République Tunisienne
*****
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
*****
Université de Monastir
*****

N° d'ordre :

Mémoire de Projet de Fin d'Etudes

Présenté en vue de l'obtention du

Diplôme Nationnal de Licence en Tecnologie d'Informations et de Communication

Spécialité :

Internet of Things (IoT)

par

…Nom et prénom de l'étudiant…

……Titre du projet……

Soutenu le xxxxxxx devant le jury composé de :

M./Mme : ……..	Président
M./Mme : ……..	Rapporteur
M./Mme : ……..	Encadrant Pédagogique
M./Mme : ……..	Encadrant Peofessionnel

Résumé

Mots clés :

Abstract

Table des matières

Mémoire de Projet 1
de Fin d'Etudes 1
Introduction Générale 9
Chapitre 1 : Contexte générale de projet 11
1.1 Introduction 11
1.2 Présentation de la société d'accueil : MEDIWAVE 11
1.3 Différents services 11
1.4 Cadre générale 12
1.4.1 Définition de la pousse seringue électrique 12
1.4.2 Le principe de fonctionnement 13
1.4.3 Les différents types des Pousse seringues électriques 14
1.4.4 Les Différents modes d'administration des Pousse Seringues connectés 14
1.4.5 Applications médicales des Pousse seringues électriques 15
3-6 Avantages et inconvénient d'un pousse seringues connectés 16
1.5 Conclusion 16
Chapitre 2 : Exigences et méthodologie de travail 18
2.1 Introduction 18
2.2 Norme IEC-62 304 18
2.2.1 Exigence générale 18
2.2.2 Processus de développement logiciel 19
2.3 Méthodologie de travail git (push pull commit) 19
2.3.1 Problématique 19
2.3.2 Solution 19
2.3.3 Implémentation 20
2.4 Conclusion 21
Chapitre 3 : Analyse et spécification de besoins 22
3.1 Introduction 22
3.2 Spécification des besoins 22
3.2.1 Expression du besoin 22
3.2.2 Cahier de charge 22
3.3 Architecture globale « Software –Hardware 24
3.3.1 Architecture Hardware : 24
3.3.2 Architecture Software : 34

3.4 Conclusion 35
Chapitre 4 : Conception et réalisation 36
4.1 Introduction 36
4.2 OS kernel 36
4.2.1 Middleware (FreeRTOS) 36
4.2.2Tâches Et Queues (File d'attente) 37
4.3 Moteur pas à pas 42
4.3.1 L6474 Driver / L6474.C : 42
4.3.2 Interruptions 46
4.3.3 Flux de données : 48
4.3.4 Calcule 48
4.4 Interface homme machine « IHM » 50
4.5 Capteurs et mesures 56
4.5.1 Capteur de diamètre 56
4.5.2 Capteur de position 57
4.5.3 Capteur de température 57
4.5.4 Capteur de pression 59
4.6 Connectivité 59
4.6.1 Solution proposée : 59
4.6.2 Architecture et protocoles : 59
4.6.3 Implémentation : 60
4.7 Taches en cours de développement 63
4.8 Conclusion 63

Liste des figures

Figure 1.1 : modèle de pousse seringue 13
Figure 2.2 : Structure du projet sur GitHub ..... **Erreur ! Signet non défini.**
Figure 3.1 : méthodologie de travail 20
Figure 4 : Architecture "Pousse seringue" ..... **Erreur ! Signet non défini.**
Figure 5: Moteur PAS à PAS (NEMAI17) 28
Figure 6 : stepper motor driver shield L6474 29
Figure 7 : LCD TFT 4.3 30
Figure 8 : Node MCU ESP8266 30
Figure 9 : Capteur de position (3590S-2-103L) 31
Figure 10 : Capteur de diamètre (PTL01-15W0-103B1) 31
Figure 11 : Capteur de proximité (OPB745WZ) 32
Figure 12 : Capteur jauge de contrainte (FSS020WNST) 32
Figure 13 : Batterie (RRC2054) 33
Figure 14 : RRC-PMM240 33
Figure 15: Buzzer (458-1402-ND) 34
Figure 16 : Structure du code 37
Figure 17: Priorités et tailles de chaque tache 38
Figure 18 : Création des taches ..... **Erreur ! Signet non défini.**
Figure 19 : h 40
Figure 20 : file d'attente et mode FIFO 41
Figure 21 : création d'une file d'attente (queue) 41
Figure 22 : définition des fonctions de lecture et écriture dans une queue .. **Erreur ! Signet non défini.**
Figure 23: schéma de câblage du driver L6474 42
Figure 24 : structure du registre STEP_MODE 43
Figure 25 : adresse des différents registres 44
Figure 27 : fichier de paramètre par défauts 45
Figure 26 : structure C L6474_Init_t 45
Figure 28 : les fonctions pour commander le moteur 46
Figure 29 :structure du registre STATUS 47
Figure 30 : Fonctions d'appel lors une interruption 47
Figure 31 : fonction pour calculer la vitesse de déplacement nécessaire 49
Figure 32 : fonction pour calculer la vitesse du moteur 49
Figure 33 :fonction pour commander le moteur selon les paramètres calculer 50
Figure 34 : Modèle de conception modèle-vue-présentateur 52
Figure 35 : Modèle-Vue-Présentateur et communication externe 53
Figure 36 :configuration Watchdog analogique 58
Figure 37:fonction d'appel lors d'une interruption wdg analogique 59
Figure 38 : structure du code Arduino pour Esp8266 63

Liste des tableaux

Aucune entrée de table d'illustration n'a été trouvée.

Introduction Générale :

.

.

.

.

.

.

.

.

.

.

.

.

.

.

Afin de bien introduire et expliquer le sujet, nous avons d´écrit 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, les problèmes soulevés et les solutions suggérées. Le deuxième chapitre tentera de présenter tous les concepts en détail ce qui aidera `a d´développer notre projet. Le troisième chapitre portera sur l'analyse et la description des besoins, et le quatrième chapitre sera consacrée `a la description globale et d´détaillée de la réalisation du projet et de nos solutions. Finalement, notre rapport s'achèvera par une conclusion générale regroupant les principaux résultats trouvés et par quelques perspectives.

Chapitre 1 : Contexte générale de 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 d´écrivons la problématique en spécifiant les besoins qui nous ont incités à réaliser note projet. Ensuite, nous analysons les solutions existantes afin d'identifier leurs imperfections. Enfin, nous proposons nos solutions envisagées d'une manière claire et méthodique.

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. A ce propos le secteur industriel envisage une grande croissance grâce aux progrès technologique 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 su profiter des bienfaits de la technologie. Elle a été mise en exploitation depuis MAI 2020.elle est située à Sousse sous la direction générale de **Mr 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 logicielle 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**

Cette partie on globe l'étude, la conception et la construction de machines industrielles 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 moulent 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, ICC, 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 effectue en électronique, automatisme, mécanique, hydraulique, pneumatique et contrat de maintenance clés en main.

1.4 Cadre générale

1.4.1 Définition de la pousse seringue électrique

Un pousse-seringue ou seringue auto pulsée (SAP) ou encore pousse-seringue électrique (PSE) est un dispositif médical de classe IIb utilisé pour administrer de faibles quantités de fluide (avec ou sans médicament) à un patient à travers une seringue allant de 1cc jusqu'à un volume de 100cc. On le retrouve également en chimie ou en recherche biomédicale. On les retrouve majoritairement dans les services de soins des Centres Hospitaliers. Facile d'utilisation, 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.

Un pousse seringue électrique (PSE) est par définition un appareil qui permet d'obtenir un débit constant d'un médicament avec une vitesse déterminée. La vitesse de déplacement du piston de la seringue varie selon une programmation définie par l'utilisateur. On peut utiliser plusieurs modèles et tailles de seringue. (La figure 1.1) illustre un modèle de pousse seringue électrique

L'utilité des pousse-seringues est d'administrer des médicaments en continu, avec un débit stable permettant l'obtention d'une concentration stable sur la durée d'administration. Cela permet d'éviter des périodes pendant lesquelles le taux de médicaments dans le sang est trop élevé ou trop faible. Ils sont largement utilisés pour l'administration d'anticancéreux, d'insuline, d'antibiotiques, d'antalgiques et d'amines vasopressives dans de nombreuses spécialités médicales : anesthésie-réanimation, chirurgie, infectiologie, soins palliatifs...

Figure 1.1 : modèle de pousse seringue

1.4.2 Le principe de fonctionnement

Le système combine des parties électriques et mécaniques. La partie mécanique sert de support pour les différents types de seringues. Elle 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 de fixation du piston. 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 circuit patient. Cette partie mécanique est mue par un moteur électrique alimenté soit par le secteur, soit par une batterie. L'utilisation d'une batterie est primordiale pour la sécurité,

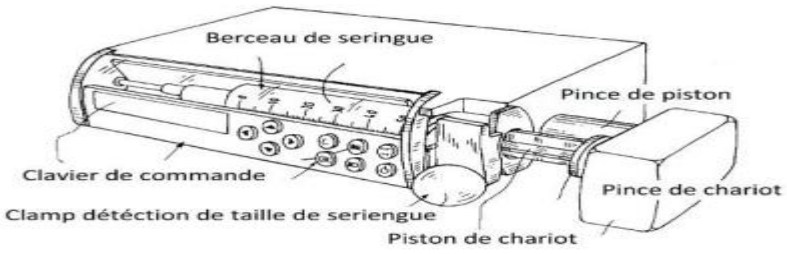


Figure 1. 2:partie mécanique de pousse seringue

Puisqu'elle permet une administration continue, même en cas de coupure de courant. Enfin la partie électronique, gère l'ensemble des autres éléments. Cette partie fonctionne aujourd'hui comme un véritable petit ordinateur capable non seulement de vérifier les débits,

les pressions, mais également d'effectuer de nombreux calculs de doses en fonction de protocoles divers.

1.4.3 Les 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 par exemple : Dopamine, le support de ce dernier peut acquérir une seule seringue de n'importe quel dosage, on les trouve généralement dans les Blocs Opératoire.

- **Pousse seringue à double voie :**

Il existe des pousses seringues double voies monobloc. Ils intègrent les mêmes contraintes qu'un 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.

Figure 1.3:pousse seringue à double voie

1.4.4 Les Différents modes d'administration des Pousse Seringues connectés

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 24H), 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 un protocole.

❖ **Le mode TIVA :**

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

❖ **Le mode AIVOC :**

Ce mode est considéré comme étant un sous 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.

1.4.5 Applications médicales des Pousse seringues électriques

Applications Médicale des PSE Cet appareil permet de perfuser en continu, à débit constant un soluté ou un médicament, dans un large domaine d'application ou on peut citer Les domaines suivants :

- Anesthésie et réanimation
- Bloc opératoire

- Urgences
- Cardiologie
- Néonatalogie Dans ces domaines d'applications nous allons injecter différent Médicaments Pour chaque Opération

3-6 Avantages et inconvénient d'un pousse seringues connectés

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 [9]. Généralement les avantages et les inconvénients des 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.

1.5 Conclusion

Dans ce premier chapitre, une présentation de l'organisation hôte et de ses différents domaines d'activité a été présentée. En outre, nous avons décrit le cycle de fonctionnement du poussoir de seringue électrique se concentrer sur Divers modes d'administration et les avantages et les inconvénients d'un PES.

Chapitre 2 : Exigences et méthodologie de travail

2.1 Introduction

[place your text here]

2.2 Norme IEC-62 304

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 [10].

2.2.1 Exigence générale

L'exigence contient sert à identifier le système de management de qualité, le système de gestion de risque a utilisé, et de classifier le niveau de sécurité du logiciel.

Les deux premiers, intéressent l'ingénieur qualité plus que l'ingénieur développement, mais la troisième étape est nécessaire pour avancer dans le développement vu que le niveau de sécurité attribué affecte directement le processus de développement. 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 alimentaire la classification sera B, par contre dans une chambre d'urgence, ou soins intensifs, ou la plupart des médicaments utilisé sont dangereux en haute concentration. Une erreur de dosage peut provoquer la mort du patient, la pousse seringue est classifié C.

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

2.2.2 Processus de développement logiciel

Le processus de développement du logiciel représente des étapes à suivre pour qu'elle soit conforme à la norme 62304. Les étapes à suivre sont régit par le niveau de sécurité attribué. Le tableau 4 décrit les étapes à suivre selon le niveau sécurité [11].

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

Etapes de développement logiciel et niveau de sécurité.

2.3 Méthodologie de travail git (push pull commit)

2.3.1 Problématique

Étant données les différentes taches dans ce projet et les diverses technologies utilisée nous sommes toujours besoins de tester le code source. Donc le basculement d'une version a un autre est difficile et consomme beaucoup du temps. D'autre part la supervision de l'avancement du projet est nécessaire.

2.3.2 Solution

L'outil le plus utilisée 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 dans un type spéciale de base de données, 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 MR SADOK BAZZINE).

2.3.3 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 méthodologie. Jusqu’à présent nous avons commité 33 commit dans la branche main.

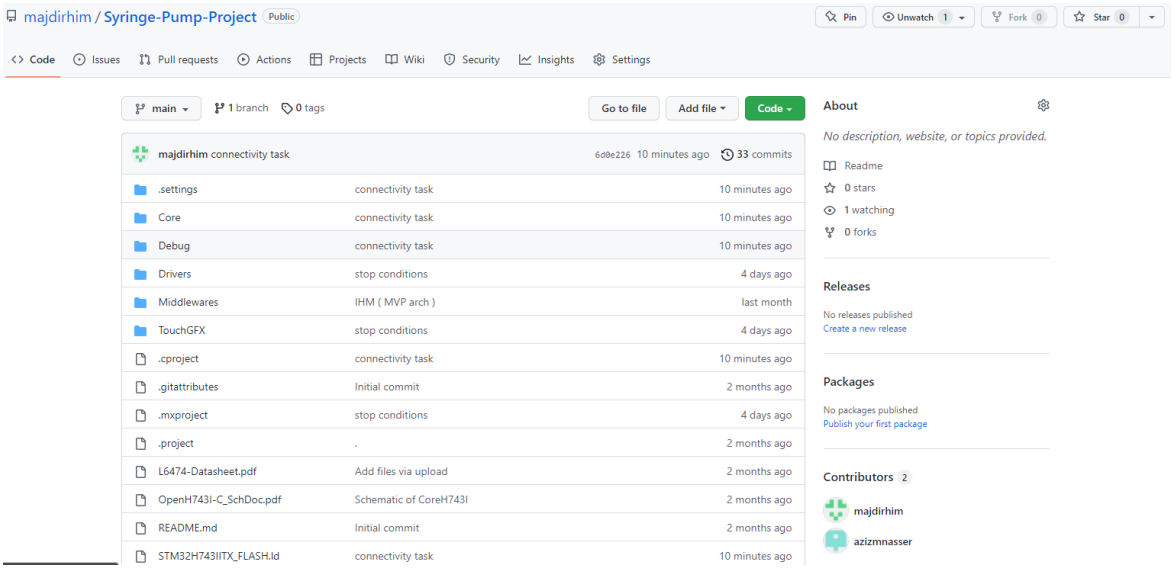


Figure 2.4:Structure du projet sur Git Hub

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 vers GitHub on utilise la commande “`git push`”. À l’aide de la commande “`git pull`” les contributeurs peuvent faire un fetch du contenu et le télécharger pour le modifier. Pour revenir en arrière on utilise la commande “`git reset --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.

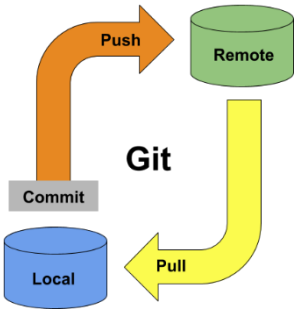


Figure 2.5 : méthodologie de travail

2.4 Conclusion

[place your text here]

Chapitre 3 : Analyse et spécification de besoins

3.1 Introduction

3.2 Spécification des besoins

3.2.1 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 assurer les médicaments au malade. Tell que le cas de covid19 et sur tous pour les malades dans un état comateux, et qui nécessite l'alimentation des malades par les médicaments 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 la pousse seringue électrique. Dans certain cas, la quantité de produit administré par injection à un patient doit être fractionnée dans le temps. La prise en une seule injection du médicament ne permet pas de maintenir un effet optimum 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.
- Contrainte pour le patient.

3.2.2 Cahier de charge

La définition d'un cahier des charges était l'un de nos principaux objectifs. Nous allons ici le détailler.

Il est question dans ce projet de développer une solution électronique pour le pilotage d'un modèle mécanique d'un Pousse Seringue Électrique (PSE).
La solution envisagée doit contenir :

- Une Interface Homme Machine (HMI) permettant une interaction étroite et un accès facile aux principales fonctionnalités du PSE.
  Elle est principalement composée d'un afficheur LCD TFT 4.3 mené d'une dalle tactile et d'un clavier avec des boutons d'accès rapide.

- Une carte de commande basée sur un microcontrôleur STM32 incluant tous les composants nécessaires pour la commande du moteur pas à pas et l'exploitation des capteurs installés sur le modèle mécanique du PSE ainsi que la gestion de la connectivité.

- Une solution logiciel exécutant les tâches du PSE au moyen d'un code source mené sur les outils de développement STMicroelectronics, CubeIDE , TouchGFX et ArduinoIde.

- Une application mobile pour la supervision à distance

3.3 Architecture globale « Software –Hardware

3.3.1Architecture Hardware :

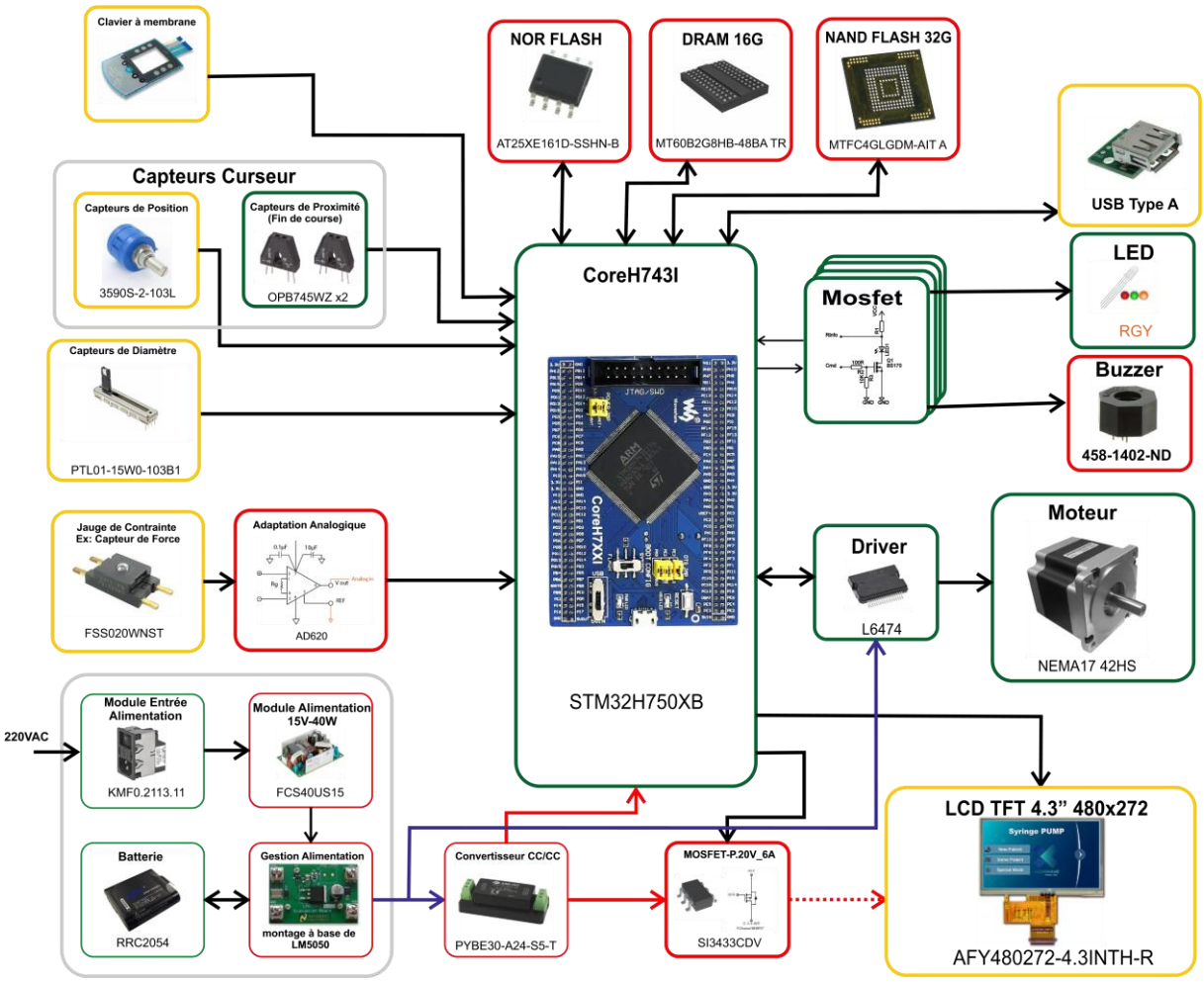


Figure 6: architecture hardware de la pousse seringue

✓ Matériels utilisés :

❖ STM32H7 :

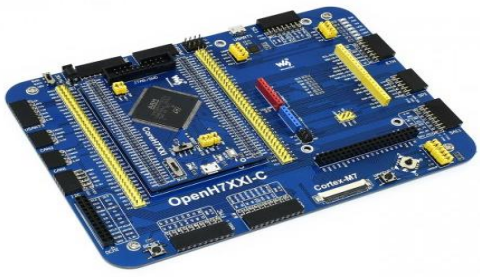


Figure 3. 7:Carte de development STM32H7

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. La conception modulaire et ouverte en fait l'outil idéal pour démarrer le développement d'applications avec les microcontrôleurs de la série STM32.

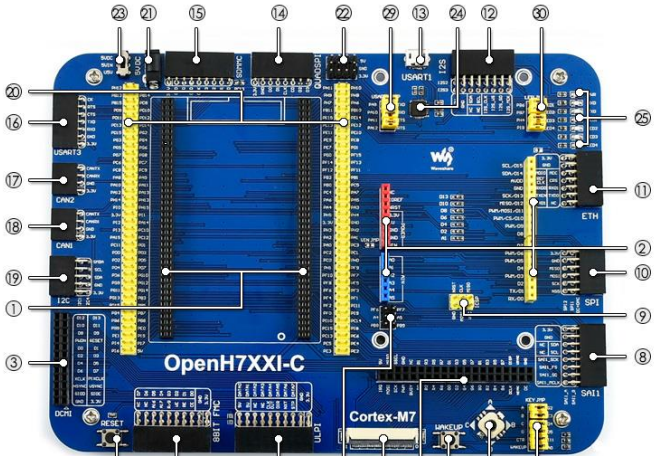


Figure 3.8 : Composants de l'OpenH7XXI-C

N°	Nom de composant	Rôle
1	MCU core board connector	easily connecting the CoreH743I
2	Arduino interface	connecting Arduino shields
3	DCMI interface	connecting camera
4	8-bit FMC interface	easily connects to peripherals such as NandFlash
5	ULPI interface	connecting high-speed USB peripheral (the STM32H743I integrates USB HS controller without any PHY device)
6	LCD interface 1	connecting 10.1inch LCD, 7inch LCD, 4.3inch LCD
7	LCD interface 2	connecting 4.3inch LCD
8	SAI1 interface:	connecting audio modules
9	ICSP interface:	Arduino ICSP
10	SPI interfaces:	✓ easily connects to SPI peripherals such as DataFlash (AT45DBxx, W25QXX), SD card, MP3 module, etc. ✓ easily connects to AD/DA modules (SPI1 features

		AD/DA alternative function
11	Ethernet interface	connecting Ethernet modules
12	I2S / I2C interface	easily connects to I2S peripherals such as audio module, etc.
13	USART1 connector	USB to UASRT via the onboard convertor CP2102
14	QUADSPI interface	4-wires SPI interface (the H7 series latest peripheral interface), for connecting serial Flash modules like W25QXX Board
15	SDMMC interface	connecting Micro SD module, features much faster access speed rather than SPI
16	USART3 interface	easily connects to RS232, RS485, USB TO 232, etc.
17	CAN2 interface	connecting CAN modules
18	CAN1 interface	connecting CAN modules
19	I2C1/I2C4 interface	easily connects to I2C peripherals such as I/O expander (PCF8574), EEPROM (AT24Cxx), 10 DOF IMU Sensor, etc.
20	MCU pins connector	all the MCU I/O ports are accessible on expansion connectors for further expansion
21	5V DC jack	
22	5V/3.3V power input/output	usually used as power output, also common-grounding with other user board
23	Power supply switch	powered from 5VDC OR USB connection of the USART1
24	CP2102	USB to UART convertor
25	LEDs	convenient for indicating I/O status and/or program running state
26	Joystick	five positions
27	WAKE UP button	used as regular button, and/or wake up the STM32 MCU from sleep
28	Reset button	
29	USART1 jumper	
30	LED jumper	✓ short the jumper to connect to default I/Os used in example code ✓ open the jumper to connect to custom I/Os via jumper wires
31	KEY jumper	✓ short the jumper to connect to default I/Os used in example code

		✓ open the jumper to connect to custom I/Os via jumper wires
32	Arduino jumper	✓ short the upper pins, A4, A5 is used as AD function ✓ short the lower pins, A4, A5 is used as I2C function

❖ CoreH743I

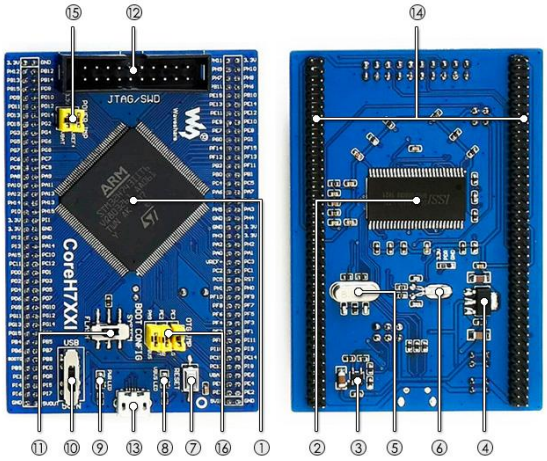


Figure 3. 9: Composants du CoreH743I

N°	Nom de composant	Description
1	STM32H743IIT6	the high performance STM32 MCU which features: **Core**: Cortex-M7 32-bit RISC + double-precision FPU + Chrom-ART graphic accelerator **Feature**: single-cycle DSP instructions **Operating Frequency**: 480MHz, 1027 DMIPS / 2.14 DMIPS/MHz **Operating Voltage**: 1.62V-3.6V **Package**: LQFP176 **Memories**: 2MB Flash, 1MB RAM (864KB User+192KB TCM+4KB Backup) **AD & DA converters**: 3 x AD (16-bit); 2 x DA (12-bit) **Debugging/Programming**: supports JTAG/SWD interfaces, supports IAP
2	IC42S16400J / IS42S16400J	SDRAM 1 Meg Bits x 16 Bits x 4 Banks (64-MBIT)
3	STMPS2151STR	onboard USB power management device
4	AMS1117-3.3:	3.3V voltage regulator
5	8M crystal	
6	32.768K crystal	for internal RTC with calibration

7	Reset button	
8	VBUS LED	USB port indicator
9	PWR LED:	Power indicator
10	Power supply switch,	powered from 5Vin or USB connection
11	Boot mode selection	for configuring BOOT0 pin
12	JTAG/SWD interface	for debugging/programming
13	USB connector	, supports Device and/or Host
14	MCU pins expander	, VCC, GND and all the I/O pins are accessible on expansion connectors for further expansion
15	POWER jumper	

- **MOTEUR PAS à PAS (NEMA17) :**

Figure 3.10: Moteur PAS à PAS (NEMAI17)

✓ 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 3.11 : stepper motor driver shield L6474

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. 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.

✓ 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 :**

Figure 3.12 : 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 3.13 : Node MCU ESP8266

L'ESP8266 est décliné en plusieurs variantes. 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) :**

Figure 3.14 : Capteur de position (3590S-2-103L)

✓ 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) :**

Figure 3.15 : 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 de proximité (OPB745WZ) :**

Figure 3.16 : Capteur de proximité (OPB745WZ)

- ✓ Caractéristiques :
    - Distance de détection : 3.81mm
    - Collecteur de tension : 15v
    - VF-tension directe : 1.8v
    - VF-tension inverse : 2v
    - Température de fonctionnement min : -40°C
    - Température de fonctionnement max : +80°C

❖ **Capteur jauge de contrainte (FSS020WNST) :**

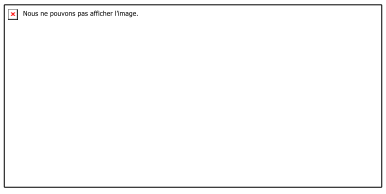


Figure 3.17 : Capteur jauge de contrainte (FSS020WNST)

✓ Caractéristique :

- Force de commande : 20N
- Précision : 0.5%
- Type de sortie : Analogique
- Style de montage : SMD/SMT
- Tension d'alimentation : 12V

❖ **Batterie (RRC2054) :**

Figure 3.18 : Batterie (RRC2054)

✓ Caractérıstıque :

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

❖ **RRC-PMM240 :**

Figure 3.19 : RRC-PMM240

C'est un Module de gestion de l'alimentation pour les applications mobiles

✓ 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'apllication
- Convient à la batterie intelligente standard de RRC au format RRC20xx

❖ **Buzzer (458-1402-ND) :**

Figure 3.20: 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

3.3.2 Architecture Software :

✓ 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. Il permet l'intégration des centaines de plugins existants qui compléter les fonctionnalités de l'IDE Eclipse®.

- **TOUCHGFX :**

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

3.4Conclusion

Chapitre 4 : Conception et réalisation

4.1 Introduction

4.2 OS kernel

4.2.1 Middleware (FreeRTOS)

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

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

- **RTOS**

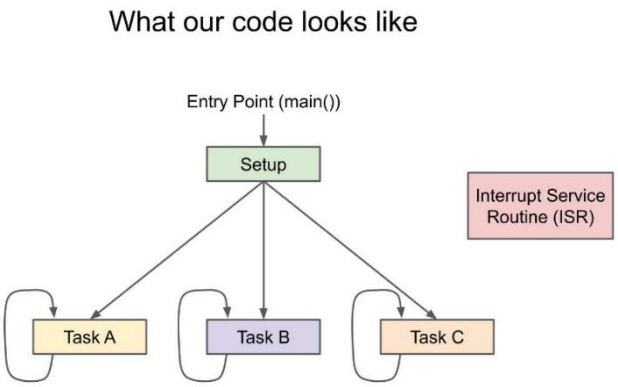
Le système d'exploitation temps réel est un OS qui gère plusieurs tâches concurrentes selon leurs degrés de priorités, il est utilisé quand il y a des exigences temporelles sur les processus. Ce type d'ordonnancement, appelé ordonnancement préemptif.

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

4.2.2Tâches 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)).**

Ci-dessous sont les taches utilisées dans le projet classé selon leurs priorités d'exécution.

```
56
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",
```

Figure 4.22: 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 `osPriorityAboveNormal`, 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 tache IHMHandle `osPriorityAboveNormal`, toutes les données nécessaires sont envoyées à travers cette tache vers le Cloud afin de les récupérer en temps réels par une application de supervision.

✓ **Tache 4 : SensorsHandle**

Cette tâche représente l'unité de traitements de tous les capteurs utilisés dans ce projet elle dispose comme priorité `osPriorityNormal1`.

✓ **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 4.23 : Création des taches

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**.**

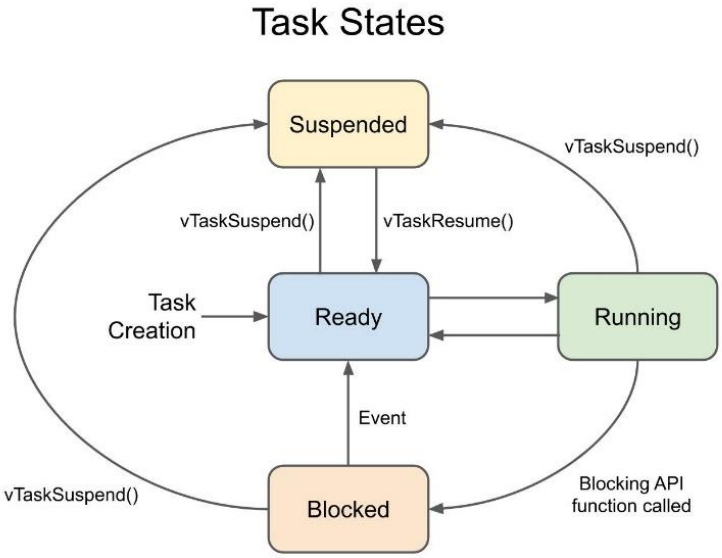


Figure 4.24 : états des tâches

❖ 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 taches. C'est très probable alors que deux taches 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.

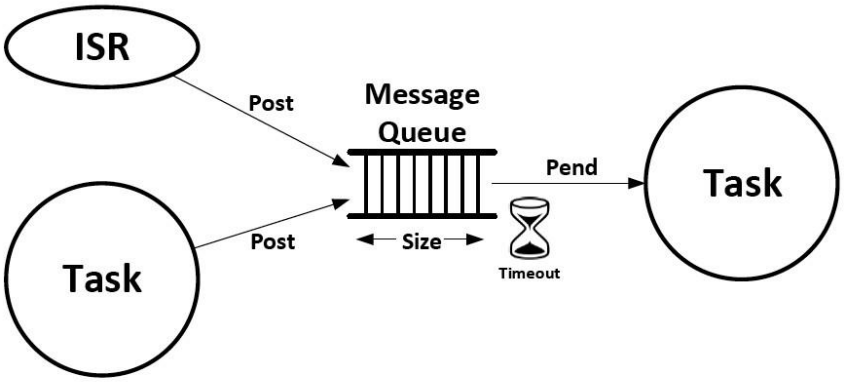


Figure 4.25 : file d'attente et mode FIFO

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`".

```
/// 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 4.26 : 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`".

```
/* 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 4.27 : définition des fonctions de lecture et écriture dans une queue

4.3 Moteur pas à pas

4.3.1 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 selon nos besoins.

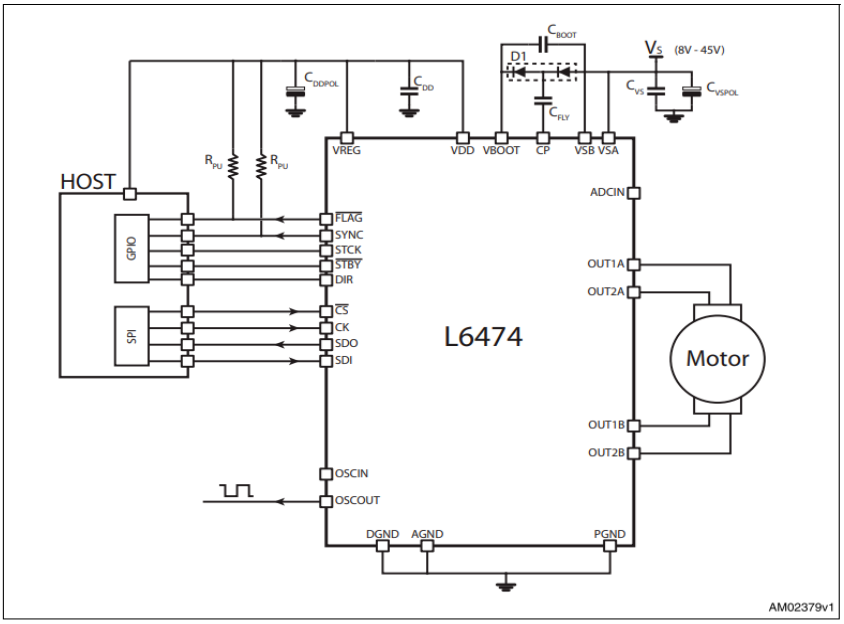


Figure 4.28: schéma de câblage du driver L6474

Nous travaillons avec la carte d'expansion **x-nucleo-ihm01a1** basée sur le L6474 Driver et le moteur pas à pas **Nema 17**.

La communication entre notre carte et le driver est à travers le protocole **SPI** 8bit (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).", **Wikipédia**

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**).

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 (1)	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 4.29 : 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 4.30 : adresse des différents registres

La bibliothèque fournit par ST "**X-CUBE-SPN1**" peut gérer tous les commandes bas niveau à travers des fonctions prédéfinit 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 (OpenH7). Les timers 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 été 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.

```
/// The maximum number of devices in the daisy chain
#define MAX_NUMBER_OF_DEVICES                (3)

/*********************** Speed Profile  ******************************/

/// Acceleration rate in step/s2 for device 0 (must be greater than 0)
#define L6474_CONF_PARAM_ACC_DEVICE_0        (1)
/// Acceleration rate in step/s2 for device 1 (must be greater than 0)
#define L6474_CONF_PARAM_ACC_DEVICE_1        (160)
/// Acceleration rate in step/s2 for device 2 (must be greater than 0)
#define L6474_CONF_PARAM_ACC_DEVICE_2        (160)

/// Deceleration rate in step/s2 for device 0 (must be greater than 0)
#define L6474_CONF_PARAM_DEC_DEVICE_0        (1)
/// Deceleration rate in step/s2 for device 1 (must be greater than 0)
#define L6474_CONF_PARAM_DEC_DEVICE_1        (160)
/// Deceleration rate in step/s2 for device 2 (must be greater than 0)
#define L6474_CONF_PARAM_DEC_DEVICE_2        (160)

/// Maximum speed in step/s for device 0 (30 step/s < Maximum speed <= 10 000 step/s )
#define L6474_CONF_PARAM_MAX_SPEED_DEVICE_0  (1000)
/// Maximum speed in step/s for device 1 (30 step/s < Maximum speed <= 10 000 step/s )
#define L6474_CONF_PARAM_MAX_SPEED_DEVICE_1  (1)
/// Maximum speed in step/s for device 2 (30 step/s < Maximum speed <= 10 000 step/s )
#define L6474_CONF_PARAM_MAX_SPEED_DEVICE_2  (1600)

/// Minimum speed in step/s for device 0 (30 step/s <= Minimum speed < 10 000 step/s)
#define L6474_CONF_PARAM_MIN_SPEED_DEVICE_0  (1000)
/// Minimum speed in step/s for device 1 (30 step/s <= Minimum speed < 10 000 step/s)
#define L6474_CONF_PARAM_MIN_SPEED_DEVICE_1  (800)
/// Minimum speed in step/s for device 2 (30 step/s <= Minimum speed < 10 000 step/s)
#define L6474_CONF_PARAM_MIN_SPEED_DEVICE_2  (800)
```

Figure 4. 32 : fichier de paramètre par défauts

```
/* 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 shutwdown (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 Swicthing 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       |
       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)    /// Alarm (ALARM_EN register).
  };
/* USER CODE END 0 */

/**
```

Figure 4.31 : structure C L6474_Init_t

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 minimale va être modifié dans le code selon le débit d'injection à l'aide des fonctions

"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 4.33 : les fonctions pour commander le moteur

4.3.2 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.

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.

Figure 4.34 :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 alarme s'agissent des toggles Led et une notification dans l'application mobile. Nous allons les modifiés aux furs 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 brigdes 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   }
```

Figure 4.35 : Fonctions d'appel lors une interruption

4.3.3 Flux de données :

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

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 on a évidemment un flux de données sortant de la tache **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`". Ainsi, on partage le nombre total des pas dans la queue "`LastStepQHandle`".

4.3.4 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 (m3/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 :

$\mathbf{q_v}$: débit volumique en [m^3/s]

v : vitesse du fluide en [m/s]

Pour calculer la section ($\pi * r^2$) 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.

```
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 36 : fonction pour calculer la vitesse de déplacement nécessaire

1.8° donc le nombre des pas totale pour une seule révolution est **360°/1.8°=200 pas/révolution.**

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)

```
348 // returns the motor speed needed (rps)
349 uint16_t Motor_Speed(uint16_t screwspeed){
350     return screwspeed / (SCREWSTEP*0.001);
351 }
```

Figure 37 : fonction pour calculer la vitesse du moteur

La vitesse du moteur dans la bibliothèque est en **pas/seconde** donc la fonction `L6474_SetMaxSpeed` prend en paramètre **N*200.**

Le temps total de l'opération d'injection est égal au rapport **volume/débit.**

```
374 //return number of seconds to finish the injection
375 float Time_Needed(float flow_rate, float volume_to_inject){
376     flow_rate = flow_rate  / 3600;
377     return (volume_to_inject/flow_rate);
378 }
379
380 void SyringeMove(uint16_t FlowRate , uint8_t radius){
381     float screwspeed , motorspeed;
382     int pps;
383     screwspeed = Screws_Speed_From_FlowRate(FlowRate,radius);
384     motorspeed = Motor_Speed(screwspeed);
385     pps=motorspeed*200*16; // 1/16 microstep
386     L6474_SetMaxSpeed(0,pps);
387     L6474_SetMinSpeed(0, pps);
388     L6474_Run(0, FORWARD);
389     /*drv8825_setSpeedRPM(drv8825, motorspeed*60);
390     drv8825_setEn(drv8825, EN_START);*/
391 }
392 void SyringeStop(){
393         //drv8825_setEn(drv8825, EN_STOP);
394     L6474_HardStop(0);
395 }
396
```

Figure 4.38 :fonction pour commander le moteur selon les paramètres calculer

4.4 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 calcules et traitements.

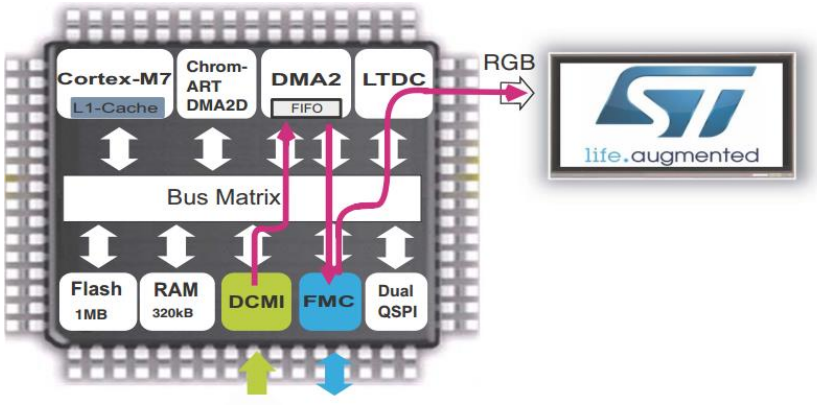


Figure 4.39: interfaçage RGB

C'est pour cela qu'on a utilisé un accélérateur **DMA2D.**

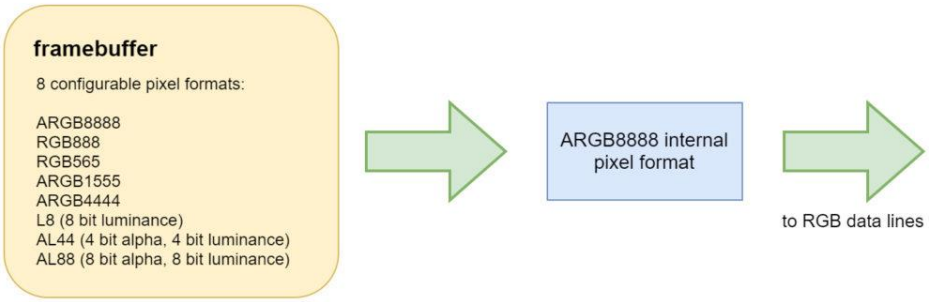


Figure 4.40: Structure de données passées dans LTDC

Le périphérique de touche résistive est un système qui va convertir les données analogiques des touches à 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 connaitre 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.

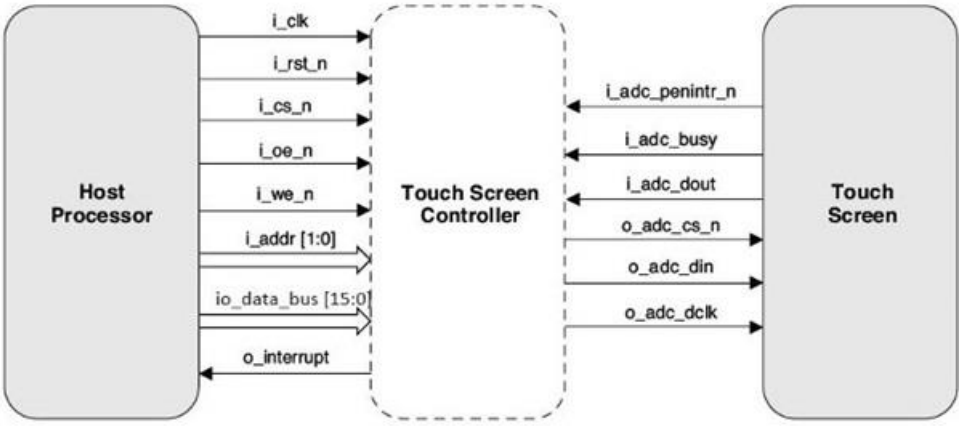


Figure 4.41 : Communication dalle tactile résistive et processeur

❖ Architecture (MVP)

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

Le X-CUBE-TOUCHGFX est une expansion développée par ST pour faciliter l'intégration avec CUBE-IDE, 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.

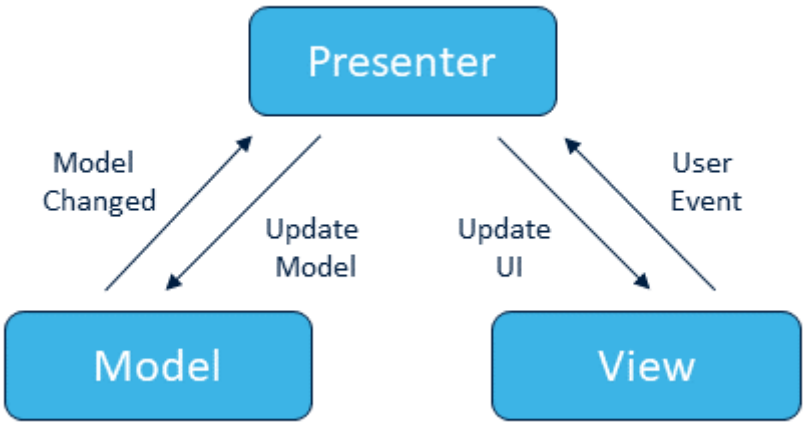


Figure 4.42 : Modèle de conception modèle-vue-présentateur

❖ **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 taches 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.

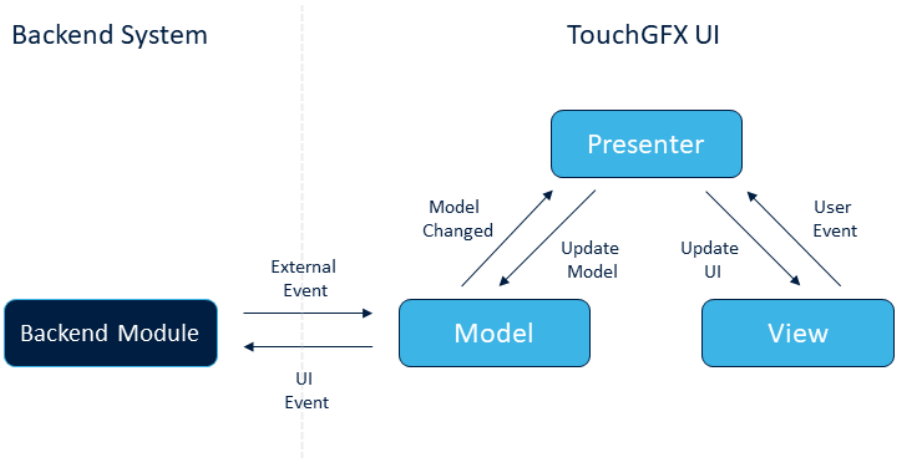


Figure 4.43 : Modèle-Vue-Présentateur et communication externe

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

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.

Figure 4.44 : interface principale

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

Enfin cette dernière fait l’appel de la fonction “`saveFlowaRate`” définit dans la classe Model.

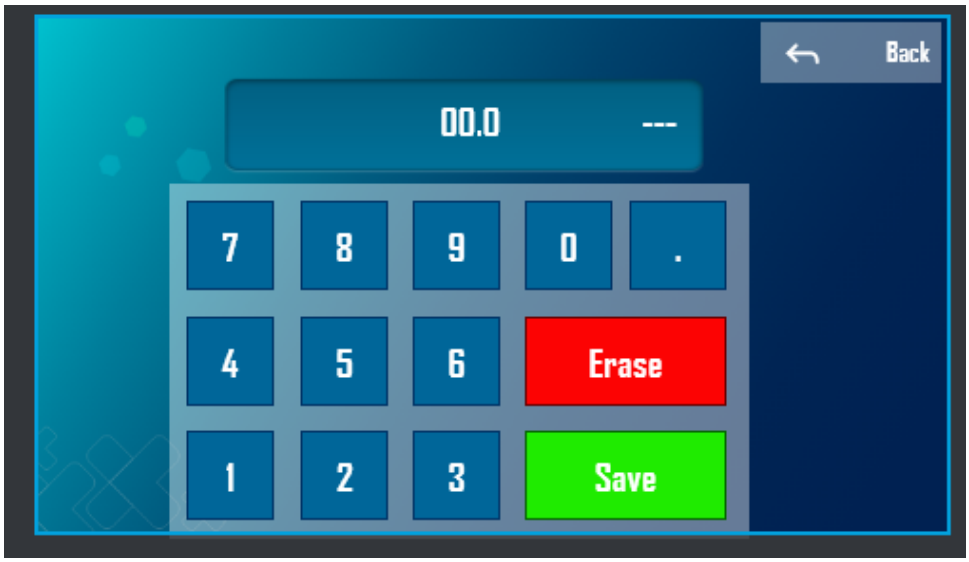


Figure 4.45 : interface clavier

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

```
220
221 /*============================PERFUSION=====================================*/
222 void Model::saveFlowaRate(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
```

La fonction “saveInfusionData” transmet les données vers le backend à travers une queue ”InfusionQHandle” sous format d’une structure C de type “Infusion_paramT” qui admet tous les paramètres d’infusion.

```
413
414 void Model::saveInfusionData(void)
415 {
416 #ifndef SIMULATOR
417     // RTOS
418     if(PerfusionParameters.Flowrate!=0 || (PerfusionParameters.InfousionVolume!=0 )){
419         osMessageQueuePut(InfusionQHandle,&PerfusionParameters,1,100);
420     }
421 #endif
422 }
```

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

Figure 4.46 : 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’autre files d’attente selon le besoin.

```
/* USER CODE END Header_Interface */
void Interface(void *argument)
{
  /* USER CODE BEGIN Interface */
    Infusion_paramT msgPerfusionParameters;
  /* Infinite loop */
  for(;;)
  {
      // ***** 0 => StopMode , 8=> PauseMode *******
      if(osMessageQueueGet(InfusionQHandle,&msgPerfusionParameters,10U,100)==osOK && msgPerfusionParamete
              &&  msgPerfusionParameters.Mode!=8  ){
          osMessageQueuePut(FlowRateQHandle,&msgPerfusionParameters.Flowrate , 1U, 100U);
          osMessageQueuePut(VolumeQHandle,&msgPerfusionParameters.InfousionVolume , 1U, 100U);
      }else{
          osMessageQueuePut(ModeQHandle,&msgPerfusionParameters.Mode , 10U, 100U);
      }

      osDelay(1);
  }
  /* USER CODE END Interface */
}
```

Figure 4.47 : structure tache IHMHandle

4.5 Capteurs et mesures

4.5.1 Capteur de diamètre

- **Solution proposée :**

Le calcule 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 connaitre le rayon.

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

Généralement, on a une idée sur les rayons possibles d'après le type de la seringue. Notre solution est de détecter dans quelles marges se trouve le diamètre de la seringue et puis assigner la valeur convenable selon cette marge.

Le capteur s'agit d'un potentiomètre linéaire ou bien "slide potentiometer" 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.

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

→ 0.34mm pour le volume 5 mm^3

→ 0.49mm pour le volume 10 mm^3

→ 1.03mm pour le volume 50mm^3

→ …

Si le capteur indique x ou y la variable rayon reçoit 0.49mm et puis on fait le calcul de la section.

- **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 entré 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.

4.5.2 Capteur de position

- **Solution proposée**

Avec le même principe On a utilisé un potentiomètre rotatif avec des dents 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. En fait, pour détecter la position de la seringue on doit

4.5.3 Capteur de température

Avec toutes les taches 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 (`HAL_ADC_LevelOutOfWindowCallback(ADC_HandleTypeDef *hadc)`)

```
/** 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();
}
```

si la conversion est hors la marge spécifiée.

Figure 4.48 :configuration Watchdog analogique

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

```
tCelsius = (110 - 30) * (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`".

```
// cpu temp interrupt
void HAL_ADC_LevelOutOfWindowCallback(ADC_HandleTypeDef *hadc){
    // do something in case of analog watchdog interrupts
    HAL_GPIO_WritePin(LED2_GPIO_Port, LED2_Pin, GPIO_PIN_SET);
    HAL_ADC_Stop_IT(&hadc);
}
```

Figure 4.49:fonction d'appel lors d'une interruption wdg analogique

4.5.4 Capteur de pression

4.6 Connectivité

La connectivité est un élément clé dans notre projet, vu qu'elle est rarement utilisée dans les pousses seringues 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.

4.6.1 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 récupèrent, faire le traitement si nécessaire et les affichent au médecin. À l'aide de cette architecture, les informations sont disponibles même si l'application est hors ligne.

L'envoie 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.

4.6.2 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.

4.6.3 Implémentation :

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

Ci-dessous est la liste des données à envoyées 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 cet ID pour chaque pousse seringue.

Cet identifiant est stocké dans une adresse mémoire à lecture seul comme il est indiqué ci-dessous figure 4.50. 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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
U_ID(31:0)																															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

Bits 31:0 **U_ID(31:0):** 31:0 unique ID bits

Address offset: 0x04

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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
U_ID(63:48)															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
U_ID(47:32)															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

Bits 31:0 **U_ID(63:32):** 63:32 unique ID bits

Address offset: 0x08

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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
U_ID(95:80)															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
U_ID(79:64)															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

Bits 31:0 **U_ID(95:64):** 95:64 Unique ID bits.

// adresses UID de l'stm32h743I

Au niveau esp8266 on pourrait encodé UID de sa forme décimal vers la Base64 (string) pour minimiser la taille de l'identifiant et le rendre plus lisible.

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.

```
/* 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];
  /* 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%f",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);
            }
    osDelay(100);

  }
  /* USER CODE END Cloud_Connectivity */
```

//structure de la tache connectivité

Dans l'autre côté, le module esp8266 admet son propre code pour gérer la communication **TCP-IP**, il est indépendant au celle de l'Stm32 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.

En fait la solution cloud choisit est "**Firebase 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 donné

Figure 4.50 : structure de la tache ConnectivityHandle

```
void loop()
{
  if (Serial.available() && (millis() - dataMillis) > 100 && digitalRead(5)==HIGH)
  {
    dataMillis = millis();
    char inChar = Serial.read(); // check first letter and assigned to a corresponding value
    switch (inChar) {
      case 'v' :
        {
          float vol = Serial.parseFloat();
          Serial.printf("a %s %f\n", Firebase.RTDB.setFloat(&fbdo, "/Volume_Left", vol) ? "ok" : fbdo.errorReason().c_str(), vol);
          break;
        }

      case 'f':
        {
          float flow = Serial.parseFloat();
          Serial.printf("b %s %f\n", Firebase.RTDB.setFloat(&fbdo, "/Flow_Rate", flow) ? "ok" : fbdo.errorReason().c_str(), flow);
          break;
        }

      case 't':
        {
```

Figure 4.52: Communication entre Firebase et la carte ESP8266

4.7 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
- Stockage des données en local dans une carte SD
- Clavier à Membrane
- Capteur de pression
- Application mobile

4.8 Conclusion

[place your ext here]