Programmation USB sous GNU/Linux : application du FX2LP pour un récepteur de radio logicielle dédié aux signaux de navigation par satellite (1/2)

Jean-Michel Friedt, enseignant-chercheur à l'université de Franche-Comté à Besançon, 16 octobre 2024

Alors que USB est souvent abordé comme un bus émulant un port série, tirer pleinement profit de sa bande passante nécessite d'exploiter les interfaces disponibles les plus appropriées, et en particulier *Human Interface Device* (HID) et transferts en volume (*Bulk*). Nous proposons d'appréhender le bus USB exposé par le noyau Linux en vue d'en tirer le maximum du débit disponible, et appliquer cette connaissance en réalisant un récepteur de radio logicielle dédié à la réception des signaux de navigation par satellite (GNSS) en bande L (1–2 GHz) grâce au MAX2771. Nous démontrons le bon fonctionnement du circuit avec l'acquisition et le traitement de signaux issus de diverses constellations de GNSS en orbite intermédiaire MEO et Iridium en orbite basse LEO observés avec une bande passante pouvant aller jusqu'à 44 MHz.

Notre objectif est de réaliser un récepteur, par radio logicielle, des signaux de navigation par satellite (GNSS pour *Global Navigation Satellite Systems*) et autres signaux satellitaires transmis dans la bande 1–2 GHz qualifiée de bande L. Les modes de modulation les plus récents (Galileo européen, L5 américain) occupant jusqu'à 41 MHz de bande passante (pour Galileo E6, voir `https://gssc.esa.int/navipedia/index.php/Galileo_Signal_Plan`), nous devons aborder le transfert "rapide" de données entre l'interface d'acquisition et l'ordinateur chargé de stocker les mesures – sans prétention de les traiter en temps réel mais tout de même de capturer tous les échantillons sans en perdre un seul. Pour ce faire, nous devons apprendre à aborder les divers modes de communication du bus USB, et leur mise en œuvre dans un composant de Cypress, le CY7C68013A aussi nommé FX2LP, aussi quelque peu abusivement nommé EZ-USB. Ce composant embarque un vénérable cœur de processeur 8051 [1] pour configurer les diverses interfaces de communication : le 8051 est trop petit, avec ses 1,5 accumulateurs (A et B) et ses 8 (R0–R7) registres 8 bits, pour être abordé par GCC et les codes seront donc compilés au moyen du *Small Device C Compiler* SDCC. Nous nous inspirerons au cours de ces développements du système fonctionnel proposé par Tomoji Takasu de l'Université de Tôkyô pour les Sciences et Technologies Maritimes (*Tôkyô University of Marine Science and Technology*), aussi auteur de RTKLib, célèbre parmi les utilisateurs qui visent la résolution centimétrique de leur récepteur GPS [2]. Ce système est nommé PocketSDR et disponible à `https://github.com/tomojitakasu/PocketSDR`. Bien que le circuit soit librement distribué au format KiCAD sur ce dépôt, le choix de l'auteur a été d'imposer de travailler dans deux bandes de fréquences différentes – dites haute (L1) et basse (L2, L5) dans la nomenclature du composant MAX2771 chargé de recevoir les signaux GNSS – et d'incorporer l'interface de communication parallèle vers USB FX2LP. Comme l'EZ-USB FX2LP peut être obtenu assemblé et prêt à l'emploi pour un peu moins de 5 euros sur AliExpress quand le composant seul coûte près de 20 euros chez Farnell (code commande 1269134), nous avons fait le choix de travailler sur un circuit prêt à l'emploi `https://fr.aliexpress.com/item/1005006134347046.html` et de lui adjoindre l'interface d'acquisition à base de MAX2771, dans un premier temps sous forme de la carte d'évaluation commercialisée par Maxim IC (maintenant Analog Devices), fabriquant du composant. Finalement, Tomoji Takasu utilise un compilateur propriétaire pour convertir son code source en binaire à destination du 8051, solution évidemment inacceptable à laquelle il faudra remédier en convertissant ses sources à SDCC, ce qui ne se fera pas sans douleur tel que nous le verrons ci-dessous. À l'issue de ces développements, nous commencerons par aborder la constellation en orbite basse Iridium qui, bien qu'émettant dans des fréquences juste au-dessus de la bande GNSS occupée par le système russe GLONASS, reste dans la bande de fréquences accessible au MAX2771 avec un signal beaucoup plus facile à détecter en vue de déverminer le bon fonctionnement des interfaces d'acquisition et de communication, avant de finalement décoder GPS.

Le plan des deux articles relatant ces explorations est le suivant :

1. partant d'une carte d'évaluation du MAX2771 fournie avec un logiciel propriétaire communiquant en HID sous MS-Windows, nous identifions le protocole de communication et l'implémentons en Python sous GNU/Linux en s'appuyant sur `libusb`,
2. ayant découvert que de toute façon ladite carte d'évaluation n'est pas conçue pour acquérir et transmettre des données mais uniquement configurer le périphérique (? !), nous reprenons la partie numérique en éliminant le microcontrôleur fourni d'origine et en le remplaçant par un FX2LP,

3. après avoir développé le premier programme de base (clignoter une LED) sur le microcontrôleur équipant le FX2LP pour valider la compréhension du compilateur, de la bibliothèque associée et des outils de communication entre le PC et le microcontrôleur pour exécuter le binaire compilé sur la cible ...
4. ... nous portons l'implémentation logicielle (*bitbang*) du bus SPI (*Serial Peripheral Interface*) proposée par PocketSDR vers le compilateur SDCC, et apprenons à communiquer par des *Vendor Requests* pour exécuter des ordres au travers du bus USB.
5. Finalement, nous complétons le portage du *firmware* fourni par PocketSDR en ajoutant les communications Bulk sur USB, et validons les divers débits de communication en fonction des registres de configuration de l'horloge cadençant les convertisseurs analogique-numérique.
6. Le bon fonctionnement du montage est validé sur les signaux "puissants" des satellites en orbite basse Iridium, puis sur les signaux sous le bruit thermique des satellites de navigation.
7. Dans le prochain article, nous aborderons la conception et réalisation d'une carte fille dédiée embarquant deux MAX2771 pour des mesures différentielles de signaux radiofréquences acquis par FX2LP et transmis par USB Bulk vers le PC pour post-traitement, en particulier pour pallier les déficiences du couplage entre signaux portés par des fils de communication trop longs dans le premier circuit de prototypage.
8. Finalement, nous ajouterons la capacité à corriger la source de fréquence qui cadence le MAX2771 afin de corriger son écart à la fréquence nominale GPS, et ce en ajoutant de nouvelles *Vendor Requests* afin de programmer ce nouveau périphérique ajouté au bus s'apparentant à SPI.

La seule modification lors de l'utilisation de la carte d'évaluation du MAX2771 est de remplacer l'oscillateur avec sa fréquence par défaut de 16,368 MHz par un oscillateur à 24 MHz afin de pouvoir utiliser tels quels les fichiers de configuration proposés par PocketSDR. Nous verrons en section 2 comment recompiler le firmware et ainsi pouvoir changer la configuration de PocketSDR pour supporter un oscillateur à 16,368 MHz.

Ainsi, dans le prochain épisode, quitte à remplacer le microcontrôleur de la carte d'évaluation à plus de 400 euros, nous poursuivrons en proposant notre propre implémentation d'une carte fille munie de deux MAX2771 s'adaptant à une carte faible coût (5 euros) munie d'un FX2LP, finalisant la démarche sur un circuit coûtant moins d'une centaine d'euros tout compris (MAX2771, FX2LP et antennes GNSS multibandes ou Iridium). En particulier, nous éliminerons de ce fait les rayonnements électromagnétiques indésirables lorsque des signaux à plusieurs MHz circulent sur des fils non-blindés d'une 10aine de centimètres de long, induisant corruption des signaux numériques et bruit en bande de base lors de l'analyse des signaux transposés par l'oscillateur local avant conversion analogique-numérique, tel que nous le verrons en conclusion.

1 Le bus USB

Historiquement, un ordinateur communique en point à point par un protocole sérialisant dans le temps les données au lieu de les communiquer en parallèle – la seconde approche est limitée en débit de communication par le couplage inductif entre fils adjacents et la taille de la nappe de fils nécessaire pour porter tous les bits. RS232 est un exemple de bus série, qualifié d'asynchrone puisque les interlocuteurs ne partagent pas d'horloge. Dans la même veine, SPI est un bus série synchrone puisqu'un maître distribue un signal d'horloge vers ses esclaves. Le vénérable protocole qu'est RS232, encore largement utilisé dans l'embarqué, n'est cependant que rarement disponible sur les ordinateurs personnels actuels, et les interfaces USB-RS232 sont pléthores. Ainsi, le protocole de communication CDC d'USB – *Communications Device Class* – fait croire à l'hôte (l'ordinateur) que le périphérique parle le RS232 quand en réalité les informations sont portées sur bus USB. Ce mode de communication est très pratique puisqu'il permet d'utiliser les outils associés à RS232 – `minicom` et `screen` pour ne citer qu'eux – mais ne permet pas de tirer partie de tout le débit accessible sur USB. Dans LUFA par exemple (`http://www.fourwalledcubicle.com/LUFA.php`), un Atmega32U4 se configure facilement en CDC par

```
#define F_CPU 16000000UL  //T=62.5ns
#define F_USB 16000000UL
#include "VirtualSerial.h"
#define N 1024
```

```
extern USB_ClassInfo_CDC_Device_t VirtualSerial_CDC_Interface;
extern FILE USBSerialStream;

int main(void){
  char tab[N]; memset(tab,'U',N);
  SetupHardware();
  CDC_Device_CreateStream(&VirtualSerial_CDC_Interface, &USBSerialStream);
  GlobalInterruptEnable();
  while(1) {fwrite(tab,1,N,&USBSerialStream); USB_USBTask(); }
}
```

et connecter le microcontrôleur programmé par cette séquence d'instructions au bus USB d'un PC fait apparaître une interface `/dev/ttyACM0` accessible par `minicom -D /dev/ttyACM0` sans que le débit (*baudrate*) n'ait d'importance ici. Quelques mesures de débit, pour `N` allant de 64 à 1024 en puissances de deux, indique par

```
timeout 10 cat < /dev/ttyACM0 > fichier
```

une taille de fichier de l'ordre 965 ± 5 kB donc des débits de l'ordre de 100 kB/s, en accord avec les tests de `https://www.pjrc.com/teensy/benchmark_usb_serial_receive.html` pour l'Atmega32U4 qui équipe une carte Arduino Leonardo. D'après ce même site, un microcontrôleur un peu plus puissant doit atteindre le MB/s, encore loin de la dizaine de MB/s que nous visons pour une application de radio logicielle. Il faut donc se tourner vers une utilisation optimisée du bus USB et ne pas se contenter de juste exposer un port série virtuel.

Nombre de dispositifs "récents" n'exploitent pas CDC mais exposent une interface plus riche, et c'est par exemple le cas du microcontrôleur qui équipe la carte d'évaluation du MAX2771 (Fig. 1). En effet, ayant acquis cette carte il y a plusieurs années, elle devint un très beau presse-papiers quand j'ai réalisé que ni GNU/Linux, ni Wine ne pouvaient communiquer avec elle au travers du logiciel propriétaire fourni par Maxim IC. Comme un presse-papiers à 450 euros acquis avec l'argent du contribuable est discutable, nous nous interrogeons à comprendre son mode de communication et les échanges entre le microcontrôleur dont la carte d'évaluation est munie, et le PC. Bien entendu sans accès au code source du *firmware* exécuté sur ledit microcontrôleur, nous ne pourrons que tenter une rétro-ingénierie en sondant le bus USB lorsque le logiciel original communique avec la carte d'évaluation : ce logiciel est exécuté depuis une machine virtuelle VirtualBox munie de MS-Windows et exécutant le logiciel propriétaire disponible sur le site du fabriquant [3].

À la connexion de l'interface USB du microcontrôleur, Linux nous informe (`dmesg`) que

```
[X] usb 1-2: Product: HID DEVICE
[X] usb 1-2: Manufacturer: mbed.org
[X] usb 1-2: SerialNumber: 0123456789
[X] hid-generic 0003:1234:0006.0005: hiddev1,hidraw2: USB HID v1.11 Device
[mbed.org HID DEVICE] on usb-0000:00:14.0-2/input0
```

donc il s'agit d'un *Human Interface Device* dont nous devons comprendre le protocole de communication.

1.1 `usbmon` pour la rétro-ingénierie d'une interface HID

HID est l'interface de nombre de périphériques USB relativement bas débit, incluant les souris et les claviers, et son analyse est donc l'objet de nombreuses études, notamment `https://www.baeldung.com/linux/usb-sniffing` qui nous enseigne que le noyau Linux propose un mode de déverminage du bus USB pour afficher les octets échangés. En-effet

```
$ sudo mount -t debugfs device_debug /sys/kernel/debug
$ sudo modprobe usbmon
```

crée les pseudo-fichiers dans `/sys/kernel/debug/usb/usbmon/Xu`, avec `X` le numéro du bus, donnant accès aux ressources du noyau. Après avoir identifié sur quel bus USB le périphérique est connecté par `lsusb` du genre

```
Bus 001 Device 039: ID 1234:0006 Brain Actuated Technologies HID DEVICE
```

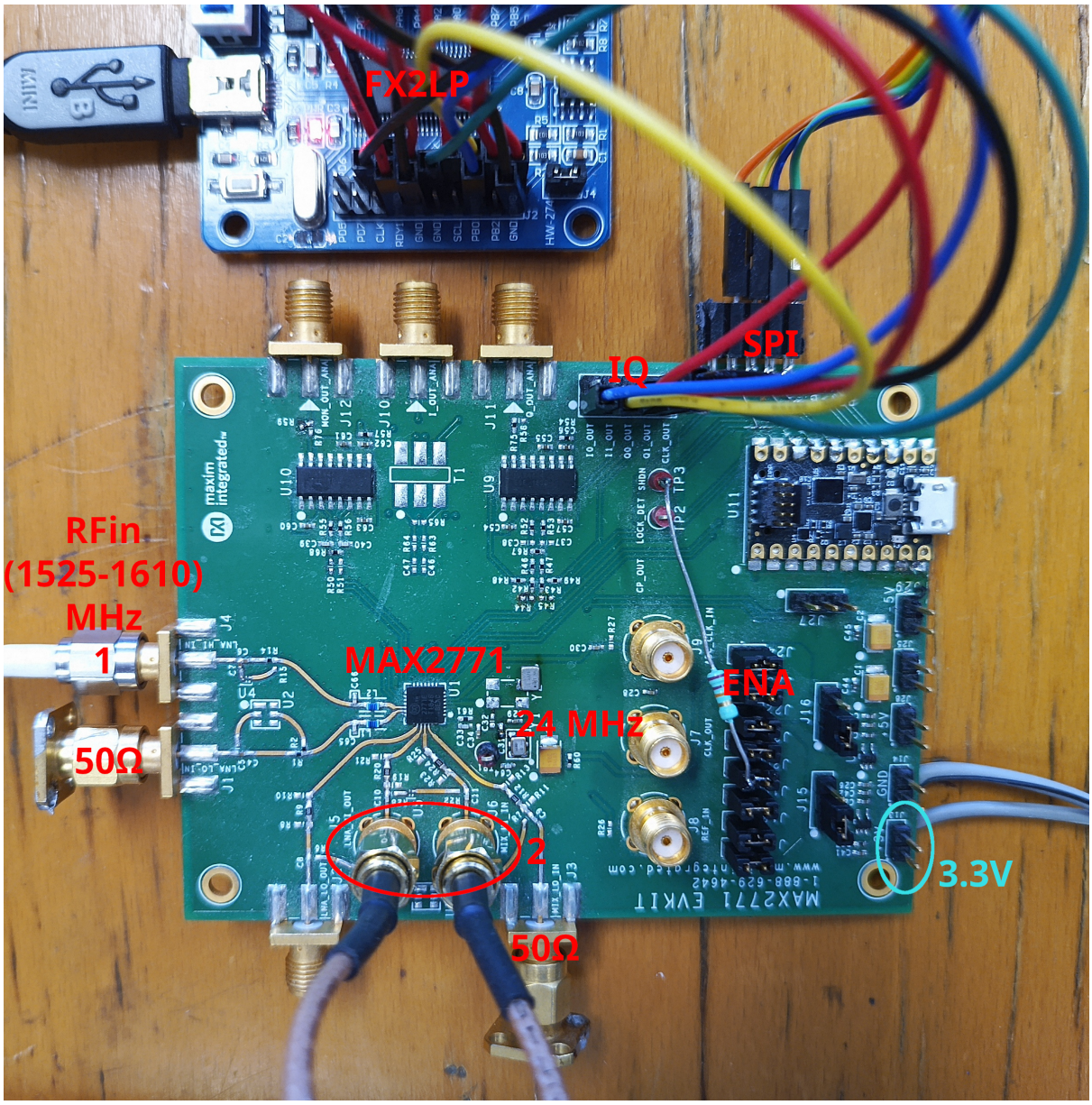


FIGURE 1 – La carte d'évaluation du MAX2771 de Maxim IC, source de tous nos déboires, avec son microcontrôleur (en haut à droite) qui ne peut communiquer qu'avec le logiciel propriétaire exécuté sous MS-Windows, et qui ne s'avère finalement capable que de convertir des commandes USB en SPI. Cette carte est déjà équipées des liaisons SPI et IQ (bornier à côté du microcontrôleur) que nous redirigerons vers l'EZ-USB afin de corriger ces déficiences. Seule l'alimentation 3 V (bas-droite, en cyan) est nécessaire pour faire fonctionner la carte, les alimentations symétriques ne servent que pour les amplificateurs opérationnels qui ne sont pas utilisés. Pour tester la bande supérieure GNSS, on branchera un signal ou une antenne active sur le connecteur de gauche (marque 1) et relierons la sortie de l'amplificateur faible bruit (LNA) à l'entrée du mélangeur (ellipse rouge 2). Une résistance de tirage vers l'alimentation sort le MAX2771 de son mode veille (ENA) lorsque le microcontrôleur de la carte n'est pas alimenté.

nous utiliserons le premier argument (ici 1) en place de `X` pour sonder les messages transmis sur ce bus. Noter qu'il peut être malin de ne *pas* brancher le microcontrôleur de la carte d'évaluation sur le même bus USB qu'une souris communiquant par cette interface pour ne pas être pollué par les messages de celle-ci chaque fois que nous déplaçons le curseur de l'interface graphique. On pourra au pire `|grep YYY` avec `YYY` le numéro de *Device* pour filtrer.

Tel quel, le microcontrôleur n'est pas causant. Cependant si nous exécutons le logiciel propriétaire

proposé par ADi pour communiquer depuis MS-Windows [3], dans VirtualBox, nous obtenons une série de messages sous réserve que l'utilisateur appartienne au groupe `vboxusers` et que l'Extension Pack soit installé (support USB de VirtualBox), de la forme

```
# cat /sys/kernel/debug/usb/usbmon/3u
ffff89247452b840 3730820207 S Io:3:106:3 -115:1 64 = c9000000 00000000 00000000 0000...
ffff89247452b840 3730821534 C Io:3:106:3 0:1 64 >
ffff89245ffbec00 3730821653 C Ii:3:106:4 0:1 64 = cb010000 00000000 00000000 0000000...
ffff89245ffbe3c0 3730822364 S Ii:3:106:4 -115:1 64 <
ffff8923b4e30480 3731821079 S Io:3:106:3 -115:1 64 = c9000000 00000000 00000000 0000...
ffff8923b4e30480 3731822566 C Io:3:106:3 0:1 64 >
ffff89245ffbe3c0 3731822679 C Ii:3:106:4 0:1 64 = cb010000 00000000 00000000 0000000...
ffff8923b4e309c0 3731823216 S Ii:3:106:4 -115:1 64 <
ffff89245ffbec00 3732762122 S Io:3:106:3 -115:1 64 = c20000a2 24160300 00000000 0000...
...
```

donc le logiciel sous MS-Windows demande périodiquement au microcontrôleur s'il est présent. Le microcontrôleur acquitte de chaque requête (message `c9` suivi de 0s). En plus de ces requêtes périodiques, le logiciel propriétaire permet de configurer les registres du MAX2771, et dans ce cas une série de messages commençant par `c2` indique le numéro du registre et la valeur à y stocker. Une fois le protocole identifié, il ne reste plus qu'à l'implémenter, par exemple en s'inspirant de `https://github.com/david0/durgod-keymapper/blob/master/remap.py` qui nous enseigne comment communiquer un message à un prériphique HID dont on connaît le Vendor ID et le Product ID (`VID:PID` – ici `1234:0006` tel qu'indiqué auparavant par `lsusb`), de la forme

```
def tohex(data):
    return '␣'.join(map(lambda x: "%02x" % x, data))

import hid
VENDOR_ID=0x1234
PRODUCT_ID=6
RESET = b"\xc9".ljust(31, b"\x00")
device_info = next(device for device in hid.enumerate()
... if device['vendor_id'] == VENDOR_ID and device['product_id'] == PRODUCT_ID)
device=hid.device()
device.open_path(device_info['path'])
device.write(RESET)                  # envoi du message C9 00 00 ...
resp=device.read(64, timeout_ms=500) # reception de la reponse
resp=bytearray(resp).rstrip(b'0x00');
print(tohex(resp))
```

pour envoyer `c9` suivi de 0s, et en effet [1] le microcontrôleur répond

```
$ python3 ./max2771.py
cb 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...
```

donc `cb 01` indiquant que le microcontrôleur acquitte notre requête formée de `0xc9` suivi de 31 zéros (`b"\xc9".ljust(31, b"\x00")`). La voie est donc toute tracée, il ne reste plus qu'à comprendre les diverses commandes du logiciel propriétaire et les implémenter en Python. Ce faisant, un point attire cependant une interrogation : aucun bouton sur le logiciel propriétaire de communication ne permet d'acquérir les données, seulement de communiquer au travers du bus SPI des configurations des registres du MAX2771. Ayant sollicité l'aide des ingénieurs de Maxim, nous avons eu confirmation que la communication des données acquises par le MAX2771 était prévue ... mais n'a jamais été implémentée. Nous avons donc acquis un convertisseur USB vers SPI à 450 euros ! Inutile de câbler les résistances R31/R32 et R44/R45 (*I0/I1 to host* et *Q0/Q1 to host*) sur la carte d'évaluation, le microcontrôleur ne saura que faire de ces signaux IQ représentatifs du signal électrique converti en signal numérique par le MAX2771. Nous devons donc nous débrouiller seuls pour connecter une interface parallèle-USB sur le bornier J26

1. Tous les codes cités dans cet article sont disponibles à `https://github.com/jmfriedt/max2771_fx2lp/` y compris ce `https://github.com/jmfriedt/max2771_fx2lp/blob/main/Maxim_EvalBoard/max2771.py`

propageant les signaux IQ issus du récepteur de signaux GNSS pour les transmettre au PC, et c'est là qu'intervient le Cypress CY7C68013A.

I0/I1 et Q0/Q1 : conversion analogique-numérique sur 2 ou 3 bits !
On pourrait être surpris de la nomenclature I0/I1 et Q0/Q1 qui laisse penser que les signaux complexes $I + jQ$ sont codés sur deux bits seulement, ou un bit de signe et un bit de valeur. Ce codage est courant dans l'analyse des signaux GNSS qui se situent 20 dB sous le bruit thermique (voir section 3) et pour lesquels un nombre important de bits ne ferait que coder le bruit, pas le signal. La puissance du code pseudo-aléatoire (CDMA) qui encode les messages GNSS est que la corrélation fait ressortir le signal du bruit d'un facteur égal au facteur de compression (*pulse compression ratio* donné par le produit bande passante multiplié par la durée du code, ou en numérique le nombre de bits du code). Avec un code sur 1023 bits pour GPS L1, le gain de compression est $\log_2(1023) \simeq 10$ bits donc les trois bits deviennent 13 bits après corrélation par le code connu, voir 16 bits pour les codes dix fois plus longs de GPS L5. À notre grande surprise, aussi peu de bits de codage resteront suffisants pour décoder Iridium qui ne bénéficie pas d'un gain de compression du codage CDMA mais profite d'un signal bien plus puissant transmis par ses satellites en orbite basse. Ainsi, le MAX2771 peut soit fournir les mesures complexes (parties réelle et imaginaire) en bande de base sous forme $I + jQ$ avec I et Q codés sur 2 bits chacun, ou bien en présence d'une fréquence intermédiaire acquérir la partie réelle uniquement (donc spectre pair) codée sur 3 bits selon $I1, I0, Q1$ (donc le bit de poids le plus faible sur $Q1$ même si l'information porte sur une valeur réelle), et il sera à la charge de l'utilisateur d'effectuer numériquement la transposition de fréquence (multiplication par un oscillateur local de fréquence égales à la fréquence intermédiaire) pour produire les valeurs complexes en bande de base.

En effet, Tomoji Takasu étant moins incompétent que Maxim IC – ou surtout plus motivé que les mercenaires payés ponctuellement à développer le logiciel de la carte d'évaluation – il fournit la solution avec l'EZ-USB du FX2LP : ce composant se connectera aux broches IQ du bornier *et son horloge associée* (communication synchrone) pour traduire vers un flux USB les mesures proposées au format parallèle par le MAX2771. Mais pour en arriver là, il faut programmer le microcontrôleur 8051 embarqué dans le CY7C68013A pour en configurer les interfaces...

1.2 EZ-UZB FX2LP

L'EZ-USB ne date pas d'hier [5], mais malheureusement la majorité des projets sur `github` qui le concernent ont plus de dix ans et bien du mal à encore fonctionner aujourd'hui. Heureusement, la bibliothèque qui est associée à `sdcc` [6] pour le FX2LP, `fx2lib` à `https://github.com/djmuhlestein/fx2lib` fonctionne encore, et une version déclinée de `fx2lib` est maintenue par `sigrok` [7].

Deux outils pour transférer le *firmware* cross-compilé sur PC vers le microcontrôleur sont `fxload` à `https://github.com/mbed-ce/fxload`, que nous préférerons à `https://github.com/esden/fxload` ou au paquet binaire Debian GNU/Linux issu de Sourceforge compte tenu des améliorations récemment amenées, pour transférer le programme en mémoire volatile RAM ou non-volatile EEPROM, et plus rapide dans un premier temps sera `cycfx2prog` aussi disponible comme paquet binaire Debian. Muni de ces outils, le premier objectif est de faire clignoter une LED pour valider la compréhension du 8051 et des outils de programmation. Malheureusement, sélectionner une carte de développement au rabais a bien sûr quelques conséquences : la version 56 broches du CY7C68013A ne route pas les interfaces de communication asynchrones (UART, compatible RS232, uniquement disponible sur les boîtiers avec plus de broches) [8, page 18] donc nous devrons nous contenter de cette LED comme mode de communication jusqu'à maîtriser les interfaces USB.

1.3 Clignotement d'une LED

Ce premier exemple s'appuie sur `https://github.com/sidd-kishan/fx2lp-blinky` faute de documentation du fonctionnement des registres du FX2LP, le cœur de 8051 étant entouré de bien plus de périphériques que le microcontrôleur original. La description des registres dans la documentation technique (incluant leur emplacement en mémoire) est des plus succintes.

La carte de développement du FX2LP est munie de deux LEDs qui sont ou non activables selon qu'un jumper les relie ou non à l'alimentation (GPIO en puits de courant, schéma de la carte à [9]). Ces deux LEDs sont commandées depuis le port A et sont connectées aux broches 0 et 1. Ainsi, le programme trivial

```c
#include <fx2regs.h>
#include <delay.h>
#define led12 3  // 1<<0 | 1<<1
void main(void)
{unsigned char val=1;
 OEA=(led12);    // PA0, PA1 output
 while (1)
 {val=led12-val; // 2 <-> 1
  IOA = val;
  delay(1000);   // wait 1 s
 }
}
```

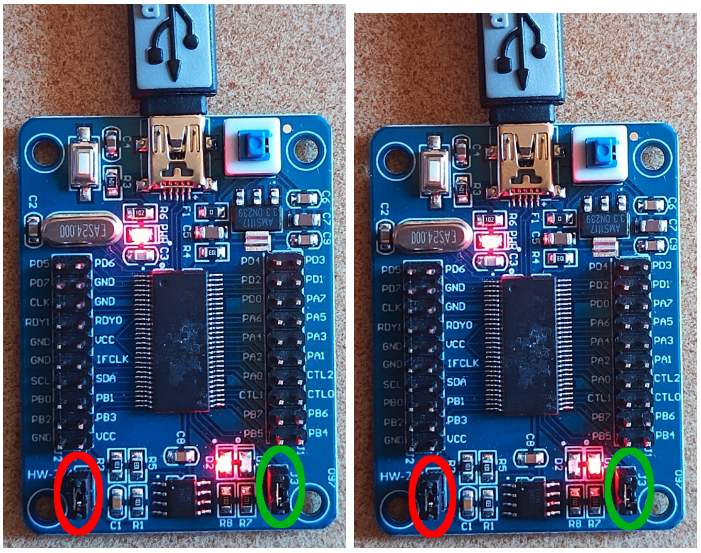


FIGURE 2 – Deux photographies de la carte de développement FX2LP alors que la LED de gauche est allumée et celle de droite éteinte (gauche) ou le contraire (droite), à côté du jumper entouré en vert qui les relie à l'alimentation. Le jumper entouré en rouge sert à invalider l'adresse mémoire de l'EEPROM au démarrage pour permettre de la reprogrammer (voir section 1.4).

initialise deux bits du port A en sortie (registre `OEA`, bit à 1 en sortie) puis manipule le registre définissant l'état des broches en sortie `IOA`, alternant la LED allumée et éteinte chaque seconde – LED de gauche allumée sur la photographie de gauche et LED de droite sur la photographie de droite dans ces illustrations. Ce programme, compilé puis lié avec la bibliothèque FX2Lib de `https://github.com/djmuhlestein/fx2lib` que nous avons au préalable compilée par `make` pour générer `lib/fx2.lib`, est converti en fichier hexadécimal compatible Intel (extension `ihx`) par `sdcc` par

```
sdcc -mmcs51  mainPA.c -I../fx2lib/include/ -L../fx2lib/lib/ fx2.lib
```

pour produire implicitement `mainPA.ihx`, qui est exécuté depuis la RAM du 8051 par `sudo cycfx2prog prg:mainPA.ihx run`. Nous constatons sur les deux figures accompagnant le code que les deux jumpers sont en place, le rouge pour faire booter le FX2LP en mode bootloader et charger l'exécutable au format Intel hexadécimal (`ihx`) en RAM, et le vert pour connecter les LEDs à l'alimentation. Divers `Makefile` pour FX2LP ajoutent des options d'édition de lien du type `--code-size 0x1c00 --xram-size 0x0200 --xram-loc 0x1c00` mais les valeurs par défaut semblent suffisantes pour ne pas devoir les expliciter.

Ce faisant, nous avons programmé le 8051 comme n'importe quel microcontrôleur généraliste, sans tirer parti d'une de ses originalités. En effet pour faire clignoter une autre LED externe que nous connectons entre PD7 et la masse par exemple, le code

```c
#include <fx2regs.h>
#include <delay.h>
void main()
{OED = (1 << 7);             // direction PD7 out
 while (1)
 {PD7 = 0; // IOD = 0;       // IOD: set register
  delay(1000);
  PD7 = 1; // IOD = (1<<7); // setbit sets one bit only
  delay(1000);
 }
}
```

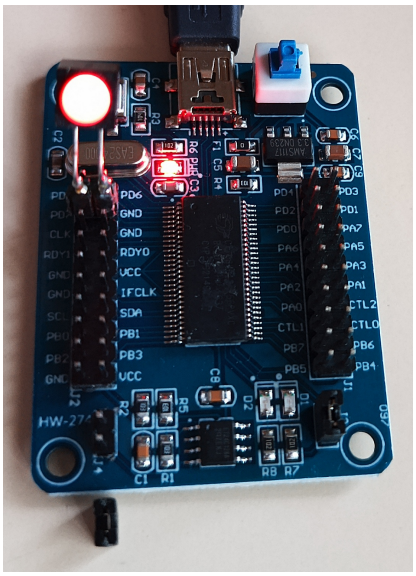


FIGURE 3 – Exemple de LED qui clignote sur PD7. Noter le jumper, en bas à gauche, qui a été retiré pour permettre l'identification de l'EEPROM et exécuter le code stocké en mémoire non volatile par `fxload`.

(Fig. 3) manipule l'unique bit PD7, en laissant en commentaire la manipulation du registre de données `IOD` associé au port D, toujours après avoir placé la broche en sortie en manipuant le bit 7 de `OED`. En effet, Maxim IC nous rappelle les fondamentaux de SDCC [10] et en particulier la capacité du 8051 à adresser un bit unique (SBIT) sans manipuler le registre complet qui le contient (SFR pour *Special Function Register*). Cette fonctionnalité était limpide lors de la programmation en assembleur, mais est rendue quelque peu obscure lors du passage au langage C avec des macros du type `__sbit __at(0xB0+7) PD7;` dans `include/fx2regs.h` de FX2Lib pour faire appel à l'instruction assembleur `SBIT` spécifique au 8051.

Une fois ce programme compilé comme auparavant, cette fois l'exécutable est placé en mémoire non-volatile EEPROM pour exécution lors de la mise sous tension par

```
sudo fxload load_eeprom --device 04b4:8613 --ihex-path main.ihx -t FX2LP --control-byte 0xC2 -s Vend_Ax.hex
```

qui a besoin du *firmware* `Vend_Ax.hex` pour communiquer avec l'EEPROM et identifier le VID et PID du FX2LP en mode *bootloader* (`04B4:8613` tel qu'indiqué par `lsusb`). Noter qu'avec des anciennes versions de `fxload`, la commande
`sudo fxload -D /dev/bus/usb/001/035 -I main.ihx -c 0xc2 -s Vend_Ax.hex -t fx2lp`
était beaucoup plus pénible puisque le pseudo-fichier dans `/dev/bus/usb` change de nom à chaque mise hors tension puis sous tension du FX2LP avec un identifiant qui s'incrémente à chaque fois. Attention cependant, nous pouvons écrire une fois en mémoire non-volatile EEPROM, mais pas une seconde fois, la solution à ce problème sera fournie plus bas (section 1.4). En attendant, nous conseillons de tester le bon fonctionnement du logiciel en RAM par `cycfx2prog`.

Le code source complet de cet exemple est disponible à `https://github.com/jmfriedt/max2771_fx2lp/tree/main/FX2LP/LED_blink`.

Ayant validé la compilation et la programmation du composant en faisant clignoter une LED, nous pouvons nous attaquer au cœur du sujet, la communication sur bus USB.

1.4 Reprogrammation de l'EEPROM

Nous avons mentionné qu'une fois l'EEPROM flashée, nous n'arrivions pas à y placer un second programme différent du premier. En effet, un premier transfert de programme depuis le PC vers l'EEPROM se solde par un succès avec le message

```
FX2:  config = 0x42, disconnected, I2C = 100 KHz
Done.
```

mais au second essai, le message est cette fois

```
WARNING: don't see a large enough EEPROM
```

qui indique un échec. Le problème vient des attentes de `fxload` qui ne sont plus respectées une fois l'EEPROM flashée une première fois, tel que nous allons l'expliciter.

D'après le schéma du circuit contenant le FX2LP [9], le jumper J4 le plus éloigné des LEDs doit être mis en place pour passer le FX2LP en mode bootloader afin que la programmation de l'EEPROM soit possible. Ce faisant, la capacité du FX2LP à trouver une mémoire non-volatile contenant un firmware valable est invalidée au démarrage, forçant le 8051 à passer en mode *bootloader*. Dans l'implémentation du circuit que nous exploitons, cet objectif est atteint en court-circuitant le bit de poids faible de l'adresse de l'EEPROM à la masse, donc en plaçant sa valeur à 0. Cependant, ceci implique que le jumper doit être *retiré* au moment de programmer l'EEPROM, puisque la *datasheet* du FX2LP explicite que l'adresse attendue pour le périphérique de stockage doit être 0x51 sur le bus I^2C (dans la nomenclature qui ne conserve que les 7 bits de poids fort de l'adresse et omet de mentionner que les deux adresses transmises sur I^2C sont 0xA2 ou 0xA3 pour une transaction en écriture ou lecture respectivement). Dans la configuration proposée, le jumper impose un potentiel nul au bit de poids faible de l'EEPROM, définissant donc une adresse sur le bus I^2C de 0x50 qui ne peut être reconnue lors de la programmation. Il faut donc absolument *retirer le jumper avant de flasher l'EEPROM*, faute de quoi le composant ne peut être détecté sur le bus I^2C. Pour s'en convaincre, nous suivons les consignes de [11] à savoir

1. transférer en mémoire volatile le programme `Vend_Ax.hex` disponible par exemple dans les archives de `fxload` par `sudo cycfx2prog prg:Vend_Ax.hex run` (ou si on préfère rester avec `fxload`, par `sudo fxload load_ram --device 04b4:8613 --ihex-path Vend_Ax.hex -t FX2LP`)
2. exécuter le programme qui lance l'ordre *Vendor Request* 0xA2 afin de lire le contenu de l'EEPROM selon

```
import usb
import binascii
VID = 0x04B4
PID = 0x8613
dev = usb.core.find(idVendor = VID, idProduct = PID)
ret=dev.ctrl_transfer(0xC0,0xa9, 0, 0, 32)  # read EEPROM content
print(binascii.hexlify(ret))
```

3. si nous laissons le jumper en place, la réponse `b'cdcdcdcdcdcd...` indique que l'EEPROM n'a pas été lue, son adresse ne correspond pas à celle attendue par `Vend_Ax`
4. si nous retirons le jumper, alors la réponse `b'c2b404138605a04203f20000` commence bien par 0xC2 tel que documenté en page 8 du manuel technique [8] pour indiquer "*During the power-up sequence, internal logic checks the I^2C port for the connection of an EEPROM whose first byte is either 0xC0 or 0xC2. If found, it uses the VID/PID/DID values in the EEPROM in place of the internally stored values (0xC0), or it boot-loads the EEPROM contents into internal RAM (0xC2).*" suivi du VID et PID du composant en format *little endian*, donc "à l'envers" pour un lecteur occidental qui lit de gauche à droite : b4041386 s'interprète comme 04b4 :8613.

D'après le code source de `fxload` à `https://github.com/mbed-ce/fxload/blob/master/src/ezusb.c#L674-L676` celui-ci attend une réponse de 1 à la requête `GET_EEPROM_SIZE` (*Vendor Reqest* 0xA5 prise en charge par `Vend_Ax`) et sinon refuse de reflasher l'EEPROM. Or émettre la commande 0xA5 vers un FX2LP exécutant `Vend_Ax` renvoie la valeur 0 et donc `fxload` refuse de continuer la programmation.

Par ailleurs, le premier octet de l'EEPROM contient 0xC2 ou 0xC0 selon la façon de configurer l'identifiant USB. L'alternative de retirer le jumper pour passer en mode *bootloader* échoue aussi puisque le FX2LP voit une EEPROM dont le premier octet contient 0xC2 et donc en exécute le code. La solution, fort peu élégante, que nous avons trouvée pour reflasher l'EEPROM du FX2LP est d'exécuter un programme depuis la RAM, donc chargé par `cycfx2prog`, pour écraser les premiers octets de l'EEPROM avec 0xFF, et ainsi forcer le microcontrôleur avec le jumper retiré à exécuter son bootloader en l'absence de *firmware* valide. Pour ce faire, nous chargeons en RAM (`cycfx2prog`) le programme `Vend_Ax.hex` par `sudo cycfx2prog prg:Vend_Ax.hex run` et allons profiter de sa *Vendor Request* 0xA9 pour écrire dans l'EEPROM tel que décrit à [11], et ainsi écraser le premier octet pour forcer le lancement du *bootloader* qui permettra de reflasher l'intégralité de l'EEPROM :

```
import usb
import binascii

VID = 0x04B4
PID = 0x8613
dev = usb.core.find(idVendor = VID, idProduct = PID)
ret=dev.ctrl_transfer(0xC0,0xa9, 0, 0, 64)  # read EEPROM content
print(binascii.hexlify(ret))
msg=bytearray([0xff,0xff,0xff,0xff]);       # write EEPROM (erase)
dev.ctrl_transfer(0x40 , 0xa9, 0x0, 0, msg)
ret=dev.ctrl_transfer(0xC0,0xa9, 0, 0, 64)  # read EEPROM content
print(binascii.hexlify(ret))
```

Le contenu de l'EEPROM est maintenant écrasé, et démarrer le FX2LP sans jumper (donc identification correcte de l'adresse I^2C) permet de reflasher la mémoire non-volatile.

Denis Bodor fait remarquer qu'une fonctionnalité identique est proposée par `http://www.triplespark.net/elec/periph/USB-FX2/eeprom/` qui contient `erase_eeprom` aux fonctionnalités identiques, quitte éventuellement à remplacer l'adresse I^2C 0xA2 ("*assumes that A0 is tied to positive supply and A1,A2 of the EEPROM are tied to ground* ") par 0xA0 selon que le jumper ne soit pas en place ou le soit. Ce programme efface tout le contenu de l'EEPROM au lieu de juste effacer le premier octet, seule condition nécessaire à refaire fonctionner `fxload` pour reflasher l'EEPROM déjà flashée.

1.5 Communication SPI commandée par USB

Afin d'apprendre de façon incrémentale à aborder le bus USB du point de vue de l'hôte (le PC sous GNU/Linux) et du device (le FX2LP), nous allons dans un premier temps nous appuyer sur le *firmware* pré-compilé de PocketSDR disponible à `https://github.com/tomojitakasu/PocketSDR/blob/master/FE_2CH/FW/v2.1/pocket_fw.hex` et flashé par `fxload` en EEPROM lorsque le microcontrôleur a été démarré avec le jumper associé à l'EEPROM en place, mais retiré au moment de la programmation (voir section 1.4). Une fois le jumper retiré, nous savons que le logiciel de PocketSDR est celui en cours d'exécution si `lsusb` indique au redémarrage (mise hors tension puis sous tension) :

```
Bus 001 Device 003: ID 04b4:1004 Cypress Semiconductor Corp. There
```

avec maintenant un couple `VID:PID` qui vaut `04b4:1004`, le nouveau PID étant défini dans le firmware. Il faut maintenant comprendre comment PocketSDR, dont le code source est disponible, enclenche des communications entre le FX2LP et le MAX2771 selon un protocole que la documentation technique de ce dernier qualifie abusivement de compatible SPI, abus de langage puisque MOSI et MISO sont confondus en un unique “SDATA” qui change d’impédance selon que la transaction se fasse en écriture ou en lecture (exactement la raison pour laquelle nous détestons I2C dont la direction de la transaction ne peut être déduite de la trace acquise sur oscilloscope ou analyseur logique).

Nous apprenons dans l’entête de `PocketSDR/FE_2CH/FW/v2.1/pocket_fw.c` qu’un certain nombre de *Vendor Requests* permettent, au même titre que les `ioctl()` dans un module noyau Linux, d’attribuer des opérations à des codes de commande arbitrairement sélectionnés. Dans le cas particulier de PocketSDR, la commande 0x40 renverra la version et statut du firmware, 0x41 lit un registre du MAX2771 et 0x42 y écrit (Fig. 4).

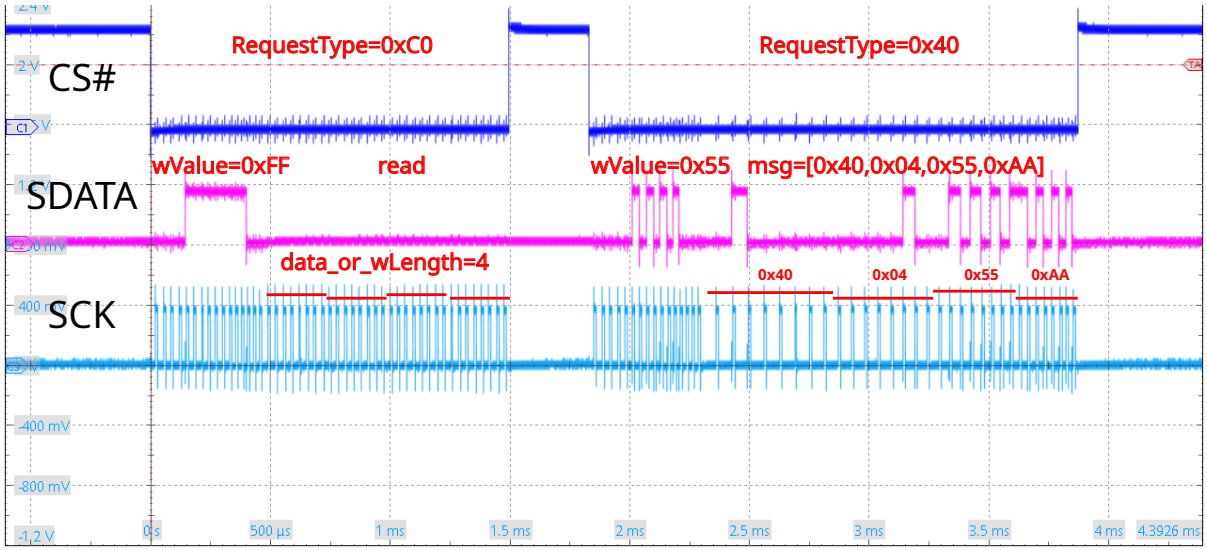


FIGURE 4 – Signaux produits par l’implémentation logicielle de SPI (bien que regroupant sur un même fil les signaux MOSI et MISO) en lecture et en écriture selon le code de la *Vendor Request* transmis. De haut-en bas la sélection du composant CS#, le bus de données SDATA et le bus d’horloge SCK.

Il faut donc apprendre à envoyer des *Vendor Requests* depuis GNU/Linux afin de se familiariser avec ces échanges implémentés dans le firmware précompilé de PocketSDR, en vue de s’assurer que notre implémentation du firmware compatible `sdcc` atteindra les mêmes objectifs, à savoir les transactions sur bus SPI. Le résultat est surprenamment facile à atteindre au moyen de la bibliothèque USB de Python3 une fois qu’on a compris le sens des arguments :

```
import usb
import binascii

VID = 0x04B4
PID = 0x8613
dev = usb.core.find(idVendor = VID, idProduct = PID)
msg=bytearray([0x40,0x04,0x55,0xaa]);
# dev.ctrl_transfer(bmRequestType, bRequest, wValue, wIndex, data)
ret=dev.ctrl_transfer(bmRequestType=0xC0 ,bRequest=0x41, wValue=0xFF, wIndex=0, data_or_wLength=4)
# 0x155 to activate second CS#, 0x55 to activate first CS#
dev.ctrl_transfer(0x40 , 0x42, 0x55, 0, msg) # write on SPI bus reg @ 0x55 int msg
# FX2LP firmware status
ret=dev.ctrl_transfer(0xC0,0x40, 0, 0, 10)  # read VR_STAT: returns 6 bytes (EP0BCL=6;)
print(binascii.hexlify(ret))                # b'105dc0010100'
```

Nous apprenons à `https://www.beyondlogic.org/usbnutshell/usb6.shtml` que les arguments `wValue` et `wIndex` de la méthode `ctrl_transfer()` dépendent de la nature des transactions, mais pour faire simple le premier argument `bmRequestType` vaut 0x40 pour une écriture et 0xC0 pour une lecture,

suit `bRequest` le code de la *Vendor Request* en adéquation avec le firmware de PocketSDR, suit un mot `wValue` qui sera décodé par le 8051 du FX2LP pour identifier le registre du MAX2771 à atteindre (octet de poids faible) et, dans le cas du circuit "officiel" de la PocketSDR muni de deux MAX2771, lequel des deux composants activer par son Chip Select adéquat (octet de poids fort). Finalement, `wIndex` le numéro de l'interface est toujours nul, et si la transaction est en écriture le dernier argument est un tableau d'octets transmis sur bus SPI, sinon il s'agit du nombre d'octets à lire. Si nous émettons la commande 0x40 (lecture du statut) en mode lecture (`bmRequestType=0xC0`) avec le code Python proposé auparavant, alors nous recevons `b'105dc0010100'` qui correspond au code source `PocketSDR/FE_2CH/FW/v2.1/pocket_fw.c` à savoir au début la version du firmware 0x10, suivi de deux octets contenant la fréquence de l'oscillateur en kHz (24000 dans notre cas, donc 0x5DC0 en hexadécimal), suivi du divers statuts sans importance pour le moment. La transaction est donc convenablement effectuée pour lire une valeur du microcontrôleur par USB.

Maintenant, pour déclencher des transactions sur USB, nous envoyons l'ordre (`bRequest`) 0x41 (lecture) ou 0x42 (écriture) suivi du numéro du registre du MAX2771 dans `wValue` et soit la séquence d'octets à écrire, soit le nombre d'octets à lire. Nous constatons à l'oscilloscope que les signaux SCLK, SDATA, et CS# se comportent comme prévu dans une liaison SPI (Fig. 4). Bien qu'il s'agisse d'une implémentation logicielle (*bitbang*) de SPI dans le 8051, la période et le rapport cyclique de SCLK, que l'on voit bien ne pas être constants sur la courbe du bas de Fig. 4, n'ont aucune importance puisqu'il s'agit d'un protocole synchrone dont seuls les fronts importent. Lorsque nous lisons un résultat, nous constatons dans le code source de `PocketSDR/FE_2CH/FW/v2.1/pocket_fw.c` que le cas `VR_REG_READ` se conclut pas `EP0BCL=4;` donc le renvoi de 4 octets ou les 32 bits contenus dans chaque registre du MAX2771. Nous mettons en pratique par

```
#!/usr/bin/env python3
import usb
import binascii
VID = 0x04B4
PID = 0x1004
dev = usb.core.find(idVendor = VID, idProduct = PID)
#PLL Fractional Division Ratio register 0x05
#default value 0x08000070
#                       ^^ reserved
#dec2hex(586329)
#ans =           08F259
msg=bytearray([0x08,0xF2,0x59,0x70]);
ret=dev.ctrl_transfer(0xC0 ,0x41, 0x05, wIndex=0, data_or_wLength=4)
print(binascii.hexlify(ret))
dev.ctrl_transfer(0x40 , 0x42, 0x05, 0, msg) # write on SPI bus reg @ 0x5
ret=dev.ctrl_transfer(0xC0 ,0x41, 0x05, wIndex=0, data_or_wLength=4)
print(binascii.hexlify(ret))
```

qui accède au registre 5 du MAX2771 ou la partie fractionnaire de la division de l'oscillateur dans la boucle à verrouillage de phase (PLL), en écriture et en lecture, pour vérifier que l'information a bien été stockée dans le registre.

1.6 Configuration de *Vendor Requests* par `sdcc`

Nous sommes convaincus de savoir échanger des messages *Vendor Request* depuis GNU/Linux pour déclencher une action sur le FX2LP : ceci a été prouvé avec le firmware précompilé au moyen du compilateur propriétaire Keil. Nous devons maintenant reproduire ce comportement avec `sdcc`. Pour ce faire, nous partons de l'exemple `bulkloop/` de `fx2lib`. Nous y identifions la fonction `BOOL handle_vendorcommand(BYTE cmd) {...}` qui semble correspondre à nos besoins, et en particulier avec la gestion d'une commande nommée `VC_EPSTAT` identifée par le code 0xB1 : nous avons déjà mentionné que tout comme `ioctl()` dans le noyau Linux, ces codes attribués aux commandes sont arbitraires et doivent être cohérentes avec le code émis par l'application. Nous copions donc la séquence de *Vendor Requests* du firmware de PocketSDR depuis la fonction `BOOL handle_req(void) {...}` de `PocketSDR/FE_2CH/FW/v2.1/pocket_fw.c`, en remplaçant les conditions imbriquées par une séquence `switch ... case` un peu plus lisible. Par exemple la commande `VR_STAT` de code `0x40` remplit les 6 premiers octets du tampon `EP0BUF` avec des valeurs permettant d'identifier la version du firmware ou la fréquence du quartz cadençant le MAX2771, et renvoie (`EP0BCH = 0; EP0BCL = 6;`) ce tampon à la fonction appelante. Côté Python, le pendant de cette

requête est celle que nous avions vue auparavant, mais que nous comprenons n'attendre que 6 octets en retour

```
import usb
import binascii
VID = 0x04B4
PID = 0x8613
dev = usb.core.find(idVendor = VID, idProduct = PID)
ret=dev.ctrl_transfer(0xC0,0x40, 0, 0, 6)  # read VR_STAT
print(binascii.hexlify(ret))
```

donc une requête en lecture (0xC0) de la *Vendor Request* 0x40 (`VR_STAT`) et la réponse est `b'105dc0010100'` pour indiquer 0x10 la version du *firmware*, correspondant aux `EP0BUF[0] = VER_FW;` dans le code source du *firmware* PocketSDR, puisque `#define VER_FW 0x10`, suivi de 0x5DC0 puisque dans PocketSDR

```
EP0BUF[1]=MSB(F_TCXO); EP0BUF[2] = LSB(F_TCXO);
```

avec `#define F_TCXO 24000`. Suivent deux états de broches en entrée que nous ne contrôlons pas pour le moment. Le code source complet pour `sdcc` est consultable à `https://github.com/jmfriedt/max2771_fx2lp/blob/main/FX2LP/python_access_USB` qui bénéficie en partie de ce que Keil et FX2Lib pour SDCC s'appuient tous deux sur des fichiers de constantes partageant la même nomenclature et donc facilement interchangeables.

Nous voici donc capables de recevoir des messages depuis le FX2LP grâce aux *Vendor Requests*, et nous continuous le plagiat du code de PocketSDR en copiant depuis `PocketSDR/FE_2CH/FW/v2.1/pocket_fw.c` l'implémentation logicielle (*bitbang*) du bus SPI puisque la syntaxe de `digitalRead()`, `digitalWrite()` qui accèdent aux broches, mais aussi `write_sclk()`, `write_sdata()` et `write_head()`, sont directement compatible avec SDCC, la bibliothèque FX2Lib ayant le bon goût d'utiliser les mêmes constantes que le compilateur Keil tel que nous le constatons en comparant `cypress/dscr.a51` de `PocketSDR/FE_2CH/FW/v2.1` pour Keil et `fx2lib/fw/dscr.a51` pour SDCC. Seul piège : PocketSDR définit une fonction `delay()` comme boucle de `cyc` itérations, alors que FX2Lib propose une fonction du même nom mais dont le prototype est `void delay(WORD millis);` et qui prend donc un argument en millisecondes (et non en cycles d'horloge).

On voit bien que la fonction FX2Lib sera considérablement plus lente que celle de PocketSDR, et en effet nos observations sur le bus SPI cadencé par logiciel présentaient des variations très lentes de l'horloge et du bus de données. Une nouvelle fonction de délai avec un nom différent et effectuant bien un décompte rapide permet de retrouver presque à l'identique le comportement de PocketSDR grâce au programme compilé par SDCC proposé dans `https://github.com/jmfriedt/max2771_fx2lp/tree/main/FX2LP/python_access_USB`.

Nous en sommes au point où nous sommes capables de communiquer des ordres depuis le PC au FX2LP par un programme Python, que le 8051 interprète ces commandes et produise le motif adéquat sur le bus SPI, donc programme les registres du MAX2771 et en relise le contenu. Il ne reste maintenant "plus" qu'à capturer les octets placés sur le bus parallèle par le MAX2771 et cadencés par le signal IFCLK en vue de remplir la mémoire FIFO qui se vide périodiquement sur le bus USB dans une transaction bulk. Quelques points de "détail" restent cependant à régler avant d'en arriver là.

1.7 Communication des données acquises du MAX2771 par FX2LP en USB bulk

Toujours en partant de l'exemple `bulkloop/` de FX2Lib, la dernière étape pour finaliser le portage du *firmware* PocketSDR vers `sdcc` en maîtrisant les transferts en mode Bulk sur USB est moins pénible qu'il n'y parait puisque de nouveau FX2Lib exploite la même nomenclature que le compilateur Keil et donc il suffit de reprendre la fonction de configuration d'USB de PocketSDR nommée `void setup(void) {...}` qui configure tous les endpoints ainsi que `static void start_bulk(void) {...}` appelée en fin de `main()` (ainsi que `stop_bulk(void) {...}` qui peut être appelée par une *Vendor Request*) pour voir toutes les fonctions de communication. Nous éliminons dans un premier temps toutes les fonctions liées aux accès à la mémoire non-volatile sur bus I^2C : le pendant des fonctions Keil du type `EZUSB_WriteI2C()` et `EZUSB_ReadI2C()` nécessaires à `write_eeprom()` et `read_eeprom()` de PocketSDR existent dans FX2Lib sous forme de `eeprom_read()` et `eeprom_write()` mais ne seront pas nécessaires pour les premiers tests.

Parmi les autres subtilités de syntaxe, l'espace d'adressage en 16 bits du 8051, `xdata` de Keil, devient `__xdata` chez SDCC, et les mnémoniques assembleur inclus dans le code C sont préfixés de `__asm` suivi de l'instruction, par exemple `nop`, et se concluent avec `__endasm`. Évidemment les vecteurs d'interruptions s'appellent différemment, par exemple `void ISR_Highspeed(void) interrupt 0 {...}` de Keil devient chez SDCC `void hispeed_isr(void) __interrupt (HISPEED_ISR) {...}` mais comme l'exemple `bulkloop/` est fonctionnel, ce dernier point nous concerne peu.

À l'issue de cet exemple, nous sommes capables de compiler un firmware proposant toutes les fonctionnalités d'origine, cette fois compilé par `sdcc`, incluant les *Vendor Requests* pour communiquer les configurations sur bus SPI ainsi que le transfert de données acquises du MAX2771 en transactions USB Bulk.

1.8 Validation en finalisant (presque) le câblage du MAX2771 au FX2LP

Nous commençons à comprendre comment tout cela fonctionne, donc nous désirons maintenant effectuer la vraie communication entre le MAX2771 et le FX2LP en vue de récupérer des données. Pour ce faire, nous câblons la carte AliExpress EZ-USB vers la carte d'évaluation du MAX2771 au moyen de 5 fils reliant I0, I1, Q0 et Q1 à PB0 à PB3 respectivement et surtout DCLK (CLKOUT du MAX2771) à IFCLK/PE0 du FX2LP, ainsi que les broches de communication SPI horloge, données, et activation (CS#) à PD2, PD3 et PD0 respectivement ... et rien ne fonctionne. En effet, il faut aussi câbler les deux signaux qui définissent la direction de remplissage de la FIFO du FX2LP, soit en lecture soit en écriture depuis le PC, dans notre cas en suivant le schéma de `https://github.com/tomojitakasu/PocketSDR/tree/master/FE_2CH/HW/v2.3` en connectant `RDY0/SLRD` vers l'alimentation et `RDY1/SLWR` vers la masse. Nous relançons une mesure... et toujours rien. Impossible de recevoir la moindre trame USB depuis le FX2LP sur le PC.

Ici encore, Tomoji Takasu a la réponse puisqu'il a subi les mêmes déboires, qu'il a documentés à `https://blog.goo.ne.jp/osqzss/e/d86df04de96123fd5c73bbb6db6e8bc5` (à passer dans Google Translate pour les moins souples en japonais) : certaines cartes chinoises de développement du FX2LP ont inversé la sérigraphie de RDY0 et RDY1 qui définissent, en les polarisant à la masse ou à la tension d'alimentation, la direction de la communication. Notre carte (Fig. 5) est sujette à cette erreur, et inverser les signaux d'alimentation et masse entre RDY0 et RDY1 pour retrouver la configuration du schéma de la PocketSDR finit par permettre la communication. Baptiste Maréchal (SpacePNT, Neuchatel) fait remarquer avec amusement que diverses illustrations de cette carte sur un même site Amazon sont incohérentes dans leur sérigraphie entre les diverses photographies qui font la promotion du produit, mais il fallait le savoir pour identifier l'erreur !

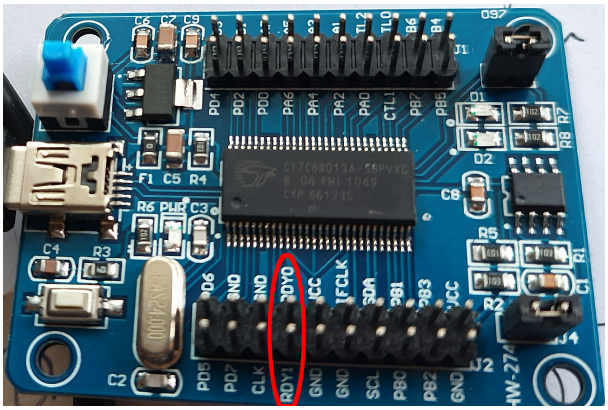


FIGURE 5 – La carte équipée du FX2LP prête à être utilisée, et pourtant impossible de la faire communiquer en USB Bulk malgré un code compilé par Tomoji Takasu au moyen du compilateur propriétaire Keil auquel nous ne pouvons que faire confiance. Le problème vient d'une erreur de sérigraphie sur le circuit chinois qui induisait une configuration erronée de la direction de communication, tel que nous en a informé l'auteur japonais de PocketSDR.

Une fois cette erreur corrigée, nous achevons de vérifier les fonctionnalités de communication en profitant de `PocketSDR/app/pocket_dump/pocket_dump` pour recevoir des données. Ce faisant, nous n'aurons

pas (encore) appris à recevoir nous mêmes des octets depuis la FIFO vers le PC, mais gardons cette compréhension pour la prochaine section. Par ailleurs, un point un peu "surprenant" est que, quelle que soit la configuration que nous proposons au MAX2771 par `PocketSDR/app/pocket_conf/pocket_conf`, le débit de communication reste toujours le même puisque `pocket_dump` indique

```
$ sudo app/pocket_dump/pocket_dump
  TIME(s)    T   CH1(Bytes)   T   CH2(Bytes)   RATE(Ks/s)
      5.3    I    126943232   I    126943232      23996.8
```

soit toujours une communication d'un réel (sans partie imaginaire, indiqué par `I`) au débit de 24 Méchantillons/s (que nous noterons désormais MS/s pour *Msamples/s*).

Un petit détour par la compréhension détaillée du transfert en mode Bulk côté *firmware* et lecture côté PC sous GNU/Linux nous permettrait presque de cacher un gros dysfonctionnement qui subsiste, mais que nous allons dévoiler plus tard et s'avèrera bien plus intéressant qu'il n'y paraît.

1.9 Communication d'un motif connu en USB Bulk

Avant de vouloir lire un flux rapide de données, nous désirons déjà valider la communication microcontrôleur-PC sur bus USB par interface Bulk en envoyant un motif connu. Un auteur rencontré sur `github`, Siddharth Deore [12], a bien voulu fournir ses exemples, même s'il fallut une fois de plus se battre avec les évolutions de FX2Lib pour faire fonctionner ces programmes. Cependant, cet exemple simple de communication bulk depuis le FX2LP, disponible à `https://github.com/jmfriedt/max2771_fx2lp/tree/main/FX2LP/bulk_read_example`, est aussi l'occasion de voir comment gérer ce flux de données depuis un programme C exploitant `libusb`

Du point du vue du microcontrôleur, l'exemple déclare l'endpoint 6 en interface bulk et initialise les propriétés de la FIFO, que nous allons remplir manuellement dans cet exemple :

```
#define ALLOCATE_EXTERN
#include <fx2regs.h>
#include <fx2macros.h>
#include <delay.h>    // needed for SYNCDELAY4
#include <fx2ints.h>
#include <autovector.h>

static void initialize(void)
{ SETCPUFREQ(CLK_48M); // set the CPU clock to 48MHz
  SETIF48MHZ();        // set the slave FIFO interface to 48MHz IFCONFIG |= 0x40;*/
  // Set DYN_OUT and ENH_PKT bits, as recommended by the TRM.
  REVCTL = bmNOAUTOARM | bmSKIPCOMMIT; // REVCTL = 0x03;
  SYNCDELAY4;

  /* out endpoints do not come up armed */
  /* set NAKALL bit to NAK all transfers from host */
  EP6CFG = 0xe2; SYNCDELAY4; // 1110 0010 (bulk IN, 512 bytes, double-buffered)
  FIFORESET = 0x80; SYNCDELAY4; // NAK all requests from host.
  FIFORESET = 0x82; SYNCDELAY4; // Reset individual EP (2,4,6,8)
  FIFORESET = 0x84; SYNCDELAY4;
  FIFORESET = 0x86; SYNCDELAY4;
  FIFORESET = 0x88; SYNCDELAY4;
  FIFORESET = 0x00; SYNCDELAY4; // Resume normal operation.
}
```

La FIFO sera remplie du contenu lu sur les ports B et D : il peut donc être amusant de modifier périodiquement le statut d'un bit de ces ports, et comme il s'avère dans l'exemple précédent que la LED était connectée sur le port D7, nous déclenchons une interruption timer périodique qui change l'état de la broche PD7 qui doit aussi affecter l'affichage des données lues sur ce port et transmises par USB :

```
volatile __bit led_flag;
volatile char t0_counter;

void timer0_isr(void) __interrupt (TF0_ISR)
{ t0_counter++;
```

```
  if (t0_counter == 20)
    { led_flag = 1-led_flag;
      if (led_flag) { PD7 = 0; }
      else { PD7 = 1; }
      t0_counter = 0;
    }
}
```

Enfin, la fonction principale se contente d'initialiser les périphiériques, et de boucler indéfiniment en testant si le tampon des FIFOs est vide, et le cas échéant remplir avec le contenu des ports B et D, ou en commentaire pour se convaincre de la validité de la démarche, avec une séquence d'octets connue :

```
void main(void)
{ int i;
  initialize();
  led_flag=0;
  t0_counter=0;  // init timer0 vvv
  TMOD = 0x11;
  EA=1;          // enable interrupts
  ENABLE_TIMER0();
  TR0=1;         // start timer0

  OEB = 0x0; SYNCDELAY4; // set PORT-B to input
  OED = 0x0; SYNCDELAY4; // set PORT-D to input
  OED = (1 << 7);        // PD7 as output for blinking the LED !
  while (1)
    {if (!(EP2468STAT & bmEP6FULL)) // Wait for EP6 buffer to become non-full
       {for (i=0; i<512; i+=2)
          {EP6FIFOBUF[i] = IOB;     // fill buffer with port b and d
           EP6FIFOBUF[i + 1] = IOD;
           // EP6FIFOBUF[i] = 0x55; // testing with fixed values
           // EP6FIFOBUF[i + 1] = 0xAA;
            }
        // Arm the endpoint. Set BCH *before* BCL because BCL access
        // actually arms the endpoint.
        EP6BCH = 0x02; // commit 512 bytes
        EP6BCL = 0x00;
       }
    }
}
```

Ce programme est compilé et transféré en RAM du microcontrôleur du FX2LP pour exécution depuis le répertoire `FX2LP/bulk_read_example` de notre dépôt par
`make && sudo cycfx2prog prg:build/fifo_ep6.ihx run`

Du point de vue de l'hôte sous GNU/Linux, la récupération des trames s'obtient en exploitant `libusb` (paquet `libusb-1.0-0-dev` sous Debian GNU/Linux) dont la configuration pour l'emplacement des fichiers d'entête et des bibliothèques peut s'obtenir avec `pkg-config`). Une fois la séquence d'incantations connue, le programme est simple et compact avec

```
#include <stdio.h>
#include <stdlib.h>
#include <libusb-1.0/libusb.h> // ou <libusb.h> avec pkg-config --cflags libusb-1.0
#define N 512
#define tout_ms 1000
#define vid 0x04b4
#define pid 0x8613 // 0x1004;

int main()
{int i,j,xferred,res;
 unsigned char buf[N];
 libusb_context *ctx = NULL;   // initialize libusb
 libusb_init(&ctx);
 libusb_device_handle *hndl=libusb_open_device_with_vid_pid(ctx, vid, pid);
```

```
 libusb_claim_interface(hndl, 0);
 libusb_set_interface_alt_setting(hndl, 0, 1);
 while (1) {
   libusb_bulk_transfer(hndl, LIBUSB_ENDPOINT_IN | 6, buf, N, &xferred, tout_ms);
   for (i=0; i<N; i+= 2)  // affichage du contenu de la FIFO en binaire
     {for (j=0;j<8;j++) printf("%d",((buf[i]>>(7-j))&1));
      printf("␣");
      for (j=0;j<8;j++) printf("%d",((buf[i+1]>>(7-j))&1));
      printf("\n");
     }
 libusb_release_interface(hndl, 0);
 libusb_close(hndl);
 libusb_exit(ctx);
}
```

dont les noms de fonctions semblent suffisamment explicites pour suivre naturellement la séquence de communications. Ce programme se compile avec `gcc` en pensant à se lier à `libusb` en complétant la commande de compilation avec `-lusb-1.0`.

Les mêmes fonctionnalités sont implémentée en Python3 avec

```
import usb.core
import usb.util
import binascii

N = 512
tout_ms = 1000
vid = 0x04b4
pid = 0x8613  # 0x1004

dev = usb.core.find(idVendor=vid, idProduct=pid)
dev.set_configuration()
usb.util.claim_interface(dev,0)
cfg = dev.get_active_configuration()
interface_number = cfg[(0, 0)].bInterfaceNumber
data = dev.read(0x86, N, tout_ms)  # 0x80 | 6
print(data)
```

Une fois le firmware flashé sur le FX2LP, l'exécution du programme Python sous GNU/Linux indique

```
$ sudo ./bulk_read.py
array('B', [255, 127, 255, 127, 255, 127, 255, 127, 255, 127, 255, 127, 255 ...
```

alternant l'état des ports B et D (sur 8 bits), tandis que l'exécutable issu du programme C affiche

```
$ sudo ./fx2lp_ep6_in_fifo
11111111 01111111
11111111 01111111
11111111 11111111
...
```

avec le 7ème bit qui change d'état selon que PD7 soit à l'état haut ou bas sous contrôle de l'interruption *timer*.

Noter qu'en cas d'échec des transactions, il peut être judicieux de vérifier si le module noyau `usbtest` n'a pas été automatiquement chargé, et le retirer le cas échéant tel que décrit à `http://www.triplespark.net/elec/periph/USB-FX2/eeprom/` par `sudo rmmod usbtest`.

2 Communication des données acquises du MAX2771 par FX2LP (pour de vrai)

Nous avons identifié les différences de syntaxe entre compilateurs Keil et SDCC, identifié les erreurs de sérigraphie sur le circuit imprimé, tout devrait donc fonctionner. Nous avions conclu la section 1.8 en insinuant qu'un dysfonctionnement subsistait, un point de détail sans grand importance. En effet,

lorsque nous configurons la carte avec notre firmware et lançons une acquisition par
`PocketSDR/app/pocket_dump/pocket_dump`
la communication se fait toujours à 24 MS/s, même si nous tentons de configurer le convertisseur analogique numérique pour un débit différent, par exemple passer à 12 MS/s en modifiant uniquement le registre `REFDIV` de 3 (×1) à 2 (/2). Pire, en modifiant cet unique registre avec une valeur qui semble cohérente, la communication est perdue. Surement un registre mal configuré quelquepart... mais le problème s'est avéré à peine plus ardu. Il nous faut résoudre ce dernier problème pour affirmer que nous pouvons compiler le firmware de la PocketSDR fonctionnel au moyen de SDCC.

Nous connaissions les différents boutismes (*endianness*) des diverses architectures des processeurs – avec Intel en *little endian* qui place l'octet de poids faible d'un mot codé sur plusieurs octets à l'adresse la plus faible en mémoire, et Motorola qui en *big endian* place l'octet de poids fort à l'adresse la plus faible – et nous connaissions l'absence de définition de la nature signée ou non des entiers selon les déclinaisons des compilateurs C, mais nous découvrons maintenant que divers compilateurs pour une même architecture peuvent choisir des *endianness* différentes ! Tant que nous programmions le 8051 en assembleur, la question de représenter des grandeurs sur plusieurs octets ne se posait pas, puisque les registres du 8051 ne contiennent qu'un seul octet. Mais comme nous devons passer à la programmation du microcontrôleur en C au lieu de l'assembleur, nous devons être capables de représenter des nombres sur 2 octets (`short`) voir 4 octets (`long`) même pour une cible qui ne comporte que des registres codant des valeurs sur 8 bits. Et là les choses se compliquent.

Des diverses conventions du `char` en fonction des déclinaisons de `gcc`

On pourrait croire que définir `char c;` en langage C est une instruction déterministe et reproductible. Il n'en est rien en l'absence du préfixe explicite `signed` ou `unsigned`, ou de l'utilisation de l'extention `stdint.h` qui permet de définir `uint8_t` pour un `unsigned char` et `int8_t` pour un `signed char` (cf par exemple `/usr/msp430/include/stdint.h` pour le MSP430). En effet sans ces précautions, le programme *a priori* trivial

```
int main() {volatile char i=240;}
```

se compile avec l'option `-pedantic` pour indiquer les dépassements de capacité, vers un microcontrôleur AVR, un PC x86 ou un ARM sans système d'exploitation respectivement en

```
$ avr-gcc -c -pedantic demo.c
demo.c: In function ‘main’:
demo.c:2:18: warning: overflow in implicit constant conversion [-Woverflow]
 {volatile char i=240;
                  ^~~
$ gcc -c -pedantic demo.c
demo.c: In function ‘main’:
demo.c:2:18: warning: overflow in conversion from ‘int’ to ‘char’ changes value
from ‘240’ to ‘-16’ [-Woverflow]
    2 | {volatile char i=240;
      |                  ^~~
$ arm-none-eabi-gcc -c -pedantic demo.c
```

donc les deux premiers compilateurs se plaignent que 240 dépasse la capacité de stockage d'un `signed char` (donc `signed` a été ajouté implicitement), mais `arm-none-eabi-` ne s'en plaint pas, laissant présager que là le `char` est implicitement `unsigned`, un gros soucis si une boucle teste le passage sous 0 pour cesser ses itérations. On peut compléter cet exemple par la recherche de la valeur minimum de `char` par le préprocesseur (`gcc -E`) :

```
$ gcc -E -dM demo.c  | grep CHAR_MIN
#define SCHAR_MIN (-SCHAR_MAX - 1)
#define CHAR_MIN SCHAR_MIN
$ avr-gcc -E -dM demo.c  | grep CHAR_MIN
#define SCHAR_MIN (-SCHAR_MAX - 1)
#define CHAR_MIN SCHAR_MIN
```

```
$ arm-none-eabi-gcc -E -dM demo.c  | grep CHAR_MIN
#define CHAR_MIN 0
```

démontrant donc que pour `gcc` (Intel) et `avr-gcc`, un `char` est signé avec une valeur minimale de -128, alors que pour `arm-none-eabi-gcc` la valeur minimum d'un `char` est 0 donc non-signé. Ce dernier cas est peut être lié à la norme imposée par ARM dans `https://developer.arm.com/documentation/dui0472/m/C-and-C---Implementation-Details/Basic-data-types-in-ARM-C-and-C--` qui indique

`char 8 1 (byte-aligned) 0 to 255 (unsigned) by default.`

Sans vouloir donner de grain à moudre aux détracteurs du C, ce changement de signe du `char` entre déclinaisons du même compilateur nous a causé bien des soucis, en particulier dans les boucles du type `for (k=N;k>=0;k--)` qui deviennent infinies avec un `unsigned char` (toujours positif).

En effet, [13] enseigne que bien que ciblant une même architecture 8 bits, Keil est un compilateur *big endian* et SDCC est un compilateur *little endian*. Tant que les calculs sont gérés par un ordinateur exécutant un code compilé par un seul compilateur, l'organisation des données en mémoire est cohérente et il n'y a pas de problème. Cependant en convertissant un code écrit pour Keil vers SDCC qui effectue des *cast* de tableaux d'octets (`char*`) vers un entier codé sur plus de 8 bits (`short` ou `int`, ou pour les utilisateurs de `stdint.h`, `int16_t` et `int32_t`), il faut penser à intervertir les octets qui seront sinon transférés dans le mauvais ordre vers le MAX2771. Ainsi, les lignes pour compilateur Keil

```
  else if (SETUPDAT[1] == VR_REG_READ) {
    *(uint32_t *)EP0BUF = read_reg(SETUPDAT[3], SETUPDAT[2]);
    EP0BCH = 0;
    EP0BCL = 4;
  }
  else if (SETUPDAT[1] == VR_REG_WRITE) {
    EP0BCH = EP0BCL = 0;
    while (EP0CS & bmEPBUSY) ;
    write_reg(SETUPDAT[3], SETUPDAT[2], *(uint32_t *)EP0BUF);
  }
```

deviennent pour SDCC (on en a profité pour remplacer les `if ... else` imbriqués par un `switch ... case`) :

```
  case VR_REG_READ:
  { val32=read_reg(SETUPDAT[3], SETUPDAT[2]);
    *(uint32_t *)EP0BUF = bswap32(val32);
    EP0BCH = 0;
    EP0BCL = 4;
    return TRUE;break;
  }
  case VR_REG_WRITE:
  { EP0BCH = EP0BCL = 0;
    while (EP0CS & bmEPBUSY) ;
    val32=*(uint32_t *)EP0BUF;
    val32=bswap32(val32);
    write_reg(SETUPDAT[3], SETUPDAT[2], val32);
    return TRUE;break;
  }
```

avec

```
#define bswap32(x) (((x) >> 24) | (((x) & 0x00FF0000) >> 8) \
                  | (((x) & 0x0000FF00) << 8) | ((x) << 24))
```

issu de `https://www.keil.com/dd/docs/c51/silabs/shared/si8051base/endian.h` qui se charge d'intervertir les octets (le fameux `htonl()` de `/usr/include/netinet/in.h` sous GNU/Linux). Il s'agit heureusement des deux seuls *cast* de tableaux d'octets vers un entier de plus de 8 bits qu'il faut corriger

de la sorte, tel qu'on s'en convainc en recherchant `grep "int32_t\ \*" complete_fw.c` dans le code source du firmware.

Une fois ces dernières erreurs corrigées, nous avons un *firmware* complet compatible PocketSDR mais compilable par `sdcc` sans dépendre d'un compilateur propriétaire, que Tomoji Takasu nous informe par ailleurs être limité à des exécutables de 4 KB pour sa version gratuite.

Nous avons mentionné en introduction que la carte d'évaluation du MAX2771 est munie d'un oscillateur à 16,368 MHz et que PocketSDR est configurée pour un oscillateur à 24 MHz. Cette information est renseignée dans le firmware, à `https://github.com/tomojitakasu/PocketSDR/blob/master/FE_2CH/FW/v2.1/pocket_fw.c#L27` pour la version originale pour Keil ou à `https://github.com/jmfriedt/max2771_fx2lp/blob/main/FX2LP/complete_fw/complete_fw.c#L28` pour notre version pour SDCC : cette constante `F_TCXO` sera remplacée par 16368 pour que le firmware ainsi recompilé fonctionne directement avec la carte d'évaluation munie de son oscillateur d'origine.

Ainsi, nous pourrons profiter de tous les exécutables fournis par PocketSDR pour tester le bon fonctionnement de la carte d'évaluation MAX2771 couplée au FX2LP. Pour ce faire, nous allons explorer quelques signaux satellitaires transmis dans la moitié supérieure de la bande L, autour de 1500–1600 MHz.

3 Résultats : Iridium et GPS

Nous allons acquérir, au rythme de quelques dizaines de mégaoctets par seconde, des gigaoctets de données dont nous voulons valider la pertinence, alors que nous ne sommes même pas certains du bon fonctionnement de la plateforme matérielle et de sa configuration logicielle. Il nous faut donc évaluer la difficulté à détecter le signal que nous espérons observer par rapport au bruit avant de se lancer dans des mesures.

En effet trois paramètres doivent être configurés pour déterminer les caractéristiques d'une acquisition :

- la fréquence centrale de l'acquisition, par programmation du facteur de multiplication de l'oscillateur de référence (24 MHz) par PLL pour asservir l'oscillateur commandé en tension (VCO). Cette porteuse n'a aucune importance dans l'analyse car elle est éliminée lors du pré-traitement analogique par mélange avec le signal à acquérir.
- la fréquence d'échantillonnage f_s, définie par le rythme auquel le convertisseur analogique-numérique (ADC) acquiert les données,
- inclure une fréquence intermédiaire ou travailler directement en bande de base. La conséquence de ce choix est la production de signaux réels uniquement dans le premier cas, et complexes dans le second. En effet dans le premier cas, un unique mélangeur amène un signal réel proche de la bande de base mais écarté de la fréquence intermédiaire, et c'est la transpoition numérique (par traitement logiciel après conversion analogique-numérique) depuis la fréquence intermédiaire f_{IF} vers la bande de base en multipliant par $\exp(j2\pi f_{IF}t)$ avec $t = [0 : N-1]/f_s$ le temps discret le long des N échantillons acquis à la fréquence d'échantillonnage f_s, qui produira les complexes I et Q attendus. Le bénéfice du passage par la fréquence intermédiaire est de rejeter les bruits, notamment de l'électronique numérique proches de 0 Hz, hors de la bande d'acquisition.

Les deux objectifs que nous nous fixons sont dans un premier temps d'observer le spectre des signaux des satellites Iridium, centré sur 1622 MHz mais suffisamment puissant pour être bien visibles, et dans un second temps GPS centré sur 1575,42 MHz, l'objectif ultime de ces développements mais sous le bruit thermique donc difficile à déceler quand le bon fonctionnement du système électronique est en cours d'évaluation. En effet, rappelons que les 50 W émis par un satellite GPS avec ses antennes de 13 dBi de gain n'arrivent au sol après avoir traversé les 20000 km qui le séparent du récepteur qu'avec une puissance (équation de Friis de la conservation d'énergie sur la sphère sur laquelle se distribue la puissance émise) de

$$10\log_{10}(\underbrace{50\times1000}_{\text{W}\to\text{mW}}) - 20\log_{10}(\underbrace{1575,42\cdot10^6}_{\text{porteuse}}) - 20\log_{10}(\underbrace{20000\times1000}_{\text{distance km}\to\text{m}}) + \underbrace{147,55}_{20\log_{10}(\frac{c}{4\pi})} + \underbrace{13}_{\text{gain}} = -122 \text{ dBm}$$

qui se compare avec l'intégration sur une bande passante de 2 MHz du plancher de bruit thermique à température ambiante donc $\underbrace{-174}_{\text{dBm/Hz}} + 10\log_{10}(\underbrace{2\cdot10^6}_{\text{Hz}}) = -111$ dBm, donc un signal 11 dB sous le

bruit thermique. Cette condition n'est pas la plus confortable pour valider un circuit inconnu avec une configuration inconnue.

3.1 Acquisition et analyse d'Iridium en temps réel

Nous avons discuté de la réception d'Iridium récemment [14] mais avec un récepteur de radio logicielle généraliste fournissant nombre de bits de résolution et donc une excellent quantification. Ici nous désirons valider la détection, voir le décodage, de Iridium avec des convertisseurs codant l'information sur 3 bits. L'issue de la tentative n'est pas évidente : en effectuant la corrélation entre un signal et la séquence pseudo-aléatoire de N bits ("chips") codant chaque bit de message émis par un satellite, la compression d'impulsion (inter-corrélation) accumule lors de l'intégrale toute l'énergie distribuée dans les N chips dans un unique pic de corrélation qui voit donc sa quantification améliorée de $\log_2(N)$ bits. Pour les 1023 chips de GPS, le gain est de l'ordre de 10 bits et les 3 bits initiaux deviennent 13 bits sur chaque pic de corrélation qui se répète chaque milliseconde (soit 1,023 Mchips/s divisé par $N = 1023$). La situation n'est pas aussi favorable avec Iridium qui transmet un signal puissant mais codé en phase sur deux états (BPSK) ou 4 états (QPSK) qu'il faut être capable d'identifier malgré la quantification médiocre et l'absence du gain de compression.

La première chose à voir est la gamme de fonctionnement de l'oscillateur commandé en tension, puisque la documentation technique en limite les caractéristiques à la bande utile, donc jusqu'à 1610 MHz, borne supérieure des fréquences nécessaires à décoder le système de navigation par satellite russe GLONASS. Bien que les registres du MAX2771 laissent configurer une large gamme de valeurs et notamment de fréquence d'oscillateur local commandé en tension (VCO), rien ne garantit que le matériel puisse respecter ces demandes. Nous avons testé, en configurant grâce à `app/pocket_conf/pocket_conf` de PocketSDR et en connectant l'entrée du MAX2771 à un signal radiofréquence continu issu d'un synthétiseur, les diverses combinaisons de $RDIV \in [0 : 1023]$ et $NDIV \in [36 : 32767]$ afin de produire une fréquence de l'oscillateur local $f_{LO} = f_{Xtal}/RDIV \times NDIC$ avec $f_{Xtal} = 24$ MHz afin de mesurer une raie décalée de 1 MHz de f_{LO}, et constatons le bon fonctionnement jusqu'à $NDIV = 68$ si $RDIV = 1$ pour produire $f_{LO} = 1632$ MHz, ou bien $NDIV = 546$ avec $RDIV = 8$ pour produire $f_{LO} = 1638$ MHz, mais au-delà ($NDIV = 547$) l'oscillateur local décroche. Ainsi, la fréquence centrale de Iridium de 1622 MHz est bien compatible avec le MAX2771. Noter que la loi complète gouvernant la fréquence de l'oscillateur local au moyen d'une boucle à verrouillage de phase *fractionnaire* est

$$f_{LO} = \frac{f_{Xtal}}{RDIV} \times \left(NDIV + \frac{FDIV}{2^{20}} \right)$$

avec $FDIV \in [36 - 32767]$ la partie fractionnaire du facteur multiplicatif de la fréquence. Pour chaque configuration, on pourra relire l'état des registres en lançant la commande `pocket_conf` sans argument : bien entendu en l'absence du second MAX2771 qui équipe la PocketSDR mais est absent de la carte d'évaluation, la configuration et la lecture des registres du second composant seront incohérentes mais sans conséquences.

Le choix du débit de données sur le bus USB est déterminé par l'horloge qui cadence les ADC et donc le signal IFCLK qui rythme le remplissage de la FIFO du FX2LP. L'oscillateur principal ($f_{Xtal} = 24$ MHz) qui cadence le FX2LP peut être multiplié ou divisé par 1, 2 ou 4 selon la valeur placé dans `REFDIV` lorsque $ADCCLK = 0$, puis les registres $REFCLK\_L\_CNT$ et $REFCLK\_L\_CNT$ déterminent la cadence des données par

$$f_{ADC} = f_{Xtal}(REFDIV) \frac{L\_CNT}{4096 - M\_CNT + L\_CNT}$$

où $f_{XTal}(REFDIV)$ est l'opération indexée par $REFDIV$ sur f_{XTal} (/4, /2, ×1, ×2, ×4) et nous constatons que choisir $L\_CNT = 2048$ avec $M\_CNT = 0$ permet de diviser par 3, donc par exemple de produire 8 MS/s si $REFDIV = 3$ pour multiplier par 1 f_{Xtal}.

À l'issue de ces explorations, la configuration finalement utilisée exploite une fréquence d'échantillonnage de 24 MHz (afin d'atteindre après transposition et décimation plus que les 10 MHz que couvre Iridium) et une fréquence intermédiaire de 6,5 MHz, choisie comme plus que les 5 MHz nécessaires à conserver 10 MHz efficaces en bande de base après transposition. Pour ce faire, `NDIV=26925` et `RDIV=400` avec `INT_PLL=1` pour une PLL sans partie fractionnaire, tel que la fréquence de l'oscillateur local soit $24 \cdot 10^6/400 \times 26925 = 1615.5 \cdot 10^9$ Hz soit $1622 - 6.5$ MHz avec 1622 MHz la fréquence centrale d'Iridium. Pour l'ADC, comme nous conservons la fréquence d'horloge, nous choisissons simplement `REFDIV=3`.

La seconde grande différence entre GPS et Iridium est qu'alors que le premier émet en continu, le second ne transmet que par **impulsions brèves** à des instants arbitraires dépendant de la position des satellites dans le ciel et des requêtes des terminaux au sol. Ainsi, enregistrer en aveugle quelques minutes d'Iridium pour se rendre compte que peu ou pas de signal est exploitable est frustrant : nous désirons visualiser en temps-réel le signal au moyen de l'analyseur de spectres (`Frequency QT GUI Sink`) de GNU Radio. Bien entendu il n'existe pas d'interface GNU Radio pour le FX2LP, mais comme PocketSDR fournit `pocket_dump` capable d'écrire dans un fichier, il nous suffit de créer un *pipe* nommé [16] pour transférer les données acquises depuis le port USB vers un pseudo-fichier lu par le `File Source` de GNU Radio qui alimente l'analyseur de spectres. La chaîne de traitement un peu triviale GNU Radio Companion est proposée en Fig. 6 (haut-gauche), et sous réserve d'avoir retiré le filtre passe-bande (bas, gauche) d'une antenne GPS active [14] et avoir inséré un T de polarisation entre le MAX2771 et l'antenne pour alimenter son amplificateur [14], nous observerons un spectre représentatif des signaux Iridium.

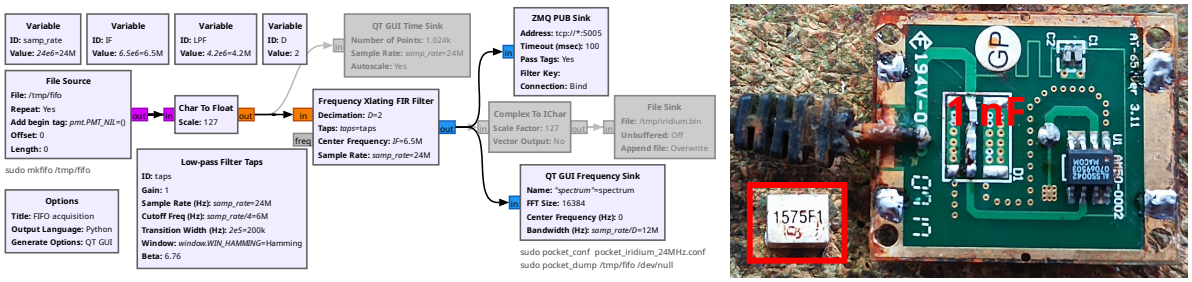


FIGURE 6 – Gauche : chaîne de traitement GNU Radio prenant en entrée une FIFO sous forme de *pipe* nommée `/tmp/fifo` créée par `mkfifo /tmp/fifo`. Ce flux d'entiers réels codés sur 8 bits au rythme de 24 MS/s (défini dans le `samp_rate`) est converti en nombres à virgule flottante, transposition de fréquence par $IF = 6,5$ MHz, filtrage et décimation, pour affichage dans l'analyseur de spectres sur une bande de 12 MHz et émission vers un ZeromMQ Publish dont on pensera à activer `Pass Tags: Yes` pour être cohérent avec `gr-iridium`. Ici la socket est connectée au port 5005 de l'hôte local mais les données pourraient très bien être transmises sur réseau pour un traitement déporté. Droite : antenne GPS L1 modifiée pour la réception des signaux Iridium en retirant son filtre passe-bande centré sur 1575,42 MHz pour le remplacer par un condensateur de 1 nF, donc une impédance de 0,1 Ω à cette fréquence, tout en évitant d'amener la composante DC d'alimentation de l'amplificateur au plus près de l'antenne sur la broche radiofréquence de l'amplificateur.

Par ailleurs, étant donné que `gr-iridium` de Sec et Schneider [15] ne connaît pas le concept de fréquence intermédiaire, nous profitons de l'affichage du spectre par GNU Radio, qui nécessite déjà d'éliminer cet écart à la porteuse nominale par un `Xlating FIR Filter`, pour transmettre le résultat de la transposition dans un flux ZeroMQ Publish. Comme le `Xlating FIR Filter` a rendu la moitié du spectre réel inutile, il est muni d'un filtre passe-bas et une décimation d'un facteur 2, de tel sorte que `gr-iridium` reçoit un flux de complexes en nombres à virgule flottante à 12 MS/s. Nous avions initialement tenté de sauver dans un fichier, mais la taille grossit tellement vite à 12 MS/s et les informations Iridium tellement rares qu'un traitement en temps réel est plus pertinent : 184 MB en format ASCII ('0' et '1') après 3.3 heures d'acquisition et de décodage au lieu de 132 GB qu'auraient occupées les données brutes.

En effet, `gr-iridium` sait lire le flux depuis ZeroMQ Subscribe pour le traiter, il faut juste l'informer du débit de données en modifiant `examples/zeromq-sub.conf` avec `center_freq=1622000000` et `sample_rate=12000000` tout en s'assurant que `address=tcp://127.0.0.1:5005` est cohérent avec le port sélectionné dans le ZeroMQ Publish de GNU Radio.

Une fois toutes ces briques assemblées, la séquence d'acquisition (Fig. 6) est

1. `sudo mkfifo /tmp/fifo` pour créer le tuyau,
2. `sudo app/pocket_conf/pocket_conf pocket_iridium_24MHz.conf` pour configurer le MAX2771,
3. `sudo app/pocket_conf/pocket_conf` pour vérifier que la configuration a bien été prise en compte,
4. `sudo app/pocket_dump/pocket_dump /tmp/fifo /dev/null` pour transmettre le flux IQ de l'unique MAX2771 vers la tuyau, l'autre flux partant à la poubelle.

5. GNU Radio lit depuis le tuyau pour transposer, décimer, afficher le spectre et re-émettre vers ZeroMQ Publish par `python3 fifo_acq.py` avec le script Python produit par GNU Radio Companion depuis `fifoinout.grc`,
6. enfin, `gr-iridium` reçoit le flux IQ pour en extraire les informations numériques et les sortir sur `stdout`. Un premier essai peut consister à attendre les trames *Iridium Ring Alert* (IRA) qui démontrent que des signaux de satellites sont bien reçus avec

   ```
   gr-iridium$ ./apps/iridium-extractor -D 4 --multi-frame ./examples/zeromq-sub.conf | \
     python3 -u ../iridium-toolkit/iridium-parser.py --harder | grep IRA
   ```

   mais comme il y a bien plus d'informations qui transitent que IRA, nous préférerons stocker ces bits dans un fichier pour post-traitement par

   ```
   gr-iridium$ ./apps/iridium-extractor  -D 4 --multi-frame ./examples/zeromq-sub_12MSps.conf > \
      /tmp/240807iridium.bits
   ```

7. nous traitons ces bits pour essayer de retrouver des phrases intelligibles avec

   ```
   iridium-toolkit$ iridium-parser.py -p 240807iridium.bits --harder --uw-ec > 240807iridium.parser
   ```

8. nous pouvons finalement rechercher dans les phrases les trames de localisation et d'identification des avions selon le protocole ACARS

   ```
   iridium-toolkit$ reassembler.py -i 240807iridium.parser -m acars
   ```

   ou produire un fichier KML contenant l'emplacement des faisceaux transmis dans les trames IRA par les satellites détectés

   ```
   iridium-toolkit$ grep ^IRA 240807iridium.parser | perl mkkml tracks > 240807iridium.kml
   ```

Le résultat de ces traitements est deux aéronefs détectés

```
2024-08-07T14:02:03 [hdr: 0339010100000001] Dir:DL Mode:2 REG:F-GXLI  ACK:7 Label:_? (Demand mode) bID:F
2024-08-07T14:42:12                         Dir:DL Mode:2 REG:GFHFX   ACK:8 Label:_? (Demand mode) bID:Z
```

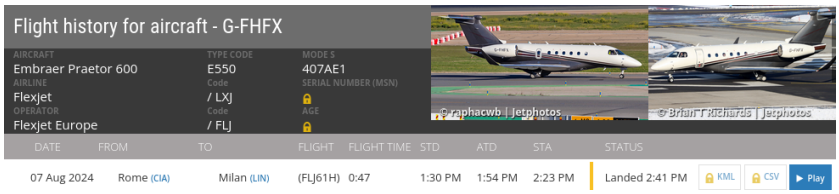


FIGURE 7 – Capture d'écran de FlightRadar24 validant l'analyse d'un message ACARS reçu par Iridium.

avec un petit avion effectivement identifié par `flightradar24.com` à ce moment entre Rome et Milan (Fig. 7), et un Beluga d'Airbus qui normalement ne devait pas voler à ce moment mais a peut être effectué un test de communication depuis le sol. La carte des faisceaux (Fig. 8) est cohérente avec une réception depuis Clermont-Ferrant (étoile jaune) selon une vue partiellement obstruée en direction de l'est/sud-est.

Nous pouvons donc être à peu près confiants que nous avons bien reçu et décodé des trames Iridium. Rappelons que l'objectif n'est pas de réaliser un récepteur Iridium robuste mais de valider le bon fonctionnement du MAX2771 sur un signal puissant, qui a tout de même permis de décoder nombre de trames numériques, un joli *hack* du MAX2771 au sens original du terme [17].

3.2 Analyse de GPS en post-traitement

L'acquisition du signal GPS est validée dans un premier temps en détectant les signaux des satellites reçus – phase d'acquisition d'un récepteur qui ne connaît rien de son emplacement et de la géométrie de la constellation – en recherchant par corrélation tous les codes d'identification des satellites (Gold Codes) possibles pour tous les décalages Doppler possibles. Cependant cette étape, prise en charge par `python/pocket_acq.py` de PocketSDR, doit aussi tenir compte d'un éventuel décalage de l'oscillateur local du récepteur : par défaut ce programme ne recherche que les décalages Doppler dans la gamme

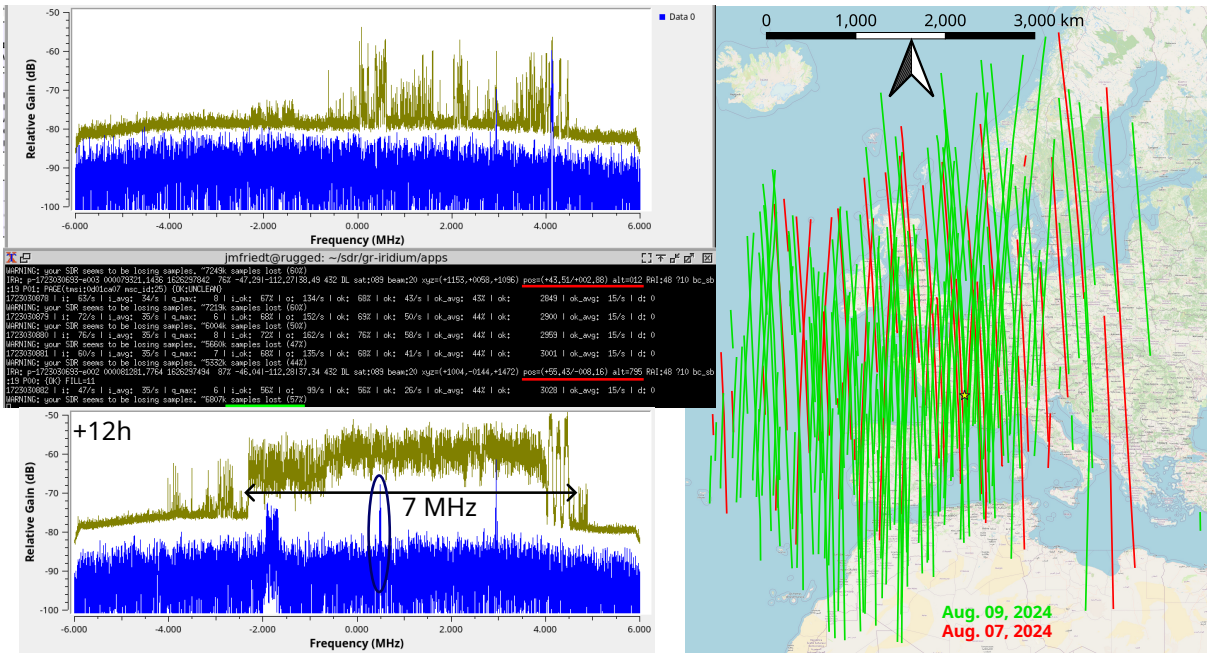


FIGURE 8 – Gauche : en haut le spectre affiché par GNU Radio Companion après transposition de fréquence par le `Xlating FIR Filter` et décimation d'un facteur deux pour fournir 12 MHz de bande passante, suffisant pour couvrir tous les signaux d'Iridium en même temps. En haut en début d'acquisition les canaux individuels de communication sont visibles, en bas après plus de 12 h d'acquisition, le spectre est devenu uniforme sur les plus de 7 MHz nécessaires à couvrir toutes les sous bandes. Sur ce spectre, en ellipse bleue foncée un bref *burst* de communication capturé dans cette image. Droite : carte des faisceaux identifiés par la trame IRA (fond de carte : OpenStreetMap dans QGIS) pour deux jours d'acquisition, les 7 et 9 Août 2024 pendant 12 h, pour valider la reproductibilité. Au milieu, `gr-iridium` nous informe de nombreuses trames perdues (sous-ligné vert) mais les trames IRA sont tout de même détectées (sous-ligné rouge).

±5 kHz correspondant uniquement à la vitesse de déplacement du satellite, mais il est prudent en cas d'échec du décodage d'étendre cette gamme (option `-d`, une valeur de 30000 pour 30 kHz est raisonnable pour un résonateur à quartz de qualité convenable). Une fois les satellites identifiés, garantie de la qualité du signal acquis, nous pourrons finaliser la chaîne de traitement en trouvant la solution optimale de positionnement du récepteur compte tenu du temps de vol observé pour au moins 4 satellites de la constellation : cette solution en position, vitesse et temps (PVT) sera obtenue au moyen de `gnss-sdr` que nous alimenterons avec le fichier des acquisitions IQ issues du MAX2771, puis obtiendrons en temps réel.

Dans un premier temps, le MAX2771 est configuré : comme avec Iridium, nous profitons de
`app/pocket_conf/pocket_conf`
mais cette fois avec un fichier de configuration `conf/pocket_L1L2_8MHz.conf`. Ainsi le MAX2771 est configuré avec une fréquence centrale de la bande L1 décalée de la fréquence intermédiaire de 2 MHz, l'échantillonnage à 8 MS/s, et la configuration du second MAX2771 pour la bande L2 simplement ignorée. Une fois la configuration relue et validée par `app/pocket_conf/pocket_conf` sans argument, nous acquérons les fichiers de mesures par `app/pocket_dump/pocket_dump -t 2 ch1.bin ch2.bin` avec `-t` pour indiquer que nous ne voulons enregistrer que deux secondes, et les arguments optionnels des noms de fichiers indiquent les deux voies de sortie. De nouveau en l'absence de second MAX2771, le fichier `ch2.bin` ne contiendra aucune valeur exploitable et seul `ch1.bin` sera exploitable. Afin d'économiser l'espace disque, `ch2.bin` pourra avantageusement être remplacé par `/dev/null` sans perte de fonctionnalité.

Afin de valider la pertinence des données acquises dans `ch1.bin`, nous allons tenter une acquisition des satellites, en corrélant séquentiellement les 32 codes pseudo-aléatoires possibles pour tous les décalages Doppler possibles. Ce résultat s'obtient depuis le répetoire de PocketSDR par
`python3 ./python/pocket_acq.py ch1.bin -f 8 -fi 2 -sig L1CA -prn 1-32 -d 30000`

pour indiquer que la fréquence d'échantillonnage est 8 MHz, fréquence intermédiaire de 2 MHz, impliquant implicitement que les données sont des réels (sans partie imaginaire), et que nous recherchons les satellites 1 à 32 de GPS avec un décalage de fréquence maximal de 30 kHz. Le résultat est illustré en Fig. 9 qui valide par ailleurs la cohérence de l'analyse avec l'observation d'un téléphone portable utilisé comme récepteur GNSS.

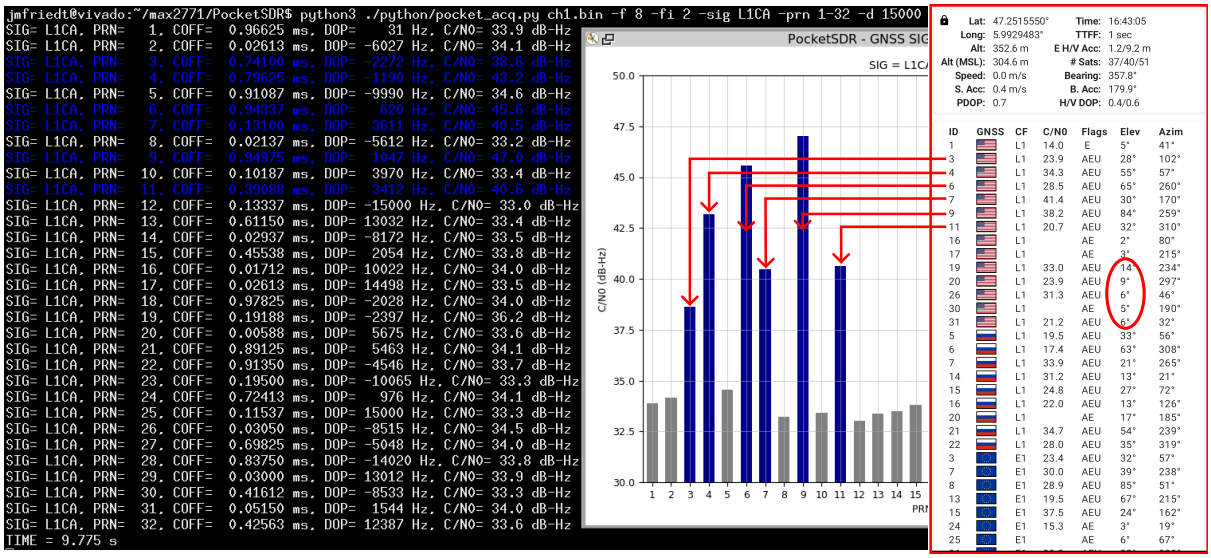


FIGURE 9 – Gauche : le résultat de `pocket_acq.py` de PocketSDR appliqué sur un fichier acquis par `pocket_dump` du même projet. Milieu : la sortie graphique de la phase d'acquisition des satellites, avec en abscisse le numéro du satellite (PRN) et en ordonnée le rapport signal à bruit. Droite : observation par téléphone portable sous Android exécutant `GPSTest` (cadre rouge), en parfaite cohérence avec le signal acquis par le MAX2771 (flèches rouges), à l'exception des satellites aux indices les plus élevés qui sont trop bas sur l'horizon pour être détectables (ellipse rouge).

En configurant de la même façon le MAX2771 en mode IQ, sans fréquence intermédiaire, pour acquérir au débit de 4 MS/s grâce à `conf/pocket_L1L2_4MHz.conf` de PocketSDR, alors `pocket_dump` affichera un message de la forme

```
$ sudo ./app/pocket_dump/pocket_dump -t 2 ch1.bin ch2.bin
  TIME(s)     T    CH1(Bytes)    T    CH2(Bytes)    RATE(Ks/s)
      2.0    IQ      15990784   IQ      15990784        3993.7
```

qui permet de bien vérifier le débit (ici 4 MS/s) et la nature (ici IQ, en accord avec la configuration) même si le second MAX2771 de la PocketSDR est absent.

Le décodage par `gnss-sdr` nécessite un fichier de configuration qui connecte les diverses étapes de traitement entre elles selon un graphique ordonnancé par GNU Radio, prenant en particulier compte du fait que chaque type de donnée en sortie corresponde au type en entrée du bloc de traitement suivant. Naturellement, un récepteur de radio logicielle voudrait traiter des valeurs complexes en bande de base, mais nous constatons que notre montage perd la liaison USB rapidement lorsqu'un tel type de donnée est transféré, probablement à cause d'un couplage électrique excessif entre les lignes portant les signaux d'horloge et de données au FX2LP (voir conclusion). Ayant donc abandonné l'option de traiter des complexes (type `ibyte` dans la nomenclature `gnss-sdr`) selon la configuration vue juste au-dessus (`pocket_L1L2_4MHz.conf`), il reste deux voies à explorer, le traitement en temps réel et le post-traitement de fichiers enregistrés selon la configuration `pocket_L1L2_8MHz.conf`.

La première solution est rapidement éliminée puisque le type FIFO qui permettrait de transmettre des données par un *pipe* nommé, type `Fifo_Signal_Source` du `SignalSource.implementation` dans la configuration de `gnss-sdr` ne sait pas traiter de nombre réel (sans partie imaginaire), et il ne reste donc que l'option de traiter un fichier contenant des acquisitions avec fréquence intermédiaire, et convaincre `gnss-sdr` de transposer le signal de cette fréquence intermédiaire avant d'en extraire les informations de temps de vol, donc de peudo-range, donc de solution PVT. Le fichier de configuration commence par le classique

```
[GNSS-SDR]
GNSS-SDR.internal_fs_sps=8000000
```

qui indique que les informations sont acquises à 8 MS/s. La source venant d'un fichier contenant des données sur 8 bits est renseignée par

```
SignalSource.implementation=File_Signal_Source
SignalSource.filename=ch1.bin
SignalSource.item_type=byte
SignalSource.sampling_frequency=8000000
```

qui est intuitif. L'étape suivante est la plus complexe puisque nous devons convertir les entiers en nombres à virgule flottante complexes après avoir transposé de la fréquence intermédiaire de 2 MHz grâce au `Frequency Xlating FIR Filter` de GNU Radio, à savoir un mélange avec un oscillateur local, un filtre et une décimation :

```
SignalConditioner.implementation=Signal_Conditioner
DataTypeAdapter.implementation=Byte_To_Short
InputFilter.implementation=Freq_Xlating_Fir_Filter
InputFilter.input_item_type=short
InputFilter.output_item_type=gr_complex
```

commence par convertir les données d'entrée de 8 à 16 bits en vue d'alimenter le `Frequency Xlating FIR Filter` qui sortira des complexes en virgule flottante. Les propriétés du filtre sont déterminées par sa bande passant et sa bande de coupure (Fig. 10), toujours normalisées à la fréquence de Nyquist (demi fréquence d'échantillonnage f_s) :

```
InputFilter.taps_item_type=float
InputFilter.number_of_taps=5
InputFilter.number_of_bands=2
InputFilter.band1_begin=0.0
InputFilter.band1_end=0.40
InputFilter.band2_begin=0.50
InputFilter.band2_end=1.0
InputFilter.ampl1_begin=1.0
InputFilter.ampl1_end=1.0
InputFilter.ampl2_begin=0.0
InputFilter.ampl2_end=0.0
InputFilter.band1_error=1.0
InputFilter.band2_error=1.0
InputFilter.filter_type=bandpass
InputFilter.grid_density=16
InputFilter.sampling_frequency=8000000
InputFilter.IF=2000000
```

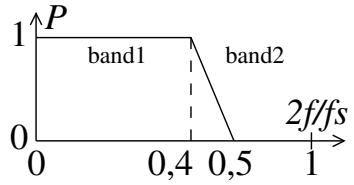


FIGURE 10 – Gabarit du filtre passe-bas. Noter l'abscisse graduée en fréquence normalisée, comme dans tout système échantillonné en temps discret.

donc 40% de la bande (allant de 0 à la demi-fréquence d'échantillonnage) est passante, et la coupure commence à 50% de cette bande, toujours représentée en fréquence normalisée, donc en attribuant la valeur 1 à la fréquence de Nyquist égale à la demi-fréquence d'échantillonnage. Le nombre de coefficients du filtre (`taps`) est de 5. Finalement,

```
Resampler.implementation=Pass_Through
Resampler.sample_freq_in=8000000
Resampler.sample_freq_out=8000000
Resampler.item_type=gr_complex
```

ne décime pas le flux mais le transmet aux phases d'acquisition et de traitement *tracking* pour trouver la solution PVT issue du signal L1 de la constellation GPS en exploitant au maximum 12 canaux, donc potentiellement les signaux de 12 satellites. Ce nombre de canaux est limité par la puissance de calcul de l'ordinateur pour un traitement en temps réel, et en pratique par le nombre de satellites visibles depuis un site donné :

```
Channel.signal=1C
Channels.in_acquisition=1
Channels_1C.count=12
Acquisition_1C.implementation=GPS_L1_CA_PCPS_Acquisition
Acquisition_1C.item_type=gr_complex
Acquisition_1C.doppler_max=30000
```

```
Acquisition_1C.doppler_step=250
cracking_1C.implementation=GPS_L1_CA_DLL_PLL_Tracking
cracking_1C.item_type=gr_complex
Tracking_1C.early_late_space_chips=0.5
Tracking_1C.pll_bw_hz=25.0;
Tracking_1C.dll_bw_hz=3.0;
Tracking_1C.dump=false;
```

qui comme auparavant tente de compenser un décalage Doppler jusqu'à 30 kHz, par pas de 250 Hz choisi comme une valeur petite devant l'inverse de la durée de chaque bit qui est 1 ms, donc petit devant 1 kHz.

Le reste n'est que classique pour conclure la recherche des solutions PVT et est imposé d'après la documentation de `https://gnss-sdr.org/docs/sp-blocks/` :

```
TelemetryDecoder_1C.implementation=GPS_L1_CA_Telemetry_Decoder
PVT.implementation=RTKLIB_PVT
PVT.positioning_mode=PPP_Static
PVT.output_rate_ms=1000
```

À l'issue de l'exécution par `gnss-sdr -c fichier.conf` avec le fichier de configuration contenu dans `fichier.conf`, nous avons le plaisir d'obtenir

```
Initializing GNSS-SDR v0.0.19.git-main-87fcfd237 ... Please wait.
RF Channels: 1
Processing file /tmp/ch1.bin, which contains 3137273856 samples (3137273856 bytes)
GNSS signal recorded time to be processed: 392.059 [s]
Current receiver time: 1 s
Tracking of GPS L1 C/A signal started on channel 0 for satellite GPS PRN 01 (Block IIF)
Tracking of GPS L1 C/A signal started on channel 1 for satellite GPS PRN 13 (Block IIR)
Tracking of GPS L1 C/A signal started on channel 2 for satellite GPS PRN 14 (Block III)
Tracking of GPS L1 C/A signal started on channel 3 for satellite GPS PRN 15 (Block IIR-M)
Tracking of GPS L1 C/A signal started on channel 4 for satellite GPS PRN 16 (Block IIR)
Tracking of GPS L1 C/A signal started on channel 5 for satellite GPS PRN 17 (Block IIR-M)
Tracking of GPS L1 C/A signal started on channel 6 for satellite GPS PRN 18 (Block III)
Tracking of GPS L1 C/A signal started on channel 7 for satellite GPS PRN 19 (Block IIR)
Tracking of GPS L1 C/A signal started on channel 8 for satellite GPS PRN 20 (Block IIR)
Tracking of GPS L1 C/A signal started on channel 9 for satellite GPS PRN 21 (Block IIR)
Tracking of GPS L1 C/A signal started on channel 10 for satellite GPS PRN 22 (Block IIR)
Tracking of GPS L1 C/A signal started on channel 11 for satellite GPS PRN 23 (Block III)
Current receiver time: 2 s
Current receiver time: 3 s
...
Current receiver time: 13 s
GPS L1 C/A tracking bit synchronization locked in channel 8 for satellite GPS PRN 20 (Block IIR)
GPS L1 C/A tracking bit synchronization locked in channel 1 for satellite GPS PRN 13 (Block IIR)
...
Current receiver time: 22 s
New GPS NAV message received in channel 8: subframe 2 from satellite GPS PRN 20 (Block IIR) with CN0=45 dB-Hz
New GPS NAV message received in channel 1: subframe 2 from satellite GPS PRN 13 (Block IIR) with CN0=44 dB-Hz
...
New GPS NAV message received in channel 1: subframe 1 from satellite GPS PRN 13 (Block IIR) with
CN0=44 dB-Hz
GPS L1 C/A tracking bit synchronization locked in channel 10 for satellite GPS PRN 07 (Block IIR-
M)
First position fix at 2024-Jul-22 17:57:18.100000 UTC is Lat = 47.2517 [deg], Long = 5.99328 [deg],
Height= 364.788 [m]
Current receiver time: 1 min 17 s
Position at 2024-Jul-22 17:57:19.000000 UTC using 4 observations is Lat = 47.251622 [deg],
Long = 5.993225 [deg], Height = 361.46 [m]
Velocity: East: 0.32 [m/s], North: -0.04 [m/s], Up = -0.05 [m/s]
Current receiver time: 1 min 18 s
Position at 2024-Jul-22 17:57:20.000000 UTC using 4 observations is Lat = 47.251622 [deg],
Long = 5.993205 [deg], Height = 360.35 [m]
...
```

dont nous n'avons conservé que les principales étapes pour décrire le sens des messages. Les informations commençant par `Tracking` n'indiquent nullement la présence d'un signal exploitable dans l'enregistrement, mais uniquement que la signature (code pseudo-aléatoire PRN) d'un certain satellite (`GPS PRN XX`) est en cours de recherche dans le signal traité, assigné au canal "channel". Le point crucial est l'apparition de `tracking bit synchronization locked` qui indique que le signal d'un satellite a été identifié et pourra être analysé en vue de décoder le message de navigation : le numéro du satellite

correspond au canal qui lui avait été attribué. Enfin toutes les informations nécessaires à placer le satellite dans l'espace sont acquises lors de `New GPS NAV message received`, même si en pratique il faudra décoder les 5 "sub-frames" successives puisque par exemple la première phrase indique les corrections à appliquer aux horloges atomiques embarquées, les deux suivantes les éphémérides du satellite, puis la date et les conditions ionosphériques et enfin le statut de la constellation. Nous constaterons que le succès est souvent au rendez-vous après le décodage d'une "sub-frame" 3 (même si ici ce fut après une sub-frame 1), sous réserve d'avoir décodé les messages de navigation d'au moins quatre satellites (pour résoudre l'équation à 4 inconnues que sont la position et le temps) pour fournir le résultat tant attendu de `First position fix` qui se répétera ensuite tant que suffisamment de satellites sont visibles. Comme chaque *frame* dure 30 secondes, il faut attendre au minimum cette durée une fois le premier message de navigation obtenu.

3.3 Analyse de GPS en temps-réel

Pour conclure cette étude, il peut sembler désirable d'obtenir la position GPS en temps réel et non en post-traitement d'un fichier enregistré qui est nécessairement contraint en espace et donc en durée. Cependant, la source FIFO de `gnss-sdr` n'accepte que des valeurs complexes, et en l'état actuel du montage spaghetti, le transfert de complexes induit une corruption du flux USB au bout de quelques secondes à dizaines de secondes. Nous avons donc abordé le problème de la façon suivante :

- création d'un *pipe* nommé `sudo mkfifo /tmp/fifo` pour acquérir les données (appartenant à root pour que `pocket_dump` lancé en `sudo` puisse y écrire),
- `sudo pocket_dump /tmp/fifo /dev/null` pour alimenter le *pipe* avec les données réelles acquises à 8 MS/s avec une fréquence intermédiaire de 2 MHz, le second canal partant dans le vide de `/dev/null` puisqu'un seul MAX2771 équipe la carte d'évaluation,
- une chaîne de traitement GNU Radio se charge de transposer en fréquence (`Frequency Xlating FIR Filter` et en profite pour filtrer et décimer le flux résultant dont la majorité des composantes spectrales est devenue inutile (Fig. 11, gauche). Le débit de sortie est 2 MS/s complexes
- ayant constaté qu'un fichier de sortie vers un autre *pipe* nommé communique mal avec `gnss-sdr`, nous avons opté pour une sortie de GNU Radio sous forme de socket ZeroMQ Publish, puisque `gnss-sdr` propose l'interface ZeroMQ Subscribe.

Pour atteindre ce résultat, les modifications au fichier de configuration de `gnss-sdr` sont la définition de la source et retirer la transposition de fréquence dans `gnss-sdr` puisque GNU Radio s'en est déjà chargé en amont :

```
GNSS-SDR.internal_fs_sps=2000000
SignalSource.implementation=ZMQ_Signal_Source
SignalSource.endpoint=tcp://127.0.0.1:5555
SignalSource.sample_type=gr_complex
SignalSource.sampling_frequency=2000000
SignalConditioner.implementation=Pass_Through
Channels_1C.count=12
Channels.in_acquisition=1
Channel.signal=1C
```

et le reste est identique à la configuration précédente.

```
Tracking of GPS L1 C/A signal started on channel 0 for satellite GPS PRN 01 (Block IIF)
Current receiver time: 1 min 49 s
New GPS NAV message received in channel 9: subframe 1 from satellite GPS PRN 21 (Block IIR) with CN0=42 dB-Hz
New GPS NAV message received in channel 5: subframe 1 from satellite GPS PRN 02 (Block IIR) with CN0=43 dB-Hz
New GPS NAV message received in channel 6: subframe 1 from satellite GPS PRN 08 (Block IIF) with CN0=44 dB-Hz
New GPS NAV message received in channel 4: subframe 1 from satellite GPS PRN 32 (Block IIF) with CN0=43 dB-Hz
First position fix at 2024-Jul-26 09:31:48.120000 UTC is Lat = 47 [deg], Long = 6 [deg], Height= 3.8e+02 [m]
Current receiver time: 1 min 50 s
The RINEX Navigation file header has been updated with UTC and IONO info.
Position at 2024-Jul-26 09:31:49.000000 UTC using 4 observations is Lat = 47.251620 [deg], Long = 5.993221 [deg],
Height = 366.06 [m]
Velocity: East: 0.91 [m/s], North: 0.65 [m/s], Up = 3.82 [m/s]
Current receiver time: 1 min 51 s
Loss of lock in channel 11!
Tracking of GPS L1 C/A signal started on channel 11 for satellite GPS PRN 19 (Block IIR)
Position at 2024-Jul-26 09:31:49.989988 UTC using 4 observations is Lat = 47.251560 [deg], Long = 5.993090 [deg],
Height = 311.77 [m]
Velocity: East: -0.83 [m/s], North: -1.32 [m/s], Up = -2.92 [m/s]
```

qui démontre la convergence de la solution en moins de deux minutes, en accord avec les observations de GPSTest sur téléphone mobile Android utilisé comme récepteur de référence (Fig. 11, droite).

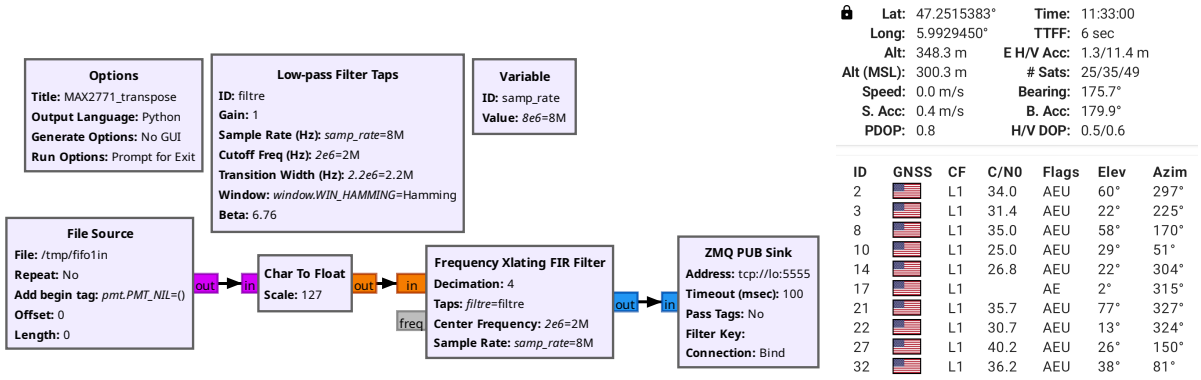


FIGURE 11 – Gauche : chaîne de traitement GNU Radio lisant des entiers sur 8 bits depuis un fichier qui est en réalité une FIFO (*pipe* nommé), suivi de la conversion vers un nombre flottant avec une homothétie pour ramener le résultat entre -1 et 1, transposition de fréquence et communication du résultat par socket ZeroMQ Publish, la sortie dans un fichier connecté à un autre *pipe* n'ayant pas donné de bons résultats. Droite : capture d'écran de GPSTest exécuté sur téléphone portable utilisé comme récepteur de référence, démontrant la cohérence avec les sattelites exploités par `gnss-sdr` pour obtenir la solution (PRN 2, 8, 21 et 32).

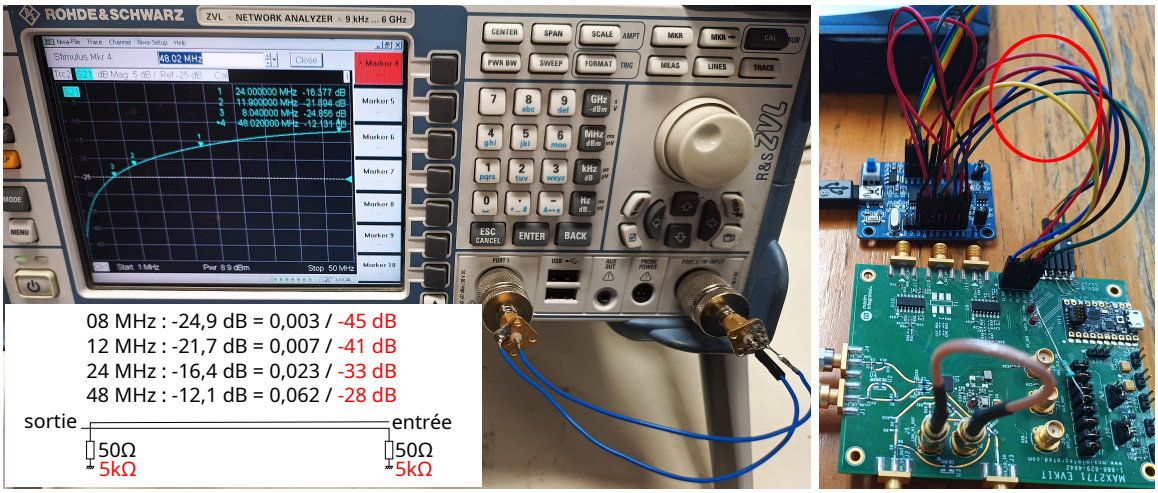
Une vidéo de cette séquence de traitements, fournissant notamment la séquence de commandes et la dynamique d'acquisition que nous ne pouvons reproduire dans ces pages statiques, se trouve à `https://www.youtube.com/watch?v=B5UcFnkbXIk`

4 Conclusion

Nous avons identifié le protocole de la carte d'évaluation du MAX2771, compris que seule la conversion de commandes USB vers SPI est prise en charge mais pas le transfert de données haut-débit, appris à configurer le FX2LP comme interface de transfert entre les mots formés de bits en parallèle et une horloge vers USB, et acquis ces données sur PC. Cependant, le montage spaghetti avec les longs fils de communication portant des signaux à plusieurs MHz voir dizaine de MHz ne peut garantir une transaction robuste, et un circuit dédié s'avère nécessaire pour tirer pleinement parti des performances du MAX2771.

Couplage entre fils adjacents

La mesure à l'analyseur de réseau du couplage entre deux fils adjacents illustre le problème du potentiel induit par un signal sur le fil portant le signal voisin. Ce problème est d'autant plus important que la fréquence de communication augmente.



Un analyseur de réseau mesure par son paramètre de transmission S_{21} le ratio du potentiel d'entrée V_i au potentiel de sortie V_o (ce sont bien des tensions et non des puissances). La figure ci-dessus reporte en échelle logarithmique décibels (dB) ainsi que la conversion en échelle linéaire $V_o/V_i = 10^{dB/10}$ pour avoir le rapport des tensions entre le signal transmis sur un fil de données et le couplage sur le fil voisin. Nous avons reporté en noir la mesure lorsque chaque fil est terminé à la masse par une résistance 50 Ω visant à adapter l'impédance, mais maximisant le courant dans le fil et donc le couplage. Remplacer la résistance de charge par une valeur de 5 kΩ pourrait laisser penser que le couplage baisse (en rouge) mais cela induit un autre problème puisque la désadaptation de la ligne émettrice crée une onde stationnaire qui perturbera la détection des signaux numériques. On pourrait penser que travailler à 8 MHz n'induit qu'un couplage minime de l'ordre du millième, mais il faut se rappeler que la transmission de signaux carrés tel que l'horloge émise par le FX2LP porte toutes les harmoniques de la fréquence d'horloge avec une puissance qui décroît comme le numéro de l'harmonique. Ainsi, un signal carré de 3,3 V à 8 MHz transporte encore une composante de 0,66 V à 40 MHz (harmonique 5) et 0,47 V à 56 MHz (harmonique 7 puique seules les harmoniques impaires représentent un signal carré) qui eux couplent inductivement efficacement entre les deux fils. On comprend donc que le plat de spaghettis du montage reliant la carte d'évaluation du MAX2771 au FX2LP n'est pas garant de transactions numériques robustes.

Tout comme le PocketSDR qui a inspiré cette étude, nous visons à exploiter les signaux acquis simultanément par plusieurs MAX2771 pour évaluer la direction d'arrivée d'un signal et éventuellement annuler une source de leurrage ou de brouillage. Le circuit dédié à ces développements, et bien d'autres, fera l'objet de l'article qui poursuivra cette description.

Tous les codes sources sont disponibles à `https://github.com/jmfriedt/max2771_fx2lp/` et en particulier dans le sous répertoire `Maxim_EvalBoard` pour cet article. Les ouvrages de la bibliographie ont été obtenus sur Library Genesis.

Références

[1] J.-M. Friedt, É. Carry, *Enregistrement de trames GPS — développement sur microcontrôleur 8051/8052 sous GNU/Linux*, GNU/Linux Magazine France, **81** (Février 2006)

[2] J.-M Friedt, *Exploitation des signaux de référence de navigation par satellite pour un positionnement centimétrique : RTKLib fait appel à Centipède et l'IGN pour afficher dans QGis*, Hackable **48** (Mai/Juin 2023)

[3] Analog Devices Inc., *Software Development : MAX2771EVkit GUI, 1.0.0* à `https://www.analog.com/en/resources/evaluation-hardware-and-software/evaluation-boards-kits/max2771evkit.html`

[4] J.-M. Friedt, W. Feng, *Anti-leurrage et anti-brouillage de GPS par réseau d'antennes*, MISC **110** (Juillet-Août 2020)

[5] les projets https://fpga4u.epfl.ch/wiki/FX2.html et https://fpga4u.epfl.ch/wiki/FPGA4RF.html datent de 2011 mais restent d'actualité

[6] https://sdcc.sourceforge.net/ mise à jour le 6 Mars 2024, aussi disponible comme paquet Debian du même nom

[7] codes sources du firmware de l'analyseur logique sigrok à https://sigrok.org/download/source/sigrok-firmware-fx2lafw/

[8] Cypress, *CY7C68013 EZ-USB FX2 USB Microcontroller – High-Speed USB Peripheral Controller* (2002) à https://www.keil.com/dd/docs/datashts/cypress/cy7c68xxx_ds.pdf

[9] https://saturn.ffzg.hr/rot13/index.cgi/U2CY7C68013-56.pdf?action=attachments_download;page_name=fx2lp;id=20140817120222-0-10210

[10] *Using the Free SDCC C Compiler to Develop Firmware for the DS89C430/450 Family of Microcontrollers* à https://www.analog.com/en/resources/app-notes/using-the-free-sdcc-c-compiler-to-develop-firmware-for-the-ds89c430450-family-of-microcontroller

[11] "*Reading Serial EEPROM which is Connected to the FX2*" à https://community.infineon.com/t5/Knowledge-Base-Articles/Reading-Serial-EEPROM-which-is-Connected-to-the-FX2/ta-p/254704 (2008)

[12] https://github.com/siddharthdeore/fx2lp_usb_dev

[13] Keil est *big endian* selon https://groups.google.com/g/comp.arch.embedded/c/Cabbhrl9_oM/m/3yOfgiqm7_YJ tandis que SDCC est *little endian* tel que mentionné à https://community.silabs.com/s/article/common-pitfalls-when-using-sdcc

[14] J.-M Friedt, *À l'écoute des messages transmis par satellite en orbite basse : Iridium*, MISC Hors Série **29** (2024)

[15] https://github.com/muccc/gr-iridium présenté à Sec et Scheider, *Iridium Hacking*, Chaos Communication Camp (2015) à https://av.tib.eu/media/38121

[16] J.-M Friedt, *Échanges de données pour un traitement distribué : communication par réseau ou entre langages*, GNU/Linux Magazine France **267** (Jan-Fév. 2024)

[17] comme les lecteurs de ce journal le savent bien et les journalistes de France Info ou les auteurs de https://www.orangecyberdefense.com/fr/solutions/services-manages/micro-soc-poste-de-travail/edr ne le savent pas, un *hack* est le détournement d'une technologie de son objectif initial, et aucunement un acte malveillant *a priori*. Tel que l'indique A. Guitton dans *Hackers. Au cœur de la résistance numérique* (Éd. au Diable Vauvert, 2013), le *hacker* est une personne s'attachant à "comprendre le fonctionnement d'un mécanisme, afin de pouvoir bidouiller pour le détourner de son fonctionnement originel" ou de façon répétée dans le début de son ouvrage, "*comprendre, bidouiller, détourner*". La définition au début de R. des Bois, "Lève toi et code : Confessions d'un hacker" (La Martinière, 2018) aurait aussi pu être en relation avec l'introduction de cet article en affirmant "voir quelque chose de cassé et ne pas pouvoir s'empêcher de ne rien faire, soit tu l'exploites, soit tu le répares, mais impossible d'ignorer ce dysfonctionnement et le laisser comme tel." Quel domage que l'auteur ne se souvienne pas de cette définition dans la suite de son discours, ouvrage sans intérêt après cette courte mention pertinente, dont le protagoniste se réduit à acheter sur le web des failles de sécurité pour les exploiter contre des utilisateurs crédules et incompétents sans faire preuve de créativité technique.