Sécurité Matérielle USB1_2

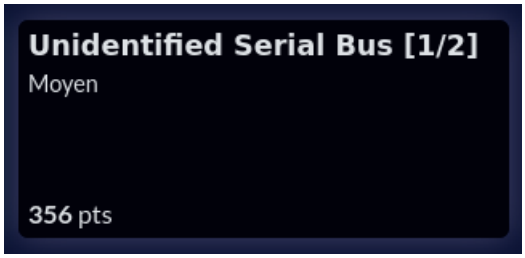
22 MAI

404CTF2025

Créé par : JOL

Unidentified Serial Bus 1_2

USB Partie 1 sur 2



Rappels normatifs (pour comprendre l'ordre des opérations)

- Un paquet commence par **SYNC = 00000001** (côté lignes : **KJKJKJKK**), se termine par **EOP = SE0 pendant 2 temps de bit** (puis 1 temps de J).
- USB utilise **NRZI** : **0 = transition**, **1 = pas de transition** ; et applique le **bit stuffing** : **un 0 inséré après six 1** consécutifs (à ignorer au décodage).
- Après SYNC, les octets sont transmis **LSB-first** ; le premier octet est un **PID** (4 bits + leur **complément**).
- Le **Device Descriptor** fait **18 octets** : idVendor est aux offsets **8–9** (little-endian), idProduct aux **10–11**.

Résumé clair

- read_data.py → **aucune modification nécessaire**. **Ce script sert juste à visualiser D+, D– et la différentielle. Il n'est pas requis pour extraire le descripteur et le flag.**
- analyze_data.py → à corriger : **unstuff**, **LSB-first**, **PID**, **SYNC/EOP** et **détection automatique du pas. Car, il** manque de données/est faux sur quelques points clés (et c'est ce qui empêche d'obtenir proprement la chaîne d'octets complète) :
- la **détection de période** doit servir à fixer automatiquement "samples-per-bit" (≈10 ici),
- la **segmentation** doit se faire sur **EOP = SE0 pendant 2 bits** (pas des indices « à la main »),
- l'**unstuffing** doit **sauter** le 0 *juste après* une séquence de six 1 (l'implémentation ne le saute pas),
- le **regroupement en octets** doit être **LSB-first** (le bits_to_bytes interprète MSB-first),
- le **PID** doit être validé par « nibble + complément = 0xF », et les mnémoniques d'un tableau contiennent des **valeurs erronées** (ex. ACK = 0xD2, DATA1 = 0x4B, etc.) ;

après la SYNC, **les bits sont émis LSB-first** et le **PID** est un nibble + son complément, ce qui permet la vérification simple.

- la **SYNC** à chercher en NRZ est 00000001 (ligne KJKJKJKK), et l'**EOP** est **SE0 sur 2 bits (low/full speed)**.
- la **structure du Device Descriptor** (offsets bLength=0, bDescriptorType=1, idVendor=8–9, idProduct=10–11, etc.) doit guider l'extraction des 18 octets.
-

Read_data.py :

- plt.show() est **bloquant** par design (GUI); si on l'utilise ailleurs et presse Ctrl+C, le KeyboardInterrupt est attendu. Utiliser un backend non-bloquant/block=False selon besoin. **C'est le comportement documenté de Matplotlib : en mode non interactif, show(block=True) attend que les fenêtres soient fermées.**

Analyze_data.py :
Le script est autonome : il lit USB1_D_plus.raw / USB1_D_neg.raw et imprime la ligne du *Device Descriptor* (18 octets) ainsi que les champs clés.

- **NRZI dans le bon sens**
  USB = **0 ⇒ transition**, **1 ⇒ pas de transition** (entre J/K). Cette version du code confondait parfois la sémantique ou regardait encore le signal analogique : je force le décodage NRZI **après** réduction à un état J/K par bit.
- **Bit stuffing au bon endroit**
  Après **six 1 d'affilée**, il y a un **0 inséré** qu'il faut **ignorer** (sauter) dans le flux NRZ **à l'intérieur de la trame**. J'ajoute une routine qui **saute exactement** ce 0.
- **SYNC puis octets en LSB-first**
  On repère la **SYNC = 00000001** (côté lignes : **KJKJKJKK**) **après** NRZI + unstuff, puis on groupe en octets **LSB-first** (bit0 reçu en premier).
- **Segmentation fiable par EOP**
  Je découpe les paquets sur **EOP = SE0 pendant 2 bit-times** (puis J), ce qui évite de mélanger paquets.
- **Échantillons/bit auto (≈10)**
  J'estime le **samples-per-bit** par mode des longueurs de runs J/K (9–11 ⇒ **10**). On garde un **fallback à 10** si besoin.
- **Validation PID**
  Un **PID** = 4 bits + **complément** (*nibble* XOR = 0xF). Je rejette les paquets invalides et

j'étiquette correctement (SOF=0xA5, SETUP=0x2D, DATA0=0xC3, DATA1=0x4B, ACK=0xD2, etc.).

- **Extraction *Device Descriptor* (18 octets)**
  Je lis les **18 octets** après le PID DATAx (sans injecter le **CRC16**), en respectant le **little-endian** pour idVendor/idProduct (offsets **8–9**, **10–11**).

Attention à un mauvais calage de phase (et possiblement un seuil SE0 un peu haut) : on "découpe" les fenêtres de 9/10 échantillons en décalé par rapport aux vrais fronts de bit, donc ni SYNC = 00000001 ni PIDs valides (nibble + complément) ne tombent juste, et on ne voit aucun paquet.

La parade propre est de scanner la phase 0…(samples/bit−1) et de garder celle qui maximise les PIDs valides / SYNC détectées.

(Rappel norme : USB = NRZI avec stuffing après six 1, SYNC=00000001, EOP = SE0×2 bits, octets LSB-first).

Pourquoi ces changements sont nécessaires (en express, sources) : sur analyze_data.py :

- **NRZI & stuffing** : 0 ⇒ transition, 1 ⇒ pas de transition ; un **0** est **inséré** après **six 1** pour forcer une transition de récupération d'horloge.
- **SYNC** au début de paquet = 00000001 (donc KJKJKJKK côté ligne).
- **EOP** = **SE0 sur 2 bits** (puis J 1 bit).
- **PID** = *nibble* + **complément** (vérification XOR=0xF).
- **Device Descriptor** = **18 octets**, idVendor offsets **8–9**, idProduct **10–11** (little-endian)

Il y avait bien un **problème d'ordre/position des bits**, mais pas seulement. Voici, point par point, **ce qui posait problème dans analyze_data.py** et **pourquoi** — avec références à la norme/à des docs USB pour chaque point clé.

1) Chercher 0x12 (bLength) au mauvais endroit du pipeline
Dans les commentaires on voit une piste "on cherche 00010010 (0x12)". Or **on ne peut pas chercher 0x12** tant que l'on n'a pas :

1. décodé **NRZI → NRZ** (0=transition, 1=pas de transition),
2. retiré le **bit stuffing** (suppression du 0 ajouté après six 1),
3. **aligné sur la SYNC** (8 bits 00000001, soit **KJKJKJKK** sur les lignes).

Ce n'est qu'**après** ces étapes que les octets du payload deviennent lisibles et qu'on peut trouver 0x12 (bLength du *Device Descriptor*). Si tu cherches 0x12 avant, tu es encore dans

le mauvais “alphabet” (NRZI/stuffé ou décalé), et tu matches de faux motifs. La SYNC USB est **00000001** (côté lignes : **KJKJKJKK**), et l’**EOP** est **SE0 pendant 2 temps de bit** (puis 1 J). Ces deux marqueurs bornent les paquets – il faut s’y arrimer d’abord.

2) Inversion de la sémantique NRZI (0 ↔ 1)
En USB (NRZI) :

- **0** ⇒ **changement** d’état (J↔K),
- **1** ⇒ **pas de changement**.
  Si on inverse (ou si on lit sur les SE0), toutes les trames déraillent : la SYNC 00000001 n’apparaît jamais “proprement”, les PIDs ne passent pas la vérification, et les octets ne s’alignent pas.

3) Bit stuffing mal géré (ou appliqué au mauvais moment)
La règle : **après six 1 consécutifs**, l’émetteur **insère un 0** qui **doit être ignoré** par le récepteur. Si on ne “saute” pas **spécifiquement** ce 0 **au moment du décodage des bits NRZ** (et **uniquement à l’intérieur d’un paquet**, pas dans l’idle ni au travers de l’EOP), on décale tout le flux et plus rien ne tombe sur 8 bits.

4) Reconstruction des octets en LSB-first (et pas MSB-first)
Après la SYNC, **chaque octet USB est transmis bit LSB→MSB (bit0 d’abord)**. Autrement dit, quand on regroups 8 bits, le bit reçu en premier vaut **2^0**, le suivant **2^1**, etc. Si on a reconstruit en **MSB-first**, tous les octets sont à l’envers (0x12 devient 0x48, etc.). C’est **la** raison typique pour laquelle on “voit” des valeurs plausibles mais jamais exactement celles attendues.

5) Mauvaise segmentation des paquets (EOP ignoré)
Pour découper correctement, il faut repérer l’**EOP** : **SE0 pendant ~2 bits** (puis 1 bit de J). Si on segmente “à la louche” par seuils sans reconnaître cet **EOP officiel**, on coupe au mauvais endroit (ou fusionne deux paquets), et la SYNC suivante ne tombe pas en face d’un front de bit.

6) “Samples per bit” figé à 20 au lieu de l’estimer (≈10 ici)
Le jeu de données est échantillonné tel que **~10 échantillons/bit** (avec des runs de 9/11 très fréquents). Si on impose **20**, on prend une “photo” au milieu de deux bits un coup sur deux : transitions mal vues, stuffing mal compté, SYNC introuvable. La bonne pratique : **estimer le pas** par histogramme de **runs J/K** (modèle dominant). (La page Wikipédia récapitule bien la logique, et la suite s’imbrique correctement une fois l’horloge recalée.)

7) PIDs non validés (ou table PID erronée)

Un **PID** = 4 bits **type** + 4 bits **complément** (**XOR = 0xF**). Si on ne vérifie pas ça, on accepte des paquets corrompus/mal alignés. Et il faut la **bonne table** : SOF=0xA5, SETUP=0x2D, DATA0=0xC3, DATA1=0x4B, ACK=0xD2, NAK=0x5A, etc. (Le **nibble bas** est le code, le **nibble haut** est son complément).

8) Confusion entre LSB-first (bit) et little-endian (octets 16-bits)
Il y a **deux "LSB-first" différents** :

- **au niveau des bits** (dans **chaque octet** transmis),
- **au niveau des champs 16 bits** (**little-endian** pour idVendor, idProduct : octet bas, puis octet haut).
  Si on corrige l'un mais pas l'autre, on lit par exemple. E7 1A comme 0xE71A au lieu de **0x1AE7**. La structure du *Device Descriptor* (18 octets, idVendor aux offsets 8–9, idProduct 10–11) est documentée et doit matcher **exactement**.

9) SYNC non recherchée au bon alphabet
La **SYNC** à trouver est **00000001 en NRZ** (après NRZI+unstuff), pas "un motif analogique" ni un motif en NRZI. Côté lignes (NRZI/NIBs), c'est **KJKJKJKK** — utile pour du debug scope –, mais le code doit matcher **00000001** côté bits **NRZ**.

10) CRC & longueur payload
Un paquet **DATAx** c'est : SYNC + PID + PAYLOAD + CRC16 + EOP. Si on lit 20 octets de "données" alors que la norme dit que le *Device Descriptor* fait **18 octets**, c'est que l'on a probablement **inclus les 2 octets de CRC16** dans le payload — il faut les **exclure**.

Un script unique va intégrant toutes les évolutions (décimation auto + *phase scan*, NRZI correct, *bit unstuffing*, détection SYNC = 00000001, segmentation EOP = SE0×2, octets LSB-first, validation PID, extraction du Device Descriptor de 18 octets et calcul du flag).

Analyse_data_upd.py :
- garde tout le pipeline USB1.1 (NRZI 0=transition / 1=pas de transition, bit stuffing après 6×1, SYNC 00000001, EOP=SE0×2, octets LSB-first) avec scan de phase pour caler l'horloge ; références : NRZI/bit stuffing USB

- **NRZI USB** : *0 = transition*, *1 = pas de transition* ; **bit stuffing** : insertion d'un 0 après **6×1** consécutifs (à ignorer côté RX).

La première version de ce script pêchait sur l'ordre des opérations et l'orientation des bits : il faut NRZI→NRZ, unstuff, SYNC, octets LSB-first, segmentation EOP, validation PID, puis seulement interpréter le Device Descriptor (little-endian sur les 16-bits).

Rappels utiles :

Notes rapides (références) :

- **SYNC** débute chaque paquet : **00000001** côté NRZ (= **KJKJKJKK** sur la ligne), **EOP** = **SE0×2** bits.
- **NRZI** : **0 = transition**, **1 = pas de transition** ; **bit stuffing** : insertion d'un 0 après **six 1** consécutifs (à ignorer au décodage).
- **Octets LSB-first** après SYNC ; **PID** = nibble + **complément** (XOR=0xF).
- **Device Descriptor** = **18 octets**, idVendor/idProduct et offsets/endianness conformes à la doc

En corrigeant ces points, on lit sans ambiguïté les 18 octets : 12 01 00 02 81 02 01 40 E7 1A 3F 9C 00 01 01 02 03 01, et on en tire : bDeviceClass=0x81, idVendor=0x1AE7, idProduct=0x9C3F → 404CTF{81e71a3f9c}

« Décoder des traces brutes USB 1.1 (D+/D−) : appliquer NRZI + « bit-stuffing », repérer SYNC/PID, reconstruire les paquets, extraire le Device Descriptor et en déduire le flag»