Développement d'une télécommande infrarouge Yamaha sur Raspberry Pi 5 : Retour d'expérience technique

Nicolas Loisy - Février 2025

Résumé du projet

Ce document retrace le développement d'une télécommande infrarouge pour amplificateur Yamaha sur Raspberry Pi 5. L'objectif était de créer un système capable d'envoyer des commandes infrarouges (allumer, changer le volume, changer la source audio, etc.) exactement comme le ferait une vraie télécommande.

Le projet a débuté par une validation de faisabilité sur Arduino, qui a fonctionné immédiatement. La migration vers Raspberry Pi 5, qui était l'objectif final, a révélé des défis techniques importants liés à la précision temporelle. Après plusieurs itérations et optimisations, le système fonctionne désormais avec une fiabilité comparable à Arduino, validant l'approche pour une intégration domotique.

---

1. Contexte et objectif du projet

1.1 Le besoin initial

J'ai un amplificateur Yamaha que je contrôle habituellement avec sa télécommande infrarouge. Mon objectif était de pouvoir contrôler cet amplificateur depuis un Raspberry Pi 5, pour l'intégrer dans un système domotique plus large. Concrètement, je voulais pouvoir :

- Allumer et éteindre l'amplificateur
- Augmenter ou diminuer le volume
- Changer la source audio (radio, lecteur CD, entrée auxiliaire, etc.)
- Contrôler la lecture (play, pause, stop)

1.2 Pourquoi l'infrarouge

Même si des protocoles modernes comme le WiFi ou le Bluetooth existent, beaucoup d'équipements audio-vidéo utilisent encore l'infrarouge. Mon amplificateur Yamaha ne possède pas de connectivité réseau, donc l'infrarouge était la seule option pour le piloter à distance.

1.3 L'approche choisie

J'ai décidé de procéder en deux étapes :

1. **Phase de validation avec Arduino** : Vérifier que je comprends bien le protocole et que l'approche fonctionne
2. **Migration vers Raspberry Pi 5** : Mon objectif final, plus complexe techniquement mais nécessaire pour l'intégration domotique

---

2. Comprendre le protocole infrarouge

2.1 Comment fonctionne l'infrarouge

Une télécommande infrarouge envoie des signaux lumineux invisibles à l'œil nu. Ces signaux sont émis par une LED spéciale qui émet dans l'infrarouge (longueur d'onde 940 nanomètres). Le récepteur dans l'amplificateur capte ces signaux et les décode pour exécuter les commandes.

2.2 Le protocole NEC

Les appareils Yamaha utilisent un protocole de communication appelé “NEC”. C’est un standard développé par l’entreprise japonaise NEC Corporation et largement utilisé dans l’électronique grand public.

Les éléments clés du protocole NEC :

a) La modulation à 38 kHz

Le signal n’est pas simplement une LED qui s’allume et s’éteint. La LED clignote très rapidement 38 000 fois par seconde (38 kHz). Cette fréquence de clignotement rapide permet au récepteur de distinguer le signal de la télécommande de la lumière ambiante.

Analogie : Imaginez que vous voulez envoyer un message morse avec une lampe de poche. Si vous allumez simplement la lampe, elle peut se confondre avec d’autres sources lumineuses. Mais si vous faites clignoter la lampe très rapidement pendant que vous envoyez votre morse, un récepteur spécial pourra reconnaître votre signal parmi toutes les autres lumières.

b) Le duty cycle (rapport cyclique)

Le duty cycle représente le pourcentage de temps pendant lequel la LED est allumée pendant chaque clignotement.

Un cycle à 38 kHz dure 26.3 microsecondes (un millionième de seconde). Avec un duty cycle de 33%, voici ce qui se passe :

```
Un clignotement (26.3 microsecondes au total)

 ALLUMÉ        ÉTEINT
 8.7µs         17.6µs
 (33%)          (67%)
```

La LED est allumée pendant 8.7 microsecondes, puis éteinte pendant 17.6 microsecondes. Ce cycle se répète 38 000 fois par seconde.

Pourquoi 33% et pas 50% : - Économie d’énergie : moins de temps allumé - Protection de la LED contre la surchauffe - Meilleure détection par le récepteur (les récepteurs IR sont optimisés pour ce ratio)

c) L’encodage des bits

Le protocole NEC encode les informations en modulant la durée des espaces entre les impulsions :

- **Bit “0”** : Impulsion courte de 560 microsecondes allumée, suivie de 560 microsecondes éteinte
- **Bit “1”** : Impulsion courte de 560 microsecondes allumée, suivie de 1690 microsecondes éteinte

d) La structure complète d’un message

Un message NEC complet ressemble à ceci :

1. **Signal de démarrage** : 9 millisecondes allumé + 4.5 millisecondes éteint (pour signaler le début)
2. **Adresse** : 8 bits qui identifient l’appareil (pour Yamaha : 0x78 en hexadécimal, soit 120 en décimal)
3. **Adresse inversée** : 8 bits (pour vérifier qu’il n’y a pas d’erreur)
4. **Commande** : 8 bits qui indiquent l’action (par exemple 0x0F pour POWER)
5. **Commande inversée** : 8 bits (vérification d’erreur)
6. **Bit d’arrêt** : 560 microsecondes

La durée totale d’un message est d’environ 67 millisecondes.

2.3 Les codes spécifiques Yamaha

J'ai identifié que mon amplificateur Yamaha utilise le code d'adresse 0x78. Voici quelques exemples de codes de commande :

Fonction	Code (hexadécimal)	Code (décimal)
POWER (allumer/éteindre)	0x0F	15
Volume +	0x1E	30
Volume -	0x1F	31
Play	0x02	2
Pause	0x0A	10
Stop	0x01	1

---

3. Première étape : validation avec Arduino

3.1 Pourquoi commencer par Arduino

Arduino est un microcontrôleur simple et peu coûteux. J'ai choisi de commencer avec Arduino pour deux raisons :

1. **Valider ma compréhension du protocole** : M'assurer que les codes fonctionnent vraiment
2. **Simplicité** : Arduino possède des bibliothèques dédiées à l'infrarouge qui simplifient le développement

3.2 Le matériel Arduino

Composants utilisés : - Arduino Uno (microcontrôleur ATmega328P) - LED infrarouge 940nm (5mm de diamètre) - Pas de résistance (la protection interne de l'Arduino suffit) - Fils de connexion

Câblage :

```
Arduino Pin 3  LED IR (patte longue : anode +)
Arduino GND  LED IR (patte courte : cathode -)
```

3.3 Le code Arduino

J'ai utilisé la bibliothèque "IRremote" qui gère automatiquement toute la complexité du protocole NEC.

```
#include <IRremote.h>

#define IR_SEND_PIN 3
#define YAMAHA_ADDRESS 0x78

void setup() {
    IrSender.begin(IR_SEND_PIN);
}

void sendPower() {
    // Premier envoi
    IrSender.sendNEC(YAMAHA_ADDRESS, 0x0F, 0);
    delay(108);  // Pause de 108 millisecondes
    // Deuxième envoi (nécessaire pour Yamaha)
    IrSender.sendNEC(YAMAHA_ADDRESS, 0x0F, 0);
}

void loop() {
```

```
    sendPower();
    delay(5000);  // Attendre 5 secondes entre chaque commande
}
```

3.4 Résultats Arduino

Le succès immédiat : Dès le premier essai, l'amplificateur Yamaha a répondu parfaitement. La télécommande fonctionnait à une distance de plus de 6 mètres sans aucun problème.

Pourquoi ça marche si bien sur Arduino :

Arduino possède des "timers hardware" (horloges matérielles internes) qui peuvent générer des signaux très précis sans intervention du processeur. La bibliothèque IRremote utilise le Timer2 pour générer automatiquement la porteuse à 38 kHz avec un duty cycle précis de 33%.

Avantages constatés : - Installation en 10 minutes - Code très simple (quelques lignes) - Fiabilité absolue (100% de réussite) - Portée excellente (6+ mètres) - Consommation électrique très faible

Limites pour mon projet : - Pas de connexion réseau native - Interface utilisateur limitée - Difficile à intégrer dans un système domotique complet

---

4. Migration vers Raspberry Pi 5

4.1 Pourquoi le Raspberry Pi 5

Le Raspberry Pi 5 est un nano-ordinateur qui exécute un système d'exploitation Linux complet. Il offre plusieurs avantages pour mon projet :

- **Connectivité réseau** : WiFi et Ethernet intégrés
- **Puissance de calcul** : Processeur 4 cœurs ARM Cortex-A76 à 2.4 GHz
- **Flexibilité logicielle** : Python, interfaces web, bases de données, etc.
- **Intégration domotique** : Peut communiquer avec d'autres systèmes

C'était donc la plateforme idéale pour mon objectif final d'intégration domotique.

4.2 Le matériel Raspberry Pi

Composants utilisés : - Raspberry Pi 5 (4GB RAM) - LED infrarouge 940nm (même que pour Arduino) - Câble duplex 1 mètre pour connecter les 2 LEDs - Pas de résistance (fonctionne sans, testée en conditions réelles)

Câblage :

```
Raspberry Pi GPIO 18 (Pin physique 12)  LED IR 1 (anode +)
Raspberry Pi GPIO 18 (Pin physique 12)  LED IR 2 (anode +)
Raspberry Pi GND (Pin physique 6)  LEDs IR (cathode -)
```

Note sur l'absence de résistance : Normalement, on devrait mettre une résistance de limitation de courant. Dans mon cas, le système fonctionne sans résistance car le GPIO du Raspberry Pi a une résistance interne et le duty cycle de 33% limite naturellement le courant moyen.

4.3 Premier obstacle : pigpio incompatible

Le problème : J'ai d'abord essayé d'utiliser "pigpio", une bibliothèque Python standard pour contrôler les GPIO du Raspberry Pi. Mais au démarrage, j'ai obtenu cette erreur :

```
gpioHardwareRevision: unknown rev code (d04170)
Sorry, this system does not appear to be a raspberry pi.
aborting.
```

Explication : Le Raspberry Pi 5 est tellement récent que pigpio ne le reconnaît pas. Le code "d04170" est l'identifiant du Raspberry Pi 5, que pigpio ne connaît pas encore.

La solution : J'ai dû migrer vers "lgpio", la nouvelle bibliothèque officielle pour le Raspberry Pi 5.

Différences entre pigpio et lgpio :

Aspect	pigpio (ancienne génération)	lgpio (Raspberry Pi 5)
Architecture	Nécessite un démon (service en arrière-plan)	Accès direct sans démon
Installation	`sudo systemctl start pigpiod` requis	Aucune configuration nécessaire
Latence	~100 microsecondes	~10 microsecondes
Compatibilité Pi 5	Non	Oui

4.4 Premier code Python avec lgpio

Voici ma première tentative de code :

```
import lgpio
import time

# Ouvrir l'accès GPIO
h = lgpio.gpiochip_open(0)
lgpio.gpio_claim_output(h, 18, 0)

# Paramètres 38 kHz
period_us = 26.3  # Période d'un cycle en microsecondes
cycles = 1000

# Tentative de génération 38 kHz
for _ in range(cycles):
    lgpio.gpio_write(h, 18, 1)  # Allumer
    time.sleep(0.0000087)  # 8.7 microsecondes
    lgpio.gpio_write(h, 18, 0)  # Éteindre
    time.sleep(0.0000176)  # 17.6 microsecondes
```

Test de fonctionnement : - LED visible avec caméra smartphone : Oui, elle clignote - Amplificateur répond : Non, aucune réaction - Distance testée : 50 cm, 1 mètre, 2 mètres → Aucune ne fonctionne

Conclusion : La LED clignote, donc le code s'exécute, mais l'amplificateur ne répond pas. Il y a un problème de qualité du signal.

---

5. Les problèmes rencontrés

5.1 Problème de timing avec time.sleep()

Le problème fondamental : En Python sur Linux, `time.sleep()` n'est pas assez précis pour des durées aussi courtes.

Explication technique :

Sur un système d'exploitation comme Linux, plusieurs programmes s'exécutent en même temps. Le "scheduler" (ordonnanceur) du système décide quand chaque programme peut s'exécuter. Quand on appelle `time.sleep()`, on demande au système de mettre notre programme en pause, mais :

1. Le système ne peut pas garantir une pause précise en dessous de 100 microsecondes
2. D'autres programmes peuvent s'exécuter pendant notre pause

3. Le réveil du programme n'est pas instantané

Calcul de l'erreur :

```
Précision time.sleep() = environ 100 microsecondes
Durée nécessaire pour 33% duty cycle = 8.7 microsecondes
Erreur relative = 100 / 8.7 = 1150%
```

Cette imprécision est catastrophique pour notre signal.

Conséquences mesurées : - Avec un smartphone, le clignotement de la LED semblait irrégulier - La fréquence variait entre 35 kHz et 42 kHz au lieu de 38 kHz stable - Le duty cycle variait entre 45% et 55% au lieu de 33%

5.2 Problème du duty cycle incorrect

Avec `time.sleep()`, impossible d'obtenir un duty cycle précis de 33%. Le système oscillait autour de 50%, ce qui explique la faible portée.

Impact constaté : - Portée maximale : seulement 1 à 2 mètres - Fiabilité : environ 20% des commandes passaient à 1 mètre

Comparaison :

Duty cycle	Résultat observé	Distance fonctionnelle
50% (version initiale)	Signal détecté faiblement	1-2 mètres maximum
33% (version optimisée)	Signal détecté clairement	6+ mètres

---

6. Les solutions développées

6.1 Solution au problème de timing : le busy-wait

Au lieu d'utiliser `time.sleep()` qui donne le contrôle au système, j'ai utilisé une technique appelée "busy-wait" (attente active).

Principe : Au lieu de dire "réveille-moi dans X microsecondes", on demande l'heure très fréquemment jusqu'à ce que le temps souhaité soit écoulé.

Code optimisé :

```
import lgpio
import time

def send_ir_burst(duration_us):
    """Génère une rafale IR modulée à 38kHz avec duty cycle 33%"""

    # Paramètres
    period_us = 26.3  # Période d'un cycle 38 kHz
    duty_cycle = 0.33  # 33%

    # Calcul des durées en nanosecondes pour plus de précision
    on_time_ns = int(period_us * duty_cycle * 1000)  # 8700 nanosecondes
    off_time_ns = int(period_us * (1 - duty_cycle) * 1000)  # 17600 nanosecondes

    # Nombre de cycles à générer
    cycles = int(duration_us / period_us)
```

```
    # Heure de départ
    start_time = time.time_ns()  # Temps en nanosecondes

    # Génération du signal
    for cycle in range(cycles):
        # Phase ON : allumer la LED
        lgpio.gpio_write(h, 18, 1)

        # Attente active jusqu'au moment d'éteindre
        target_time = start_time + (cycle * period_us * 1000) + on_time_ns
        while time.time_ns() < target_time:
            pass  # Attente active (busy-wait)

        # Phase OFF : éteindre la LED
        lgpio.gpio_write(h, 18, 0)

        # Attente active jusqu'au prochain cycle
        target_time = start_time + ((cycle + 1) * period_us * 1000)
        while time.time_ns() < target_time:
            pass  # Attente active

    # S'assurer que la LED est éteinte à la fin
    lgpio.gpio_write(h, 18, 0)
```

Avantages de cette approche :

1. **Précision nanoseconde** : `time.time_ns()` a une résolution inférieure à 100 nanosecondes
2. **Pas d'interruption** : Le programme reste actif, le système ne peut pas l'interrompre
3. **Timing absolu** : On calcule par rapport au temps de départ, pas par accumulation d'erreurs

Inconvénient : - Le processeur travaille à 100% pendant l'envoi du signal (environ 70 millisecondes) - Sur un Raspberry Pi avec 4 cœurs, cela reste acceptable car seulement un cœur est utilisé

6.2 Amélioration de la priorité du processus

Pour minimiser les risques d'interruption par le système, j'ai augmenté la priorité de mon programme :

```
import os

try:
    os.nice(-10)  # Priorité haute (nécessite sudo)
except PermissionError:
    pass  # Continue sans priorité élevée
```

Explication : - `os.nice(-10)` demande au système de donner plus d'importance à notre programme - Cela nécessite les droits administrateur (sudo) - Sans sudo, le programme fonctionne quand même mais avec une fiabilité légèrement réduite

Impact mesuré : - Avec sudo : Fiabilité de 98% - Sans sudo : Fiabilité de 92%

6.3 Découverte de la spécificité Yamaha

Un problème persistait avec la commande POWER : elle ne fonctionnait pas de manière fiable.

Tests effectués :

Configuration	Taux de réussite
Un seul envoi	42%
Double envoi avec pause 50ms	78%
Double envoi avec pause 108ms	100%

Solution finale : Envoyer la commande POWER deux fois avec une pause de 108 millisecondes entre les deux.

```
def send_power():
    # Premier envoi
    send_command('POWER')
    time.sleep(0.108)  # 108 millisecondes
    # Deuxième envoi
    send_command('POWER')
```

Cette particularité semble spécifique aux amplificateurs Yamaha. Les autres commandes (volume, play, etc.) fonctionnent avec un seul envoi.

6.4 Optimisation du duty cycle

Avec le busy-wait en nanosecondes, j'ai pu ajuster précisément le duty cycle à 33%.

Résultats avant/après :

Version	Duty cycle mesuré	Portée
time.sleep()	45-55%	1-2 mètres
Busy-wait nanoseconde	33% ±1%	6+ mètres

---

7. Code final et architecture

7.1 Structure complète du code

Le code final est organisé en classe pour faciliter l'intégration :

```
import lgpio
import time
import os

class YamahaRemote:
    def __init__(self, ir_pin=18):
        """Initialise la télécommande Yamaha"""
        self.ir_pin = ir_pin
        self.YAMAHA_ADDRESS = 0x78
        self.carrier_freq = 38000
        self.duty_cycle = 0.33

        # Codes des commandes Yamaha
        self.commands = {
            'POWER': 0x0F,
            'VOL_UP': 0x1E,
            'VOL_DOWN': 0x1F,
            'PLAY': 0x02,
            'PAUSE': 0x0A,
            'STOP': 0x01,
```

```
            # ... autres commandes
        }

        # Initialisation GPIO
        self.h = lgpio.gpiochip_open(0)
        lgpio.gpio_claim_output(self.h, self.ir_pin, 0)

        # Augmenter la priorité si possible
        try:
            os.nice(-10)
        except:
            pass

    def nec_encode(self, address, command):
        """Encode une commande au format NEC"""
        data = []

        # Signal de démarrage
        data.extend([9000, 4500])

        # Adresse (8 bits)
        for i in range(8):
            if (address >> i) & 1:
                data.extend([560, 1690])  # Bit 1
            else:
                data.extend([560, 560])   # Bit 0

        # Adresse inversée
        address_inv = (~address) & 0xFF
        for i in range(8):
            if (address_inv >> i) & 1:
                data.extend([560, 1690])
            else:
                data.extend([560, 560])

        # Commande (8 bits)
        for i in range(8):
            if (command >> i) & 1:
                data.extend([560, 1690])
            else:
                data.extend([560, 560])

        # Commande inversée
        command_inv = (~command) & 0xFF
        for i in range(8):
            if (command_inv >> i) & 1:
                data.extend([560, 1690])
            else:
                data.extend([560, 560])

        # Bit d'arrêt
        data.append(560)

        return data

    def send_ir_burst(self, duration_us):
```

```
        """Génère une rafale IR à 38kHz avec duty cycle 33%"""
        period_us = 26.3
        on_time_ns = int(period_us * self.duty_cycle * 1000)

        cycles = int(duration_us / period_us)
        start_time = time.time_ns()

        for cycle in range(cycles):
            lgpio.gpio_write(self.h, self.ir_pin, 1)
            target = start_time + (cycle * period_us * 1000) + on_time_ns
            while time.time_ns() < target:
                pass

            lgpio.gpio_write(self.h, self.ir_pin, 0)
            target = start_time + ((cycle + 1) * period_us * 1000)
            while time.time_ns() < target:
                pass

        lgpio.gpio_write(self.h, self.ir_pin, 0)

    def send_ir_signal(self, pulses):
        """Envoie un signal IR complet"""
        start_time = time.time_ns()

        for i, duration in enumerate(pulses):
            if i % 2 == 0:  # Impulsion ON
                self.send_ir_burst(duration)
            else:  # Pause OFF
                lgpio.gpio_write(self.h, self.ir_pin, 0)
                target = start_time + sum(pulses[:i+1]) * 1000
                while time.time_ns() < target:
                    pass

        lgpio.gpio_write(self.h, self.ir_pin, 0)

    def send_command(self, command_name, double_send=False):
        """Envoie une commande Yamaha"""
        if command_name not in self.commands:
            return False

        command_code = self.commands[command_name]
        pulses = self.nec_encode(self.YAMAHA_ADDRESS, command_code)

        self.send_ir_signal(pulses)

        if double_send:
            time.sleep(0.108)
            self.send_ir_signal(pulses)

        return True

    def send_power(self):
        """Envoie la commande POWER (double envoi)"""
        return self.send_command('POWER', double_send=True)

    def cleanup(self):
```

```
        """Nettoie les ressources"""
        lgpio.gpio_write(self.h, self.ir_pin, 0)
        lgpio.gpiochip_close(self.h)
```

7.2 Utilisation du code

Exemple simple :

```
# Créer la télécommande
remote = YamahaRemote(ir_pin=18)

# Allumer l'amplificateur
remote.send_power()

time.sleep(3)

# Augmenter le volume 5 fois
for _ in range(5):
    remote.send_command('VOL_UP')
    time.sleep(0.5)

# Nettoyer à la fin
remote.cleanup()
```

Utilisation en ligne de commande :

```
# Allumer
sudo python3 yamaha_remote.py --command POWER

# Volume +
sudo python3 yamaha_remote.py --command VOL_UP

# Play
sudo python3 yamaha_remote.py --command PLAY
```

---

8. Résultats finaux et comparaison

8.1 Tableau comparatif final

Critère	Arduino	Raspberry Pi 5
Facilité de mise en œuvre	Très simple (10 minutes)	Complexe (plusieurs jours)
Code nécessaire	~20 lignes	~200 lignes
Portée du signal	6.5 mètres	6.2 mètres
Fiabilité avec sudo	100%	98%
Fiabilité sans sudo	100%	92%
Temps de réponse	45 microsecondes	125 microsecondes
Consommation électrique	~20 mA	~800 mA
Coût du matériel	~5€	~80€
Connectivité réseau	Non	Oui (WiFi, Ethernet)
Interface web possible	Non	Oui
Intégration domotique	Difficile	Facile

8.2 Tests de fiabilité

J'ai effectué 100 tests de chaque commande pour mesurer la fiabilité :

Arduino : - POWER : 100/100 réussites - VOL_UP : 100/100 réussites - PLAY : 100/100 réussites

Raspberry Pi 5 (avec sudo) : - POWER : 98/100 réussites - VOL_UP : 99/100 réussites - PLAY : 98/100 réussites

Raspberry Pi 5 (sans sudo) : - POWER : 92/100 réussites - VOL_UP : 93/100 réussites - PLAY : 91/100 réussites

8.3 Test de portée

J'ai testé la distance maximale de fonctionnement dans ma pièce :

Plateforme	Distance maximale	Fiabilité à cette distance
Arduino	6.5 mètres	100%
Raspberry Pi (sudo)	6.2 mètres	95%
Raspberry Pi (sans sudo)	5.8 mètres	90%

Note : Au-delà de ces distances, le taux de réussite diminue rapidement. À 1-3 mètres (distance d'utilisation normale), les deux plateformes ont une fiabilité quasi-parfaite.

---

9. Leçons apprises

9.1 Différence timing hardware vs software

Arduino (timing hardware) : - Le timer interne génère le signal de manière autonome - Le processeur n'intervient pas dans la génération de la porteuse - Précision garantie au niveau matériel - Aucune interruption possible

Raspberry Pi (timing software) : - Le processeur doit gérer chaque changement d'état - Le système d'exploitation peut interrompre le programme - Nécessite des optimisations importantes pour être fiable - Consomme 100% d'un cœur CPU pendant l'envoi

9.2 Importance de la précision temporelle

J'ai appris que pour les signaux infrarouges : - Une imprécision de quelques pourcents sur le duty cycle réduit drastiquement la portée - Le timing doit être précis à la microseconde, idéalement à la nanoseconde - Les bibliothèques standard (`time.sleep()`) ne sont pas adaptées - Le busy-wait est la seule solution viable sur Linux

9.3 Spécificités constructeur

Chaque fabricant a ses particularités : - Yamaha nécessite un double envoi pour la commande POWER - Le gap de 108 millisecondes entre les envois est critique - D'autres fabricants peuvent avoir d'autres exigences

9.4 Compromis entre simplicité et fonctionnalités

Arduino : - Simple et fiable immédiatement - Limité pour une intégration avancée - Idéal pour un usage autonome

Raspberry Pi : - Complexe à mettre en œuvre - Performances équivalentes après optimisation - Offre toutes les fonctionnalités nécessaires pour la domotique

---

10. Intégration dans un système domotique

10.1 Architecture finale

J'ai développé un adaptateur Python qui s'intègre dans mon système domotique :

Fonctionnalités : - API REST pour contrôler l'amplificateur via HTTP - Interface web pour le contrôle manuel - Intégration avec d'autres services (par exemple, baisser le volume automatiquement lors d'un appel téléphonique) - Macros personnalisées (par exemple, "Soirée film" allume l'ampli, sélectionne la source TV, et ajuste le volume)

Exemple d'utilisation via API :

```
# Allumer l'amplificateur
curl -X POST http://raspberry-pi:8080/yamaha/power

# Augmenter le volume
curl -X POST http://raspberry-pi:8080/yamaha/volume/up

# Changer de source
curl -X POST http://raspberry-pi:8080/yamaha/source/aux
```

10.2 Fiabilité en production

Le système fonctionne depuis plusieurs semaines sans problème : - Disponibilité : Aucun plantage observé - Réponse quasi-instantanée (< 150 millisecondes du clic à l'exécution)

10.3 Évolutions possibles

Court terme : - Ajouter plus de commandes Yamaha - Créer des scénarios automatisés - Ajouter un contrôle vocal

Long terme : - Support d'autres protocoles IR (pour contrôler d'autres appareils) - Apprentissage automatique de nouveaux codes - Interface mobile dédiée

---

11. Recommandations pratiques

11.1 Quand utiliser Arduino

Je recommande Arduino si : - Vous voulez une solution simple et rapide - Vous n'avez pas besoin de connectivité réseau - Le budget est limité - Vous voulez une fiabilité maximale garantie - Vous créez un appareil autonome

11.2 Quand utiliser Raspberry Pi

Je recommande Raspberry Pi si : - Vous voulez intégrer dans un système domotique - Vous avez besoin d'une interface web - Vous voulez pouvoir faire évoluer le système facilement - Vous avez du temps pour l'optimisation initiale - Vous avez des compétences en Python

11.3 Conseils pour reproduire ce projet

Si vous partez sur Arduino : 1. Utilisez la bibliothèque IRremote (très simple) 2. Connectez la LED IR sur le pin 3 3. Le code fait moins de 20 lignes 4. Ça fonctionnera du premier coup

Si vous partez sur Raspberry Pi 5 : 1. Installez lgpio : `sudo apt install python3-lgpio` 2. Utilisez le busy-wait nanoseconde (pas `time.sleep()`) 3. Lancez avec sudo pour une meilleure fiabilité 4. Testez d'abord à courte distance (50 cm) 5. Vérifiez le clignotement avec un smartphone 6. N'oubliez pas le double envoi pour POWER

Matériel : - LED IR 940nm 5mm suffit amplement - Pas besoin de résistance si vous utilisez les GPIO directement - Deux LEDs augmentent la couverture angulaire - Câble de 1 mètre fonctionne sans problème

11.4 Dépannage

La LED ne clignote pas du tout : - Vérifiez les connexions (GPIO 18 = Pin physique 12) - Vérifiez que lgpio est bien installé - Testez avec un code simple qui allume/éteint la LED

La LED clignote mais l'ampli ne répond pas : - Vérifiez que vous utilisez bien le busy-wait nanoseconde - Lancez avec sudo - Rapprochez-vous à 50 cm pour tester - Vérifiez les codes de commande Yamaha (0x78 pour l'adresse)

L'ampli répond parfois seulement : - Augmentez le duty cycle à 40-50% pour tester - Vérifiez que vous lancez avec sudo - Pointez directement vers le capteur de l'ampli - Pour POWER, utilisez bien le double envoi

---

12. Conclusion

Ce projet m'a permis de créer une télécommande infrarouge fonctionnelle pour mon amplificateur Yamaha sur Raspberry Pi 5. Le chemin n'a pas été simple, avec plusieurs obstacles techniques importants, mais le résultat final est à la hauteur de mes attentes.

12.1 Objectifs atteints

- Télécommande Arduino fonctionnelle en quelques heures ou jours ...
- Migration réussie vers Raspberry Pi 5 après optimisations
- Performances finales comparables entre les deux plateformes
- Intégration complète dans mon système domotique
- Documentation complète du processus pour faciliter la reproduction

12.2 Contributions techniques

Les points techniques importants de ce projet :

1. **Validation du protocole NEC sur Raspberry Pi 5** : Première documentation complète avec la bibliothèque lgpio
2. **Technique de busy-wait nanoseconde** : Solution générique pour le timing précis sous Linux
3. **Identification des spécificités Yamaha** : Documentation du double envoi POWER et du gap de 108ms
4. **Code production-ready** : Adaptateur intégrable dans des systèmes domotiques réels

12.3 Résultats chiffrés

Performances finales : - Raspberry Pi : 98% de fiabilité avec sudo, portée de 6.2 mètres - Équivalent à Arduino à moins de 5% près - Latence totale : 150 millisecondes de la commande à l'exécution

Développement : - Arduino : 2 heures (installation + tests) - Raspberry Pi : 5 jours (recherches, optimisations, tests)

12.4 Ce que j'ai appris

Sur le protocole infrarouge : - Le timing est absolument critique - La moindre imprécision se traduit par une perte de portée - Chaque fabricant a ses particularités

Sur le Raspberry Pi : - Linux n'est pas conçu pour le temps réel - Avec les bonnes techniques, on peut obtenir un timing suffisamment précis - Le busy-wait est acceptable pour des signaux courts (70ms)

Sur le développement embarqué : - Toujours valider avec une plateforme simple d'abord (Arduino) - Les bibliothèques standards ne sont pas toujours adaptées - L'optimisation nanoseconde fait toute la différence

12.5 Perspectives

Ce projet ouvre la voie à plusieurs évolutions :

Extension du système : - Support d'autres appareils IR (TV, décodeur, etc.) - Apprentissage automatique de nouveaux codes - Création d'un hub IR universel

Optimisations possibles : - Utilisation du PWM hardware du Raspberry Pi (GPIO 12/13) - Tests sur d'autres plateformes ARM - Comparaison avec ESP32 (qui a un RTOS)

Documentation et partage :

Ce projet démontre qu'il est possible de créer des solutions domotiques performantes sur Raspberry Pi, même pour des protocoles exigeants comme l'infrarouge, à condition d'être prêt à optimiser en profondeur. Le choix entre Arduino et Raspberry Pi dépend vraiment de vos besoins : simplicité vs fonctionnalités avancées.

Nicolas Loisy - Février 2025