

République Tunisienne

\*\*\*\*\*

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

\*\*\*\*\*

Université de Monastir

Institut Supérieur d'Informatique et de Mathématiques de Monastir

\*\*\*\*\*

Département informatique



N° d'ordre :

# Mémoire de Projet De Fin d'Etudes

Présenté en vue de l'obtention du

**Diplôme National de Licence en Technologie  
d'Informations et de Communication**

Spécialité :

**Internet of Things (IoT)**

Par :

**MOHAMED AZIZ MNASSER  
MAJDI RHIM**

---

**Etude, conception et réalisation d'un pousse seringue  
électrique et connectée**

---

Soutenu le xxxxxx devant le jury composé de :

M./Mme : .....

M./Mme : .....

M./Mme : .....

M./Mme : .....

Président

Rapporteur

Encadrant Pédagogique

Encadrant Professionnel

# Résumé

---

---

Ce travail s'inscrit dans le cadre du projet de fin d'étude en vue de l'obtention de la licence en Technologie de l'information et des communications.

Le projet consiste à développer une pousse seringue électrique et connecté selon les normes et les exigences internationales en utilisant un système d'exploitation temps réel et un microcontrôleur STM32H7 basé sur ARM Cortex M7.

Un pousse seringue électrique et connecté permet au médecin de perfuser en continu, à débit constant un médicament et lui permet de superviser l'état d'injection à distance.

---

---

Mots clés : ARM Cortex M7, RTOS, GUI, IHM, ESP8266, Cloud.

# Abstract

---

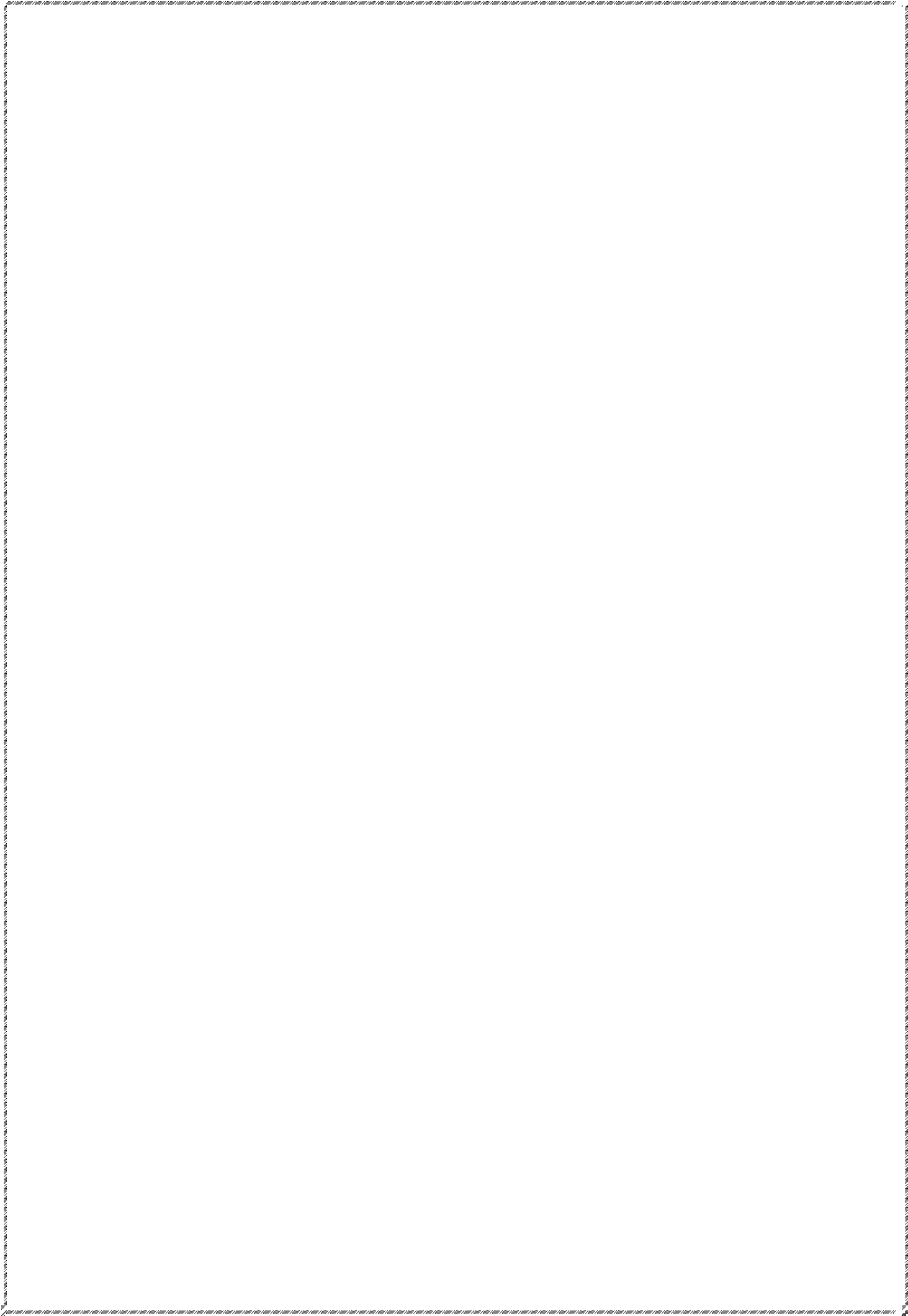
This work is part of the end of study project for the degree in Information and Communication Technology.

The project consists in developing an electric and connected syringe pump according to international standards and requirements using a real time operating system and a STM32H7 microcontroller based on ARM Cortex M7.

A connected electric syringe pump allows the doctor to continuously infuse a drug at a constante rate and allows him to supervise the injection status remotely.

---

Keywords : ARM Cortex M7, RTOS, GUI, HMI, ESP8266, Cloud.



# Table des matières

<b>Introduction Générale :</b> .....	11
<b>1. Chapitre : Contexte général du projet</b> .....	13
<b>1.1 Introduction</b> .....	13
<b>1.2 Présentation de la société d'accueil : MEDIWAVE</b> .....	13
<b>1.3 Différents services</b> .....	13
<b>1.4 Pousse seringue électrique</b> .....	14
<b>1.4.1 Définition</b> .....	14
<b>1.4.2 Principe de fonctionnement</b> .....	15
<b>1.4.3 Différents types des Pousse seringues électriques</b> .....	15
<b>1.4.4 Différents modes d'administration des Pousse Seringues</b> .....	16
<b>1.4.5 Avantage et critique de l'existant</b> .....	17
<b>1.5 Normes et exigences</b> .....	18
<b>1.5.1 Exigence générale</b> .....	18
<b>1.5.2 Processus de développement logiciel</b> .....	19
<b>1.6 Méthodologie de travail</b> .....	19
<b>1.6.1 Problématique</b> .....	19
<b>1.6.2 Solution</b> .....	19
<b>1.6.3 Implémentation</b> .....	20
<b>1.7 Conclusion</b> .....	21
<b>2. Chapitre : Analyse et spécification des besoins</b> .....	22
<b>2.1 Introduction</b> .....	22
<b>2.2 Spécification des besoins</b> .....	22
<b>2.2.1 Expression du besoin</b> .....	22
<b>2.2.2 Cahier des charges</b> .....	23
<b>2.3 Architecture globale du PSC</b> .....	24
<b>2.3.1 Schéma synoptique du PSC</b> .....	24
<b>2.3.2 Détails du matériel utilisé :</b> .....	25
<b>2.3.3 Architecture Software du PSC</b> .....	33
<b>2.3.4 Outils et logiciels utilisés :</b> .....	34
<b>2.3 Conclusion</b> .....	35
<b>3. Chapitre : Conception et réalisation</b> .....	36
<b>3.1 Introduction</b> .....	36
<b>3.2 OS kernel</b> .....	36

3.2.1	Système d'exploitation temps réel .....	36
3.2.2	Tâches Et Queues (File d'attente) .....	37
3.3	Moteur pas à pas.....	42
3.3.1	L6474 Driver / L6474.C : .....	42
3.3.2	Interruptions.....	46
3.3.3	Flux de données : .....	48
3.3.4	Calcule .....	48
3.4	Interface homme machine « IHM » .....	51
3.5	Capteurs et mesures .....	58
3.5.1	Capteur de diamètre.....	58
3.5.2	Capteur de position .....	60
3.5.3	Capteur de température.....	61
3.6	Connectivité .....	62
3.6.1	Solution proposée : .....	62
3.6.2	Architecture et protocoles : .....	62
3.6.3	Implémentation : .....	62
3.7	Taches en cours de développement .....	69
3.8	Conclusion .....	69

# Liste des figures

Figure 1.1 : Modèle de pousse seringue .....	14
Figure 1.2: Partie mécanique de pousse seringue .....	15
Figure 1.3:Pousse seringue à double voie .....	16
Figure 1.4:Structure du projet sur Git Hub.....	20
Figure 1.5 : Cycle de travail avec l'outils Git.....	21
Figure 2.1 : Diagramme de cas d'utilisation de la pousse seringue .....	23
Figure 2.2: Diagramme des composants de la pousse seringue .....	24
Figure 2.3:Schéma synoptique du PSC.....	25
Figure 2.4 : Carte de développement OpenH743I-C .....	25
Figure 2.5: Composants du L'STM32H743IIT6.....	26
Figure 2.6: Moteur PAS à PAS (NEMA17).....	27
Figure 2.7 : Stepper motor driver shield L6474.....	28
Figure 2.8 : LCD TFT 4.3 .....	29
Figure 2.9 : Node MCU ESP8266 .....	29
Figure 2.10 : Capteur de position .....	30
Figure 2.11 : Capteur de diamètre (PTL01-15W0-103B1) .....	31
Figure 3.1 : Structure du code .....	36
Figure 3.2: états des tâches.....	38
Figure 3.3: Priorités et tailles de chaque tache .....	39
Figure 3.4 : Création des taches .....	40
Figure 3.5: File d'attente et mode FIFO.....	41
Figure 3.6 : Création d'une file d'attente (queue) .....	41
Figure 3.7 : Définition des fonctions de lecture et écriture dans une queue .....	42
Figure 3.8 : Schéma de câblage du driver L6474.....	42
Figure 3.9 : Structure du registre STEP_MODE .....	43
Figure 3.10 : Adresse des différents registres .....	44
Figure 3.11 : Structure C L6474_Init_t .....	45
Figure 3.12:Fichier de paramètre par défauts .....	45
Figure 3.13 : Les fonctions pour commander le moteur .....	46
Figure 3.14 : Structure du registre STATUS .....	47
Figure 3.15 : Fonctions d'appel lors une interruption.....	47
Figure 3.16:Flux de données .....	48
Figure 3.17 : Fonction pour calculer la vitesse de déplacement nécessaire .....	49
Figure 3.18 : Formule pour calculer la vitesse de l'arbre récepteur .....	50
Figure 3.19:Formule pour calculer la vitesse de l'arbre moteur .....	50
Figure 3.20 : Fonction pour commander le moteur selon les paramètres calculer .....	51
Figure 3.21 : LTDC+DMA2D .....	52
Figure 3.22 : Structure de données passées dans LTDC.....	52
Figure 3.23 : Communication dalle tactile résistive et processeur .....	53
Figure 3.24 : Modèle de conception modèle-vue-présentateur .....	54
Figure 3.25 : Modèle-Vue-Présentateur et communication externe.....	55
Figure 3.26 : Interface principale .....	56
Figure 3.27 : Interface clavier .....	56
Figure 3.28: Structure de clavier .....	57

Figure 3.29 : Transmission des données de l'IHM vers le backend .....	57
Figure 3.30 : Structure C Infusion_paramT .....	57
Figure 3.31 : Structure tache IHMHandle.....	58
Figure 3.32: Diamètres interne et diamètres externe des seringues.....	59
Figure 3.33:Course de potentiomètre à glissière .....	59
Figure 3.34:Capteur de position.....	60
Figure 3.35 : Configuration de l'analogue Watchdog 1.....	61
Figure 3.36:Fonction d'appel lors d'une interruption wdg analogique .....	62
Figure 3.37 : Adresses UID de l'stm32h743l.....	64
Figure 3.38 : Structure de la tache ConnectivityHandle.....	64
Figure 3.39 : Structure de la tache ConnectivityHandle.....	65
Figure 3.40 : Structure du code Arduino pour Esp8266.....	66
Figure 3.41 : Interface SDMMC .....	67
Figure 3.42:FATFS.....	68
Figure 3.43 : TACHE STORAGE HANDLE.....	68



# Liste des tableaux

Tableau 1.1: Etapes de développement logiciel et niveau de sécurité.....	19
Tableau 2.1 : Les différents composants de STM32H7 .....	27
Tableau 3.1: Les différents composants de OpenH7 .....	73

# Liste des abréviations

## **A :**

AIVT : Anesthésie Intra Veineuse Totale

AIVOC : Anesthésie Intra Veineuse à Objectif de Concentration

## **P :**

PSE : pousse seringue électrique

PSC : pousse seringue connectée

## **I :**

IHM : interface homme machine

## **R :**

RTOS : real time operating system

## **S :**

SPI : Serial Peripheral Interface

## **M :**

MVP : model-view-presenter

MVC : model-view-controler

# Introduction Générale

La technologie de l'information et de la communication est maintenant présente dans tous les secteurs, y compris l'industrie, l'agriculture et la santé. Le centre d'intérêt principal est la santé et les dispositifs médicaux, ce marché progresse avec un chiffre d'affaires de 30 milliards d'euros en 2019 (en France).

On peut distinguer 3 couches fondamentales dans le domaine de TIC : Couche matérielle (Hardware) couche logicielle (Software) et couche réseau (Network). La couche matérielle représente généralement les actionneurs, les capteurs et les unités de traitement tels que les microprocesseurs et les microcontrôleurs ... La couche logicielle est un élément-clé dans toute application, elle sert à acquérir les données et les traiter pour commander les actionneurs. La couche réseau est un mélange entre le hardware et le software qui sert à accéder à un réseau tel que l'Internet pour échanger les données avec l'extérieur selon divers protocoles.

Un système d'exploitation est un programme qui agit comme un intermédiaire entre l'utilisateur et la machine. Son but est de fournir un environnement dans lequel un utilisateur peut exécuter des programmes et son rôle est de coordonner l'exécution simultanée de plusieurs tâches utilisateurs.

Un Système d'Exploitation Temps Réel ou RTOS est utilisé quand il y a des exigences temporelles fixes sur les opérations d'un processeur ou sur le flux de données. Il possède des contraintes de temps fixes et bien définies : le traitement doit être effectué dans la contrainte de temps sinon le système échoue.

En fait, le travail effectué dans ce projet de fin d'étude met en œuvre le domaine de TIC en association avec le domaine de santé pour construire un dispositif médical selon les couches indiquées précédemment. C'est dans ce cadre, que s'inscrit notre stage effectué au sein de startup « 3Dwave » qui s'attache au but d'étudier et développer un pousse seringue électrique et connecté.

Afin de bien introduire et expliquer le sujet, nous avons décrit dans ce rapport tout le travail effectué pendant le stage. Dans le premier chapitre, nous allons présenter le contexte du projet, y compris une brève introduction à l'organisation d'accueil et la méthodologie du travail. Le deuxième chapitre portera sur l'analyse et la description des besoins ainsi que les problèmes soulevés et les solutions suggérées en présentons l'architecture globale du système.

Le troisième chapitre sera consacré à la conception, le développement et la réalisation pratique de la solution. Finalement, notre rapport s'achèvera par une conclusion générale regroupant les principaux résultats trouvés et par quelques perspectives.

# Chapitre 1 : Contexte général du projet

## 1.1 Introduction

Le premier chapitre de ce rapport vise à mettre le projet dans son cadre général. Nous commençons tout d'abord par présenter l'entreprise d'accueil. Puis, nous décrivons le système et ses différends. Ensuite, nous analysons les solutions existant afin d'identifier leurs imperfections. Enfin, nous indiquons les normes et les exigences générales et la méthode de travail.

## 1.2 Présentation de la société d'accueil : MEDIWAVE

L'électronique excite aujourd'hui dans divers domaines et secteurs. Pour cela, la présence des systèmes embarqués est devenue un critère de base pour assurer le bon développement d'une société ou d'une industrie. À ce propos le secteur industriel envisage une grande croissance grâce aux progrès technologiques notamment dans le domaine robotique et celui de l'automatisation des machines. Cet aspect encourage plusieurs sociétés à s'investir dans la recherche et le développement des solutions afin d'améliorer le processus de production. **MEDIWAVE** est un excellent exemple des industries qui ont profité des bienfaits de la technologie. Elle a été mise en exploitation depuis MAI 2020. Elle est située à Sousse sous la direction générale de **M. Farid KAMEL**. Son domaine d'activité est principalement l'étude, la conception et la réalisation des machines industrielles spéciales.

## 1.3 Différents services

« MEDIWAVE » regroupe plusieurs services :

- **Etude, conception et mise en œuvre de solutions médicales**

C'est une conception des produits en trois dimensions avec des logiciels bien spécifiques CAP. « MEDIWAVE » élabore des dessins techniques, qui sont une étape indispensable située entre la conception et la réalisation du produit final. La création haut de gamme est facilitée par l'utilisation d'outils spécialisés dans la mécanique.

- **Conception des appareils médicaux spéciaux**

Spéciales et tout autre outil de production en série répondant aux besoins des clients.

- **Usinage mécanique**

MEDWAVE se caractérise par la fabrication de divers outils, notamment les petits moules d'injection, les outils de soudage, les outils de montage et les outils de commande.

- **Electricité et électronique médicale**

C'est la conception des armoires électrique. Une étude complète de l'architecteur des différents réseaux (dimensionnement câbles, protection) se fait afin d'assurer le meilleur rapport qualité-prix. Le domaine de fabrication de la société est large et englobe : la gestion de production et de commande, la gestion des alarmes et surveillance des défauts, la supervision et les réseaux de courant forts ou faibles. La conception électronique consiste à la réalisation d'ensembles électronique consiste à la réalisation d'ensembles électroniques médicale et des cartes électroniques médicale d'après un cahier des charges.

- **Maintenance médicale**

Le domaine de la maintenance est vaste et peut être effectué en électronique, automatisme, mécanique, hydraulique, pneumatique et contrat de maintenance clés en main.

## 1.4 Pousse seringue électrique

### 1.4.1 Définition

Un pousse-seringue électrique (PSE) ou seringue auto pulsée (SAP) est un dispositif médical de classe B (voir 1.5.1) utilisé pour administrer de faibles quantités de fluide (avec ou sans médicament) à un patient à travers une seringue allant de 1ml jusqu'à un volume de 100ml. On les retrouve majoritairement dans les services de soins des Centres Hospitaliers. Facile à utiliser, leur programmation rapide permet aux personnels soignants de lancer une perfusion en quelques secondes, de manière complètement sécurisée et ainsi permettre la bonne observance médicamenteuse pour les patients. [1]



Figure 1.1 : Modèle de pousse seringue

## 1.4.2 Principe de fonctionnement

La figure 1.2 présente les différentes parties vues de l'extérieur d'un PSE, le système combine des parties électriques et mécaniques. La partie mécanique comprend un berceau et un piston qui vont recevoir le corps de la seringue. Le berceau est généralement muni d'un capteur et d'une encoche pour verrouiller la seringue. La collerette du piston se fixe sur le chariot du piston de seringue au moyen de griffes. Il comprend également un système de capteurs qui vont permettre de vérifier la bonne position. Le piston du PSE se déplace grâce à un système de vis sans fin qui va littéralement pousser le contenu de la seringue vers le corps du patient. Cette partie mécanique est commandée par un moteur électrique alimenté soit par le secteur soit par batterie.

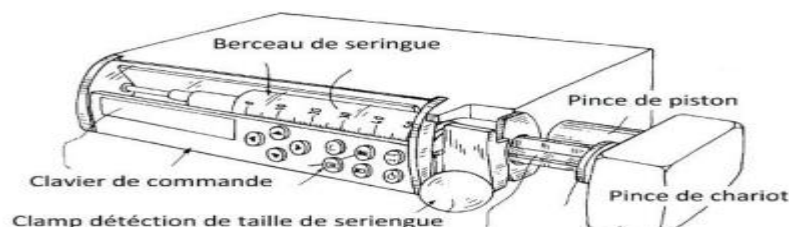


Figure 1.2: Partie mécanique de pousse seringue

## 1.4.3 Différents types des Pousse seringues électriques

Il existe des pousse-seringues à une simple ou plusieurs voies (généralement deux) ce qui permet une injection en différents sites de différents médicaments à un même patient.

### ❖ Pousse seringue à simple voie :

Une Pousse Seringue à simple voie ou mono voie c'est un dispositif qui permet d'injecter une seule solution à la fois, le support de ce dernier peut acquérir une seule seringue de n'importe quel dosage, on les trouve généralement dans les blocs opératoires.

### ❖ Pousse seringue à double voie :

Il existe des poussettes seringues double voies monobloc (voir figure 1.3). Ils intègrent les



Figure 1.3:Pousse seringue à double voie

Mêmes contraintes qu'une pousse-seringue standard (simple voie). Mais ce dernier peut acquérir et gérer deux seringues avec deux solutions il permet de faire la fusion des solutions ou bien gérer chaque seringue indépendamment.

#### 1.4.4 Différents modes d'administration des Pousse Seringues

Un pousse seringue peut fonctionner selon un ou plusieurs modes :

- ✧ Le mode "**PERFUSION CONTINUE**" qui permet de régler la quantité à injecter ainsi que le débit.
- ✧ Le mode "**AIVT**" (Anesthésie Intra Veineuse Totale) qui permet de régler le débit ainsi que la posologie en fonction de l'âge, du poids et du sexe du patient. C'est le pousse seringue qui calcule lui-même la quantité à injecter.
- ✧ Le mode "**AIVOC**" (Anesthésie Intra Veineuse à Objectif de Concentration) dont la dose est calculée en fonction de la concentration plasmatique souhaitée.

##### ❖ Le mode perfusion continue :

C'est le plus simple, le plus basique et le plus utilisé. La très grande majorité des PSE sont destinés à cet usage. Il suffit de régler un débit en millilitres par heure et l'appareil le délivre. Les PSE modernes proposent de plus en plus de régler une dose/kg/heure (voir par minute ou par heure), mais sans effectuer le calcul de posologie. C'est à dire que c'est l'opérateur lui-même qui détermine la dose et non le PSE qui va la calculer selon une formule. [2]

##### ❖ Le mode AIVT :

Dans ce mode, l'utilisateur va régler le débit de perfusion, une posologie et c'est le PSE qui va décider de la quantité de produit à perfuser. Pour ces appareils, il faut renseigner l'âge du patient, son sexe et son poids. En fonction des algorithmes les champs à remplir peuvent différer. Plus souvent utilisés en anesthésie, plus rarement en réanimation, ils permettent par



exemple de délivrer une dose d'induction (la dose pour endormir le patient au début d'une procédure), puis un débit constant en fonction de la posologie souhaitée. [3]

#### ❖ Le mode AIVOC :

Ce mode est considéré comme étant un sous mode du mode TIVA, mais son fonctionnement diffère assez largement, nous le traiterons donc de façon spécifique. Il propose de délivrer une médication selon le principe d'une dose à objectif de concentration plasmatique, c'est à dire en quantité de médicament dans le plasma sanguin. Quels que soient les modes, un certain nombre de PSE sont conçus afin de pouvoir se brancher sur une station d'accueil. Source d'énergie pour maintenir les batteries en charge et faire fonctionner l'appareil, ces stations peuvent proposer des fonctions de commande à distance ou d'asservissement. On peut ainsi commander ou surveiller à distance les PSE ou encore effectuer un relais de médicament lorsqu'une seringue arrive à son terme.[4]

### 1.4.5 Avantage et critique de l'existant

La fiabilité de ces dispositifs repose essentiellement sur la qualité (constance et précision) des débits annoncés et mesurés. On ne doit enregistrer aucun changement de rythme de perfusion qui n'ait été programmé. La prise en une seule injection du médicament ne permet pas de maintenir un effet optimal et constant de l'action thérapeutique. Au cours des premières minutes qui suivent une injection unique la concentration peut atteindre une valeur élevée, pouvant provoquer dans certains cas des incidents graves. C'est pourquoi on lui préfère la méthode des injections multiples à doses réduites, administrées en continu ou à intervalles de temps régulièrement espacés. Cependant l'injection à intervalles de temps régulièrement espacés présente les inconvénients suivants :

- ↳ Accroissement du nombre de manipulations et des risques d'erreurs ;
- ↳ Interventions plus fréquentes du personnel infirmier ;
- ↳ Augmentation des risques septiques ;
- ↳ Contraintes pour le patient.

De plus, l'utilisation du pousse seringue pour des injections continues permet une injection lente et très précise de l'agent thérapeutique. Généralement les avantages et les PES sont :

- ↳ Précision, facilité de mise en place et perfusion de grands volumes.
- ↳ Matériel adapté aux médicaments photosensibles (Lasix).
- ↳ Fonctionnement de l'appareil de façon autonome.

Malgré tous ces avantages, les PSE classiques manquent de connectivité, de commandabilité et de supervision à distance... Ces inconvénients peuvent être comblés en ayant recours aux outils de l'IOT pour concevoir une nouvelle génération de PSE les Pousse seringue connectés (PSC)

## 1.5 Normes et exigences

La norme internationale IEC 62304 – logiciels de dispositifs médicaux – processus du cycle de vie des logiciels est une norme qui spécifie les exigences du cycle de vie pour le développement de logiciels médicaux et de logiciels au sein des dispositifs médicaux. Il est harmonisé par l'Union européenne et les États-Unis et peut donc être utilisé comme référence pour se conformer aux exigences réglementaires de ces deux marchés. La norme est composée d'une exigence générale, et de 5 processus, dont seul le processus de développement du logiciel nous concerne [5].

### 1.5.1 Exigence générale

L'exigence consiste à identifier le système de management de qualité, le système de gestion de risque à utiliser, et de classifier le niveau de sécurité du logiciel. Ci-dessous les niveaux de sécurité possible et leur conséquence au cas d'une défaillance du système :

- ↳ Classe A : Aucune blessure ou atteinte à la santé n'est possible.
- ↳ Classe B : Une BLESSURE NON GRAVE est possible.
- ↳ Classe C : La mort ou une BLESSURE GRAVE est possible.

La pousse seringue électrique à un niveau de sécurité variable selon l'utilisation. Par exemple d'un département pédiatrie ou l'utilisation fréquente est l'infusion des compléments alimentaires la classification sera B, par contre dans une chambre d'urgence, ou soins intensifs, ou la plupart des médicaments utilisés sont dangereux en haute concentration. Une erreur de dosage peut provoquer la mort du patient, le pousse seringue est classifié C.

Vue que notre produit vise toute utilisation possible de la pousse seringue nous avons attribué le niveau de sécurité B au processus de développement logiciel.

## 1.5.2 Processus de développement logiciel

Le processus de développement du logiciel représente des étapes à suivre pour qu'elle soit conforme à la norme IEC 62304. Les étapes à suivre sont régies par le niveau de sécurité attribué. Le tableau 1.1 décrit les étapes à suivre selon le niveau de sécurité.

Etape	Classe A	Classe B	Classe C
Planification du développement logiciel	X	X	X
Analyse des exigences logicielles	X	X	X
Conception architecturale du logiciel		X	X
Conception détaillée du logiciel			X
Implémentation de l'unité logicielle	X	X	X
Vérification de l'unité logicielle		X	X
Intégration logicielle et tests d'intégration		X	X
Test du système logiciel	X	X	X
Version du logiciel	X	X	X

Tableau 1.1: Etapes de développement logiciel et niveau de sécurité.

## 1.6 Méthodologie de travail

### 1.6.1 Problématique

Afin de respecter les exigences de la norme IEC 62304 et étant donné les différentes tâches dans ce projet et les diverses technologies utilisées nous aurons toujours besoin de tester le code source. Donc le basculement d'une version à une autre, est difficile et consomme beaucoup de temps. D'autre part la supervision de l'avancement du projet est nécessaire, ainsi que, la nécessité de garder un journal décrivant l'historique des différentes étapes du cycle de développement de la solution proposée.

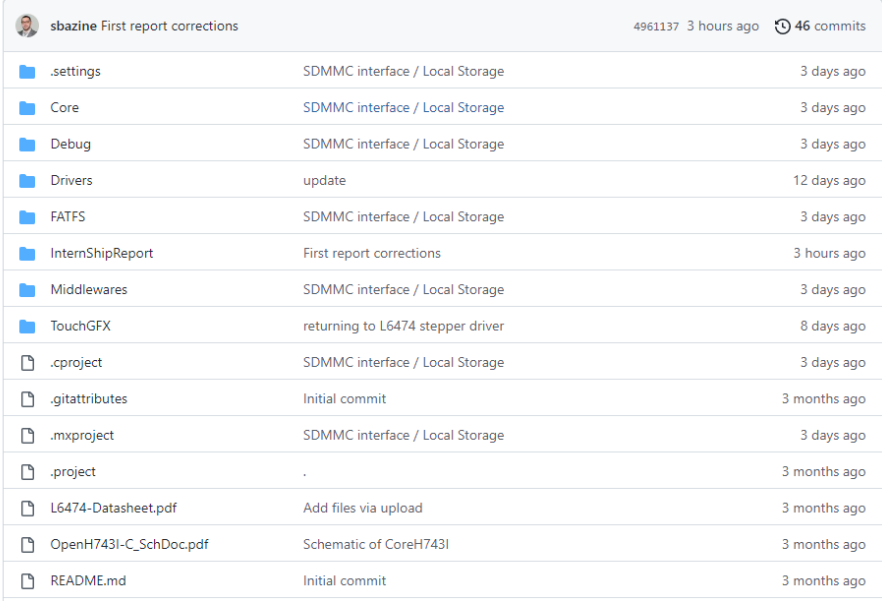
### 1.6.2 Solution

L'outil le plus utilisé aujourd'hui est **GIT** pour le contrôle de version. Git est un projet open source développé par le fameux **Linus Torvald** (le créateur du noyau linux). A chaque commit, on garde une trace du changement apporté au code. Si une erreur est commise on peut revenir en arrière et comparer les versions antérieures de code. Pour héberger les dépôts

Git on a utilisé GitHub qui nous accompagne dans notre collaboration avec nos contributeurs (l'équipe 3DWAVE, l'équipe ACTIA, Notre encadreur universitaire M. Sadok BAZINE.

### 1.6.3 Implémentation

Le projet sur GitHub est créé sous le nom de “Syringe-Pump-Project”, il admet le code source ainsi que les datasheets des composants utilisés, même ce rapport est disponible il est développé de la même manière (voir figure 1.4).



















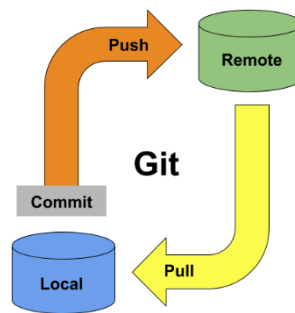
 sbazine First report corrections	4961137 3 hours ago 46 commits
 .settings	SDMMC interface / Local Storage 3 days ago
 Core	SDMMC interface / Local Storage 3 days ago
 Debug	SDMMC interface / Local Storage 3 days ago
 Drivers	update 12 days ago
 FATFS	SDMMC interface / Local Storage 3 days ago
 InternShipReport	First report corrections 3 hours ago
 Middlewares	SDMMC interface / Local Storage 3 days ago
 TouchGFX	returning to L6474 stepper driver 8 days ago
 .cproject	SDMMC interface / Local Storage 3 days ago
 .gitattributes	Initial commit 3 months ago
 .mxproject	SDMMC interface / Local Storage 3 days ago
 .project	. 3 months ago
 L6474-Datasheet.pdf	Add files via upload 3 months ago
 OpenH743I-C_SchDoc.pdf	Schematic of CoreH743I 3 months ago
 README.md	Initial commit 3 months ago

Figure 1.4: Structure du projet sur Git Hub

Dès que nous sommes satisfaits du code on fait un commit vers la branche main à l'aide de la commande “git commit -m “message””. Puis pour publier les changements locaux et les charger sur le serveur on utilise la commande “git push”. À l'aide de la commande “git pull” les contributeurs peuvent faire un ketch du contenu et le télécharger pour le modifier (figure 1.5). Pour revenir en arrière on utilise la commande “git reset -



*Figure 1.5 : Cycle de travail avec  
l'outils Git*

Hard”. Cette méthodologie nous a aidés dans beaucoup de situations dont on a besoin de retourner en arrière vers une version spécifique où on a avancé tellement loin que c’est très difficile de retourner manuellement.

## 1.7 Conclusion

Dans ce chapitre, nous avons présenté le cadre général du projet ainsi que les solutions existantes dans le marché et leurs faiblesses. Enfin nous avons présenté les normes et les exigences générales du développement logiciel ainsi que la méthodologie de travail. Dans le chapitre suivant nous mettons la main sur la solution proposée ainsi que l'architecture globale matériels et logicielle.

# **Chapitre 2 : Analyse et spécification des besoins**

## **2.1 Introduction**

Dans ce chapitre, nous présenterons la solution à développer ainsi que l'architecture globale du projet en précisant l'ensemble des matériels et logiciels utilisés.

## **2.2 Spécification des besoins**

### **2.2.1 Expression du besoin**

Dans le domaine de la santé, le cadre médical est confronté à plusieurs situations, parfois critiques et nécessite une disponibilité à plein temps des infirmiers pour administrer les médicaments aux patients. Tel que le cas de covid19 et surtout pour les malades dans un état comateux, qui nécessitent une alimentation sous forme de perfusion, suivant plusieurs modes et sur une période de temps long, qui peut prendre des semaines et parfois des mois. Parmi les appareils utilisés pour la réalisation de cette opération on peut citer le pousse seringue électrique. Dans certains cas, la quantité de produit administré par injection à un patient doit être fractionnée dans le temps. (Voir figure 2.1)

Outre les avantages de l'utilisation des PSE, qui ont été largement détaillés dans le chapitre précédent, nous avons évoqué l'intérêt que peut apporter le mariage des outils de l'IOT à la technologie des PSE, ceci sera exprimé avec plus de détails dans la section suivante.

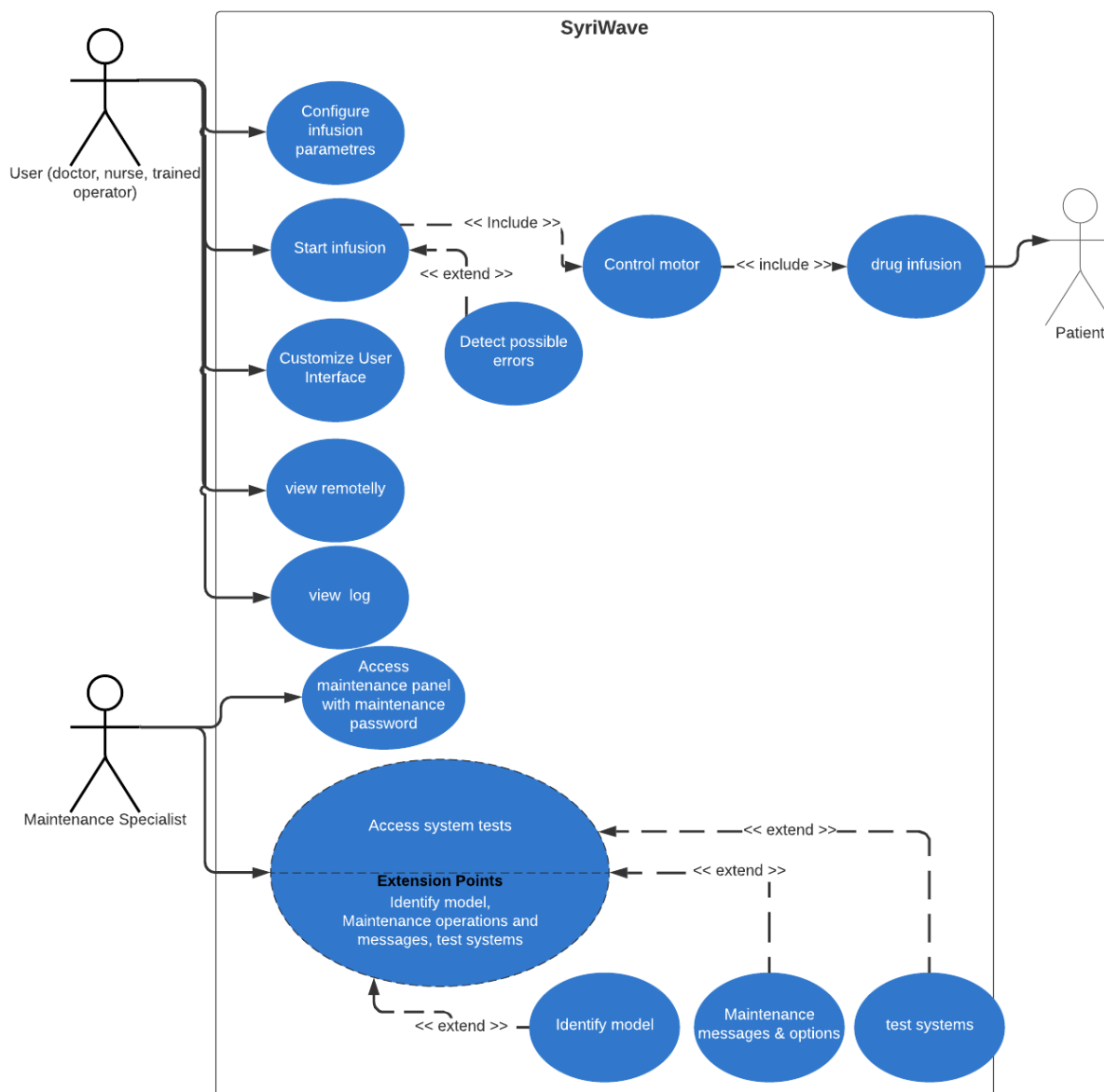


Figure 2.1 : Diagramme de cas d'utilisation de la pousse seringue

## 2.2.2 Cahier des charges

La définition d'un cahier des charges était l'un de nos principaux objectifs.

Il est question dans ce projet de développer une solution électronique pour le pilotage d'un modèle mécanique d'un Pousse Seringue Électrique connectée (PSC).

La solution envisagée doit satisfaire les exigences suivantes :

- Propose une Interface Homme Machine (IHM) permettant une interaction étroite et un accès facile aux principales fonctionnalités du PSE.

- Elaborer une carte de commande basée sur un microcontrôleur STM32 incluant tous les composants nécessaires pour la commande du moteur et l'exploitation des capteurs installés sur le modèle mécanique du PSE, ainsi que la gestion de l'IHM.
- Doter le prototype de(s) moyen(s) de communication nécessaire(s).

Une solution logiciel exécutant les tâches du PSE développer selon la norme IEC 62304.  
(Voir figure 2.2)

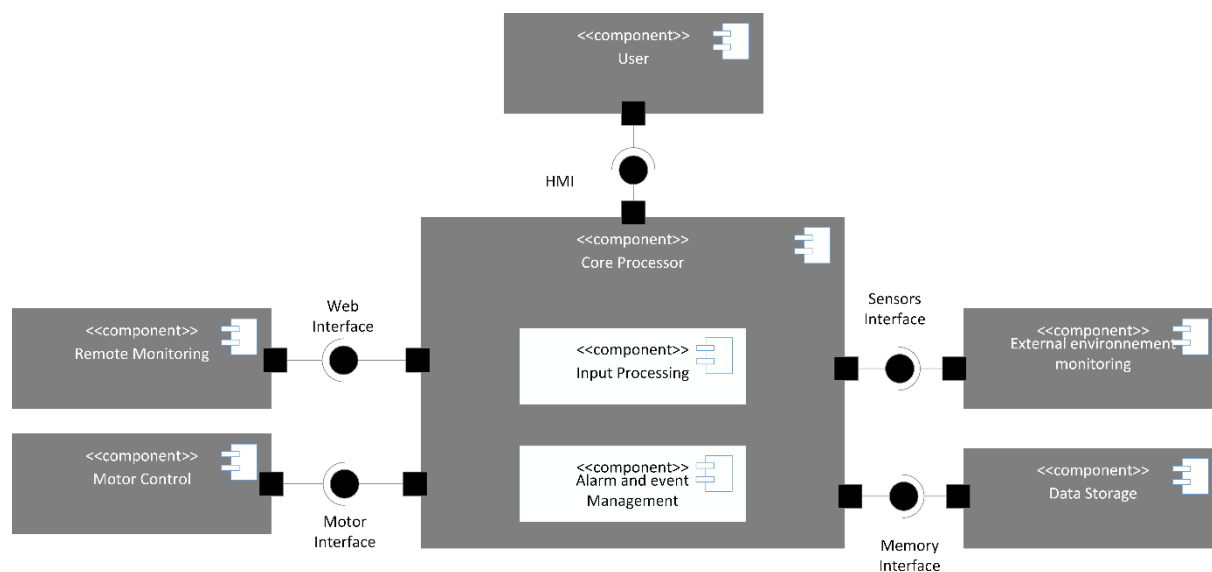


Figure 2.2: Diagramme des composants de la pousse seringue

## 2.3 Architecture globale du PSC

### 2.3.1 Schéma synoptique du PSC

Nous nous limitons dans cette partie à l'architecture et aux composants électroniques dont on aura besoin pour le développement de la partie électronique embarquée du PSC. Sachant que, la partie mécanique est développée par une autre équipe au sein de la société d'accueil.  
(Voir figure 2.3).



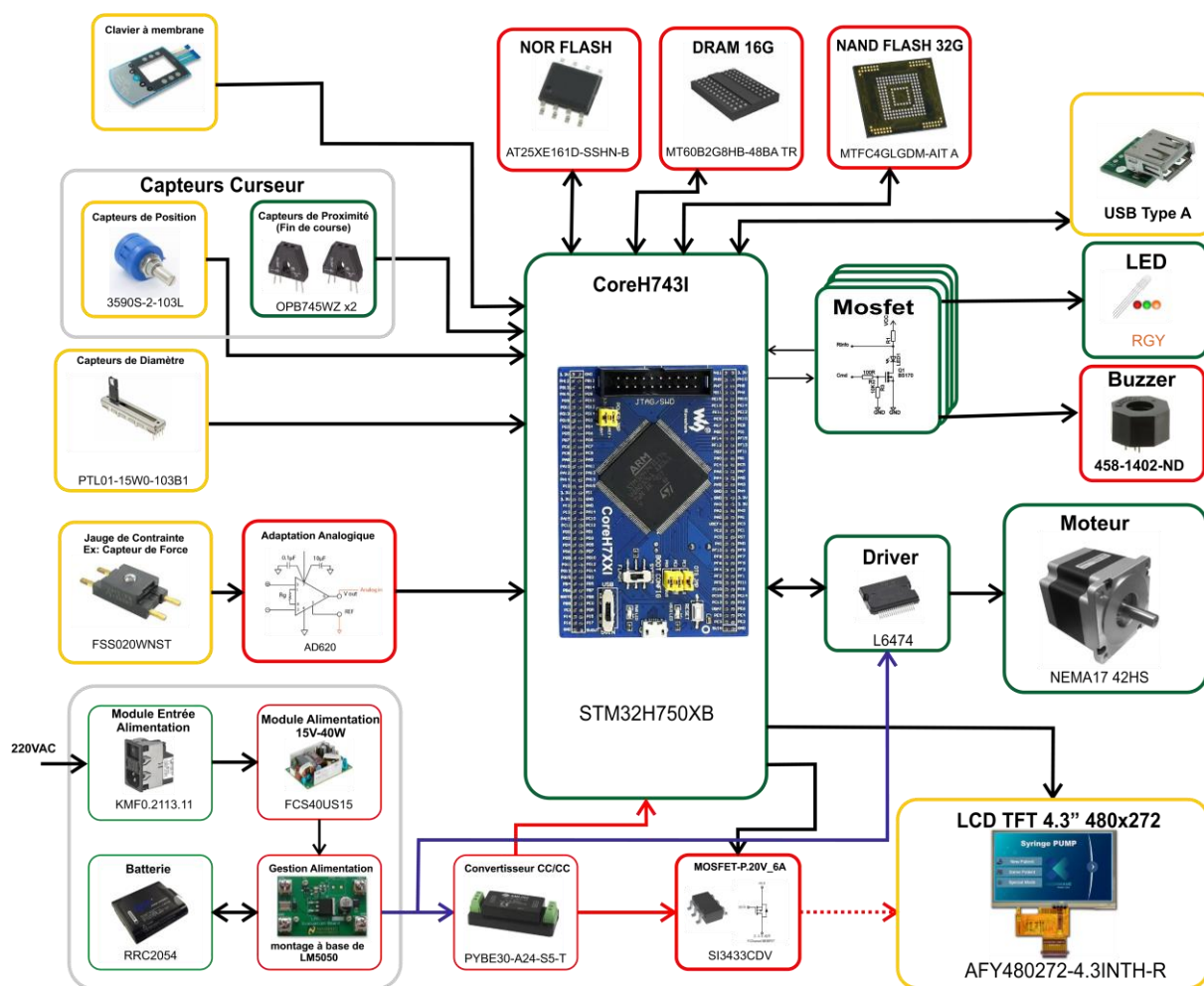


Figure 2.3: Schéma synoptique du PSC

## 2.3.2 Détails du matériel utilisé :

### ❖ OpenH743I-C

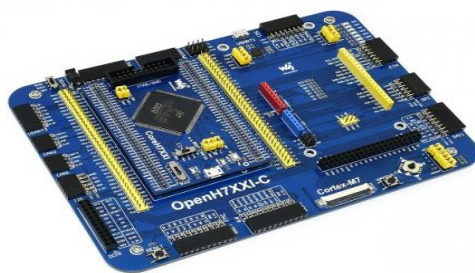


Figure 2.4 : Carte de développement OpenH743I-C

OpenH743I-C est une carte de développement conçue pour le microcontrôleur STM32H743IIT6. L'OpenH743I-C prend en charge une extension supplémentaire avec diverses cartes d'accessoires en option pour une application spécifique (voir Annexe 1).

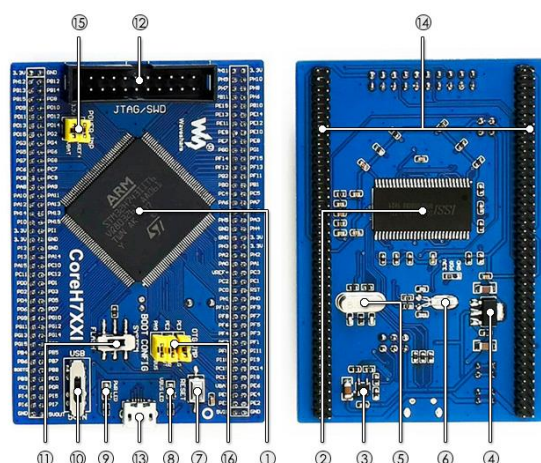
❖ **STM32H743IIT6 :**

Figure 2.5: Composants du L'STM32H743IIT6

N°	Nom de composant	Description
1	STM32H743IIT6	Le MCU STM32 haute performance qui comprend : Noyau : RISC 32 bits Cortex-M7 + FPU double précision + accélérateur graphique Chrom-ART Fonctionnalité : instructions DSP à cycle unique Fréquence de fonctionnement : 480 MHz, 1027 DMIPS / 2,14 DMIPS/MHz Tension de fonctionnement : 1,62 V-3,6 V Forfait : LQFP176 Mémoires : 2MB Flash, 1MB RAM (864KB utilisateur+192KB TCM+4KB Backup) Convertisseurs AD et DA : 3 x AD (16 bits); 2 x DA (12 bits) Débogage/programmation : prend en charge les interfaces JTAG/SWD et IAP
2	IC42S16400J / IS42S16400J	SDRAM 1 Meg Bits x 16 Bits x 4 Banks (64-MBIT)
3	STMP52151STR	Dispositif de gestion d'alimentation USB embarqué
4	AMS1117-3.3 :	3.3V voltage régulateur
5	8M Crystal	
6	32.768K Crystal	Pour RTC interne avec étalonnage
7	Reset Button	
8	VBUS LED	Indicateur de port USB
9	PWR LED :	Power Indicateur
10	Commutateur d'alimentation	Alimenté par une connexion 5Vin ou USB

11	Sélection du mode de démarrage	Pour configurer la broche BOOT0
12	JTAG/SWD interface	Pour le débogage/programmation
13	Connecteur USB	Prend en charge l'appareil et/ou l'hôte
14	Extension des broches du MCU	VCC, GND et toutes les broches d'E/S sont accessibles sur les connecteurs d'expansion pour une expansion ultérieure
15	SAUTEUR DE PUISSANCE	

Tableau 2.1 : Les différents composants du STM32H743IIT6

### ❖ MOTEUR PAS à PAS (NEMA17) :

Les moteurs pas à pas sont généralement utilisés pour les tâches de positionnement exigeantes qui demandent un niveau de précision très élevé. Les moteurs pas à pas sont entraînés par un champ électromagnétique qui fait tourner le rotor d'un petit angle, le pas, ou d'un multiple de ce pas.



Figure 2.6: Moteur PAS à PAS  
(NEMA17)

#### ✓ Caractéristiques :

- 200 pas par révolution : 1,8 degrés
- Bobine 1 : Rouge (A+) et Bleu (A-). Bobine # 2 Vert (B+) & Noir (B-).
- Moteur bipolaire, nécessite 2 ponts en H ponts !
- Dimension 42 mm / 1,65 "corps carré
- Fixation : 31mm / 1,22 trous de montage carrés, vis métriques de 3mm "(M3)
- Arbre d'entraînement de diamètre 5mm, 23.5 mm de long, avec un méplat usiné
- Tension nominale 12V (vous pouvez utiliser une tension inférieure, mais le couple chutera) avec courant 1.7A max
- Couple de maintien 40 N.cm.Min
- Couple de détente 2,2 N.cm

- Vendu avec un câble de connexion de 100cm avec extrémité en DuPont femelle au pas de 2.54mm

❖ **X-NUCLEO-IHM01A1 (L6474) :**



*Figure 2.7 : Stepper motor driver shield*

Le X-NUCLEO-IHM01A1 est une carte d'extension de pilote moteur pas à pas basée sur le L6474. Elle fournit une solution facile à utiliser pour piloter un moteur pas à pas, Le contrôle de courant avancé du L6474 offre des niveaux élevés de performance et de robustesse. Comme le montre la figure 2.7 la carte est compatible avec le connecteur Arduino UNO R3 et supporte l'ajout d'autres cartes qui peuvent être empilées pour piloter jusqu'à trois moteurs pas à pas avec une seule carte STM32.(Voir figure2.7).

✓ Caractéristiques :

- Tension de fonctionnement : 8 - 45 V
- Courant de crête de sortie de 7,0 A (3,0 A r.m.s.)
- MOSFET à faible puissance RDS (on)
- Vitesse de balayage MOS de puissance programmable
- Jusqu'à 1/16 de micro-pas
- Contrôle du courant avec décroissance adaptative
- Détection de courant non dissipative
- Interface SPI
- Faibles courants de repos et de veille
- Surintensité programmable non dissipative
- Protection sur tous les MOS de puissance
- Protection contre la surchauffe à deux niveaux

❖ **LCD TFT 4.3 :**

Les écrans LCD TFT offrent plusieurs avantages par rapport aux autres types d'écrans (CRT, Plasma). Il est léger, fin, économe en énergie et peu coûteux, ce qui les rend dominants dans le monde de l'affichage. (Voir figure 2.8).



Figure 2.8 : LCD TFT 4.3

✓ Caractéristiques :

- Dimensions du contour : 105,14 x 66,2 mm
- Résolution : 480 x 272.
- Zone active : 95,04 x 53,856 mm
- Interface : RVB.
- Driver IC : HX8527A.
- Tactile : résistive

✚ **ESP8266 :**



Figure 2.9 : Node MCU ESP8266

L'esp8266 est un module Wi-Fi qui facilite la communication TCP/IP et l'accès à l'internet. Un exemple de caractéristiques est indiqué ci-dessous.

✓ Caractéristique :

- 32-bit RISC CPU : Tensilica Xtensa LX106, 80 MHz ;
- 64 Kio de RAM instruction, 96 Kio de RAM data ;
- QSPI flash externe - 512 Kio à 4 Mio (supporte jusqu'à 16 Mio) ;
- IEEE802.11 b/g/n WIFI :

- TR switch intégré, balun, LNA, amplificateur de puissance
- Authentification par WEP ou WPA/WPA2 ou bien réseau ouvert
- Certaines variantes supportent une antenne externe
- 16 broches GPIO
- Interfaces SPI
- Interface avec DMA (partageant les broches avec les GPIO) ;
- UART sur des broches dédiées, plus un UART dédié aux transmissions pouvant être géré par GPIO2 ;
- 1 10-bit ADC

❖ **Capteur de position (3590S-2-103L) :**

Il s'agit d'un potentiomètre rotatif à dix tours, utilisé pour mesurer la position de la seringue



Figure 2.10 : Capteur de position

✓ Caractéristiques :

- Support de douille
- Fonction de broche AR en option
- Arbre et douilles en plastique ou en métal
- Bobiné
- Cosses à souder ou broches PC
- Scellable (joint complet du corps)
- Conçu pour une utilisation dans les applications IHM

❖ **Capteur de diamètre (PTL01-15W0-103B1) :**

IL s'agit d'un Potentiomètre linéaire utilisé pour mesurer le diamètre de la seringue



Figure 2.11 : Capteur de diamètre (PTL01-15W0-103B1)

✓ Caractéristiques :

- Type : Slide Potentiometer with LED
- Course : 100mm
- Resistance : 10khoms
- Données de puissance : 200M/W
- Tolérance : 20%
- Type de bande résistive : Linéaire
- Longueur : 35mm
- Largeur : 9mm
- Hauteur : 7mm

❖ Capteur jauge de contrainte (FSS020WNST) :



Figure 2.12 : Capteur jauge de contrainte

✓ Caractéristique :

- Force de commande : 20N
- Précision : 0.5%
- Type de sortie : Analogique

- Style de montage : SMD/SMT
- Tension d'alimentation : 12V

❖ **Batterie (RRC2054) :**

RRC Power Solutions Smart Battery Packs sont des batteries standard avec des certifications et des approbations mondiales. RRC Power Solutions Smart Battery Packs sont capables de communiquer via SMBus (SMART standard) / I2C et disposent de la technologie Lithium-Ion pour les applications médicales, militaires et industrielles. Ces batteries sont conçues pour être utilisées avec les chargeurs RRC-SMB-MBC et RRC-SMB-UBC. (Voir figure 2.13).



Figure 2.13 : Batterie (RRC2054)

✓ Caractéristique :

- Nombre de batteries : 4 Battery
- Tension de sortie : 15v
- Capacité : 3200 mAh
- Longueur : 77.4 mm
- Hauteur : 22.4 mm

❖ **RRC-PMM240 :**

C'est un Module de gestion de l'alimentation pour les applications mobiles (voir figure 2.14).  
Un exemple des caractéristiques ci-dessous :



Figure 2.14 : RRC-PMM240



✓ Caractéristique :

- 240,00 W max. puissance de sortie à l'application
- 82,00 W max. puissance de sortie vers la batterie
- Sélection automatique de la source d'alimentation
- Large plage de tension d'entrée CC
- Données 3D disponibles du module de gestion de l'alimentation et des batteries pour une intégration facile dans l'utilisateur vers l'application
- Convient à la batterie intelligente standard de RRC au format RRC20xx

❖ **Buzzer (458-1402-ND) :**



*Figure 2.15 : Buzzer (458-1402-ND)*

Il s'agit d'un actionneur utilisé lorsqu'il y a un problème, y compris l'arrêt du déplacement de la seringue.

✓ Caractéristique :

- Type d'entrée : DC
- Voltage : 5V
- Fréquence : 2.075KHZ
- Durée : variable
- Mode d'opération : Pin select able

### 2.3.3 Architecture Software du PSC

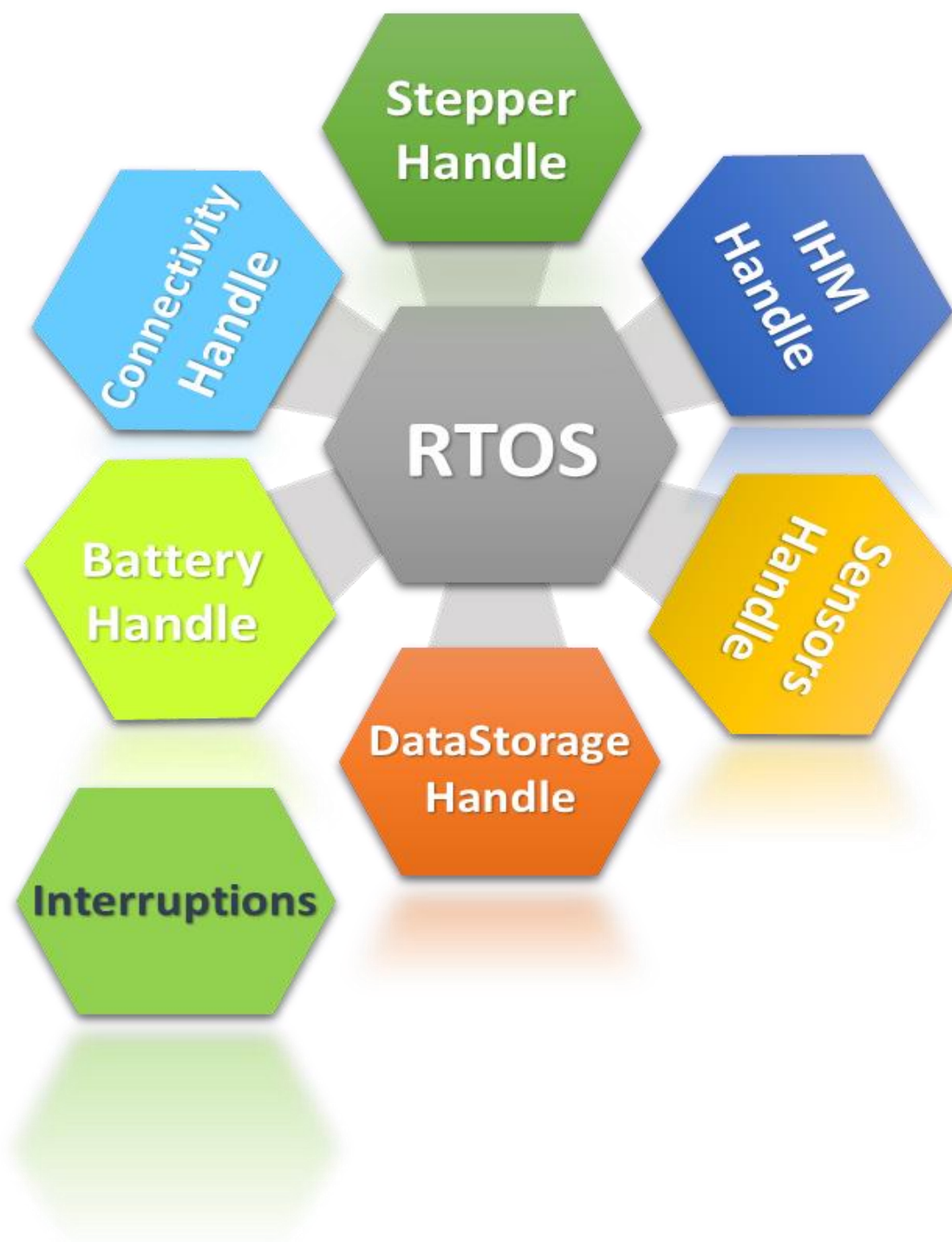


Figure 2.16 : Architecture SOFTWARE du PSC

Dans le chapitre suivant on va présenter les détails sur le choix de cette architecture logicielle.

### 2.3.4 Outils et logiciels utilisés :

❖ STM32CUBEIDE :



STM32CubeIDE : est un outil de développement multi-OS tout-en-un, qui fait partie de l'écosystème logiciel STM32Cube. STM32CubeIDE est une plate-forme de développement C/C++ avancée avec configuration périphérique, génération de code, compilation de code, et des fonctionnalités de débogage pour les microcontrôleurs et microprocesseurs STM32. Il est basé sur le framework Eclipse®/CDT™ et GCC toolchain pour le développement et GDB pour le débogage.



**TOUCHGFX :**



TouchGFX : est un cadre logiciel graphique gratuit avancé optimisé pour les Microcontrôleurs STM32. Profitant des fonctionnalités graphiques STM32 et architecture, TouchGFX ce qui accélère la création des objets de l'IHM grâce à la création des interfaces graphiques de type smartphone.

## 2.3 Conclusion

Dans ce chapitre nous avons introduit le cahier de charge de la solution proposée puis nous avons présenté l'architecture matériel et logicielle et enfin nous avons finis par une description générale des différents éléments utilisé dans ce projet.

# Chapitre 3 : Conception et réalisation

## 3.1 Introduction

Après identification des éléments clé pour développer un pousse seringue connecté il est temps de mettre la main en œuvre et commencer la réalisation pratique de la solution. Dans ce chapitre nous parlons tout d'abord sur le cœur du notre système ainsi que la solution logicielle utilisée. Puis nous passons par la présentation en détails des différents tâches existantes dans le système et on finira par introduire les tâches qui sont encore de développent.

## 3.2 OS kernel

### 3.2.1 Système d'exploitation temps réel

L'un des exigences majeures pour les dispositifs médicaux est la stabilité du système (**soft + le soft externe + le hardware autour**).

Notre Solution logiciel était alors de travailler dans le domaine temps réels, où le taux d'erreur est minimal et la stabilité est maximale.

#### ➤ RTOS

Le système d'exploitation temps réel est un OS qui gère plusieurs tâches concurrentes selon leurs degrés de priorités, il est utilisé quand il y a des exigences temporelles sur les processus. Ce type d'ordonnancement, appelé ordonnancement préemptif. (Voir figure 3.1)

RTOS garantis la performance maximale du processeur et la bonne gestion de la mémoire

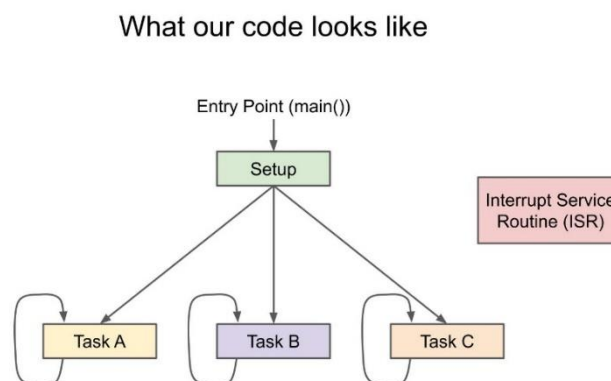


Figure 3.1 : Structure du code

ainsi que le fonctionnement sans erreur (Error-Free) offert par ces types de systèmes.

### ➤ FreeRTOS/CMSISV2

Dans le domaine de l'embarqué les ressources sont relativement limitées en termes de mémoire et de traitements. Dans ce projet on utilise un microcontrôleur Stm32H7 (**Arm Cortex-M7**) qui est de la catégorie haute performance, mais on reste toujours limités de ressources.

C'est pour cela qu'on a choisis **FreeRTOS**, c'est un système d'exploitation embarqué multitâches temps réel préemptif supporte actuellement 35 architectures. Il est aujourd'hui parmi les plus utilisés dans le marché des systèmes d'exploitation temps réel pour l'embarquée grâce à sa faible taille qui est de l'ordre de 4000 à 9000 octets.

On a utilisé le **CMSIS-RTOS API v2** comme une couche d'abstraction à FreeRTOS afin de garantir un système optimisé et améliorer la portabilité du code entre les différents processeurs ARM.

## 3.2.2 Tâches Et Queues (File d'attente)

### ❖ Création des Taches

Pour développer une application basée sur un OS, on décompose l'application en un ensemble de tâches. Dans FreeRTOS une tâche est fonction C contenant une boucle infinie et ne renvoie pas un résultat.

```
Void vATaskFunction (void *paramètres)
{
    for (;;)
    {
    }
}
```

Une tâche est créée par l'intermédiaire de la fonction "osThreadNew" qui retourne l'id de la tache RTOS

```
osThreadNew (osThreadFunc_t func, void *
argument, const osThreadAttr_t * attr)

[in] fun    thread function.

[in] argument pointer that is passed to the
thread function as start argument.

[in] attr thread attributes (les priorités sont spécifiées
(figure ci-dessous)).
```

Une tâche FreeRTOS peut se trouver dans l'un des états suivants :

- ↳ **Prête (Ready)** : une tâche qui possède toutes les ressources nécessaires à son exécution. Elle lui manque seulement le processeur.
- ↳ **Active (Running)** : Tâche en cours d'exécution, elle est actuellement en possession du processeur.
- ↳ **Attente (Blocked)** : Tâche en attente d'un événement (queue de messages, sémaphores, timeout ...). Une fois l'événement arrivé, la tâche concernée repasse alors à l'état prêt.
- ↳ **Suspendu (Suspended)** : tâche à l'état dormant, elle ne fait pas partie de l'ensemble des tâches ordonnançables. (Voir figure 3.2).

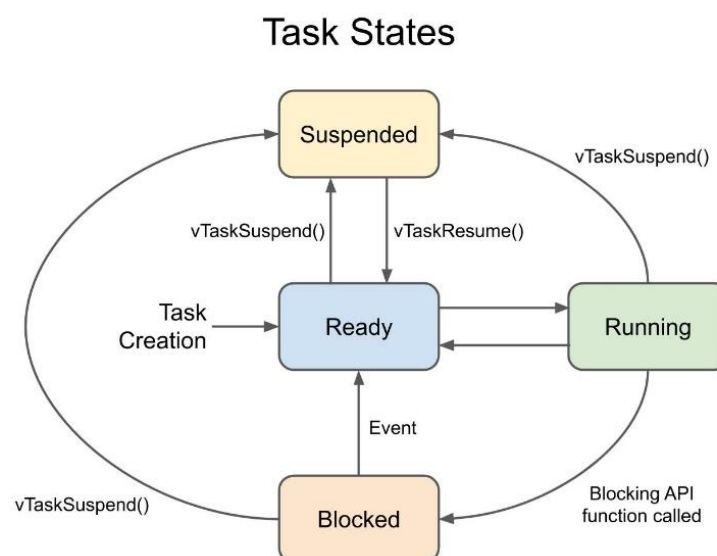


Figure 3.2: états des tâches

Ci-dessous Sont les taches utilisées dans le projet classé selon leurs (plus de détails sur le développement de ses tâches sont à partir de la section 3.3)

```

57  /* USER CODE END Variables */
58  /* Definitions for battery_manage */
59  osThreadId_t battery_manageHandle;
60  const osThreadAttr_t battery_manage_attributes = {
61    .name = "battery_manage",
62    .stack_size = 128 * 4,
63    .priority = (osPriority_t) osPriorityNormal,
64  };
65  /* Definitions for Stepper */
66  osThreadId_t StepperHandle;
67  const osThreadAttr_t Stepper_attributes = {
68    .name = "Stepper",
69    .stack_size = 256 * 4,
70    .priority = (osPriority_t) osPriorityHigh,
71  };
72  /* Definitions for Connectivity */
73  osThreadId_t ConnectivityHandle;
74  const osThreadAttr_t Connectivity_attributes = {
75    .name = "Connectivity",
76    .stack_size = 128 * 4,
77    .priority = (osPriority_t) osPriorityAboveNormal,
78  };
79  /* Definitions for Sensors */
80  osThreadId_t SensorsHandle;
81  const osThreadAttr_t Sensors_attributes = {
82    .name = "Sensors",
83    .stack_size = 256 * 4,
84    .priority = (osPriority_t) osPriorityNormal,
85  };
86  /* Definitions for IHM */
87  osThreadId_t IHMHandle;
88  const osThreadAttr_t IHM_attributes = {
89    .name = "IHM",
90    .stack_size = 128 * 4,
91    .priority = (osPriority_t) osPriorityNormal,
92  };

```

Figure 3.3: Priorités et tailles de chaque tache

#### ✓ Tache 1 : stepperHandle

Cette tache dispose la priorité maximale `osPriorityHigh` car elle gère le moteur pas à pas qui est le cœur du projet et tous les autres services fonctionnent en fonction de son état et de son avancement.

#### ✓ Tache 2 : IHMHandle

Elle dispose comme priorité `osPriorityNormal_1`, elle gère les flux de données entre l'interface homme machine et les autres taches.

#### ✓ Tache 3 : ConnectivityHandle

La connectivité admet le même degré de priorité que la tâche IHMHandle osPriorityAboveNormal comme propriétés toutes les données nécessaires sont envoyées à travers cette tâche vers le Cloud afin de les récupérer en temps réels par une application mobile de supervision.

#### ✓ Tache 4 : SensorsHandle

Cette tâche représente l'unité de traitements de tous les capteurs utilisés dans ce projet elle dispose comme priorité osPriorityNormal

#### ✓ Tache 5 : Battery\_manageHandle

La gestion de batterie est assurée par cette tâche avec une priorité osPriorityNormal. (Elle est en cours de développement).

```
/* Create the thread(s) */
/* creation of battery_manage */
battery_manageHandle = osThreadNew(StartBatteryManage, NULL, &battery_manage_attributes);

/* creation of Stepper */
StepperHandle = osThreadNew(Stepper_motor, NULL, &Stepper_attributes);

/* creation of Connectivity */
ConnectivityHandle = osThreadNew(Cloud_Connectivity, NULL, &Connectivity_attributes);

/* creation of Sensors */
SensorsHandle = osThreadNew(Sensors_measurements, NULL, &Sensors_attributes);

/* creation of IHM */
IHMHandle = osThreadNew(Interface, NULL, &IHM_attributes);

/* creation of DataStorage */
DataStorageHandle = osThreadNew(StartDataStorage, NULL, &DataStorage_attributes);
```

Figure 3.4 : Création des tâches

(La figure 3.4) ci-dessus montre comment on à faire la création des tâches.

### ❖ Queues (File d'attente) :

Avant de parler des queues il faut parler des problèmes majeurs lors de l'utilisation d'un système temps réels et surtout s'il s'agit d'un système préemptif. Les variables globales ne sont plus une solution optimale pour stocker l'information à cause de la concurrence des tâches. C'est très probable alors que deux tâches écrivent en même temps dans une variable x, les données ne sont plus utilisables dans ce cas. Beaucoup d'autres problèmes sont rencontrés lors de l'utilisation des variables globales.



Les queues sont utilisées pour résoudre ces problèmes avec une opération atomique c – à - d une écriture ou lecture dans la queue ne peut pas être interrompue avant la fin de l’opération. Comme le montre la figure 3.5 ci-dessous.

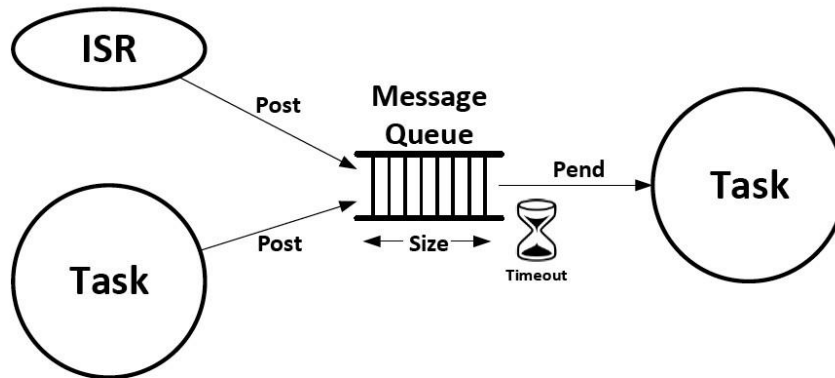


Figure 3.5: File d’attente et mode FIFO

Il s’agit d’un système FIFO (first input first output). Les files d’attentes sont utilisées dans ce projet dans tous les flux de communications inter tâches

La bibliothèque “Cmsis\_Os2” dispose les fonctions nécessaires pour créer, Lire et écrire dans les files d’attente (queues).

Pour créer une queue on utilise la fonction “osMessageQueueNew”, elle retourne en résultat l’id pour la file créée de type “osMessageQueueId\_t”. (Voir figure 3.6)

```

// Create and Initialize a Message Queue object.
// \param[in] msg_count maximum number of messages in queue.
// \param[in] msg_size maximum message size in bytes.
// \param[in] attr message queue attributes; NULL: default values.
// \return message queue ID for reference by other functions or NULL in case of error.
osMessageQueueId_t osMessageQueueNew (uint32_t msg_count, uint32_t msg_size, const osMessageQueueAttr_t *attr);

```

Figure 3.6 : Création d’une file d’attente (queue)

Pour déposer un message dans la file d’attente on utilise la fonction “osMessageQueuePut”, elle place le message pointé par “msg\_ptr” dans la file d’attente spécifiée par le paramètre “mq\_id”. Le paramètre “msg\_prio” est utilisé pour trier les messages en fonction de leur priorité (les chiffres les plus élevés indiquent une plus grande priorité) lors de l’insertion.

La fonction “osMessageQueueGet” lit le contenu de la file d’attente (queue) dont l’id est passé en paramètre, si la queue est vide (pas de messages) et elle a dépassé le délai maximal d’attente “timeout”, la fonction retourne “osErrorTimeout” si non elle retourne “osOK”. Comme la montre (la figure 3.7) ci-dessous.

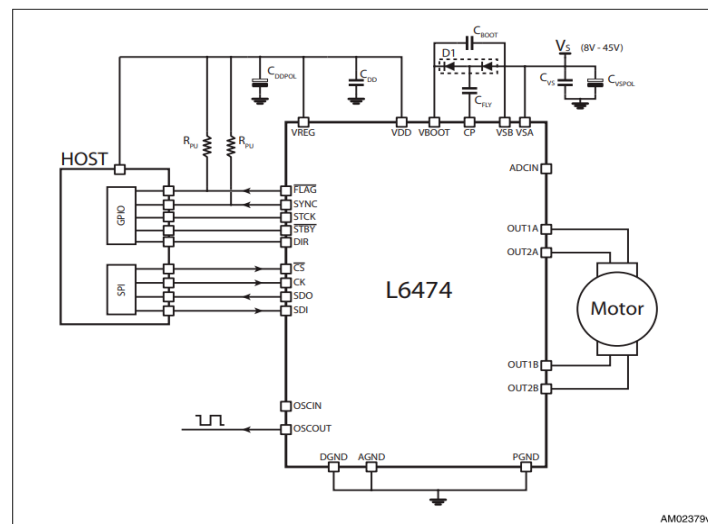
```
/* Infinite loop */
for(;;)
{
    if(osMessageQueueGet(FlowRateQHandle,&Flowrate , 10U, 100)==osOK && osMessageQueueGet(VolumeQHandle,&
        timeneeded= Time_Needed(Flowrate, volume_to_inject);
        laststep = timeneeded*L6474_GetCurrentSpeed(0);
        osMessageQueuePut(LastStepQHandle, &laststep, 1, 100);
    }
    // ***** 0 => StopMode , 8=> PauseMode *****
    if(osMessageQueueGet(ModeQHandle, &mode, 10U, 10U)==osOK && (mode==0 || mode == 8)){
        SyringeStop();
    }
    SyringeMove(Flowrate,radius);
}
/* USER CODE END Stepper_motor */
```

Figure 3.7 : Définition des fonctions de lecture et écriture dans une queue

### 3.3 Moteur pas à pas

#### 3.3.1 L6474 Driver / L6474.C :

Afin d'utiliser un moteur pas à pas il est nécessaire d'utiliser un "driver". Ces drivers permettent de transmettre la puissance électrique au moteur afin de le faire tourner (voir figure 3.8)



DocID022529 Rev 4



Figure 3.8 : Schéma de câblage du driver L6474

Nous travaillons avec la carte d'expansion **x-nucleo-ihm01a1** basée sur le driver L6474

La communication entre la carteH7 et le driver est à travers le protocole **SPI** (Serial Peripheral Interface) où Le microcontrôleur représente le Master or que le driver est l'esclave.

“Une liaison SPI (pour Serial Peripheral Interface) est un bus de données série synchrone baptisé ainsi par Motorola, au milieu des années 1980 qui opère en mode full-duplex. Les circuits communiquent selon un schéma maître-esclave, où le maître contrôle la communication. Plusieurs esclaves peuvent coexister sur un même bus, dans ce cas, la sélection du destinataire se fait par une ligne dédiée entre le maître et l'esclave appelée « Slave Select (SS).”, [6]

Le L6474 driver admet plusieurs registres qui sont responsable à convertir la commande SPI en une commande analogique du moteur pas à pas. Chaque registre admet une adresse bien déterminée, par exemple le registre **STEP\_MODE** admet l'adresse **0X16** est responsable à changer le mode du micro-pas (**Microstepping**). (Voir figure 3.10).

Programming manual

L6474

#### 9.1.10 STEP\_MODE

The STEP\_MODE register has the following structure:

Table 18. STEP\_MODE register

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1	SYNC_SEL			1 <sup>(1)</sup>	STEP_SEL		

1. When the register is written this bit should be set to 1.

The STEP\_SEL parameter selects one of five possible stepping modes:

Table 19. Step mode selection

STEP_SEL[2...0]			Step mode
0	0	0	Full step
0	0	1	Half step
0	1	0	1/4 microstep
0	1	1	1/8 microstep
1	X	X	1/16 microstep

Every time the step mode is changed, the electrical position (i.e. the point of microstepping sinewave that is generated) is reset to the first microstep.

Figure 3.9 : Structure du registre STEP\_MODE

Register address of the L6474 from L6474\_Registers\_t enum:

```
typedef enum {
    L6474_ABS_POS      = ((uint8_t) 0x01),
    L6474_EL_POS       = ((uint8_t) 0x02),
    L6474_MARK          = ((uint8_t) 0x03),
    L6474_TVAL          = ((uint8_t) 0x09),
    L6474_T_FAST        = ((uint8_t) 0x0E),
    L6474_TON_MIN       = ((uint8_t) 0x0F),
    L6474_TOFF_MIN      = ((uint8_t) 0x10),
    L6474_ADC_OUT        = ((uint8_t) 0x12),
    L6474_OCD_TH        = ((uint8_t) 0x13),
    L6474_STEP_MODE     = ((uint8_t) 0x16),
    L6474_ALARM_EN      = ((uint8_t) 0x17),
    L6474_CONFIG        = ((uint8_t) 0x18),
    L6474_STATUS        = ((uint8_t) 0x19),
    L6474_INEXISTENT_REG = ((uint8_t) 0x1F)
} L6474_Registers_t;
```

Figure 3.10 : Adresse des différents registres

La bibliothèque fournit par ST “X-CUBE-SPN1” peut gérer tous les commandes bas niveau à travers des fonctions prédéfini qui envoient des trames bien déterminés contenant l’adresse du registre ainsi que le code commande correspondant et des arguments si nécessaire, mais le problème que cette bibliothèque est compatible qu’avec les Nucléo F4, F3, F0, L0.

Dans ce cas, la première étape était d’adapté les fichiers “.h” (header files) avec notre carte. Les trimer et leurs channels, les brochages des pins SPI (MISO – MOSI – CLK – NSS(CS)), le pin de flags ainsi que de la remise à zéro, tous sont modifiés ...

Pour l’initialisation du driver on pourrait choisir entre utiliser le fichier des valeurs par défaut des registres “**l6474\_target\_config.h**”, ou bien de déclarer une variable de type Structure C “**L6474\_Init\_t**” où on spécifie tous les paramètres à initialiser comme la vitesse du moteur maximale et minimale (pas/s), l’accélération et la décélération (pas/s^2), les paramètres relatifs au courant, les alarmes...

Dans notre cas nous avons modifié le fichier des paramètres par défaut selon nos besoins, comme la montre la figure 3.12.

```

/* Private user code -----*/
/* USER CODE BEGIN 0 */

L6474_Init_t gL6474InitParams =
{
    1, // Acceleration rate in step/s2. Range: (0..inf).
    1, // Deceleration rate in step/s2. Range: (0..inf).
    1000, // Maximum speed in step/s. Range: (30..10000).
    1000, // Minimum speed in step/s. Range: (30..10000).
    250, // Torque regulation current in mA. (TVAL register). Range: 31.25mA to 4000mA.
    750, // Overcurrent threshold (OCD_TH register). Range: 375mA to 6000mA.
    L6474_CONFIG_OC_SD_ENABLE, // Overcurrent shutdown (OC_SD field of CONFIG register).
    L6474_CONFIG_EN_TQREG_TVAL_USED, // Torque regulation method (EN_TQREG field of CONFIG register).
    L6474_STEP_SEL_1_16, // Step selection (STEP_SEL field of STEP_MODE register).
    L6474_SYNC_SEL_1_2, // Sync selection (SYNC_SEL field of STEP_MODE register).
    L6474_FAST_STEP_12us, // Fall time value (T_FAST field of T_FAST register). Range: 2us to 32us.
    L6474_TOFF_FAST_8us, // Maximum fast decay time (T_OFF field of T_FAST register). Range: 2us to 32us.
    3, // Minimum ON time in us (TON_MIN register). Range: 0.5us to 64us.
    21, // Minimum OFF time in us (TOFF_MIN register). Range: 0.5us to 64us.
    L6474_CONFIG_TOFF_044us, // Target Switching Period (field TOFF of CONFIG register).
    L6474_CONFIG_SR_320V_us, // Slew rate (POW_SR field of CONFIG register).
    L6474_CONFIG_INT_16MHZ, // Clock setting (OSC_CLK_SEL field of CONFIG register).
    (L6474_ALARM_EN_OVERCURRENT | // Alarm (ALARM_EN register).
    L6474_ALARM_EN_THERMAL_SHUTDOWN |
    L6474_ALARM_EN_THERMAL_WARNING |
    L6474_ALARM_EN_UNDERVOLTAGE |
    L6474_ALARM_EN_SW_TURN_ON |
    L6474_ALARM_EN_WRONG_NPERF_CMD)
};
/* USER CODE END 0 */

//**

```

Figure 3.11 : Structure C L6474\_Init\_t

```

47 // The maximum number of devices in the daisy chain
48 #define MAX_NUMBER_OF_DEVICES (3)
49
50 /***** Speed Profile *****/
51
52 // Acceleration rate in step/s2 for device 0 (must be greater than 0)
53 #define L6474_CONF_PARAM_ACC_DEVICE_0 (1)
54 // Acceleration rate in step/s2 for device 1 (must be greater than 0)
55 #define L6474_CONF_PARAM_ACC_DEVICE_1 (160)
56 // Acceleration rate in step/s2 for device 2 (must be greater than 0)
57 #define L6474_CONF_PARAM_ACC_DEVICE_2 (160)
58
59 // Deceleration rate in step/s2 for device 0 (must be greater than 0)
60 #define L6474_CONF_PARAM_DEC_DEVICE_0 (1)
61 // Deceleration rate in step/s2 for device 1 (must be greater than 0)
62 #define L6474_CONF_PARAM_DEC_DEVICE_1 (160)
63 // Deceleration rate in step/s2 for device 2 (must be greater than 0)
64 #define L6474_CONF_PARAM_DEC_DEVICE_2 (160)
65
66 // Maximum speed in step/s for device 0 (30 step/s < Maximum speed <= 10 000 step/s)
67 #define L6474_CONF_PARAM_MAX_SPEED_DEVICE_0 (1000)
68 // Maximum speed in step/s for device 1 (30 step/s < Maximum speed <= 10 000 step/s)
69 #define L6474_CONF_PARAM_MAX_SPEED_DEVICE_1 (1)
70 // Maximum speed in step/s for device 2 (30 step/s < Maximum speed <= 10 000 step/s)
71 #define L6474_CONF_PARAM_MAX_SPEED_DEVICE_2 (1600)
72
73 // Minimum speed in step/s for device 0 (30 step/s <= Minimum speed < 10 000 step/s)
74 #define L6474_CONF_PARAM_MIN_SPEED_DEVICE_0 (1000)
75 // Minimum speed in step/s for device 1 (30 step/s <= Minimum speed < 10 000 step/s)
76 #define L6474_CONF_PARAM_MIN_SPEED_DEVICE_1 (800)
77 // Minimum speed in step/s for device 2 (30 step/s <= Minimum speed < 10 000 step/s)
78 #define L6474_CONF_PARAM_MIN_SPEED_DEVICE_2 (800)
79
80

```

Figure 3.12: Fichier de paramètre par défauts

Pour avoir une vitesse constante toute au long de l'opération de l'injection, l'accélération et la décélération sont initialisés à 1 (sans accélération / décélération).

La vitesse maximale et la vitesse minimale vont être modifiées dans le code selon le débit d'injection à l'aide des fonctions ci-dessous.

```
"uint16_t L6474_SetMaxSpeed(uint8_t deviceId, uint16_t newMaxSpeed)"
"uint16_t L6474_SetMinSpeed(uint8_t deviceId, uint16_t newMinSpeed)"
```

```
582 void L6474_AttachErrorHandler(void (*callback)(uint16_t)); //Attach a user callback to the erro
583 void L6474_AttachFlagInterrupt(void (*callback)(void)); //Attach a user callback to the flag
584 void L6474_CmdDisable(uint8_t deviceId); //Send the L6474_DISABLE command
585 void L6474_CmdEnable(uint8_t deviceId); //Send the L6474_ENABLE command
586 uint32_t L6474_CmdGetParam(uint8_t deviceId, //Send the L6474_GET_PARAM command
587 uint32_t param);
588 uint16_t L6474_CmdGetStatus(uint8_t deviceId); // Send the L6474_GET_STATUS command
589 void L6474_CmdNop(uint8_t deviceId); //Send the L6474_NOP command
590 void L6474_CmdSetParam(uint8_t deviceId, //Send the L6474_SET_PARAM command
591 uint32_t param,
592 uint32_t value);
593 uint16_t L6474_GetAcceleration(uint8_t deviceId); //Return the acceleration in pps^2
594 float L6474_GetAnalogValue(uint8_t deviceId, uint32_t param); //Get parameters (the value is for
595 uint16_t L6474_GetCurrentSpeed(uint8_t deviceId); //Return the current speed in pps
596 uint16_t L6474_GetDeceleration(uint8_t deviceId); //Return the deceleration in pps^2
597 motorState_t L6474_GetDeviceState(uint8_t deviceId); //Return the device state
598 motorDir_t L6474_GetDirection(uint8_t deviceId); //Return the device direction
599 motorDrv_t* L6474_GetMotorHandle(void); //Return handle of the motor driver
600 uint32_t L6474_GetFwVersion(void); //Return the FW version
601 int32_t L6474_GetMark(uint8_t deviceId); //Return the mark position
602 uint16_t L6474_GetMaxSpeed(uint8_t deviceId); //Return the max speed in pps
603 uint16_t L6474_GetMinSpeed(uint8_t deviceId); //Return the min speed in pps
604 uint8_t L6474_GetNbDevices(void); //Return the number of devices
605 int32_t L6474_GetPosition(uint8_t deviceId); //Return the ABS_POSITION (32b signe
606 motorStepMode_t L6474_GetStepMode(uint8_t deviceId); //Return the Step mode
607 motorStopMode_t L6474_GetStopMode(uint8_t deviceId); //Return the stop mode
608 void L6474_GoHome(uint8_t deviceId); //Move to the home position
609 void L6474_GoMark(uint8_t deviceId); //Move to the Mark position
610 void L6474_GoTo(uint8_t deviceId, int32_t targetPosition); //Go to the specified position
611 void L6474_HardStop(uint8_t deviceId); //Stop the motor
612 void L6474_HizStop(uint8_t deviceId); //Stop the motor and disable the pow
613 void L6474_Init(void* pInit); //Start the L6474 library
614 void L6474_Move(uint8_t deviceId, //Move the motor of the specified nu
```

Figure 3.13 : Les fonctions pour commander le moteur

### 3.3.2 Interruptions

Le L6474 contient Un ensemble très riche de protections (thermique, faible tension de bus, surintensité ...) On peut détecter ces irrégularités à travers le registre STATUS qui contient des flags indiquant l'état du driver, comme la montre (la figure 3.14) ci-dessous.

## 9.1.13 STATUS

Table 28. STATUS register

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8
1	1	1	OCD	TH_SD	TH_WRN	UVLO	WRONG_CMD
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
NOTPERF_CMD	0	0	DIR	0	0	1	HiZ

When HiZ flag is high, it indicates that the bridges are in high impedance state. Enable command makes the device exit from High Z state unless error flags forcing a High Z state are active.



DocID022529 Rev 4

41/53

Figure 3.14 : Structure du registre STATUS

Lors de l'initialisation on définit une limite pour chaque grandeur, si elle est dépassée une interruption est lancée dans notre programme, elle dispose toujours la priorité maximale. En cas d'interruption, une alarme est activée en fonction de l'état du driver. Pour le moment les alarmes s'agissent des toggles Led et une notification dans l'application mobile. Nous allons les modifier aux fins et à mesure avec des alarmes sonores selon les exigences générales des systèmes d'alarmes (EN 60601-1-8).

```

263 void MyFlagInterruptHandler(void)
264 {
265     /* Get the value of the status register via the L6474 command GET_STATUS */
266     uint16_t statusRegister = L6474_CmdGetStatus(0);
267
268     /* Check HiZ flag: if set, power bridges are disabled */
269     if ((statusRegister & L6474_STATUS_HIZ) == L6474_STATUS_HIZ)
270     {
271         // HiZ state
272         HAL_GPIO_TogglePin(LED2_GPIO_Port, LED2_Pin);
273         HAL_Delay(1000);
274
275         // Action to be customized
276     }
277
278     /* Check direction bit */
279     if ((statusRegister & L6474_STATUS_DIR) == L6474_STATUS_DIR)
280     {
281         // Forward direction is set
282         HAL_GPIO_TogglePin(LED2_GPIO_Port, LED2_Pin);
283         HAL_Delay(1000);
284         // Action to be customized
285     }
286     else
287     {
288         // Backward direction is set
289         HAL_GPIO_TogglePin(LED2_GPIO_Port, LED2_Pin);
290         HAL_Delay(1000);
291         // Action to be customized
292     }
293 }

```

Figure 3.15 : Fonctions d'appel lors une interruption



### 3.3.3 Flux de données :

On trouvera un échange de données entre la tâche **StepperHandle** et les autres tâches, cet échange est assuré par les queues (files d'attentes). (Voir Figure 3.16)

```

void Stepper_motor(void *argument)
{
    /* USER CODE BEGIN Stepper_motor */
    float Flowrate=0, radius = 7;
    float volume_to_inject=0;
    int timeneeded = 0;
    uint8_t mode=0;
    uint16_t laststep;
    /* Infinite loop */
    for (;;) {
        osMessageQueueGet(FlowRateQHandle, &Flowrate, 10U, 100);
        osMessageQueueGet(VolumeQHandle, &volume_to_inject, 10U, 100);
        if ( Flowrate!=0 && volume_to_inject!=0 &&(osMessageQueueGet(ModeQHandle, &mode, 10U,
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_7,GPIO_PIN_RESET );
        timeneeded = Time_Needed(Flowrate, volume_to_inject);
        laststep = timeneeded * (L6474_GetCurrentSpeed(0) / 16); //1/16 microstep
        osMessageQueuePut(LastStepQHandle, &laststep, 1, 100);
        SyringeMove(Flowrate, radius);
    }
    // ***** 0 => StopMode , 8=> PauseMode *****
    if((mode==0 || mode == 8)){
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_7,GPIO_PIN_SET );
        SyringeStop();
    }
    osDelay(10);
}

```

Figure 3.16:Flux de données

On reçoit le débit d'injection à l'aide de la file "FlowRateQHandle", cette queue peut contenir 8 message maximum de type "float".

Le volume à injecter et le rayon de la seringue sont placés respectivement dans "VolumeQHandle" et "RadiusQHandle". Selon ces 3 paramètres la vitesse du moteur est calculée comme il est indiqué dans la partie 3.4.

Il y a évidemment un flux de données sortant de la tâche **StepperHandle** vers les tâches qui ont besoin d'informations concernant le moteur. Par exemple, après calculer le temps total d'injection, on dépose cette information dans une queue "TotalTimeQHandle". Et, on partage le nombre total des pas dans la queue "LastStepQHandle".

### 3.3.4 Calcule

Afin de contrôler le débit d'injection à l'aide d'un moteur pas à pas et un système vis écrou, il faut utiliser la mécanique de fluide.



On commencera avec le terme “**débit volumique**” qui désigne la quantité de liquide qui circule dans une canalisation durant un laps de temps déterminé. Exprimé en litres par seconde (L/s), litres par minute (L/mn) ou en mètres-cubes par heure (m<sup>3</sup>/h). Dans notre cas, le liquide c’est le médicament à injecter et le canal c’est la seringue.

Étant donné que la **viscosité** des médicaments est de même ordre que celle de l’eau, elle influence peu sur le calcul, donc elle va être négligée.

Pour une section d’un canal donnée, plus la vitesse de passage est grande, plus le débit d’écoulement sera important :

$$V = q_v / S$$

**Avec :**

**q<sub>v</sub>** : débit volumique en [m<sup>3</sup>/s]

**v** : vitesse du fluide en [m/s]

Pour calculer la section ( $\pi \cdot r^2$ ) on obtient le rayon de la seringue à travers la tâche “SensorsHandle”. Le débit est tapé par le médecin dans l’écran TFT. La vitesse du fluide ou la vitesse de déplacement est calculée d’après la formule précédente, comme le montre (Voir figure 3.17)

```

337 // returns the speed of Screws (mm/s) needed for a given flow_rate (mm/h) and syringe radius(mm)
338 uint16_t Screws_Speed_From_FlowRate(uint16_t flow_rate , uint8_t radius ){
339     radius = radius*0.001;
340     uint8_t section = radius*radius*3.14159;
341     flow_rate = (flow_rate * 0.001) / 3600;
342     return flow_rate/section ;
343 }
```

Figure 3.17 : Fonction pour calculer la vitesse de déplacement nécessaire

A l’aide d’un système poulie courroie, l’arbre moteur peut transmettre son mouvement de rotation vers un arbre récepteur où le vise sans fin est couplé.

On doit tout d’abord calculer la vitesse de l’arbre récepteur selon la vitesse de déplacement désirée et le pas de la vis. (Figure 3.18)

$$N=v/p$$

Avec :

N : nombre de tours par seconde(tr/s)

V : vitesse de déplacement nécessaire(m/s)

P : le pas de la vis(m)

```

378 // returns the shaft speed needed (rps) to drive the screws
379 float Shaft_speed (float screwspeed){
380     return screwspeed / (SCREWSTEP);
381 }

```

Figure 3.18 : Formule pour calculer la vitesse de l'arbre récepteur

Maintenant on peut calculer la vitesse de l'arbre moteur selon la formule ci-dessous (Figure 3.19)

$$n=D*N/d$$

Avec :

n : Nombre de tours par seconde(tr/s) de l'arbre moteur

d : Diamètre de l'arbre moteur

N : Nombre de tours par seconde(tr/s) de l'arbre récepteur

D : Diamètre de l'arbre récepteur

```

382 // returns the motor speed needed (rps) to drive the shaft
383 float Motor_Speed(float shaftspeed){
384
385     return (DSHAFT*shaftspeed)/DMOTOR;
386 }

```

Figure 3.19: Formule pour calculer la vitesse de l'arbre moteur

Une révolution représente  $360^\circ$ , or qu'un pas complet du moteur (Full Step) représente  $1.8^\circ$  donc le nombre des pas totale pour une seule révolution est  $360^\circ/1.8^\circ=200$  **pas/révolution**. Pour plus de précision et un comportement silencieux on a utilisé un micro-pas 1/16.

La vitesse du moteur dans la bibliothèque est en **pas/seconde** donc la fonction L6474\_SetMaxSpeed prend en paramètre **N\*200\*16. (micro-pas 1/16 )**

Le temps total de l'opération d'injection est égal au rapport **volume/débit**, (Voir figure 3.20).

```

387 //return number of seconds to finish the injection
388 float Time_Needed(float flow_rate, float volume_to_inject){
389     flow_rate = flow_rate/3600;
390     return (volume_to_inject/flow_rate);
391 }
392
393 void SyringeMove(float FlowRate , uint8_t radius){
394     float screwspeed , motorspeed, shaftspeed;
395     int pps;
396     screwspeed = Screws_Speed_From_FlowRate(FlowRate, radius);
397     shaftspeed = Shaft_speed(screwspeed);
398     motorspeed = Motor_Speed(shaftspeed);
399     pps = motorspeed*200*16; // 1/16 microstep
400     L6474_SetMaxSpeed(0, pps);
401     L6474_SetMinSpeed(0, pps);
402     L6474_Run(0, BACKWARD);
403     /*drv8825_setSpeedRPM(drv8825, motorspeed*60);
404     drv8825_setEn(drv8825, EN_START);*/
405 }
406 void SyringeStop(){
407     //drv8825_setEn(drv8825, EN_STOP);
408     L6474_HardStop(0);
409     L6474_HizStop(0);
410 }

```

Figure 3.20 : Fonction pour commander le moteur selon les paramètres calculer

### 3.4 Interface homme machine « IHM »

L'écran tactile résistive (4.3inch 480x272 TFT) va assurer l'interaction entre le médecin et la pousse seringue et vice versa, en fait toutes les données relatives à l'injection sont tapées à travers le médecin. Or que les données d'avancement et de l'état de la seringue ainsi que les alarmes sont affichées à travers l'écran.

#### ❖ LTDC (LCD-TFT display Controller) et contrôleur de touches résistives.

Grace au périphérique LTDC on pourra interfacer l'écran avec la carte STM32, ce périphérique est responsable à transmettre l'ensemble des pixels d'une image sous une format

Bien déterminé (hors de portée de ce rapport due à sa taille). Le LTDC consomme beaucoup de ressources que ce soit au niveau des pins (>40 pins) ou au niveau calculs et traitements. C'est pour cela qu'on a utilisé un accélérateur **DMA2D**, (voir figure 3.19).

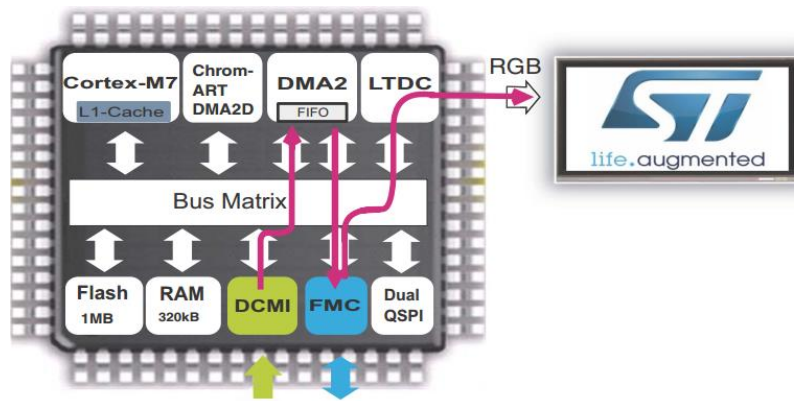


Figure 3.21 : LTDC+DMA2D

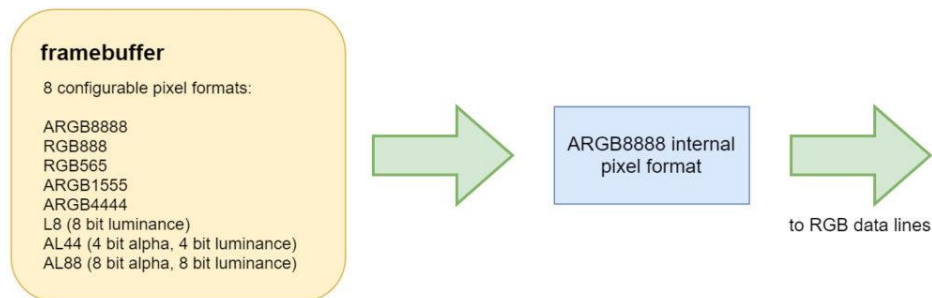


Figure 3.22 : Structure de données passées dans LTDC

Le contrôleur de touche résistive est un système qui va convertir les données analogiques générées par le panneau tactile en des coordonnées x, y puis il les transmet à travers le protocole i2c ou spi pour que le processeur l'interprète. En fait le processeur doit connaître le protocole dont le contrôleur de touche utilise pour transmettre l'information. Dans ce cas un pilote logiciel doit être programmé pour assurer la communication avec le périphérique.

Comme la montre la figure 2.23 ci-dessous.

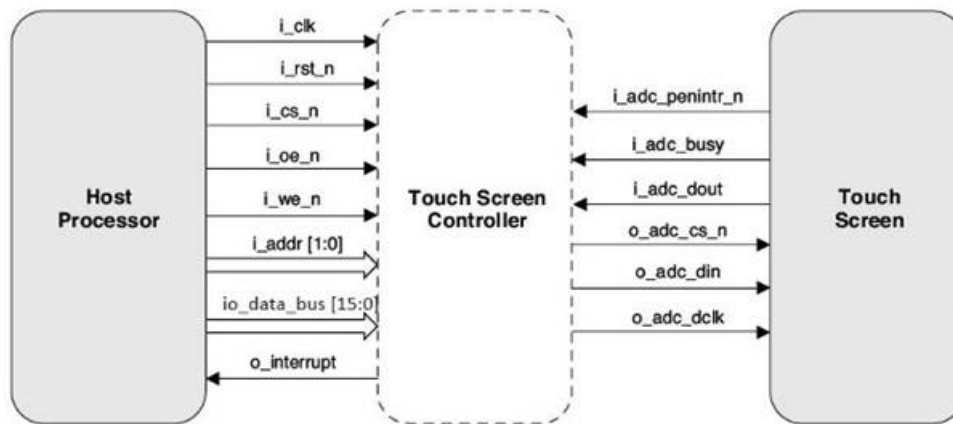


Figure 3.23 : Communication dalle tactile résistive et processeur

### ❖ Architecture (MVP)

On a utilisé le Framework TouchGFX pour développer l'interface homme machine.

, en fait grâce à la méthode drag and drop, on peut faire le design de l'ensemble des vues (view) et puis on génère le code en C++ selon une architecture appelé MVP avec l'approche orienté objets.

#### ↳ Modèle de conception modèle-vue-présentateur

Les interfaces utilisateur TouchGFX suivent un modèle architectural appelé MVP qui est une dérivation du modèle Modèle-Vue-Contrôleur (MVC). Les deux sont largement utilisés pour créer des applications d'interface homme machine.

Dans MVP, les trois classes sont définies comme suit :

- ↳ **Le modèle (Model)** est une interface de données, elle sert également de lien entre la partie non-UI (Backend system) et la partie UI (User Interface) du projet : c'est le cœur de l'interface graphique.
- ↳ **La vue (View)** est une interface passive interface passive qui affiche les données et acquiert les informations de l'utilisateur (via les différents widgets de touchgfx ex : zone de texte, image, bouton, menu déroulant, curseur...)

➤ **Le présentateur (Presenter)** est une classe qui agit sur le modèle et la vue. Elle récupère les données du modèle et les formate pour les afficher dans la vue. Comme la montre (la figure 3.24) ci-dessous.

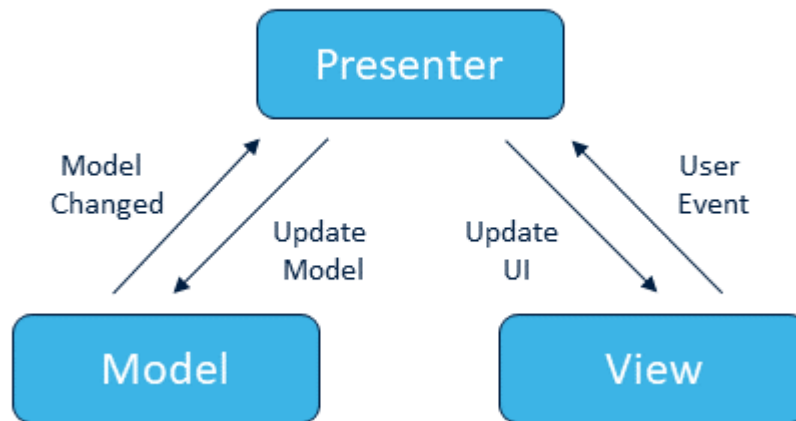


Figure 3.24 : Modèle de conception modèle-vue-présentateur

### ❖ Les interactions dans TouchGFX

Dans TouchGFX Designer, une interaction est constituée d'un trigger et d'une action :

- Un trigger est ce qui va démarrer l'interaction - ce qui doit se passer dans notre application pour que l'action ait lieu.
- Une action est ce qui va se passer après qu'un déclencheur ait été émis.

Un écran vide n'aura que quatre actions disponibles :

- Call new virtual function
- Change screen
- Execute C++ code
- Wait for

### ❖ Flux des données

Dans TouchGFX, la communication avec la partie non-UI de l'application, appelée ici le **backend**, se fait à partir de la classe **Model**. Le système backend dans notre cas est FreeRTOS avec tous les tâches que nous avons parlé précédemment. Du point de vue TouchGFX, cela n'a pas vraiment d'importance, tant qu'il s'agit d'un composant avec lequel il est capable de communiquer. Comme la montre (la figure 3.25) ci-dessous.

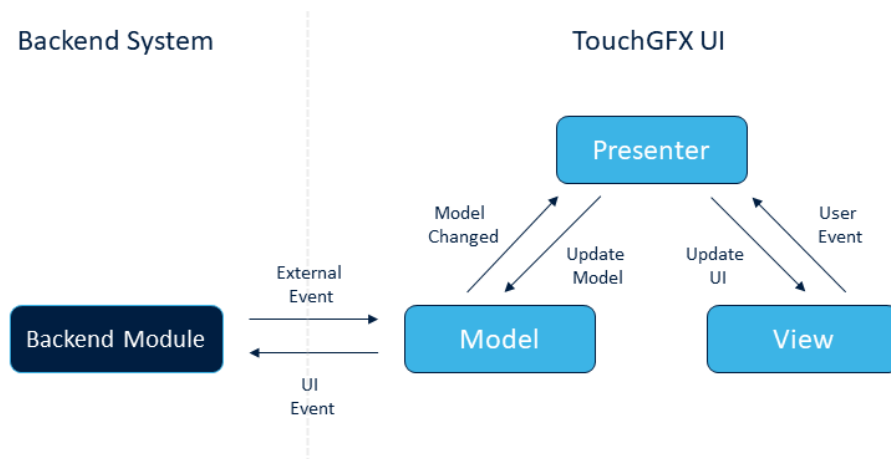


Figure 3.25 : Modèle-Vue-Présentateur et communication externe

Dans notre cas, le protocole de communication entre la partie graphique et le backend est géré à travers les queues (files d'attente) que nous avons déjà utilisées pour la communication inter-tâches.

On prend l'exemple du débit d'injection :

Le Médecin tape sur le bouton “RateBtnBuffer”, et grâce à l'interaction “KeyboardRate” il est dirigé vers une interface clavier pour taper en chiffre le débit. Pour chaque chiffre tapé une interaction eu lieu pour le stocker dans un buffer, (voir figure 3.24).

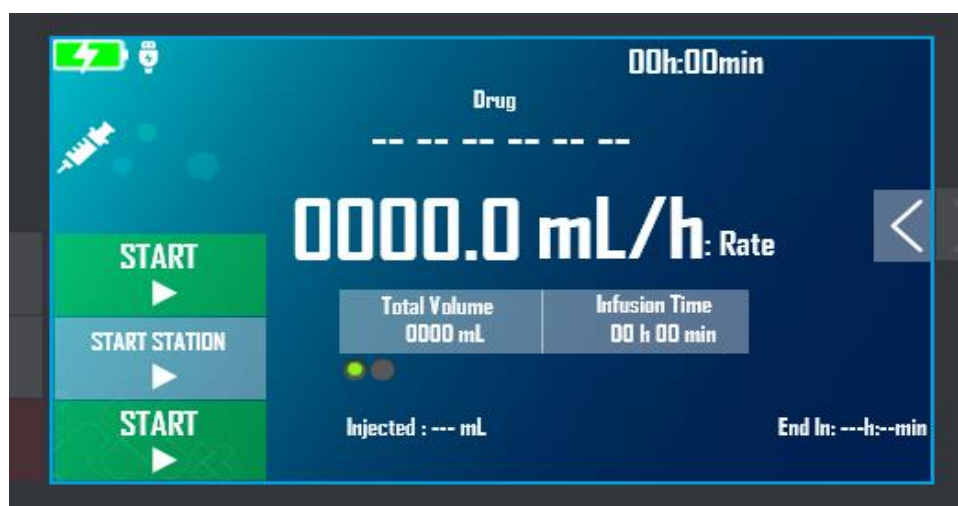


Figure 3.26 : Interface principale

Puis il tape le bouton “Save” qui fait l’appel à une fonction “SaveData” définie dans la classe View de l’interface “KeyboardNb”. Cette fonction passe la valeur du débit à la fonction “saveFlowaRate” définie dans la classe Presenter de la même interface.

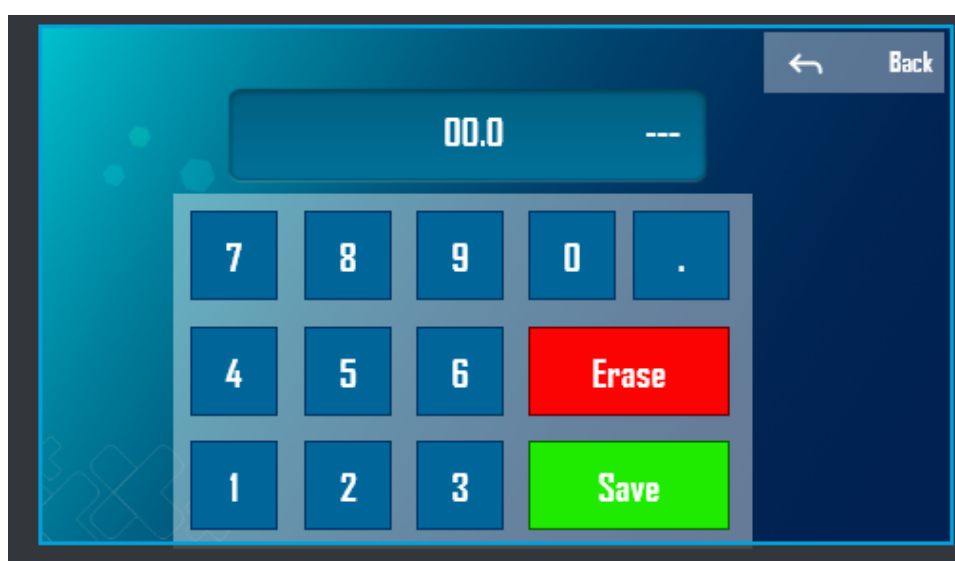


Figure 3.27 : Interface clavier

Enfin cette dernière fait l’appel à la fonction “saveFlowaRate” définie dans la classe model, Comme la montre la figure 3.28 ci-dessous.

```

67
68 void KeyboardNbPresenter::saveFlowaRate(SW_float value)
69 {
70     model->saveFlowaRate(value);
71 }
72

```



```

220
221 /*=====PERFUSION=====*/
222 void Model::saveFlowRate(SW_float value)
223 {
224     float temp = (float)value.BeforeComma;
225     temp = temp + (float)value.AfterComma / 10.0;
226     PerfusionParameters.Flowrate = temp;
227     calculateThirdParameter(CALLER_FLOWRATE);
228     saveInfusionData();
229 }
230

```

Figure 3.28: Structure de clavier

La fonction “saveInfusionData” transmet les données vers le backend à travers une queue “InfusionQHandle” (voir figure 3.27) sous format d’une structure C de type “Infusion\_paramT” qui admet tous les paramètres d’infusion (voir figure 3.29).

```

void Model::saveInfusionData(void)
{
    #ifndef SIMULATOR
    // RTOS
    float prvFlow=0, prvInfuVol=0;
    if((PerfusionParameters.Flowrate!=0 || PerfusionParameters.InfusionVolume!=0)
        && (prvFlow!=PerfusionParameters.Flowrate && prvInfuVol!=PerfusionParameters.InfusionVolume)){
        prvFlow = PerfusionParameters.Flowrate;
        prvInfuVol = PerfusionParameters.InfusionVolume;
        osMessageQueuePut(InfusionQHandle,&PerfusionParameters,1,100);
    }
    #endif
}

```

Figure 3.29 : Transmission des données de l'IHM vers le backend

```

196
197 typedef struct {
198     float Flowrate;
199     float InfusionVolume;
200     float TotalVolume;
201     float KVORate;
202     uint8_t Mode;
203     float Bolus;
204     uint8_t DataChanged;
205 } Infusion_paramT;
206

```

Figure 3.30 : Structure C Infusion\_paramT

Dans le backend, la tâche “IHMHandle” reçoit toutes les données envoyées d’après la queue d’infusion et les transmet vers d’autres files d’attente selon le besoin. Comme la montre (la figure 3.31) ci-dessous.

```

392 /* USER CODE END Header_Interface */
393 void Interface(void *argument)
394 {
395     /* USER CODE BEGIN Interface */
396     Infusion_paramT msgPerfusionParameters;
397     /* Infinite loop */
398     for(;;)
399     {
400         // ***** 0 => StopMode , 8=> PauseMode *****
401         if(osMessageQueueGet(InfusionQHandle,&msgPerfusionParameters,10U,100)==osOK && msgPerfusionParameters
402             && msgPerfusionParameters.Mode!=8 ){
403             osMessageQueuePut(FlowRateQHandle,&msgPerfusionParameters.Flowrate , 1U, 100U);
404             osMessageQueuePut(VolumeQHandle,&msgPerfusionParameters.InfousionVolume , 1U, 100U);
405         }else{
406             osMessageQueuePut(ModeQHandle,&msgPerfusionParameters.Mode , 10U, 100U);
407         }
408     }
409     osDelay(1);
410 }
411 /* USER CODE END Interface */
412 }
413

```

Figure 3.31 : Structure tache IHMHandle

## 3.5 Capteurs et mesures

### 3.5.1 Capteur de diamètre

#### Solution proposée :

Le calcul de débit d'injection, comme il est indiqué précédemment, nécessite une connaissance de la section de la seringue. Afin de calculer ce paramètre il faut connaître le rayon.

$$S = \pi \times r^2$$

Généralement, on a une idée sur les diamètres possibles d'après le type de la seringue utilisée

Le capteur s'agit d'un potentiomètre linéaire ou bien "slide potentiomètre" qui est lié au vérin de la pince à seringue (syringe clamp). Le potentiomètre va retourner une valeur pour chaque niveau du curseur et selon cette valeur on attribue un diamètre qui représente le diamètre interne de la seringue.

Par exemple, on travaille avec les seringues de type Hamilton série 1000 qui admettent ces diamètres possibles :

Model	Volume	Outer Diameter Nominal	Inner Diameter Theoretical Bore
1001	1	9 mm (0.355 in)	4.61 mm (0.181 in)
1001.25	1.25	8.6 mm (0.329 in)	5.15 mm (0.203 in)
1002 (thin wall)	2.5	9.7 mm (0.375 in)	7.29 mm (0.287 in)
1002 (thick wall)	2.5	10.3 mm (0.407 in)	7.29 mm (0.287 in)
1005	5	13.5 mm (0.53 in)	10.3 mm (0.406 in)
1010	10	17.7 mm (0.695 in)	14.57 mm (0.574 in)
1025	25	27.1 mm (1.067 in)	23.03 mm (0.907 in)
1050	50	36.9 mm (1.452 in)	32.54 mm (1.282 in)
1100	100	36.9 mm (1.452 in)	32.54 mm (1.282 in)

Figure 3.32: Diamètres interne et diamètres externe des seringues

Si le capteur indique 30mm de course la variable rayon reçoit 20/2mm et puis on fait calcul de la section.

Electrical Characteristics	
Standard Resistance Range	.....1K ohms to 1 megohm
Standard Resistance Tolerance....	±20 %
End Resistance	
20 mm Travel .....	10 ohms max.
30 mm Travel .....	20 ohms max.
45 mm Travel .....	20 ohms max.
60 mm Travel .....	30 ohms max.
100 mm Travel .....	30 ohms max.

Figure 3.33: Course de potentiomètre à glissière

### Implémentation

Le potentiomètre est alimenté avec une tension 3.3v qui représente sa valeur maximale et la sortie du potentiomètre linéaire est reliée à une entrée ADC (Analogue to digital Converter) du microcontrôleur qui admet une résolution 16 bit, donc les valeurs possibles sont entre 0 (0v) et 65535 (3.3v).

On a activé La fonction de mode continu (continuous mode) qui permet à l'ADC de travailler en arrière-plan. L'ADC convertit les canaux en continu sans aucune intervention du CPU.

## 3.5.2 Capteur de position

### Solution proposée

Avec le même principe On a utilisé un potentiomètre rotatif avec engrené avec le vice sans fine pour qu'il puisse suivre la rotation de l'arbre moteur il s'agit d'un feedback pour contrôler l'erreur du moteur et augmenter sa performance. D'après les valeurs retournées par le potentiomètre on peut aussi conclure la position de la seringue ainsi que le volume restant.

### Implémentation

Le potentiomètre rotatif est interfacé de la même manière que celui utilisé pour capturer le diamètre. La seule chose qui se diffère est la partie de l'interprétation des valeurs retournées par le capteur (voir figure 3.34). En fait, pour détecter la position de la seringue on doit mapper la position zéro et le nombre maximale des pas avec la valeur minimale et la valeur maximale du potentiomètre rotatif

```
traveled_steps= position();
volumeleft=calculate_volume_left(traveled_steps,Flowrate,volume_to_inject); //mm^3
timeleft=volumeleft/Flowrate; // seconds
osMessageQueuePut(VolumeLeftQHandle, &volumeleft, 1, 100);
osMessageQueuePut(TimeQHandle, &timeleft, 1, 100);
if(traveled_steps>=laststep || volumeleft<=0 || timeleft <=0)
    osMessageQueuePut(ModeQHandle,0, 10U, 100U); // ***** 0 => StopMode , 8=> PauseMode *****
```

*Figure 3.34:Capteur de position*

### 3.5.3 Capteur de température

Avec toutes les tâches que notre carte (Stm32H7) va gérer, il est probable qu'elle surchauffe pour une longue durée d'utilisation, même si elle peut supporter jusqu'à 140°C avec une fréquence de 480MHZ.

Mais pour se protéger et rester dans la zone hors stress, on a utilisé le capteur de température interne qui est connecté directement à l'ADC3. La conversion analogique numérique est faite dans le background avec le mode continu (810.5 cycles) et le **watchdog analogique** qui génère une interruption si la conversion est or la marge spécifiée, (voir figure 3.35).

```

/** Configure Analog WatchDog 1
 */
AnalogWDGConfig.WatchdogNumber = ADC_ANALOGWATCHDOG_1;
AnalogWDGConfig.WatchdogMode = ADC_ANALOGWATCHDOG_SINGLE_REG;
AnalogWDGConfig.Channel = ADC_CHANNEL_TEMPSENSOR;
AnalogWDGConfig.ITMode = ENABLE;
AnalogWDGConfig.HighThreshold = 15896;
AnalogWDGConfig.LowThreshold = 10881;
if (HAL_ADC_AnalogWDGConfig(&hadc3, &AnalogWDGConfig) != HAL_OK)
{
    Error_Handler();
}

```

Figure 3.35 : Configuration de l'analogue Watchdog 1

La limite supérieur ( HighThreshold ) et la limite inférieur (LowThreshold ) sont calculées d'après la formule suivante :

$$t_{\text{Celsius}} = (110 - 30) * (\text{readValue} - *TEMPSENSOR\_CAL1\_ADDR) / (*TEMPSENSOR\_CAL2\_ADDR - *TEMPSENSOR\_CAL1\_ADDR) + 30$$

- **tcelsius** : la température en degré Celsius.
- **readValue** : le résultat de conversion analogique numérique.
- **TEMPSENSOR\_CAL1\_ADDR** : contenant l'adresse de la valeur de calibrage pour la température 110°C (stocké dans la mémoire morte de la carte).
- **TEMPSENSOR\_CAL2\_ADDR** : contenant l'adresse de la valeur de calibrage pour la température 30°C (stocké dans la mémoire morte de la carte).

L'interruption est appelée quand le résultat de conversion est en dehors de cette fenêtre :

Max : **15896** qui correspond à **100°C**

Min : **10881** qui correspond à **0°C**

Pour chaque interruption de l'ADC3 on lance une alarme de type "overheating".

```
336 // cpu temp interrupt
337 void HAL_ADC_LevelOutOfWindowCallback(ADC_HandleTypeDef *hadc){
338     // do something in case of analog watchdog interrupts
339     HAL_GPIO_WritePin(LED2_GPIO_Port, LED2_Pin, GPIO_PIN_SET);
340     HAL_ADC_Stop_IT(&hadc);
341 }
342 /** Configure Analog WatchDog 1
343 */
```

Figure 3.36: Fonction d'appel lors d'une interruption wdg analogique

## 3.6 Connectivité

La connectivité est un élément clé dans notre projet, vu qu'elle est rarement utilisée dans les poussettes existantes dans le marché. Son rôle est de permettre le médecin de la surveillance en temps réel de l'état d'injection à distance et n'importe où avec l'aide d'une application mobile.

### 3.6.1 Solution proposée :

Les données de supervision sont stockées et synchronisées en temps réels dans une base de données hébergée dans le cloud puis L'application mobile peut les récupérer, fait le traitement si nécessaire et les affiche au médecin.

L'envoi des données nécessite un accès à l'internet, pour cela on a intégré un module Wi-Fi qui va assurer la communication avec un point d'accès vers le WEB selon le protocole TCP-IP.

### 3.6.2 Architecture et protocoles :

Le module Wi-Fi et la carte se communiquent à l'aide du protocole UART (RX/TX), la communication entre le module Wi-Fi et la base de données hébergée dans le cloud est assurée par le protocole HTTP et l'application mobile utilise le protocole Websockets pour récupérer les données déposées dans le cloud.

### 3.6.3 Implémentation :

La connectivité est gérée par la tâche "ConnectivityHandle", elle reçoit les données déposées par les autres tâches dans les queues puis elles les envoient vers le module Wi-Fi esp8266.

Ci-dessous est la liste des données à envoyer vers le cloud avec leurs queues correspondantes :

- Débit d'injection => FlowRateQHandle
- Temps restant => TimeQHandle
- Volume restant => VolumeLeftQHandle
- Alarmes => envoyées directement auprès de leurs interruptions.
- L'ID de la seringue => il s'agit de l'UID de l'STM32H7

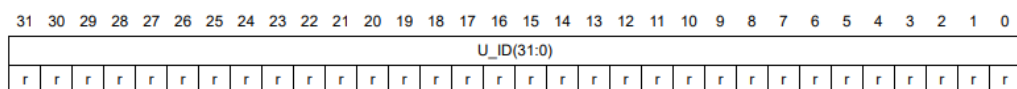
Chaque stm32 admet un UID de 96 bits de longueur et il est propre à chaque pièce manufacturée. On a pensé alors d'utiliser comme ID pour chaque pousse seringue.

Cet identifiant est stocké par le fabricant dans une adresse mémoire flash à lecture seul comme il est indiqué ci-dessous dans la figure 3.37. On peut récupérer les bits à l'aide d'un pointeur à l'adresse mémoire correspondant.

**Base address: 0x1FF1 E800**

**Address offset: 0x00**

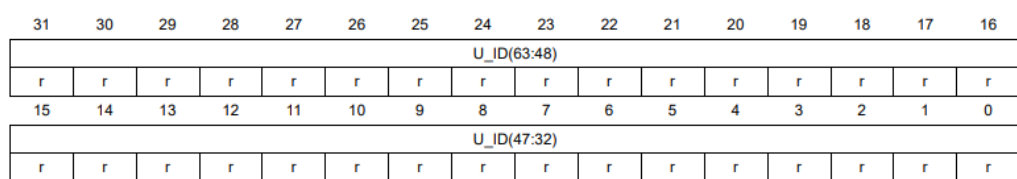
Read only = 0XXXXX XXXX where X is factory-programmed



Bits 31:0 **U\_ID(31:0)**: 31:0 unique ID bits

**Address offset: 0x04**

Read only = 0XXXXX XXXX where X is factory-programmed



Bits 31:0 **U\_ID(63:32)**: 63:32 unique ID bits

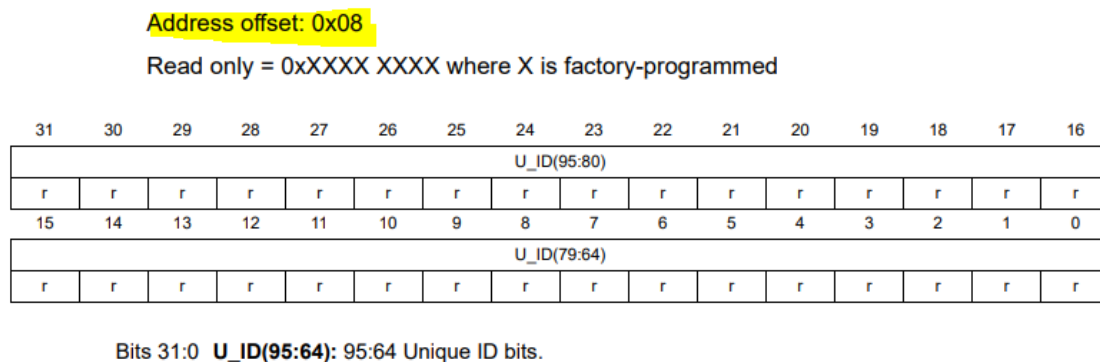


Figure 3.37 : Adresses UID de l'stm32h743I

Après la réception des queues on passe chaque valeur à la fonction “sprintf” qui retourne une chaîne formatée en utilisant les arguments %d, %f, %s, %.3f ....

Les données sont envoyées à travers UART avec une vitesse de transmission 9600 bauds vers le module Wi-Fi selon un format bien déterminé à l'aide de la fonction :

```
“HAL_UART_Transmit UART_HandleTypeDef *huart, const
uint8_t *pData, uint16_t Size, uint32_t Timeout)”
```

En fait, chaque trame doit commencer par une grandeur (v=>volume restant, t=> temps restant, f=> débit d'injection ...) suivie par sa valeur, par exemple “v20” correspond à un volume restant vaut 20ml, or que “f50” correspond à un débit d'injection qui vaut 50 ml/h.

Pour synchroniser la lecture et l'écriture dans le port série, on a utilisé un pin digital qui bascule en état haut à chaque transmission, puis revient à l'état bas à la fin de l'écriture. Elle semble à une horloge qui déclenche un signal au module Wi-Fi pour faire la lecture. (voir figure 3.39).



```

/* USER CODE END Header_Cloud_Connectivity */
void Cloud_Connectivity(void *argument)
{
    /* USER CODE BEGIN Cloud_Connectivity */
    HAL_GPIO_WritePin(GPIOC, GPIO_PIN_14, RESET); // UART clock
    float Flowrate=0 , Timeleft=0, Volumeleft=0;
    char flowbuff[10], timebuff[10] , volumebuff[10];

    /****** UNIQUE ID *****/
    uint32_t (*unique_id_1) = (uint32_t*)(0xFF1E800); // BASE address (reference manual stm32h743)
    uint32_t (*unique_id_2) = (uint32_t*)(0xFF1E804); // BASE address + 0x04 offset
    uint32_t (*unique_id_3) = (uint32_t*)(0xFF1E808); // BASE address + 0x08 offset
    char Id[85];
    int n =sprintf(Id,"%lu%lu%lu",*unique_id_1,*unique_id_2,*unique_id_3);
    HAL_GPIO_WritePin(GPIOC, GPIO_PIN_14, SET);
    HAL_UART_Transmit(&huart3,Id ,n , 100);
    osDelay(10);
    HAL_GPIO_WritePin(GPIOC, GPIO_PIN_14, RESET);
    /* Infinite loop */
    // ***** f==> flowrate t==> timeleft v==>volumeleft *****
    for(;;)
    {
        if(osMessageQueueGet(FlowRateQHandle,&Flowrate , 1U, 100U)==OSOK){
            int nflow =sprintf((uint8_t *)flowbuff,"f%.3f",Flowrate);
            HAL_GPIO_WritePin(GPIOC, GPIO_PIN_14, SET);
            HAL_UART_Transmit(&huart3, (uint8_t *)flowbuff, nflow, 10);
            HAL_GPIO_WritePin(GPIOC, GPIO_PIN_14, RESET);
        }
        if(osMessageQueueGet(TimeQHandle,&Timeleft , 1U, 100U)==OSOK){
            int ntime =sprintf((uint8_t *)timebuff,"t%.3f",Timeleft);
            HAL_GPIO_WritePin(GPIOC, GPIO_PIN_14, SET);
            HAL_UART_Transmit(&huart3, (uint8_t *)timebuff, ntime, 10);
            HAL_GPIO_WritePin(GPIOC, GPIO_PIN_14, RESET);
        }
        if(osMessageQueueGet(VolumeLeftQHandle,&Volumeleft , 1, 100U)==OSOK){
            int nvol =sprintf((uint8_t *)volumebuff,"v%.3f",Volumeleft);
            HAL_GPIO_WritePin(GPIOC, GPIO_PIN_14, SET);
            HAL_UART_Transmit(&huart3, (uint8_t *)volumebuff, nvol, 10);
            HAL_GPIO_WritePin(GPIOC, GPIO_PIN_14, RESET);
        }
    }
}

```

Figure 3.39 : Structure de la tâche ConnectivityHandle

Dans l'autre côté, le module esp8266 admet son propre code pour gérer la communication **TCP-IP**, il est indépendant au celle de l'Arm32 pour à la fois réduire le stress à notre carte principale et aussi assurer la sécurité de la pousse seringue en éliminant la partie réseau au reste du système. Chaque fois le pin **D1** est en état haut l'esp8266 lit les données déposées dans le port série à travers le pin **Rx**, les interprète puis les redirige vers le cloud. Comme la montre (la figure 3.37) ci-dessous

En fait la solution cloud choisie est "**Firestore Realtime Database**" elle est gratuite et efficace au moins pour nos besoins actuels. On pourrait migrer facilement vers GCP (Google

Cloud Platform) pour garantir la disponibilité et la performance maximale puisque les services sont payants.

Pour gérer la communication entre Firebase et le module Wi-Fi, on utilise une bibliothèque open source disponible en **GitHub**, développé par “**mobizt**”. Elle s’agit d’un ensemble des fonctions qui utilisent les requêtes HTTP (**GET – POST – DELETE ...**) pour déposer les données dans **RTDB**. Comme la montre (la figure 3.40) ci-dessous.

```
92 void loop()
93 {
94   if (Serial.available() && (millis() - dataMillis) > 100 && digitalRead(5)==HIGH)
95   {
96     dataMillis = millis();
97     char inChar = Serial.read(); // check first letter and assigned to a corresponding value
98     switch (inChar) {
99       case 'v' :
100       {
101         float vol = Serial.parseFloat();
102         Serial.printf("a %s %f\n", Firebase.RTDB.setFloat(fbdo, "/Volume_Left", vol) ? "ok" : fbdo.errorReason().c_str(), vol);
103         break;
104       }
105       case 'f':
106       {
107         float flow = Serial.parseFloat();
108         Serial.printf("b %s %f\n", Firebase.RTDB.setFloat(fbdo, "/Flow_Rate", flow) ? "ok" : fbdo.errorReason().c_str(), flow);
109         break;
110       }
111       case 't':
112       {
113       }
114     }
```

Figure 3.40 : Structure du code Arduino pour Esp8266

## 3.7 Stockage en local

Les exigences et les normes internationaux exigent un périphérique de stockage de données en local. Ces données représentent l’état d’injection (débit, volume, temps, nature de médicament...) De chaque patient ainsi que ses informations générales tel que le nom, le prénom, l’âge, le poids....

Les données stockées doivent être enregistrer pendant minimum 10 ans.

### Implémentation :

Nous avons choisi l’interface SDMMC intégrée dans l’STM32H7 il fournit une interface de communication pour que le microcontrôleur puisse communiquer avec les Multi-MediaCards, les cartes mémoire SD et les périphériques SDIO. Avec l’interface SDMMC, on pourrait facilement gérer les opérations de lecture et d’écriture à grande vitesse dans les mémoires Flash externes. (Voir figure ci-dessous 3.41).

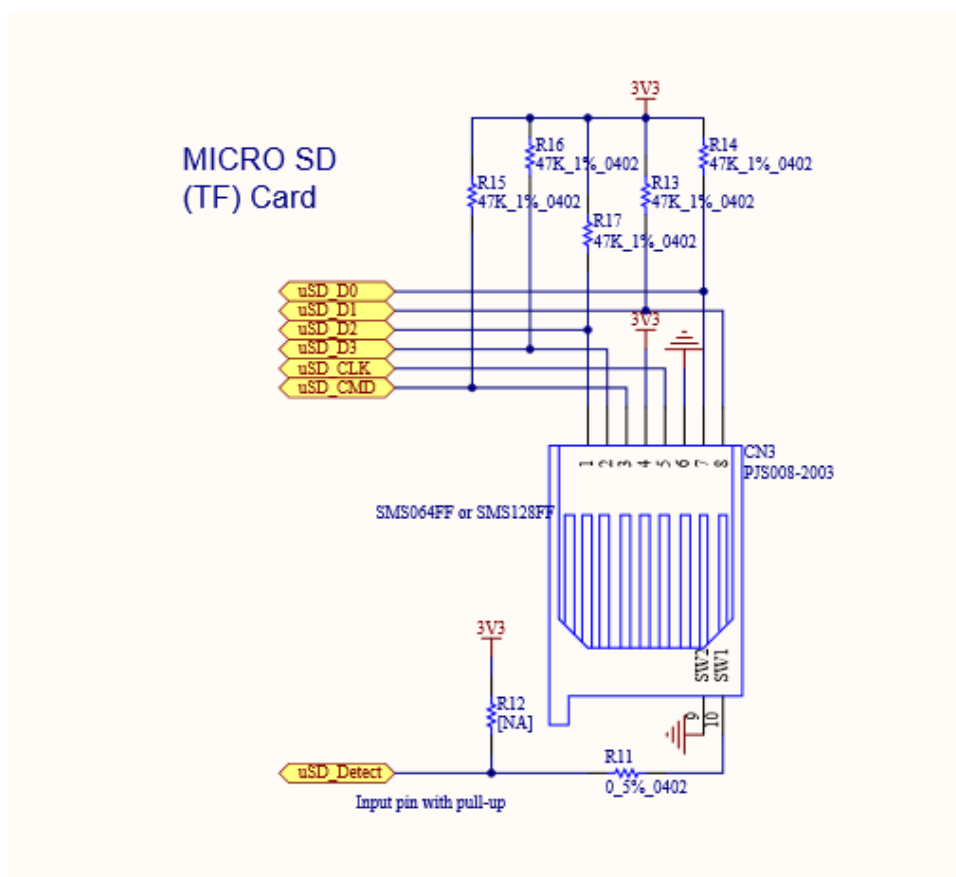


Figure 3.41 : Interface SDMMC

Pour gérer les fichiers de chaque patient dans la carte SD, Il faut utiliser un système de gestion de fichier. On a choisi FAT qui est un standard de l'industrie et il est supporté par STM32. En fait, STM32 possède une bibliothèque générique appelé FATFS que nous avons l'utilisé pour gérer les fichiers dans la carte mémoire tel que l'ouverture, la fermeture, l'écriture et la lecture des fichiers. (Figure ci-dessous3.42)

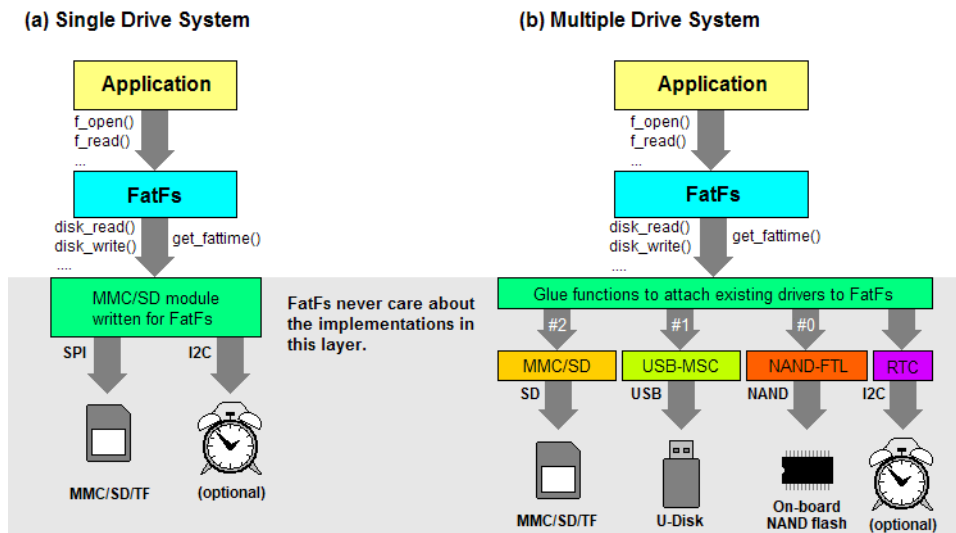


Figure 3.42: FATFS

Dans la figure ci-dessous, on récupère les données d'infusion (déposées dans la queue "InfusionQHandle", puis on ouvre le fichier "SWlog" et on écrit le texte dont on a formaté à travers la fonction "sprintf" enfin nous fermons le fichier pour consommer moins de mémoire.

```
for (;;) {
    osMessageQueueGet(InfusionQHandle, &msgPerfusionParameters, 1U, 100U);
    sprintf((uint8_t *)wtext, "Flow Rate = %d", msgPerfusionParameters.Flowrate);
    if(f_mount(&SDFatFS, (TCHAR const*)SDPath, 0) != FR_OK)
    {
        Error_Handler();
    }
    else
    {
        if(f_mkfs((TCHAR const*)SDPath, FM_ANY, 0, rtext, sizeof(rtext)) != FR_OK)
        {
            Error_Handler();
        }
        else
        {
            //Open file for writing (Create)
            if(f_open(&SDFile, "SWlog.TXT", FA_CREATE_ALWAYS | FA_WRITE) != FR_OK)
            {
                Error_Handler();
            }
            else
            {
                //Write to the text file
                res = f_write(&SDFile, wtext, strlen((char *)wtext), (void *)&byteswritten);
                if((byteswritten == 0) || (res != FR_OK))
                {
                    Error_Handler();
                }
                else
                {
                    f_close(&SDFile);
                }
            }
        }
    }
}
```

Figure 3.43 : TACHE STORAGE HANDLE

### 3.8 Taches en cours de développement

Plusieurs taches sont encore de développement, soit à cause de non disponibilité des composants matériels ou à des contraintes de temps.

- ↳ Gestion d'alimentation : La gestion d'alimentation est une tâche très critique. En fait, on a passé la phase de conception de cette tache mais il reste encore la phase de développement.
- ↳ Clavier à Membrane : Nous sommes en attente du fournisseur qui est en train de faire le design du clavier
- ↳ Capteur de pression : Ce capteur est en cours de livraison. Il est utilisé pour contrôler la pression du sang du patient
- ↳ Application mobile : Une autre équipe est en train de développer l'application mobile qui sert à récupérer les données qu'on a déposées dans le cloud.

### 3.9 Conclusion

Dans ce chapitre on a détaillé la phase de développement des différentes taches passant par le moteur pas à pas et l'interface homme machine jusqu'à la tache de connectivité. En dernier lieu, nous avons indiqué quelques tâches en cours d'élaboration.

## Conclusion générale :

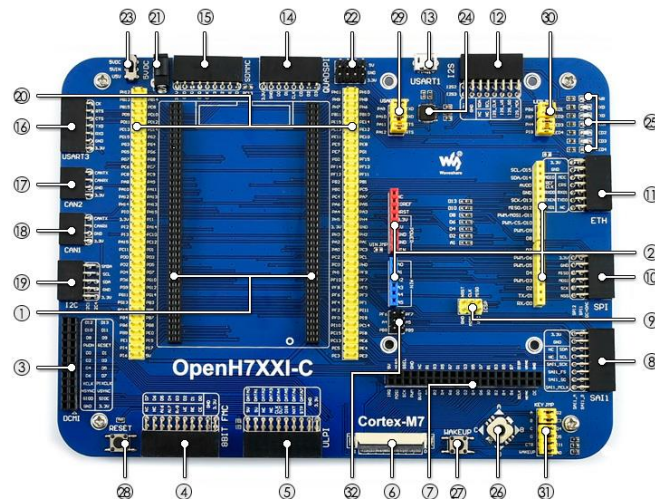
Grace à ce stage de fin d'étude au sein la société "3DWave", nous avons découvert le monde professionnel, nous avons bien utilisé les compétences acquises pendant les Cinq semestres de la licence TIC spécialité IOT et nous avons acquis d'autre compétences comme les systèmes temps réels, le graphique pour l'embarquer et le développement des processeur ARM grâce au microcontrôleur STM32H7 basé au processeur CORTEX M7.

Le développement des dispositifs médicaux nous a permis de découvrir le monde de la santé ainsi que les différents exigences et normes internationales pour réaliser un tel projet.

Notre travail est encore d'actualité et ne s'arrête pas à ce niveau. En effet l'entreprise d'accueil nous a proposé un contrat temps partiel pour compléter le développement de ce projet et passer à la phase de la production puis la commercialisation.

Finalement, Nous souhaitons très fortement que le projet soit le fruit du progrès, de l'évolution et qu'il reste à la hauteur des exigences de la société et de l'ISIMM, nous souhaitons par ailleurs la satisfaction de l'équipe 3Dwave, ainsi que les membres de jury.

# Annexe 1



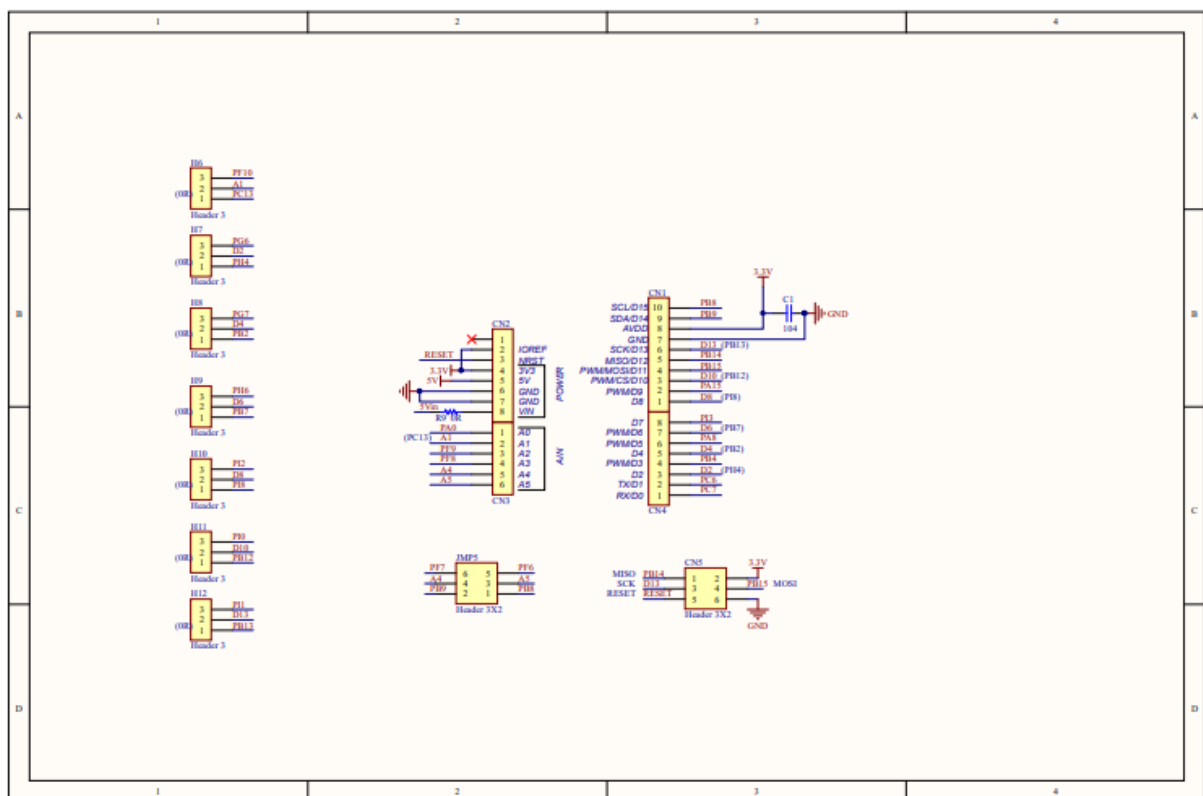
N°	Nom de composant	Rôle
1	Connecteur de la carte principale du MCU	Connexion facile du CoreH743I
2	Interface Arduino	Connexion des boucliers Arduino
3	Interface DCMI	Connecteur de camera
4	Interface 8-bit FMC	Connecte facilement à des périphériques tels que NAND Flash
5	Interface ULPI interface	Connexion d'un périphérique USB haute vitesse (le STM32H743I intègre un contrôleur HS USB sans dispositif PHY)
6	Interface LCD 1	Connexion LCD 10,1 pouces, LCD 7 pouces, LCD 4,3 pouces
7	Interface LCD 2	Connexion de l'écran LCD 4,3 pouces
8	Interface SAI1	Connexion de modules audio
9	Interface ICSP	Arduino ICSP
10	Interfaces SPI	<ul style="list-style-type: none"> <li>✓ Connecte facilement aux périphériques SPI tels que Data Flash (AT45DBxx, W25QXX), carte SD, module MP3, etc.</li> <li>✓ Connecte facilement aux modules AD/DA</li> </ul>

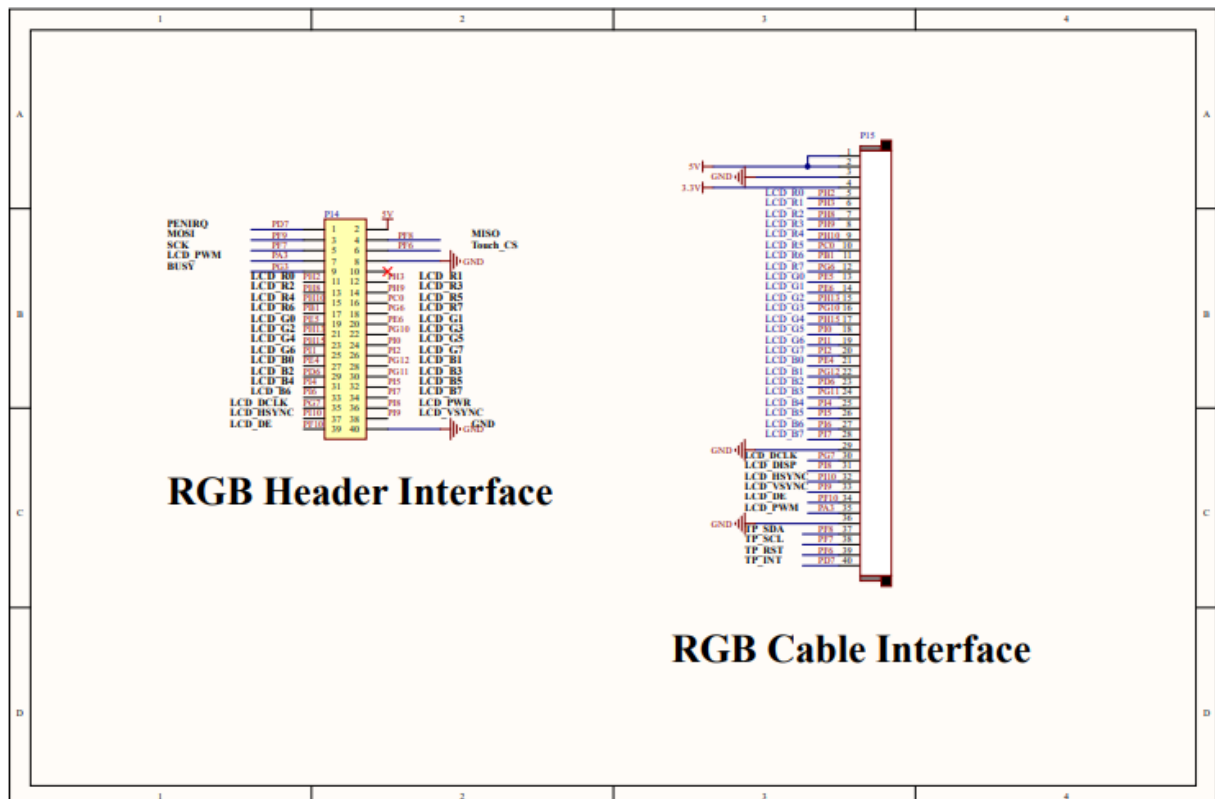
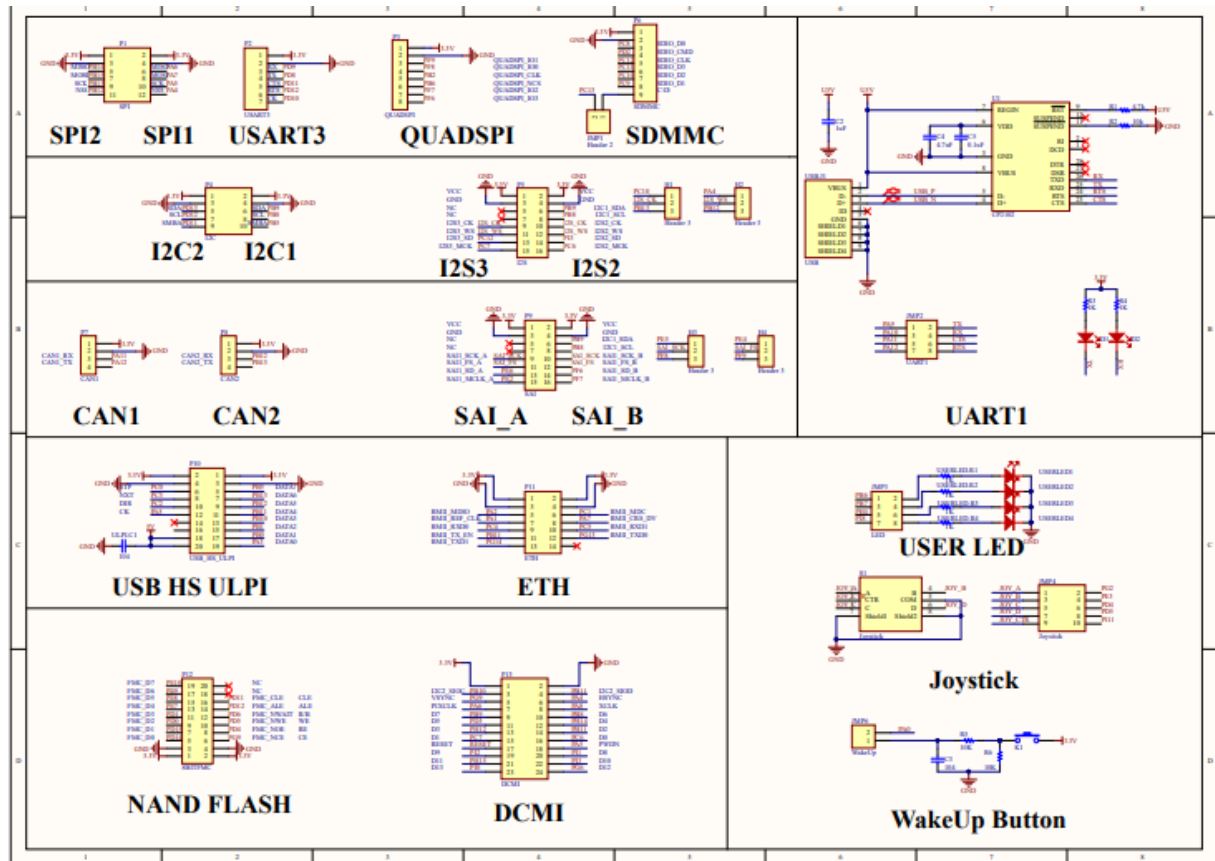
		(fonctionnalités de SPI1 AD/DA
11	Interface Ethernet	Connections Ethernet modules
12	Interface I2S / I2C	Connecte facilement aux périphériques I2S tels que les modules audios, etc.
13	Connecteur USART1	USB vers UASRT via le convertisseur embarqué CP2102
14	Interface QUADSPI	Interface SPI 4 fils (la dernière interface périphérique de la série H7), pour connecter des modules Flash série comme la carte W25QXX
15	Interface SDMMC	Module de connexion Micro SD, dispose d'une vitesse d'accès beaucoup plus rapide que SPI
16	Interface USART3	Connection facile to RS232, RS485, USB TO 232, etc.
17	Interface CAN2	Connections CAN modules
18	Interface CAN1	Connections CAN modules
19	Interface I2C1/I2C4	Connection facilement aux périphériques I2C tels que l'extension E/S (PCF8574), l'EEPROM (AT24Cxx), le capteur 10 DOF IMU, etc.
20	Connecteur MCU pins	Tous les ports d'E/S du MCU sont accessibles sur les connecteurs d'expansion pour une expansion ultérieure
21	5V DC jack	
22	Entrée/sortie de puissance 5 V/3,3 V	Habituellement utilisé comme sortie de puissance, également la mise à la terre avec d'autres cartes utilisateur
23	Commutateur d'alimentation électrique	Alimenté par 5 VCC OU une connexion USB de l'USART1
24	CP2102	Convertisseur USB to UART
25	LEDs	Pratique pour indiquer l'état E/S et/ou l'état d'exécution du programme
26	Manette	5 positions
27	BOUTON DE RÉVEIL	Utilisé comme bouton régulier, et/ou réveiller le MCU STM32 de sommeil
28	Button reset	
29	USART1 jumper	

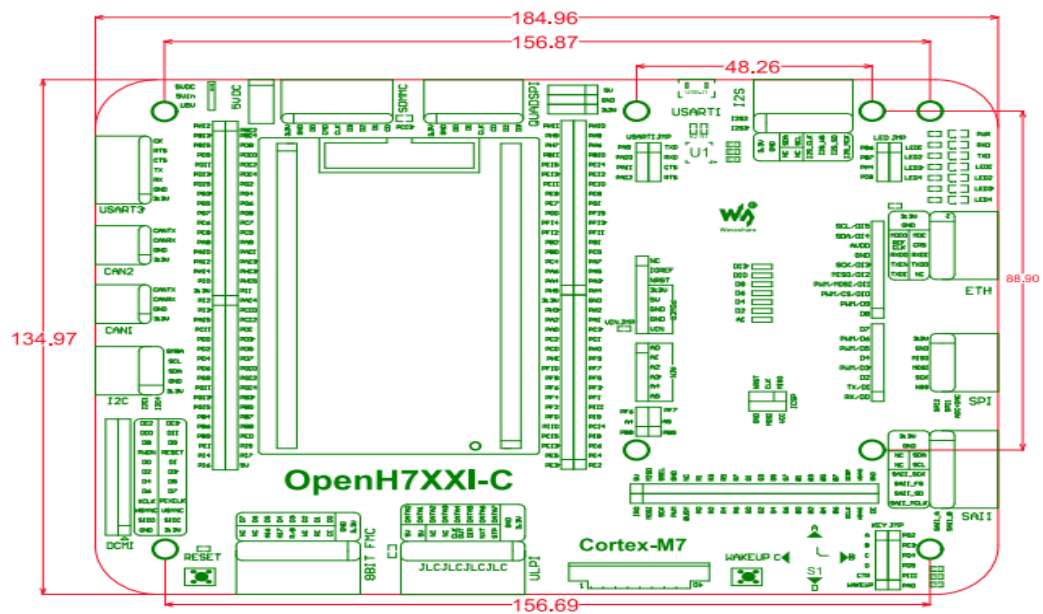
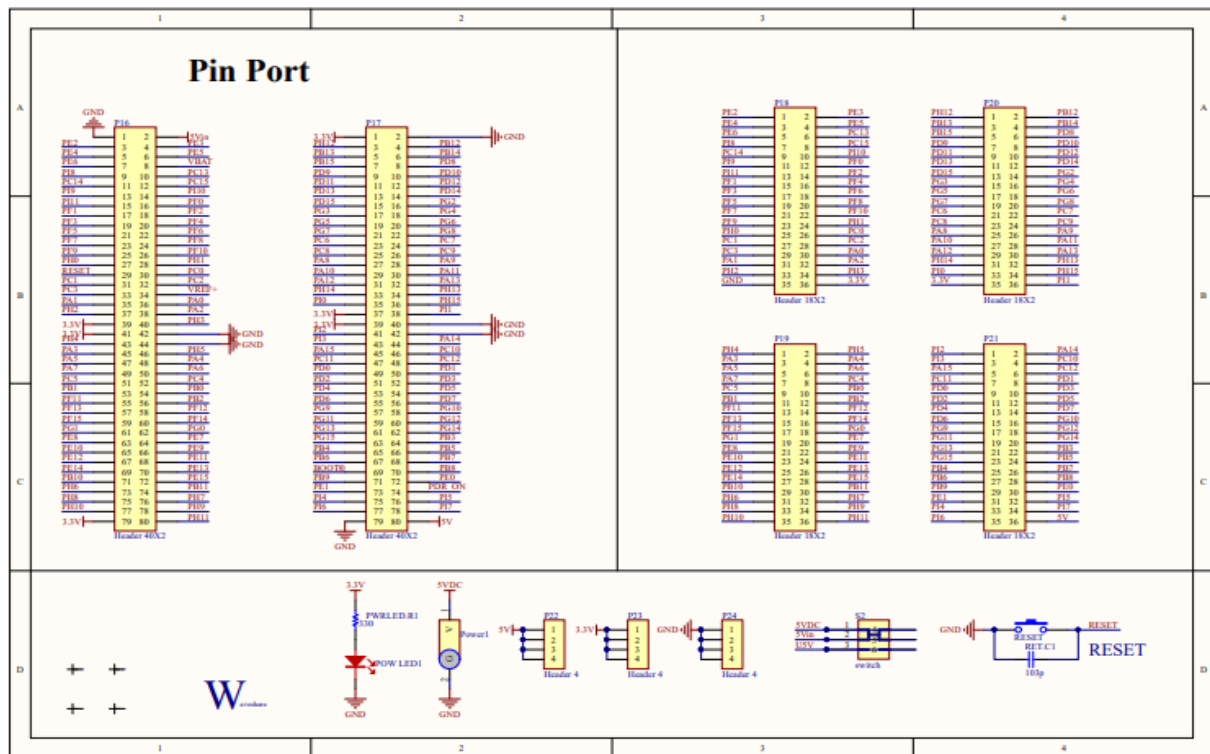


30	LED jumper	<ul style="list-style-type: none"> <li>✓ Court-circuiter le cavalier pour se connecter aux E/S par défaut utilisées dans l'exemple de code</li> <li>✓ Ouvrir le cavalier pour le connecter à des E/S personnalisés via les fils de cavalier</li> </ul>
31	KEY jumper	<ul style="list-style-type: none"> <li>✓ Court-circuiter le cavalier pour se connecter aux E/S par défaut utilisées dans l'exemple de code</li> <li>✓ Ouvrir le cavalier pour le connecter à des E/S personnalisés via les fils de cavalier</li> </ul>
32	Arduino jumper	<ul style="list-style-type: none"> <li>✓ Court-circuiter les broches supérieures, A4, A5 est utilisé comme fonction AD</li> <li>✓ Court les broches inférieures, A4, A5 est utilisé comme fonction I2C</li> </ul>

Tableau 4.1: Les différents composants de OpenH7







**Unit: mm**

Figure 4.2 : Datasheet de l'OPENH7XXI-C

# Bibliographie

- [1] <https://fiches-de-soins.eu/content/post.php?id=pousse-seringue-electrique-pse>
- [2] <https://ispdz.forumactif.org/t602-pousse-seringues>
- [3] <https://www.infirmiers.com/etudiants-en-ifsu/cours/les-pousse-seringue-electriques.html>
- [4] <http://dSPACE.univtlemcen.dz/bitstream/112/10721/1/Ms.EBM.Mokeddem%2BBachra.pdf>
- [5] <https://www.lothen.org/Synthese.php?Requete=10518>
- [6] [https://fr.wikipedia.org/wiki/Serial\\_Peripheral\\_Interface](https://fr.wikipedia.org/wiki/Serial_Peripheral_Interface)