RAPPORT DE PROJET DE SESSION PRÉSENTÉ À

PROFESSEUR MILOUD BAGAA

DU COURS

SYSTÈMES D'EXPLOITATION EMBARQUÉS (**GEI1089**)

TRIMESTRE (**A25**)

PAR

HATEM CHAABANE

AZIZ AMIRI

BAC EN GÉNIE INFORMATIQUE

Construction d'un système Linux embarqué complet

Pour Raspberry Pi 5

Département de génie électrique et génie informatique

18 DECEMBRE 2025

Table des matière

1. RÉSUMÉ

Ce projet porte sur la conception, la construction et le déploiement d'un système d'exploitation Linux embarqué "sur mesure" pour le Raspberry Pi 5. Contrairement à l'utilisation d'une distribution généraliste précompilée (telle que Raspberry Pi OS), ce projet exige la maîtrise complète de la chaîne de production logicielle. Nous avons utilisé l'outil **Buildroot** pour générer une image système minimaliste et optimisée.

Le système final intègre un ensemble complexe de fonctionnalités : un point d'accès WiFi (AP) autonome, un serveur DHCP/DNS, un broker MQTT pour la communication IoT, et une interface graphique utilisateur (GUI) développée en PyQt5 pour le contrôle d'un robot "Tank". Une attention particulière a été portée à l'automatisation du démarrage via des scripts d'initialisation et à la personnalisation du noyau pour inclure un Splash Screen. Ce rapport détaille les choix techniques, l'architecture du système de fichiers (Overlay), ainsi que la méthodologie rigoureuse de débogage qui a permis de surmonter les défis liés aux drivers propriétaires Broadcom du Raspberry Pi 5.

---

2. INTRODUCTION ET OBJECTIFS

Dans le contexte des systèmes embarqués modernes, la capacité à produire un OS léger, sécurisé et dédié à une tâche unique est une compétence critique. Le Raspberry Pi 5, avec son architecture ARM Cortex-A76, offre une puissance de calcul significative mais nécessite une configuration logicielle précise pour exploiter ses périphériques.

Les objectifs principaux de ce mandat sont :

1. **Construction de l'image système :** Générer, à partir des sources, le bootloader, le noyau Linux et le système de fichiers racine (Rootfs).
2. **Autonomie Réseau :** Le système doit transformer le Raspberry Pi en routeur WiFi (Access Point) sans dépendre d'une infrastructure externe.
3. **Support Applicatif :** Assurer l'exécution automatique d'une application Python/Qt5 (TankApp) dès le démarrage.
4. **Support Communication :** Intégrer et configurer le protocole MQTT pour le pilotage temps réel du robot.
5. **Expérience Utilisateur :** Masquer les logs de démarrage (boot console) derrière un écran de chargement personnalisé (Splash Screen).

Le défi majeur réside dans l'intégration cohérente de ces services tout en maintenant un système léger, contrairement aux solutions basées sur Yocto ou Debian qui peuvent s'avérer plus lourdes pour ce type d'application spécifique.

---

3. JUSTIFICATION DU CHOIX TECHNOLOGIQUE : BUILDROOT VS YOCTO

Pour la réalisation de ce système, nous avions le choix entre deux standards de l'industrie : Le Projet Yocto et Buildroot. Notre choix s'est porté sur **Buildroot** pour les raisons suivantes :

1. **Adéquation avec le projet :** Buildroot excelle dans la création de systèmes simples et monolithiques pour une architecture matérielle unique. Notre objectif est de produire une seule image pour le Raspberry Pi 5. Yocto, avec son système de "Layers" et de recettes complexes, est conçu pour maintenir une famille de distributions Linux sur plusieurs années et sur des matériels hétérogènes. Pour un projet de session ciblé, la complexité de Yocto (Bitbake) aurait alourdi le processus sans apporter de gain technique immédiat.
2. **Facilité de configuration :** Buildroot utilise `kconfig` (via `make menuconfig`), une interface familière identique à celle du noyau Linux. Cela permet de sélectionner les paquets, le noyau et le bootloader dans une interface unifiée.
3. **Support matériel natif :** Buildroot fournit une configuration par défaut `raspberrypi5_defconfig` très robuste qui gère déjà l'essentiel du BSP (Board Support Package), incluant les firmwares GPU et les configurations de boot `config.txt`.
4. **Rapidité de compilation :** Étant plus simple, Buildroot génère l'image finale plus rapidement que Yocto lors des cycles de développement itératifs, ce qui est un atout majeur en phase de débogage.

Bien que Yocto soit supérieur pour la gestion des mises à jour à long terme (OTA) et la modularité binaire, ces fonctionnalités n'étaient pas requises ici.

4. ARCHITECTURE DU SYSTÈME ET CONFIGURATION

La base du système a été générée via la commande `make raspberrypi5_defconfig`. Cette commande pré-configure l'architecture cible pour le processeur (Cortex-A76, 64-bit ARMv8).

4.1. Configuration de la cible et du Noyau

Dans le fichier `.config`, les options suivantes ont été validées automatiquement ou ajustées :

- **Architecture :** `BR2_aarch64=y` et `BR2_cortex_a76=y`.
- **Noyau Linux :** Utilisation d'une version "Custom Tarball" du dépôt officiel Raspberry Pi, assurant la compatibilité avec les périphériques récents du Pi 5.
- **Bootloader :** Intégration des fichiers de firmware propriétaires (`start4.elf`, `fixup4.dat`) via `BR2_PACKAGE_RPI_FIRMWARE`.

4.2. Analyse des paquets et dépendances

Via `make menuconfig`, nous avons sélectionné les composants logiciels nécessaires. Voici une analyse détaillée de ces choix :

- **Gestion du WiFi (Point Critique) :**
  - `BR2_PACKAGE_BRCMFMAC_SDIO_FIRMWARE_RPI_WIFI=y` : C'est le paquet le plus critique. Sans lui, le driver noyau `brcmfmac` ne trouve pas le fichier binaire `.bin` pour initialiser la puce WiFi.
  - `BR2_PACKAGE_IW` et `BR2_PACKAGE_KMOD` : Outils pour manipuler les interfaces sans-fil et charger les modules noyau manuellement si nécessaire.
  - `BR2_PACKAGE_HOSTAPD` : Le démon gérant le mode Point d'Accès.
- **Services Réseau :**
  - `BR2_PACKAGE_DNSMASQ` : Choisi pour sa légèreté par rapport à ISC-DHCP-SERVER, il gère à la fois le DHCP et le DNS.
  - `BR2_PACKAGE_DROPBEAR` : Serveur SSH léger pour la maintenance à distance.
  - `BR2_PACKAGE_MOSQUITTO` : Broker MQTT essentiel pour la communication avec le contrôleur du robot.
- **Interface Graphique et Python :**
  - `BR2_PACKAGE_PYTHON3` et `BR2_PACKAGE_PYTHON_PYQT5` : Moteur d'exécution de l'application.
  - **Problème de Polices :** Lors des premiers tests, l'interface graphique s'affichait sans texte (carrés blancs). Pour corriger cela, nous avons dû activer manuellement

les paquets de polices : `BR2_PACKAGE_DEJAVU`, `BR2_PACKAGE_LIBERATION` et `BR2_PACKAGE_FONT_AWESOME` et d'autres paquets.

---

5. IMPLÉMENTATION ET PERSONNALISATION (OVERLAY)

La personnalisation du système de fichiers se fait via le mécanisme de "Rootfs Overlay". Le chemin configuré est `BR2_ROOTFS_OVERLAY="/home/hch/buildroot/overlay"`. Tous les fichiers présents dans ce dossier sont copiés écrasés sur l'image finale.

5.1. Structure du Rootfs

L'arborescence de notre dossier overlay est structurée comme suit pour organiser les configurations et l'application :

```
overlay
├── etc
│   ├── dnsmasq.conf            # Config DHCP
│   ├── hostapd
│   │   └── hostapd.conf        # Config SSID "TankControl"
│   ├── init.d                  # Scripts de démarrage
│   │   ├── S05splash
│   │   ├── S40wifi             # Script réseau critique
│   │   └── S57app              # Lancement application
│   ├── mosquitto
│   │   └── mosquitto.conf
│   └── network
│       └── interfaces
├── home
│   └── pi
│       └── splash.png          # Image source Splash
└── usr
    └── bin
        └── TankApp             # Application complète
            ├── ressources      # Images (png, jpg, ui)
            │   ├── background.png
            │   ├── gearbox.jpg
            │   └── ...
            └── Tank_Interface_GUI.py
```

5.2. Gestion du réseau et des services (Script S40wifi)

Le script `/etc/init.d/S40wifi` est le cœur de l'initialisation réseau. Il a été conçu pour être robuste aux erreurs et séquentiel.

Logique du script :

1. Vérification de la présence du module `brcmfmac`. S'il est absent, tentative de chargement via `modprobe`.
2. Vérification de l'existence des fichiers de configuration (hostapd, dnsmasq).
3. Initialisation de l'interface : `ip link set up`.
4. Passage en mode AP : `iw dev wlan0 set type __ap`.
5. Attribution IP statique (192.168.4.1).
6. Lancement des démons : Hostapd, Dnsmasq et Mosquitto.
7. Gestion propre de l'arrêt (Killall et flush IP).

(Voir Annexe pour le code complet du script)

5.3. Interface graphique et démarrage (Script S57app)

Le script `S57app` lance l'application TankApp. Une particularité importante est la gestion de l'affichag. Nous utilisons la variable d'environnement : `export QT_QPA_PLATFORM=linuxfb`. Cela force Qt à dessiner directement dans le Framebuffer Linux (`/dev/fb0`), économisant ainsi les ressources d'un gestionnaire de fenêtres complet. Le script utilise `nohup` pour détacher le processus Python du terminal de boot.

5.4. Intégration du Splash Screen

Plutôt que d'utiliser un outil user-space comme `psplash` qui démarre tardivement, nous avons opté pour l'intégration dans le noyau via l'option Buildroot : `BR2_LINUX_KERNEL_CUSTOM_LOGO_PATH="/home/hch/buildroot/overlay/home/pi/splash.png"` Ceci permet d'afficher le logo "Tank Project" dès les premières secondes, masquant le défilement du texte du noyau et offrant un rendu professionnel.

6. DÉVELOPPEMENT ET MÉTHODOLOGIE DE DÉBOGAGE

6.1. Environnement de compilation

La compilation complète du système prend du temps. Nous avons optimisé ce processus en utilisant la commande `make -j16` sur le serveur de compilation, permettant d'utiliser 16 cœurs en parallèle.

6.2. Approche de résolution de problèmes en 7 étapes

Face aux difficultés d'initialisation du WiFi sur le RPi5, j'ai développé et appliqué une méthodologie de débogage stricte "bas niveau vers haut niveau" :

1. **Matériel :** `dmesg | grep -i mmc` (Vérifier que le bus SDIO voit la carte).
2. **Driver :** `dmesg | grep -i brcm` (Vérifier que le module noyau est chargé).
3. **Firmware :** `dmesg | grep -i firmware` (Vérifier que le fichier .bin est trouvé - **Étape critique**).
4. **Interface :** `ip link` (Vérifier l'existence de wlan0).
5. **Blocage RF :** `rfkill list` (Vérifier si le WiFi est "Hard/Soft blocked").
6. **Services :** `ps | grep hostapd` (Vérifier que les démons tournent).
7. **Logs Applicatifs :** Consultation des fichiers logs générés par nos scripts dans `/tmp`.

6.3. Analyse des erreurs rencontrées (Logs et Solutions)

Erreur A : Firmware manquant Lors des premiers builds, le WiFi échouait. L'analyse via console série a montré :

Plaintext

```
[ 3.452155] brcmfmac: brcmf_sdio_probe_attach: Direct firmware load for brcm/brcmfmac43455-sdio.bin failed with error -2
```

Solution : L'erreur -2 signifie "Fichier introuvable". J'ai corrigé cela en activant `BR2_PACKAGE_BRCMFMAC_SDIO_FIRMWARE_RPI_WIFI` et en vérifiant manuellement la présence des fichiers dans `/lib/firmware/brcm/` sur la cible.

Erreur B : Permissions des scripts Les services ne démarraient pas au boot. *Cause :* Les fichiers dans `init.d` n'avaient pas les droits d'exécution. *Solution :* Application de `chmod +x` sur les fichiers dans le dossier overlay source et vérification sur la cible (`ls -l /etc/init.d/`).

Erreur C : Interface graphique incomplète L'application se lançait mais les boutons étaient vides. *Solution :* Installation des polices `font-awesome` et `dejavu` dans Buildroot pour permettre à Qt de rendre le texte.

---

7. TESTS ET VALIDATION FONCTIONNELLE

Une fois l'image flashée sur la carte SD, les tests suivants ont validé le fonctionnement.

7.1. Validation SSH et Réseau

Depuis un PC connecté au SSID "TankControl" (IP client 192.168.4.15) :

Bash

```
$ ssh root@192.168.4.1
# ifconfig wlan0
wlan0     Link encap:Ethernet  HWaddr B8:27:EB:XX:XX:XX
          inet addr:192.168.4.1  Bcast:192.168.4.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
```

Le résultat confirme que le Pi agit bien comme routeur et que le serveur SSH répond.

7.2. Validation MQTT

Sur le Raspberry Pi, nous avons écouté le trafic MQTT local. Lors de l'utilisation de l'interface graphique :

Plaintext

```
# mosquitto_sub -h localhost -t "#" -v
Tanks/Tank_1/cmd/move left 20
Tanks/Tank_1/cmd/move right 15
```

Ces logs confirment que l'application GUI envoie correctement les commandes au broker Mosquitto local.

7.3. Validation Visuelle (GUI et Boot)

- **T+0s :** Mise sous tension.
- **T+5s :** Affichage du Splash Screen (Logo personnalisé).

- **T+15s :** Apparition de l'application TankApp. Les images (`gearbox.png`, flèches) sont correctement chargées grâce aux chemins relatifs définis dans `/usr/bin/TankApp`.

---

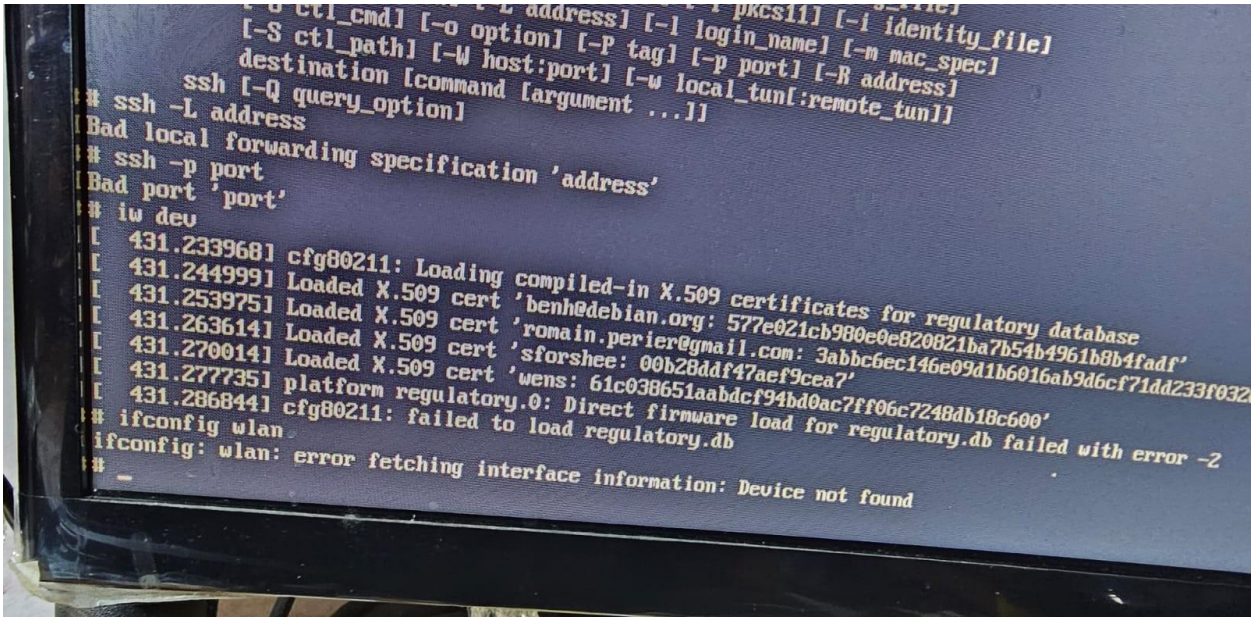
8. CONCLUSION

Ce projet a permis de démontrer la faisabilité de créer un système embarqué complexe et autonome sur Raspberry Pi 5 en utilisant uniquement Buildroot. Nous avons réussi à intégrer une pile réseau complète (AP, DHCP, MQTT) et une interface graphique réactive, le tout démarrant en moins de 15 secondes.

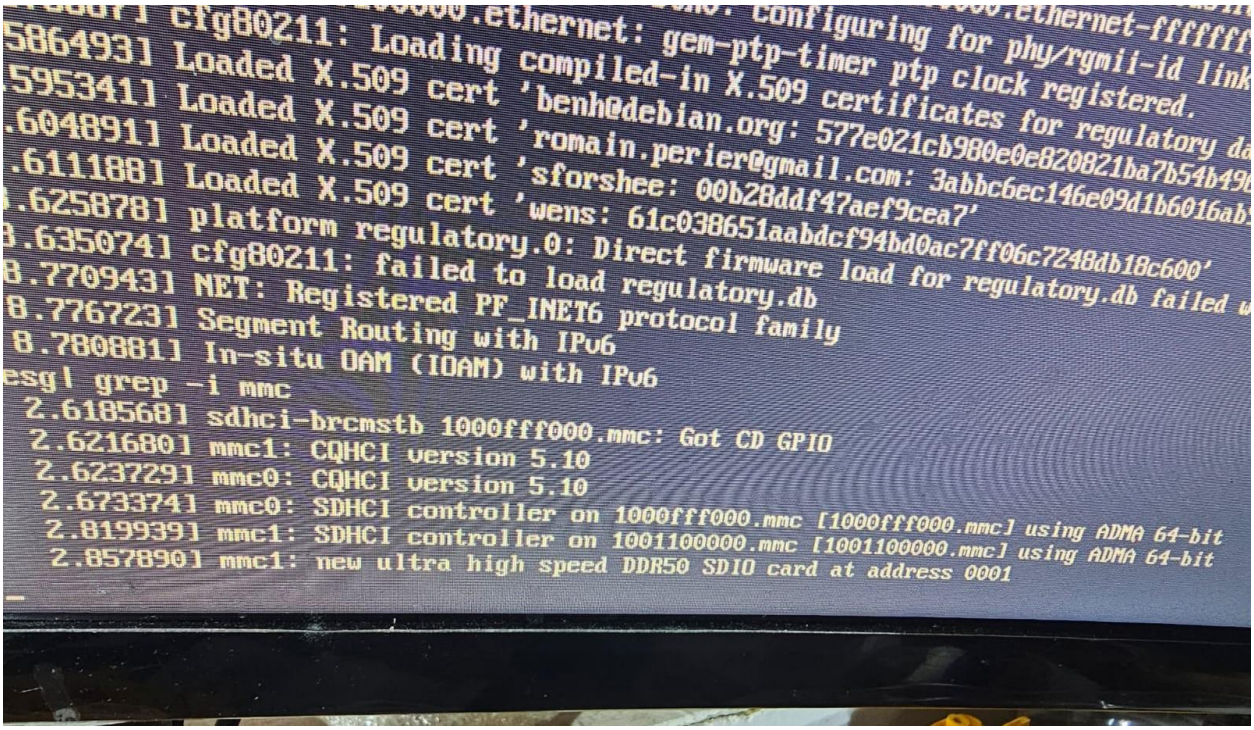
Les principaux défis techniques concernaient la gestion des firmwares propriétaires Broadcom et la configuration fine de Qt5 . La méthodologie de débogage par couches (Hardware > Kernel > Userspace) a été déterminante pour stabiliser le produit. Le système final répond à toutes les exigences du cahier des charges et offre une base solide pour des applications robotiques embarquées.

---

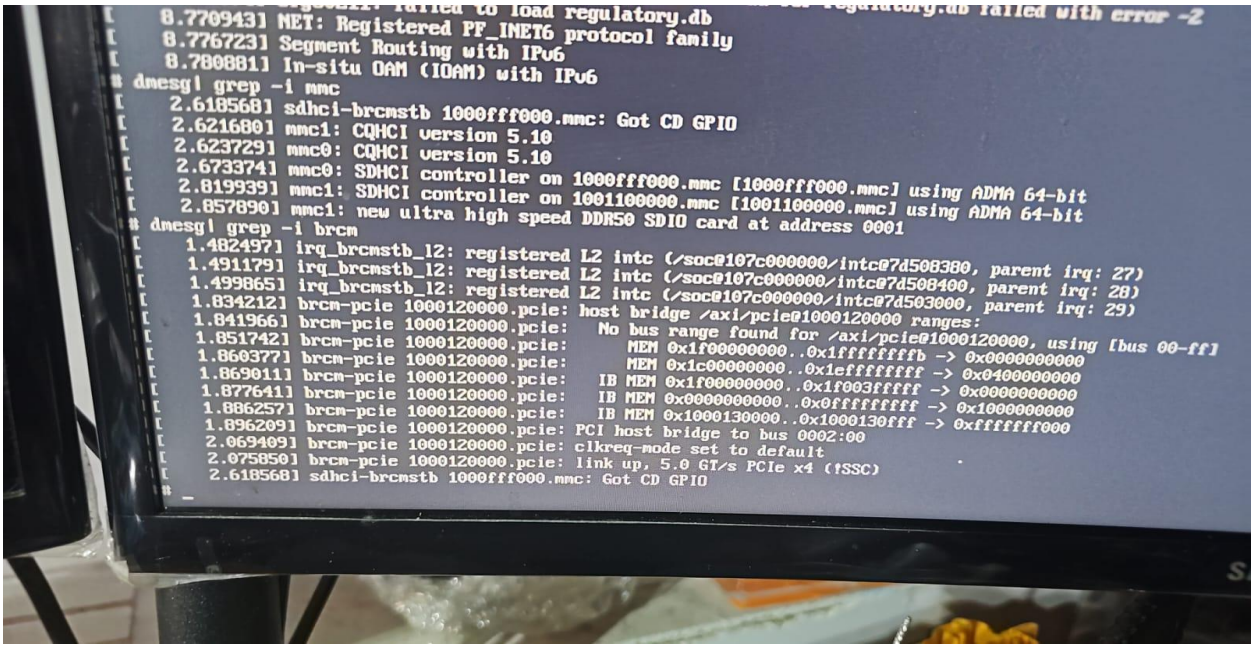
9. ANNEXES (PHOTOS PENDANT LE DÉVELOPPEMENT)



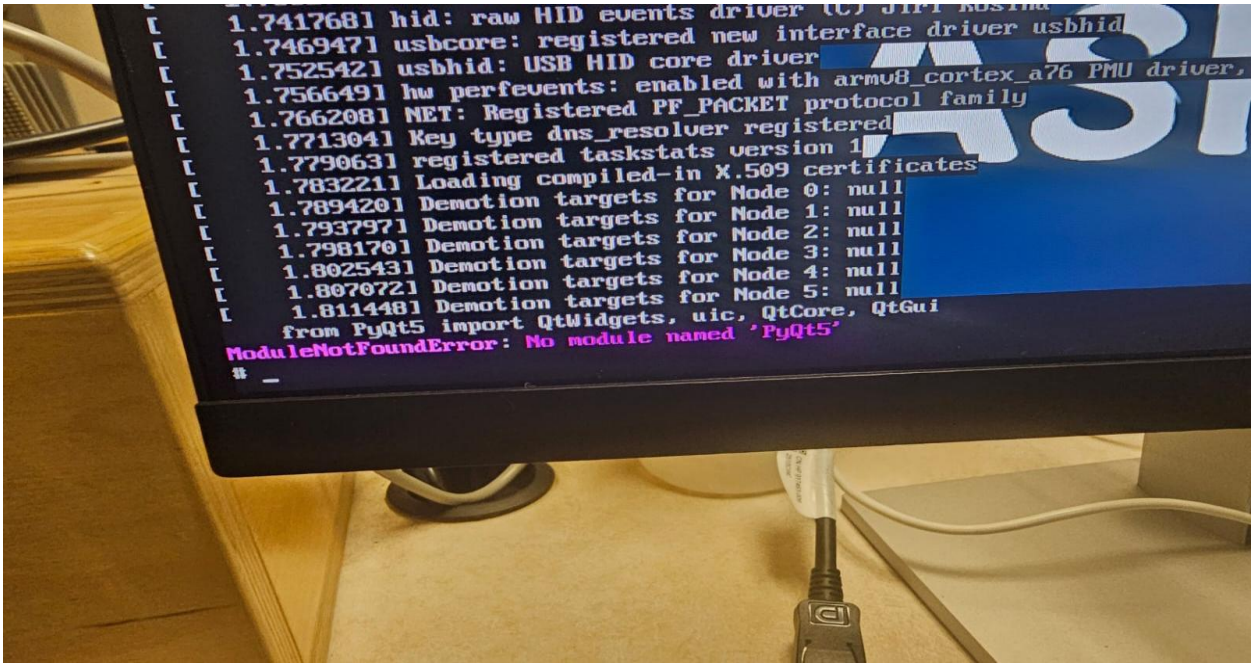
Problème Wlan



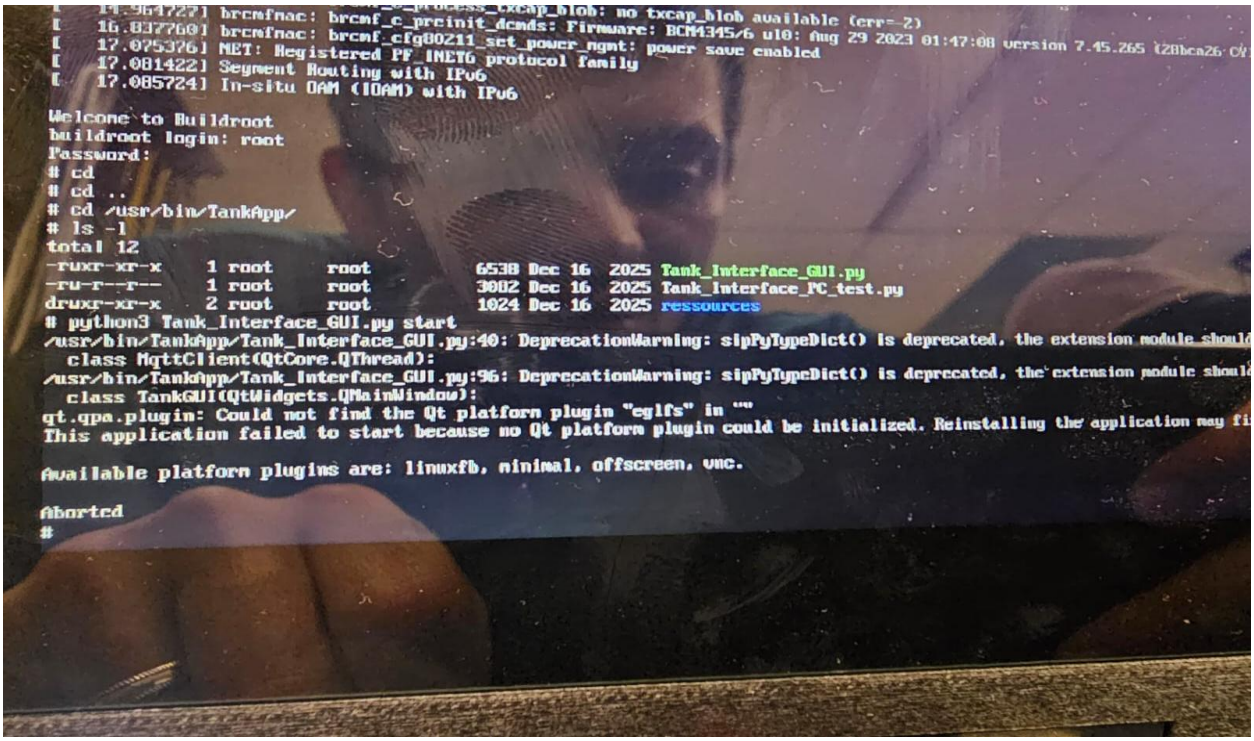
Périphérique détecté



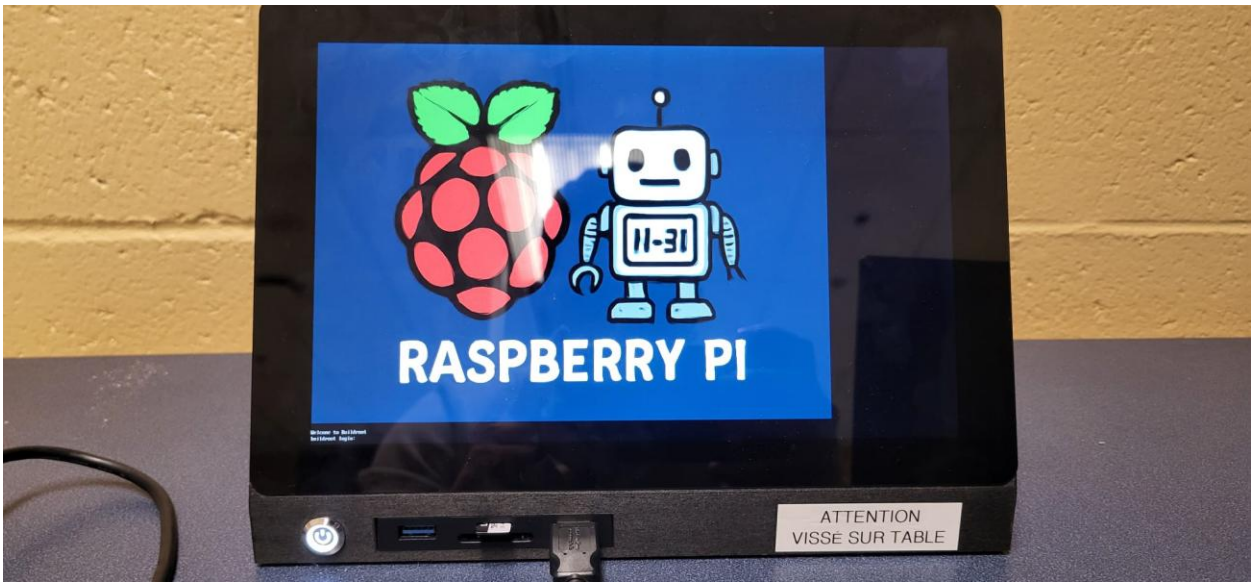
Brcmfmac pas activé



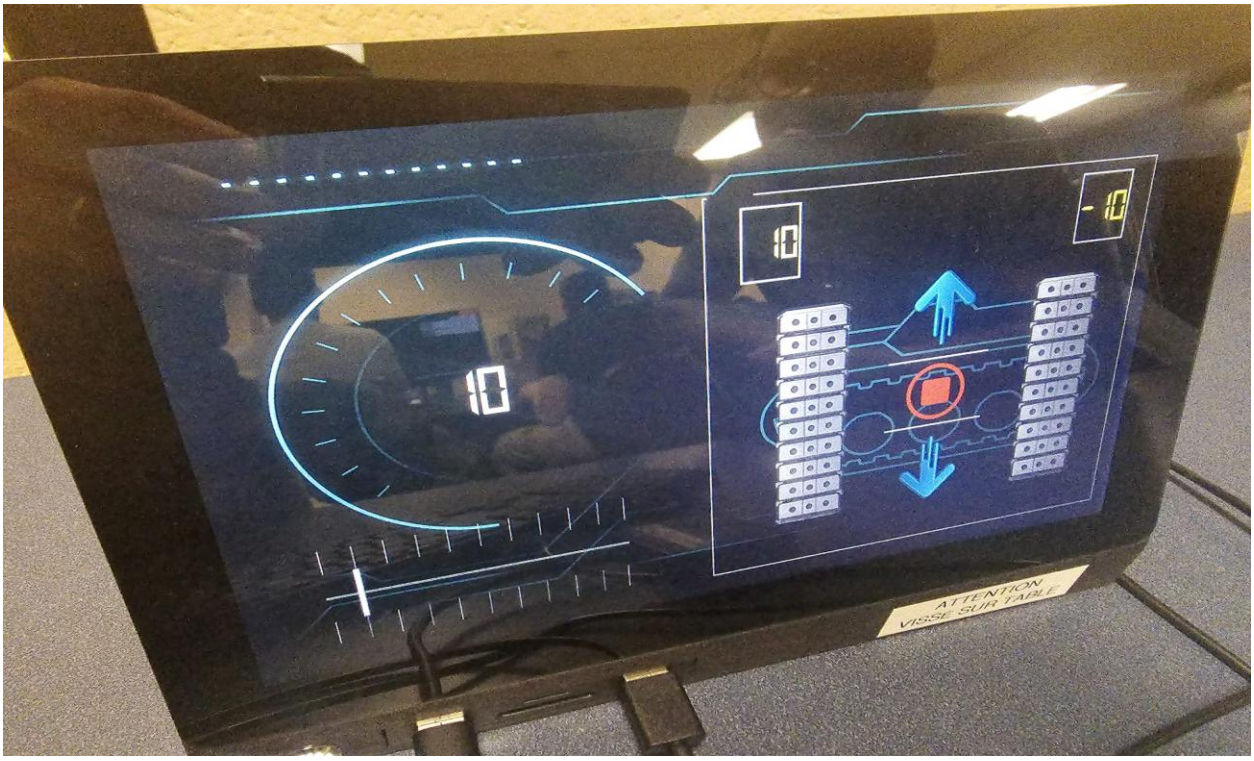
PyQt5 librairie problème avec l'app



Ici le problème avec la librairie QT5 un Make clean a résolu le problème



Mon splash personnalisé



Aperçu de l’app