RAPPORT IT365

Nicolas Micha, Guillaume MARANINCHI

novembre 2025

Table des matières

Introduction

Nous avons réalisé deux travaux pratiques complémentaires ayant pour objectif d'explorer différentes approches du développement embarqué. Le premier porte sur la mise en œuvre d'un système Linux embarqué sur carte Raspberry Pi 3B, tandis que le second traite de la conception d'un objet connecté à l'aide de MicroPython sur Raspberry Pi Pico W. Ces deux expériences s'inscrivent dans une même continuité : comprendre comment un système embarqué est construit, configuré et connecté à un réseau.

Dans la première partie, nous avons appris à créer une distribution Linux minimale adaptée à une carte ARM. Cela comprend la génération d'un système de fichiers root (*root filesystem*), la compilation croisée du noyau, l'intégration d'applications de contrôle d'E/S et la mise en place de services réseau tels que HTTP, TCP/IP et SMTP. L'objectif était de maîtriser les outils et les étapes nécessaires pour faire fonctionner un système Linux complet sur un matériel embarqué.

La seconde partie transpose ces compétences à un environnement plus applicatif en utilisant la carte Raspberry Pi Pico W. Nous avons développé un objet connecté capable de mesurer la température et la pression ambiantes, d'envoyer ces données au broker MQTT d'Adafruit, puis de les afficher sur un tableau de bord en ligne. Ce travail nous a permis de manipuler la connectivité Wi-Fi, le protocole MQTT et les interactions avec un service cloud.

L'ensemble de ces deux TPs nous a permis de relier les aspects bas niveau du système (bootloader, noyau, drivers, E/S) aux couches applicatives plus hautes du monde de l'IoT. Nous avons ainsi mis en œuvre la chaîne complète de développement d'un système embarqué communicant, de la configuration matérielle jusqu'à l'échange de données avec Internet.

1 Mise en œuvre de Linux embarqué sur Raspberry Pi 3B

1.1 Introduction

L'objectif de ce TP est de concevoir et déployer une distribution Linux embarquée complète sur une carte Raspberry Pi 3B. Nous avons cherché à comprendre les différentes étapes qui permettent à un système Linux de démarrer et de fonctionner sur une plateforme matérielle à ressources limitées. Cela passe par la mise en place du `bootloader u-boot`, la compilation croisée du noyau Linux pour architecture ARM, et la création d'un système de fichiers minimal à l'aide de BusyBox.

Une fois la base système opérationnelle, nous avons développé plusieurs applications simples destinées à tester le bon fonctionnement du matériel et du système : affichage sur la console, contrôle des LEDs à travers le `Board Support Package (BSP)` et réalisation d'un chenillard. Nous avons ensuite ajouté une couche réseau en mettant en place un serveur HTTP (`Boa`), un échange client/serveur TCP et un client SMTP permettant l'envoi d'e-mails depuis la carte.

Ce TP nous a permis de comprendre concrètement la structure logicielle d'un système Linux embarqué et d'expérimenter toutes les étapes de sa mise en œuvre — de la génération du noyau jusqu'à l'exécution d'applications réseau sur la cible. Il s'agit d'une étape essentielle pour aborder la conception de systèmes embarqués communicants.

1.2 TP0 — Prise en main de l'environnement

La première étape consistait à mettre en place l'environnement de développement. La carte hôte utilisée était `ibicella` (adresse : 10.7.2.112) et la carte cible `rpi04` (adresse : 10.7.2.121). La connexion série a été établie avec `minicom -b 115200 -D /dev/ttyUSB0`, et la communication réseau via TFTP.

Le bootloader `u-boot` a servi à charger les différents éléments du système avant le lancement du noyau. Les macros pré-définies dans `u-boot` ont chacune un rôle précis : `gok` télécharge le noyau Linux, `gor` charge le système de fichiers root (RAM disk), `godtb` transfère le Device Tree Blob décrivant le matériel, `go` exécute le noyau à l'adresse indiquée, et `ramboot` regroupe l'ensemble de ces actions pour un démarrage complet automatique.

Une fois le système lancé, les applications compilées sur le PC hôte peuvent être transférées sur la cible par TFTP et exécutées directement à l'aide de :

```
RPi3# tftp -g -r mon_appli @IP_host
RPi3# chmod u+x mon_appli
RPi3# ./mon_appli
```

1.3 TP1 — Génération du RAM disk

Ce TP consistait à créer un système de fichiers `root_fs` chargé en mémoire (RAM disk), donc volatil. Le script `goskel` construit un squelette de système de fichiers minimal comportant les répertoires classiques (/bin, /sbin, /etc...).

Lors de la compilation de BusyBox, les variables `ARCH` et `CROSS_COMPILE` désignent respectivement l'architecture visée (ARM) et le préfixe du compilateur croisé `arm-buildroot-linux-gnueabihf-`. La commande d'installation copie ensuite les exécutables générés dans le dossier `root_fs`.

Le script `golib` ajoute les bibliothèques dynamiques nécessaires (libc, ld-linux...), puis `goramdisk` assemble le tout en une image compressée `ramdisk.img.gz` qui constitue le système de fichiers root à charger sur la carte. Le caractère « \» précédant la commande `cp` permet d'utiliser la commande système brute, sans tenir compte d'un éventuel alias.

1.4 TP2 — Compilation du noyau Linux et démarrage

La compilation du noyau Linux s'effectue avec la commande :

```
make ARCH=arm CROSS_COMPILE=arm-buildroot-linux-gnueabihf- zImage
```

Le noyau obtenu est de version `5.7.19-v7+`. Une fois copiés dans `/tftpboot/`, le noyau, le Device Tree et le RAM disk sont chargés en mémoire depuis `u-boot` à l'aide de `run ramboot`. Les traces affichées dans `minicom` montrent le démarrage du noyau, l'initialisation des périphériques et le montage du système de fichiers.

1.5 EX1 — Application `Hello World`

L'application `hello.c` affiche simplement « Hello World ! ». Le `Makefile` fourni est configuré pour la compilation croisée avec `arm-buildroot-linux-gnueabihf-gcc`. Après transfert par TFTP et exécution sur la Raspberry Pi, le message s'affiche correctement dans le terminal, validant la chaîne de compilation et de transfert.

1.6 EX2 — Chenillard

À l'aide du `BSP` (`bsp.c/h`), un chenillard a été réalisé sur les six LEDs de la carte d'E/S. L'initialisation s'effectue par `BSP_init()`, qui configure les GPIO et éteint toutes les LEDs au départ. Le programme alterne ensuite leur allumage de LED1 à LED6, puis dans le sens inverse, avec une temporisation d'environ 100 ms entre chaque étape. Les fonctions `BSP_setLED()` et `BSP_clrLED()` permettent respectivement l'allumage et l'extinction des LEDs, tandis que `BSP_release()` libère les ressources à la fin du programme.

1.7 EX3 — Serveur Web embarqué (Boa)

Cet exercice introduit la mise en place d'un serveur HTTP embarquée nommé Boa, utilisé pour piloter les LEDs à distance via une interface Web. Le script `go` compile le serveur, et `goinstall` copie les exécutables et fichiers de configuration dans le système de fichiers root. Après redémarrage, le serveur est lancé sur la cible avec `boa -c /etc`.

Les scripts CGI écrits en C utilisent des appels `printf()` pour générer les réponses HTTP, tandis que la variable d'environnement `QUERY_STRING` permet de récupérer les paramètres transmis par le navigateur. Le programme `putleds.c` interprète ces paramètres et pilote les LEDs : par exemple, l'URL `http://@IP_cible/cgi-bin/putleds?1` allume les LEDs et `?0` les éteint.

1.8 EX4 — Application client/serveur TCP

Dans cet exercice, un serveur TCP (`myserver`) a été développé sur la carte RPi afin de recevoir des commandes de contrôle des LEDs envoyées depuis un client Linux (`myclient`) exécuté sur le PC hôte. Chaque requête comporte deux octets : le premier indique le numéro de LED (1 à 6), le second son état (0 pour éteinte, 1 pour allumée). Le serveur renvoie un code 0 en cas de succès et 1 en cas d'erreur.

Le serveur repose sur une socket AF_INET de type SOCK_STREAM et fonctionne de manière itérative sur le port 2000. Le client envoie successivement les ordres nécessaires pour créer un chenillard à distance. Le code client est compilé nativement sur le PC hôte tandis que le serveur est compilé en croisé pour la cible.

1.9 EX5 — Client SMTP sous Linux

L'objectif est ici de comprendre le protocole SMTP en dialoguant directement avec le serveur de messagerie du PC hôte. Le protocole est textuel et basé sur un échange commande/réponse. Une session typique commence par `MAIL FROM:`, suivie de `RCPT TO:`, `DATA` puis `QUIT`.

L'application `mysmtp` implémente ces échanges en C, en utilisant des sockets TCP pour communiquer avec le serveur local (Postfix). Elle permet d'envoyer un message au format texte ou HTML à l'utilisateur du PC hôte (`se01`).

1.10 EX6 — Client SMTP sous Linux embarqué

Cet exercice reprend le client précédent, porté sur la carte RPi. L'entrée `linux01` est ajoutée dans le fichier `/etc/hosts` de la cible pour associer l'adresse du PC hôte. L'application `mysmtp`, compilée en croisé et transférée par TFTP, envoie alors un courriel depuis la cible vers le serveur SMTP du PC hôte. L'adresse de l'expéditeur est `rpi04@linux01`

et celle du destinataire `se01@linux01`. L'envoi est validé par la réception du message sur le poste hôte à l'aide de la commande `mail`.

Cette méthode de connectivité, bien que basique, illustre la capacité d'un système embarqué à utiliser des protocoles standards pour notifier ou échanger des informations, par exemple dans des applications IoT.

1.11 Conclusion

Ce travail a permis de comprendre la structure et le fonctionnement d'un système Linux embarqué sur Raspberry Pi 3B. De la génération du système de fichiers à la compilation du noyau et au développement d'applications locales ou réseau, l'ensemble du flux de mise en œuvre a été reproduit et testé avec succès. Les exercices abordés démontrent la flexibilité de Linux embarqué et la facilité avec laquelle on peut y intégrer des services variés : contrôle matériel, serveur HTTP ou client SMTP. Ces compétences constituent une base solide pour la conception de systèmes connectés et d'objets intelligents sous Linux.

2 Contrôle d'un objet connecté avec Micropython

Introduction

Dans ce nous utilisons la carte **Raspberry Pi Pico W** et l'environnement **MicroPython**. L'objectif principal de ces travaux est de comprendre comment développer, déployer et contrôler un système embarqué capable de collecter des données environnementales, de les transmettre via un réseau Wi-Fi, et de les visualiser en temps réel.
La configuration matérielle utilisée dans ce TP repose sur deux éléments principaux :

- **Raspberry Pi Pico W** : C'est la carte cible sur laquelle les programmes MicroPython sont développés et exécutés. Elle est équipée d'un processeur ARM Cortex-M0+ et d'une connectivité Wi-Fi.
- **Carte Maker Pi Pico** : Cette carte d'accueil se connecte à la Raspberry Pi Pico W pour étendre ses fonctionnalités. Elle fournit des périphériques supplémentaires tels que des LEDs, des boutons, un socket pour le capteur BME680...

Ce rapport détaille les étapes suivies, les résultats obtenus, ainsi que les défis rencontrés et les solutions apportées. Il vise à fournir une compréhension claire et complète de la mise en œuvre d'un objet connecté utilisant MicroPython.

2.1 Application "Hello World"

L'objectif de cet exercice est d'afficher le message "Hello World !" dans la console de la Raspberry Pi Pico W.

Le code utilisé pour cet exercice est le suivant :

```
1 print("Hello World!")
```

En exécutant ce code, le message "Hello World !" est affiché dans la console de Thonny.

2.2 Clignotement d'une LED

L'objectif de cet exercice est de faire clignoter une LED connectée à la broche GP10.
Voici le code utilisé :

```
1  from machine import Pin
2  import time
3
4  p0 = Pin(10, Pin.OUT)
5
6  while(True):
7      p0.on()
8      time.sleep(1)
9      p0.off()
10     time.sleep(1)
```

On accède aux broches de la carte avec la bibliothèque **machine**.
En exécutant ce code, la LED clignote toutes les secondes.

2.3 Application de lecture des capteurs BME680

L'objectif de cet exercice est de lire les valeurs de température, d'humidité et de pression atmosphérique à partir du capteur BME680 connecté à la Raspberry Pi Pico W

via le bus I2C.

Voici le code utilisé pour lire les données du capteur BME680 :

```
from machine import *
import time
from bme680 import *

i2c = I2C(0, scl=Pin(1), sda=Pin(0), freq=400_000)
bme = BME680_I2C(i2c=i2c)

while(True):
    temp = str(round(bme.temperature,2)) + ' C  '
    hum = str(round(bme.humidity,2)) + '% '
    press = str(round(bme.pressure,2)) + 'hPa '
    time.sleep(1)

    print(temp, hum, press)
```

On utilise la bibliothèque spécifique du capteur BME680. Puis, on initialise le bus I2C avec les broches SCL (GP1) et SDA (GP0) à une fréquence de 400 kHz. De plus, lors de la récupération des mesures du capteur, on arrondit la valeur à deux décimales (**round**).

En exécutant ce code, les valeurs de température, d'humidité et de pression sont affichées dans la console toutes les secondes. Par exemple :

```
25.5°C 50.25% 1013.25hPa
```

2.4 Connexion Wi-Fi

L'objectif de cet exercice est de connecter la Raspberry Pi Pico W à un réseau Wi-Fi.

```
import machine
import network
import socket
import time

WIFI_SSID = "SE"
WIFI_PASSWORD = "sesesese"

wlan = network.WLAN(network.STA_IF)
wlan.active(True)
wlan.connect(WIFI_SSID, WIFI_PASSWORD)

while wlan.isconnected() == False:
    print('Waiting for connection...')
    time.sleep(1)

print("Connected to Wifi " + WIFI_SSID)
print(wlan.ifconfig())
```

— `wlan = network.WLAN(network.STA_IF)` : Initialise l'interface Wi-Fi en mode station (client).

— `wlan.active(True)` : Active l'interface Wi-Fi.

— `wlan.connect(WIFI_SSID, WIFI_PASSWORD)` : Se connecte au réseau Wi-Fi spécifié.

On affiche les informations de configuration réseau (adresse IP, masque de sous-réseau, passerelle, DNS).

```
Connected to Wifi SE
('192.168.5.68', '255.255.255.0', '192.168.5.1', '10.210.18.138')
```

2.5 Envoi de données MQTT

L'objectif de cet exercice est d'envoyer les données de température et de pression lues depuis le capteur BME680 à un broker MQTT (Adafruit IO) via une connexion Wi-Fi.

```
from machine import *
from bme680 import *
import network
import socket
import time
from mqtt import MQTTClient

i2c = I2C(0, scl=Pin(1), sda=Pin(0), freq=400_000)
bme = BME680_I2C(i2c=i2c)
WIFI_SSID = "SE"
WIFI_PASSWORD = "sesesese"

wlan = network.WLAN(network.STA_IF)
wlan.active(True)
wlan.connect(WIFI_SSID, WIFI_PASSWORD)
while wlan.isconnected() == False:
    print('Waiting for connection...')
    time.sleep(1)

client = MQTTClient("device_id", "io.adafruit.com", user="se04",
   password="aio_VGab60TD1k7HRKBeu5a3gTwc32A1", port=1883)

while True:
    print("j'envoi la temp")
    temp = str(round(bme.temperature,2)) + ' C  '
    client.connect()
    client.publish(topic="se04/feeds/temperature", msg=temp)
    client.disconnect()

    time.sleep(5)
    print("j'envoi la press")
    press = str(round(bme.pressure,2)) + 'hPa '

    client.connect()
    client.publish(topic="se04/feeds/pressure", msg=press)
    client.disconnect()

    time.sleep(5)
```

Pour gérer la communication MQTT, on crée notre client (l.20) pour se connecter au broker Adafruit IO.

- `client.connect()` : Se connecte au broker MQTT.
- `client.publish(topic="se04/feeds/temperature", msg=temp)` : Publie la température sur le topic spécifié.
- `client.disconnect()` : Se déconnecte du broker MQTT.
- `time.sleep(5)` : Attend 5 secondes avant d'envoyer la pression.

En exécutant ce code, les données de température et de pression sont envoyées au broker MQTT toutes les 5 secondes.

2.6 Dashboard Adafruit

Sur le site io.adafruit.com, on peut créer des widgets afin d'afficher nos données de manières différentes.

2.7 Client MQTT

```
1  import paho.mqtt.client as mqtt
2
3  MQTT_SERVER = "io.adafruit.com" # MQTT server address
4  MQTT_TOPIC = "se04/feeds/temperature" # Topic name
5
6  def on_connect(client, userdata, flags, rc):
7   print("Connection : " + str(rc))
8   # Subscribe to the topic
9   client.subscribe(MQTT_TOPIC)
10
11 # A publish message is received from the server
12 def on_message(client, userdata, msg):
13  print("Sujet : " + msg.topic + " Message : " + str(msg.payload))
14
15 client = mqtt.Client()
16
17 client.username_pw_set(username="se04", password="
      aio_VGab60TD1k7HRKBeu5a3gTwc32A1")
18
19 client.on_connect = on_connect
20 client.on_message = on_message
21 client.connect(MQTT_SERVER, 1883, 60)
22
23 client.loop_forever()
```

Ce code est exécuté sur la carte RPi hôte.

Importation des modules

- **paho.mqtt.client** : Module pour gérer la communication MQTT.

Configuration du client MQTT

— `MQTT_SERVER = "io.adafruit.com"` : Adresse du broker MQTT.
— `MQTT_TOPIC = "se04/feeds/temperature"` : Topic auquel le client s'abonne.

Fonctions de rappel

— **on_connect** : Affiche un message lorsque le client se connecte au broker.
— **on_message** : Affiche un message lorsqu'un message est reçu sur le topic abonné.

Connexion et abonnement

— `client.connect(MQTT_SERVER, 1883, 60)` : Se connecte au broker MQTT sur le port 1883 avec un timeout de 60 secondes.
— `client.subscribe(MQTT_TOPIC)` : S'abonne au topic spécifié.
— `client.loop_forever()` : Boucle infinie pour recevoir les messages.

3 Conclusion

Au terme de ces deux travaux pratiques, nous avons exploré deux approches complémentaires de l'informatique embarquée et des objets connectés.

Dans un premier temps, le TP Linux embarqué nous a permis de comprendre les fondements d'un système d'exploitation minimal fonctionnant sur une carte Raspberry Pi. Nous avons appris à compiler un noyau Linux, à créer un système de fichiers root (RAM disk) et à déployer des applications embarquées telles que le chenillard, un serveur web HTTP, un client serveur TCP et un client SMTP. Ces étapes ont montré concrètement comment construire et maîtriser une distribution Linux sur mesure, adaptée à une architecture matérielle donnée, tout en développant des outils de communication réseau à bas niveau.

Dans un second temps, le TP IoT (MicroPython) a introduit une approche plus légère et orientée objets connectés à l'aide de la carte Raspberry Pi Pico W. Nous avons expérimenté la programmation en MicroPython pour piloter des capteurs (température, pression), établir une connexion Wi-Fi et transmettre des données à un broker MQTT hébergé sur la plateforme io.adafruit.com. La création d'un dashboard Adafruit a permis de visualiser en temps réel les mesures issues de la carte, illustrant l'intégration complète entre matériel, connectivité et interface utilisateur.