

La langue du lecteur : ISO/IEEE 11073

Ouvrir le tuyau USB, c'est bien. Mais si on envoie n'importe quoi dedans, le lecteur ne répond rien, ou pire, il coupe la communication. Il faut parler sa langue. Et c'est là que commence vraiment le **reverse engineering**.

D'où vient la connaissance du protocole ?



Roche ne publie pas de documentation sur le protocole de ses lecteurs. Par contre, il existe un endroit où quelqu'un l'a déjà implémenté : le pilote Roche du [Tidepool uploader](#), écrit en JavaScript.

Le chemin a donc été le suivant :

1. **emogenet**, l'auteur d'origine, a lu le code JavaScript de Tidepool, récupéré les constantes et l'ordre des messages, et réécrit l'ensemble en C++ au-dessus de libusb. Il le dit lui-même en commentaire dans le code : le pilote Tidepool "ne marche que dans Chrome", et son auteur "avait sans doute accès à un manuel décrivant le protocole".
2. En lisant ces constantes, on retrouve des noms comme `MDC_ATTR_TIME_ABS` ou `MDC_MOC_VMO_PMSTORE`. En cherchant ces noms, on tombe sur la norme **ISO/IEEE 11073-20601**, celle des "Personal Health Devices" de la [Continua Health Alliance](#). C'est le protocole qui va avec la classe USB PHDC qu'on a vue dans `lsusb`.
3. Pour tout ce que Tidepool ne lisait pas (l'identité du lecteur, son horloge, les marqueurs repas), je suis reparti de la norme et des projets libres qui l'implémentent, comme [Antidote](#), puis j'ai comparé avec les vraies réponses de notre Accu-Chek Guide, capturés octets par octet.

Pour voir passer chaque paquet, il suffit de lancer `ACCUCHEK_DBG=1 accuchek`. Le programme écrit alors sur la sortie d'erreur un "hexdump" de chaque message envoyé et reçu, avec le numéro de l'étape. C'est l'outil principal de tout ce travail : on regarde les octets, on fait une

Le principe : un "agent" et un "manager"

Dans cette norme, il y a deux rôles :

- **l'agent** : l'appareil de santé, ici le lecteur. C'est lui qui a les données ;
- **le manager** : celui qui les récupère, ici notre PC.

Le lecteur ne se contente pas d'envoyer une liste de chiffres. Il se décrit comme un ensemble d'**objets**, un peu comme une petite base de données :

- l'objet **MDS** (Medical Device System, handle 0) : le lecteur lui-même, avec son fabricant, son modèle, son numéro de série, son horloge ;
- l'objet **PM-Store** (Persistent Metric Store) : la mémoire où sont rangées les mesures ;
- des **segments** à l'intérieur du PM-Store : un segment pour les glycémies, un pour les mesures de contrôle, un pour les erreurs, un pour les marqueurs repas...

Chaque objet a un numéro (son "handle") et des **attributs**. Chaque attribut a un identifiant numérique standard, c'est la "nomenclature MDC". Par exemple :

Nom	Valeur	Ce que c'est
<code>MDC_MOC_VMO_PMSTORE</code>	61	la classe "PM-Store", la mémoire des mesures
<code>MDC_ATTR_NUM_SEG</code>	2385	nombre de segments dans le PM-Store
<code>MDC_ATTR_ID_MODEL</code>	2344	fabricant et modèle
<code>MDC_ATTR_TIME_ABS</code>	2439	l'horloge du lecteur
<code>MDC_ATTR_SEG_USAGE_CNT</code>	2427	nombre d'entrées dans un segment
<code>MDC_CTXT_GLU_MEAL_FASTING</code>	29268	marqueur "à jeun"

Dans `protocol.h`, toutes ces constantes sont rangées dans une grande liste `MDC_LIST`. Une petite astuce de préprocesseur C (les "X-macros") permet d'en tirer à la fois une énumération et une fonction `findKeyByValue` qui retrouve le nom à partir du numéro. Bien pratique quand on lit des logs.

La forme d'un message

Tous les messages s'appellent des **APDU**. Ils commencent tous par 2 octets de type et 2 octets de longueur :

Type	Message	Qui l'envoie
E2 00	demande d'association ("on se connecte ?")	le lecteur
E3 00	réponse d'association ("d'accord")	le PC
E4 00	demande de fin d'association ("on raccroche")	le PC
E5 00	réponse de fin d'association	le lecteur
E6 00	abandon ("je coupe tout")	l'un ou l'autre
E7 00	message de données (tout le reste)	les deux

Les messages de données E7 00 ont une structure un peu plus riche :

```

E7 00    type : message de données
00 0E    longueur de ce qui suit
00 0C    longueur du contenu
00 11    invoke-id : numéro de la question
01 03    type d'opération : ici "GET" (lire des attributs)
00 06    longueur de l'opération
00 00    handle de l'objet visé (0 = le lecteur lui-même)
...      le reste dépend de l'opération

```

Le type d'opération (le "data choice") dit ce que fait le message :

Code	Signification
01 01	le lecteur signale un événement et attend un accusé de réception
02 01	l'accusé de réception de cet événement
01 03	demande de lecture d'attributs (GET)
02 03	réponse à cette lecture
01 07	demande d'action (lister les segments, lancer un transfert, régler l'heure...)
02 07	réponse à cette action

L'invoke-id, ou le numéro de ticket

L'**invoke-id**, c'est le numéro de ticket de la conversation. Chaque question porte un numéro, et la réponse reprend le même numéro. Le lecteur s'attend à ce que le numéro de la question suivante soit le précédent plus un. Si on se trompe de numéro, il considère que la conversation est cassée.

Dans le code, après chaque message reçu, `updateInvokeId()` relit ce numéro (octets 6 et 7), et les messages que l'on construit utilisent `invokeId + 1`. C'est une des choses les plus faciles à rater quand on réécrit le protocole, et le lecteur ne prévient pas : il abandonne, tout simplement.

Tout est en "big endian"

Les nombres sur plusieurs octets sont écrits **octet de poids fort en premier** (big endian), comme on lit un nombre de gauche à droite. `00 70` vaut `0x0070`, soit 112. C'est l'inverse de ce que fait un processeur x86, donc on ne peut pas juste copier les octets dans un entier : `protocol.cpp` a ses petites fonctions `be16`, `be32` (écriture) et `be16r`, `be32r` (lecture). On appelle ces



règles d'encodage **MDER** (Medical Device Encoding Rules).

Les dates en BCD

Dernière curiosité, et pas des moindres : les dates. Le lecteur les stocke sur 8 octets en **BCD** (Binary Coded Decimal). Chaque octet contient deux chiffres décimaux, un par moitié d'octet. Il suffit donc de lire l'hexadécimal "comme si c'était du décimal" :

```
20 26 10 01 21 06 58 00
| | | | | | | | centièmes de seconde : 00
| | | | | | | | secondes : 58
| | | | | | | | minutes : 06
| | | | | | | | heures : 21
| | | | | | | | jour : 01
| | | | | | | | mois : 10
| | | | | | | | année : 26
| | | | | | | | siècle : 20
```

=> 2026/10/01 21:06:58

Le code d'origine qualifiait ça de "weird-ass encoding", et ça m'a bien fait rire ! Je dois avouer que la première fois qu'on voit `0x26` qui veut dire 26 et pas 38, ça surprend. La fonction `decodeBcd` fait la conversion, et renvoie `-1` si un des deux chiffres dépasse 9 : dans ce cas l'octet n'est pas du BCD valide, et la date est marquée invalide au lieu de produire n'importe quoi.

```
int decodeBcd(uint8_t x) {  
    auto hi = (x >> 4);  
    auto lo = (x & 0xF);  
    if(9<hi || 9<lo) {  
        return -1;  
    }  
    return 10*hi + lo;  
}
```

Avec ces quelques règles en tête (APDU, invoke-id, big endian, BCD, nomenclature MDC), on a tout ce qu'il faut pour suivre la conversation complète entre le PC et le lecteur. C'est le sujet du chapitre suivant.

Révision #4

Créé 2026-10-02 20:08:53 UTC par Renard

Mis à jour 2026-10-03 12:35:36 UTC par Renard