USB programming under GNU/Linux: application of the FX2LP for a software defined radio receiver dedicated to satellite navigation signals (1/2)

Jean-Michel Friedt, associate professor, University of Franche-Comté in Besançon (France),
November 5, 2024

While USB is often tackled as a bus emulating a serial port, fully exploiting its bandwidth requires using the most appropriate available interfaces, particularly Human Interface Device (HID) and Bulk transfers. We aim at understanding the USB bus exposed by the Linux kernel in order to make the most of the available bandwidth and to apply this knowledge by designing a software-defined radio receiver dedicated to receiving satellite navigation signals (GNSS) in the L band (1–2 GHz) using the MAX2771 frontend. We demonstrate the proper functioning of the circuit with the acquisition and processing of signals from various GNSS constellations in Medium Earth Orbit (MEO) and Low Earth Orbit (LEO) Iridium, recorded with a bandwidth of up to 44 MHz.

Our objective is to create a software-defined radio receiver for satellite navigation signals (GNSS for Global Navigation Satellite Systems) and other satellite signals transmitted in the 1–2 GHz range known as L band. The most recent modulation modes (European Galileo, American L5) occupy up to 41 MHz of bandwidth (for Galileo E6, see `https://gssc.esa.int/navipedia/index.php/Galileo_Signal_Plan`), requiring "fast" transfer of data between the acquisition interface and the computer in charge of storing measurements -— without the ambitious aim of real-time processing at first, but still wanting to capture all samples without losing a single one. To achieve this aim, we must learn to handle various communication modes over the USB bus and their implementation in a Cypress component, the CY7C68013A also known as FX2LP, and somewhat abusively referred to as EZ-USB. This component embeds a venerable 8051 [1] processor core to configure various communication interfaces: the 8051 is too small, with its 1.5 accumulators (A and B) and 8 (R0–R7) 8-bit registers, for GCC to be used, so the code will be compiled using the Small Device C Compiler (SDCC). During these developments, we will take inspiration from the functional system proposed by Tomoji Takasu of Tokyo University of Marine Science and Technology, also the author of RTKLib, famous among users seeking centimeter resolution from their GPS receiver [2]. This system is called PocketSDR and is available at `https://github.com/tomojitakasu/PocketSDR`. Although the circuit is freely distributed in KiCAD format on this repository, the author's choice has been to work in two different frequency bands -— referred to as high (L1) and low (L2, L5) in the nomenclature of the MAX2771 component responsible for receiving GNSS signals -— and to incorporate the parallel to USB communication in the FX2LP interface. Since the EZ-USB FX2LP can be obtained assembled and ready to use for just under 5 euros on AliExpress, when the chip alone costs nearly 20 euros at Farnell (order code 1269134), we chose to work on a ready-to-use circuit `https://fr.aliexpress.com/item/1005006134347046.html` and to add the acquisition interface based on MAX2771, initially in the form of the evaluation board sold by Maxim IC (now Analog Devices), the manufacturer of the component. Ultimately, Tomoji Takasu uses proprietary software to convert his source code into binary for the 8051, a solution that is obviously unacceptable and must be remedied by converting his sources to SDCC, which will not be without pain as we will see below. After these developments, we will first address the Iridium Low Earth Orbit constellation, which, although transmitting at frequencies just above the GNSS band occupied by the Russian GLONASS system, remains within the frequency range accessible to the MAX2771 with a signal much easier to detect for verifying the proper functioning of the acquisition and communication interfaces, before finally decoding GPS.

The outline of the two articles reporting these explorations is as follows:

1. Starting from a MAX2771 evaluation board provided with proprietary software communicating in HID under MS-Windows, we identify the communication protocol and implement it in Python under GNU/Linux based on `libusb`,

2. Discovering that said evaluation board is not designed to acquire and transmit data but only to configure the device (?!), we take over the digital part by eliminating the originally supplied microcontroller and replacing it with a FX2LP,

3. After developing the first basic program (blinking an LED) on the microcontroller equipping the FX2LP to validate understanding of the compiler, the associated library, and communication tools between the PC and the microcontroller to execute the binary compiled on target...

4. ... we port the software implementation (bitbang) of the SPI (Serial Peripheral Interface) bus proposed by PocketSDR to the SDCC compiler, and learn to communicate via Vendor Requests to execute commands through the USB bus.

5. Finally, we complete the porting of the firmware provided by PocketSDR by adding Bulk communications on USB, and validate the various communication rates based on the clock configuration registers locking the analog-to-digital converters.

6. The correct functioning of the setup is validated on the "strong" signals from Iridium satellites in Low Earth Orbit, then on signals below the thermal noise of navigation satellites.

7. In the next article, we will address the design and realization of a dedicated daughterboard embedding two MAX2771 for differential measurements of radio frequency signals acquired by FX2LP and transmitted via USB Bulk to the PC for post-processing, particularly to compensate for deficiencies in cross-talk between signals carried by communication wires that are too long in the first prototyping circuit.

8. Finally, we will add the ability to correct the frequency source that clocks the MAX2771 to correct its deviation from the nominal GPS frequency, by adding new Vendor Requests to program this new peripheral added to the bus resembling SPI.

The only modification when using the MAX2771 evaluation board is to replace the default 16.368 MHz oscillator with a 24 MHz oscillator in order to use the configuration files proposed by PocketSDR as is. We will see in section 2 how to recompile the firmware and thus change PocketSDR's configuration to support an oscillator clocked at 16.368 MHz.

Thus, in the next episode, even if it means replacing the microcontroller of the evaluation board worth over 400 euros, we will continue by proposing our own implementation of a daughterboard equipped with two MAX2771 adapted to a low-cost (5 euros) board equipped with a FX2LP, finalizing the approach on a circuit costing less than a hundred euros all-inclusive (MAX2771, FX2LP and multi-band GNSS or Iridium antennas). In particular, we will eliminate unwanted electromagnetic radiation when signals at several MHz flow over unshielded wires of about ten centimeters long, inducing corruption of digital signals and noise in baseband during the analysis of the signals transposed by the local oscillator before analog-to-digital conversion, as we will see in conclusion.

1 The USB Bus

Historically, a computer communicates point-to-point via a protocol that serializes data over time rather than communicating it in parallel -— the latter approach is limited in communication throughput by inductive coupling between adjacent wires and the size of the wire harness needed to carry all the bits. RS232 is an example of a serial bus, classified as asynchronous since the communicating parties do not share an clock. In the same vein, SPI is a synchronous serial bus since a master distributes a clock signal to its slaves. The venerable RS232 protocol, still widely used in embedded systems, is rarely available on current personal computers, and USB-RS232 interfaces are plentiful. Thus, the USB CDC communication protocol -— Communications Device Class —- deceives the host (the computer) into thinking the device communicates as a RS232 interface when in reality the information is transmitted over the USB bus. This mode of communication is very convenient as it allows the use of the tools associated with RS232 -— `minicom` and `screen` to name just a few—but does not allow taking advantage of all the throughput available on USB. In LUFA for example (`http://www.fourwalledcubicle.com/LUFA.php`), an Atmega32U4 easily configures as CDC by

```
#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(); }
}
```

and connecting the microcontroller programmed by this sequence of instructions to a computer's USB bus reveals an interface `/dev/ttyACM0` accessible via `minicom -D /dev/ttyACM0` without the baud rate mattering here. A few throughput measurements, for `N` ranging from 64 to 1024 in powers of two, indicate via

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

a file size on the order of 965 ± 5 kB thus throughput around 100 kB/s, in agreement with tests from `https://www.pjrc.com/teensy/benchmark_usb_serial_receive.html` for the Atmega32U4 which equips an Arduino Leonardo board. According to this same site, a slightly more powerful microcontroller should reach MB/s, still far from the tens of MB/s we aim for in a software-defined radio application. We therefore need to turn toward optimized use of the USB bus and not settle for merely exposing a virtual serial port.

Many "recent" devices do not exploit CDC but expose a richer interface, and this is the case for instance with the microcontroller that equips the MAX2771 evaluation board (Fig. 1). Indeed, having acquired this board several years ago, it became a very nice paperweight when I realized that neither GNU/Linux nor Wine could communicate with it through the proprietary software provided by Maxim IC. Since a 450-euro paperweight acquired with taxpayer money is questionable, we investigate its communication protocol and the exchanges between the microcontroller with which the evaluation board is equipped and the PC. Of course, without access to the source code of the firmware running on said microcontroller, we can only attempt retro-engineering by probing the USB bus when the original software communicates with the evaluation board: this software runs from a VirtualBox virtual machine equipped with MS-Windows and running the proprietary software available on the manufacturer's site [3].

Upon connecting the USB interface of the microcontroller, Linux informs (`dmesg`) that

```
[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/input
```

indicating it is a *Human Interface Device* whose communication protocol we must understand.

1.1 `usbmon` for reverse-engineering a HID interface

HID is the interface for many relatively low throughput USB peripherals, including mice and keyboards, and its analysis is the subject of numerous studies, notably `https://www.baeldung.com/linux/usb-sniffing`, which teaches us that the Linux kernel offers a USB bus debugging mode to display the bytes exchanged. Indeed,

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

creates pseudo-files in `/sys/kernel/debug/usb/usbmon/Xu`, with `X` being the bus number, providing access to kernel resources. After identifying on which USB bus the device is connected using `lsusb` such as

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

we will use the first argument (here 1) in place of `X` to probe messages transmitted on this bus. Note that it may be wise not to connect the microcontroller of the evaluation board to the same USB bus as

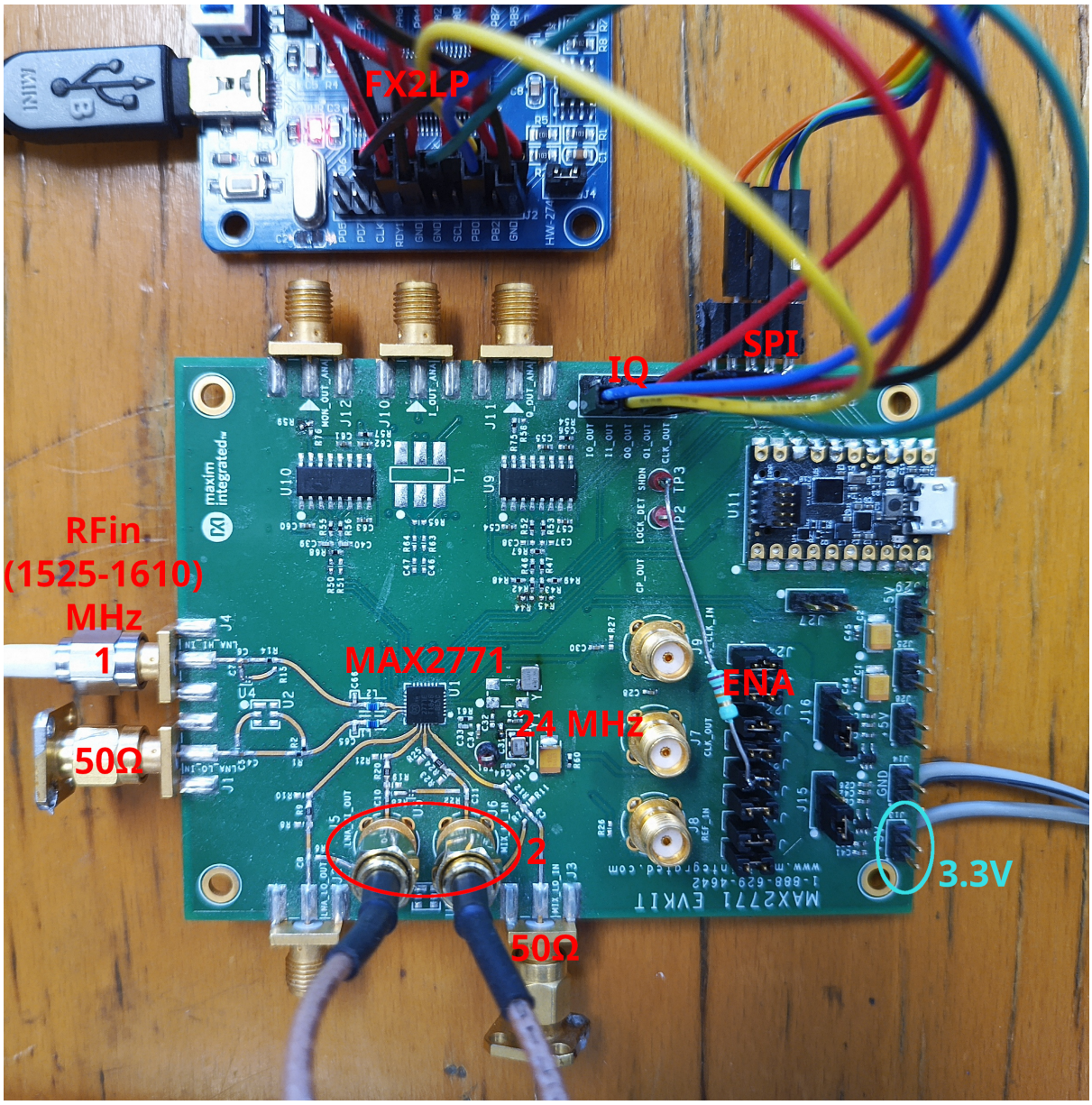


Figure 1: The evaluation board of the MAX2771 from Maxim IC, source of all our troubles, with its microcontroller (top right) which can only communicate with proprietary software running under MS-Windows, and which ultimately proves capable only of converting USB commands into SPI. This board is already equipped with SPI and IQ (terminal next to the microcontroller) connections that we will redirect to the EZ-USB in order to correct these deficiencies. Only the 3 V supply (bottom right, in cyan) is necessary to power the board, the symmetrical power supplies serve only for the operational amplifiers that are not used. To test the upper GNSS band, a signal or an active antenna will be connected to the left connector (mark 1) and we will connect the output of the low noise amplifier (LNA) to the input of the mixer (red ellipse 2). A pull-up resistor to the power exits the MAX2771 from its sleep mode (ENA) when the microcontroller of the board is not powered.

a mouse communicating via this interface to avoid being polluted by its messages every time we move the cursor of the graphic interface. At worst, one can `|grep YYY` with `YYY` being the device number to filter.

As it stands, the microcontroller is not talkative. However, if we execute the proprietary software provided by ADi to communicate from MS-Windows [3] in VirtualBox, we obtain a series of messages provided the user belongs to the group `vboxusers` and the Extension Pack is installed (USB support for

VirtualBox), in the form:

```
# 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...
...
```

thus the software on MS-Windows periodically asks the microcontroller if it is present. The microcontroller acknowledges each request (message `c9` followed by 0s). In addition to these periodic requests, the proprietary software allows configuring the registers of the MAX2771, and in this case, a series of messages starting with `c2` indicates the register number and the value to be stored there. Once the protocol is identified, it remains to implement it, for example by taking inspiration from `https://github.com/david0/durgod-keymapper/blob/master/remap.py` which teaches how to communicate a message to a HID device whose Vendor ID and Product ID (VID:PID -— here `1234:0006` as previously indicated by `lsusb`) are known, in the form:

```
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)                  # sending the message C9 00 00 ...
resp=device.read(64, timeout_ms=500) # receiving the response
resp=bytearray(resp).rstrip(b'0x00');
print(tohex(resp))
```

to send `c9` followed by zeros [1], and indeed

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

indicating that the microcontroller acknowledges our request made of `0xc9` followed by 31 zeros (`b"\xc9".ljust(31, b"\x00")`). The path is therefore clear; all that remains is to understand the various commands of the proprietary software and implement them in Python. However, one point raises a question: no button on the proprietary communication software allows acquiring data, only to communicate through the SPI bus configurations of the MAX2771 registers. Having sought the assistance of Maxim engineers, we received confirmation that the communication of the data acquired by the MAX2771 was planned... but never implemented. Thus, we acquired a 450 euros USB-to-SPI converter! There is no need to wire resistors R31/R32 and R44/R45 (I0/I1 to host and Q0/Q1 to host) on the evaluation board, as the microcontroller won't know what to do with these IQ signals representative of the converted electrical signal into a digital signal by the MAX2771. We must therefore manage alone to connect a parallel-USB interface to the terminal J26 carrying IQ signals from the GNSS signal receiver to transmit them to the PC, and it is here that the Cypress CY7C68013A comes into play.

[1] All codes cited in this article are available at `https://github.com/jmfriedt/max2771_fx2lp/` including this `https://github.com/jmfriedt/max2771_fx2lp/blob/main/Maxim_EvalBoard/max2771.py`

I0/I1 and Q0/Q1: analog-to-digital conversion on 2 or 3 bits!
One might be surprised by the nomenclature I0/I1 and Q0/Q1 which suggests that the complex signals $I + jQ$ are encoded on just two bits, or one sign bit and one value bit. This encoding is common in the analysis of GNSS signals which sit 20 dB below the thermal noise (see section 3), for which a large number of bits would only code the noise, not the signal. The power of the pseudo-random code (CDMA) encoding the GNSS messages is that correlation brings the signal out of the noise by a factor equal to the compression ratio (pulse compression ratio given by the product of bandwidth multiplied by the duration of the code, or in digital terms the number of bits of the code). With a 1023-bit code for GPS L1, the compression gain is $\log_2(1023) \simeq 10$ bits, so the three bits become 13 bits after correlation by the known code, or 16 bits for codes ten times longer for GPS L5. To our great surprise, such few bits of coding will still be sufficient to decode Iridium, which does not benefit from a CDMA coding compression gain but relies on a much stronger signal transmitted by its satellites in Low Earth Orbit. Thus, the MAX2771 can either provide the complex measurements (real and imaginary parts) in baseband in the form $I + jQ$ with I and Q each coded on 2 bits, or in the presence of an intermediate frequency acquire only the real part (thus even spectrum) coded on 3 bits according to $I1, I0, Q1$ (thus the least significant bit on $Q1$ even if the information carries on a real value), and the user will be responsible for performing frequency transposition numerically (multiplying by a local oscillator frequency equal to the intermediate frequency) to produce complex values in baseband.

Indeed, Tomoji Takasu being less incompetent than Maxim IC -— or, more particularly, more motivated than the mercenaries occasionally paid to develop the evaluation board software – provides the solution with the EZ-USB FX2LP: this component will connect to the IQ pins of the terminal and its associated clock (synchronous communication) to translate the measurements proposed in parallel format by the MAX2771 to a USB stream. But to get there, one must program the embedded 8051 microcontroller in the CY7C68013A to configure its interfaces...

1.2 EZ-UZB FX2LP

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

The EZ-USB is not a recent development [5], but unfortunately, most projects related to it on GitHub are over ten years old and have a hard time functioning today. Fortunately, the library associated with `sdcc` [6] for the FX2LP, fx2lib at `https://github.com/djmuhlestein/fx2lib`, is still operational, and a derived version of `fx2lib` is maintained by `sigrok` [7].

Two tools for transferring the cross-compiled firmware from PC to the microcontroller are fxload at `https://github.com/mbed-ce/fxload`, which we prefer over `https://github.com/esden/fxload` or the Debian GNU/Linux binary package from Sourceforge, considering the recent improvements, to transfer the program to RAM or EEPROM, and faster at first will be `cycfx2prog`, which is also available as a Debian binary package. Equipped with these tools, the first goal is to blink an LED to validate the understanding of the 8051 and programming tools. Unfortunately, selecting a cheap development board does have some consequences: the 56-pin version of the CY7C68013A does not route the asynchronous communication interfaces (UART, compatible with RS232, only available on boards with more pins) [8, page 18], so we will have to make do with this LED as a communication mode until we master USB interfaces.

1.3 Blinking LED

This first example relies on `https://github.com/sidd-kishan/fx2lp-blinky` due to the lack of documentation on the operation of the FX2LP registers, the core of the 8051 being surrounded by many more peripherals than the original microcontroller. The description of the registers in the technical documentation (including their memory location) is rather succinct.

The FX2LP development board is equipped with two LEDs that can or cannot be activated depending on whether a jumper connects them to the power supply (GPIO as current sink, the board schematic at [9]). These two LEDs are controlled from port A and are connected to pins 0 and 1. Thus, the trivial program

```
#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
 }
}
```

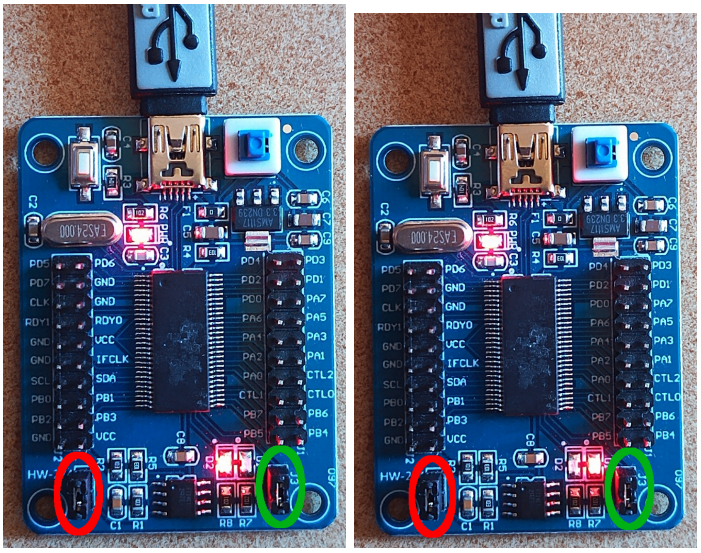


FIGURE 2 – Two photographs of the FX2LP development board while the left LED is on and the right LED is off (left) or the opposite (right), next to the jumper circled in green that connects them to the power supply. The jumper circled in red is used to disable the memory address of the EEPROM at startup to allow it to be reprogrammed (see section 1.4).

initializes two bits of port A as output (register `OEA`, bit set to 1 for output) and then manipulates the register defining the state of the output pins `IOA`, alternating the LED on and off every second -– the left LED lit in the left photograph and the right LED in the right photograph in these illustrations. This program, compiled and linked with the FX2Lib library from `https://github.com/djmuhlestein/fx2lib`, which we have previously compiled with `make` to generate `lib/fx2.lib`, is converted into an Intel-compatible hexadecimal file (extension `ihx`) by `sdcc`.

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

to implicitly generate `mainPA.ihx`, which is executed from RAM of the 8051 with `sudo cycfx2prog prg:mainPA.ihx run`. We note in the two figures accompanying the code that both jumpers are in place, the red one to boot the FX2LP in bootloader mode and load the executable in Intel hexadecimal (`ihx`) into RAM, and the green one to connect the LEDs to power. Various `Makefile` for FX2LP add linker options like `--code-size 0x1c00 --xram-size 0x0200 --xram-loc 0x1c00` but the default values seem sufficient not to have to specify them.

In doing so, we have programmed the 8051 like any general-purpose microcontroller, without taking advantage of any of its original features. Indeed, to blink another external LED that we connect between PD7 and ground, for example, the code

```
#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);
 }
}
```

(Fig. 3) manipulates the single bit PD7, while leaving the manipulation of the data register `IOD` associated with port D commented out, always after setting the pin as output by manipulating bit 7 of `OED`. Indeed, Maxim IC reminds us of the fundamentals of SDCC [10], particularly the ability of the 8051 to address a single bit (SBIT) without manipulating the entire register that contains it (SFR for *Special Function Register*). This functionality was clear during assembly programming but seems somewhat obscure when transitioning to the C language with macros like `__sbit __at(0xB0+7) PD7;` in `include/fx2regs.h` of FX2Lib to call the specific `SBIT` assembly instruction for the 8051.

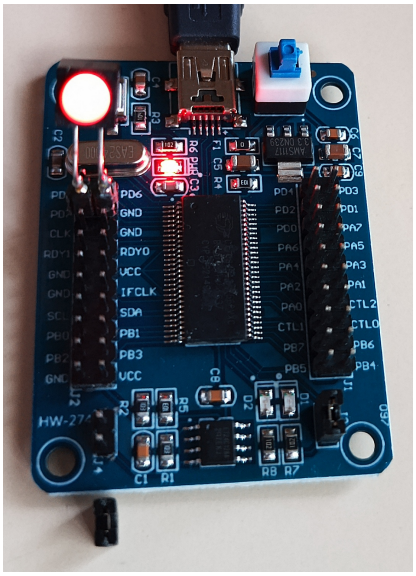


FIGURE 3 – Example of blinking LED on PD7. Notice the jumber, bottom left, which was removed to allow for identifying the EEPROM and execute the code stored in non-volatile memory with `fxload`.

Once this program is compiled as before, this time the executable is placed in non-volatile EEPROM memory for execution on power-up with

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

which requires the `Vend_Ax.hex` firmware to communicate with the EEPROM and identify the VID and PID of the FX2LP in bootloader mode (`04B4:8613` as indicated by `lsusb`). Notice that with older versions of `fxload`, the command
`sudo fxload -D /dev/bus/usb/001/035 -I main.ihx -c 0xc2 -s Vend_Ax.hex -t fx2lp`
was much more annoying since the pseudo-file in `/dev/bus/usb` changes each time the FX2LP is powered off and on, with its identifier incrementing each time. Beware though, we can write once to non-volatile EEPROM memory, but not a second time. The solution to this problem will be provided below (section 1.4). In the meantime, we recommend testing the proper functioning of the software in RAM using `cycfx2prog`.

The complete source code of this example is available at `https://github.com/jmfriedt/max2771_fx2lp/tree/main/FX2LP/LED_blink`.

Having validated the compilation and programming of the component by making an LED blink, we can tackle the heart of the matter, communication over the USB bus.

1.4 Reprogramming the EEPROM

We have noted that once the EEPROM is flashed, we cannot place a second program different from the first. Indeed, the first transfer of a program from the PC to the EEPROM succeeds with the message

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

but upon the second attempt, the message is instead

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

indicating failure. The problem lies with the expectations of `fxload`, which are no longer met once the EEPROM is flashed for the first time, as we will explain.

According to the circuit diagram containing the FX2LP [9], the jumper J4 furthest from the LEDs must be in place to put the FX2LP into bootloader mode to allow EEPROM programming. In doing so, the FX2LP's capability to find a non-volatile memory containing valid firmware is invalidated at startup, forcing the 8051 to enter bootloader mode. In the implementation of the circuit that we are using, this goal is achieved by shorting the least significant bit of the EEPROM address to ground, thus setting its value to 0. However, this implies that the jumper must be *removed* when programming the EEPROM, since the FX2LP datasheet specifies that the expected address for the storage device must be 0x51 on the I^2C bus (in the naming convention which only keeps the 7 most significant bits of the address et omits to mention that the two addresses broadcast on the I^2C bus are 0xA2 or 0xA3 for a write or read transaction respectively). In the proposed configuration, the jumper imposes a zero potential on the least significant bit of the EEPROM, thus defining an address on the I^2C bus of 0x50, which cannot be recognized during programming. Therefore, we must absolutely *remove the jumper before flashing the EEPROM*, otherwise the component cannot be detected on the I^2C bus. To convince oneself, we follow the guidelines from [11], namely:

1. Transfer the program `Vend_Ax.hex` into volatile memory, available for example in the `fxload` archives, by `sudo cycfx2prog prg:Vend_Ax.hex run` (or if preferred to stay with `fxload`, with `sudo fxload load_ram --device 04b4:8613 --ihex-path Vend_Ax.hex -t FX2LP`)

2. Execute the program that launches the Vendor Request commands 0xA2 to read the contents of the EEPROM following

```
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. If we leave the jumper in place, the response `b'cdcdcdcdcdcd...` indicates that the EEPROM has not been read, as its address does not match that expected by `Vend_Ax`.

4. If we remove the jumper, then the response `b'c2b404138605a04203f20000` indeed starts with the expected 0xC2 byte for booting the internal logic as documented on page 8 of the technical manual [8] by indicating that "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)." followed by VID and PID of the component encoded in a *little endian* format, hence "upside down" for a western reader used to read from left to right: b4041386 is interpreted as 04b4:8613.

According to the source code of `fxload` at `https://github.com/mbed-ce/fxload/blob/master/src/ezusb.c#L674-L676`, it expects a response of 1 to the `GET_EEPROM_SIZE` request handled by `Vend_Ax` and otherwise refuses to reflash the EEPROM. However, issuing the command to a FX2LP executing `Vend_Ax` returns the value 0 and thus `fxload` refuses to continue programming.

In addition, the first byte of the EEPROM contains 0xC2 or 0xC0 depending on how the USB identifier is configured. The alternative to remove the jumper to switch to bootloader mode also fails since the FX2LP sees an EEPROM whose first byte contains 0xC2 and thus executes its code. The solution, albeit inelegant, that we found to reflash the FX2LP's EEPROM is to execute a program from RAM, thus loaded by `cycfx2prog`, to overwrite the first bytes of the EEPROM with 0xFF, thus forcing the microcontroller with the jumper removed to execute its bootloader in the absence of valid firmware. To achieve this result, we load in RAM (`cycfx2prog`) the `Vend_Ax.hex` with `sudo cycfx2prog prg:Vend_Ax.hex run` and benefit from its Vendor Request 0xA9 to write to EEPROM as described at [11], and hence overwrite the first bute to force the execution of the bootloader which will allow reflashing the whole 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))
```

The content of the EEPROM has now been overwritten, and booting the FX2LP without jumper (allowing thus the identification of the EEPROM chip on the I^2C bus) allows for reflashing the non-volatile memory content.

Denis Bodor points out that a similar functionality is offered by `http://www.triplespark.net/elec/periph/USB-FX2/eeprom/` that contains `erase_eeprom` with similar capabilities, if only after replacing the I^2C 0xA2 EEPROM address ("*assumes that A0 is tied to positive supply and A1,A2 of the EEPROM are tied to ground* ") with A0 depending whether the jumper is set or not. This program erases all the content of the EEPROM instead of just erasing the first byte, the only necessary condition to make `fxload` work again to reflash the already flashed EEPROM.

1.5 SPI communication under USB control

In order to learn incrementally how to approach the USB bus from the host's perspective (the PC under GNU/Linux) and the device (the FX2LP), we will initially rely on the pre-compiled firmware from PocketSDR available at `https://github.com/tomojitakasu/PocketSDR/blob/master/FE_2CH/FW/v2.1/pocket_fw.hex`, flashed into EEPROM when the microcontroller has been started with the corresponding jumper in place, but removed at the time of programming (see section 1.4). Once the jumper is removed, we know that the PocketSDR software is the one currently executing, since `lsusb` now indicates

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

with this new `VID:PID` values equal to `04b4:1004` defined in the firmware. It is now important to understand how the PocketSDR firmware, whose source code is available, facilitates communications between the FX2LP and the MAX2771 following a protocol abusively qualified as SPI compatible, since MISO and MOSI are merged in a unique "SDATA" signal whose impedance changes whether the transaction is a read or a write operation (exactly the reason we hate I2C whose transaction direction cannot be deduced from the oscilloscope or logic analyzer trace).

We learn from the header of `PocketSDR/FE_2CH/FW/v2.1/pocket_fw.c` that several *Vendor Requests* allow for, similar to `ioctl()` system calls in the Linux kernel modules, to associate arbitrary command codes to some operations. In the case of PocketSDR, command 0x40 returns the version and status of the firmware, 0x41 reads a register from the MAX2771 and 0x42 writes to a register (Fig. 4).

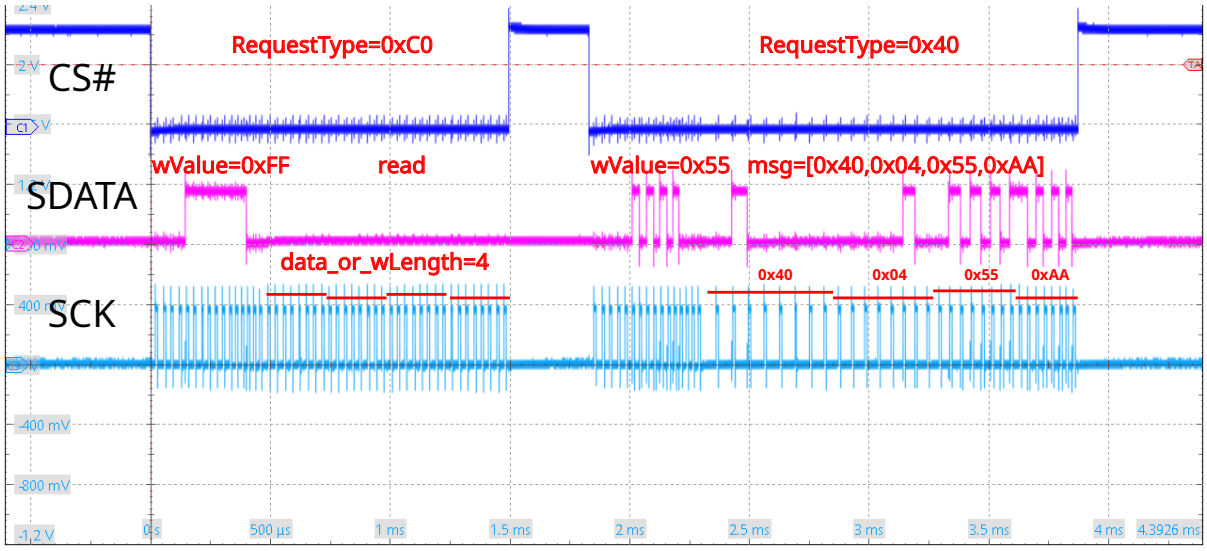


Figure 4: Signals generated by the software implementation of SPI (despite grouping on a same wire the MOSI and MISO signals) for reading and writing, depending on the transmitted *Vendor Request* code. From top to bottom the Chip Select CS#, the data bus SDATA and the clock bus SCK.

It is thus necessary to learn how to send *Vendor Requests* from GNU/Linux to become familiar with these exchanges implemented in the precompiled firmware of PocketSDR, with a view to ensuring that our implementation of the compatible `sdcc` firmware will meet the same goals, namely transactions on the SPI bus. The result is surprisingly easy to achieve using the Python3 USB library once one understands the meaning of the 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'
```

We learn from `https://www.beyondlogic.org/usbnutshell/usb6.shtml` that the `wValue` and `wIndex` arguments of the `ctrl_transfer()` method depend on the nature of the transactions, but to simplify, the first argument `bmRequestType` is 0x40 for a write and 0xC0 for a read. Then follows `bRequest`, the code

for the *Vendor Request* corresponding to the PocketSDR firmware, followed by a word `wValue` that will be decoded by the 8051 of the FX2LP to identify the register of the MAX2771 to be reached (least significant byte) and, in the case of the "official" PocketSDR circuit equipped with two MAX2771, which of the two components to activate via its appropriate Chip Select (high byte). Finally, `wIndex`, the interface number, is always zero, and if the transaction is a write, the last argument is an array of bytes transmitted over the SPI bus; otherwise, it is the number of bytes to read. If we issue the command 0x40 (read status) in read mode (`bmRequestType=0xC0`) with the previously suggested Python code, we then receive `b'105dc0010100'` which corresponds to the source code `PocketSDR/FE_2CH/FW/v2.1/pocket_fw.c`, namely at the beginning the firmware version 0x10, followed by two bytes containing the frequency of the oscillator in kHz (24000 in our case, so 0x5DC0 in hexadecimal), followed by various statuses that are not important for now. The transaction is therefore correctly carried out to read a value from the microcontroller via USB.

Now, to trigger transactions over USB, we send the command (`bRequest`) 0x41 (read) or 0x42 (write) followed by the register number of the MAX2771 in `wValue` and either the sequence of bytes to write or the number of bytes to read. We observe on the oscilloscope that the SCLK, SDATA, and CS# signals behave as expected in an SPI link (Fig. 4). Although this is a software implementation (*bitbang*) of SPI in the 8051, the period and duty cycle of SCLK, which can clearly be seen not to be constant on the lower curve of Fig. 4, are of no importance since it is a synchronous protocol where only the edges matter. When we read a result, we note in the source code of `PocketSDR/FE_2CH/FW/v2.1/pocket_fw.c` that the case `VR_REG_READ` concludes with `EP0BCL=4;` thus returning 4 bytes or the 32 bits contained in each register of the MAX2771. We put into practice by

```
#!/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))
```

which accesses the register 5 of the MAX2771 or the fractional part of the division of the oscillator in the phase-locked loop (PLL), in write and read, to check that the information has indeed been stored in the register.

1.6 Configuration of *Vendor Requests* by `sdcc`

We are convinced that we know how to exchange *Vendor Request* messages from GNU/Linux to trigger an action on the FX2LP: this has been proven with the precompiled firmware using the proprietary Keil compiler. We must now reproduce this behavior with `sdcc`. To do this, we start from the example `bulkloop/` of `fx2lib`. We identify the function `BOOL handle_vendorcommand(BYTE cmd) {...}` that seems to meet our needs, particularly with the handling of a command named `VC_EPSTAT` identified by code 0xB1: we already mentioned that like `ioctl()` in the Linux kernel, these codes assigned to commands are arbitrary and must be consistent with the code emitted by the application. We therefore copy the sequence of *Vendor Requests* from the PocketSDR firmware from the function `BOOL handle_req(void) {...}` of `PocketSDR/FE_2CH/FW/v2.1/pocket_fw.c`, replacing the nested conditions with a somewhat more readable `switch ... case` sequence. For example, the command `VR_STAT` of code `0x40` fills the first 6 bytes of the buffer `EP0BUF` with values that identify the firmware version or the frequency of the crystal locking the MAX2771, and returns (`EP0BCH = 0; EP0BCL = 6;`) this buffer to the calling function. On the Python side, the counterpart of this request is the one we had seen earlier, but which we understand now expects only 6 bytes in return

```
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))
```

so a read request (0xC0) of the *Vendor Request* 0x40 (`VR_STAT`) and the response is `b'105dc0010100'` indicating 0x10 the firmware version, corresponding to `EP0BUF[0] = VER_FW;` in the PocketSDR firmware source code, since `#define VER_FW 0x10`, followed by 0x5DC0 since in PocketSDR

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

with `#define F_TCXO 24000`. Two input pin states follow that we do not control at the moment. The complete source code for `sdcc` can be consulted at `https://github.com/jmfriedt/max2771_fx2lp/blob/main/FX2LP/python_access_USB` which partly benefits from the fact that both Keil and FX2Lib for SDCC rely on constant files sharing the same nomenclature and are therefore easily interchangeable.

Thus, we are now able to receive messages from the FX2LP via *Vendor Requests*, and we continue to plagiarize the PocketSDR code by copying from `PocketSDR/FE_2CH/FW/v2.1/pocket_fw.c` the software implementation (*bitbang*) of the SPI bus since the syntax of `digitalRead()`, `digitalWrite()` which access the pins, but also `write_sclk()`, `write_sdata()` and `write_head()`, are directly compatible with SDCC, the FX2Lib library having the good taste to use the same constants as the Keil compiler as we see by comparing `cypress/dscr.a51` from `PocketSDR/FE_2CH/FW/v2.1` for Keil and `fx2lib/fw/dscr.a51` for SDCC. One pitfall: PocketSDR defines a function `delay()` as a loop of `cyc` iterations, while FX2Lib offers a function of the same name but whose prototype is `void delay(WORD millis);` and thus takes an argument in milliseconds (and not in clock cycles).

It is clear that the FX2Lib function will be considerably slower than that of PocketSDR, and indeed our observations on the software clocked SPI bus showed very slow variations in the clock and data bus. A new delay function with a different name that performs a quick countdown allows us to almost identically replicate the behavior of PocketSDR thanks to the program compiled by SDCC proposed in `https://github.com/jmfriedt/max2771_fx2lp/tree/main/FX2LP/python_access_USB`.

We have reached the point where we are able to communicate commands from the PC to the FX2LP via a Python program, which the 8051 interprets these commands and produces the appropriate pattern on the SPI bus, thus programming the registers of the MAX2771 and reading back the contents. It remains now "only" to capture the bytes placed on the parallel bus by the MAX2771 and clocked by the IFCLK signal in order to fill the FIFO memory that periodically empties on the USB bus in a bulk transaction. However, a few points of "detail" still need to be resolved before we get there.

1.7 Communication of acquired data from MAX2771 via FX2LP in USB bulk

Starting again from the `bulkloop/` example of FX2Lib, the final step to complete the porting of PocketSDR firmware to `sdcc` while mastering the bulk mode transfers over USB is less painful than it seems since once again FX2Lib exploits the same nomenclature as the Keil compiler and therefore only requires us to take over the USB configuration function from PocketSDR named `void setup(void) {...}` which configures all endpoints as well as `static void start_bulk(void) {...}` called at the end of `main()` (as well as `stop_bulk(void) {...}` which can be called by a *Vendor Request*) to see all communication functions. We first eliminate all functions related to accesses to non-volatile memory over I^2C: the equivalents of Keil functions such as `EZUSB_WriteI2C()` and `EZUSB_ReadI2C()` necessary for `write_eeprom()` and `read_eeprom()` of PocketSDR exist in FX2Lib in the form of `eeprom_read()` and `eeprom_write()` but will not be necessary for the initial tests.

Among the other syntactic subtleties, the 16-bit addressing space of the 8051, `xdata` in Keil, becomes `__xdata` in SDCC, and the assembly mnemonics included in C code are prefixed with `__asm` followed by the instruction, for example `nop`, and end with `__endasm`. Obviously, the interrupt vectors are named differently, for example, `void ISR_Highspeed(void) interrupt 0 {. }` from Keil becomes `void hispeed_isr(void) __interrupt (HISPEED_ISR) {...}` in SDCC, but since the `bulkloop/` example is functional, this last point concerns us little.

At the end of this example, we are able to compile firmware offering all the original functionalities, this time compiled by `sdcc`, including the *Vendor Requests* for communicating configurations over the SPI bus as well as the transfer of acquired data from the MAX2771 in USB Bulk transactions.

1.8 Validation by finalizing (almost) wiring of the MAX2771 to FX2LP

We are beginning to understand how all this works, so we now wish to carry out real communication between the MAX2771 and FX2LP to retrieve data. To do this, we wire the AliExpress EZ-USB board to the evaluation board of the MAX2771 using 5 wires connecting I0, I1, Q0, and Q1 to PB0 to PB3 respectively and especially DCLK (CLKOUT of the MAX2771) to IFCLK/PE0 of the FX2LP, as well as the clock, data, and activation (CS#) SPI communication pins to PD2, PD3, and PD0 respectively. and nothing works. Indeed, we also need to wire the two signals that define the direction of filling the FIFO of the FX2LP, either in read or write from the PC, in our case following the diagram at `https://github.com/tomojitakasu/PocketSDR/tree/master/FE_2CH/HW/v2.3` by connecting `RDY0/SLRD` to power and `RDY1/SLWR` to ground. We restart a measurement. and still nothing. Impossible to receive any USB frame from the FX2LP on the PC.

Again, Tomoji Takasu has the answer since he experienced the same troubles, which he documented at `https://blog.goo.ne.jp/osqzss/e/d86df04de96123fd5c73bbb6db6e8bc5` (for those less adept in Japanese, you can use Google Translate): some Chinese development boards of the FX2LP have inverted the silkscreen of RDY0 and RDY1 which define, by polarizing them to ground or to the power supply voltage, the direction of communication. Our board (Fig. 5) is subject to this error, and reversing the power and ground signals between RDY0 and RDY1 to regain the configuration of the PocketSDR scheme ends up allowing communication. Baptiste Maréchal (SpacePNT, Neuchatel) amusingly points out that various illustrations of this board on the same Amazon site are incoherent in their silkscreen among the various photographs promoting the product, but one had to know it to identify the error.

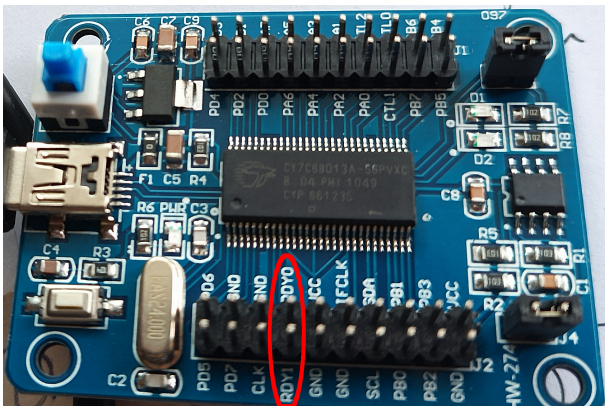


Figure 5: The board equipped with the FX2LP ready to be used, yet impossible to communicate in USB Bulk despite code compiled by Tomoji Takasu using the proprietary Keil compiler which we can only rely on. The problem comes from a silkscreen error on the Chinese circuit leading to a faulty configuration of the communication direction, as informed by the Japanese author of PocketSDR.

Once this error is corrected, we finish verifying the communication functions by using `PocketSDR/app/pocket_dump/po` to receive data. In doing so, we will not (yet) have learned to receive bytes from the FIFO to the PC ourselves, but we will keep this understanding for the next section. Furthermore, one somewhat "surprising" point is that regardless of the configuration we suggest to the MAX2771 through `PocketSDR/app/pocket_conf/pocket_conf`, the communication rate remains the same since `pocket_dump` indicates

```
$ 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
```

thus always a communication of a real (without imaginary part, indicated by `I`) at a rate of 24 Million samples/second (which we will now denote as MS/s for *Msamples/s*).

A brief digression into the detailed comprehension of transfers in Bulk mode on the *firmware* side and reading on the PC side under GNU/Linux would allow us to almost hide a major dysfunction that persists, but which we will unveil later and will prove to be much more interesting than it appears.

1.9 Communication of a known pattern in USB Bulk

Before wanting to read a fast data stream, we already wish to validate microcontroller-PC communication over USB via the Bulk interface by sending a known pattern. An author encountered on `github`, Siddharth Deore [12], kindly provided his examples, even though once again it was necessary to fight with the evolutions of FX2Lib to make these programs work. However, this simple example of bulk communication from the FX2LP, available at `https://github.com/jmfriedt/max2771_fx2lp/tree/main/FX2LP/bulk_read_example`, is also an opportunity to see how to manage this data stream from a C program using `libusb`

From the perspective of the microcontroller, the example declares the endpoint 6 in bulk interface and initializes the properties of the FIFO, which we will fill manually in this example:

```
#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.
}
```

The FIFO will be filled with the content read from ports B and D: it can therefore be amusing to periodically change the status of a bit of these ports, and as it turns out in the previous example that the LED was connected to port D7, we trigger a periodic timer interrupt that changes the state of the pin PD7 which should also affect the display of the data read on this port and transmitted by 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;
    }
}
```

Finally, the main function simply initializes the peripherals and loops indefinitely checking if the FIFO buffer is empty, and if so fills it with the content of ports B and D, or as a comment to convince oneself of the validity of the approach, with a known byte sequence:

```
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;
       }
    }
}
```

This program is compiled and transferred to the RAM of the FX2LP microcontroller for execution from the `FX2LP/bulk_read_example` directory of our repository by
`make && sudo cycfx2prog prg:build/fifo_ep6.ihx run`

From the host's perspective under GNU/Linux, retrieving packets is done by leveraging `libusb` (the `libusb-1.0-0-dev` package under Debian GNU/Linux) whose configuration for the location of header files and libraries can be obtained with `pkg-config`). Once the sequence of incantations is known, the program is simple and compact with

```
#include <stdio.h>
#include <stdlib.h>
#include <libusb-1.0/libusb.h> // or <libusb.h> with 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)  // display the content of the FIFO in binary
     {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("
");
     }
```

```
 libusb_release_interface(hndl, 0);
 libusb_close(hndl);
 libusb_exit(ctx);
}
```

whose function names seem sufficiently explicit to naturally follow the sequence of communications. This program is compiled with `gcc` while remembering to link against `libusb` by completing the compilation command with `-lusb-1.0`.

The same functionalities are implemented in Python3 with

```
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)
```

Once the firmware is flashed to the FX2LP, executing the Python program under GNU/Linux indicates

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

alternating the states of ports B and D (over 8 bits), while the executable resulting from the C program displays

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

with the 7th bit changing state depending on whether PD7 is high or low under the control of the *timer* interrupt.

Note that in case of transaction failures, it may be wise to check if the kernel module `usbtest` has not been automatically loaded, and remove it if necessary as described at `http://www.triplespark.net/elec/periph/USB-FX2/eeprom/` by `sudo rmmod usbtest`.

2 Communication of acquired data from MAX2771 via FX2LP (for real)

We have identified the syntactic differences between Keil and SDCC compilers, identified the silkscreen errors on the printed circuit, everything should therefore work. We concluded section 1.8 by hinting that a dysfunction persisted, a detail without great importance. Indeed, when we configure the board with our firmware and launch an acquisition via `PocketSDR/app/pocket_dump/pocket_dump` communication still occurs at 24 MS/s, even if we attempt to configure the analog-to-digital converter for a different rate, for example switching to 12 MS/s by modifying only the `REFDIV` register from 3 ($\times 1$) to 2 ($/2$). Worse, modifying this one register with a value that seems coherent results in lost communication. Surely a poorly configured register somewhere. but the problem turned out to be just a little more complicated. We need to resolve this latter issue to claim we can compile the PocketSDR functional firmware using SDCC.

We knew the different endianness of the various architectures of processors – with Intel in *little endian* which places the least significant byte of a multi-byte word at the lowest memory address, and Motorola

which in *big endian* places the most significant byte at the lowest address – and we knew of the absence of definition of signed or unsigned nature of integers according to different versions of C compilers, but we now discover that various compilers for the same architecture can choose different endianness. As long as we programmed the 8051 in assembly language, the question of representing quantities over multiple bytes did not arise, since the registers of the 8051 only contain a single byte. But as we must move to programming the microcontroller in C instead of assembly, we need to be able to represent numbers as 2 bytes (`short`) or even 4 bytes (`long`) even for a target that consists only of registers coding 8-bit values. And here, things get complicated.

Regarding the various conventions of `char` depending on compiler variations of `gcc`

One might think that defining `char c;` in C is a deterministic and reproducible instruction. This is not the case in the absence of the explicit prefix `signed` or `unsigned`, or the use of the extension `stdint.h` which allows defining `uint8_t` for an `unsigned char` and `int8_t` for a `signed char` (see for example `/usr/msp430/include/stdint.h` for the MSP430). Indeed, without these precautions, the program which appears trivial a priori

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

compiles with the flag `-pedantic` to indicate overflow for a microcontroller AVR, a PC x86 or an ARM without operating system, respectively as

```
$ 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
```

Thus the first two compilers complain that 240 exceeds the capacity of a `signed char` (thus `signed` has been implicitly added), while `arm-none-eabi-` does not complain, leading us to suspect that the `char` is implicitly `unsigned` there, a major annoyance if a loop tests for a value under 0 to cease its iterations. This example can be completed by searching for the minimum value of `char` with the preprocessor (`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
```

demonstrating that for `gcc` (Intel) and `avr-gcc`, a `char` is signed with a minimum value of -128, while for `arm-none-eabi-gcc` the minimum value of a `char` is 0 thus unsigned. This last case may be linked to the standard imposed by ARM in `https://developer.arm.com/documentation/dui0472/m/C-and-C---Implementation-Details/Basic-data-types-in-ARM-C-and-C--` which states

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

Without wanting to give ammunition to detractors of C, this change of sign of `char` between versions of the same compiler has caused us many troubles, particularly in loops of the type `for (k=N;k>=0;k--)` which become infinite with an `unsigned char` (always positive).

Indeed, [13] teaches that although targeting the same 8-bit architecture, Keil is a *big endian* compiler and SDCC is a *little endian* compiler. As long as calculations are handled by a computer running code compiled by a single compiler, the organization of data in memory is consistent and there is no problem. However, when converting code written for Keil to SDCC that performs *cast* of byte arrays (`char*`) to an integer coded on more than 8 bits (`short` or `int`, or for users of `stdint.h`, `int16_t` and `int32_t`), it is necessary to think about swapping the bytes which will otherwise be transferred in the wrong order to the MAX2771. Thus, the lines for the Keil compiler

```
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);
}
```

become for SDCC (we used the opportunity to replace nested `if ... else` with a `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;
}
```

with

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

originating from `https://www.keil.com/dd/docs/c51/silabs/shared/si8051base/endian.h` which is responsible for swapping bytes (the famous `htonl()` from `/usr/include/netinet/in.h` under GNU/Linux). Fortunately, these are the only two *casts* from byte arrays to an integer of more than 8 bits that need to be corrected in this way, as one can convince themselves by searching `grep "int32_t\ \*" complete_fw.c` in the firmware source code.

Once these last errors are corrected, we have a complete *firmware* compatible with PocketSDR but compilable by `sdcc` without depending on proprietary compiler, which Tomoji Takasu informs us is limited to executables of 4 KB for its free version.

We mentioned in the introduction that the evaluation board of MAX2771 is equipped with a 16.368 MHz oscillator and that PocketSDR is configured for a 24 MHz oscillator. This information is included in the firmware, at `https://github.com/tomojitakasu/PocketSDR/blob/master/FE_2CH/FW/v2.1/pocket_fw.c#L27` for the original version for Keil or at `https://github.com/jmfriedt/max2771_fx2lp/blob/main/FX2LP/complete_fw/complete_fw.c#L28` for our version for SDCC: this constant `F_TCXO` will be replaced by 16368 so that the firmware thus recompiled works directly with the evaluation board equipped with its original oscillator.

Thus, we can benefit from all the executables provided by PocketSDR to test the proper functioning of the MAX2771 evaluation board coupled with the FX2LP. To do this, we will explore some satellite signals transmitted in the upper half of the L band, around 1500–1600 MHz.

3 Results: Iridium and GPS

We will acquire, at a rate of several tens of megabytes per second, gigabytes of data that we want to validate, while we are not even sure about the proper functioning of the hardware platform and its software configuration. We therefore need to assess the difficulty of detecting the signal we hope to observe against the noise before embarking on measurements.

Indeed, three parameters must be configured to determine the characteristics of an acquisition:

- the central frequency of the acquisition, by programming the multiplication factor of the reference oscillator (24 MHz) by PLL to control the voltage-controlled oscillator (VCO). This carrier is irrelevant in the analysis as it is eliminated during analog pre-processing by mixing with the signal to be acquired.
- the sampling frequency f_s, defined by the rate at which the analog-to-digital converter (ADC) acquires the data,
- include an intermediate frequency or work directly in baseband. The consequence of this choice is the production of real signals only in the first case, and complex in the second. Indeed, in the first case, a unique mixer brings a signal real close to the baseband but offset from the intermediate frequency, and it is the digital transposition (by software processing after analog-to-digital conversion) from the intermediate frequency f_{IF} to the baseband by multiplying by $\exp(j2\pi f_{IF}t)$ with $t = [0 : N-1]/f_s$ the discrete time along the N samples acquired at the sampling frequency f_s, which will produce the expected complexes I and Q. The benefit of going through the intermediate frequency is to reject noise, especially from digital electronics close to 0 Hz, out of the acquisition band.

The two objectives we set ourselves are, firstly, to observe the spectrum of the signals of Iridium satellites, centered on 1622 MHz but powerful enough to be well visible, and secondly GPS centered on 1575.42 MHz, the ultimate objective of this development but under thermal noise thus difficult to detect when the proper functioning of the electronic system is under evaluation. Indeed, let us recall that the 50 W emitted by a GPS satellite with its 13 dBi gain antennas reach the ground after traversing the 20000 km separating it from the receiver only with a power (Friis equation of energy conservation over the sphere on which the emitted power is distributed) of

$$10\log_{10}(\underbrace{50\times1000}_{\text{W}\Rightarrow\text{mW}}) - 20\log_{10}(\underbrace{1575.42\cdot10^6}_{\text{carrier}}) - 20\log_{10}(\underbrace{20000\times1000}_{\text{distance km}\Rightarrow\text{m}}) + \underbrace{147.55}_{20\log_{10}(\frac{c}{4\pi})} + \underbrace{13}_{\text{gain}} = -122 \text{ dBm}$$

which compares with the integration over a bandwidth of 2 MHz of the thermal noise floor at ambient temperature thus $\underbrace{-174}_{\text{dBm/Hz}} + 10\log_{10}(\underbrace{2\cdot10^6}_{\text{Hz}}) = -111$ dBm, thus a signal 11 dB below the thermal noise.
This condition is not the most comfortable to validate an unknown circuit with an unknown configuration.

3.1 Real-time Acquisition and Analysis of Iridium

We recently discussed the reception of Iridium [14] but with a general-purpose software radio receiver providing a number of bits of resolution and thus excellent quantization. Here we wish to validate the detection, see the decoding, of Iridium with converters coding the information on 3 bits. The outcome of the attempt is not obvious: by performing the correlation between a signal and the pseudo-random sequence of N bits ("chips") coding each message bit emitted by a satellite, pulse compression (intercorrelation) accumulates during the integral all the energy distributed across the N chips in a single peak of correlation that thus sees its quantization improved by $\log_2(N)$ bits. For the 1023 chips of GPS, the gain is of the order of 10 bits and the 3 initial bits become 13 bits on each correlation peak that repeats every millisecond (i.e. 1.023 Mchips/s divided by $N = 1023$). The situation is not as favorable with Iridium which transmits a powerful signal but phase-coded in two states (BPSK) or 4 states (QPSK) that must be identified despite mediocre quantization and the absence of compression gain.

The first thing to consider is the operating range of the voltage-controlled oscillator, since the technical documentation limits the characteristics to the useful band, thus up to 1610 MHz, the upper limit of the frequencies necessary to decode the Russian satellite navigation system GLONASS. Although the

registers of the MAX2771 allow configuring a wide range of values, including the frequency of the voltage-controlled local oscillator (VCO), nothing guarantees that the hardware can meet these demands. We tested, by configuring through `app/pocket_conf/pocket_conf` of PocketSDR and connecting the input of the MAX2771 to a continuous radio frequency signal from a synthesizer, the various combinations of $RDIV \in [0 : 1023]$ and $NDIV \in [36 : 32767]$ in order to produce a local oscillator frequency $f_{LO} = f_{Xtal}/RDIV \times NDIV$ with $f_{Xtal} = 24$ MHz to measure a frequency offset of 1 MHz from f_{LO}, and we notice good operation up to $NDIV = 68$ if $RDIV = 1$ to produce $f_{LO} = 1632$ MHz, or $NDIV = 546$ with $RDIV = 8$ to produce $f_{LO} = 1638$ MHz, but beyond ($NDIV = 547$) the local oscillator fails. Thus, the central frequency of Iridium at 1622 MHz is indeed compatible with the MAX2771. Note that the complete law governing the local oscillator frequency using a *fractional* phase-locked loop is

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

with $FDIV \in [36 - 32767]$ the fractional part of the multiplicative frequency factor. For each configuration, one can read back the state of the registers by launching the command `pocket_conf` without argument: of course in the absence of the second MAX2771 which equips the PocketSDR but is absent from the evaluation board, the configuration and the reading of the registers of the second component will be inconsistent but without consequences.

The choice of the data rate on the USB bus is determined by the clock that frequencies the ADC and thus the IFCLK signal that rhythmically fills the FIFO of the FX2LP. The main oscillator ($f_{Xtal} =$ 24 MHz) that clocks the FX2LP can be multiplied or divided by 1, 2 or 4 depending on the value placed in `REFDIV` when $ADCCLK = 0$, then the registers $REFCLK\_L\_CNT$ and $REFCLK\_L\_CNT$ determine the data rate by

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

where $f_{XTal}(REFDIV)$ is the operation indexed by $REFDIV$ on f_{XTal} (/4, /2, ×1, ×2, ×4) and we find that choosing $L\_CNT = 2048$ with $M\_CNT = 0$ allows us to divide by 3, thus for example to produce 8 MS/s if $REFDIV = 3$ to multiply by 1 f_{Xtal}.

At the end of these explorations, the configuration ultimately used exploits a sampling frequency of 24 MHz (in order to reach after transposition and decimation more than the 10 MHz that Iridium covers) and an intermediate frequency of 6.5 MHz, chosen as more than the 5 MHz necessary to retain 10 MHz effective in baseband after transposition. To do this, `NDIV=26925` and `RDIV=400` with `INT_PLL=1` for a PLL without fractional part, so that the frequency of the local oscillator is $24 \cdot 10^6/400 \times 26925 = 1615.5 \cdot 10^9$ Hz or $1622 - 6.5$ MHz with 1622 MHz the central frequency of Iridium. For the ADC, as we retain the clock frequency, we simply choose `REFDIV=3`.

The second major difference between GPS and Iridium is that while the former emits continuously, the latter transmits only by **brief pulses** at arbitrary times depending on the positions of the satellites in the sky and the requests from ground terminals. Thus, recording blindly a few minutes of Iridium to realize that few or no signal is exploitable is frustrating: we wish to visualize in real-time the signal using the spectrum analyzer (`Frequency QT GUI Sink`) of GNU Radio. Of course, there is no GNU Radio interface for the FX2LP, but since PocketSDR provides `pocket_dump` capable of writing into a file, we simply need to create a named *pipe* [16] to transfer the acquired data from the USB port to a pseudo-file read by the `File Source` of GNU Radio which feeds the spectrum analyzer. The rather trivial processing chain GNU Radio Companion is proposed in Fig. efgrc (top-left), and provided that the bandpass filter (bottom, left) of an active GPS antenna [14] has been removed and a polarization Tee inserted between the MAX2771 and the antenna to feed its amplifier [14], we will observe a representative spectrum of Iridium signals.

Furthermore, given that `gr-iridium` from Sec and Schneider [15] does not know the concept of intermediate frequency, we take advantage of the spectrum displayed by GNU Radio, which already needs to eliminate this deviation from the nominal carrier using a `Xlating FIR Filter`, to transmit the result of the transposition into a ZeroMQ Publish stream. Since the `Xlating FIR Filter` has rendered half of the real spectrum useless, it is equipped with a low-pass filter and a decimation factor of 2, such that `gr-iridium` receives a stream of complexes in floating point numbers at 12 MS/s. We initially attempted to save into a file, but the size grows so fast at 12 MS/s and the Iridium information so rare that real-time processing is more relevant: 184 MB in ASCII format ('0' and '1') after 3.3 hours of acquisition and decoding instead of 132 GB that would have been occupied by the raw data.

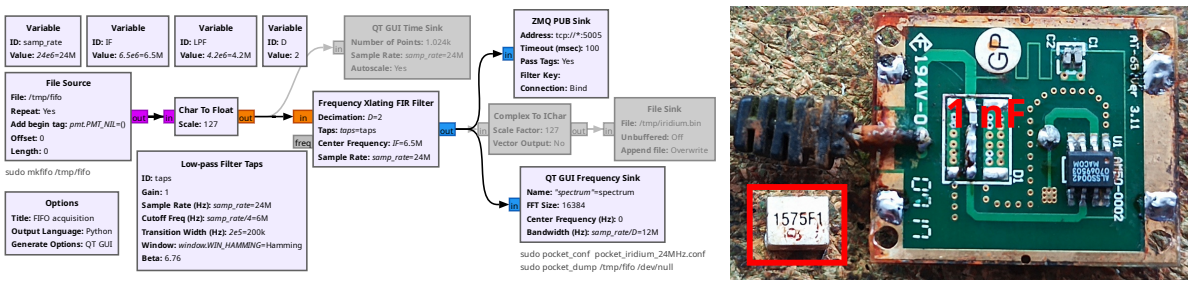


Figure 6: Left: GNU Radio processing chain taking as input a FIFO in the form of a named *pipe* `/tmp/fifo` created by `mkfifo /tmp/fifo`. This flow of real integers coded on 8 bits at a rate of 24 MS/s (defined in the `samp_rate`) is converted to floating point numbers, frequency transposition by $IF = 6.5$ MHz, filtering and decimation, for display in the spectrum analyzer over a bandwidth of 12 MHz and transmission to a ZeromMQ Publish for which one will remember to activate `Pass Tags: Yes` for consistency with `gr-iridium`. Here the socket is connected to port 5005 of the local host but the data could equally well be transmitted over a network for remote processing. Right: modified GPS L1 antenna for receiving Iridium signals by removing its bandpass filter centered on 1575.42 MHz to replace it with a 1 nF capacitor, thus an impedance of 0.1 Ω at this frequency, while avoiding bringing the DC power component of the amplifier as close as possible to the antenna on the RF pin of the amplifier.

Indeed, `gr-iridium` can read the stream from ZeroMQ Subscribe for processing, it just needs to be informed of the data rate by modifying `examples/zeromq-sub.conf` with `center_freq=1622000000` and `sample_rate=12000000` while ensuring that `address=tcp://127.0.0.1:5005` is consistent with the port selected in the ZeroMQ Publish of GNU Radio.

Once all these components are assembled, the acquisition sequence (Fig. efgrc) is

1. `sudo mkfifo /tmp/fifo` to create the pipe,
2. `sudo app/pocket_conf/pocket_conf pocket_iridium_24MHz.conf` to configure the MAX2771,
3. `sudo app/pocket_conf/pocket_conf` to check that the configuration has been taken into account,
4. `sudo app/pocket_dump/pocket_dump /tmp/fifo /dev/null` to transmit the IQ flow of the single MAX2771 to the pipe, the other flow going to the trash.
5. GNU Radio reads from the pipe to transpose, decimate, display the spectrum and re-emit toward ZeroMQ Publish with `python3 fifo_acq.py` using the Python script produced by GNU Radio Companion from `fifoinout.grc`,
6. finally, `gr-iridium` receives the IQ flow to extract the numerical information and output to `stdout`. A first attempt may consist of waiting for the *Iridium Ring Alert* (IRA) frames that demonstrate that satellite signals are indeed received with

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

   but since there is much more information that transits than IRA, we will prefer to store these bits in a file for post-processing by

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

7. we process these bits to try to recover intelligible phrases with

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

8. we can finally search in the phrases for the localization and identification frames of aircraft according to the ACARS protocol

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

oindent or produce a KML file containing the location of the beams transmitted in the IRA frames by the detected satellites

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

The result of these processes is two detected aircraft

```
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
```

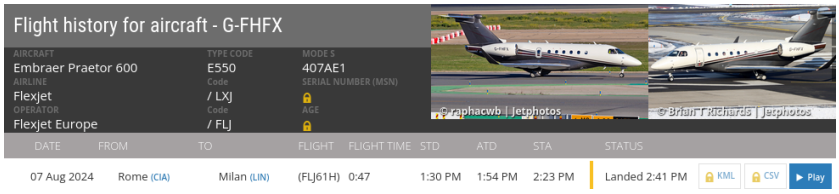


Figure 7: Screenshot from FlightRadar24 validating the analysis of an ACARS message received by Iridium.

with a small aircraft indeed identified by `flightradar24.com` at this moment between Rome and Milan (Fig. 7), and an Airbus Beluga that normally should not have flown at this moment but may have performed a communication test from the ground. The map of the beams (Fig. 8) is consistent with a reception from Clermont-Ferrant (yellow star) according to a partially obstructed view towards the east/southeast.

We can thus be fairly confident that we have indeed received and decoded Iridium frames. Let us recall that the goal is not to create a robust Iridium receiver but to validate the proper functioning of the MAX2771 on a powerful signal, which has nonetheless allowed decoding numerous digital frames, a nice *hack* of the MAX2771 in the original sense of the term [17].

3.2 Post-processing analysis of GPS

The acquisition of the GPS signal is first validated by detecting signals from the received satellites – the acquisition phase of a receiver that knows nothing of its location and the geometry of the constellation – by searching through correlation all possible satellite identification codes (Gold Codes) for all possible Doppler shifts. However, this step, handled by `python/pocket_acq.py` from PocketSDR, must also take into account an eventual shift of the receiver's local oscillator: by default, this program searches only for Doppler shifts in the range ±5 kHz corresponding only to the satellite's speed, but it is prudent in case of decoding failure to extend this range (option `-d`, a value of 30000 for 30 kHz is reasonable for a quartz resonator of decent quality). Once the satellites are identified, ensuring the quality of the acquired signal, we will finalize the processing chain by finding the optimal positioning solution of the receiver considering the observed time of flight for at least 4 satellites of the constellation: this position, velocity and time (PVT) solution will be obtained using `gnss-sdr` that we will feed with the file of IQ acquisitions from the MAX2771, and then obtain in real-time.

Initially, the MAX2771 is configured: as with Iridium, we take advantage of
`app/pocket_conf/pocket_conf`
but this time with a configuration file `conf/pocket_L1L2_8MHz.conf`. Thus the MAX2771 is configured with a central frequency of the L1 band shifted from the intermediate frequency of 2 MHz, sampling at 8 MS/s, and the configuration of the second MAX2771 for the L2 band simply ignored. Once the configuration is read and validated by `app/pocket_conf/pocket_conf` without argument, we will acquire the measurement files by `app/pocket_dump/pocket_dump -t 2 ch1.bin ch2.bin` with `-t` to indicate that we only want to record for two seconds, and the optional file name arguments indicate the two output channels. Once again in the absence of a second MAX2771, the file `ch2.bin` will not contain any exploitable values and only `ch1.bin` will be exploitable. In order to save disk space, `ch2.bin` can advantageously be replaced by `/dev/null` without loss of functionality.

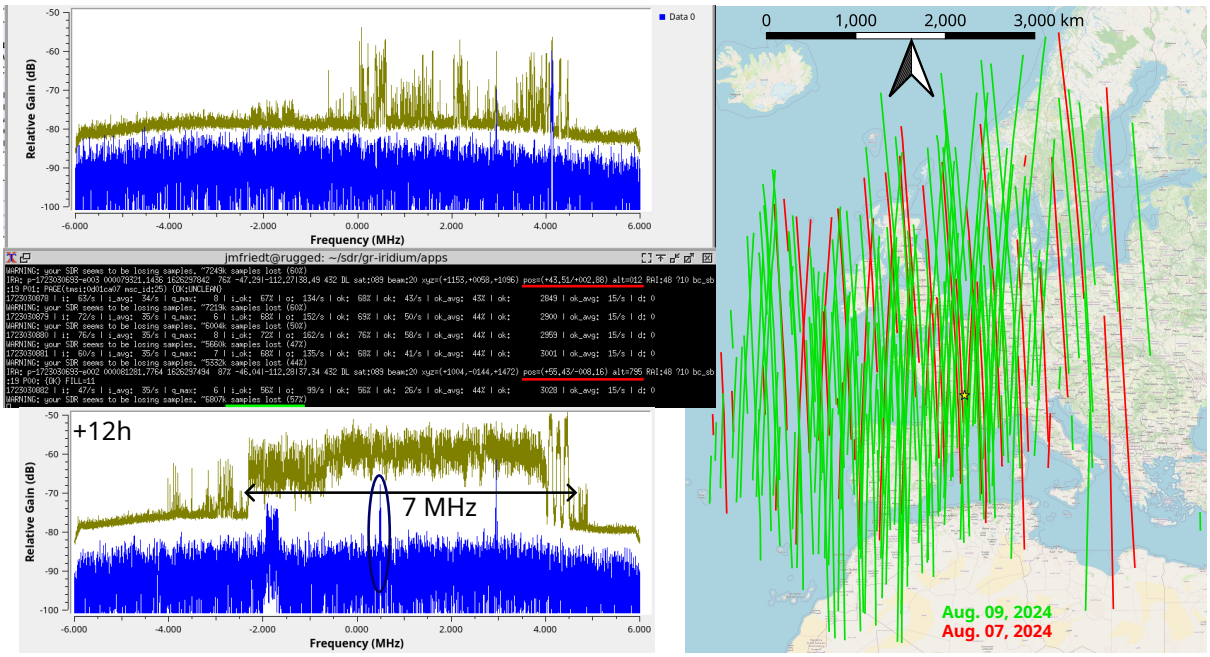


Figure 8: Left: at the top, the spectrum displayed by GNU Radio Companion after frequency transposition by the `Xlating FIR Filter` and decimation by a factor of two to provide 12 MHz of bandwidth, sufficient to cover all Iridium signals at the same time. At the top at the beginning of acquisition the individual communication channels are visible, at the bottom after more than 12 hours of acquisition, the spectrum has become uniform over the more than 7 MHz necessary to cover all the subbands. In this spectrum, in a dark blue ellipse a brief *burst* of communication captured in this image. Right: map of the beams identified by the IRA frame (map background: OpenStreetMap in QGIS) for two days of acquisition, August 7 and 9, 2024 during 12 hours, to validate reproducibility. In the middle, `gr-iridium` reports many lost frames (green underlined) but the IRA frames are still detected (red underlined).

To validate the relevance of the data acquired in `ch1.bin`, we will attempt to acquire the satellites by sequentially correlating the 32 possible pseudo-random codes for all possible Doppler shifts. This result is obtained from the PocketSDR directory by

```
python3 ./python/pocket_acq.py ch1.bin -f 8 -fi 2 -sig L1CA -prn 1-32 -d 30000
```

to indicate that the sampling frequency is 8 MHz, intermediate frequency of 2 MHz, implicitly implying that the data are real (without imaginary part), and that we are searching for GPS satellites 1 to 32 with a maximum frequency offset of 30 kHz. The result is illustrated in Fig. 9 which additionally validates the consistency of the analysis with the observation of a mobile phone used as a GNSS receiver.

By configuring the MAX2771 in IQ mode similarly, without intermediate frequency, to acquire at a rate of 4 MS/s using `conf/pocket_L1L2_4MHz.conf` from PocketSDR, then `pocket_dump` will display a message of the form

```
$ 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
```

which allows us to verify the rate (here 4 MS/s) and the nature (here IQ, in accordance with the configuration) even if the second MAX2771 of the PocketSDR is absent.

Decoding by `gnss-sdr` requires a configuration file that connects the various processing steps to each other according to a graph scheduled by GNU Radio, taking into particular account the fact that each type of output data corresponds to the input type of the next processing block. Naturally, a software radio receiver would like to process complex baseband values, but we find that our setup quickly loses the USB connection when such a type of data is transferred, probably due to excessive electrical coupling between the clock signal and data lines at the FX2LP (see conclusion). Having therefore abandoned the option to process complexes (type `ibyte` in the `gnss-sdr` nomenclature) according to the configuration

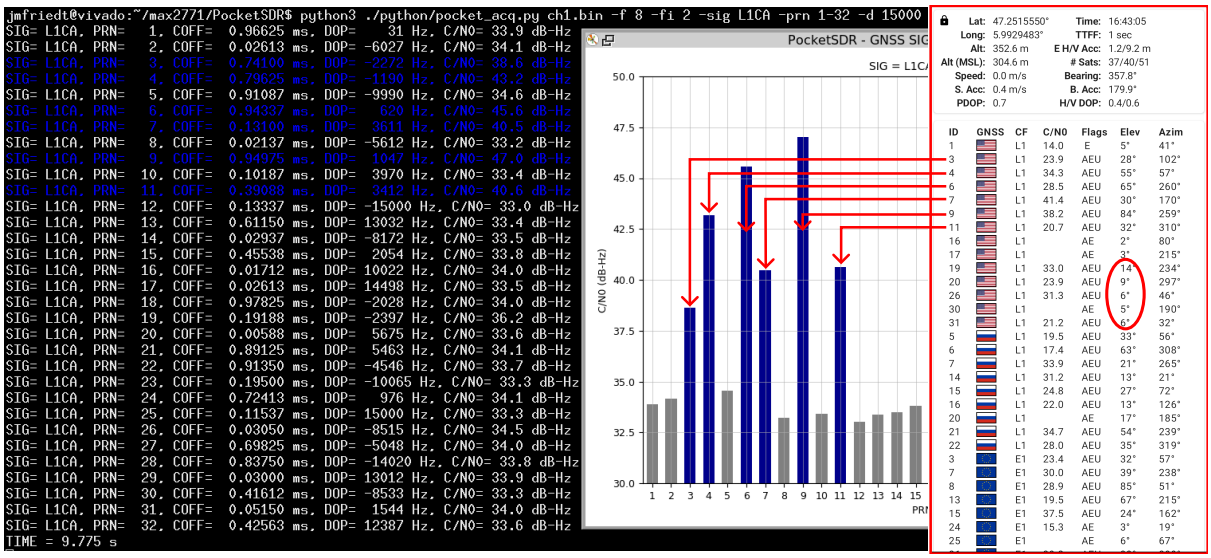


Figure 9: Left: the result of `pocket_acq.py` from PocketSDR applied to a file acquired by `pocket_dump` from the same project. Middle: the graphical output of the satellite acquisition phase, with the satellite number (PRN) on the x-axis and the signal-to-noise ratio on the y-axis. Right: observation by mobile phone under Android running `GPSTest` (red frame), in perfect coherence with the signal acquired by the MAX2771 (red arrows), except for the satellites with the highest indices that are too low on the horizon to be detectable (red ellipse).

shown just above (`pocket_L1L2_4MHz.conf`), two paths remain to explore: real-time processing and post-processing of recorded files according to the configuration `pocket_L1L2_8MHz.conf`.

The first solution is quickly eliminated since the FIFO type that would allow data transmission via a named *pipe*, of type `Fifo_Signal_Source` from `SignalSource.implementation` in the `gnss-sdr` configuration does not know how to process real numbers (without imaginary part), and therefore only the option of processing a file containing intermediate frequency acquisitions remains, and convincing `gnss-sdr` to transpose the signal from this intermediate frequency before extracting the time of flight information, thus pseudo-range, and therefore PVT solution. The configuration file starts with the classic

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

which indicates that the information is acquired at 8 MS/s. The source coming from a file containing data on 8 bits is provided by

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

which is intuitive. The next step is the most complex since we must convert the integers into complex floating-point numbers after transposing from the intermediate frequency of 2 MHz using the `Frequency Xlating FIR Filter` from GNU Radio, namely mixing with a local oscillator, filtering, and decimation:

```
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
```

starts by converting the input data from 8 to 16 bits in order to feed the `Frequency Xlating FIR Filter` which will output floating-point complexes. The properties of the filter are determined by its passband and cutoff frequency (Fig. 10), always normalized to the Nyquist frequency (half of the sampling frequency 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
```

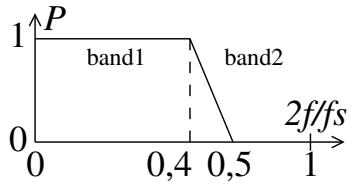


FIGURE 10 – Template of the low-pass filter. Note the x-axis graduated in normalized frequency, as in any discretely sampled time system.

thus 40% of the band (ranging from 0 to half the sampling frequency) is passband, and the cutoff starts at 50% of this band, always represented in normalized frequency, thus assigning the value 1 to the Nyquist frequency equal to the half sampling frequency. The number of filter coefficients (`taps`) is 5. Finally,

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

does not decimate the stream but transmits it to the acquisition and tracking phases to find the PVT solution from the L1 signal of the GPS constellation by maximizing the use of 12 channels, thus potentially the signals from 12 satellites. This number of channels is limited by the computing power of the computer for real-time processing and, in practice, by the number of satellites visible from a given site:

```
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;
```

which, as before, tries to compensate for a Doppler shift of up to 30 kHz, with a step of 250 Hz chosen as a small value compared to the inverse of the duration of each bit, which is 1 ms, thus small compared to 1 kHz.

The rest is just classic to conclude the search for PVT solutions and is imposed according to the documentation at `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
```

At the end of the execution by `gnss-sdr -c file.conf` with the configuration file contained in `file.conf`, we are pleased to obtain

```
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]
...
```

from which we have retained only the main steps to describe the meaning of the messages. The information starting with `Tracking` does not indicate the presence of a usable signal in the recording, but only that the signature (pseudo-random code PRN) of a certain satellite (`GPS PRN XX`) is being searched for in the processed signal assigned to the channel "channel". The crucial point is the appearance of `tracking bit synchronization locked` which indicates that the signal from a satellite has been identified and can be analyzed to decode the navigation message: the satellite number corresponds to the channel assigned to it. Finally, all the information necessary to locate the satellite in space is acquired during `New GPS NAV message received`, even though in practice it will be necessary to decode the 5 successive "sub-frames" since, for example, the first sentence indicates the corrections to be applied to the onboard atomic clocks, the next two the satellite ephemerides, then the date and ionospheric conditions, and finally the status of the constellation. We will find that success often comes after decoding a "sub-frame" 3 (even though here it was after a sub-frame 1), provided that the navigation messages of at least four satellites have been decoded (to solve the equation with 4 unknowns: position and time) to provide the long-awaited result of `First position fix` which will then repeat as long as enough satellites are visible. Since each *frame* lasts 30 seconds, one must wait at least that duration once the first navigation message is obtained.

3.3 Real-time GPS analysis

To conclude this study, obtaining GPS position in real-time rather than in post-processing of a recorded file which is necessarily constrained in space and thus in duration may seem desirable. However, the FIFO source of `gnss-sdr` only accepts complex values, and in the current spaghetti setup, transferring complexes induces a corruption of the USB stream after a few seconds to tens of seconds. We therefore approached the problem as follows:

- creation of a named *pipe* `sudo mkfifo /tmp/fifo` to acquire the data (owned by root so that `pocket_dump` launched in `sudo` can write to it),
- `sudo pocket_dump /tmp/fifo /dev/null` to feed the *pipe* with real data acquired at 8 MS/s with an intermediate frequency of 2 MHz, the second channel going into the void of `/dev/null` since a single MAX2771 equips the evaluation board,

- a GNU Radio processing chain takes care of transposing in frequency (`Frequency Xlating FIR Filter` and takes the opportunity to filter and decimate the resulting flow whose majority of spectral components have become unnecessary (Fig. 10, left). The output rate is 2 MS/s complex
- having found that an output file to another named *pipe* communicates poorly with `gnss-sdr`, we opted for a GNU Radio output in the form of a ZeroMQ Publish socket, since `gnss-sdr` offers the ZeroMQ Subscribe interface.

To achieve this result, the modifications to the configuration file of `gnss-sdr` are the definition of the source and removing the frequency transposition in `gnss-sdr` since GNU Radio has already taken care of that upstream:

```
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
```

and the rest is identical to the previous configuration.

```
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]
```

which demonstrates the convergence of the solution in less than two minutes, in line with the observations from GPSTest on an Android mobile phone used as a reference receiver (Fig. 10, right).

A video of this sequence of processes, providing particularly the command sequence and the acquisition dynamics that we cannot reproduce in these static pages, can be found at `https://www.youtube.com/watch?v=B5UcFnkbXIk`

4 Conclusion

We have identified the protocol of the MAX2771 evaluation board, understood that only the conversion of USB commands to SPI is supported but not the high-speed data transfer, learned to configure the FX2LP as a transfer interface between words formed from parallel bits and a clock to USB, and acquired this data on PC. However, the spaghetti layout with long communication wires carrying signals at several MHz to tens of MHz cannot guarantee a robust transaction, and a dedicated circuit is necessary to fully leverage the performance of the MAX2771.

Coupling between adjacent wires

The measurement at the network analyzer of the coupling between two adjacent wires illustrates the problem of the potential induced by a signal on the wire carrying the neighboring signal. This problem becomes all the more important as the communication frequency increases.

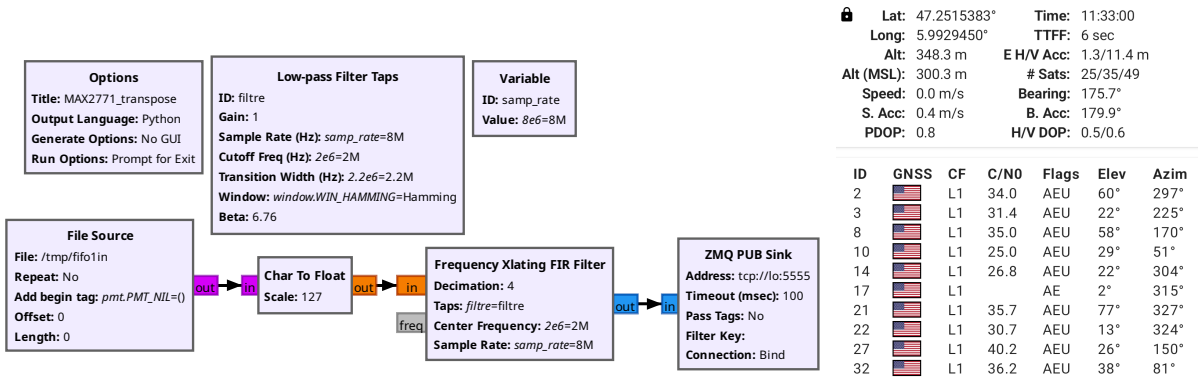
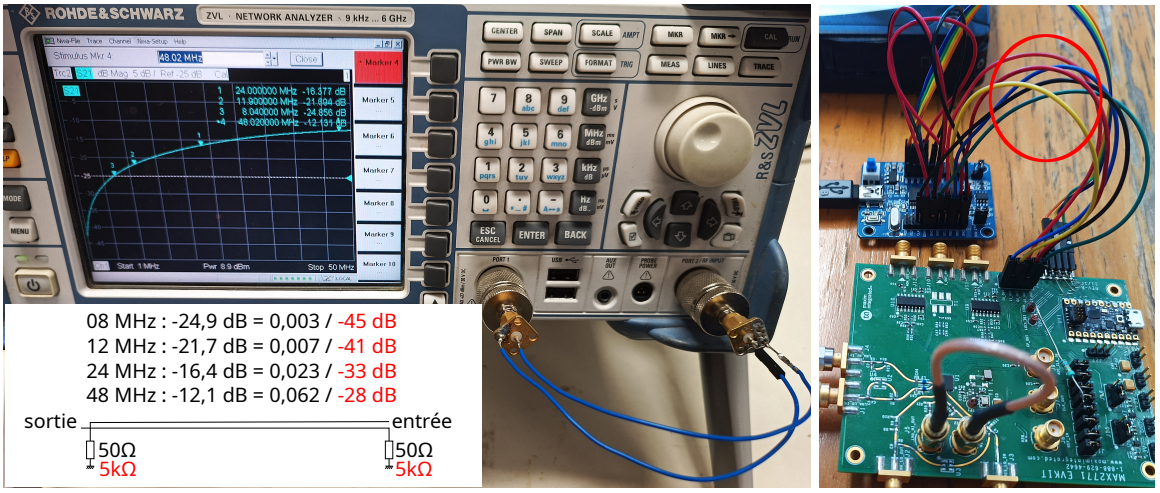


Figure 10: Left: a GNU Radio processing chain reading 8-bit integers from a file that is actually a FIFO (*named pipe*), followed by conversion to a floating-point number with a homothety to bring the result between −1 and 1, frequency transposition, and communication of the result via ZeroMQ Publish socket, the output to a file connected to another *pipe* having not given good results. Right: a screenshot of GPSTest run on a mobile phone used as a reference receiver, demonstrating the coherence with the satellites exploited by `gnss-sdr` to obtain the solution (PRN 2, 8, 21, and 32).



A network analyzer measures by its transmission parameter S_{21} the ratio of the input potential V_i to the output potential V_o (these are indeed voltages and not powers). The figure above reports in logarithmic scale in decibels (dB) as well as the conversion to linear scale $V_o/V_i = 10^{dB/10}$ to have the ratio of voltages between the signal transmitted on a data wire and the coupling on the neighboring wire. We have reported in black the measurement when each wire is terminated to ground by a 50 Ω resistor aimed at matching the impedance, but maximizing current in the wire and thus the coupling. Replacing the load resistor with a value of 5 kΩ might suggest that coupling decreases (in red) but this induces another problem since the misadaptation of the emitting line creates a standing wave that will disrupt the detection of digital signals. One might think that working at 8 MHz induces only minimal coupling of the order of a thousandth, but it should be remembered that the transmission of square signals such as the clock emitted by the FX2LP carries all harmonics of the clock frequency with power that decreases with the harmonic number. Thus, a square signal of 3.3 V at 8 MHz still carries a component of 0.66 V at 40 MHz (harmonic 5) and 0.47 V at 56 MHz (harmonic 7 since only odd harmonics represent a square signal) which couple inductively efficiently between the two wires. It is therefore understood that the spaghetti plate of the setup connecting the MAX2771 evaluation board to the FX2LP does not guarantee robust digital transactions.

Just as the PocketSDR that inspired this study, we aim to exploit signals simultaneously acquired by

multiple MAX2771 to evaluate the direction of arrival of a signal and possibly cancel a source of spoofing or jamming. The dedicated circuit for these developments, and many others, will be the subject of the article that will continue this description.

All source codes are available at `https://github.com/jmfriedt/max2771_fx2lp/` and in particular in the subdirectory `Maxim_EvalBoard` for this article. The bibliography works have been obtained from Library Genesis.

References

[1] J.-M. Friedt, É. Carry, *Enregistrement de trames GPS — développement sur microcontrôleur 8051/8052 sous GNU/Linux*, GNU/Linux Magazine France, **81** (Feb. 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* at `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] the projects `https://fpga4u.epfl.ch/wiki/FX2.html` et `https://fpga4u.epfl.ch/wiki/FPGA4RF.html` are dated from 2011 but remain relevant to this date

[6] `https://sdcc.sourceforge.net/` updated March 6, 2024, also available as Debian package with the same name

[7] source code of the logic analyzer sigrok at `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.html`

[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 is *big endian* according to `https://groups.google.com/g/comp.arch.embedded/c/Cabbhrl9_oM/m/3yOfgiqm7_YJ` while SDCC is *little endian* as stated at `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) at `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] as the readers of this magazine well know, and the journalists of France Info or the authors of `https://www.orangecyberdefense.com/fr/solutions/services-manages/micro-soc-poste-de-travail/edr` do not, a *hack* is the diversion of a technology from its initial purpose, and not at all a malicious act a priori. As A. Guitton indicates in "Hackers. At the Heart of Digital Resistance" (Ed. au Diable Vauvert, 2013), a hacker is a person who aims to "understand the functioning of a mechanism, in order to be able to tinker with it to divert it from its original functioning," or, as she repeatedly states at the beginning of his work, "understand, tinker, divert." The definition at the beginning of R. des Bois, "Get Up and Code: Confessions of a Hacker" (La Martinière, 2018) could also have been related to the introduction of this article by stating, "to see something broken and not be able to help but do nothing, either you exploit it or you fix it, but it is impossible to ignore this dysfunction and leave it as is." What a pity that the author does not remember this definition later in his discourse, a book that is of no interest after this brief relevant mention, in which the protagonist resorts to buying security vulnerabilities on the web to exploit them against credulous and incompetent users without demonstrating technical creativity.