Protocole d'utilisation — Connexion & applications UDP MicroZed / Zynq sous PetaLinux 2014

Objectif

Ce document explique comment utiliser le projet. On décrit l'utilisation de **Linux embarqué (PetaLinux 2014)**.

1 Pré-requis (commun aux 3 applications)

1.1 Câblage et réseau

.

Brancher le câble Ethernet entre le PC et la carte.
Fixer les IP en **statique**.
Exemple utilisé :

— Carte : `192.168.1.50`

— PC : `192.168.1.100`

La communication PC ↔ carte se valide avec `ping`.

1.2 Accès console série

— Ouvrir une console série (ex : PuTTY).

— Se connecter au port COM de la carte.

— Identifiant : **root** et PW : **root**

On utilise la console pour saisir les commandes Linux.

1.3 Configuration IPv4 de l'interface Ethernet (sur la carte)

PetaLinux minimal ne configure pas toujours IPv4 automatiquement. On force l'IP de la carte :

```
ifconfig -a
ifconfig eth0 192.168.1.50 netmask 255.255.255.0 up
```

Ensuite :

```
ping 192.168.1.100
```

Et depuis le PC :

```
ping 192.168.1.50
```

Si le ping marche dans les deux sens, le réseau est OK.

1.4 Exécuter des binaires sur la carte (méthode SD)

On ne compile pas sur la carte. On **cross-compile** sur le PC, puis on copie sur SD.

1) Monter la carte SD (sur la carte)

```
ls /dev/mmc*
cat /proc/partitions
mkdir -p /mnt/sd
mount -t vfat /dev/mmcblk0p1 /mnt/sd
ls -la /mnt/sd
```

2) Lancer un exécutable depuis la SD

```
cd /mnt/sd
chmod +x mon_binaire
./mon_binaire
```

3) (Option) Installer dans le rootfs

Pour ne plus dépendre de la SD :

```
cp /mnt/sd/mon_binaire /root/
chmod +x /root/mon_binaire
/root/mon_binaire
```

Ou en standard :

```
cp /mnt/sd/mon_binaire /usr/bin/
chmod +x /usr/bin/mon_binaire
mon_binaire
```

1.5 Pare-feu Windows (important pour UDP)

Si Wireshark voit les paquets mais Python ne reçoit rien. Le profil réseau Windows est souvent en **Public**. Il faut passer en **Privé**.

Commande PowerShell admin :

```
Set-NetConnectionProfile -InterfaceAlias "Ethernet" -NetworkCategory Private
```

2 Application 1 — Serveur UDP `LedOn/LedOff`

2.1 Principe

La carte exécute `udp_led`. Le programme écoute en UDP. Le PC envoie :

— `LedOn` → LED ON

— `LedOff` → LED OFF

La carte répond `OK` ou `ERR`.

2.2 GPIO utilisé (LED D3)

La LED D3 est reliée à `PS_MIO47`. Le `gpiochip` observé est `138`. Donc GPIO Linux = `138 + 47 = 185`.

Commandes sysfs (rappel) :

```
echo 185 > /sys/class/gpio/export
echo out > /sys/class/gpio/gpio185/direction
echo 1 > /sys/class/gpio/gpio185/value # ON
echo 0 > /sys/class/gpio/gpio185/value # OFF
```

2.3 Port UDP

Le serveur écoute sur le port `50000`.

2.4 Étapes (côté carte)

1. Configurer l'IP :

```
ifconfig eth0 192.168.1.50 netmask 255.255.255.0 up
```

2. Monter la SD et lancer :

```
mkdir -p /mnt/sd
mount -t vfat /dev/mmcblk0p1 /mnt/sd
cd /mnt/sd
chmod +x udp_led
./udp_led
```

2.5 Étapes (côté PC)

Envoyer une commande UDP avec `netcat` :

```
echo -n "LedOn" | nc -u 192.168.1.50 50000
echo -n "LedOff" | nc -u 192.168.1.50 50000
```

Résultat attendu : `OK`.

3 Application 2 — Envoi d'image UDP + réception PC (Python)

3.1 Principe

UDP ne garantit pas l'ordre. Donc on découpe l'image en **chunks**. Chaque chunk a un **header**. Le PC reconstitue l'image quand tout est reçu.

3.2 Mini-protocole (format des datagrammes)

Taille payload : `CHUNK_SIZE = 1400` octets (évite la fragmentation MTU 1500).
Header binaire (big-endian réseau), total 16 octets :

— `magic` (2) : `0xCAFE`
— `version` (1) : `1`
— `flags` (1) : réservé
— `frame_id` (4) : ID image
— `chunk_id` (2) : index chunk
— `total_chunks` (2) : nombre total
— `payload_len` (2) : taille utile
— `reserved` (2) : complétion à 16 octets

Le datagramme UDP contient :

`[header(16)] + [payload(payload_len)]`

3.3 Ports et IP

— Carte (sender) : `192.168.1.50`
— PC (receiver) : `192.168.1.100`
— Port UDP réception PC : `50001`

3.4 Étapes (côté PC) : réception Python

1. Vérifier le pare-feu Windows (profil **Privé**).
2. Lancer le script :

```
python udp_img_receiver.py 50001 reconstructed.jpg
```

3. Logs attendus :

```
Listening UDP on port 50001...
Receiving frame_id=1 from ('192.168.1.50', xxxx)
got 10/24
got 20/24
Frame complete: wrote XXXXX bytes to reconstructed.jpg
```

4. Ouvrir l'image :

```
start reconstructed.jpg
```

3.5 Étapes (côté carte) : envoi de l'image

1. Configurer l'IP :

```
ifconfig eth0 192.168.1.50 netmask 255.255.255.0 up
```

2. Copier binaire et image (exemple) :

```
mkdir -p /mnt/sd
mount -t vfat /dev/mmcblk0p1 /mnt/sd

cp /mnt/sd/udp_img_sender /root/
cp /mnt/sd/image.jpg /root/
chmod +x /root/udp_img_sender
```

3. Lancer l'envoi :

```
./udp_img_sender 192.168.1.100 50001 /root/image.jpg 1
```

Sortie typique :

```
Sent file 'image.jpg' (32531 bytes) as frame_id=1 in 24 chunks
```

4 Application 3 — LHM (pilotage bas niveau / extension projet)

4.1 But

LHM regroupe les actions **bas niveau** pour piloter le matériel. Exemple typique :

— Lire / écrire des registres (capteur).

— Ajouter des commandes réseau (ex : `GetFrame`, `SetExposure`, `SetGain`).

4.2 Point bloquant identifié : driver `spidev`

Sur l'image Linux utilisée :

— Le kernel ne contient pas `spidev`.

— Le device tree seul ne suffit pas.

— Message typique : `# Config_SPI_SPIDEV is not set`.

4.3 Ce que l'utilisateur doit vérifier (sur la carte)

1. Vérifier si `spidev` existe :

```
ls -l /dev/spidev*
```

2. Si rien n'apparaît, alors LHM ne peut pas utiliser SPI via `/dev/spidevX.Y`.
3. Dans ce cas, rester sur les applications UDP validées (LED + image).

Récapitulatif rapide

Toujours commencer par :

```
ifconfig eth0 192.168.1.50 netmask 255.255.255.0 up
```

LED :

— Carte : `./udp_led`

— PC : `echo -n "LedOn" | nc -u 192.168.1.50 50000`

Image :

— PC : `python udp_img_receiver.py 50001 reconstructed.jpg`

— Carte : `./udp_img_sender 192.168.1.100 50001 image.jpg 1`

Si Python ne reçoit pas (Windows) :

— Passer le réseau en **Privé** (PowerShell admin).