucLIS3DSHLire*****
//  Nom de la fonction : ucLIS3DSHLire
//  Auteur : Alain Champagne
//  Date de création : 24-01-2016
//  Description : Routine de lecture d'un registre du circuit LIS3DSH.
//                Les paramètres sont un octet représentant le registre à
//                lire et un octet de retour de la valeur lue. La
//                définition de tous les registres du LIS3DSH est faite
//                dans le fichier LIS3DSH.h.
//
//  Appel : ucSPIData = ucLIS3DSHLire(LIS3DSH_Reg_WHO_AM_I);
//          Lire un registre 8 bits du LIS3DSH.
//
//  Fonctions appelées : HAL_SPI_GetState, HAL_SPI_Transmit, HAL_SPI_Receive
//  Paramètres d'entrée : uint8_t LIS3DSH_Reg
//  Paramètres de sortie : Un retour se fait sur un uint8_t
//  Variables utilisées : ucBufferSPI[]
//  Equate : Aucun
//  #Define : HAL_SPI_STATE_RESET, HAL_SPI_STATE_BUSY_TX
//            HAL_SPI_STATE_BUSY_RX, HAL_SPI_STATE_READY,
//            CS_LIS3DH_H_L, CS_LIS3DH_H_H
//
//*****
```

vLIS3DSHEcrire() : Fonction d'écriture sur un registre 8 bits du LIS3DSH.

```
//*****vLIS3DSHEcrire*****
//  Nom de la fonction : vLIS3DSHEcrire
//  Auteur : Alain Champagne
//  Date de création : 24-01-2016
//  Description : Routine d'écriture d'un registre du circuit LIS3DSH.
//                Les paramètres sont un octet représentant le registre à
//                modifier ainsi qu'un octet à écrire. La définition de
//                tous les registres du LIS3DSH est faite
//                dans le fichier LIS3DSH.h.
//
//  Appel : vLIS3DSHEcrire(LIS3DSH_Reg_Ctrl_4, 0x67);
//          Écrire dans un registre 8 bits du LIS3DSH.
//
//  Fonctions appelées : HAL_SPI_GetState, HAL_SPI_Transmit
//  Paramètres d'entrée : char ucBufferIn[], uint8_t ucAddRegister
//  Paramètres de sortie : Un retour se fait par un pointeur sur tableau
//
//  Variables utilisées : ucBufferOut[]
//  Equate : Aucun
//  #Define : HAL_SPI_STATE_RESET, HAL_SPI_STATE_BUSY_TX
//            HAL_SPI_STATE_BUSY_RX, HAL_SPI_STATE_READY,
//            CS_LIS3DH_H_L, CS_LIS3DH_H_H
//
//*****
```



**fLIS3DSHLireSortieX()** : Fonction de lecture de l'axe X du LIS3DSH.

```
//*****fLIS3DSHLireSortieX*****
//  Nom de la fonction : fLIS3DSHLireSortieX
//  Auteur : Alain Champagne
//  Date de création : 24-01-2016
//  Description : Routine de lecture de la coordonnée X du circuit
//                LIS3DSH. Le seul paramètre d'entrée représente la
//                constante "g" que l'on fixe à 16g pour une valeur
//                résultante le plus grande.
//                On fait une inversion des registres X et Y parce que
//                le device est soudé dans un angle différent.
//
//  Appel : fXData = fLIS3DSHLireSortieX(LIS3DSH_Sense_16g);
//          Lire un axe sur le LIS3DSH.
//
//  Fonctions appelées : ucLIS3DSHLire
//  Paramètres d'entrée : uint8_t LIS3DSH_Reg
//  Paramètres de sortie : Un retour se fait sur un float
//  Variables utilisées : Temp
//  Equate : Aucun
//  #Define : LIS3DSH_Reg_X_Out_H, LIS3DSH_Reg_X_Out_L,
//            LIS3DSH_Sense
//
//*****
```

**fLIS3DSHLireSortieY()** : Fonction de lecture de l'axe Y du LIS3DSH.

```
//*****fLIS3DSHLireSortieY*****
//  Nom de la fonction : fLIS3DSHLireSortieY
//  Auteur : Alain Champagne
//  Date de création : 24-01-2016
//  Description : Routine de lecture de la coordonnée Y du circuit
//                LIS3DSH. Le seul paramètre d'entrée représente la
//                constante "g" que l'on fixe à 16g pour une valeur
//                résultante le plus grande.
//                On fait une inversion des registres X et Y parce que
//                le device est soudé dans un angle différent.
//
//  Appel : fYData = fLIS3DSHLireSortieY(LIS3DSH_Sense_16g);
//          Lire un axe sur le LIS3DSH.
//
//  Fonctions appelées : ucLIS3DSHLire
//  Paramètres d'entrée : uint8_t LIS3DSH_Reg
//  Paramètres de sortie : Un retour se fait sur un float
//  Variables utilisées : Temp
//  Equate : Aucun
//  #Define : LIS3DSH_Reg_Y_Out_H, LIS3DSH_Reg_Y_Out_L,
//            LIS3DSH_Sense
//
//*****
```

fLIS3DSHLireSortieZ() : Fonction de lecture de l'axe Z du LIS3DSH.

```
//*****fLIS3DSHLireSortieZ*****
//  Nom de la fonction : fLIS3DSHLireSortieZ
//  Auteur : Alain Champagne
//  Date de création : 24-01-2016
//  Description : Routine de lecture de la coordonnée Z du circuit
//                LIS3DSH. Le seul paramètre d'entrée représente la
//                constante "g" que l'on fixe à 16g pour une valeur
//                résultante le plus grande.
//
//  Appel : fZData = fLIS3DSHLireSortieZ(LIS3DSH_Sense_16g);
//          Lire un axe sur le LIS3DSH.
//
//  Fonctions appelées : ucLIS3DSHLire
//  Paramètres d'entrée : uint8_t LIS3DSH_Reg
//  Paramètres de sortie : Un retour se fait sur un float
//  Variables utilisées : Temp
//  Equate : Aucun
//  #Define : LIS3DSH_Reg_Z_Out_H, LIS3DSH_Reg_Z_Out_L,
//            LIS3DSH_Sense
//
//*****
```

## CAN Bus

vInit\_CAN\_Bus () : Fonction d'initialisation du CAN Bus.

```
//*****vInit_CAN_Bus*****
//  Nom de la fonction : vInit_CAN_Bus
//  Auteur : Alain Champagne
//  Date de création : 13-06-2016
//  Description : Initialisation du CAN_Bus.
//
//  Appel : vInit_CAN_Bus();
//          Initialiser le CAN Bus.
//
//  Fonctions appelées : HAL_CAN_Init, HAL_CAN_ConfigFilter
//  Paramètres d'entrée : Aucun
//  Paramètres de sortie : Aucun
//  Variables utilisées : Aucune
//  Equate : Aucun
//  #Define : Aucun
//
//*****
```

## SOURCES

- Carmine Noviello  
<http://www.carminenoviello.com/mastering-stm32/>  
**Mastering STM32**  
© 2015 - 2016
- ST Microelectronic  
<http://www.st.com/>  
**User Manual UM1725**  
**User Manual UM1730**  
**Data brief**  
**Reference Manual**