

Transfert de données via laser

Encadrant


Floréal Morandat
Antoine Rollet

Équipe projet

Louis Deguillaume
Nathan Nguyen
Adrien Rodrigues



Introduction



Dans le cadre du module PR311 de développement système et réseaux, nous avons travaillé sur un projet nommé "Challenge laser". Ce projet consiste à faire transiter de l'information au moyen d'un laser et d'un capteur reliés à des Raspberry Pis.

Ce projet ne possède pas d'objectifs concrets si ce n'est la recherche d'améliorations de l'existant, c'est-à-dire augmenter le débit, ajouter de la correction d'erreur, la distance émetteur/-récepteur, etc.... Le transfert d'information ou de fichier est une opération très courante en informatique surtout d'où l'intérêt du projet.

Nous présentons ici la mise en place de notre dispositif physique, la démarche de notre travail sur ce projet ainsi que les résultats obtenus.

Table des matières

I. Résultats précédents	3
I. 1 Protocole de communication	3
I. 2 Résultats et principaux défauts	3
II. Matériel utilisé et montage	4
II. 1 Matériel	4
II. 1.1 Raspberry Pi	4
II. 1.2 Diode laser	4
II. 1.3 Phototransistors	4
II. 1.4 Transistor	5
II. 1.5 Résistance	5
II. 2 Montage	5
II. 3 Procédure pour désactiver le Bluetooth	5
III. Principe de fonctionnement	7
III. 1 Principe de base	7
III. 2 Envoi et réception de données	7
IV. Limitations	8
IV. 1 Alignement du laser	8
IV. 2 Buffer UART plein	8
V. Résultats	10
VI. Améliorations	11
VI. 1 Envoi de "vraies" trames	11
VI. 2 Détection d'erreur et réémission	11
VI. 3 Correction d'erreur	12
VI. 4 Parallélisation	12
VII. Conclusion	13

I. Résultats précédents

Une équipe a déjà travaillé sur le transfert de données via laser [6]. À des fins de référence, les méthodes utilisées et les résultats obtenus par cette précédente équipe seront énoncés dans cette section.

I. 1 Protocole de communication

Pour faire transiter les données, un codage Manchester était utilisé couplé à un protocole de niveau 2 maison. Leur protocole utilisait un découpage en trame (figure 1). Un temps de pause permettait de séparer deux trames.

11111111	n° séq	taille	charge utile
4 bits	6 bits	63 octets max	

FIGURE 1: Trame utilisée . Source [6]

Avant l'émission en elle-même, huit bits à 1 sont envoyés pour synchroniser les fréquences de l'émetteur et du récepteur. Un temps de pause est marqué et l'envoi de trames peut commencer.

I. 2 Résultats et principaux défauts

La solution a été réalisée en python, permettant une écriture et des modifications rapides. Des exemples de débits atteints avec un taux de réussite de 99% étaient alors de 458bps utiles (pour 475bps de débit total) ou encore 787bps utiles (pour 4000bps de débit total). Ces débits faibles étaient principalement dûs à plusieurs problèmes :

Préemption L'intégralité du programme était en python. De ce fait, l'envoi, comme la lecture de bits se faisait en python. Comme tout programme classique, ce dernier peut être préempté pour permettre à d'autres programmes de s'exécuter sur le CPU. Le problème étant que cette préemption peut arriver pendant l'émission ou la réception, causant ainsi une incapacité temporaire à lire ou envoyer des signaux lumineux. De ce fait, l'équipe a été obligée de traiter cette préemption comme elle le pouvait en chronométrant le temps écoulé pour ne pas prendre en compte les bouts de trames reçus avant et après reprise du programme.

Pluri-émission Pour que les données parviennent tout de même au récepteur, il est nécessaire d'émettre plusieurs fois les données pour espérer qu'elles soient reçues au moins une fois.

Fréquence de processeur variable De plus, en fonction de la configuration des Raspberry Pi, le CPU était probablement en mode dynamique : la fréquence du CPU n'est pas fixe. De ce fait, l'émetteur et le récepteur pouvait être désynchronisés. Grâce au débit assez faible, l'équipe n'a probablement pas remarqué ce phénomène.

II. Matériel utilisé et montage

II. 1 Matériel

Le matériel utilisé pour ce projet est détaillé ici.

II. 1.1 Raspberry Pi

Deux Raspberry Pi 3 ; une pour l'émission, l'autre pour la réception. Il a été nécessaire de désactiver le Bluetooth sur ces cartes pour des raisons expliquées plus tard. Cette opération est uniquement logicielle et est facilement réversible.

L'UART de la raspberry PI [3] supporte officiellement les baud rate présentés dans la table 1. Par soucis d'expérience utilisateur, nous avons attribué une catégorie à chaque baudrate, utilisable dans nos programmes.

baud rate_cat	baud rate
1	110
2	1200
3	2400
4	9600
5	14400
6	19200
7	38400
8	57600
9	76800
10	115200
11	230400
12	460800

TABLE 1: baud rate_cat \leftrightarrow baud rate

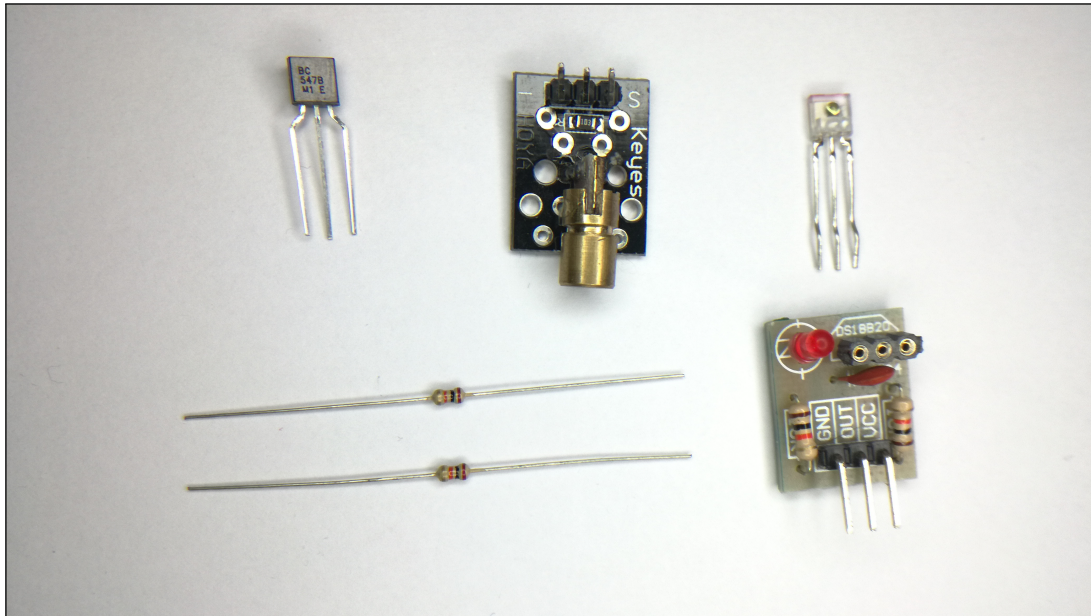


FIGURE 2: Matériel utilisé (transistor, diode laser, phototransistor et sa carte, résistances)

II. 1.2 Diode laser

Une diode laser rouge provenant de kits Arduino. Il s'agit d'une diode KY-008 [2] de longueur d'onde 650nm.

II. 1.3 Phototransistors

Un phototransistor [4] associé avec un composant [1] permettant de facilement le brancher sur du 3.3V.

II. 1.4 Transistor

Un transistor NPN BC547B [7] afin de convertir la tension de contrôle de la Raspberry Pi (3.3V) en une tension plus élevée (5V) pour contrôler la diode laser, nécessitant du 5V.

II. 1.5 Résistance

Deux résistances de 10k Ω , ou une de 20k Ω .

II. 2 Montage

Une des deux Raspberry Pi est connectée à la diode laser, l'autre est connecté au mécanisme de réception (phototransistor monté sur sa carte).

Les figures 3 et 4 montrent respectivement un schéma du montage de la Raspberry Pi d'émission et de réception.

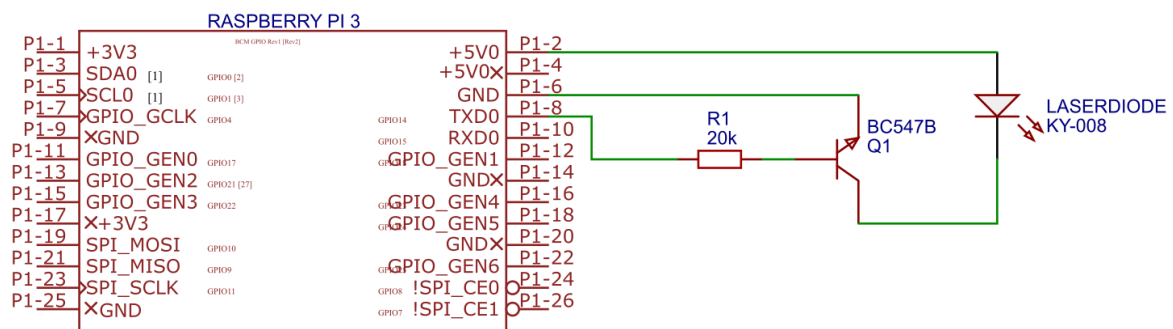


FIGURE 3: Schéma électronique de la carte émettrice

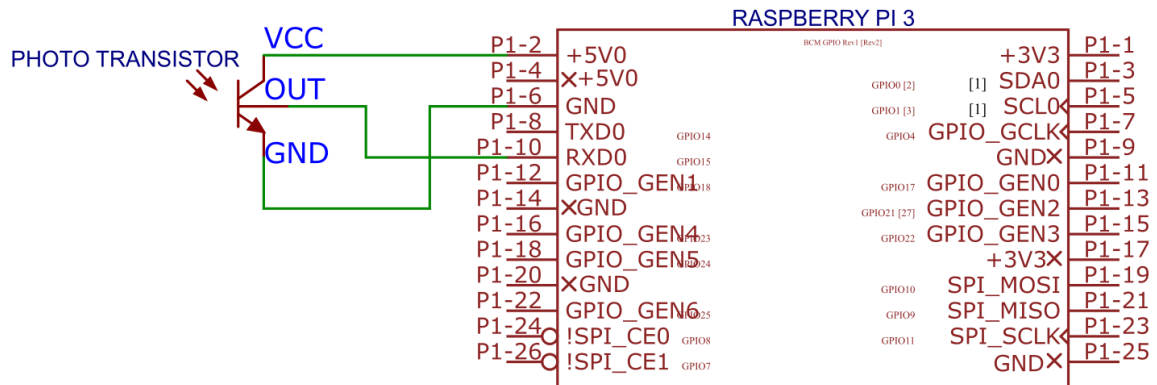


FIGURE 4: Schéma électronique de la carte réceptrice

II. 3 Procédure pour désactiver le Bluetooth

Les Raspberry Pi 3 possèdent deux ports séries, l'un est utilisé par le Bluetooth et l'autre est connecté aux ports TX et RX de la Raspberry. Cependant cette dernière ne possède qu'une seule horloge dédiée pour les communications séries. Par défaut elle est utilisée pour la connectivité Bluetooth.

La seconde connexion série utilise en fait l'horloge du processeur afin de fonctionner. Utiliser cette *mini-uart* peut être problématique puisqu'elle (entre-autre [5]) :

- dépend de l'horloge du processeur ne fonctionne pas à une fréquence constante dans le but d'économiser de l'énergie.
- ne permet pas de détecter si la lecture de la séquence de données à commencé au bon endroit.
- ne permet pas l'utilisation du bit de parité.
- ne permet pas de recevoir des interruption de type timeout.
- ne permet pas la gestion des signaux DCD, DSR, DTR et RI.

Il nous est donc nécessaire de désactiver le Bluetooth de la Raspberry Pi si l'on veut être en mesure d'utiliser l'horloge dédiée aux connections séries sur les ports Rx et Tx.

La manipulation à faire est la suivante :

1. Ajouter la ligne `dtoverlay = pi3-disable-bt` au fichier `/boot/config.txt`.
2. Supprimer `console=serial0,115200` dans le fichier `/boot/cmdline.txt`.
3. Redémarrer la Raspberry Pi .

Il est possible de le vérifier avec la commande `ls -l /dev/serial*`. `serial0` doit pointer vers `ttyAMA0` (voir figure 5).

```
pi@laser1:~/laser/src/python/data $ ls -l /dev/serial*
lrwxrwxrwx 1 root root 7 janv. 14 13:17 /dev/serial0 -> ttyAMA0
lrwxrwxrwx 1 root root 5 janv. 14 13:17 /dev/serial1 -> ttyS0
```

FIGURE 5: Résultat montrant que la désactivation du Bluetooth a bien fonctionné

III. Principe de fonctionnement

III. 1 Principe de base

Afin de régler le problème de préemption et de fréquence variable du processeur, nous avons décidé d'utiliser les ports séries (UART) de la Raspberry Pi afin d'envoyer des données au laser (Tx) et de récupérer les données du récepteur (Rx). Utiliser le protocole UART présente nombreux avantages :

- Il est très facile de connecter l'émetteur et le récepteur à ces ports et de les faire communiquer directement (en utilisant la bibliothèque `Serial` en Python par exemple).
- La gestion de ces ports se faisant au niveau matériel, nous n'avons pas de problème de préemption et les données sont bien envoyées de manière continue sur ces ports. Nous avons tout de même rencontré un problème de bufferisation que nous détaillerons section IV. 2.
- Il n'y a pas de problème par rapport à la fréquence variable de l'horloge du processeur de la Raspberry Pi . Comme évoqué section II. 3, il est nécessaire d'avoir désactivé le Bluetooth afin d'utiliser cette horloge dédiée.

III. 2 Envoi et réception de données

Dans un premier temps nous envoyions les données encodées en base64 d'un seul bloc. Seulement cette solution n'était pas viable (cf sec. IV. 2). Nous avons donc découpé la transmission par blocs. L'encodage en base64 a été choisi car Python récupère le payload avec un `readLine` et dans le cas où `\n` apparaîtrait dans le binaire du payload, la réception serait arrêtée. En effet Python n'est pas adapté pour lire caractère par caractère à une telle vitesse ; la primitive `readLine` a été préférée. L'encodage en base64 pallie à ce problème (`\n` ne peut pas être présent dans une chaîne en base64) mais nécessite un overhead de 30%.

Supposons que l'on découpe les données à envoyer de la manière suivante :

$$data = b_1.b_2.b_3. \dots .b_k$$

Supposons également que les blocs ont une taille fixe (à l'exception du dernier) :

$$|b_1| = |b_2| = |b_3| = \dots = |b_{k-1}| = n, \text{ et que } |data| = \sum_{i=1}^k |b_i|$$

Afin de pouvoir décoder ces données à l'arrivée dans leur globalité et non pas bloc par bloc, il faut que :

$$base64(data) = base64(b_1).base64(b_2).base64(b_3). \dots .base64(b_k)$$

Cette condition impose donc que $3|n$ car l'encodage en base64 se fait sur des groupes de trois octets [9]. En accord avec les contraintes évoqués dans la section IV. 2, nous avons choisi une taille de blocs de 3333 octets.

IV. Limitations

Au fil du projet, nous avons rencontré plusieurs limitations. Cette section expose les solutions de contournement mises en place.

IV. 1 Alignement du laser

Lors de nos expérimentations, nous nous sommes rendu compte que l'alignement du laser était problématique et qu'un bon alignement pouvait multiplier le baud rate (vitesse de transmission) utilisable par 24 (passant de 9600 baud à 230400 baud pour des petits fichiers). Nous supposons qu'un mauvais alignement empêche le capteur de se charger et se décharger assez vite par rapport à la vitesse d'émission et la puissance lumineuse reçue.

Nous avons donc, dans le but de faciliter cet alignement, créé un script envoyant une séquence de caractères prédéfinie à intervalle régulier. La carte réceptrice lit simplement ce qu'elle reçoit sur son port UART. Comme la séquence de caractères est connue, la Raspberry Pi est capable de déterminer quelle quantité d'information a bien été reçue. Il est ainsi possible d'aligner plus facilement le laser avec le récepteur. De plus, nous nous sommes rendu compte lors de cette opération que pour que la communication fonctionne le mieux possible le laser devait être dirigé en dessous à droite du récepteur (et non pas au centre comme on pourrait le penser) comme sur la figure 6.

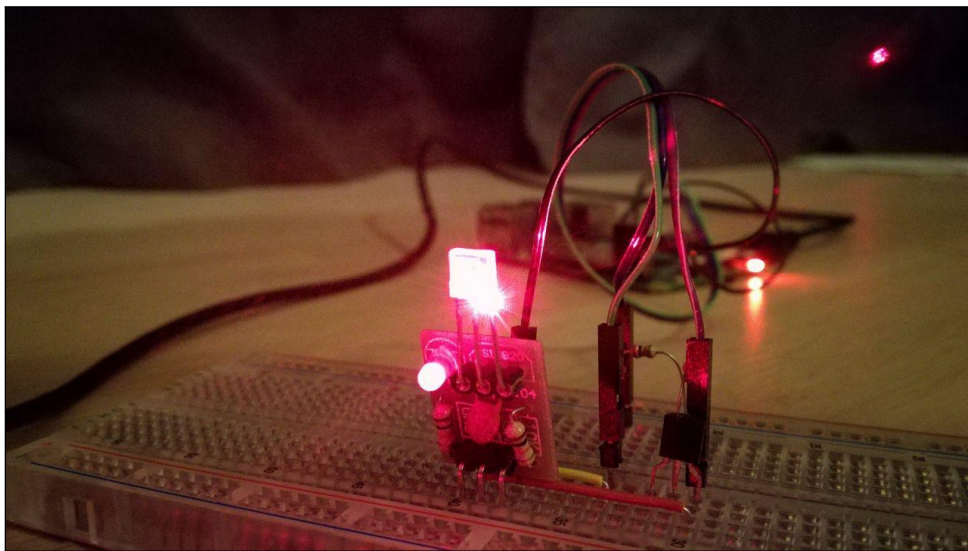


FIGURE 6: Alignement correct du laser sur le récepteur

Même avec le script d'alignement il est parfois compliqué d'obtenir un alignement satisfaisant. Ceci étant dit, il semblerait qu'augmenter la distance n'impacte pas le temps de transfert et simplifie l'alignement.

IV. 2 Buffer UART plein

Seulement, même avec un bon alignement, le baudrate maximal utilisable était de 230400 pour des petits fichiers. Pour des plus volumineux, le maximum était de 115200 baud.

Pour déterminer ce qui limitait le baud rate utilisable, nous avons branché directement le port **Tx** de la Raspberry Pi émettrice sur le **Rx** de la réceptrice, en utilisant le même code pour l'échange. Pour des fichiers volumineux et avec un baud rate inférieur ou égal à 115200, les données étaient bien reçues. Au delà, elles n'étaient plus valides : le problème ne venait donc pas du matériel.

Après investigation, la piste la plus plausible est que l'émetteur envoie les données trop rapidement pour que le récepteur puisse les traiter. De ce fait, un décalage se crée, et le buffer de l'UART se remplit de plus en plus, jusqu'à ce qu'il ne puisse plus accepter de données : le récepteur ne traite donc pas les données assez vite.

Nous avons donc ajouté des temps d'attentes (**sleep**) entre chaque émission de bloc pour résoudre ce problème. Avec de durées d'attente relativement faible (0.6 s), nous avons atteint les 230400 baud. Cette durée devant être d'au moins 2 s pour un débit de 460800 baud. L'inconvénient évident est que ces attentes réduisent notre vitesse d'émission, parfois au point de préférer un baud rate inférieur pour éviter les **sleep**.

En supposant que Python soit le problème, nous avons donc écrit un programme hybride C & Python. Celui-ci :

- Lit les données reçues et les enregistre dans un fichier temporaire (partie écrite en C).
- Fait la vérification de la somme de contrôle et le décodage du fichier (partie en Python).

De **Rx** à **Tx**, ce programme peut atteindre les 921600 baud sans problème, et sans **sleep**. En effet, la partie qui précédemment s'effectuait trop lentement à cause de l'interprétation Python s'effectue maintenant suffisamment vite en C.

Attention toutefois, pour utiliser le laser il faut faire attention à l'alignement avec le récepteur. Cet alignement est relativement aisé à obtenir pour un baud rate de 460800 mais relève de la chance pour 921600 baud.

V. Résultats

Notre objectif était principalement l'augmentation du débit utile. L'utilisation du protocole UART qui se base sur l'horloge matérielle ainsi qu'un bon alignement du laser nous permet d'augmenter significativement le débit. On atteint en effet des débits aux alentours de 46 ko/s. Ce qui équivaut à un débit utile multiplié par 460 par rapport aux résultats précédents.

Des exemples de résultats obtenus sont montrés table 2.

Fichier	taille	type d'envoi	baud rate	temps émission	débit utile
40x40.bmp	6.564 o	python	230400	0.87 s	7504 o/s
		hybrid	460800	0.14 s	47633 o/s
bigtext.txt	517.868 o	python	230400	29.56 s	17517 o/s
		hybrid	460800	11.25 s	46019 o/s
400x400.bmp	640.164 o	python	115200	55.57 s	11520 o/s
		python	230400	36.64 s	17470 o/s
		hybrid	460800	13.91 s	46017 o/s
Mountain.jpg	1.582.952 o	python	230400	89.97 s	17595 o/s
		hybrid	460800	34.42 s	45992 o/s

TABLE 2: Résultats obtenus avec différents types et vitesse d'envoi

Par ailleurs, en supposant que le laser soit bien aligné avec le phototransistor, il n'y a pas d'altération des données. Dans l'éventualité où l'alignement change pendant l'envoi, le récepteur va détecter que le(s) paquet(s) (si l'on a choisi d'envoyer un checksum à chaque paquet) est(sont) corrompus. Dans tous les cas, l'intégrité du fichier dans son ensemble sera vérifiée.

VI. Améliorations

Des pistes d'amélioration sont envisageables mais une réflexion approfondie est nécessaire en amont avant toute implémentation.

VI. 1 Envoi de "vraies" trames

Pour éviter l'envoi des trames en base64, il serait envisageable d'envoyer de vraies trames numérotées. Cela éviterait l'overhead de l'encodage en base64. De plus, il serait alors possible de faire de la détection d'erreur et de la réémission plus simplement.

VI. 2 Détection d'erreur et réémission

Un problème récurrent survenu au cours du projet est l'apparition d'erreur ; c'est-à-dire une mauvaise réception de paquet(s) envoyé(s) par l'émetteur qui. Le(s)-dit(s) paquet(s) peut(vent) alors être perdu(s), ou erroné(s). À supposer que de vraies trames soient utilisées, il serait judicieux d'utiliser un CRC, plus léger que le MD5 actuellement utilisé.

Une ébauche d'idée serait de regrouper les trames en segment et de rajouter un code de correction d'erreur à la fin de ces segments numérotés : nous faisons ainsi un compromis pour avoir un maximum de débit utile tout en ayant une correction d'erreur acceptable.

Pour utiliser une quelconque réémission, il faut avoir un mécanisme d'acquittement, et donc une communication bidirectionnelle. L'émetteur pourrait alors rémettre les segments que le récepteur a mal reçu.

Lors de tests pour déterminer des sources d'erreurs, il a été réalisé une découpe des données en blocs avec émission d'un hash md5 pour détecter de quel portion venait l'erreur. Seulement nous n'avons pas de notion d'acquittement et ne disposons que d'une connexion unidirectionnelle : la réémission de bloc n'était donc pas possible. Un montage comme présenté sur la figure 7 peut être réalisé pour avoir une communication bidirectionnelle en full duplex.

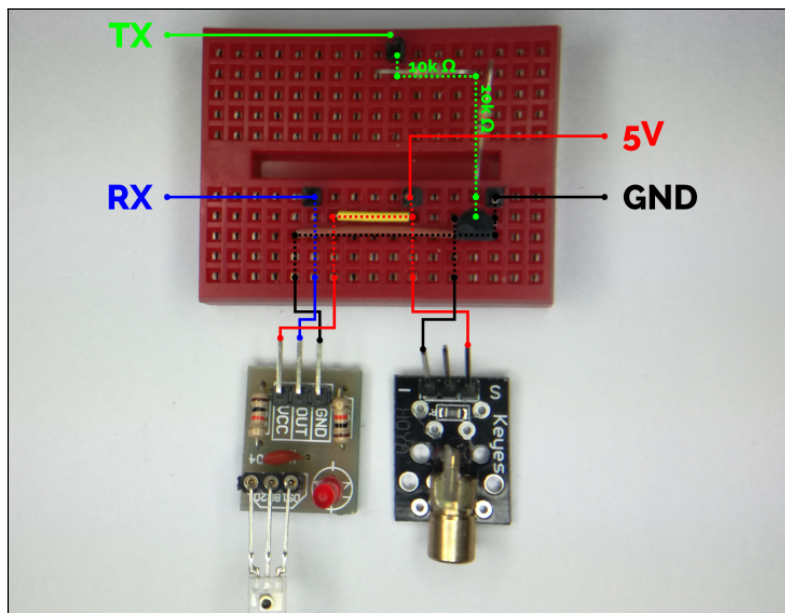


FIGURE 7: Communication bidirectionnelle

VI. 3 Correction d'erreur

Il est également possible grâce à des algorithmes de correction d'erreurs comme celui de Reed–Solomon [8] de se passer de la réémission des blocs corrompus. Cet algorithme nécessite cependant d'envoyer plus de données avec le fichier afin que d'être en mesure de corriger les blocs corrompus.

VI. 4 Parallélisation

Une autre piste d'amélioration possible est la parallélisation. En mettant des Raspberrys Pi en parallèle avec leurs homologues récepteurs, on augmente le débit directement en ayant plusieurs émetteurs en parallèle. Cela nécessite un protocole pour désassembler le payload à envoyer, par exemple :

1. Numéroté les paquets résultant du désassemblage du payload pour les réassembler à l'arrivée
2. Arbitrairement numéroté les Raspberry Pi en imposant que le payload reçu par une Raspberry Pi k est en réalité la partie k du payload complet.

La parallélisation a été envisagée mais nous avons préféré nous focaliser sur l'analyse des limites d'un système mono-laser. Cela nous a permis d'augmenter significativement le débit avec toujours le même dispositif matériel, contrairement à la parallélisation.

VII. Conclusion



En utilisant un protocole déjà existant et implémenté de manière matérielle dans les Raspberry Pis, on a pu s'affranchir des problèmes que l'équipe précédente avait eu avec leur solution. Cette nouvelle implémentation s'est toutefois révélée ne pas être sans encombre puisqu'il nous a fallu pallier à d'autres problèmes. Ainsi, nous avons dû utiliser la base64 pour encoder les informations envoyées pour pouvoir les lire correctement à l'arrivée. Nous avons également eu des problèmes avec la taille des buffers attribués aux `tty` par Linux.

Malgré ces problèmes d'une nature différente il nous a été possible de multiplier les débits obtenus par l'équipe précédente par 460 environ en atteignant des débits de 46 ko/s montrant ainsi que cette implémentation pourrait s'avérer prometteuse.

Enfin la trop grande difficulté pour aligner les lasers au dessus d'un certain baud rate semble indiquer que pour continuer d'améliorer les résultats obtenus en suivant cette piste, il faudra mettre en place d'autres moyens. Ainsi arrêter d'utiliser l'encodage en base64 permettrait de réduire la taille des données à transférer, on peut aussi penser à de la compression afin de limiter encore plus la quantité de données à envoyer pour un fichier données.

Références



- [1] H23L1 support card. <https://www.elabpeers.com/ky-008-laser-x-laser-detector.html>, 2016.
- [2] arduinomodules. Ky-008 laser transmitter module. <http://arduinomodules.info/ky-008-laser-transmitter-module>.
- [3] ARM. Primecell® uart (pl011) r1p4. <http://infocenter.arm.com/help/topic/com.arm.doc.ddi0183f/DDI0183.pdf>, 2005.
- [4] GE Solid State. H23L1 datasheet. <http://www.alldatasheet.com/datasheet-pdf/pdf/115548/GESS/H23L1.html>.
- [5] @JamesH65, @rarvolt, @HairyFotr. The raspberry pi uarts. <https://www.raspberrypi.org/documentation/configuration/uart.md>, 2017.
- [6] Romain Pétro Mathieu Petit, Hervé Périn. Transfert de fichier entre raspberry pi via laser - rapport de projet pr311, 2017.
- [7] multcomp. Bc547b general purpose transistor datasheet. <https://www.farnell.com/datasheets/410427.pdf>, 2008.
- [8] Reed Solomon. Reed solomon error correction. https://en.wikipedia.org/wiki/Reed-Solomon_error_correction, 1960.
- [9] Wikipedia. Base64. <https://fr.wikipedia.org/wiki/Base64>.