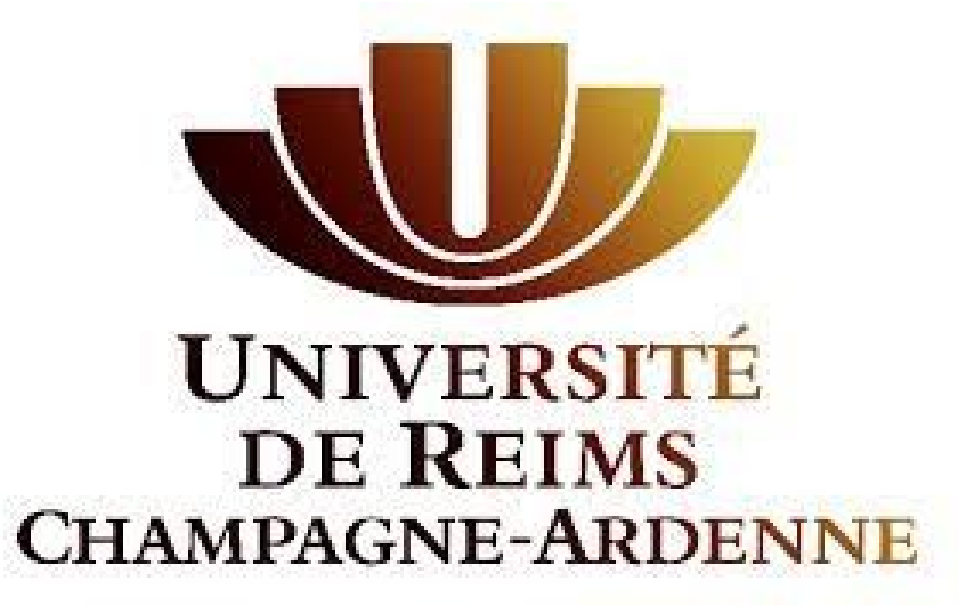
RAPPORT DE STAGE

Lab-I*



VERPOORTE William 22/05/24
PPRO0403
Groupe S4F3

Sommaire

I/ Présentation de l’entreprise

I.A/ Le Lab-I*

Le Lab-I* est un laboratoire de recherche en informatique lié à l’UFR des Sciences Exactes et Naturelles de Reims, créé par Hacène FOUCHAL en janvier 2024.

Le laboratoire est composé de nombreux chercheurs et enseignants dont ceux qui se sont occupés de nous : Hacène FOUCHAL, Emilien BOURDY-LIEBART et Geoffrey WILHELM.

I.B/ Objectif du Lab-I*

L’un projet actuel du Lab-I* sur lequel nous avons travaillé, InDiD, se focalise sur le développement de véhicules intelligents et connectés.

I.C/ Contexte

Un véhicule connecté est un véhicule qui communique avec de nombreux autres véhicules ainsi qu'avec toutes les infrastructures liées au trafic. Ces échanges de données et d'informations permettent d'améliorer la sécurité et la gestion du trafic sur la portion de route où se trouve le véhicule en question.

Pour atteindre les objectifs visés par les véhicules connectés, nous utilisons le projet geonet, un code développé en C++ permettant de gérer toutes les communications entre différents véhicules. Il gère notamment l'envoi et la réception de paquets CAM ainsi que d'autres paquets essentiels à la communication des véhicules. Une autre particularité du geonet est qu’il est développé avec le support framework Qt ce qui sera utile pour la suite.

I.D/ Mes missions

Mes missions dans ce projet étaient de permettre le portage du code de geonet sur un MK5 fabriqué par Cohda Wireless, un système embarqué qui se branche directement à bord du véhicule pour récupérer tous types de données utiles, qui a le gros avantage d'être doté d'une antenne WiFi 802.11p. Nous avons choisi le MK5 pour sa renommée dans le wifi véhiculaire. Un autre avantage pour nous est que nous avons accès au SDK du MK5 pour effectuer tous nos tests et nous avons aussi eu la chance d'en récupérer 16 du projet Scoop@f.

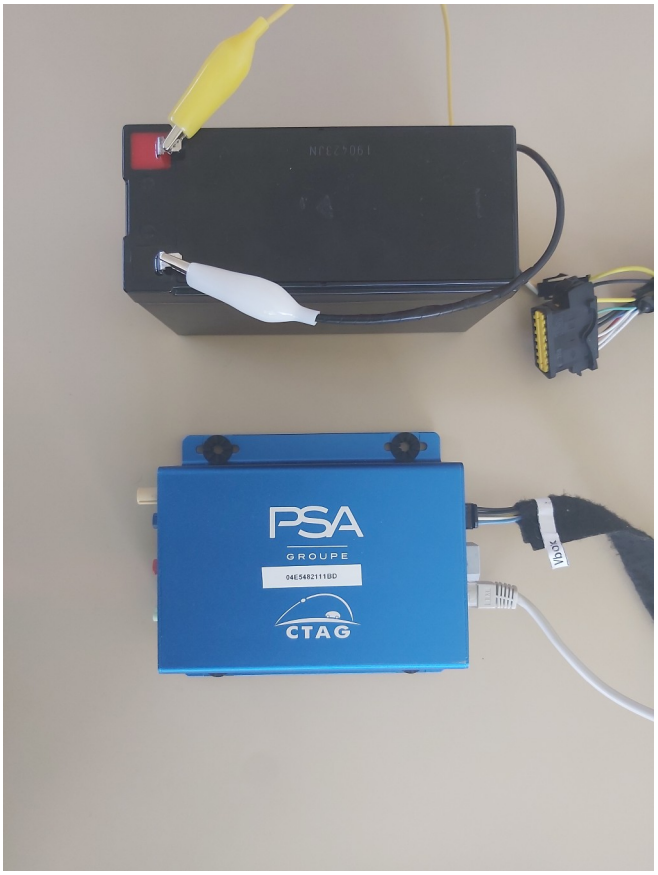


Figure 1: MK5 branché à une batterie

Pour parvenir à remplir ces missions, je dois donc commencer par permettre l'utilisation de la dernière version de Qt pour l'architecture du MK5, puis me procurer toutes les dépendances nécessaires au bon fonctionnement de geonet.

II/ Déroulement du stage

II.A/ Configuration de l'environnement de travail

Pour effectuer le portage, je dois utiliser une machine virtuelle (VM) contenant un SDK fourni par Cohda Wireless, l'entreprise qui produit les MK5.

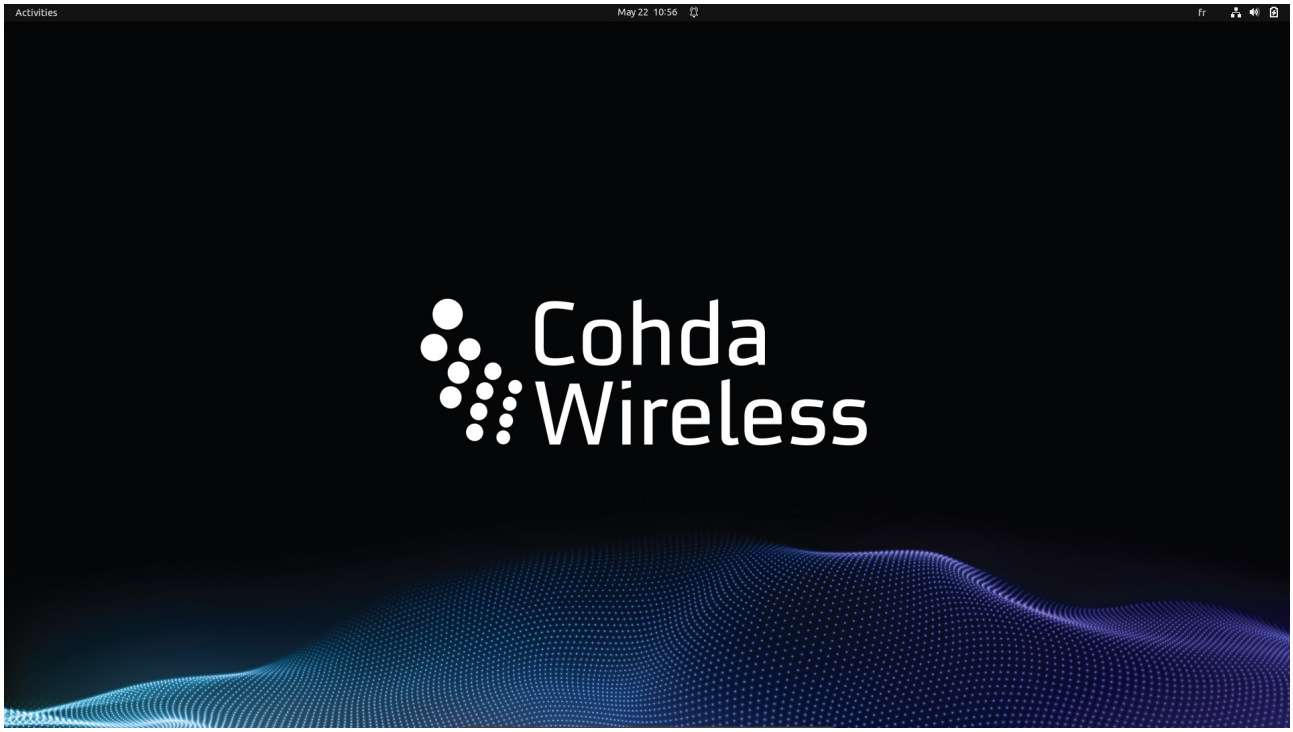


Figure 2: Machine Virtuel avec le SDK fourni par Cohda Wireless

Cette VM contient le sysroot du MK5, donc normamlement toutes les bibliothèques, binaires et fichiers d'en-tête correspondant à l'architecture du MK5.

J'ai donc commencé par configurer la VM selon la documentation fournie par Cohda. La VM est sous Ubuntu 18.04. Ensuite, j'ai configuré et compilé le sysroot en question afin d'obtenir tout les fichiers utiles.

II.B/ Compilation de Qt pour ARM32

La première grande étape de mon travail est donc d'effectuer la cross-compilation de Qt 6.7.0 pour ARM32. Qt est nécessaire pour configurer la stack de geonet, qui fonctionne avec un fichier .pro, le type de fichier pris en charge par Qt pour configurer un projet.

B.1) Problématique

La principale difficulté pour effectuer cette compilation est qu'il y très peu de documentation spécifique à laquelle se référer. Toutes les informations sur la cross-compilation de Qt que l'on peut trouver concernent soit Android, soit Raspberry Pi. Elles sont cependant toujours utiles pour avoir une idée de la marche à suivre.

J'avais aussi à ma disposition la documentation d'un stagiaire ayant tenté de réaliser la même tâche que moi par le passé.

Une autre grande difficulté à laquelle j'ai été confronté après un certain temps est la version très ancienne de la VM, un Ubuntu 18.04, ayant au moins cinq ans de retard sur les mises à jour actuelles. Étant donné que nous souhaitons compiler une version très récente de Qt, de nombreux problèmes de dépendances ont fini par apparaître, me bloquant dans mon travail après plusieurs jours.

B.2) Résolution

Pour essayer de continuer à avancer, j'ai donc décidé de mettre à jour la version d'Ubuntu vers la 22.04. Cela a, comme je l'espérais, résolu de nombreux problèmes de versions de bibliothèques, ainsi que quelques problèmes plus complexes liés à des fichiers générés par le code, qui ont fini par disparaître également.

Après avoir compris le fonctionnement correct de la cross-compilation de Qt utilisant CMake, j'ai pu résoudre des erreurs plus simples de dépendances manquantes. J'ai également créé un répertoire plus propre que l'ancien et assez conséquent servant de sysroot, un répertoire contenant toutes les bibliothèques, tous les en-têtes et tous les fichiers binaires nécessaires au fonctionnement d'une architecture cible en cross-compilation.

On commence par récupérer les sources de Qt à partir du dépôt git (voir V.A.1).

En utilisant CMake, il nous faut avant tout créer un fichier « toolchain.cmake » (voir V.A.2), un fichier permettant d'ajouter des options à la configuration du CMakeList, notamment les options de compilation qui précisent les bons compilateurs et les options de liaison de certaines bibliothèques nécessaires au fonctionnement de Qt.

Dans ce fichier toolchain, nous précisons donc le sysroot, le prefix, qui est l'emplacement où sera installé Qt après la compilation, puis tous les flags qui indiquent où chercher les fichiers d'en-tête et les liaisons de bibliothèques spécifiques.

Une fois les fichiers nécessaires créés et configurés, il ne reste plus qu'à lancer les commandes CMake correctes. Si Qt est installé pour la première fois, il faut donc, comme dit précédemment, l'installer une première fois pour l'hôte (voir V.A.3).

Puis une fois une version de Qt installé pour l'hôte, nous pouvons configurer et installer Qt pour la cible (voir V.A.4).

J'utilise l'option -j4 après ninja pour allouer 4 cœurs à la compilation, sinon il faut attendre plus de 2 heures.

La cross-compilation, après quelques semaines de recherches, a enfin pu s'achever après l'installation de Qt par les sources une première fois pour l'hôte, puis une nouvelle fois pour la cible. À noter que la compilation de Qt prend énormément de temps en raison du grand nombre de fichiers présents dans les sources, environ 10 000, pour un temps de compilation d'environ une heure.

Une fois la cross-compilation terminée, nous avons donc à notre disposition un fichier binaire « qmake6 » permettant de configurer un projet avec les paramètres du sysroot. Avec tous les changements nécessaires dans un Makefile, nous obtenons en sortie de compilation un exécutable pour l'architecture ARM32, correspondant au MK5.

II.C/ Cross-compilation des dépendances

Après l'installation de Qt, il nous faut désormais préparer la compilation de geonet, et pour ce faire, nous avons besoin de plusieurs bibliothèques et/ou logiciels.

C.1) Libpcap

La cross-compilation de libpcap ressemble beaucoup à celle de Qt, nous utilisons également cmake et ninja.

Comme toujours avec la cross-compilation, il faut se procurer les sources de la bibliothèque :

git clone https://github.com/the-tcpdump-group/libpcap

cd libpcap

Nous avons à nouveau besoin de créer un toolchain.cmake (voir V.B.1) qui diffère un peu de celui de Qt dans lequel nous supprimons simplement tout les flags liés à Qt.

Il ne reste plus qu'à lancer la configuration avec cmake (voir V.B.2).

Ainsi nous avons installé libpcap pour ARM32, qui fonctionnera donc pour le MK5

C.2) Cryptopp

Pour Cryptopp, je me suis simplifié la tâche en utilisant un site qui répertorie de nombreux packages pré-compilés pour ARM32 ou arm64, ArchLinuxArm.

Voici le lien du package que j'ai téléchargé : https://archlinuxarm.org/packages/armv7h/crypto++

À noter que pour la version actuelle du geonet, la version 8.7.0 de Cryptopp est indispensable, les versions les plus récentes étant instables.

Après l'avoir téléchargé et extrait, je copie les différents répertoires souvent présents dans /usr du package dans mon sysroot afin de regrouper toutes les dépendances.

La cross-compilation du package est aussi possible mais prend plus de temps. Les sources se compilent souvent avec autoconf. Une documentation l'utilisant pour la cross-compilation est disponible dans la partie Xsd-cxx et Bluez.

C.3) Gpsd

Pour Gpsd, j'ai procédé de la même manière que pour Cryptopp. J'ai récupéré le package au lien suivant : https://archlinuxarm.org/packages/armv7h/gpsd.

Ensuite, j'ai copié tous les répertoires utiles dans le sysroot.

C.4) Xsd-cxx (avec xerces-c)

Pour cette dépendance, nous avons également besoin de Xerces-C. Nous commençons donc par récupérer Xerces-C, dont le package est disponible à cette adresse : https://archlinuxarm.org/packages/armv7h/xerces-c.

Cependant, plus tard lors de la compilation de la stack, j'ai rencontré quelques problèmes, donc je l'ai compilé à partir des sources cette fois ci en utilisant autoconf.

On aussi doit définir des variables pour être sûr que les bons compilateurs, linkers, etc., soient utilisés.

À noter que la commande peut parfois changer étant donné que chaque "./configure" possède des options différentes qui change selon les projets.

Le "configure" créé un Makefile qu'il faut parfois revoir pour bien mettre certains chemins ou compilateurs. Il ne reste plus qu'à compiler le code (voir V.C.1).

Ensuite pour xsd-cxx, la version à l'adresse suivante : https://archlinuxarm.org/packages/armv7h/xsd m'a suffit.

C.5) Bluez

Bluez est un peu différent des autres car il s'agit d'un logiciel et non d'une bibliothèque. Cependant, le processus de compilation reste le même que pour les autres. Ici, on utilise à nouveau autoconf pour effectuer la cross-compilation.

On commence par récupérer le code source (voir V.D.1) sur le GitHub de Bluez .

Comme vu avec Xerces-C, nous devons définir des variables d'environnement (voir V.D.2) pour être sûr que les bons fichiers binaires et sysroot soient pris en compte.

Enfin il ne reste plus qu'à lancer ./configure (voir V.D.3)avec les bonnes options.

On définit l'hôte sur notre cible, qui sera construit par l'option –build. Ici, le compilateur de la VM est utilisé. On précise un préfixe pour définir où l'installer, puis on ajoute l'option –with-sysroot pour que le compilateur cherche directement les dépendances dans les fichiers du sysroot.

Ensuite, on exécute la commande make pour compiler le code source. Ici, j'ai dû modifier deux lignes du Makefile pour que la compilation réussisse (voir V.D.4).

À la fin de la compilation, on exécute : make install et Bluez sera installé dans le répertoire défini par l'option –prefix, dans la bonne architecture.

II.D/ Compilation de la stack pour le MK5

Une fois toutes les dépendances installées, il ne reste plus qu'à compiler la stack pour obtenir une version de geonet pouvant fonctionner sur le MK5. Cependant, il manque encore deux bibliothèques dont nous avons besoin pour que tout fonctionne.

D.1) Compilation libDATEX

La première bibliothèque est libDATEX, que nous retrouvons dans un répertoire de la stack. Celle-ci, tout comme la deuxième bibliothèque, fonctionne avec un fichier .pro, donc nous aurons besoin d'utiliser Qt6.

Pour démarrer la compilation, il faut donc configurer le code source à l'aide de la commande qmake6 que nous trouvons dans le répertoire "bin" de l'emplacement où Qt a été installé, en suivant le plan que nous avons établi plus tôt dans le rapport. Voici les commandes à suivre :

```
/home/duser/arm-linux-gnueabihf/qt6-arm/bin/qmake6 DATEX.pro
```

La partie "/home/duser/arm-linux-gnueabihf/qt6-arm" peut être remplacée par le préfixe précisé lors de la configuration de Qt. Le fichier DATEX.pro porte un autre nom que j'ai simplifié pour le rapport.

Une fois la commande effectuée, un Makefile sera généré, avec plusieurs erreurs qu'il faudra corriger, notamment les compilateurs qui ne sont pas les bons dès le début. Nous devons remplacer les gcc ou g++ par arm-linux-gnueabihf-gcc ou -g++.

De même, les chemins pour accéder au binaire qmake6 sont souvent incorrects, il faut les remplacer par le chemin réel menant au binaire qmake6 de l'installation pour ARM32.

Enfin, nous pouvons lancer la compilation avec make, mais il risque d'y avoir des erreurs liées à des en-têtes manquants. Si c'est le cas, nous devons les trouver dans le système ou les installer s'ils ne sont pas là, puis les inclure dans la variable INCPATH du Makefile avec -I/chemin/vers/les/entêtes.

Il ne reste plus qu'à exécuter make install après la compilation et déplacer les bibliothèques dans notre sysroot et dans une répertoire que la sortie de la commande qmake6 pour le projet principal nous donneras.

D.2) Compilation libASN

Pour la compilation de la deuxième bibliothèque, libASN, le déroulement est identique à celui de la compilation de libDATEX. Nous configurons avec la commande qmake6 :

```
/home/duser/arm-linux-gnueabihf/qt6-arm/bin/qmake6 ASN.pro
```

Le Makefile généré contient les mêmes erreurs à corriger que celui de libDATEX, donc il faudra vérifier les quelques lignes précisées précédemment. Ensuite, nous lançons la commande make et encore une fois ajoutons les en-têtes dans INCPATH si nécessaire.

À la fin de la compilation, nous exécutons make install et déplaçons les fichiers générés si nécessaire.

D.3) Compilation de la stack

Après l'installation des deux dernières bibliothèques, nous devons maintenant compiler le geonet, qui est bien plus conséquent que les bibliothèques précédentes.

Le fonctionnement est encore une fois le même. Nous utilisons qmake6 pour configurer le projet avec la commande suivante :

```
/home/duser/arm-linux-gnueabihf/qt6-arm/bin/qmake6 ITS.pro
```

Le Makefile fourni contient encore les mêmes petites erreurs, mais il peut aussi contenir des chemins vers des en-têtes ou des bibliothèques incorrects qui ressemblent à ceci :

/home/duser/arm-linux-gnueabihf/home/duser/arm-linux-gnueabihf/fichier…

Il faudra donc remplacer ces mauvais chemins par les bons.

Après la commande qmake6, il se peut que des avertissements à propos de répertoires inexistants soient présents. Si c'est le cas, il vaut mieux créer les répertoires à l'emplacement indiqué qui contient libDATEX ou libASN. Sinon, de très nombreuses erreurs pénibles à corriger concernant des fichiers d'en-têtes manquants seront présentes tout au long de la compilation.

Pour la compilation, j'ai encore une fois décidé d'utiliser la commande make -j4. Cela permettra de gagner du temps vu le volume du projet, qui même avec l'option -j4 prend un certain temps à se compiler complètement. Il est possible de rencontrer des erreurs d'en-têtes manquants ou de bibliothèques non trouvées. Dans ce cas, il faudra soit inclure les en-têtes comme fait précédemment, soit préciser dans les flags de LIBS les chemins vers les bibliothèques concernées.

Je suis aussi tombé sur des erreurs internes au code, comme des fonctions indéfinies contenues dans string.h malgré sa version correcte, que j'ai donc dû légèrement modifier le geonet pour remplacer ces fonctions par d'autres quasiment identiques, par exemple, j'ai remplacé, strncpy par strlcpy.

Si tout se passe bien, nous arrivons donc à la fin de la compilation, avec un exécutable geonet, qui en effectuant la commande file geonet, nous le format attendu ARM32..

II.E/ Test sur le MK5

Maintenant que nous avons réussi à obtenir notre exécutable compatible avec le MK5, nous allons devoir le tester directement dessus pour vérifier si tout fonctionne comme prévu.

Pour cela, nous devons nous connecter en SSH au MK5. Pour commencer, sur la VM, nous devons modifier les paramètres réseau pour activer la première interface réseau en NAT et la deuxième en connexion par pont. Une fois cela fait, nous devons brancher le MK5 en Ethernet à notre ordinateur. Si tout est bien configuré, nous pourrons nous connecter en exécutant les commandes suivantes dans un terminal :

ssh root@192.168.0.189 et entrer le mot de passe : root.

À noter que l'adresse d'un MK5 change sur le dernier octet en fonction du dernier nombre de l'adresse MAC écrit dessus.

Maintenant que nous sommes connectés au MK5, nous pouvons transférer notre exécutable dans le MK5 en utilisant la commande suivante :

scp geonet root@192.168.0.189:~/ et entrer le mot de passe root.

E.1) Problématique

Malheureusement, l'exécutable ne fonctionne pas directement, car il a besoin de dépendances que le MK5 ne contient pas, telles que les bibliothèques de Qt6 ou d'autres dépendances comme brotli, une bibliothèque de compression utilisée par Qt..

Nous pouvons utiliser la commande ldd sur le programme geonet pour obtenir une liste des dépendances ainsi que le chemin où chacune est attendue. Ensuite, il suffit de transférer les bibliothèques concernées en utilisant la commande scp.

Cependant, le plus gros problème se pose avec les dépendances manquantes, notamment libc. Cette bibliothèque est utilisée par d'autres bibliothèques que nous avons importées dans le MK5. Malheureusement, la version de libc sur le MK5 est vieille d'au moins 5 ans, ce qui est beaucoup plus ancien que notre version de Qt. Le problème est que libc est une bibliothèque de base de tout système, elle est ancrée tellement profondément que si nous transférons la version que nous avons sur la VM, le système d'exploitation du MK5 se casse.

Il nous faut donc trouver un moyen de contourner ce problème pour réussir à exécuter notre geonet.

E.2) Potentielles résolutions

Une des premières solutions que nous avons essayées a été de transférer le libc de la VM sur un autre chemin que celui de base sur le MK5. Ensuite, nous avons défini la

variable d’environnement LD_LIBRARY_PATH sur le chemin où se trouve cette version de libc, avant d'essayer d’exécuter le geonet. Cependant, d’autres fichiers liés à libc qui n’étaient pas à la bonne version ont provoqué des erreurs. Même après le transfert de ces fichiers, nous avons rencontré une erreur de segmentation à l’exécution, qui se produisait avant même que le fichier ne tente de s’exécuter. Nous avons donc dû abandonner cette idée.

Une autre solution qui paraissait prometteuse était de compiler le geonet avec toutes les dépendances affichées par le ldd précédent, de manière statique. Ainsi, l’exécutable aurait pu contenir directement les bibliothèques nécessaires à la bonne version, et ne provoquerait pas d'erreur à l’exécution. Cependant, même après avoir récupéré toutes les librairies nécessaires sous forme statique avec le fichier .a, nous avons rencontré plusieurs problèmes. Tout d'abord, à la fin de la compilation du geonet, une très longue liste d’erreurs assez compliquées à corriger s’est affichée. De plus, libc ne peut pas être utilisée de manière statique, car elle est la bibliothèque qui permet la liaison des librairies statiques à l’exécutable. Par conséquent, nous aurions toujours rencontré le problème d'un libc de mauvaise version.

Il nous reste deux solutions qui n’ont pas encore été entièrement testées :

La première consiste à utiliser QtCreator pour déployer le projet sur le MK5 et voir comment il réagit. Cependant, pour une raison inconnue, QtCreator ne fonctionne pas sur la VM. Pour le remplacer, nous allons essayer de procéder avec CQtDeployer, qui doit être d'abord compilé à partir des sources pour du ARM32, ce qui est en cours de traitement.

L'autre solution serait de mettre à jour le MK5 en ayant accès au dépôt de packages. Cependant, Cohda Wireless facture l’accès à ces mises à jour, ce qui implique une somme plutôt conséquente, sans avoir de réel certitude que la bibliothèque libc y soit présente à la bonne version. Cette solution est donc envisagée comme un dernier recours.

III/ Conclusion

III.A/ Travail effectué

Nous rappelons que la mission d’origine était d’effectuer le portage du geonet sur le MK5, en passant par plusieurs étapes, à savoir :

- La cross-compilation de Qt 6.7.0
- La cross-compilation des différentes dépendances du projet
- La cross-compilation de la stack du geonet
- Les tests sur le MK5 et la vérification du bon fonctionnement

Au vu de l'avancement de mon travail et de la liste des tâches à effectuer, je peux affirmer que l'objectif de ce stage a été quasiment atteint, malgré certaines difficultés rencontrées, notamment la compilation de Qt à partir des sources et la nécessité de développer de nouvelles compétences.

III.B/ Apport de cette expérience

Cette expérience s'est révélée être d'une utilité conséquente, notamment en ce qui concerne le développement de nouvelles compétences liées à la compilation et l'amélioration de mes connaissances sur les systèmes UNIX.

Ce stage m'a permis d'entraîner mes capacités d'apprentissage sur des sujets nouveaux, mais également d'améliorer ma compétence en travail d'équipe. En effet, même si je ne travaillais pas sur les mêmes sujets que mes collègues, j'ai pu leur apporter mon aide dans certains cas, et eux-mêmes m'ont grandement aidé à résoudre certaines difficultés.

De plus, ce stage m'a offert un aperçu des méthodes de travail en laboratoire et dans un environnement plus professionnel que celui de l'université. Cela a été une occasion précieuse d'acquérir une expérience pratique et de mieux comprendre les exigences du monde professionnel.

III.C/ Apport futur

Cette expérience ne prend pas fin pour moi immédiatement, car j'ai demandé à prolonger ma participation pour un mois supplémentaire. Cela me permettra de continuer à avancer autant que possible dans le but de finalement réussir à exécuter le geonet sur un MK5.

Je saisis cette opportunité de prolongation pour accompagner certains de mes collègues à Valenciennes pour l'événement final InDiD, qui se déroulera à l'Université Polytechnique de Valenciennes. Nous aurons l'occasion de présenter notre travail à d'autres chercheurs et personnes intéressées lors de cet événement.

IV/ Remerciements

Je tiens à exprimer mes sincères remerciements à Hacène FOUCHAL pour m'avoir offert cette opportunité de stage enrichissante. Je suis reconnaissant également à Emilien BOURDY LIEBART pour avoir supervisé mon travail avec diligence tout au long de cette période. Leur soutien et leurs conseils ont été précieux pour mon développement professionnel et personnel.

V/ Annexe

V.A/ Commande Qt

A.1) Clone à partir du dépôt git

```
git clone https://code.qt.io/qt/qt5.git qt6
cd qt6
git checkout 6.7.0  # Remplacez par le bon tag ou branche pour Qt 6.7.0, si disponible
git submodule update --init --recursive
#ou pour moins de modules
./init-repository -module-subset=essential,qtbase,qtconnectivity,qtpositioning,qtwebengine,qtwebsockets,qthttpserver,qtlocation
```

A.2) Contenu du fichier toolchain.cmake

```
include_guard(GLOBAL)

set(CMAKE_SYSTEM_NAME Linux)
set(CMAKE_SYSTEM_PROCESSOR arm)

set(TARGET_SYSROOT /home/duser/arm-linux-gnueabihf)
set(CMAKE_SYSROOT ${TARGET_SYSROOT})
set(CMAKE_INSTALL_PREFIX ${TARGET_SYSROOT}/qt6-arm)

set(CMAKE_C_COMPILER /usr/bin/arm-linux-gnueabihf-gcc-11)
set(CMAKE_CXX_COMPILER /usr/bin/arm-linux-gnueabihf-g++-11)

set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -I${TARGET_SYSROOT}/include")
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -I${TARGET_SYSROOT}/include")

set(QT_COMPILER_FLAGS "-march=armv7-a")
set(QT_COMPILER_FLAGS_RELEASE "-O2 -pipe")
set(QT_LINKER_FLAGS "${QT_LINKER_FLAGS} -lGL")  # Ajout des options de liaison pour OpenGL

set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_PACKAGE ONLY)
set(CMAKE_INSTALL_RPATH_USE_LINK_PATH TRUE)
set(CMAKE_BUILD_RPATH ${TARGET_SYSROOT})
```

A.3) Configuration pour l'hôte

```
cmake -B build -H../qt6/ 
  -GNinja \
  -DCMAKE_BUILD_TYPE=Release \
  -DQT_BUILD_EXAMPLES=OFF \
  -DQT_BUILD_TESTS=OFF \
  -DCMAKE_INSTALL_PREFIX=/opt/qt6 \
  -DCMAKE_CXX_COMPILER=/usr/bin/x86_64-linux-gnu-g++-11 \
  -DCMAKE_C_COMPILER=/usr/bin/x86_64-linux-gnu-gcc-11 \
  -DCMAKE_ASM_COMPILER=/usr/bin/x86_64-linux-gnu-gcc-11 \
  -DCMAKE_MAKE_PROGRAM=/usr/bin/ninja \
  -DQT_QMAKE_TARGET_MKSPEC=linux-g++-64

cd build && ninja -j4
sudo ninja install
```

A.4) Configuration pour la cible

```
cmake -B build -H../qt6/ \
   -GNinja \
   -DCMAKE_TOOLCHAIN_FILE=toolchain.cmake \
   -DCMAKE_BUILD_TYPE=Release \
   -DQT_HOST_PATH=/opt/qt6 \
   -DCMAKE_MAKE_PROGRAM=/usr/bin/ninja \
   -DQT_BUILD_EXAMPLES=OFF \
   -DQT_BUILD_TESTS=OFF 

cd build && ninja -j4
sudo ninja install
```

V.B/ Commande Libpcap

B.1) Contenu du toolchain.cmake

```
include_guard(GLOBAL)

set(CMAKE_SYSTEM_NAME Linux)
set(CMAKE_SYSTEM_PROCESSOR arm)

set(TARGET_SYSROOT /home/duser/arm-linux-gnueabihf)
set(CMAKE_SYSROOT ${TARGET_SYSROOT})
set(CMAKE_INSTALL_PREFIX /home/duser/libpcap-arm)

set(CMAKE_C_COMPILER /usr/bin/arm-linux-gnueabihf-gcc-11)
set(CMAKE_CXX_COMPILER /usr/bin/arm-linux-gnueabihf-g++-11)

set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -I${TARGET_SYSROOT}/include")
```

```
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -I${TARGET_SYSROOT}/include")

set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_PACKAGE ONLY)
set(CMAKE_INSTALL_RPATH_USE_LINK_PATH TRUE)
set(CMAKE_BUILD_RPATH ${TARGET_SYSROOT})
```

B.2) Configuration de Libpcap

```
cmake -B build -H../libpcap-1.10.4/ \
   -GNinja \
   -DCMAKE_TOOLCHAIN_FILE=toolchain.cmake \
   -DCMAKE_BUILD_TYPE=Release \
   -DCMAKE_MAKE_PROGRAM=/usr/bin/ninja

cd build && ninja -j4
sudo ninja install
```

V.C/ Commande Xsd-cxx

git clone https://github.com/apache/xerces-c

cd xerces-c

autoconf // ça va créer l'exécutable ./configure

puis

```
export TOOL_PREFIX=/usr/bin/arm-linux-gnueabihf
export CXX=$TOOL_PREFIX-g++
export AR=$TOOL_PREFIX-ar
export RANLIB=$TOOL_PREFIX-ranlib
export CC=$TOOL_PREFIX-gcc
export LD=$TOOL_PREFIX-ld
export CCFLAGS="-march=armv7-a -mfpu=vfp"
export TOOL_HOME=/home/duser/arm-linux-gnueabihf/
export ARM_TARGET_LIB=${TOOL_HOME}
```

On définit toutes ces variables pour être sûr que les bons compilateurs, linkers, etc., soient utilisés.

Et enfin :

```
./configure   --host=arm-linux-gnueabihf   --build=x86_64-linux-gnu   --prefix=/home/duser/xerces  --with-sysroot=/home/duser/arm-linux-gnueabihf
(toujours le même sysroot)
```

make -j4

make install.

V.D/ Commande Bluez

D.1) Clone des sources

```
git clone https://github.com/bluez/bluez
cd bluez
```

D.2) Configuration de l'environnement

```
export TOOL_PREFIX=/usr/bin/arm-linux-gnueabihf
export CXX=$TOOL_PREFIX-g++
export AR=$TOOL_PREFIX-ar
export RANLIB=$TOOL_PREFIX-ranlib
export CC=$TOOL_PREFIX-gcc
export LD=$TOOL_PREFIX-ld
export CCFLAGS="-march=armv7-a -mfpu=vfp"
export TOOL_HOME=/home/duser/arm-linux-gnueabihf/
export ARM_TARGET_LIB=${TOOL_HOME}
```

D.3) Configuration de la compilation

```
./configure           --host=arm-linux-gnueabihf           --build=x86_64-linux-gnu
--prefix=/home/duser/bluez --with-sysroot=/home/duser/arm-linux-gnueabihf (toujours
le même sysroot)
```

D.4) Modification apportée au MakeFile et Compilation

```
CFLAGS += -g -O2 -I/home/duser/arm-linux-gnueabihf/lib/glib-2.0/include
LDFLAGS  +=  -L/home/duser/arm-linux-gnueabihf/lib  -Wl,-rpath,/home/duser/arm-linux-
gnueabihf/lib -Wl,-rpath-link,/home/duser/arm-linux-gnueabihf/lib

make -j4
make install
```

Je vous remercie de votre lecture et de votre attention.

VERPOORTE William