INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
CENTRE VAL DE LOIRE

Minh-Thuc PHAM
5A GSI
Option : ACAD
Année universitaire :
2019 - 2020

RAPPORT DE STAGE

CONTROLE DU BRAS ROBOTIQUE LYNXMOTION AL5D

ELSYS DESIGN

Tuteur entreprise :
Marc RENOU
Enseignant référent :
Francesco COLAMARTINO

Elsys Design Sophia Antipolis
2323 Chemin de Saint-Bernard, 06220 Vallauris

Génie des Systèmes Industriels
Institut National des Sciences Appliquées

Blois, 20 Août 2020

Remerciements

Je profite dès les premières lignes de ce rapport pour remercier toutes les personnes qui ont contribué au suivi et au succès de mon stage.

Tout d'abord, j'adresse mes remerciements à M. Marc RENOU et M. Adrien PAGLIANO, mes maîtres de stage au sein de l'entreprise Elsys Design pour leur accueil, le temps passé ensemble et leur confiance en me recrutant dans cette entreprise. Grâce à cette confiance, j'ai pu trouver un stage qui était en totale adéquation avec mes attentes. En outre, ils m'ont aidé beaucoup pour m'intégrer dans la société et m'adapter l'environnement de travail chez Elsys Design.

Je tiens à remercier vivement à mes consultants techniques Olivier PARA et Gaël GUEGUAN pour leur enthousiasme et le partage de leur expertise au quotidien référent. Ils m'ont donné les conseils les plus importants au niveau technique ainsi que fonctionnel dans les moments les plus difficiles. Ils m'ont montré mes points faibles à améliorer et m'ont orienté vers les technologies utilisées dans l'industrie 4.0.

Je remercie également à mon tuteur référent, M. Francesco COLAMARTINO, pour son soutien constant tout au long de ce stage.

Je remercie sincèrement toute l'équipe de Elsys Design Sophia Antipolis, pour l'intérêt qu'ils m'ont porté, l'esprit du travail en équipe et en particulier M. Etienne RAQUIN pour sa gentillesse et sa générosité.

Enfin, je tiens à remercier mon école de m'avoir donné une opportunité de faire un stage de 6 mois qui m'a aidé à acquérir plus d'expériences dans ma future profession, je remercie aussi tous les personnes qui m'ont conseillé et relu lors de la rédaction de ce rapport de stage.

Résumé

Dans le cadre de mes études d'ingénieur en informatique et systèmes embarqués à l'INSA Centre Val de Loire, ce stage de fin d'études s'est effectué au sein de la société ELSYS DESIGN Sophia Antipolis entre février et août 2020. L'objectif du stage était de réaliser des missions techniques dans mon domaine d'études pendant une période significative, afin d'appliquer toutes les connaissances et compétences acquises au cours de mes études.

Ma mission de 6 mois a été divisée en plusieurs parties principales concernant les méthodes pour contrôler le bras robotique Lynxmotion AL5D. D'abord, j'ai développé un logiciel embarqué fonctionné sur le systèmes Linux comme Raspberry PI ou PC qui permet les manettes de contrôler le bras robotique via le protocole USB. Ensuite, il faut créer un service web embarqué en C/C++ et une interface web client qui nous permettent de contrôler le bras robotique via le navigateur web. En fin, j'étais responsable de développer un logiciel FreeRTOS sur la carte MIMXRT1010-EVK. Ce logiciel permet cette carte de communiquer avec le bras robotique par le protocole USB ou série (UART). Avec cette méthode, j'ai dû concevoir un script pour le mouvement du bras.

Le challenge de ce projet était de rechercher des solutions pour contrôler le bras robotique avec les matériaux existantes et les déployer pour démontrer la faisabilité de ces méthodes. Par manque du temps et des difficultés dans la période de confinement à cause du COVID 19, je n'ai pas pu réaliser toutes les idées et aussi prendre en compte l'aspect de cybersécurité. Pourtant, au final, je suis arrivé à fournir quelques solutions qui répondent aux exigences définies au début de mon stage.

Mot clés : programmation Linux, microcontrôleur, FreeRTOS, USB, UART, serveur embarqué, HTTP/HTTPS, OpenGL, Yocto, C/C++, HTML, CSS, robotique.

Abstract

As part of my studies as an engineer in IT and embedded systems at INSA Centre Val de Loire, i completed my end-of-studies internship which tool place at the ELSYS DESIGN Sophia Antipolis between february and august 2020. The goal of this internship were to perform a technical mission in my field for a significant period, in order to apply all the knowledges and skills acquired in university.

My mission in 6 month was divided into several main parts regarding the methods to control the Lynxmotion AL5D robotic arm. First, i developed an embedded software running on Linux systems like Raspberry PI or PC which allows the joysticks to control the robotic arm via the USB protocol. Next, created an embedded web service in C/C++ and an interface client which allow us to control the robotic arm via the web browser. In the end, i was responsible for developing FreeRTOS software on the MIMXRT1010-EVK board. This software request the board to communicate with the robotic arm via the USB or serial (UART) protocol. I had to design a script for arm's movement.

The challenge of this project was to find solutions to control the robotic arm with existing hardware and deploy them to demonstrate the feasibility of these methods. Because of the lack of time and difficulties in the confinement period due to COVID 19, i could not do all the ideas and also measure the aspects security in the communication. However, in the end, i finished some solutions that meet the requirements defined in the begin of my internship.

Key words : Linux programming, microcontroller, FreeRTOS, USB, UART, embedded server, HTTP/HTTPS, OpenGL, Yocto (build root), C/C++, HTML, CSS, robotic.

Sommaire

Introduction

Elsys Design est une entreprise spécialisée dans le domaine de systèmes embarqués et un des directions que l'entreprise se concentre et apprécie ces dernières années est la robotique. Dans le cadre du projet robotique, l'entreprise a envie d'entraîner leurs ingénieurs en construisant une plateforme de multiples solutions pour contrôler le bras robotique Lynxmotion AL5D (un système abordable avec une conception solide qui offert un mouvement rapide, précis et reproductible. Le robot comprend : rotation de la base, épaule sur un seul plan, coude, mouvement du poignet, une pince fonctionnelle et rotation du poignet en option). Bien que le projet ait été commencé par un groupe des stagiaires dans l'année dernière en fournissant des méthodes de contrôler du bras robotique comme Bluetooth Low Energie manette, GPIO manette, I2C, PLC... l'entreprise voudrait trouver plus de manières faisables qui sont compatibles avec ce type de robot. De cette façon, les stagiaires pourront approcher de plus en plus des activités dans le domaine robotique et ainsi renforcer les compétences essentielles d'un ingénieur du développement du logiciel embarqué comme programmation Linux, microcontrôleur, développement du driver et temps réel.

Pendant mon stage de fin d'études, j'assurais le rôle de développeur du logiciel embarqué et me concentre sur une application plus concrète : contrôle du bras robotique à partir des matériaux existants. Mon objective est de chercher des solutions faisables pour la telle problématique et réaliser des tests nécessaires et la démonstration.

La première direction qui a été poursuit dans ce projet était le logiciel dans l'espace d'utilisateur sur un système qui a le système de l'exploitation GNU et le noyau Linux. Ce système joue un rôle d'un serveur qui récupère les événements des manettes, les analyse et envoyer les commandes au bras robotique. Cette méthode m'a permis d'améliorer la compétence de la programmation Linux qui est un des secteurs clés dans le domaine embarqué. La deuxième méthode que j'ai approché est déployer un serveur embarqué qui attache des commandes à partir du navigateur web (Front-end) et contrôler le bras ensuite (back-end). La dernière solution que j'ai tenté de réaliser, est de développer une application sous FreeRTOS avec la carte MIMXRT1010 pour contrôler le bras via le protocole USB ou Série. Avec ces choix, j'ai réussi à contrôler le bras par plusieurs manières possibles et appréhendé toutes les compétences dans les aspects différents. Ce rapport est alors constitué de 4 sessions principales :

— Présentation de Elsys Design.
— l'introduction des ressources utilisées dans le projet.
— Les solutions du contrôle de bras robotique Lynxmotion AL5D.
— La conclusion du stage.

Pour noter, la deuxième session se compose d'une présentation des spécifications du bras robotique Lynxmotion AL5D et des outils de développement utilisés au cours du stage.

Ci-dessous, le diagramme de Gantt qui représente les différentes étapes de mon travail durant mon stage de 6 mois.

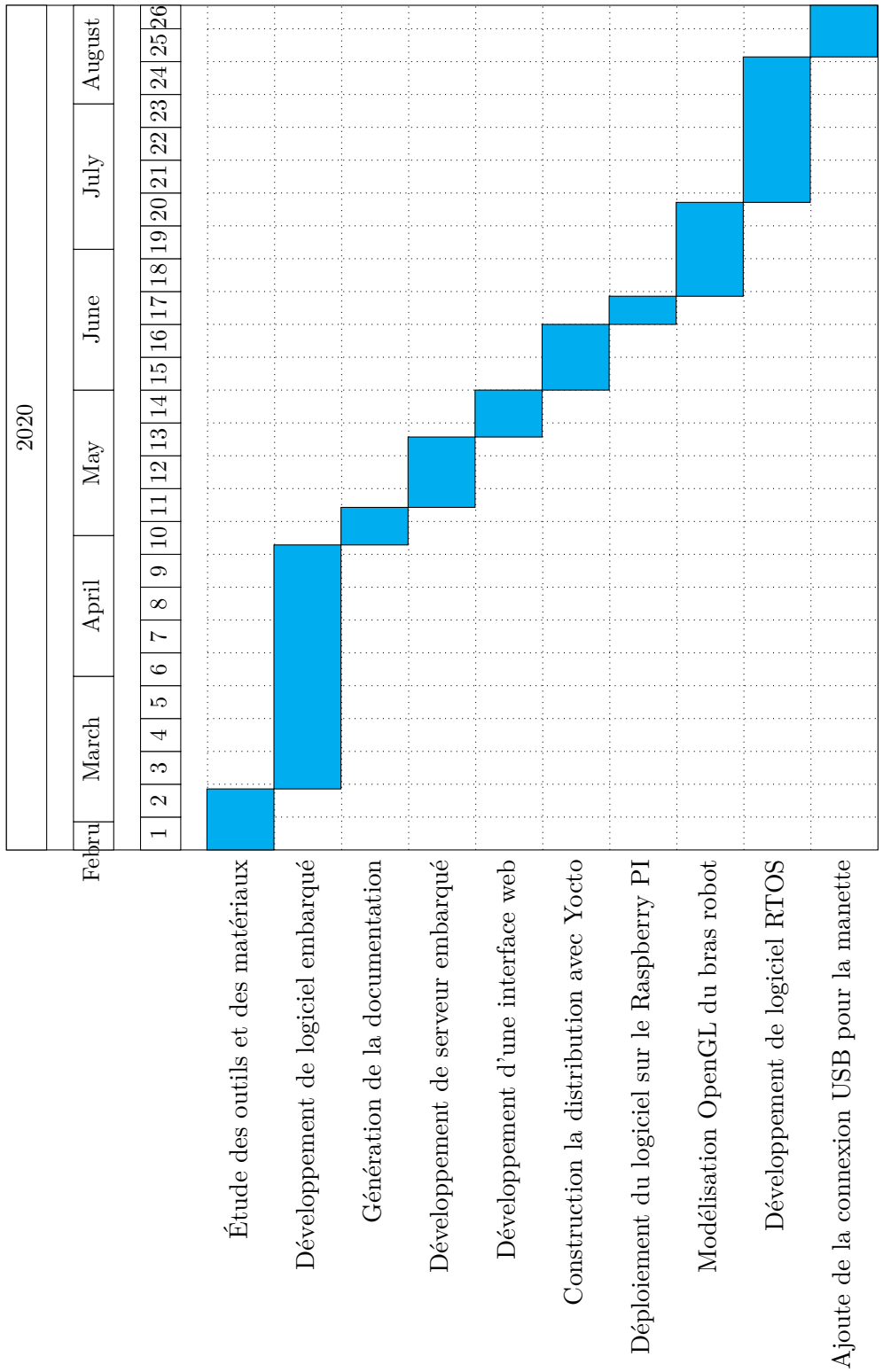

FIGURE 1 – Diagramme de Gantt

1 Présentation de l'entreprise

Ce chapitre présente la société ELSYS Design où je termine le stage de fin d'études. ELSYS Design appartient au groupe ADVANS avec trois autres sociétés. J'ai effectué mon stage à Sophia Antipolis, mais ELSYS Design est implanté dans de nombreux pays et possède de nombreux centres en France.

1.1 Group ADVANS

Le Groupe ADVANS, fondé en 2000 et géré par des ingénieurs, propose une large gamme de services d'ingénierie dans les domaines des systèmes embarqués, des logiciels d'application et de la mécanique. Le groupe conçoit des solutions technologiques et accompagne le développement de jeunes entreprises innovantes.

Le Groupe ADVANS emploie plus de 1000 ingénieurs dans le monde qui travaillent pour :
- Grandes entreprises internationales.
- PME technologiques [PME : petites et moyennes entreprises].

Le Groupe ADVANS est implanté sur 19 sites dans 7 pays :

FIGURE 2 – Groupe Advans sur le monde

Le Groupe ADVANS est composé de 4 sociétés : TES Electronic Solution, AVISTO, MECAGINE et ELSYS Design.

1.2 Elsys Design

ELSYS Design, créée en 2000 par deux ingénieurs nommés François AGNETTI et Radomir JOVANOVIC, est une société d'ingénierie spécialisée dans la conception de systèmes électroniques. Les

compétences d'ELSYS Design couvrent toutes les étapes de la conception matérielle, du développement de logiciels embarqués et de la conception de systèmes, de la faisabilité à la validation finale des systèmes complets. L'entreprise a réuni une équipe d'ingénieurs spécialisés dans tous les domaines de la conception de systèmes électroniques :

- Architecture matérielle et logicielle pour cartes et calculatrices.
- Électronique de puissance, analogique.
- Conception ASIC, FPGA (*FPGA : matrice de portes programmable sur site*) et IC (*IC : circuit intégré*)
- Développement de logiciels embarqués
- Développement de logiciels en temps réel
- Bancs d'essai et intégration

ELSYS Design est certifié ISO-9001 : 2008.
Depuis sa création, l'entreprise a défini la R&D comme un axe stratégique majeure. Il a été récompensé en 2002 par le "Innovative Award" décerné par l'organisme public français ANVAR. L'entreprise appartient également au réseau "Bpifrance Excellence".

ELSYS Design propose trois types de services :

- Assistance technique et conseil : les ingénieurs rejoignent les équipes de développement client et travaillent sur leurs projets.
- Projet à prix fixe : une équipe composée d'ingénieurs et d'experts supervisés par des chefs de projet qui livrent un projet complètement défini pour une période, un coût et une qualité déterminée.
- Partenariat client : le centre dédié (équipe R&D externe) combine les compétences et l'expertise d'ELSYS Design avec les méthodologies client.

Elle est implantée dans plusieurs pays et possède de nombreux centres techniques dans :

- France : Paris, Rennes, Nantes, Grenoble, Lyon, Sophia Antipolis, Aix en Provence et Toulouse.
- Serbie : Belgrade et Novi Sad.
- États-Unis : Santa Clara.

1.3 Les membres concernés

Ils sont les personnes que j'ai travaillé avec au cours de mon stage de 6 mois.

Marc Renou	Business Manager
Adrien Pagliano	Ingénieur d'Affaires
Olivier Parra	Ingénieur
Gael Guegan	Ingénieur

2 Description du système de contrôle de bras robotique

2.1 Description du système

Le projet se concentre sur le développement d'un programme de contrôle pour le bras robotique Lynxmotion AL5D en respectant les exigences industrielles telles que la modularité, la pratique pour la maintenance, l'extension et le débogage. Donc, l'architecture du programme est conçu en divisant aux parties différentes qui jouent des rôles spécifiques. Pour mieux comprendre le fonctionnement du système, le schéma ci-dessous décrit complètement la structure du programme et aussi toutes les relations entre ses modules.

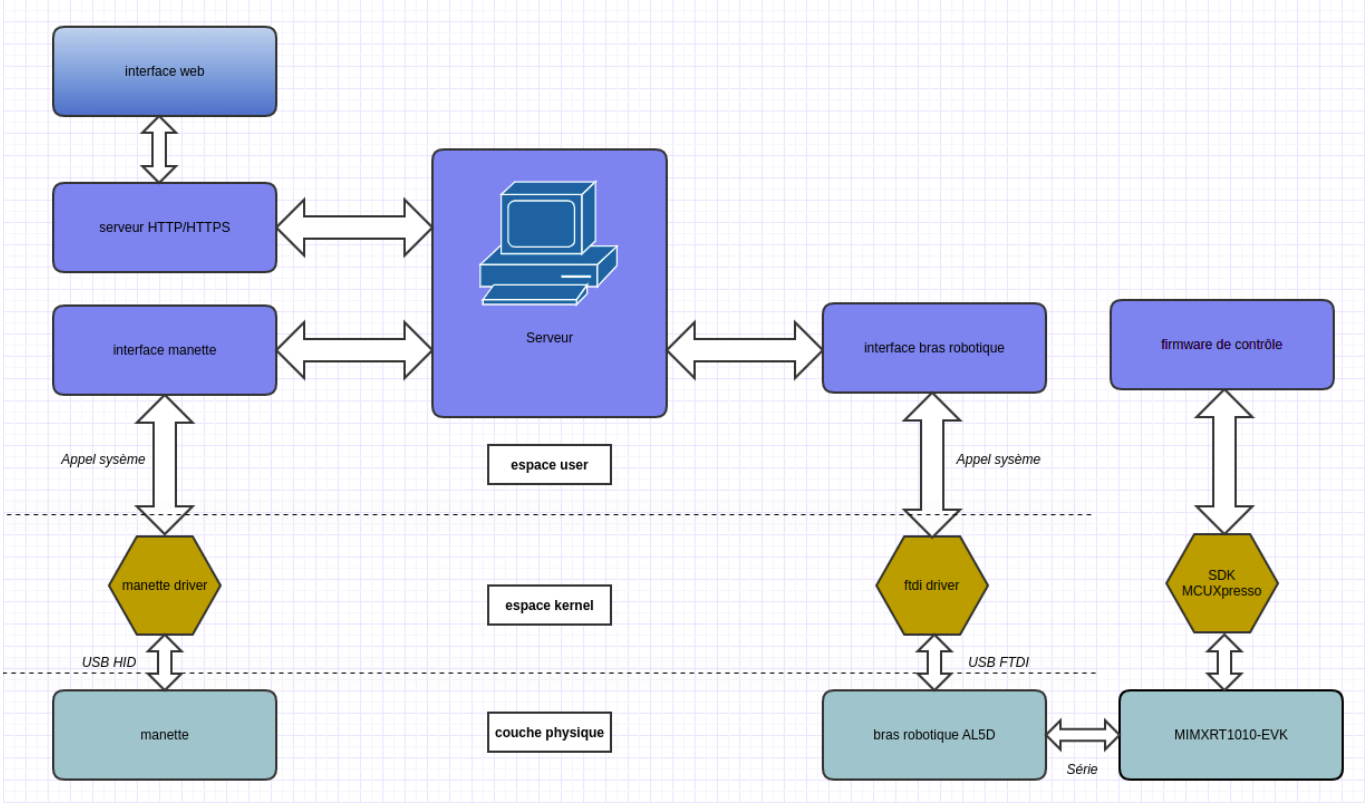

FIGURE 3 – Vue ennsemble de l'architecture du système

En regardant la figure 2, les blocs en bleu sont les parties que j'étais responsable de développer, alors que les blocs marron soient le paquet développé par le fabricant NXP et aussi les drivers déjà intégrés dans les systèmes Linux. La couche physique présente les matériels principaux utilisés dans ce projet à côté d'une machine avec système opérateur Linux tel que le PC ou le Raspberry PI.

Dans les 2 premières méthodes, j'ai effectué le contrôle sur une machine avec OS Linux ce qui reconnaît la manette et le bras robotique grâce aux drivers USB déjà intégrés ou par l'installation du driver à partir le code source ouverte sur l'Internet. En haut niveau, le logiciel de contrôle communique avec le matériel via l'interface appel système au milieu qui est un pont entre l'espace d'utilisateur et l'espace du noyau. Les demandes sont envoyées par la manette ou à travers l'interface web. Le serveur ensuite les analyse et construit de commandes finales afin de transférer à l'interface de bras robotique.

La connexion entre la machine hôte et la manette est le protocole USB avec la classe HID (human interface device) alors que le protocole USB FTDI (la classe spécifique de l'entreprise FTDI pour convertir de USB à UART) soit appliqué dans la connexion entre la machine hôte et le bras robotique.

Dans le cas dernier, le contrôle est réalisé par la carte MIMXRT1010-EVK où j'ai développé un firmware basé sur le SDK MCUXpresso (software development kit). La carte utilise le protocole UART pour la communication en combinant avec le FreeRTOS afin de profiter sa fonctionnalité de multitâche.

2.2 Equipements

2.2.1 Bras robotique Lynxmotion AL5D et la carte contrôle de servos SSC-32U

Lynxmotion AL5D est un robot qui comprend 6 moteurs de servo correspondant les 6 articulations (la base, l'épaule, le coude, le poignet, la pince et le poignet de rotation) et 4 degrés de la liberté. Il est connecté avec la carte contrôle de servos SSC-32U qui fonctionne à l'aide du processeur ATmega168. Cette carte nous permet de contrôler simultanément jusqu'à 32 moteurs de servo via nombre de protocoles différents tels que XBEE, USB, Serial (TX, RX, GND). La carte est mise en place 2 leds pour signaler les événements reçus et transférés des données sur le convertisseur FT232RL de USB à UART et aussi 2 leds pour la validation des données reçues dans le processeur ATmega168 à partir du convertisseur. Ce sont les fonctionnalités intégrées qui aident des développeurs à reconnaître la position des erreurs d'une façon plus visible, plus facile dans le processus de débogage. La puce FT232RL intégrée sur la carte sera alimentée et l'ordinateur pourra détecter et installer le pilote correspondant. L'ordinateur ne fournit pas assez de puissance pour la puce ATmega principale, elle doit donc être alimentée séparément via VS1 avec la tension plus 5.3V.

(a) Bras Lynxmotion AL5D

(b) Carte SSC-32U

FIGURE 4 – Architecture de bras robotique

2.2.2 Microcontrôleur Raspberry PI (ARM)

Dans le cadre du projet, j'ai utilisé le Raspberry PI 3 Model B+. Il s'agit d'un tout petit appareil faisant office d'ordinateur et ayant la forme d'une seule carte mère simple. Il fonctionne sur le processeur ARM et exécute un système d'exploitation Linux.

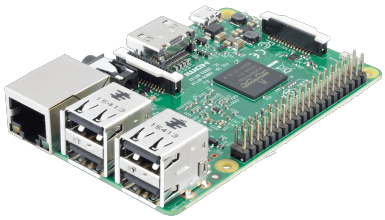

FIGURE 5 – Raspberry PI 3

Le Raspberry PI est composé d'un processeur ARM (ARM1176JZF-S core CPU 700 MHz), 512 MB de RAM, 2 portes USB 2.0, la sortie HDMI et LCD, le prise audio 3.5 mm, la carte de stockage SD, la porte Ethernet (RJ45) et les GPIOs. Normalement, le Raspberry PI est installé le Debian (une distribution Linux) qui est une version Linux complète. Pourtant, l'objectif est de sortir la zone confort et de m'entraîner la nouvelle technologie, mon travail donc était de créer une distribution Linux spécifique avec le Yocto Project. Il me fournit des outils pour concevoir une distribution qui peut exécuter sur un système avec une architecture matérielle limitée. Un serveur SSH est implanté ensuite pour simplifier la procédure du développement en contrôlant le Raspberry via le réseau interne Elsys-Design.

2.2.3 Microcontrôleur MIMXRT1010-EVK

MIMXRT1010-EVK utilise le microprocesseur ARM Cortex M7. C'est une carte à faible coût, pourtant, elle peut être utilisée dans les applications en temps réel grâce à la possibilité de fonctionner avec le FreeRTOS. Elle est aussi soutiente complètement par le SDK de NXP et surtout par le Zephyr OS (un système d'exploitation RTOS soutient les tas de réseau utilisés souvent dans IoT comme IPv4, IPv6, BLE, MQTT) qui permet cette carte de devenir le choix dans le domaine Internet of Things. Dans une des solutions réalisées, cette carte joue le rôle de contrôler le bras robotique par UART en temps réel.

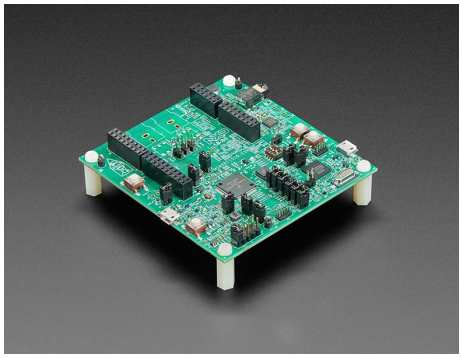

FIGURE 6 – MIMXRT1010-EVK

2.3 Outils

2.3.1 Makefile-Make

Les Makefiles sont des fichiers utilisés par le programme make pour exécuter un ensemble d'actions, comme la compilation d'un projet, l'archivage de document, la mise à jour de site, etc. Dans le processus de développement du logiciel embarqué, le Makefile est très important pour compiler, lier des bibliothèques et générer le fichier exécutable. Pourtant, dans ce cadre de mon projet, à cause de la taille de code source, la tâche d'écrire un Makefile n'est pas le choix optimal et cause de la difficulté lorsque le projet est de plus en plus élargi. Donc la solution alternative est l'outil Cmake. Cmake est un outil de source ouvert multiplateforme utilisée pour contrôler le processus de compilation du logiciel

surtout le logiciel C++. Il prend des paramètres de la configuration, les chemins du code source, le compilateur et générer le makefile en dépendant les entrées définies dans le Cmake. Généralement, le Cmake contient 3 fichiers principaux tel que **CMakeLists.txt**, **CMakeCache.txt** et ***.cmake**.

- CMakeLists.txt est le fichier principal qui récupère les configurations du projet. C'est nous qui devons définir les configurations dans ce fichier pour que le cmake puisse comprendre la description du projet, du compilateur...
- CMakeCache.txt est le fichier généré après que nous lançons le cmake. L'usage du fichier est d'enregistrer toutes les informations sur la configuration du logiciel, le compilateur, le linker dans le temps compilé.
- *.cmake est le fichier qui contient toutes les configurations utilisées par le cmake afin de chercher les bibliothèques dynamiques ou les chaînes d'outils pour la compilation croisée. Sous Linux, nous pouvons récupérer ce fichier dans le système de fichiers lorsqu'on télécharge la bibliothèque.

Sous Linux, sur le terminal, la ligne de commande suivante est utilisée pour lancer le cmake sur la répertoire du fichier CMakeList.txt.

```
1 # Lancer cmake
2 $ cmake ${directorie}
```

Listing 1 – ligne de commande pour le cmake

Voir l'annexe pour le détail du cmake.

2.3.2 Script Shell

Le script shell est un fichier spécifique sous Linux qui contient nombre de commandes différentes. Ces commandes permettent de réaliser automatiquement des opérations de manière séquentielle. Dans la phase du développement de logiciel embarqué Linux, j'ai utilisé le script shell pour implanter l'environnement et l'espace du travail en téléchargeant des bibliothèques, créant l'arborescence de répertoires et lancer la compilation avec le cmake et le make à la fin. Script Shell donc simplifie la tâche de construire un programme et de synchroniser l'espace de travail des développeurs.

2.3.3 Logiciel Wireshark

Le problème de la communication entre des appareils existes toujours dans le processus du développement de système embarqué. Afin de visualiser et vérifier des paquets des données échanges entre la machine hôte et les autres MCUs ou entre le serveur et le client, le logiciel Wireshark - un logiciel gratuit et performant, est utilisé pour analyser des données transférées, déterminer l'étape qui se produit l'erreur...

Précisément, les commandes envoyées par la machine hôte à la carte SSC-32U via l'USB UART sont contrôlées par le logiciel WireShark pour assurer que les données sont bien transférées à la cible en premier temps. En outre, dans le cas que je contrôle le bras par une interface web via le serveur embarqué, le Wireshark me permet d'observer le processus de la connexion serveur-client par le HTTP/HTTPS - le processus de la poignée de main qui implante la communication avec ou sans sécurité et aussi de vérifier si les données sont bien cryptées et décryptées avec la clé secrète partagée ou non. Le logiciel Wireshark joue un rôle très important en fournissant aux développeurs un outil efficace pour le débogage surtout dans le secteur de système embarqué et cyber-sécurité.

2.3.4 Doxygen

Doxygen est l'outil standard pour générer de la documentation à partir de sources C annotées. Avec l'objectif de l'héritage de code source pour les prochains développeurs, j'ai commenté le code avec un bloc de commentaires spéciaux qui est un bloc de commentaires de style C ou C ++ avec quelques marquages supplémentaires compris par le Doxygen. Grâce au Doxygen, la documentation est générée automatiquement au lieu de la rédaction traditionnelle des documentations. Avec cette documentation, la tâche de comprendre l'usage des fonctions, des paramètres, des fichiers de code source, des variables... et la structure de code source devient de plus en plus significativement facile.

```
/**
* @brief this method enumerate all subsystem device connect
* @param subsystem where search for device e.g:input
* @return true/false for success or fail
*/
bool Enumeration(const char* subsystem);
```

Listing 2 – Bloc de commentaires

Les marquages sont déjà définis par Doxygen et accompagnons le symbole @ devant. Par exemple, le marquage **@brief** est utilisé pour décrire précisément l'usage de la fonction **Enumeration** qui sera généré à côté de cette fonction dans la documentation finale. Le Doxygen fournit un fichier de script **Doxyfile** qui contient toutes les configurations qui servent à chercher le code source, à mettre en place l'interface, la structure du document... Sous Linux, je peux générer la documentation par la ligne de commande :

```
# generate doc
$ doxygen [path of doxyfile]
```

Listing 3 – ligne de commande pour le doxygen

2.3.5 Intégration continue Travis

Dans le cycle de développement, lorsque nous faisons des changements sur le code source, il risque toujours que le code source ne peut pas être construit ou plein d'erreurs apparaissent. En outre, le temps nécessaire pour réaliser tous les tests après chaque changement devient gaspillage et inefficace. Donc nous avons besoin d'un outil qui nous permet de tester, notifier des erreurs à tous les membres de l'équipe pour déterminer le coupable à l'origine de l'erreur... d'une façon automatique et continue.

En tant que plate-forme d'intégration continue, Travis CI prend en charge le processus de développement en créant et en testant automatiquement les modifications de code, fournissant un retour immédiat sur la réussite de la modification. Travis CI peut également automatiser d'autres parties du processus de développement en gérant les déploiements et les notifications.

En combinant Travis CI avec Github, le management du projet devient plus simple. Toutes les actions sur le Github (commit, merge) notifient Travis CI à faire un retour correspondant. Dans mon projet, le comportement du Travis CI est géré par le fichier .travis.yml placé dans le répertoire racine qui a la forme suivante :

- Définir la machine, son système d'exploitation, version du compilateur
- Créer des tâches différentes à réaliser sous quelques conditions (type des actions sur git, branche).

— Mise en place l'environnement en téléchargeant des dépendances, choisissant langage de programmation.
— Définir les actions pour chaque tâche (construire le code source, réaliser des tests unitaires, déployer la documentation).

3 Détails techniques des solutions de commande

Ce chapitre décrit complètement des solutions que j'ai effectuées dans mon stage. Avant d'aller dans les détails de chaque partie, d'abord, nous devons comprendre la conception du bras Lynxmotion AL5D et aussi les soutiens existants développés par le fabricant.

3.1 Conception et soutien technique du bras robotique

Comme la présentation du bras robotique dans le chapitre dernier, le bras Lynxmotion AL5D est composé 6 moteurs de servo qui sont contrôlés par la carte de contrôle SSC-32U. Puisque AL5D est un kit, donc le nombre des fiches techniques sont très limitées et la fiche la plus importante est fournit sur le site web officiel concernant la carte SSC-32U. La chose le plus intéressant est la manière de communiquer avec cette carte pour qu'elle puisse comprendre les commandes et envoyer ensuite les signaux correspondants au bras.

Avec l'architecture du matériel de la carte SSC-32U, elle peut accepter 3 types de connexion à partir des contrôleurs : USB, UART et XBEE. Dans le cadre du projet, j'ai utilisé le protocole USB dans le cas où une machine Linux joue le rôle d'hôte qui envoie des commandes à la carte, car les machines Linux n'ont que les portes USB. Malgré les broches GPIO sur le Rapsberry PI, le USB est le meilleur choix lorsqu'il est un protocole plus performant et plus stable. Par contre, la connexion UART est utilisé dans le cas où le MCU IMXRT1010-EVK est responsable d'envoyer les commandes. Théoriquement, je peux utiliser le MCU IMXRT1010-EVK contrôle la carte SSC-32U, pourtant, le IC FT232RL situé sur la carte (USB UART Integrated circuit) qui est responsable de transférer l'ensemble des données sous format de USB à UART, est un appareil spécifique de l'entreprise FTDI. Donc, la norme USB utilisée n'est pas le même avec les normes normales sur le marché comme HID, CDC, MSC... En outre, l'entreprise FTDI protège son code source, alors la tâche de créer mon propre driver est une tâche très difficile qui exige une approfondie connaissance sur le protocole USB.

La carte de contrôle SSC-32U est déjà intégrée un firmware qui est responsable de définir quelques commandes sous des formats déterminés et réaliser la mission de tourner des moteurs de servo sur le bras après avoir analysé la commande reçue. Pour contrôler des moteurs du bras, cette carte va récupérer la broche dont nous avons envie de contrôler déclarée dans la commande et ensuite créer une modulation de largeur d'impulsion (PWM) sur cette broche. Il y a différents types de la commande acceptée par la carte, car elle est une carte générique pour contrôler différents types de robots (bras, hexapodes...). Néanmoins, il y a une seule commande utilisée dans ce cas-là :

```
1 #<ch>P<pw>S<spd>T<time><cr>
```

Listing 4 – commande utilisée pour contrôler le bras AL5D

— <ch> est le canal où le PWM est généré. Le numéro de canal est marqué sur la carte SSC-32U de 0 à 31 correspondant le numéro des broches physiques. Dans le cadre de mon projet, il y a 6 broches utilisées pour 6 articulations avec les indices comme ci-dessous :

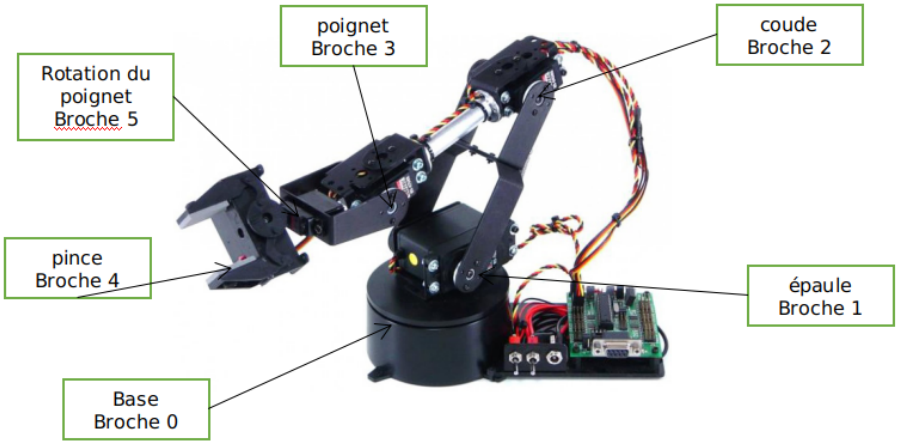

FIGURE 7 – Les broches utilisées par les articulations du bras

— <pw> est la largeur d'impulsion souhaitée, normalement, la valeur est réglée entre 500 et 2500 μs. Une impulsion de 500 μs à 5 V réalise une rotation de -90° et une impulsion de 2500 μs à 5 V réalise une rotation de + 90°. Pour garder la position actuelle du bras, l'impulsion doit être répétée chaque 20 ms. Nous pouvons comprendre plus simplement que ce facteur présente la position voulue de chaque articulation.

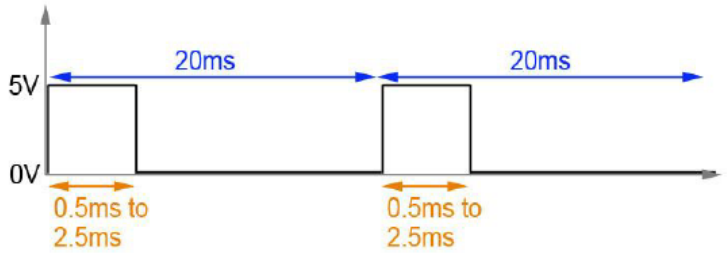

FIGURE 8 – Modulation de largeur des impulsions pour contrôler le moteur de servo

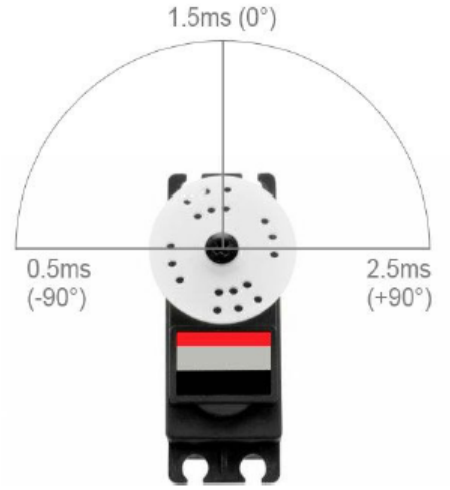

FIGURE 9 – Angle de fonctionnement des moteurs de servo standards

— <spd> vitesse de déplacement du servo en microsecondes par seconde. Par exemple, une vitesse de 2000 μs permet le moteur de tourner un angle de 90° pendant 0.5 second, ou 100 μs permet de tourner un angle de 90° pendant 10 seconds.
— <time> temps en microsecondes pour se déplacer de la position actuelle à la position souhaitée. Pour une chaîne de la commande, le temps doit être déclaré une seule fois et ce temps détermine le temps nécessaire pour tous les mouvements dans la chaîne (par exemple : avec la commande #5 P1600 #17 P750 S500 #2 P2250 T2000 <cr>, le temps 2000 ms est déclaré pour le mouvement entier).
— <cr> est le **carriage return** correspondant le code 13 dans le tableau ASCII. Chaque commande doit être finie par ce caractère pour être exécutée.

Exemple : **#5 P1600 #10 P750 S1000 <cr>**

L'exemple au-dessus déplace le servo 5 à la position 1600, le servo 10 à la position 750, avec la vitesse de 1000 μs par seconde (il prend 1 seconde pour une rotation de 90°).

En conclusion, grâce au firmware intégré dans la carte SSC-32U, ma mission est d'envoyer les commandes sous forme ci-dessus à l'entrée de la carte de contrôle.

3.2 Logiciel du contrôle de bras robotique par la manette de jeu

La première solution est d'utiliser la manette pour contrôler le bras robotique AL5D. Le centre de traitement entre les 2 appareils est une machine Linux (un PC ou un Raspberry PI). Cette méthode concentre la plupart du temps sur l'aspect du développement de logiciel Linux embarqué.

3.2.1 Architecture

A côté de l'objectif de contrôler le bras, la condition préalable du programme est la modularisation de ses parties qui nous permet la maintenabilité, la réutilisabilité et la capacité d'adaptation. Pour ce faire, je me suis concentré sur le principe d'abstraction utilisée souvent dans la programmation C++. Ce principe nécessite de passer d'une instance spécifique à un concept plus général en pensant aux informations et fonctions les plus élémentaires d'un objet. C'est-à-dire, j'analyse les informations fondamentales du bras robotique et du joystick pour construire 2 objets qui se présentent toutes les instances similaires. Le programme doit être construit à partir des briques et chacune représente un élément (un objet en C ++). Les briques peuvent être décrites comme une boîte générique avec une simple entrée / sortie. Elles réalisent des tâches spécifiques grâce à des fonctions publiques qui appellent ensuite des algorithmes plus complexes se cachent aux yeux de l'utilisateur. Ces algorithmes adaptent, contrôlent et interprètent les données entrées et sorties. Si un élément doit être modifié ou corrigé, seule la brique correspondante doit être modifiée. La définition de l'abstraction peut sembler simple, mais certains points sont plus inquiétants derrière. En effet, l'objet abstrait doit être défini soigneusement pour éviter le phénomène de passer les étapes inutiles ou de manquer les étapes nécessaires lorsque nous appelons cet objet dans le programme principal ou pour éviter de manquer d'informations ou avoir un élément de la brique dans une autre.

A l'aide de la fonctionnalité de la modularisation, mon programme pour cette méthode peut fonctionner avec n'importe quel joystick ou n'importe quel bras robotique. Ce programme contient 3 briques. Une brique est pour le joystick, une est pour le bras robotique et une pour le lien entre 2 premières briques (brique centrale). En outre, j'ai déterminé aussi les interfaces pour lier les briques avec les simples

entrées et sorties. Cela permet d'ajouter le nouvel appareil après aussi. Ci-dessous est l'architecture globale du programme :

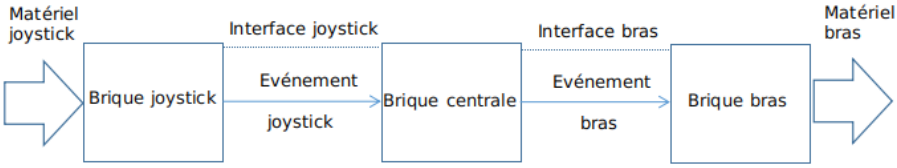

FIGURE 10 – Architecture globale

A côté des classes principales, j'ai développé aussi des classes génériques pour quelques fonctionnalités qui peuvent être hérité par toutes les autres objets. Ces fonctionnalités sont les outils qui soutient les classes principales pour un objectif déterminé. Par exemple, la classe UdevHandler est utilisé par n'importe quelle classe afin de détecter automatiquement les noeuds de l'appareil.

3.2.2 Range de mouvement

En premier temps, il faut déterminer la range du mouvement de chaque articulation en utilisant l'outil gratuit **screen**. Cet outil est un open source qui permet de créer des terminaux virtuels sous Linux. Dans ce cas-là, j'ai utilisé cet outil pour ouvrir un console série avec le débit en bauds 9600 pour communiquer avec la carte de contrôle SSC-32U. En entrant plusieurs fois des commandes sur le terminal, je peux trouver la limite de mouvement pour chaque articulation.

```
1 # open serial console with baud rate 9600
2 $ screen /dev/ttyUSBx 9600
```

Listing 5 – commande utilisée pour ouvrir le console série

Lorsque la carte de contrôle est branchée à la machine Linux, le driver FTDI sur la machine peut détecter la carte de contrôle et créer ensuite un nœud de périphérique (ttyUSBx) dans le répertoire /dev/. Connu comme un fichier de système, l'usage de ce nœud est une interface de communication entre l'espace de l'utilisateur et du noyau. Donc si j'écris des commandes sur le terminal série, ces commandes doivent être transférés à la carte pour contrôler le bras. En outre, au lieu du terminal virtuel, une autre méthode utilisée dans la programmation pour communiquer avec ce nœud est des appels systèmes.

3.2.3 Module de la manette

Pour utiliser une manette comme un contrôleur, il faut connaître les valeurs de configuration des boutons qui permettent de déterminer quel bouton se produit l'événement de contrôle. Au lieu de forcer ces valeurs dans le programme, nous passons un fichier de configuration au programme, il peut donc bien fonctionner avec n'importe quelle manette. Grâce au logiciel **Retro Arch**, je peux exporter un fichier de profile lorsque la manette est branchée sur la machine. Par exemple avec la manette PS3, j'ai le fichier de profile comme ci-dessous :

```
input_driver = "sdl2"
input_device = "Sony PLAYSTATION(R)3 Controller"
input_vendor_id = "1356"
input_product_id = "616"
input_b_btn = "0"
input_y_btn = "3"
input_select_btn = "8"
input_start_btn = "9"
input_up_btn = "13"
input_down_btn = "14"
input_left_btn = "15"
input_right_btn = "16"
input_a_btn = "1"
input_x_btn = "2"
input_l_btn = "4"
input_r_btn = "5"
input_l2_btn = "6"
input_r2_btn = "7"
input_l3_btn = "11"
input_r3_btn = "12"
```

Listing 6 – commande utilisée pour ouvrir le console série

Pour communiquer avec la manette à partir la machine hôte, j'ai utilisé la bibliothèque **SDL2** donc le driver que j'ai choisi dans le Retro Arch est SDL2. De plus, la partie la plus importante dans le fichier est la description des entrées qui présente l'étiquette des boutons et sa cartographie. Par conséquent, une fonction est développée qui permet de recevoir la configuration en lisant le fichier de système et sortant la valeur des étiquettes. Les étiquettes sont nommées par la convention du logiciel Retro Arch pour la plupart des manettes :

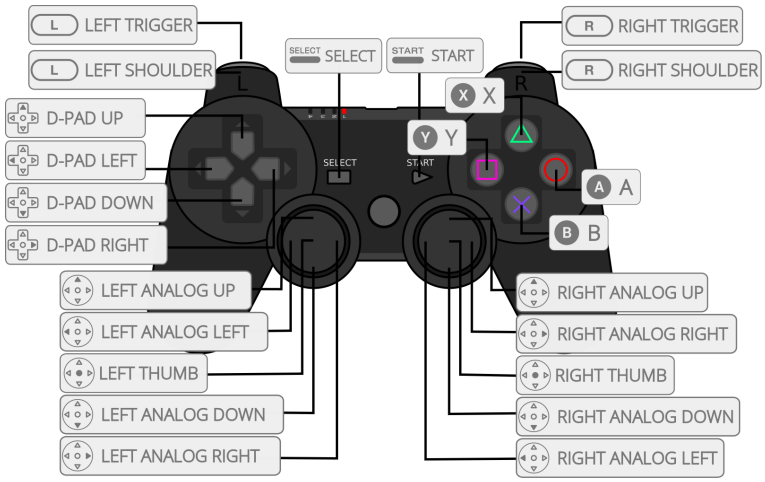

FIGURE 11 – Convention des étiquettes sur la manette

Dans l'architecture Linux, sous-système d'entrée est une collection des drivers pour prendre en charge tous les périphériques d'entrée. Ce sous-système fournit les modules différents pour communiquer avec le matériel, obtenir les événements, etc. Les modules communs dédiés à la manette est evdev et joydev. De plus, le SDL2 est une API qui se situe au-dessus de ces modules et fournit l'accès bas niveau au matériel branché sur la machine. 2 options possibles pour communiquer avec le joystick est d'utiliser le SDL2 ou m'investir sur le evdev dans la couche en bas et j'ai choisi la première option, car le SDL

fournit plusieurs fonctionnalités qui me permettent de simplifier une tâche, mais aussi de maintenir la performance du logiciel.

Avant d'accéder le matériel par le SDL2, il faut initialiser un sous-système correspondant le matériel branché (dans ce cas-là c'est la manette). Après avoir initialisé le sous-système, la manette est balayée et les drivers nécessaires pour l'appareil sont chargés pour que le SDL2 puissent les appeler. Au lieu d'interagir directement le nœud jsx dans le répertoire de système /dev/input/ (ce nœud est un fichier de système qui joue le rôle comme le /dev/ttyUSBx pour le bras), nous pouvons ouvrir une structure abstraite utilisée pour identifier la manette connectée par son index et le SDL2 va utiliser le pointeur sur cette structure pour garder les informations et communiquer avec le matériel dans la couche en bas. Ensuite, après avoir attaché la manette, le mécanisme pour prendre les événements de la manette utilisée par le SDL2 est d'attente active. Il vérifie de façon répétée si l'événement sur ce pointeur est activé. Si un événement apparaît, fonction d'attente active retourne à la structure SDL_Event les informations concernant le dernier changement d'état de l'appareil (si aucun changement d'état n'est ajouté, la fonction ne retourne rien).

```
typedef union SDL_Event
{
    Uint32 type;                    /**< Event type, shared with all events */
    SDL_JoyAxisEvent jaxis;         /**< Joystick axis event data */
    SDL_JoyBallEvent jball;         /**< Joystick ball event data */
    SDL_JoyHatEvent jhat;           /**< Joystick hat event data */
    SDL_JoyButtonEvent jbutton;     /**< Joystick button event data */
    SDL_JoyDeviceEvent jdevice;     /**< Joystick device change event data */
} SDL_Event;
```

Listing 7 – structure de l'événement SDL2

- type : est le type d'événement apparaît. Il est présente par les MACROs définis comme SDL_JOYAXISMOTION, SDL_JOYBUTTONDOWN, SDL_JOYBUTTONUP, SDL_JOYHATMOTION. Les événements sur la manette sont séparés en 4 parties différentes lorsque nous interagissons avec ses boutons, ses chapeaux et ses boules de commande.
- jaxis : est la structure qui contient les informations de l'événement si SDL_JOYAXISMOTION se produit. À partir de la structure, ses éléments indiqués quel axis l'événement se produit (élément axis) et sa valeur (élément value) permettent de construire la commande correspondant au plus tard.
- jball : est la structure qui contient les informations de l'événement si SDL_JOYBALLMOTION se produit. Dans le cadre de mon projet, cet événement n'est pas utilisé.
- jhat : est la structure qui contient les informations de l'événement si SDL_JOYHATMOTION se produit. Comme jaxis, jhat permet de déterminer le chapeau qui fait l'événement (élément hat). La manette Nintendo Switch contient quelques chapeaux, mais la manette PS3 ne les contient pas. Dans cette méthode, j'ai utilisé toutes les 2 manettes, donc il faut prendre en compte l'événement sur le chapeau.
- jbutton : est la structure qui contient les informations de l'événement si SDL_JOYBUTTONDOWN et SDL_JOYBUTTONUP comme l'état du bouton (élément state), l'indice du bouton (élément button).
- jdevice : est l'instance de la manette choisie.

Chaque fois un événement est détecté, il va passer une fonction pour que le logiciel puisse reconnaître l'origine de l'événement et le type de l'événement. Avec ces informations, il met le bit correspondant le bouton/ le chapeau pressé à 1 ou celui relâché à 0 dans un masque. Le masque est une chaîne de bits et chaque bit est attaché à un bouton. L'indice de chaque bit présente un mouvement sur le bras. Grâce à ce masque, un autre module dans le programme peut établir une commande correspondante

sur une articulation.

```
/**
 * @brief mask of all joystick's buttons
 */
typedef enum {
  mask_btn_south       = 0b1 << elbow_right,      /*!< mask of button south */
  mask_btn_east        = 0b1 << wrist_right,      /*!< mask of button east */
  mask_btn_north       = 0b1 << elbow_left,       /*!< mask of button north */
  mask_btn_west        = 0b1 << wrist_left,       /*!< mask of button west */
  mask_btn_tl          = 0b1 << gripper_open,     /*!< mask of button tl */
  mask_btn_tr          = 0b1 << gripper_close,    /*!< mask of button tr */
  mask_btn_select      = 0b1 << option1,          /*!< mask of button select */
  mask_btn_start       = 0b1 << all_home,         /*!< mask of button start */
  mask_btn_thumbl      = 0b1 << wrist_rot_left,   /*!< mask of button thumbl */
  mask_btn_thumbr      = 0b1 << wrist_rot_right,  /*!< mask of button thumbr */
  mask_btn_dpad_up     = 0b1 << shoulder_right,   /*!< mask of button dpad up */
  mask_btn_dpad_down   = 0b1 << shoulder_left,    /*!< mask of button dpad down */
  mask_btn_dpad_left   = 0b1 << base_left,        /*!< mask of button dpad left */
  mask_btn_dpad_right = 0b1 << base_right       /*!< mask of button dpad right */
}E_BTN_MASK_PS3;
```

Listing 8 – Masque des boutons

En outre, j'ai ajouté une comparaison des états des boutons, des chapeaux entre 2 boucles consécutives pour connaître s'ils sont bien relâchés ou sont pressés toujours. L'objectif est d'envoyer une seule fois la commande pour un tour de pression/relâche. En conclusion, ce module va attacher une manette branchée, interpréter ses configurations, prendre des événements et créer le masque et détacher la manette si le logiciel termine. l'opération du module de la manette est présenté en détail comme l'organigramme ci-dessous :

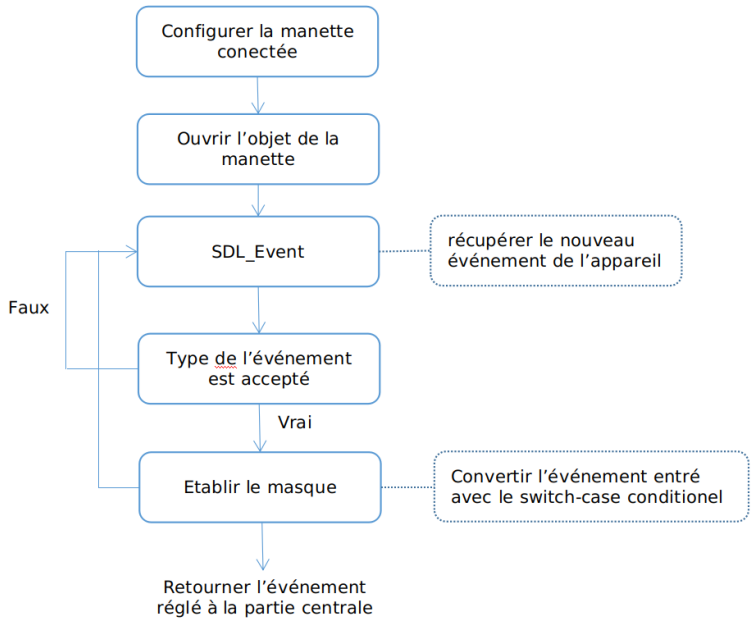

FIGURE 12 – Organigramme du module de manette physique

En outre, il y a aussi une fonctionnalité de la gestion du branchement et du débranchement à chaude avec les événements fournis par le SDL2. Donc, SDL2 soutien non seulement les événements sur les mouvements des boutons, des axes mais aussi ceux des états du joystick. Avec 2 états qui indiquent

si le joystick est vient d'être branché ou débranché sur la machine, je peux suspendre le processus de lire les mouvements sur les boutons, les axes et mettre ce module dans l'état de scanner le joystick. Si le module trouve un joystick branché, il peut sauter dans le processus d'initialisation et redémarrer le processus de prendre les commandes.

3.2.4 Module de bras robotique

Comme le module de la manette, pour ce module, j'ai créé une classe pour le bras robotique AL5D. Cette classe interagit directement avec le matériel via le driver FTDI. D'abord, pour rappel, après avoir branché le bras robotique au PC, il apparaît un noeud dans le répertoire de système /dev/ qui crée une interface entre l'espace de l'utilisateur et du noyau. En premier temps, lorsque j'ai fait le PoC (proof of concept), j'ai fixé le chemin pour trouver le noeud. Ensuite, grâce à l'appel de système IOCTL, je peux ouvrir le descripteur de fichier et envoyer les commandes au fichier. Ces commandes ensuite sont transférées au bras. Pour noter, un descripteur de fichier est un indicateur abstrait ou un gestionnaire utilisé pour pointer un fichier système ou une autre ressource d'entrée / sortie (le noeud). Pourtant, si je force un chemin défini du noeud, ce n'est pas du tout une solution pratique. Puisque le noeud de l'appareil est créé d'une façon différente pour les machines différentes et les conditions actuelles sur la machine.

Donc, afin d'assurer que le logiciel peut trouver le bon chemin, j'ai cherché une méthode pour tracer le chemin du noeud en utilisant les priorités intégrées sur le bras. L'idée a une relation intime avec l'usage de **udev** - un démon dans l'OS Linux. Udev est un sous-système qui permet d'obtenir et définir un appareil périphérique branché sur la machine. Avec ce démon, pour créer un usage d'un appareil, il nous faut quelquefois de définir une règle de fonctionnement situé dans le répertoire **/etc/udev/rules.d** lorsque cet appareil est branché. Avant d'appliquer les règles sur l'appareil, udev détecte l'appareil par ses caractéristiques. Par conséquent, avec le même objectif de détection, je peux utiliser la bibliothèque **libudev** dans mon logiciel pour hériter les fonctionnalités de udev.

Avec l'objectif de créer le module qui permet la réutilisation, j'ai créé une classe **UdevHandler** - une classe détecte automatiquement un noeud avec les propriétés spécifiques. Cette classe utilise la bibliothèque libudev qui fournit les APIs pour énumérer les périphériques sur le système local (cette classe en combinant avec la bibliothèque libevdev peut remplacer l'API SDL2 dans le module du joystick. J'ai essayé de le faire, mais au final l'API SDL est utilisée pour le logiciel principal). Chaque logiciel a besoin d'hériter cette classe pour chercher le noeud d'un matériel, il faut fournir le type du sous-système et les propriétés spécifiques du matériel. Pour noter, tous les matériaux contiennent les caractéristiques différentes pour s'identifier tel que MODEL_ID (identifiant du produit), VENDOR_ID (identifiant du vendeur)... Ces facteurs sont constants et définis par le fabricant et évidemment, nous n'avons pas le droit de les modifier. Nous avons plein de façons pour chercher ces informations par exemple : dmesg, udevadm, lsusb... Grâce à ces propriétés, la fonction pt2FindDevnode (Fig. 11) peut appeler l'API CheckProperty de libudev pour reconnaître le noeud. Le processus de la détection en utilisant le udev est divisé en 3 parties principales :

— Initialiser l'objet udev.
— Énumérer les appareils dans le sous-système.
— Récupérer le noeud en vérifiant ses priorités.

La figure ci-dessous présente en détail la mission de la classe UdevHandler

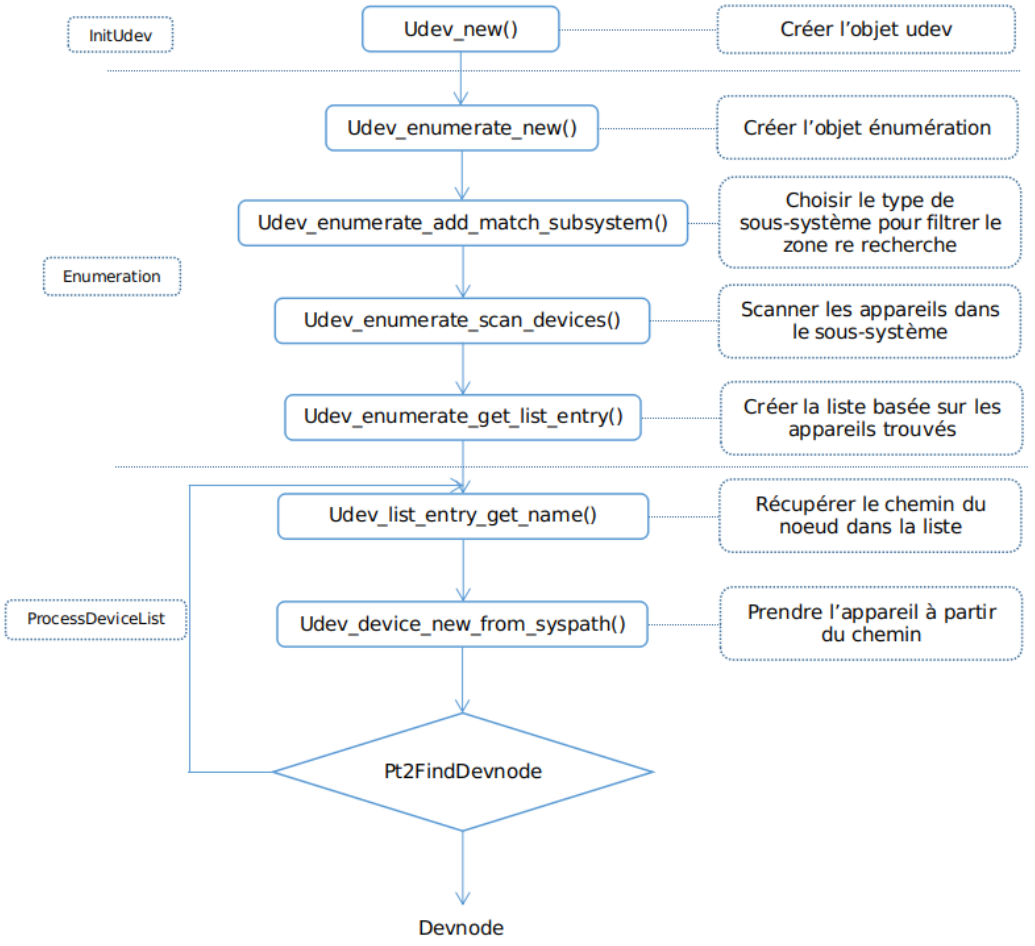

FIGURE 13 – Organigramme de l'algorithme pour détecter le noeud de l'appareil

Pour assurer que le programme fonctionne proprement, je dois résoudre le problème concernant le branchement. J'avais 2 choix : régler le branchement à chaude ou forcer l'utilisation du branchement à froid. Au final, j'ai choisi la deuxième solution malgré la possibilité de superviser le branchement à chaude avec l'API de la bibliothèque libudev, puisque ce n'est pas si nécessaire de m'investir trop de temps sur un objectif qui n'est pas dans les exigences du projet. Si j'ai assez de temps au final du stage, je vais améliorer la fonctionnalité de gestion du branchement et du débranchement à chaude.

Après avoir trouvé le noeud du bras robotique sur la machine, la suivante mission est d'obtenir les événements réglés dans le module de la manette, d'établir la commande et de l'écrire sur le fichier de système (noeud). L'algorithme utilisé dans ce cas-là est l'opération au niveau du bit (bitwise) pour extraire les données du registre et de la valeur de la table. Tel qu'indiqué dans la dernière section, le module de manette créée un masque pendant chaque période qui contient les bits correspondants aux événements sur la manette. Donc, le module du bras doit déterminer quel bit est forcé à 1 dans le masque pour comprendre la demande de contrôle de la manette.

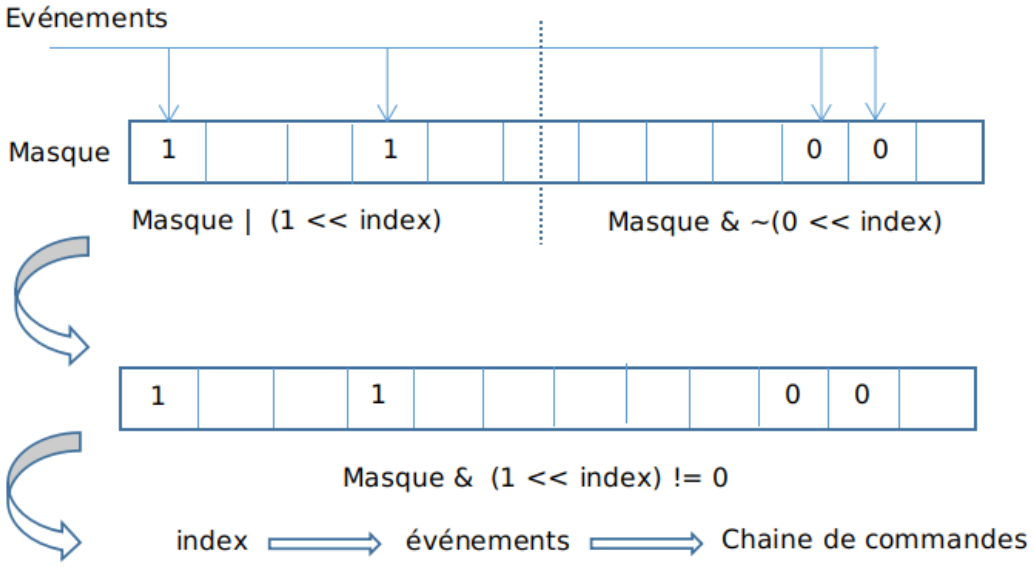

FIGURE 14 – Le processus d'analyser l'événement avec l'opération au niveau du bit

Avec ce masque, la partie du bras peut reconnaître et distinguer les événements. Ensuite, chaque information obtenue est utilisée pour construire une chaîne (string) avec le format présenté dans la section 3.1 en déterminant quelle articulation choisie et la direction de mouvement. A côté des mouvements événement sur les boutons logiques (2 états : pressé, relâché), le module du bras analyse aussi le masque pour les axes de la manette. Chaque fois nous interagissons les axes, le module de la manette fournit non seulement le masque pour identifier l'axe, mais aussi la valeur absolu varié de -32,767 à +32,6767. Cette valeur est converti à la vitesse de mouvement du bras suivant un ratio et ensuite est mis dans la commande. Dans la classe du bras AL5D, j'enregistre les positions actuelles de chaque articulation en mode privé. Lorsque je construis le string de commande, basé sur la direction, les positions sont mises à jour en ajoutant ou diminuant un pas constant et sont comparées avec les limites de mouvement. Après avoir construit la commande, ce module pousse ces commandes dans un vecteur de string. L'avantage de ce vecteur est de permettre de sauvegarder dynamiquement toutes les commandes pour envoyer sans perdre aucune commande. Par la suite, je fais une boucle sur le vecteur pour exporter une commande finale en chaînant tous les éléments dans le vecteur. Grâce à l'appel de système, ces commandes sont transférées à la carte de contrôle SSC-32U. Sous Linux, les requêtes IOCTL sont le pont entre le logiciel dans l'espace de l'utilisateur et les fichiers de système dans l'espace du noyau. Retourner à la fonction d'initialisation du bras, après avoir trouvé le chemin du noeud, je peux appeler le requête **open** pour ouvrir le descripteur du fichier sur ce noeud avec la droite de l'écriture et de la lecture. Ce descripteur de fichier est enregistré en mode privé de la classe. Lorsque le module du bras finit de gérer les événements et exporter la commande, j'utilise le requête **write** pour écrire la commande sur ce descripteur du fichier. C'est-à-dire que la commande est bien transférée à la carte de contrôle.

```
1 // open file descriptor on the node of robotic arm
2 fd_ = ::open(deviceNode_.c_str(), O_RDWR);
3 // Transfer the command to controller board with request write
4 write(fd_, finalCmd.c_str(), finalCmd.size());
```

Listing 9 – le requête IOCTL

3.2.5 Les interfaces

Dans l'informatique, une interface est une classe virtuelle ou abstraite qui permet de normaliser les fonctions possibles pour les entrées / sorties entre deux parties du programme. Dans l'industriel, c'est très pratique si nous définissons les fonctions de communication entre 2 parties du programme puise que grâce à ces fonctions, si les 2 parties respectent les formules des fonctions, ils peuvent travailler ensemble sans aucune rayure, aucun problème. En outre, il y a plein d'avantages de l'interface, par exemple, grâce aux interfaces, lorsque nous modifions une partie, cela n'influence pas sur autre partie puisque nous respectons toujours la formule des interfaces, ou la tâche d'ajout la nouvelle version d'une partie devient plus facile et pratique (C'est-à-dire que la partie est élargie pour soutenir le nouveau matériel dans le même classe). Dans un projet industriel, si nous bien séparons les modules différents et définissons l'interface au début du projet, l'efficace de travail augmente vraiment, car tous les membres du groupe peuvent travailler simultanément sur les différents modules sans aucune peur de mal fonctionner avec autre partie des autres membres. Donc l'interface est très important dans mon projet pour que les briques de la manette, du bras robotique et du logiciel principal puissent fonctionner de manière stable et avoir la possibilité d'ajouter les parties afin de soutenir les nouveaux matériaux comme des manettes, des robots différents que ceux existants dans le logiciel actuel.

En premier temps, j'analyse l'interface du bras robotique pour la communication entre la brique central et la brique du bras. Dans ce cas, l'interface utilise une seule variable - **arm-event** pour l'entrée et la sortie entre les 2 parties du programme. La variable arm-event est un élément essentiel dans le programme qui évolue au fil de la progression du codage. Au début, j'ai créé une structure qui combine directement les événements de la manette qui ne sont pas encore traités. Mais le problème est la structure n'est utilisée que pour les manettes connectés avec la machine Linux via le protocole USB mais n'est pas compatible avec les autres manettes comme manette Bluetooth. Donc, j'inspire le principe de le registre pour créer une structure arm-event qui évite les inconvénients de version ancienne.

```
/**
 * @brief struct of status of events on joystick
 */
typedef struct {
  int32_t BtnStatus;      /*!< status of pressed button */
  int32_t AbsStatus;      /*!< status of joystick axis event */
  int32_t valueAbs[6];    /*!< joystick axis for 6 joints */
}arm_event;
Les
```

Listing 10 – La structure arm-event

la structure contient 2 registres de l'état des événements de la manette : un pour les boutons et un pour les axes. Le dernier tableau est utilisé pour contenir les valeurs des axes qui se présentent la vitesse de mouvement des articulations. Chaque bit dans le registre est réservé pour chaque direction d'une articulation. Comme j'ai déjà présenté dans section du module de la manette, j'utilise le masque des bits pour enregistrer l'état des événements et ce masque est exactement le registre dans l'interface du bras robotique, autrement dit, chaque événement sur le masque est correspondant un mouvement spécifique sur le bras.

L'interface du bras robotique est une classe abstraite et elle ne peut pas être instancié. Les classes dérivées doivent implanter toutes les méthodes héritées. Cette interface ne contient que les fonctions virtuelles. Elles sont utilisées essentiellement dans le cas de l'héritage. Elles n'ont aucune mise en oeuvre donc ont besoin d'être surchargés dans la classe dérivée. Donc, le principe de créer une interface

est de définir seulement ses méthodes virtuelles et de créer les classes dérivées pour chaque matériel spécifique de la classe mère. Les méthodes dans l'interface n'indique pas la manière de faire quelques choses mais indique seulement les méthodes dont nous avons besoin d'implanter pour que la brique du bras robotique peut communiquer avec la brique centrale.

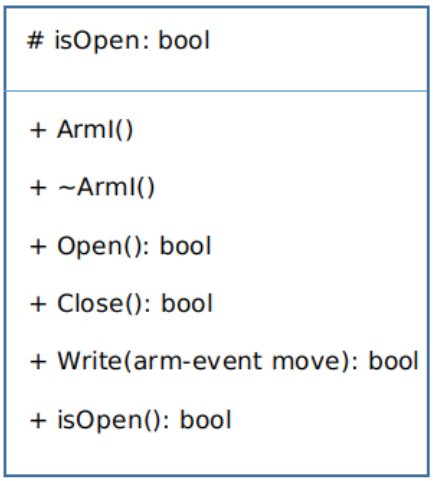

FIGURE 15 – Interface de bras robotique

Les méthodes qui sont utilisées pour le management de l'objet sont ArmI, ArmI, Open, Close, IsOpen et la méthode qui est utilisée pour la transmission des données sont Write. Pour les interfaces, nous avons besoin de réduire le nombre des fonctions possibles afin de simplifier le processus d'appel des interfaces dans le programme principal. Avec les interfaces optimisées, autre développeur peuvent utiliser ces interfaces facilement ou mieux comprendre l'architecture de l'interface pour créer la classe dérivée. La tâche difficile est la mise en œuvre des méthodes dans les classes dérivées par exemple dans la classe du module de bras robotique AL5D, la méthode Open réalise les opérations de l'énumération du bras, d'ouverture la fiche descriptive du bras et d'initialisation la position pour toutes les articulations du bras ou la méthode Write réalise les opérations d'analyse du registre des états des événements, de construction du vecteur des commandes correspondantes et de la commande finale et de la transmission de la commande au bras.

De l'autre côté de la brique central, nous avons besoin d'une interface pour la manette pour que la brique central puisse communiquer avec le module de la manette via quelques fonctions standards. Comme j'ai présenté, l'interface limite les variables entrées et sorties entre 2 parties du programme. Cela permet de gérer plus facilement les données échanges, de simplifier la tâche de maintenance. Cette interface de la manette utilise les événements SDL2 qui est converti à la structure arm-event pour transférer à la partie central.

Au niveau de la méthode, j'ai défini aussi les méthodes qui indiquent quelles sont les actions actives entre le module central et le module de la manette. Donc j'ai défini les méthodes pour manager l'objet comme JoystickI, JoystickI, Open, Close, GetConfig, IsOpen et la méthode pour la transmission des données est Read. Avec cette interface, nous pouvons ajouter les autres modules pour les autres types de manettes (manette bluetooth, manette BLE ...) et ils peuvent bien contrôler le bras via la brique centrale sans aucune influence sur les autres parties s'ils respectent complètement les méthodes au-dessus.

FIGURE 16 – Interface de la manette

3.2.6 Partie centrale

La partie centrale est le lien entre la brique de la manette et du bras robotique. Ce module est le pont de communication entre les 2 briques en contrôlant le flux des données. Grâce aux fonctionnalités comme multithreading, file d'attente, je peux gérer efficacement les données passées. En outre, cette partie aide aussi à convertir au arm_event en mappant les boutons aux mouvements du bras.

Le problème en premier temps quand je réalise la communication entre le bras et la manette est le manquant des événements dans quelques moments. Donc il me faut augmenter la performance du logiciel en appliquant le multithreading. Il va maximiser la performance du système en réalisant multiple tâches simultanément donc il peut garantir le traitement par le filtre sur chaque événement sans en manquer aucun.

Un thread est un flux d'instructions à séquence unique dans un processus. Il est exécuté indépendemment avec les autres thread dans le programme pourtant, ils peuvent partager la mémoire ensemble. Dans ce cas de mon logiciel, j'ai développé 3 threads qui exécute parallèlement :

- Le thread main : il est correspondant à la partie **main** du programme. Il obtient les entrées de l'utilisateur, initialise les parties du programme, démarre les autres threads et attend la fin de la communication.
- Le thread producer : ce thread est responsable d'attacher la manette sur la machine, recevoir les événements à partir le module de la manette, les convertir et les pousse dans une queue.
- Le thread consumer : ce thread permet d'attacher le bras robotique connecté sur la machine, recevoir et analyser l'événement converti dans la queue, pousser les commandes construites dans un vecteur et envoyer ensuite au bras robotique.

La raison que j'ai utilisée le queue ici pour la ressource partagée est à cause de l'ordre de l'ajout et de la consommation des données. L'ordre de transfert des commandes est FIFO (first in first out) donc la queue est convient parfaitement. La transmission des données entre les 2 threads producer et consumer est décrit ci-dessous :

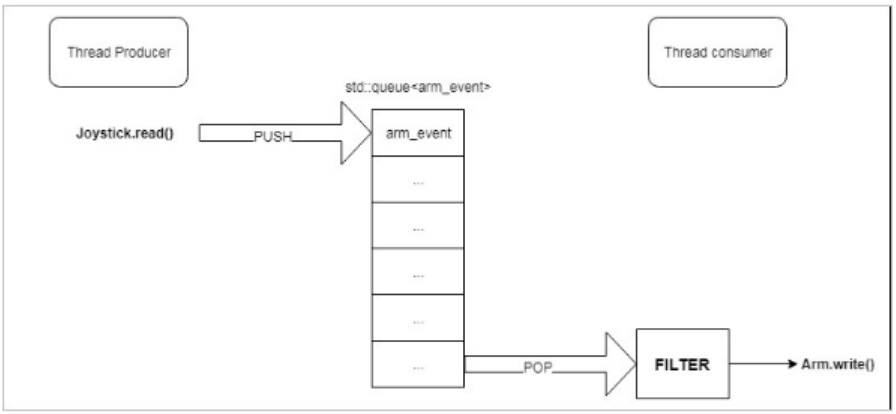

FIGURE 17 – Transmission des données entre les 2 threads

Lorsque nous utilisons le multithreading dans le logiciel et entre les threads, nous déclarons des ressources partagées, il nous faut avoir une méthode pour protéger les ressources. Puisque que si les threads partagent une ressource, les données peuvent être accédées simultanément par les threads pour faire quelques choses. La conséquence est très grave puise qu'il y a la possibilité de fonctionner incorrectement ou avoir les comportements étranges. Donc la protection est très nécessaire.

Une méthode commune utilisée pour la protection de ressource partagée dans multithreading est l'objet **multual exclusion** ou **mutex**. le principe de protection est suivant : dans le cas nous avons un objet mutex, si un thread accède la ressource, la donnée, il va bloquer le mutex. Dans ce moment, si un autre thread tente d'accéder la même ressource, il faut attendre le mutex jusqu'à ce qu'il soit relâché par le premier thread. En conséquence, avec un objet mutex, une ressource peut être accédée par un seul thread dans un moment. Dans mon programme, il y a plusieurs objets mutex qui servent le processus de management des données et le plus important est celui qui protège la queue partagée entre le thread producer et consumer. Le temps du verrouillage de l'objet mutex est assez pour une action push/pop sur la queue. Pour noté, ce temps doit être bien réduit pour ne réaliser que les tâches essentielles sur la ressource. Par exemple, s'il le thread producer est bloqué trop longtemps, la possibilité de rater des événements sur la manette est très élevée. Donc il nous faut avoir des solutions pour réduire le temps du travail sur la ressource partagée. En outre, le mutex protège aussi les variables comme **stopExec__**, **wait__**, **error__** qui sont utilisées afin de vérifier les drapeaux dans le programme correspondant la commande d'arrêt ou l'erreur apparaît avec l'objectif de sortir le programme proprement. De plus, il y a encore un mécanisme qui est utilisée pour notifier un changement à un thread à partir d'un autre thread appelée condition variable. Par exemple, si le thread principal reçoit un signal d'arrêt de l'utilisateur, il doit notifier le signal aux threads producer et consumer pour les arrêter.

Donc avec la partie centrale, je peux bien connecter les 2 modules du joystick et du bras robotique via leurs interfaces. Les données transférées sont gérées de manière prudente pour éviter les comportements étranges et aussi augmenter la performance du programme.

3.2.7 Débogage

En cours de processus du développement de logiciel linux embarqué, je rencontrais pas mal des bogues dans le programme, donc j'ai décidé de faire un système log qui permet d'imprimer (sur le terminal ou dans le fichier log) les informations, les avertissements et les erreurs. Avec le système log, je peux surveiller le processus du programme et connaître la location où une erreur se produit.

Le log est basé sur le printf mais avec les éléments supplémentaires pour décrire mieux le message retourné en ajoutant automatiquement le type du message, le temps où le message est exécuté et la position du message dans le programme (la position exacte dans un fichier et la fonction). Grâce aux macros définis dans le C++ comme **___FUNCTION___**, **___LINE___**, **___FILE___**, je peux récupérer les informations sur ce message. J'ai créé les macros log différents pour les différents types de message comme **LOG_I**, **LOG_D**... Chaque fois j'appelle ces noms, ils sont remplacés par leur contenu déclaré lors de la compilation.

Grâce le système log, je peux réutiliser dans plusieurs projets après pour surveiller les actions du programme. Je peux réduire le temps de tracer les erreurs, réaliser plus efficacement la tâche de Débogage et éviter les erreurs indignes...

```
1 < 15:06:56.0120 >| DEBUG | MoveToInitialPosition[armAL5D.cpp, 159]: Initialization
    cmd --> #0P1450S500
```

Listing 11 – exemple de message log

3.3 Contrôle de bras robotique par le serveur web embarqué

Dans cette méthode, au lieu d'utiliser le joystick pour contrôler le bras, j'ai développé un serveur web embarqué et une interface web pour que l'utilisateur puisse contrôler le bras via le navigateur web. Comme j'ai présenté dans la section avant, grâce à la modularisation du programme dans la première phase, je peux hériter beaucoup le programme avant. C'est-à-dire, je n'ajoute qu'un module du serveur web et un module de la manette virtuelle qui doit respecter les prototypes définies dans la section interface de la manette pour communiquer avec la partie central et je ne dois pas toucher le module du bras robotique. Cela a prouvé le grand avantage de la modularisation dans la programmation en réduisant le temps de développement et d'expansion des nouvelles fonctionnalités ou des soutiens pour les nouveaux appareils.

L'idée de cette méthode est d'utiliser une manette virtuelle sur une application web pour contrôler le bras réel. Donc, j'ai besoin d'un serveur embarqué déployé sur une machine Linux qui permet de récupérer les commandes venues à partir de l'application web, de les analyser et d'envoyer les commandes correspondantes au bras robotique. En outre, à côté du serveur, pour intégrer le serveur dans le programme principal, j'ai dû développer un module de la manette basé sur l'interface définie qui est considérée comme le pont entre la partie centrale et le serveur embarqué.

3.3.1 Module de la manette virtuelle

Donc le contraint que je dois respecter pour ne pas influencer le programme est la formule des fonctions définies pour l'interface du joystick. L'objectif de ce module est le même avec celui avant en fournissant le masque de bit pour l'indication des événements. La différence principale est dans la mission des prototypes **read**, **open** et **close**. Pour les manettes classiques, les APIs open et close servent à attacher ou détacher la manette. Par contre, c'est à cause d'une manette virtuelle, donc il ne faut pas régler les tâches de connecter avec les appareils physiques mais ces APIs servent à construire le serveur embarqué ou arrêter le serveur. Ensuite, pour la API read, l'interface des manettes classiques doit prendre les événements survenus sur la machine hôte comme par exemple les événements reçus par l'interface SDL. En revanche, il n'y a pas la manette connectée avec la machine hôte dans ce cas-là, donc l'événement survenu vient de serveur embarqué. Dès qu'il y a une demande venue sur le serveur,

il définit un drapeau sur 1 pour le signal et met à jour quelques informations. Ci-dessous, vous pouvez voir l'organigramme du module de la manette virtuelle :

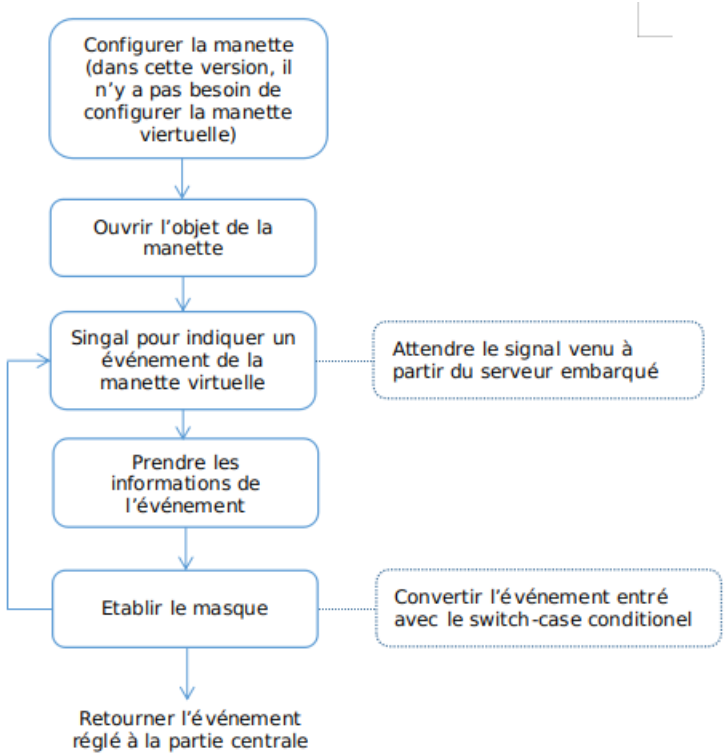

FIGURE 18 – Organigramme du module de manette virtuelle

Donc l'étape suivante est de développer un serveur qui communiquera avec la partie central via cette interface de la manette. C'est la partie principale de cette méthode.

3.3.2 Serveur web embarqué

Pour construire un serveur sur la machine Linux avec le langage de développement C, je peux utiliser la bibliothèque **microhttpd** qui permet d'exécuter un serveur HTTP dans le cadre d'une autre application. En fait, pour construire un serveur, il y a plein d'outils plus performances maintenant. Mais dans le cadre de mon projet, ce serveur n'est pas un serveur HTTP autonome, il fonctionne avec mon application C++, donc cette bibliothèque est la plus appréciée pour utiliser dans ce cas-là.

Nous avons 2 types de serveur : HTTP et HTTPS. En premier temps, je construis un serveur HTTP pour la première version du serveur. Donc HTTP est un protocole sur la couche application. HTTP se trouve au-dessus de la couche TCP dans la pile de protocoles et est utilisé par des applications spécifiques pour se parler. Dans ce cas, les applications sont des navigateurs web et des serveurs web. En fait, il y a 2 méthodes pour la communication avec HTTP qui sont thread par connexion et thread par demande. Avec le thread par demande, pour chaque demande du client (navigateur web), une connexion entre le client et le serveur est établi et sera détruit dès que la réponse de cette demande est reçue par le client. Pourtant, la méthode thread par connexion permet de maintenir la connexion serveur/client pour réaliser plusieurs demandes. Après avoir analysé la caractéristique de mon serveur, puise qu'il peut régler une demande très rapidement (récupérer seulement l'information

des articulations et établir la réponse JSON) et aussi une session de contrôle du bras robotique combine multiples demandes différentes donc j'ai choisi la deuxième méthode. La méthode thread par demande est efficace si le serveur sert multiples clients, mais ce n'est pas l'objectif de mon serveur. Les étapes pour la communication HTTP sont :

- Le navigateur connecte le serveur de nom de domaine (DNS) et récupère l'adresse IP correspondante pour le serveur web.
- Il connecte ensuite le serveur web et envoyer la demande HTTP.
- Le serveur web envoyé une page correspondant avec la demande du client. Si la page n'existe pas, une erreur HTTP 404 est envoyé au client.
- La connexion est maintenue pour la demande suivante.
- La connexion est fermée si le temps de mort est passé.

Dans la réponse du serveur, elle contient les informations du serveur et du paquet des données. Avec la bibliothèque microhttpd, nous avons une API qui permet d'établir un serveur directement avec quelques configurations comme la programmation du serveur multi-thread (ici c'est la méthode thread par connexion), le port du serveur et la fonction qui gère la demande du client. Après avoir défini toutes ces configurations, le serveur peut-être être lancé correctement.

En fait, le problème avec le protocole HTTP est la manque de l'aspect de cyber-sécurité dans la transmission des données via internet. Donc, j'ai ajouté une version du serveur en utilisant le protocole HTTPS. HTTPS est une extension du protocole HTTP pourtant il fonctionne sur la couche TLS (Transport Layer Security) qui est un standard de la technologie de sécurité pour établir une connexion cryptée entre le navigateur web et le serveur web au lieu de la couche application. Grâce aux quelques fonctionnalités supplémentaires, le HTTPS est plus fiable que le HTTP. Premièrement, alors que le protocole HTTP n'exige pas la validation du domaine, le HTTPS exige au moins une validation du domaine en vérifiant le certificat fourni. En fait, un certificat sert à valider l'identité d'un serveur web. C'est-à-dire qu'il garantit la confidentialité des données et rassure les internautes qui se connectent à votre serveur web. Normalement, avec les serveurs déployés sur internet, nous avons besoin d'utiliser le certificat fourni par une Autorité de Certification pour que la communication sécurisée du serveur web est validé. Néanmoins, dans le cadre de mon projet, je n'ai pas besoin de déployer le serveur sur internet et il est utilisé uniquement en interne, c'est pourquoi je crée un certificat auto-signé pour mon serveur. Ce certificat n'a pas la valeur sur internet (lorsque nous accède un site web et le certificat n'est pas validé, un avertissement doit apparaître pour l'annoncer au client) mais me permet de l'utiliser pour tester et vérifier le HTTPS dans mon serveur. Avec n'importe qui a signé, le certificat a toujours quelques informations principales comme l'identité de l'organisation ou de l'individualité, la date d'expiration, l'algorithme de la signature et la clé publique. Deuxièmement, à côté du certificat pour l'authentification du serveur, le HTTPS nous permet aussi communiquer en sécurité en cryptant les données transférées au lieu des textes bruts dans le cas de HTTP. Le processus pour établir une connexion sécurisée avant des échanges des données est ci-dessous :

- Le client initie la poignée de main en envoyant un message *hello* au serveur. Ce message contient quelle version TLS soutenu, la suite de chiffrement (l'algorithme pour créer une clé de session qui est utilisée pour crypter les données entre serveur et client) et une chaîne d'octets aléatoires (*client random*).
- Le serveur réponse le client avec le message qui contient le certificat du serveur, la suite de chiffrement choisie et une chaîne d'octets aléatoires générée par le serveur (*server random*).
- Le client vérifie le certificat du serveur avec l'Autorité de Certification.
- Le client ensuite envoie une chaîne d'octets aléatoires (*premaster secret*) mais elle est déjà cryptée par la clé publique dans le certificat. Cette chaîne peut être comprise par le serveur seulement puise que il n'y a que le serveur qui a la clé privée correspondante.
- Le serveur décrypte le *premaster secret*.
- Grâce aux chaînes d'octets aléatoires dans les étapes précédentes, le client et le serveur peuvent

générer une même clé de session.
— Le client envoie au serveur un message "*finish*" crypté par la clé de session.
— Le serveur envoie au client un message "*finish*" crypté par la clé de session.

Avec ce processus, il n'y a que le client et le serveur qui peut générer la même clé de session en combinant toutes les chaînes d'octets aléatoires par un algorithme de chiffrement définie. Un attaquant ne peut pas générer cette clé de session à cause du manque du *premaster secret*. Pour noter, la clé publique dans le certificat est générée par la clé privée du serveur et un message crypté par la clé publique n'est que décrypté par la clé privée. C'est très difficile ou impossible de trouver une autre clé privée correspondant cette clé publique. Après ce processus, le client et le serveur peuvent se communiquer en utilisant la clé de session pour crypter et décrypter les données transférées (clé symétrique). Pour générer la clé privée et le certificat auto-signé, j'ai utilisé openSSL - un outil robuste pour les protocoles Transport Layer Security (TLS) et Secure Sockets Layer (SSL). Il fournit une boîte complète des outils pour des usages de cryptographie dans des logiciels, des systèmes. Ci-dessous est une ligne de commande pour générer le certificat et la clé privée avec openSSL sous Linux :

```
# openssl req       : primarily creates and processes certificate requests
# -nodes            : a private key is created it will not be encrypted
# rsa:2048          : key size is 2048 bits
# -keyout domain.key: generate private key domaine.key
# -x509             : self signed certificate instead of a certificate request
# -days             : number of days to certify the certificate for
# -out domain.crt   : certificate domain.crt
openssl req -newkey rsa:2048 -nodes -keyout domaine.key -x509 -days 365 -out domain.
    crt
```

Listing 12 – commande pour générer le certificat auto-signé et la clé privée

Comme HTTP, la bibliothèque microhttpd soutient aussi une API qui me permet de construire le serveur HTTPS avec quelques configurations. A côté des configurations communes avec le HTTP, il me faut définir aussi le drapeau pour le protocole SSL, et le pointeur sur le certificat et la clé privée du serveur. Une méthode pour lire les fichiers de système qui contiennent le certificat et la clé privée est nécessaire pour les charger dans les tampons du programme.

Après la configuration du serveur, nous devons comprendre comment le serveur marche. En fait, un serveur web peut fournir quelques points finals (URL - Uniform Resource Locator) où les utilisateurs peuvent accéder pour récupérer, modifier ou interagir un objet sur le serveur. Normalement, ce sont les APIs que les applications web peuvent utiliser pour communiquer avec le serveur. Il y a plein de types différentes de point final comme **POST, PUT, GET...**, pourtant les points finals de mon serveur utilise le type GET. En fait, la demande GET nous permet de récupérer les informations des ressources présentées, autrement dire, avec ce type de demande, l'utilisateur peut récupérer la position actuelle d'une articulation ou toutes les positions du bras dans la réponse retournée de cette demande. Mais pas seulement cela, avec cette demande, mon serveur peut récupérer les conditions sur le point final pour réaliser un contrôle et mettre à jour la valeur de la position des articulations aussi. Donc il me faut définir quelques demandes acceptées par le serveur et la réponse retournée pour chaque demande venue.

Comme j'ai présenté, dans le processus de configurer le serveur, nous devons définir une prototype pour répondre toutes les demandes venues. En outre, cette fonction doit filtrer toutes les demandes acceptées pour mon serveur qui sont les demande de type GET et avoir les conditions suivantes :

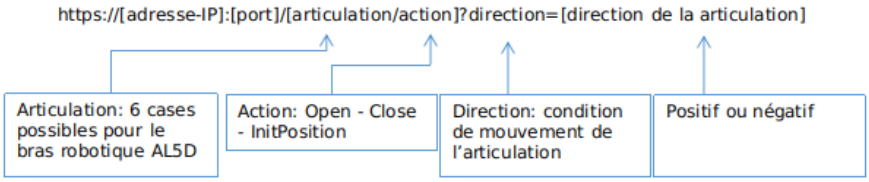

FIGURE 19 – La formule de demande acceptée

l'image ci-dessus décrit la formule URL accepté. Si nous avons besoin de contrôler une articulation sur le bras, nous devons préciser l'articulation que nous allons contrôler et aussi la direction de mouvement. Par contre, si nous avons besoin d'interagir globalement avec le bras, nous pouvons préciser les actions au lieu des articulations comme ouvrir/fermer la communication entre l'utilisateur et le serveur ou initialiser toutes les articulations du bras. Avec les APIs de microhttpd, je peux récupérer les paramètres sur l'URL envoyé au serveur et les vérifier.

Ensuite, il me faut définir la réponse retournée en dépendant les paramètres sur l'URL dans les cases que cette demande est bonne ou mauvaise. Nous pouvons répondre au client une page HTML, une chaîne de caractère... mais normalement, les services web retourne les réponses sous forme JSON - il est un format d'échange de données et est facile pour les humains à lire et à écrire. Avec la réponse JSON, la plupart des langages de programmation peuvent analyser très rapidement et très simplement. Donc c'est très idéal pour moi lors que je développe une application communiquée avec ce serveur pour contrôler le bras, cette application peut comprendre les informations dans la réponse sans difficulté. En conséquence, pour chaque chemin (articulation ou action sur le bras), je dois définir une action correspondant en envoyant les informations au module de la manette virtuelle (comme j'ai présenté, ces informations sont récupérées dans le module de la manette pour établir le masque ensuite) et construisant la réponse JSON. En C++, grâce à la bibliothèque **boost** qui fournit un type de variable **ptree** (Property Tree), la construction d'une réponse JSON devient plus facile. La variable ptree fournit une structure de données qui stocke un arbre de valeurs arbitrairement profondément imbriqué, indexé à chaque niveau par une clé. Chaque nœud de l'arborescence stocke sa propre clé, plus la valeur de cette clé ou une liste ordonnée de ses sous-nœuds et de leurs clés. Nous constatons que l'architecture de la variable ptree est cohérente avec la caractéristique du format JSON et en outre, cette bibliothèque fournie aussi une méthode pour convertir d'une variable ptree à une chaîne de caractères qui se présente sous le format JSON. A côté de la forme de la réponse, au niveau des informations, pour les demandes de contrôle des articulations, la réponse retourne la position actuelle, les positions limites et aussi l'état de cette demande. Pour les demandes d'ouvrir ou fermer la communication, l'état de succès ou d'échec avec la raison sont retournées ou le serveur retourne toutes les positions initiales dans le cas d'initialisation des articulations. Vous pouvez voir les exemplaires des réponses JSON retournées au client après les demandes sur les points finals ci-dessous :

```
{
    "elbow": {
        "positionCurrent": "1750",
        "positionInit": "1800",
        "hightLimit": "2000",
        "lowLimit": "800",
        "status": "OK"
    }
}
```

Listing 13 – La réponse réussie pour la demande de contrôler la coude du bras

```
{
    "elbow": {
        "status": "Failed",
        "error": "value of direction is wrong"
    }
}
```

Listing 14 – La réponse échouée pour la demande de contrôler la coude du bras

Après avoir construit la chaîne de caractères sous forme JSON, le serveur doit créer ensuite une réponse avec la structure *MHD_Response* que le daemon httpd (le serveur HTTP/HTTPS exécute sur la machine Linux) peut comprendre en utilisant cette chaîne de caractères. En outre, le serveur doit ajouter les en-têtes sur cette réponse pour avoir le paquet de données final. Dans le protocole HTTP/HTTPS, les en-têtes de la réponse permettent au client et au serveur de transmettre des informations supplémentaires avec la demande ou la réponse. Puise qu'il n'y a pas de contraintes exactes sur les réponses envoyées donc j'ai ajouté quelques informations communes pour que le client puisse récupérer la réponse comme *Access-Control-Allow-Origin*, *Vary*. Ces informations peuvent être ajoutées dans l'avenir pour les exigences plus précises dans la communication serveur/client via le protocole HTTP, donc je ne me suis pas concentré sur cette partie en ce moment-là.

Lorsque le paquet de données est bien construit, le serveur peut envoyer au client avec le code de réponse HTTP, par exemple : *MHD_HTTP_OK* - code 200 pour indiquer que la réponse est bien envoyée au client d'après la demande.

Maintenant, lorsque nous lançons le serveur, nous pouvons accéder le serveur web via notre navigateur en entrant les URLs définis pour tester le contrôle du bras. L'image ci-dessous présente la réponse retournée après une demande sur le serveur et je constate aussi que le bras robotique peut se déplacer à la demande :

FIGURE 20 – Tester l'api sur le navigateur web

3.3.3 Application web

Avec ce serveur web, je peux contrôler le bras en accéder les points finals du serveur sur le navigateur. Pourtant, ce n'est pas très pratique si je dois rédiger le URL chaque fois pour une commande. C'est pourquoi j'ai développé une application web qui fournit une manette virtuelle. Cette application me permet de communiquer très rapidement avec le serveur en appuyant sur les boutons de la manette virtuelle. Au niveau de l'apparence, le coeur de la manette est hérité par des sources ouvertes et j'ai ajouté les éléments supplémentaires, modifié la forme, la couleur des boutons.

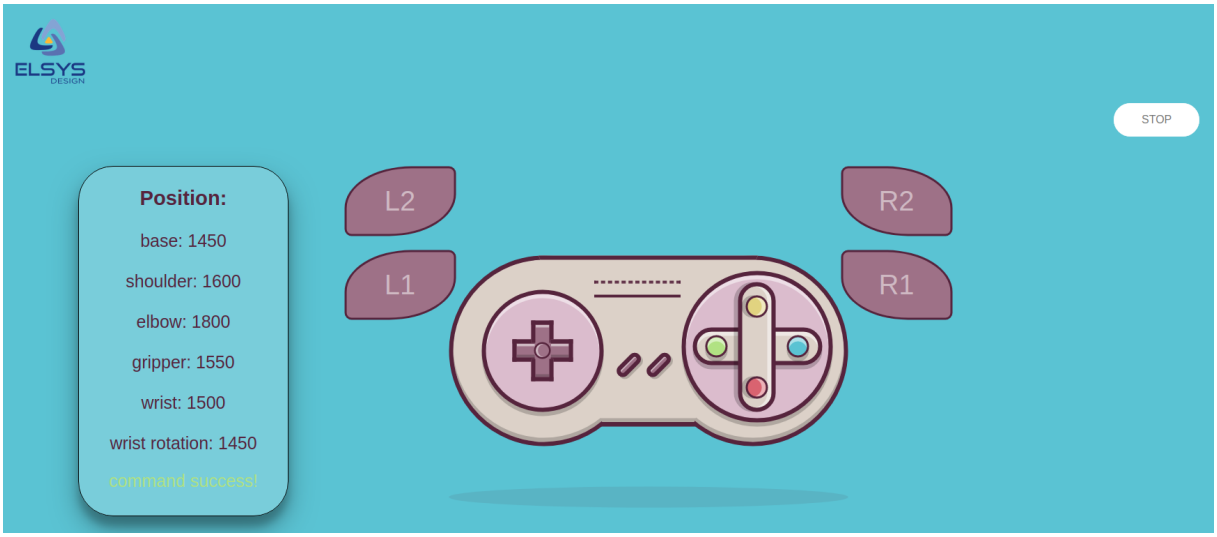

FIGURE 21 – Interface de l'application web

De côté de la manette, j'ai aussi un bouton pour démarrer la session de contrôle du bras ou arrêter la session si elle a bien démarré. A gauche de la manette, il y a un tableau qui indique la position actuelle des articulations et l'état de la réponse (réussi ou échoué). Ces valeurs sont mises à jour en temps réel si nous avons une demande de contrôle au serveur. Pour communiquer entre l'application web et le serveur, je dois capturer les événements des appuies sur les boutons de la manette virtuelle. Pour chaque événement, je peux construire un url correspondant qui doit être envoyé au serveur et attendre la réponse retournée ensuite. A cause de la forme JSON de réponse retournée, je peux déduire facilement son état et les valeurs mises à jour de la position. Une comparaison des informations est

nécessaire pour afficher les bonnes positions du bras sur l'interface web.

En conclusion, avec le serveur web et une application web, nous pouvons bien contrôler le bras robotique via internet. C'est très pratique si cette solution est utilisée pour contrôler un bras industriel à distance à partir d'un centre de contrôle. Une méthode pour enregistrer les commandes sur le cloud est très utile pour visualiser l'histoire des commandes, mais dans ce cadre de mon projet, ce n'est pas l'objectif à faire.

3.4 Déploiement du logiciel sur le Raspberry PI 3

En premier temps, ma mission était de déployer le logiciel sur la carte imxrt1010-EVK, pourtant cette carte ne fonctionne pas avec une OS Linux (quelques cartes imx de NXP peuvent installer un distro de Yocto). Donc, la solution aléatoire est d'utiliser la carte Raspberry PI. Malgré la distribution Debian - une distribution pré-construite pour le Raspberry PI, j'ai décidé d'installer un distro créé par le Yocto Project au lieu pour m'entraîner la compétence sur cet outil. Donc la raison pour laquelle nous utilisons le Yocto est la possibilité de construire et personnaliser une image Linux. Grâce à cela, nous pouvons créer des systèmes Linux personnalisés qui peuvent fonctionner sur les cartes très limites au niveau de la taille de la mémoire ou sur les produits embarqués quelle que soit l'architecture matérielle.

Bitbake est le coeur de *Yocto Project* pour construire une image. C'est un moteur d'exécution de tâches génériques qui permet de récupérer et interpréter les métadonnées, décider quelles tâches sont obligées d'exécuter et les exécuter afin de générer une image Linux ou un logiciel. Donc notre objectif est de fournir des métadonnées pour notre distribution à côté des métadonnées par défaut. Les métadonnées principales que le Yocto a définies sont la recette, la couche, la classe et les fichiers de configuration. Avant de commencer à définir les configurations de notre distro, il nous faut construire l'environnement et de télécharger le poky (la distribution de référence du Yocto Project qui contient les choses comme le système de construction OpenEmbedded et metadonnées par défaut pour construire une propre distribution). Ensuite, nous devons étudier chaque type de métadonnées pour les définir. Premièrement, les recettes qui sont désignées par l'extension de fichier .bb, sont les fichiers de métadonnées les plus élémentaires. Elles fournissent à Bitbake des instructions comme :

- Description des informations du paquet (exemple un logiciel, un driver).
- La version de la recette.
- Les dépendances existantes (les bibliothèques dépendantes utilisées dans le logiciel).
- La location de code source (la location locale ou sur internet pour que le bitbake puisse télécharger).
- La manière pour compiler le code source.
- La location sur la machine cible pour installer le paquet compilé.

Dans ce cadre de mon projet, j'ai défini une recette pour mon logiciel embarqué que j'ai présenté dans les sections au-dessus. Cette recette décrit des informations du logiciel et aussi les bibliothèques que j'ai utilisées. De plus, j'ai configuré les éléments pour que le bitbake puisse télécharger mon code source sur le github, vérifier son intégrité et le sauvegarder dans une location définie. Normalement, nous pouvons définir les instructions pour que le bitbake puisse compiler le code source, pourtant, j'ai déjà eu le CMakeList dans mon code source donc je dois déclarer seulement le cmake pour cette étape. Pour noter, heureusement, le yocto projet a nous fournit déjà quelques recettes principales comme recipes-graphics, recipes-bsp... qui permettent de fournir les drivers pour le Raspberry PI.

Deuxièmement, nous avons les fichiers de configuration qui sont désignés par l'extension de fichier .conf à définir. Ces fichiers appartiennent à plusieurs zones dans le projet afin de définir les configurations de la machine, les options de configuration de la distribution, du compilateur, des utilisateurs et aussi

les configurations communes. J'ai dû définir 2 fichiers de configuration principaux, un fichier pour configurer ma distribution (*local.conf*) et un fichier pour configurer les couches (*bblayers.conf*) utilisée pour cette distribution (présenté plus tard la définition des couches). Le premier fichier me permet de choisir la machine, les répertoires pour installer toutes les sources en amont (upstream source), fichiers à état partagé (ces fichiers permettent d'accélérer les construites en fonction de la sortie précédemment construite), la sortie de la construite et les paramètres pour ajouter ou supprimer les fonctionnalités sur l'image de distro. Le deuxième fichier me permet d'activer les couches pendant le processus de construction.

Troisièmement, les fichiers de classes qui sont désignés par l'extension de fichier .bbclass contient toutes les informations partagées entre les recettes. Ces fichiers réalisent quelques tâches de base telles que la récupération, le déballage, la compilation (exécute n'importe quel Makefile présent), l'installation et l'empaquetage. Ces fichiers sont bien créés par le Yocto Project. Pour répondre à mes besoins, je n'ai pas de créer autres fichiers de classe.

Comme j'ai présenté ci-dessous, j'ai parlé des couches. Donc, elles sont les répertoires qu'elles contiennent les métadonnées associées. C'est-à-dire, les couches contiennent toutes les recettes qui ont le même usage, les fichiers de classe. Le modèle des couches facilitent la collaboration, le partage, la personnalisation et la réutilisation dans l'environnement de développement Yocto Project en séparant logiquement les informations dans le projet. Par exemple, nous pouvons utiliser une couche qui contient toutes les configurations pour un matériel particulier. Lorsque nous travaillons sur un matériel similaire, cette couche peut être partagée ou être modifiée un peu au lieu de développer depuis le début. Avant d'implanter la recette de mon logiciel, j'ai créé une couche séparée pour mon projet et mettre la recette dedans.

Après toutes les configurations, je peux bien utiliser le bitbake pour générer l'image de la distribution pour mon Raspberry PI. Le temps de la première construction est très long mais pour les prochaines fois, il diminue clairement grâce la fonctionnalité des fichiers à état partagé. Ensuite, pour compiler le code source, je peux faire bitbake sur la recette que j'ai développée. La sortie est un paquet de logiciel **.ipk** qui peut être décompressé sur la machine cible avec la gestionnaire de paquets **OPKG** pour récupérer le fichier exécutable. Le résultat est que le logiciel est bien cross compilé après le bitbake et fonctionner sur la machine cible qui est installée avec une propre distribution. Lorsque je travaille avec le Yocto Project, je peux générer eSDK qui fournit une chaîne d'outils de développement croisé et des bibliothèques adaptées au contenu d'une image spécifique. Ces outils sont les solutions pour accélérer le processus de développement avec le Yocto Project. La pierre angulaire du SDK extensible est un outil de ligne de commande appelé **devtool**. Il me permet d'ajouter, modifier et mettre à niveau les recettes ou les couches par les lignes de commande. En conséquence, il peut simplifier beaucoup la tâche de créer une nouvelle recette ou fournir quelques options pour modifier la recette existante... En outre, Devtool soutien les commandes afin de compiler, tester les changements sur la machine hôte et déployer une application plus facilement sur la machine cible ou bien intégrer l'application dans une image.

Nous allons regarder un peu la conception du système de construction utilisée par le Yocto Project. Le diagramme suivant représente le flux de travail de haut niveau d'une construction. Il décrit les blocs logiques fondamentaux d'entrée, de sortie, de processus et de métadonnées qui composent le flux de travail.

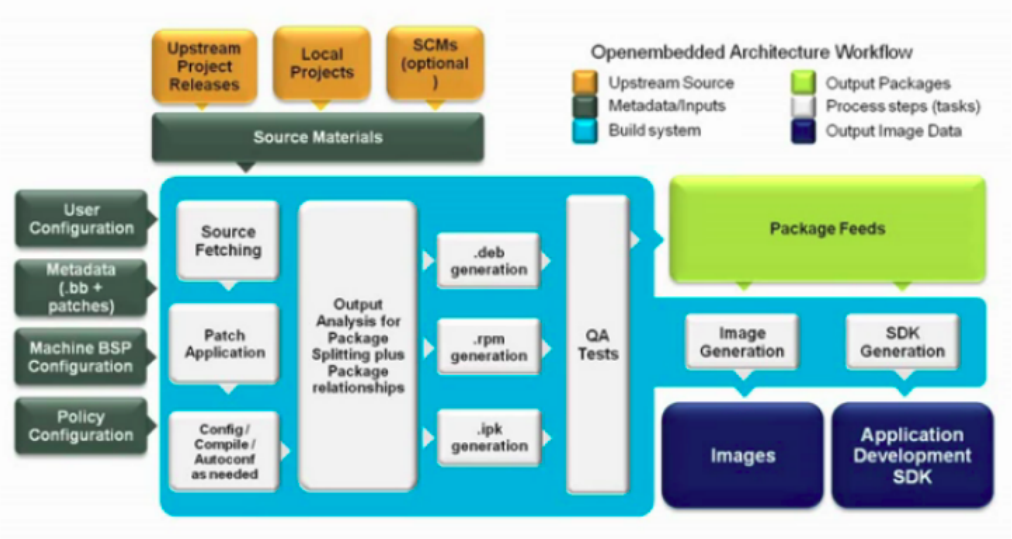

FIGURE 22 – Conception du système de construction OpenEmbedded

Donc au niveau de l'entrée, premièrement, nous avons besoin des sources locales ou des projets en amont qui permet au système d'utiliser pour générer l'application après. En outre, nous devons définir la configuration de l'utilisateur (ce sont les configurations dans les fichiers *local.conf* et *bblayers.conf* pour définir les propriétés de la construction), les métadonnées (les recettes), la couche BSP (contient les configurations spécifiques à la machine) et la couche de la distribution (la couche fournit les configurations de politique pour une image ou un SDK). Ensuite, le système de construction va récupérer, rapiécer et compiler le code source pour une machine cible en utilisant les métadonnées et les configurations. Après l'étape de compilation, le système de construction analyse les résultats et divise la sortie aux paquets différents (*.deb*, *rpm*, *ipk*). Les paquets généré sont passés les testes QA pour vérifier ses stabilités et placés dans la zone *Package Feed*. Cette zone est une étape intermédiaire dans le processus de construction. Le système OpenEmbedded va récupérer les paquets ici pour générer l'image de la distribution ou le SDK que j'ai présenté au-dessus. Nous pouvons aussi récupérer les paquets des applications dans le répertoire *build* tel que le paquet .ipk de mon logiciel que j'ai utilisé pour générer et lancer le fichier exécutable sur la machine cible.

En conclusion, le comportement du système de construction dépend principalement les configurations et les métadonnées que nous mettons à l'entrée et nous pouvons récupérer l'image, le SDK ou le paquet des applications à la sortie du système. J'ai pu bien installer ma propre image sur le Raspberry PI et déployer le logiciel sur cette image. Ce logiciel fonctionne proprement qui nous permet de contrôler le bras robotique par la manette ou par l'application web.

3.5 Logiciel du contrôle de bras robotique sur la carte MIMXRT1010-EVK

Dans cette méthode, j'ai développé une application temps réel sur la carte MIMXRT101-EVK en utilisant l'API FreeRTOS. Cette application nous permet de contrôler le bras suivant un scénario que nous pouvons choisir par la manière d'appui sur les boutons. L'idée en premier temps est de contrôler le bras robotique par une manette PS3 connectée avec la carte via le protocole USB. En fait, la manette PS3 est un appareil dans la classe USB HID (human interface device). Néanmoins, lorsque j'ai développé une application pour la connexion USB HID entre la manette et la carte en utilisant la pile USB Host (la carte imx joue le rôle hôte dans ma communication), après une demande de lecture

de la carte, le bit du registre qui permet de signaler des données entrées, n'active jamais. Par contre, si j'ai envoyé les données à la manette, ce bit est activé correctement, donc je pense que le problème est que la carte ne peut pas recevoir les données de cette manette pour les mettre dans le tampon. A cause à manque du matériel de débogage, je ne peux pas aller plus loin et surmonter ce problème.

Donc, pour contrôler le bras par cette carte, j'ai utilisé le protocole UART pour la communication. Nous pouvons connecter les 2 appareils par le USB, pourtant, la classe USB utilisée sur le bras robotique est une classe spécifique de l'entreprise FTDI et nous n'avons aucune information pour le développer ce driver (il n'y a que le driver de cette classe USB sur les machine OS Windows, Linux et Mac). La connexion UART est incluse sur 3 fils (tx, rx, gnd). Dans l'application, après avoir configuré les pins pour utiliser le protocole UART, il me faut déclarer les paramètres communs pour ce protocole comme le débit en bauds, la parité, le bit d'arrêt, l'horloge source, l'adresse UART définie par NXP et aussi le tampon pour contenir les données. Ensuite, je dois implanter un mécanisme pour envoyer les données au bras robotique dans le RTOS. Dans le driver UART de NXP, il nous fournit une méthode qui permet d'envoyer les données par la méthode interruption (send non-blocking). Cette méthode permet de retourner directement mais pas besoin d'attendre les données bien écrites sur le registre de l'émetteur. Par contre, dans le driver, lorsque les données sont bien écrites sur le registre TX, une fonction callback sera retournée avec l'état de cette transmission pour annoncer le résultat de la transmission. Grâce cela, je peux réaliser un mécanisme comme ci-dessous pour le processus d'envoyer les données au bras :

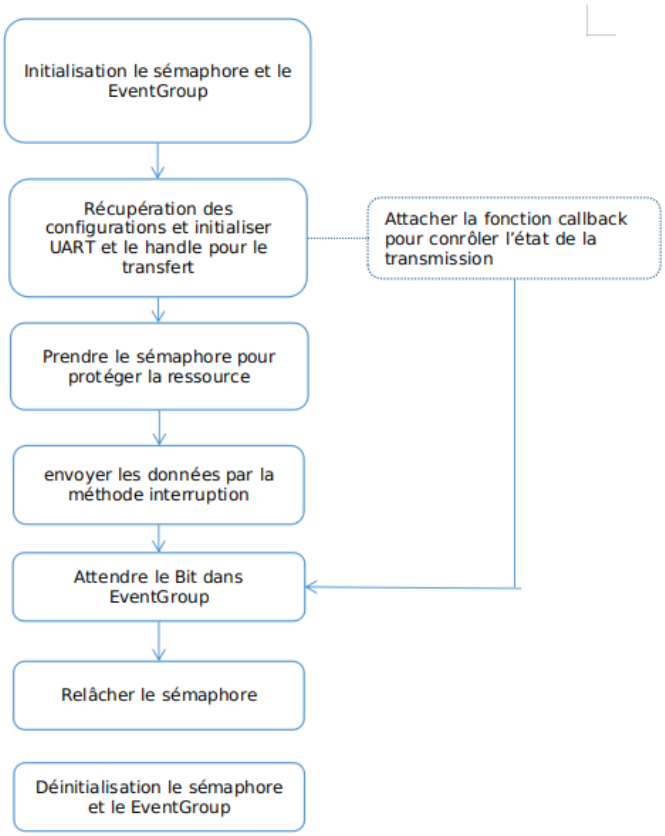

FIGURE 23 – Le mécanisme d'envoyer les données via UART dans le RTOS

Sur ce diagramme, nous pouvons regarder le handle pour le transfert. Handle est une référence abstraite qui permet de contenir les informations concernant un objet tel que le tampon, la taille du tampon,

le pointeur sur la fonction callback... Le handle est fournit pour que le driver puisse cacher toutes les choses compliquées dans la couche plus base par exemple les tâches de lecture/écriture des données à ou du registre, mettre l'état de la transmission... et ma mission n'est que de définir ou récupérer quelques paramètres dans le handle. Dans le processus de la transfert des données, j'ai utilisé le mutex pour protéger les tampons de transfert. Le mutex est une méthode très commune pour l'exclusion mutuelle dans multiplethread ou multitâche. Lors que la tâche d'envoyer les données est en cours, aucune autre tâche peut accéder ces ressources pour lire ou écrire. Grâce à cela, nous pouvons éviter les problèmes de la manque des données ou aussi des comportements étranges à cause des données incorrectes. Le mutex doit être relâché immédiatement dès que la transfert finit. En outre, j'ai utilisé aussi le EventGroup de FreeRTOS pour la synchronisation entre la tâche principale et la fonction callback. Le mécanisme du EventGroup est qu'une tâche va attendre les bits sur une chaîne des bits de déclarer avant par l'API de FreeRTOS alors qu'une autre tâche ou une interruption ait la mission d'activer le bit sur la chaîne. Chaque bit dans la chaîne se présente un événement différent correspondant l'état de la transmission. La tâche principale est bloquée en attendant un événement (par exemple : événement réussi, échoué) sur cette chaîne pendant un intervalle de temps définie. Si les données sont bien écrites sur le registre TX, la fonction callback est appelée automatiquement par le driver et récupère un état de transmission. Cette fonction ensuite peut activer un bit dans la chaîne afin d'annoncer l'événement à la tâche principale. En conséquence, la tâche principale peut lire la bonne valeur sur la chaîne et retourner le résultat correspondant (transmission réussie ou échouée). C'est un type de synchronisation dans le RTOS en gardant une tâche bloquée jusqu'à une directive donnée par une interruption/tâche.

Au-dessus, j'ai bien implanté un mécanisme pour envoyer les données par le protocole UART sur le RTOS. Maintenant, nous considérons l'architecture de l'application de contrôle. D'abord, je dois configurer la carte, tous les pins, l'horloge et aussi l'interruption du bouton. Ensuite, c'est la partie principale de l'application. Cette application constitue 2 tâches différentes, une tâche pour envoyer les données au bras (*task_uart*) et une tâche pour traiter l'appuie sur le bouton de la carte (*task_bouton*). La mission de la deuxième tâche est de compter le nombre de fois que nous appuyons sur le bouton pendant 2 secondes pour choisir le scénario des mouvements du bras. Pour chaque mode de mouvement, cette tâche va registrer un ensemble de commandes de ce mode dans un tampon. Par contre, la première tâche nous permet de lire les commandes dans le tampon et les passer à tour de rôle dans le processus d'écriture sur le bras robotique. Pourtant, nous devons avoir une façon pour que la première tâche puisse connaître quand elle doit envoyer les commandes au bras. Grâce à la méthode EventGroup, nous pouvons synchroniser ces 2 tâches. Chaque bit dans la chaîne de EventGroup présente un mode de mouvement. Donc après avoir analysé le nombre de fois que nous appuyons sur le bouton, la deuxième tâche active le bit correspondant sur la chaîne. En ce moment-là, la tâche (*task_uart*) attend ces bits et est débloquée si elle reçoit le bon bit pour commencer à envoyer l'ensemble des commandes au bras.

En outre, les 2 tâches travaillent sur le même tampon qui contient l'ensemble des données (une tâche fait la lecture et une tâche fait l'écriture sur ce tampon). Donc je dois protéger cette ressource commune afin d'éviter les erreurs inattendues. Le mutex est utilisé pour protéger la ressource. Le mécanisme du mutex a présenté au-dessus dans le processus d'envoyer les données au bras.

Avec cette solution, nous pouvons définir n'importe quel scénario pour le bras robotique. Par exemple, un bras autonome peut travailler avec un scénario défini avant pour un objectif défini dans l'industrie. En outre, nous pouvons changer le mode de l'opération du bras robotique pour les usages différents.

Conclusion

Après le stage de 6 mois, j'ai atteint globalement les objectifs proposés au début, pourtant, à cause des difficultés dans la période de virus corona, le sujet de stage est un peu changé et affecté.

Les tâches principales réalisées sont le développement d'un logiciel sous Linux qui permet de contrôler le bras robotique AL5D par les manettes différentes ou par une application web, le développement d'un logiciel RTOS installé sur la carte MIMXRT1010-EVK pour contrôler le bras suivant les scénarios différents et aussi génération la documentation de code source. Le logiciel Linux peut fonctionner proprement en testant les manettes PS3, Nintendo Switch et aussi la manette virtuelle. En outre, le logiciel RTOS n'est pas jamais cassé pendant le temps de l'utilisation. Le document généré par le Doxygen est clair et me permet de comprendre le code source facilement. Grâce à ce stage, j'ai acquis beaucoup d'expérience sur les techniques de programmation dans le C et C++, la manière de la modularisation, le processus d'intégration, la construction d'une distribution et aussi la compilation croisée avec le Yocto Project. Ce stage m'a permis aussi de comprendre les notions dans l'aspect de cyber-sécurité, de travailler avec le OpenSSL. Ce stage m'a permis de préparer les connaissances, les compétences indispensables pour mes projets professionnels dans le logiciel embarqué dans l'avenir.

Par contre, il y a aussi les choses à améliorer à la fin de mon stage. A cause de manque des matériaux donc il y a quelques petites tâches que je ne peux pas compléter comme construction de la distribution avec le Yocto sur une carte dans la série IMXRT et NXP, la connexion USB entre la carte MIMXRT1010-EVK avec la manette PS3. Dans la tendance des applications IoT, mon logiciel peut devenir plus intéressant si le processus de contrôle du bras peut être supervisé par le Cloud. Finalement, une concentration sur la cyber-sécurité est une grande valeur ajoutée pour mon logiciel.

Annexe

Système d'exploitation RTOS

Nous savons que le système d'exploitation est principalement destiné aux ordinateurs. Cependant, en réalité, il existe de nombreux appareils électroniques qui exécutent une forme raccourcie de système d'exploitation à l'intérieur. En outre, il existe également de nombreux types de systèmes d'exploitation conçus pour les microcontrôleurs. Parmi eux, certains appartiennent à la famille des systèmes d'exploitation en temps réel (RTOS), le terme temps réel indique ici que le temps de réponse du système est très rapide.

Système d'exploitation en temps réel conçu pour des tâches spéciales. Les applications doivent être exécutées en temps réel, les erreurs doivent être isolées et traitées rapidement. Tout retard ou erreur imprévue peut provoquer une panne du système. Par conséquent, RTOS est utilisé dans les applications qui nécessitent un temps de réponse rapide et précis. Les microcontrôleurs ont des ressources très limitées. Par conséquent, ce système d'exploitation se concentre uniquement sur quelques fonctionnalités. Ils essaient de maximiser le nombre de threads, de planificateur et de tâches sur un petit système. En règle générale, RTOS est un segment ou une partie d'un programme dans lequel il traite la contribution des tâches, la planification, la priorité des tâches et la capture des messages envoyés à partir des tâches.

Le noyau du système d'exploitation demande le CPU pour exécuter le traitement des tâches sur des intervalles de temps. Il vérifie également la priorité des tâches, trie les messages de la tâche et réalise la planification. Dans un système exécutant RTOS, il existe également des ressources partagées ainsi que des portions allouées à chaque tâche. La fonctionnalité de base d'un RTOS :

- Planificateur (Scheduler).
- Services en temps réel (Realtime Services).
- Synchronisation et messagerie (Synchronization and Messaging).

FreeRTOS est une couche RTOS conçue pour être suffisamment petite pour fonctionner sur un microcontrôleur.

CMakeList

```
cmake_minimum_required( VERSION 3.1 )
set (CMAKE_BUILD_TYPE Debug)
project (C++_arm_PS3joystick)
set (CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall")  #-Werror -D_REENTRANT")
set(CMAKE_MODULE_PATH ${CMAKE_CURRENT_SOURCE_DIR}/CMake)
find_package(Libevdev REQUIRED)
find_package(Libudev REQUIRED)
find_package(SDL2 REQUIRED)
#
# Declare the directories of header file
#
include_directories(${CMAKE_CURRENT_SOURCE_DIR}/includes
                    ${EVDEV_INCLUDE_DIRS}
                    ${UDEV_INCLUDE_DIRS}
                    ${SDL2_INCLUDE_DIRS}
                    /usr/lib/x86_64-linux-gnu/
                    )
#
# Add source file
```

```
#
set(SOURCES
    src/joystickPS3.cpp
    src/main.cpp
    src/udevhandler.cpp
    src/armAL5D.cpp
    src/transmitionqueue.cpp
    src/log.cpp
    src/servercontroller.cpp
    src/resthttp.cpp
    src/httpserver.cpp
    )

# option(DEBUG "use debug file" OFF)
if(CMAKE_BUILD_TYPE MATCHES Debug)
  add_definitions(-DDEBUG)
endif()

option(RPI "Compile for RPI" OFF)
if(RPI)
  add_definitions(-DRPI)
endif()

#
# Define the relation between file execute and source file
#
add_executable(armDev ${SOURCES})

set_property(TARGET armDev PROPERTY CXX_STANDARD 11)
message(STATUS "EVDEV_LEDFLAGS = ${EVDEV_LDFLAGS}")

#
# Use the library
#
string(STRIP ${SDL2_LIBRARIES} SDL2_LIBRARIES)
target_link_libraries(armDev
                      ${EVDEV_LIBRARIES} ${EVDEV_LDFLAGS}
                      ${UDEV_LIBRARIES} ${UDEV_LDFLAGS}
                      ${SDL2_LIBRARIES} ${SDL2_LDFLAGS}
                      microhttpd
                      pthread
                      )
# Install program into usr/local/bin by make install
install(PROGRAMS ${CMAKE_CURRENT_BINARY_DIR}/armDev
    DESTINATION bin
  RENAME ${CMAKE_PROJECT_NAME}-armDev)
```

Listing 15 – CMakeList

Références

[1] R. HEATON, *How does HTTPS actually work ?*, `https://robertheaton.com/2014/03/27/how-does-https-actually-work/`, 2014.

[2] D. Both, *Explore how the /dev directory gives you direct access to your devices in Linux*, `https://opensource.com/article/16/11/managing-devices-linux`, 2018.

[3] S. Rifenbark, *Yocto Project overview and concepts manual*, `https://www.yoctoproject.org/docs/2.5/overview-manual/overview-manual.html`, 2018.

[4] bootlin, *Linux Documentation*, `https://elixir.bootlin.com/linux/v4.13/source/Documentation`, 2020.

[5] Cmake, *CMake Documentation*, `https://cmake.org/cmake/help/v3.5/index.html`, 2020.

[6] GTK-Doc, *libudev reference manual*, `http://presbrey.scripts.mit.edu/doc/libudev/ch01.html`, 2020.

[7] xarch Linu, *Gamepad working in linux*, `http://www.lynxmotion.com/c-130-al5d.aspx`, 2020.

[8] Lynxmotion, *About the Robot Arm AL5D*, `http://www.lynxmotion.com/c-130-al5d.aspx`, 2020.

[9] S. Wiki, *Joystick Support*, `https://wiki.libsdl.org/CategoryJoystick`, 2020.

Table des figures