USB Programming under GNU/Linux: Application of the FX2LP for a Software Radio Receiver Dedicated to Satellite Navigation Signals (2/2)

Jean-Michel Friedt, December 31, 2024

Since Tomoji Takasu, author of RTKLib, published at `https://github.com/tomojitakasu/PocketSDR` his software-defined radio receiver dedicated to satellite navigation constellations, we dreamed of getting a copy. Not commercially available, we decided to produce our own version, following the principles of using opensource tools only. We will take advantage of this new working platform to study some aspects of satellite navigation signal exploitation, from spoofing detection to disciplining a quartz oscillator to the time reference produced by atomic clocks on board satellites, and doing so by leveraging the knowledge gained from programming USB interfaces, and communications between PocketSDR, GNU Radio, and `gnss-sdr` software.

We continue our exploration [1] of efficient USB bus programming under GNU/Linux, especially for receiving global navigation satellite system (GNSS) signals, and GPS in particular. This study provides tools to understand somewhat abstract concepts in the absence of concrete data to process regarding software-defined radio processing of GNSS signals, initiated with the reading of [2] and recently updated in [3] which extends the analysis to all constellations now available. The need to master communication techniques to achieve speeds of several tens of MB/s is justified by the proliferation of satellite navigation signals occupying increasingly significant bandwidths to provide continuously improving resolution (Fig. 1).

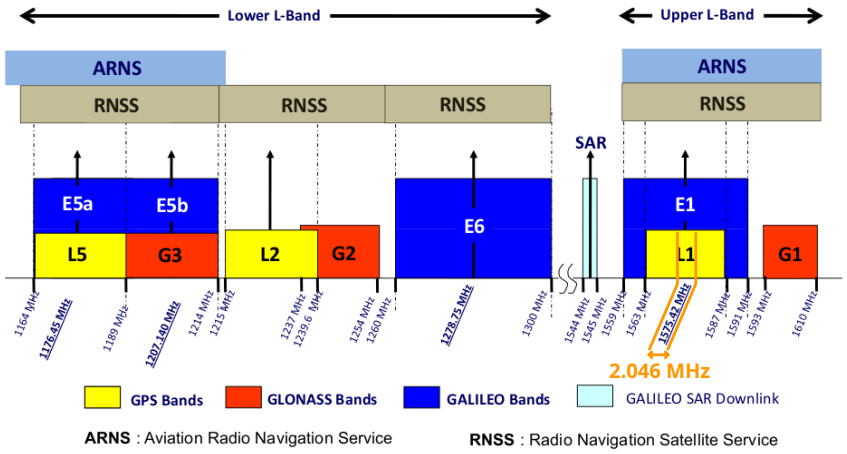


Figure 1: Spectrum of the L band dedicated to satellite navigation signals. This graph does not include the Chinese Beidou constellation and only addresses American GPS, Russian GLONASS, and European Galileo. Note that the historical L1 band is the narrowest with only 2 MHz, while L2 and then L5 occupy increasing bands of several MHz, even tens of MHz, implying communication rates as high as those achieved with efficient USB usage. Graph from ESA taken from `https://gssc.esa.int/navipedia/index.php/GNSS_signal`. Note the use of the naming "upper L band" and "lower L band" that we will find in the description of the MAX2771. The historical L1 C/A band has been highlighted in orange.

We mentioned in the first part of this series that the EZ-USB FX2LP can be obtained assembled and ready for use for a little less than 5 euros on AliExpress, while the USB interface component alone costs nearly 20 euros at Farnell (reference 1269134). We have chosen to redesign the printed circuit board of Tomoji Takasu's PocketSDR, this time as a daughter board of `https://fr.aliexpress.com/item/1005006134347046.html`, and free ourselves from the constraint of using only one antenna to power the two receivers as required in the original design, in order to allow the freedom to perform radiation pattern control to cancel a source of GNSS spoofing or jamming [4] by benefiting from the spatial diversity of two separate antennas to produce the signals of the two Maxim IC MAX2771 acquisition interfaces.

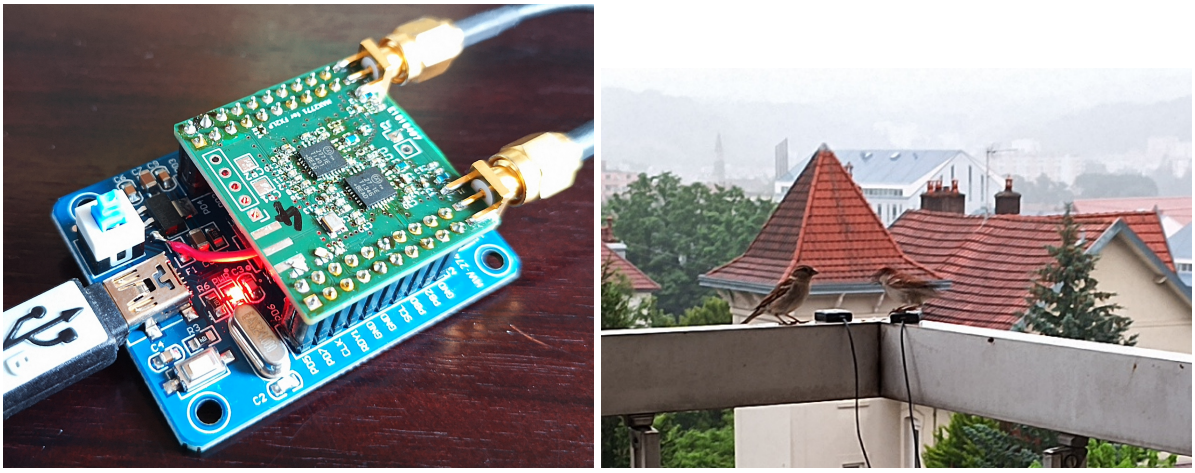


Figure 2: Left: daughter card of the development board (blue, below) FX2LP, equipped with two MAX2771 and additional passive components. The choice of frequency band – L1 or L5/L2 referred to as upper and lower respectively – is made by straps that short-circuit certain footprints in 0402 SMD format to route the signals to the correct component peripherals. Here, two L1 GPS antennas are connected and powered by bias-Ts to detect a spoofing source (right).

1 Printed circuit board design

While PocketSDR is not commercially available, all schematics and routed printed circuit boards are available on the Github repository so manufacturing is "only" a financial question of subcontracting the production of printed circuits (unless means to manufacture a 4-layer circuit with plated holes at home are available). It should be noted that other initiatives, even free ones, exist, particularly `https://github.com/WKyleGilbertson/MAX2769FT2232H`, which freely offers the schematics and routing of a MAX2769-based board limited to the upper L band between 1550 and 1610 MHz, which is sold for the modest sum of 500 euros at `https://www.navstargps.com/`. However, using the FTDI FT2232 instead of the FX2LP removes access to a programmable embedded microcontroller and slightly reduces the scope of development. Therefore, we made efforts to adapt the schematic and routing of the PocketSDR to the commercially available development board equipped with the FX2LP (Fig. 2).

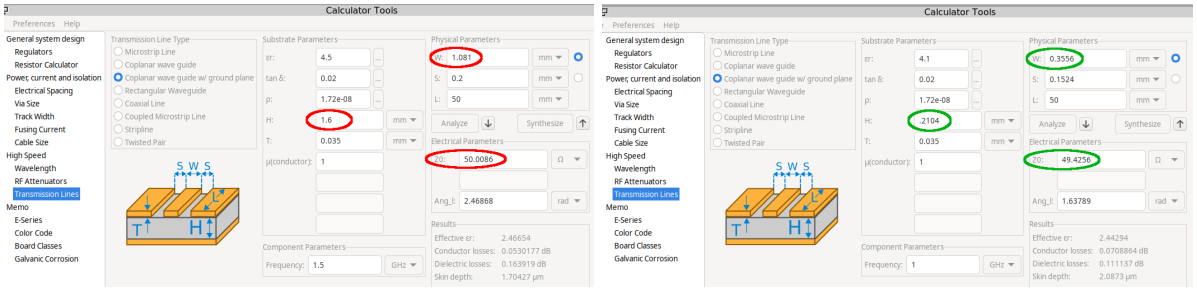


Figure 3: The KiCAD calculator allows to evaluate the impedance of a coplanar transmission line – patterned on a dielectric substrate of a printed circuit board with a uniform ground plane on its opposite side and surrounded on both sides by a ground plane. With a 1.6 mm-thick substrate as available on a rigid double-layer printed circuit board, it is impossible to achieve the characteristic radiofrequency impedance of 50 Ω with a reasonable track width (< 1 mm). To do this, a thinner dielectric substrate (typically 0.2 mm) is mandatory, whose rigidity is guaranteed by gluing it to a thick substrate, with the buried ground plane, to finally design a 4-layer circuit.

PocketSDR is designed as a 4-layer circuit. One might think that with a little effort, both MAX2771 and the FX2LP interface could be routed on a double layer. However, a key point in handling signals around 1.5 GHz is impedance matching of the track carrying the radiofrequency signals. A quick calculation with KiCAD allows to see that only a 4-layer solution allows for adaptation to 50 Ω – the standard

impedance of radio frequency transmission lines matched to their loads – with a track width compatible with the pin spacing of a surface mounted integrated circuit, and that this adaptation is impossible on a 1.6 mm thick FR4 substrate. The KiCAD calculator automatically loads with the parameters of the FR4 substrate, namely a relative permittivity of $\varepsilon_r \simeq 4.5$ and losses expressed as $\tan\delta \simeq 0.02$ (Fig. 3). If, with a substrate of thickness $H = 1.6$ mm, we adapt a "coplanar waveguide with ground plane" with a conductor-ground spacing of 0.2 mm, then the line must be 1.08 mm wide to present an impedance of 50 Ω, and must necessarily shrink when reaching the component whose legs are spaced 0.4 mm apart and are 0.2 mm wide (28-pin QFN package), creating an impedance mismatch and thus a reflection of part of the incident signal. Increasing the spacing between the track and the ground plane worsens the situation, with a track 1.94 mm wide if we leave 0.5 mm between it and the ground plane. The only chance of reaching 50 Ω with a track 200 μm wide is to reduce the thickness H of the dielectric substrate separating the track from the buried ground plane. With a thickness of $H = 0.2$ mm, which we find as the external layer thickness from most printed circuit board manufacturers with a total thickness of 1.6 mm, a trace width of 0.34 mm will be adapted to 50 Ω with a separation of 0.2 mm from the ground plane, or as proposed by PocketSDR with an isolation (transmission line distance to the ground plane) of 0.1524 mm, then the conductor will ideally be 0.315 mm wide. Tomoji Takasu has his printed circuits made at JLCPCB, where the external layer is 0.2104 mm thick (`https://jlcpcb.com/impedance`) with a substrate of relative permittivity of 4.1, thus a trace width of 0.3556 mm corresponds well to the value seen in the PocketSDR routing. At EuroCircuits, the "FR4 Improved" substrate has a relative permittivity of 3.9 but a thickness of 0.36 mm, so the trace 0.3556 mm wide and separated from the ground plane by 0.1524 mm presents an impedance of 60 Ω. We might have had to redo the calculation *before* outsourcing the manufacturing to EuroCircuits and increase the trace width to 0.515 mm! Despite this design flaw, the voltage reflection coefficient is

$$\rho = \frac{50 - 60}{50 + 60} = 0.1$$

so energy conservation indicates that what is not reflected is transmitted $1 - (0.1)^2 = 0.99_2$, or 0.04 dB of losses due to impedance mismatch.

The resulting routing circuit occupies 35×31 mm, mainly constrained by the spacing between the two connectors of the FX2LP development board, and the manufacturing of the 4 layers costs 5 euros/printed circuit for 50 circuits at Eurocircuits or 1.6 euros/circuit at SeeedStudio – who, however, does not accept purchase orders – for the same volume.

We spent the first episode of this series of articles familiarizing with the low-cost development board that includes an FX2LP and the necessary peripherals, arguing in particular that the complete board from Singapore is cheaper than purchasing the component alone locally. Consequently, starting from the schematic and routing of PocketSDR, we eliminate from the original circuit all the parts related to digital communication and route only the relevant signals, which are the IQ signals and the clock that controls their updates, the SPI bus signals including the two CS# activations, and a pair of status signals from the radio frequency receivers (Fig. 4).

Japanese electronic kit Journals

Japan has a culture of developing electronic circuits that France only knows in a marginal way – at least thanks to this publication – particularly with the distribution of journals by CQ Publishing (`https://www.cqpub.co.jp/`) including *Transistor Gijutsu* and *FPGA Magazine* and various books on microcontrollers and other embedded digital circuits. Even though the Akihabara district in Tôkyô has lost some of its splendor from the 1990s, shifting from the sale of electronic components to manga and products derived from animated series, a few stores such as Akizuki Denshi (literally "Akizuki's Electronics Store", `https://akizukidenshi.com`) continue to survive and provide kits in the form of small bags containing the printed circuit board and components to assemble yourself. In this vein, books containing a kit and descriptions of associated experiments, priced at around thirty euros (3000 yen), are available in many bookstores. For illustration, one of the books acquired in 2018 on the ARM Cortex M3 LPC1343 (left) comes with its kit and a CD containing software – unfortunately exclusively for MS-Windows (middle) – and an example of a page describing a peripheral followed by code examples (right).

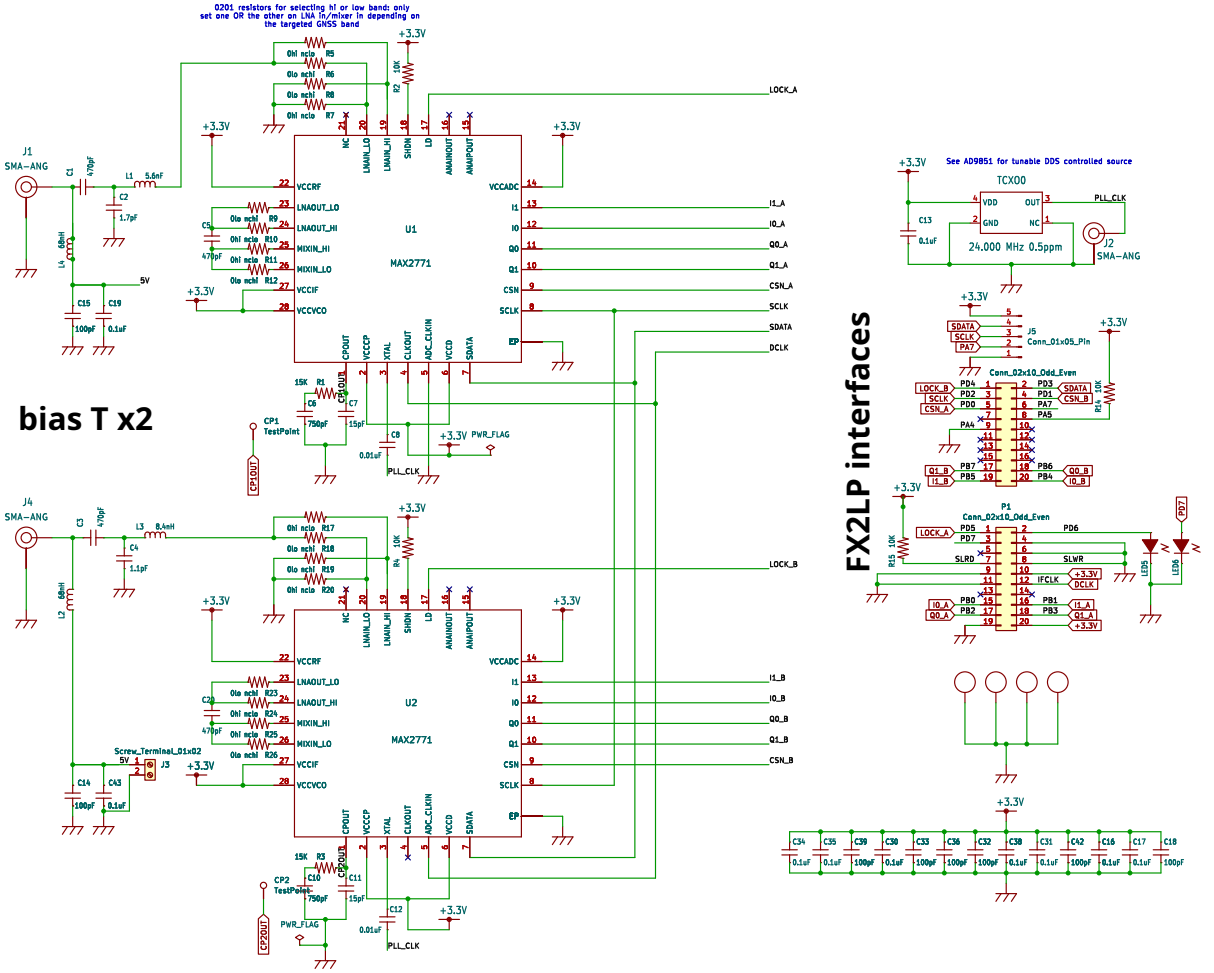


Figure 4: Schematic of the circuit, with the two MAX2771 where the clock of one is slave to the other to ensure synchronization of acquisitions, and on the right, the terminals leading to the FX2LP development board as well as the signals similar to SPI exported to control an external component, for example, a programmable oscillator to replace the fixed TCXO which will then not be populated on the printed circuit board (top right). On the left, the two bias-T power the active antennas.

Left: cover of the book dedicated to programming a microcontroller based on an ARM core, unfortunately using an interpreted language "MicroBASIC" that is reminiscent of MicroPython (`https://micropython.org/`). Middle: the thick back cover contains the circuit equipped with the microcontroller and some connectors (top) along with the CD for MS-Windows to flash the firmware. Right: despite the use of BASIC, the description of the microcontroller's operation is detailed enough to include descriptions of interrupts and routing rules for the printed circuit board.

The close link between the authors of CQ Publications and Akizuki Denshi is highlighted in the introduction of the article on using an ARM processor as a software defined radio source at `https://toragi.cqpub.co.jp/Portals/0/backnumber/2021/01/p115.pdf`, which begins with "Circuits from this article assembled with components available from Akizuki." Such synergy would likely struggle to exist in France, where electronics stores are content to sell ammeter clamps and DIL format integrated circuits, if not LED string lights. The circuit presented in this article could surely have been the subject of such a kit format associated with the text, but the financial investment and uncertainty about sales volume make the approach uncertain.

Readers wishing to reproduce the circuit described in these pages can contact the author, as a number of printed circuit boards (unpopulated with their circuits) are available.

Compared to the MAX2771 evaluation board from Maxim IC, which only offers a single acquisition interface, we now have two MAX2771 available, which are distinguished during configuration by the synchronous bus resembling SPI, through the CS# activation signal. It is therefore necessary to ensure that the most significant byte of the USB Vendor Request is interpreted as the component number with which to communicate (Fig. 5) during transactions to configure the MAX2771.

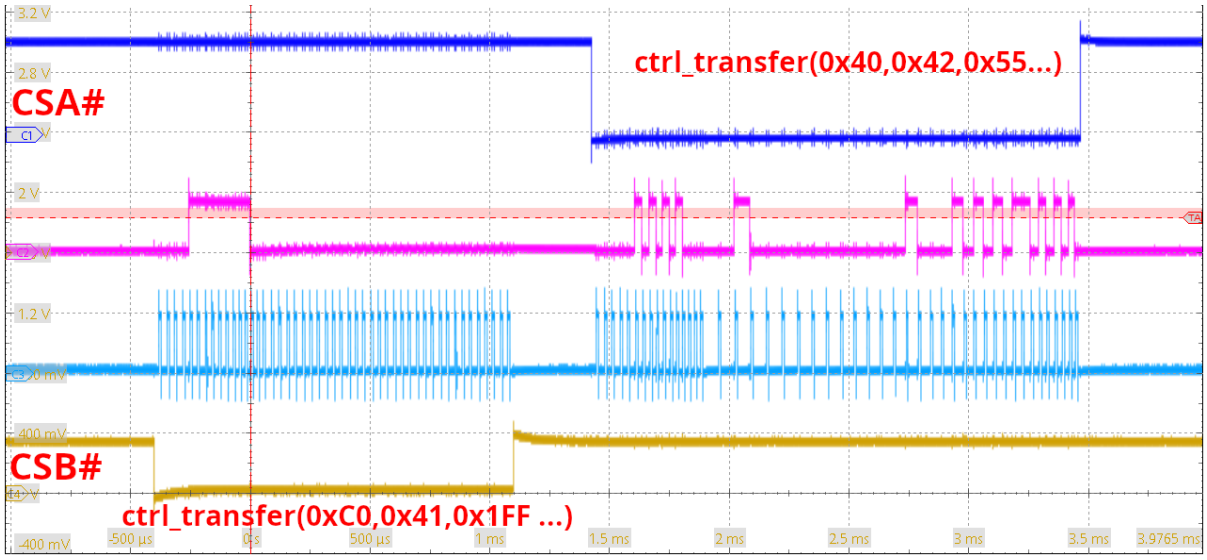


Figure 5: As in any SPI bus, each device receives the clock from the master distributed to all slaves and a unique activation signal, active low (CS#), is distributed to each slave. Unlike SPI, the synchronous data bus of the MAX2771 is shared in both input and output directions by all slaves (there is no separate communication lines from slave to master and from master to slave). Here we validate that the selection of either MAX2771 by the appropriate CS# is indeed activated based on the most significant byte transmitted in the Vendor Request since the most significant byte of 0 from 0x55 activates CSA# on top, and the most significant byte of 1 from 0x1FF activates CSB# (bottom).

We are convinced that we understand the interpretation of the Vendor Requests with the following sequence:

1. Starting from a flashed FX2LP with the content of `FX2LP/complete_fw/build/complete_fw.ihx` resulting from the compilation of the archive `https://github.com/jmfriedt/max2771_fx2lp/`, we configure the two MAX2771 using PocketSDR with

`sudo app/pocket_conf/pocket_conf pocket_L1L2_8MHz.conf` and validate that the communication rate is 8 MS/s with
`sudo app/pocket_dump/pocket_dump -t 1 1.bin 2.bin` which indicates

```
TIME(s)    T   CH1(Bytes)   T   CH2(Bytes)   RATE(Ks/s)
    1.0    I      7995392   I      7995392       7924.1
```

and `pocket_conf` without arguments indicates

```
#  [CH1] F_LO = 1573.420 MHz, F_ADC =  8.000 MHz (I ), F_FILT =  2.0 MHz, BW_FILT =  2.5 MHz
#  [CH2] F_LO = 1225.600 MHz, F_ADC =  0.000 MHz (I ), F_FILT =  2.0 MHz, BW_FILT =  2.5 MHz
```

2. We modify only bits 29 to 31 REFDIV of register number 3 (PLL Configuration) of the first MAX2771 – the master clock that drives the second MAX2771 – to change from 8 to 16 MS/s by doubling the clock frequency driving the ADC using the following Python script:

```
import usb
import binascii
import struct

VID = 0x04B4
PID = 0x1004
dev = usb.core.find(idVendor = VID, idProduct = PID)
msg=dev.ctrl_transfer(0xC0 ,0x41, 0x03, wIndex=0, data_or_wLength=4)
print(f"03␣before:␣␣{binascii.hexlify(msg)}")
msg[0]=msg[0]&(0x1F) # msg=bytearray([0x09,0x88,0x80,0x00]); # setting for 16 MS/s by XTal x2
dev.ctrl_transfer(0x40 , 0x42, 0x03, 0, msg)                 # write on SPI bus reg @ 0x03
ret=dev.ctrl_transfer(0xC0 ,0x41, 0x03, wIndex=0, data_or_wLength=4)
val=int.from_bytes(ret, "big")                               # big or little
print(f"03␣after:␣␣␣{val:#x}")
```

which starts by identifying the device by its Vendor ID=0x04B4 and Product ID=0x1004 as indicated by `lsusb | grep ypress`, then reads (Vendor Request of type 0x0C) the result of the SPI transaction initiated by the 8051 of the FX2LP (command 0x41) addressing register 0x03 to store the result in `msg`. The most significant byte (the first in the array) is masked with 0x1F to keep all bits except the 3 most significant bits forced to 0, thus determining REFDIV=0 for "XTAL frequency x2" (instead of x1 originally with 0x03). This approach is more elegant than forcing all bits (as commented) which could overwrite configurations supported by `pocket_conf` that we might not want to modify if we only want to impose a new sampling frequency. This new register is written (Vendor Request of type 0x40) to register 0x03, then read back (Vendor Request of type 0xC0) for validation with a display according to a different conversion mode than used previously. The result of this sequence is

```
b'69888000' 0x9888000
```

3. After reconfiguration, restarting `pocket_dump` as before now indicates

```
TIME(s) T CH1(Bytes) T CH2(Bytes) RATE(Ks/s) 1.0 I 16056320 I 16056320 15928.9
```

thus the configuration has been correctly applied with this time 16 MS/s.

4. Finally, to access a register of the second MAX2771, the register number provided as the third argument of `ctrl_transfer` becomes 0x103 instead of 0x03 to indicate that it is the second CS# that is activated, and not the first.

2 GPS signal reception

The benefit of using two receivers in parallel is either to simultaneously acquire the signals transmitted in two widely spaced frequency bands, for example L1 and L5 (the upper and lower parts of the L band), thus aiming for centimeter-level positioning by mitigating the contribution of the ionosphere on the propagation delay of the electromagnetic wave by taking advantage of its dispersion relationship, or to configure the two antennas on the same frequency band and analyze the direction of arrival of the signals in order to detect spoofing. Indeed, if a radio frequency source produces a signal representative of the coding and messages of GPS in order to spoof the receiver, it is highly likely that this source is unique, and thus all satellite signals appear to come from the same direction, which is impossible with a real constellation where the satellites are distributed over the celestial sphere. Therefore, identical phase between the signals acquired by two antennas for all satellites is a strong suspicion of spoofing; conversely, given the difficulty of synchronizing the transmitters for a distributed attack, a different phase for the signals received from various satellites indicates that the received signal is real. However, this analysis is only valid if the signals acquired by the two MAX2771s are synchronous, which the configuration of the clock of the ADC of the second MAX2771 as a slave to the first should normally guarantee, but this remains to be demonstrated.

Initially, we validate the ability of a PlutoSDR to produce a signal representative of GPS that can be acquired and decoded by our version of PocketSDR. To do this, PlutoSDR emits the signal produced by `pluto-gps-sim` from `https://github.com/Mictronics/pluto-gps-sim`. To achieve this, one simply needs to obtain a navigation file from the web at `https://cddis.nasa.gov/archive/gnss/data/daily/` (which requires providing an email address for registration) to position the satellites in space, and a signal, which is the sum of the signals produced by all the satellites, is generated by software and emitted by the PlutoSDR. This signal is split in two by a Mini-Circuits ZAPD-2-21-3W-S splitter to be injected into the two inputs of the two MAX2771s: all these signals will be seen with the same phase by the two receivers since the unique splitter introduces a single phase between its two paths. Note the use of DC blockers (high-pass filters, here Mini-Circuits MCL BLK-18-S, which cut the DC component but allow signals up to 18 GHz) between the output of the PlutoSDR and the inputs of the PocketSDR (Fig. 6, top): indeed, we have equipped the PocketSDR with bias Ts to power active antennas, and the PlutoSDR behaves like a short circuit to ground for these power supplies if we are not careful (balun acting as short circuits to ground of the DC component).

When we emit a signal representative of GPS and acquire it with the PocketSDR, we observe (Fig. 6) the coherence between the emitted signals (left) and the decoded ones (right and bottom), indicating that the spoofing works well. The power of the signal emitted by the PlutoSDR is adjustable and can be set above the thermal noise to facilitate the debugging of the PocketSDR circuit in case of malfunction.

If this acquisition is reproduced by connecting two active antennas to the two inputs of the PocketSDR, we observe the real signals from the distributed GPS constellation in the celestial sphere, thus introducing arbitrary phases between the two antennas. GPS signals are modulated using binary phase shift keying (BPSK), so a phase $\varphi(t)$ carries the information that can only take values of 0 or π depending on whether the transmitted bit is 0 or 1 (or the opposite). The rate at which $\varphi(t)$ is modified to carry the unique code for each satellite is 1.023 Mb/s, so the spectral spreading of this signal is on the order of 2 MHz, and the emitted power is distributed within this spectral range: the signal density (power per unit spectral, thus mW/Hz or more commonly dBm/Hz) is very low given the denominator of $2 \cdot 10^6$ Hz, and even below the spectral density of thermal noise of -174 dBm/Hz. The classic trick for detecting BPSK signals is to calculate the square of the signal $s(t) = \exp(j2\pi\delta f t + j\varphi(t))$ with δf being a frequency offset from the nominal carrier introduced both by Doppler shift and imperfections of the local oscillator:

$$s(t) = \exp(j2\pi\delta f t + j\varphi(t))^2 = \exp(j2 \times 2\pi\delta f t + j\underbrace{2\varphi(t)}_{0[2\pi]}) = \exp(j2\pi \times 2\delta f t)$$

and we find that all the energy distributed in the 2 MHz bandwidth has accumulated in a pure carrier at frequency $2\delta f$ since the phase modulation 2φ has been cancelled out, equaling 0 modulo 2π when squaring the signal, which produces double the argument. The Fourier transform of this signal will therefore be a series of spectral lines, one for each visible satellite, at a Fourier frequency of $2\delta f$, and by conservation of energy, a power that now exceeds the thermal noise and becomes visible on the spectrum.

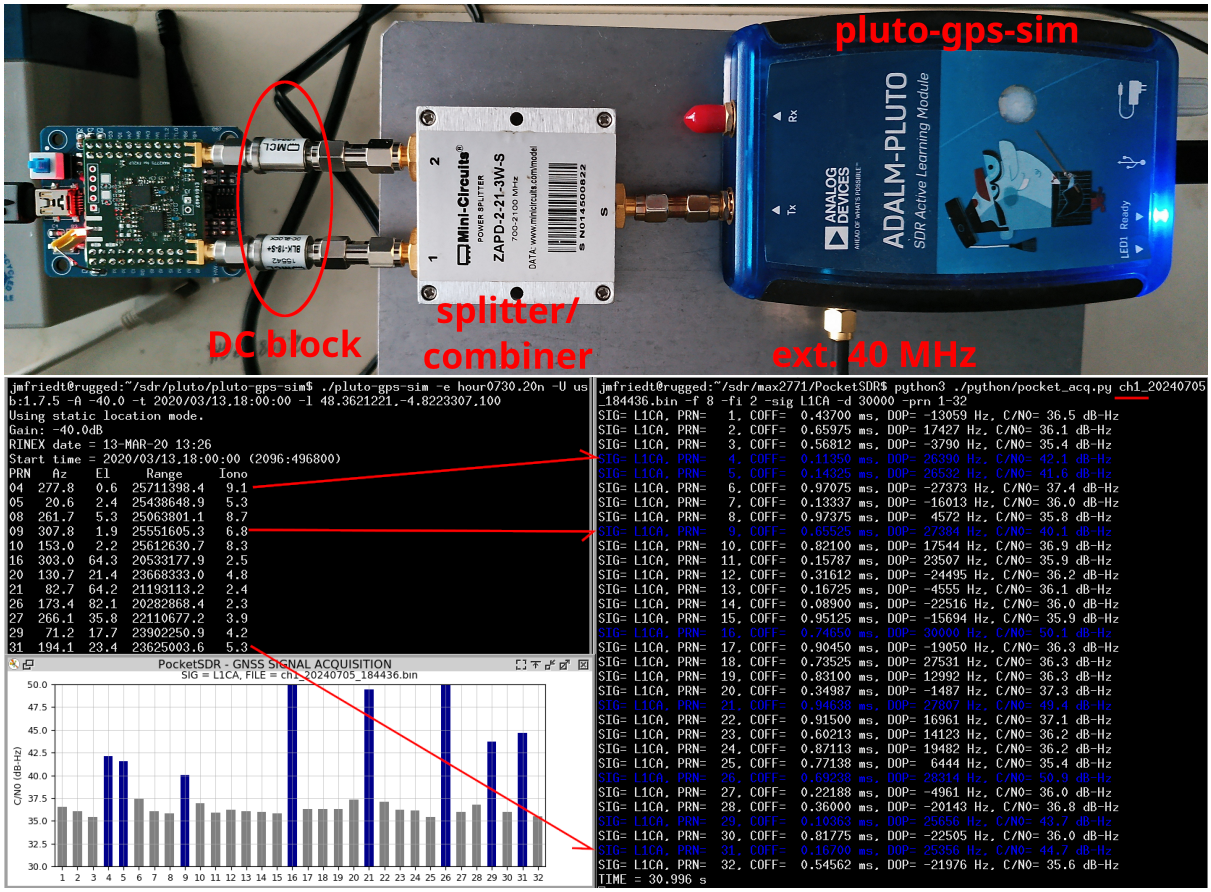


Figure 6: We mentioned the reception of Iridium as a method to facilitate the measurement of a signal above thermal noise. Another approach is to synthesize the L-band signal of a satellite navigation constellation, here using a PlutoSDR emitting the signal produced by `pluto-gps-sim`. Top: experimental setup with a splitter/combiner to inject the same signal into the two MAX2771s. Right: coherence of the signals emitted by `pluto-gps-sim` (left) and decoded by `python/pocket_acq.py` from PocketSDR (right and bottom).

The phase difference of the spectral lines at the same Fourier frequency between the two antennas is representative of the direction of arrival of the signal since the plane wave of wavelength λ reaching the two antennas separated by a distance d at an angle ϑ induces a phase difference of $\frac{2\pi}{\lambda} d \sin\vartheta$. We observe (Fig. 7) that the phases are uniformly distributed for the genuine GPS constellation (left) but that they all have the same value (right) in the case of spoofing.

Spoofing of frequency or phase modulated signals

GPS in particular, and GNSS in general, uses phase modulation, and given the low signal-to-noise ratio, it exploits few symbols in BPSK where the phase takes only two states, 0 or π, to encode one bit state or the other.

While one might think that analyzing the power of the received signal as an indicator of spoofing is somewhat naive, arguing that a suitably spoofed signal can reach the receiver with a power representative of that of a satellite, this assessment is not without meaning if the receiver's antenna is concurrently exposed to real signals as well as spoofed ones. Indeed, the question comes down to how much the power of the spoofing source needs to be increased to overwhelm the receiver, which is also seeing the real signal, thus making it believe it is in the wrong location at the wrong time.

Let's start by revisiting the case of frequency modulation (FM), well known for its "capture

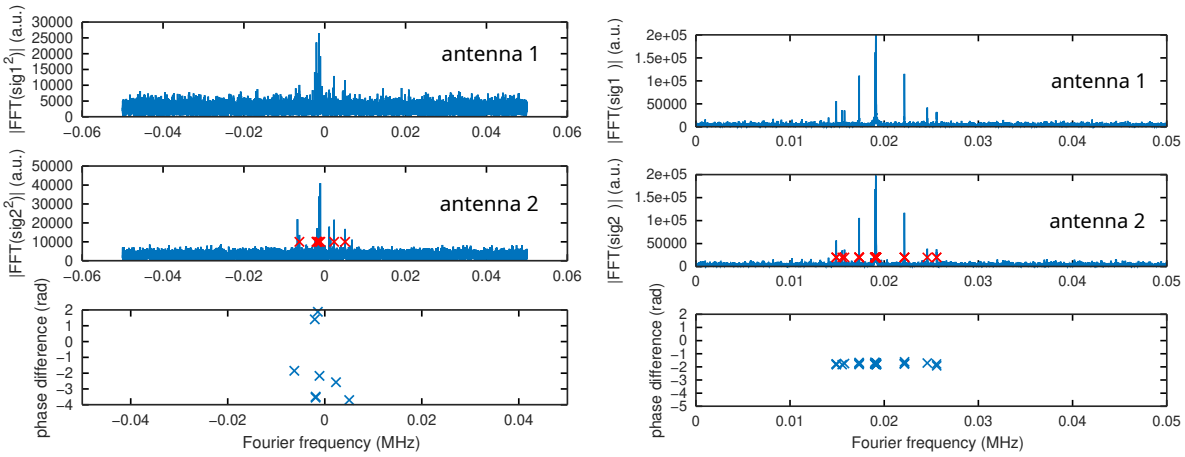
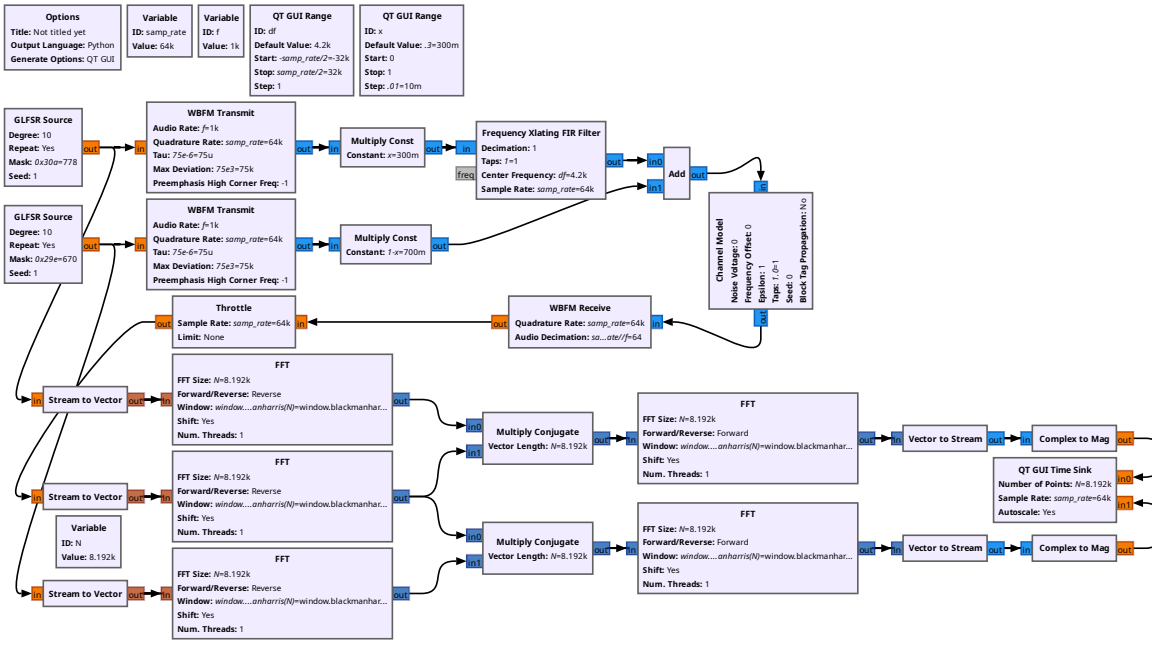


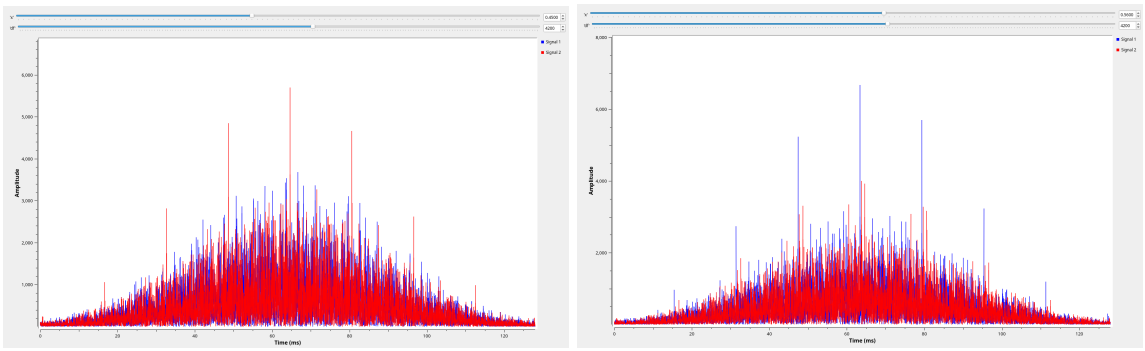
Figure 7: Analysis of the signal acquired from the GPS constellation by squaring the signal to eliminate the spectral spreading of the BPSK modulation and analyze the phase of the resulting pure carrier: each pure carrier shifted by the Doppler effect for each satellite is visible as a discrete line on the upper spectrum (antenna 1) and the middle (antenna 2). For the frequencies common to both spectra (satellite visible by both antennas, indicated by red crosses), the phase difference is displayed in the graph below, representative of the direction of arrival of each signal. On the left, these phases are distributed in the range $[-\pi, \pi]$ for the true constellation since the direction of arrival of the signal depends on the position of the satellite on the celestial sphere. Right: case of a spoofed signal produced by PlutoSDR and injected onto both channels. This time the phase for all satellites is the same. The oscillator (TCXO) of the PlutoSDR is clearly offset by a few hundred Hz from its nominal value, as illustrated by the non-zero average frequency around which the lines are distributed: 20 kHz at 1575.42 MHz corresponds to a deviation of 508 Hz at 40 MHz.

effect" [5] as presented by Tom Rondeau [6], in which an interfering signal barely more powerful than the true signal will latch onto the phase-locked loop of the FM demodulator, which will then only output information from the interfering signal, to the detriment of the true signal that has disappeared from the output.



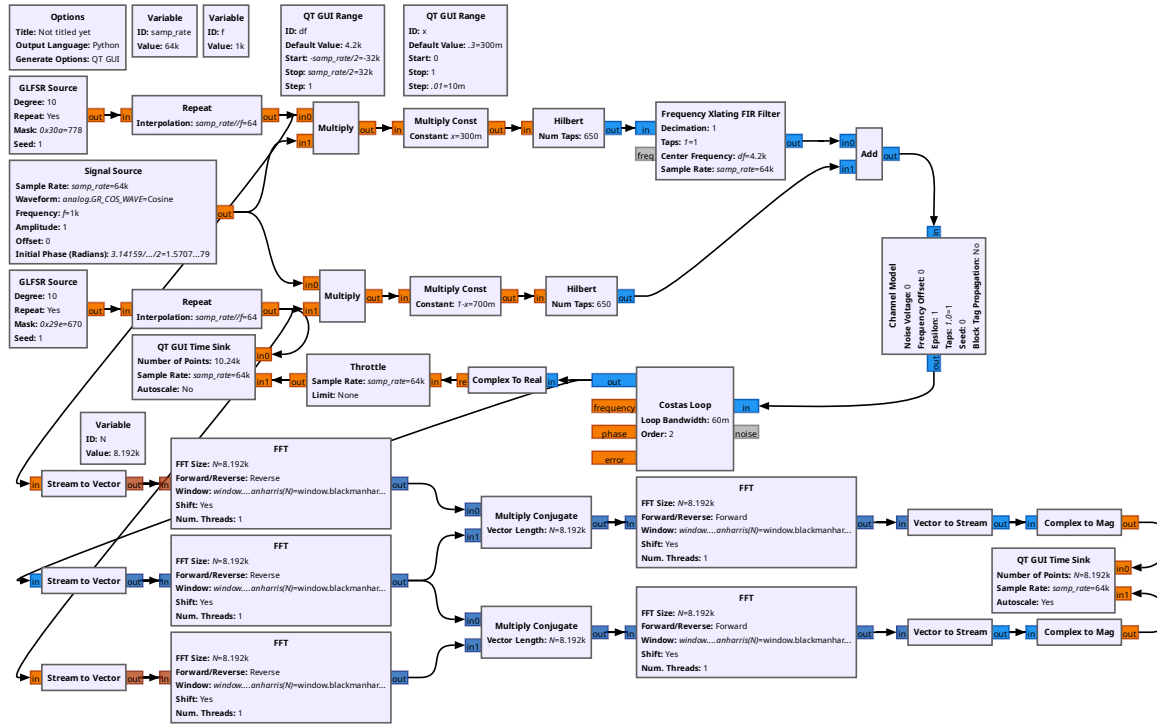
Case of frequency modulation (FM): GNU Radio Companion processing chain to modulate two pseudo-random signals in FM and calculate the correlation in the Fourier domain of the demodulated signal with respect to the transmitted signals.

In order to demonstrate this concept, two pseudo-random codes are transmitted. These codes are said to be orthogonal in the sense that the cross-correlation of one with the other yields a zero result (up to noise) while the autocorrelation of the code with itself produces a strong correlation peak at each repetition period (sequence length). The correlation is calculated in GNU Radio as the inverse Fourier transform iFT of the product of the complex conjugates of the Fourier transforms FT of the signals: $xcorr(u, v) = iFT(FT(u) \cdot FT^*(v))$ with * denoting the complex conjugate. Indeed, we note below that if one signal is only slightly stronger than the other (cursor at the top of the graphs), then only its correlation peaks (red on the left, blue on the right) stand out from the noise. In fact, the phase-locked loop (PLL) that acts as the frequency-to-voltage converter in the FM demodulator only locks onto the stronger signal and filters out the others.

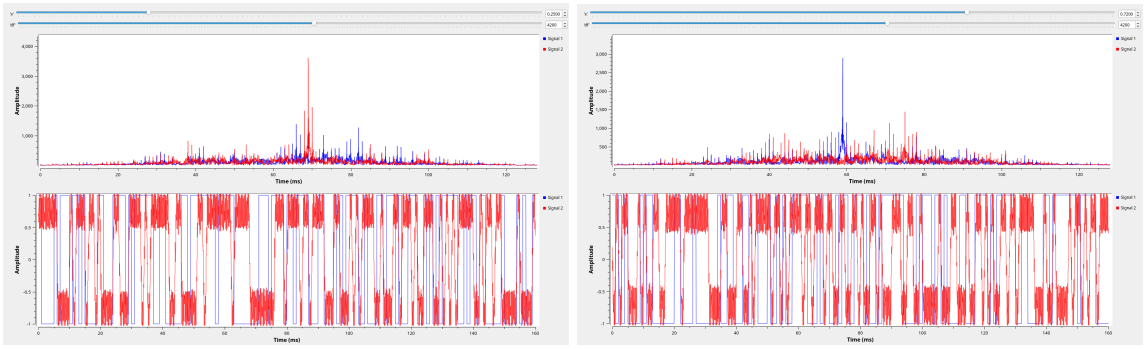


Correlation of the demodulated signal with the two transmitted signals (red and blue): on the left, the red signal is barely more powerful than the blue signal (cursor at 0.45), on the right, the blue signal is barely more powerful (cursor at 0.56), close to 0.5, the average value that equalizes the two contributions of the incident signals.

On the contrary, in the case of BPSK phase modulation illustrated below, the Costas loop that reproduces the carrier of the incident signal to allow for phase extraction is disturbed if the signal to be decoded is not sufficiently powerful. When the sum of two phase-modulated signals reaches a Costas loop, no usable output is produced if the signal-to-noise ratio of one of the signals compared to the other is not sufficient:



GNU Radio Companion processing chain for binary phase modulation (BPSK) of two pseudo-random signals, and correlation in the Fourier domain to identify which signal has been properly detected.



Result of the correlation for one signal being more powerful than the other as indicated by the cursors at the top of each graph, on the left for a relative level of 0.25 and on the right of 0.72, much further from the average value of 0.5 than for FM modulation. For a cursor position closer to 0.5, the output of the Costas loop is unusable and the result of the correlation remains in the noise, without a visible peak indicating that either code could have been decoded.

It should be noted from these two examples that while for an FM signal, it is sufficient for one of the two signals to be slightly more powerful than the other for the PLL to output a signal solely associated with the pseudo-random code modulated on the stronger signal, in the case of phase modulation, the signal-to-noise ratio of the signal feeding the Costas loop must be much higher to allow for demodulation of the stronger pseudo-random code. Without this condition of a sufficient signal-to-noise ratio, the Costas loop provides an unusable signal containing neither of the modulated codes.

Thus, dazzling a GNSS receiver involves subjecting it to a significantly stronger spoofing signal than the original signal, a power jump that would be a probable attack signature, as U-blox informs us in the documentation for its ZED-F9P receiver stating: “A detection is successful when a signal is observed to transition from an initially genuine one to a spoofed version. Hence detection is not possible if the receiver is started under spoofing conditions.”

3 Using a programmable frequency source

GPS is primarily a constellation of atomic clocks orbiting the Earth, and the information transmitted to the ground must enable the reproduction of the stability of the onboard clocks. Since the quartz oscillator that clocks the software-defined radio reception circuit has no reason to be stable or accurate, we must replace it with a programmable source to correct the frequency using information from the GNSS receiver via software. Indeed, `gnss-sdr` provides the time offset information between the GPS signal and the local oscillator, even though this information is somewhat hidden and not prominently exposed to the user. The objective of this paragraph is therefore to replace the fixed 24 MHz oscillator that clocks the MAX2771 with a programmable frequency source, with sufficiently fine granularity to correct the oscillator's drifts relative to GPS. The result is a *GPS Disciplined Oscillator* or GPSDO, widely available commercially, but implemented here with the benefit of two receivers in parallel, thus capable of detecting spoofing, and even attempting to cancel the interference source by creating destructive interference in the direction ϑ by introducing a phase opposite to that previously detected [7].

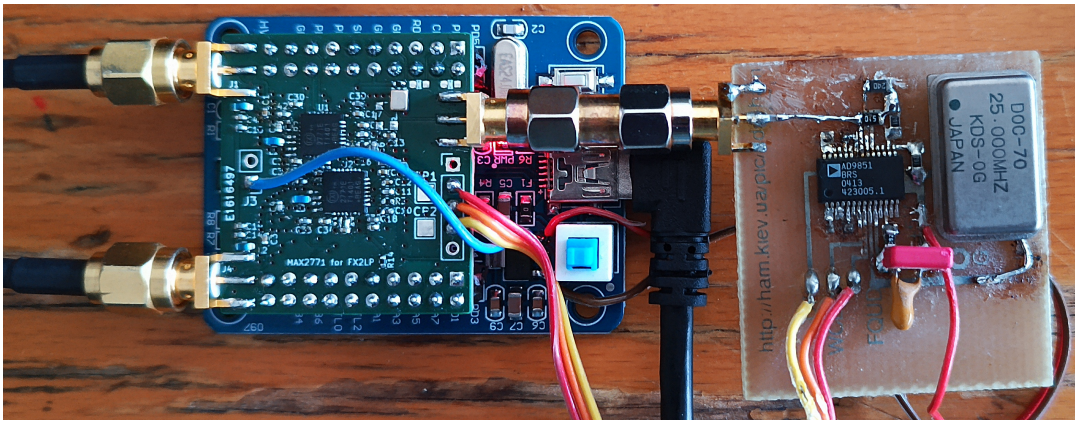


Figure 8: MAX2771 circuit (left) clocked by a digital frequency synthesis DDS AD9851 (right), itself clocked by a 25 MHz quartz oscillator multiplied internally to produce a 150 MHz clock. The DDS is programmed by the FX2LP via the same bus resembling SPI but is activated by its own signal named FQ_UD (*frequency update*).

Our choice of frequency source is the venerable digital synthesizer (*Direct Digital Synthesizer* – DDS) from Analog Devices AD9851 (Fig. 8). This component, very simple to use and route, offers a configuration of its frequency word *FTW* on 32 bits, thus providing a relative resolution of $1/2^{32} = 2 \cdot 10^{-10}$ or, with a reference clock of $25 \times 6 = 150$ MHz, a resolution of 35 mHz, which, while not exceptional in terms of oscillator stability, remains at least three orders of magnitude better than the few ppm (parts per million) variations of a quartz oscillator due to aging and ambient temperature fluctuations. From the datasheet, we learn that the AD9851 is programmed with a 40-bit word consisting of 32 bits of frequency followed by 8 bits of control and phase, with the least significant bit first, according to a synchronous protocol that could be wrongly associated with SPI but only has its appearance. Indeed, instead of activating the component with a CS#, the AD9851 always shifts its register on each clock pulse of the synchronous bus, and transfers the result to the corresponding configuration registers on the rising edge of a pulse from a pin named FQ_UD. The problem with this approach is that the shift register continues to be modified even while we are communicating with the MAX2771, resulting in an uncertain state when communicating with the AD9851. The workaround we found for this unstable state is to reinitialize the shift register with a pulse on FQ_UD just before communicating, send the 40 bits, and validate them with another FQ_UD pulse, ensuring that no other transaction has occurred on the SPI bus in the meantime (by design of the program on the 8051).

We stated in the first episode that we understood how *Vendor Requests* work: communicating with the AD9851 is an opportunity to prove this by adding a new command to communicate with this device. We will hence need to add a few functions to manipulate FQ_UD, which we have connected to PA7.

The new *Vendor Request* is added to the *firmware* of the FX2LP as

```
#define VR_AD9851    0x50       // USB vendor request: Write AD9851
BOOL handle_vendorcommand(BYTE cmd) {
 uint32_t val32;
 uint8_t ctrl;
...
 case VR_AD9851:
  {if (len < 5) return TRUE;
   EP0BCH = EP0BCL = 0;
   while (EP0CS & bmEPBUSY) ;
   val32=*(uint32_t *)EP0BUF;  // freq
   val32=bswap32(val32);
   ctrl=EP0BUF[4];
   write_AD9851(val32, ctrl);
   return TRUE;
   break;
  }
```

which calls the communication functions with AD9851

```
#define CSNAD9851    7          // EZ-USB FX2 PA7  -> AD9851 FQ_UD

static void digitalWriteA(uint8_t port, uint8_t dat) {
  OEA |= BIT(port);
  if (dat) IOA |= BIT(port); else IOA &= ~BIT(port);
}

static void write_AD9851(uint32_t freq, uint8_t ctrl) {
  int8_t i;
  jmfdelay(SCLK_CYC);
  digitalWriteA(CSNAD9851, 1);    // FQ_UD pulse
  jmfdelay(SCLK_CYC);
  digitalWriteA(CSNAD9851, 0);
  jmfdelay(SCLK_CYC);
  for (i = 0; i < 32; i++)        // AD9851 is LSB first
     write_sdata((uint8_t)(freq >> i) & 1);
  for (i = 0; i < 8; i++)
     write_sdata((uint8_t)(ctrl >> i) & 1);
  jmfdelay(SCLK_CYC);
  digitalWriteA(CSNAD9851, 1);    // FQ_UD pulse
  jmfdelay(SCLK_CYC);
  digitalWriteA(CSNAD9851, 0);
}
```

which, just like the original `write_reg()` by Tomoji Takasu, emulates synchronous communication in software, but this time activates the PA7 pin and, above all, communicates the least significant bit first (and not the most significant bit first as required by the MAX2771). Since the clock and data bus are common to all components, we utilize the original `write_sdata()` function. Finally, we configure the AD9851 at the initialization of the 8051 to emit a signal at 24 MHz with

```
void setup(void) {
...
  digitalWriteA(CSNAD9851, 0); // low rest state, rise to transfer
  write_AD9851(0x28F5C28F, 0x01);
```

To claim that it works would be quite presumptuous without verifying the spectrum of the signal produced by the AD9851. Instead of using the heavy artillery of a professional spectrum analyzer, we wish to use commonly available equipment, such as a PlutoSDR. However, we are producing a signal at 24 MHz, while the PlutoSDR is equipped with an AD9363 that can be reconfigured to appear as an AD9364, whose minimum measurement frequency is 70 MHz. Therefore, we need to resort to a trick to validate the proper functioning of the Python program that controls the AD9851 using

```
import usb
import struct
```

```
import sys
if (len(sys.argv)>=2):
    f=float(sys.argv[1])
else:
    f=24e6
fclk=150e6
ftw=int((f/fclk)*2**32)   # 687194767
msg=struct.pack('>I',ftw) # send LSB first and SDCC is little endian
msg+=bytes([1])
VID = 0x04B4
PID = 0x1004
dev = usb.core.find(idVendor = VID, idProduct = PID)
dev.ctrl_transfer(0x40 , 0x50, 0x00, 0, msg) # write msg on SPI bus
```

by producing the *Vendor Request* number 0x50 (in agreement with the definition of `VR_AD9851` in the firmware) followed by the 5 bytes of `msg` containing the frequency `ftw` followed by the control word.

The trick is to take advantage of the discrete-time sampling of the DDS output, whose datasheet explains very well the shape of the spectrum in the form of a line at the desired frequency f, two lines on either side of the clock f_{CK} that clocks the DDS at $f_{CK} \pm f$, and copies of this signal for all harmonics of the clock frequency. The power of these lines is weighted by a cardinal sine envelope $sinc(x) = \sin(\pi x)/(\pi x)$, which vanishes for integer values of x that we will take equal to the frequency normalized by the sampling frequency. Thus, by choosing to clock the AD9851 with a 25 MHz oscillator and taking advantage of its internal frequency multiplier by 6, the DDS is clocked at $f_{CK} = 150$ MHz, and producing an output signal at 24 MHz results in two spectral lines at 150 ± 24 MHz, namely 126 MHz and 174 MHz, both within the measurement range of PlutoSDR. We focus on the second one, close to the 7th harmonic of the 25 MHz oscillator, which will surely be visible on the spectrum and will serve as a reference to verify the proper functioning of the PlutoSDR even if the AD9851 configuration program fails. After a few debugging sessions, particularly on the endianness of the word transmitted from the PC to the FX2LP [1], we obtain the measurement presented in Fig. 9, which convinces us of the good communication between the Python program sending the output frequency programming commands to the AD9851 and the observed spectra.

3.1 Open loop measurement

In order to correct an oscillator, it is necessary to measure it first. In this application, the 25 MHz oscillator clocks the AD9851, which produces the 24 MHz that clocks the analog-to-digital converter of the MAX2771 and thus implicitly timestamps each of its samples, provided that none are lost. Thus, when `gnss-sdr` processes the data stream, regardless of the latencies or buffer sizes introduced by the USB communication, it is capable of retrieving the time at which each sample was acquired. Since the decoded IQ data stream carries the time information of the GPS constellation, `gnss-sdr` can compare the evolution of its local clock, materialized by the 24 MHz clocking the MAX2771 (it is important to note that only the clock that clocks the analog-to-digital converter matters, regardless of the frequency of the oscillators that clock USB or the processing unit downstream of the acquisition [8]) with the supposed perfect GPS time since produced by atomic clocks periodically compared to the primary references of the USNO in Washington. Even if the time manipulations are not documented by the USNO on the military-purpose system that is GPS, it is unlikely today, with the omnipresence of GPS and the number of users observing its performance, that this time deviates significantly from UTC, aside from leap seconds to maintain historical consistency with astronomical time.

We thus have on one hand `app/pocket_dump/pocket_dump` from PocketSDR that can acquire over USB and send the I(Q) stream into a file, and on the other hand `gnss-sdr` which can read a data stream and process it. However, if we ask `pocket_dump` to write to a named *pipe* created by `mkfifo /tmp/fifo1in`, then either `gnss-sdr` complains when opening such a file that it cannot determine its size, or if it is a FIFO interface, that it absolutely requires a complex stream and does not know how to handle a real stream without an imaginary part. Indeed, in order to free ourselves from the noise introduced by the various digital circuits in the baseband, we configured the MAX2771 to acquire a real-time stream with an intermediate frequency of 2 MHz at a rate of 8 MS/s. In this way, baseband noise sources are rejected after digitization of the signals through software transposition to eliminate the

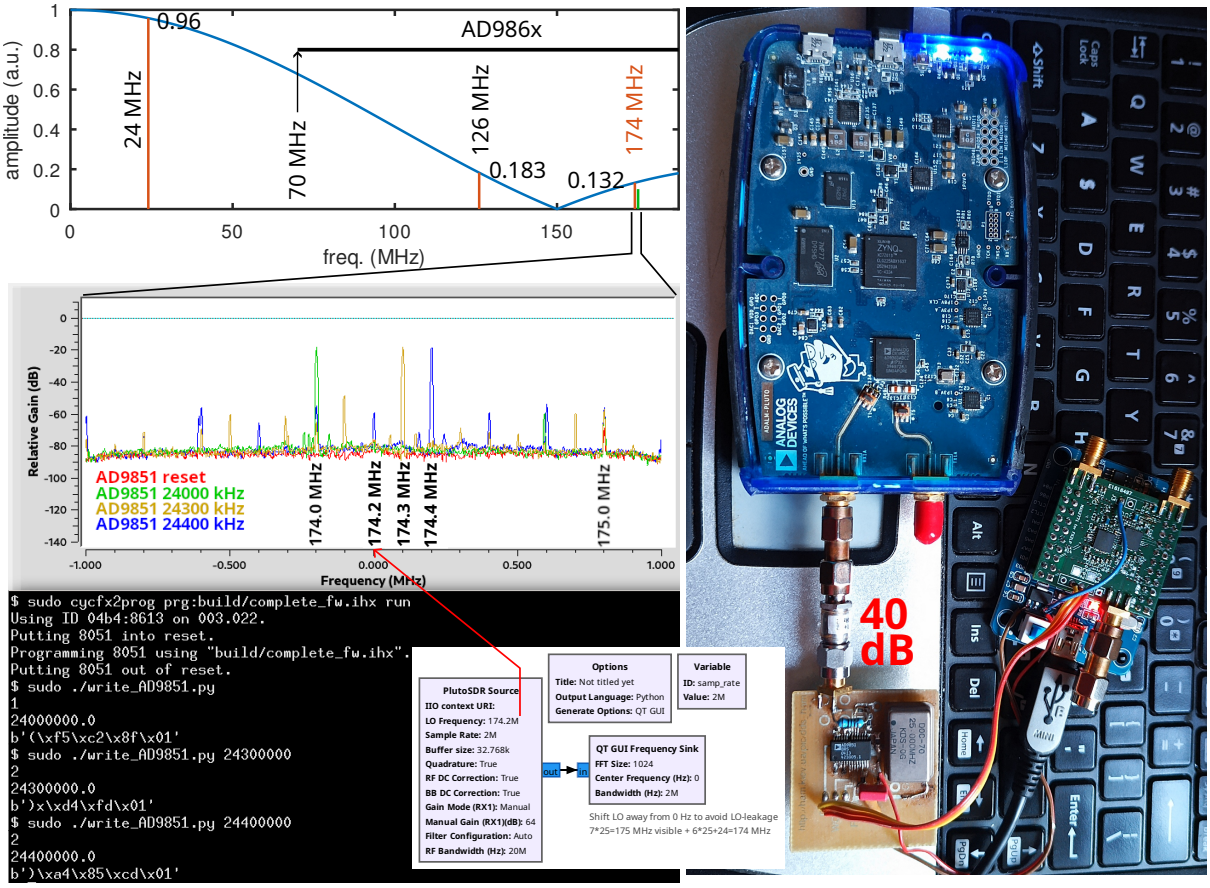


Figure 9: Top: simulation of the spectrum emitted by the AD9851 clocked at $25 \times 6 = 150$ MHz and producing a signal at 24 MHz by programming its frequency word, notably inducing a spectral line at 174 MHz, close to the 7th harmonic of the reference oscillator (green). Middle: measurement of the spectrum around 174 MHz over a bandwidth of 2 MHz by PlutoSDR, with its local oscillator intentionally offset by 200 kHz to avoid confusing the leakage of the local oscillator with the desired signal. In red, the signal observed in the absence of programming of the AD9851, when only a weak line at 175 MHz is visible. Programming the AD9851 to produce 24 MHz results in a strong line at 174 MHz, thus 200 kHz below the 174.2 MHz of the local oscillator, and reprogramming the AD9851 to produce other frequencies results in a coherent spectrum. Bottom: GNU Radio Companion acquisition chain to exploit PlutoSDR as a spectrum analyzer. Right: experimental setup, with the input of PlutoSDR connected to the output of the AD9851 clocked by a 25 MHz quartz oscillator. Note the 40 dB attenuator between the AD9851 and PlutoSDR to avoid damaging the PlutoSDR's AD9361 frontend by exposing to too powerful a signal. This attenuator will be removed to clock the MAX2771, here disconnected in this picture since only the FX2LP is used to program the AD9851 via USB.

intermediate frequency, simultaneously allowing to filter out-of-band rejected noise.

Thus, between the acquisition by `pocket_dump` of a real-time stream transposed to intermediate frequency and the decoding by GNSS-SDR, it is necessary to convert it into a complex stream compensated of its intermediate frequency and decimated. GNU Radio will handle this transposition (Fig. 10): a trivial processing chain simply reads from the pseudo-file that is the FIFO fed by `pocket_dump`, converts the 8-bit encoded integers into floats, transposes in frequency using the `Xlating FIR Filter` which is also responsible for filtering and decimating, before exporting its data stream to output. We attempted to write to another FIFO at the output for communication with `gnss-sdr`, but the result was disappointing, so we reverted to a UDP output using ZeroMQ Publish interface that `gnss-sdr` recognizes as a ZeroMQ Subscribe input [9]. Thus, `gnss-sdr` is capable of producing a position, velocity, and time (PVT) solution and displaying it on the screen [1]. However, only the acquisition date

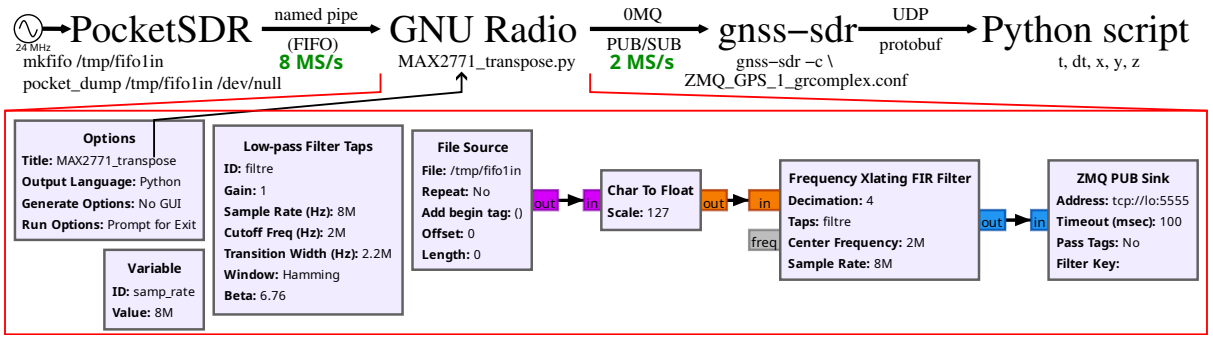


Figure 10: Processing chain for real-time analysis of GPS L1 signals, from PocketSDR for acquisition to `gnss-sdr` for producing the PVT solution (Position, Velocity, Time) and estimating the deviation between the local clock and the GPS clock, passing through GNU Radio for frequency translation. The GNU Radio Companion processing chain is highlighted in the red box, particularly with the definition of the anti-aliasing filter on the left applied before decimation by a factor of 4 in the `Xlating FIR Filter` to go from 8 to 2 MS/s.

is provided on the user interface, while the difference between the local time of the processing software and GPS time remains hidden within the PVT resolution functions. In the past [8], we modified the PVT solver to extract this information and use it to control an external oscillator locked to GPS, but this requires patching each new version of gnss-sdr, which of course changes in structure and necessitates manual insertion of functions. Fortunately, for a couple of years now, `gnss-sdr` provides means to export the state of its internal variables to software exploiting them, via a UDP link. So we can export the state of all processing channels of the PVT processing functions – the `Monitor` processing block described at `https://gnss-sdr.org/docs/sp-blocks/monitor/` – but what really interests us is the synthesis of the PVT solution, not the Doppler shift or the pseudo-distance of the receiver to each satellite (which are quite interesting otherwise). Therefore, what we are interested in is not to add the instructions `Monitor.enable_monitor=true` in the configuration file of `gnss-sdr`, but rather to add the display of the PVT processing solution, according to the instructions at the very bottom of `https://github.com/acebrianjuan/gnss-sdr-pvt-monitoring-client`, except for the last line (the one concerning `protobuf`), with

```
PVT.enable_monitor=true
PVT.monitor_client_addresses=127.0.0.1
PVT.monitor_udp_port=1234
PVT.enable_protobuf=true
```

Thus, a UDP data stream is output from a server provided by `gnss-sdr` and can be read by a client. Now that we have learned to communicate with the AD9851 in Python, we would like to not use the aforementioned client example in C++, but read this information from Python in order to use a single language for all operations. Therefore, we need to read the UDP data from Python (a client on a well-known UDP/IP socket) and interpret each field to retrieve the information that interests us. The documentation at `https://gnss-sdr.org/docs/sp-blocks/pvt/` teaches us that the data transmitted over UDP is serialized using a protocol proposed by Google called Protobuf. We discover that this protocol relies on a code generator targeting a multitude of languages from a configuration file specifying the fields, their size, and their location in the data stream. This description of the fields can be found in `gnss-sdr/docs/protobuf` in the file `monitor_pvt.proto`. Indeed, we find at the beginning of this file information such as

```
message MonitorPvt {
uint32 tow_at_current_symbol_ms = 1; // Time of week of the current symbol, in ms
uint32 week = 2;                     // PVT GPS week
double rx_time = 3;                  // PVT GPS time
double user_clk_offset = 4;          // User clock offset, in s ...
```

which starts well since `tow_at_current_symbol_ms` is the date, in milliseconds, of the analyzed symbol, and `user_clk_offset` is the information on the time offset between the GPS clock and our local clock

that clocks the analog-to-digital converter. Less interesting but still useful for verifying correct operation, the fields `pos_x`, `pos_y`, and `pos_z` give the position of the receiver in ECEF format, i.e., in meters from the center of the Earth. Thus, a Python class is automatically produced from this description by

```
protoc monitor_pvt.proto --cpp_out=. --python_out=.
```

in the file `monitor_pvt_pb2.py` which is utilized, provided that the `python3-protobuf` has been installed under Debian GNU/Linux, with the program

```
import socket
from monitor_pvt_pb2 import MonitorPvt
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind(("127.0.0.1", 1234))
foo = MonitorPvt()
m=0
while True:
    data, addr=sock.recvfrom(1500)
    foo.ParseFromString(data)
    print(f"{m}: TOW={foo.tow_at_current_symbol_ms} dt={foo.user_clk_offset}\
 x={foo.pos_x} y={foo.pos_y} z={foo.pos_z}")
    m=m+1
```

with `MonitorPvt` the keyword seen in the protocol definition at the beginning of `monitor_pvt.proto`. By launching this Python program, once `gnss-sdr` has identified an acceptable PVT solution, we obtain

```
0: TOW=108609900 dt=0.015872054942784843 x=4313730.75 y=452873.96 z=4661057.92
1: TOW=108610900 dt=0.015871647253655898 x=4313730.68 y=452873.96 z=4661057.89
2: TOW=108611900 dt=0.01587123573141401  x=4313730.67 y=452873.96 z=4661057.91
3: TOW=108612900 dt=0.015870821685370823 x=4313730.61 y=452873.96 z=4661057.87
...
```

in which a few overly optimistic decimals were removed from the position for the sake of compactness. We reassure ourselves that $\sqrt{x^2+y^2+z^2} = 6367011$ m or 6367 km, which is indeed the radius of the Earth at a latitude of 47°N: our receiver is indeed located on its surface!

Radius of the Earth

Historically, the meter was defined as the 10-millionth of a quarter of the circumference of the Earth, or in other words, the 40-millionth of the circumference of the Earth, which for a spherical Earth corresponds to a terrestrial radius of 6366.2 km since the length of the equator was then 40000 km. However, the Earth is not a sphere (without being flat) and its radius depends on latitude: in the first approximation of an oblate sphere (ellipsoid), the radius at latitude φ is, according to Wikipedia (demonstration at `https://fr.planetcalc.com/7721/`)

$$R(\varphi) = \sqrt{\frac{(a^2\cos\varphi)^2 + (b^2\sin\varphi)^2}{(a\cos\varphi)^2 + (b\sin\varphi)^2}}$$

with $a = 6378.14$ km and $b = 6356.75$ km (WGS84 convention). Thus, $R(47°) = 6366.7$ km, which is not so far from the observed value, especially if we consider that our antenna is positioned at an altitude of 357.379 m (PPP solution over 24 h of acquisition), and highlights the problem of defining the reference geoid when determining altitudes, which is much more difficult than latitude or longitude. We will return to this point in a later article.

However, more interesting is the trace of the GPS time offset from local time, which seems to be a straight line, indicated by a linear adjustment with a slope of $4 \cdot 10^{-7}$ or 400 ns/s. This offset, which may seem significant (Fig. 11, top), is only 0.4 ppm, quite reasonable for a low-cost quartz oscillator that does not claim accuracy or stability better than a few ppm.

If we numerically subtract the linear drift (frequency offset of the quartz from the nominal value), we are left with higher-order polynomial coefficients, of which the Allan variance provides stability as a function of the integration time. This Allan variance, calculated with its error bars thanks to

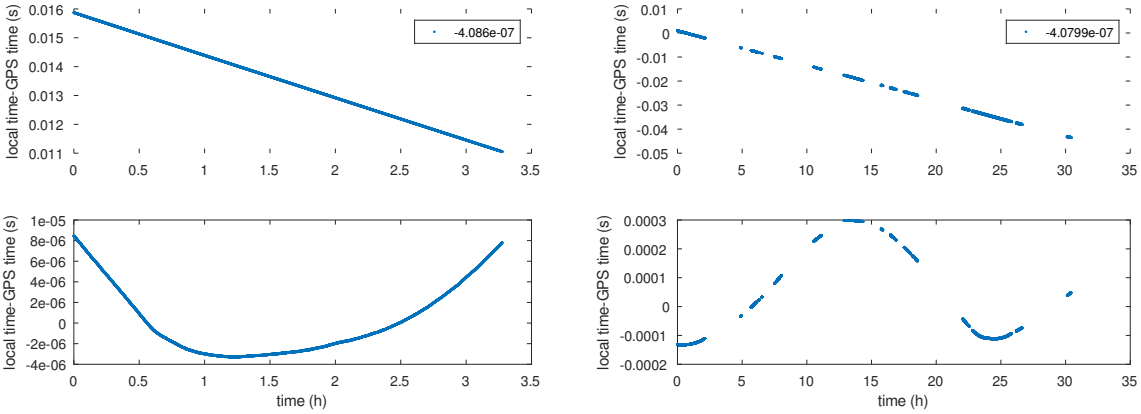


Figure 11: Temporal evolution of the time offset between the local clock and the GPS clock, on the left over a contiguous sequence of 3 h, on the right over a discontinuous sequence of a day showing the diurnal/nocturnal temperature fluctuations. At the top, the drift introduced by the frequency offset of -0.4 ppm due to the inaccuracy of the quartz oscillator. At the bottom, once this drift is numerically subtracted assuming the oscillator is properly frequency-adjusted, the residual must be dynamically compensated by a programmable frequency source controlled according to environmental conditions.

SigmaTheta from our colleagues at the time-frequency laboratory in Besançon available at `https://gitlab.com/sigmatheta1`, requires converting the offsets between GPS date and local oscillator date into differences of time offsets, and then calculating the Allan variance. Assuming the GPS time offsets to the local clock are stored in ASCII in the file `df`, then `X2Y df` produces `df.ykt` as the difference of successive `df`, and `SigmaTheta df.ykt` generates the graph in Fig. 12. We observe that the curves on this graph present a minimum at 200 seconds after which the variance rises in the absence of correction and due to thermal fluctuations: this is the time constant with which to program the AD9851 to make the best use of the short-term stability of the quartz oscillator and the long-term stability of GPS.

Stability of oscillators

Not all oscillators are equal in the face of the god of stability. We initiated this study by locking the MAX2771 with a 24 MHz surface-mount oscillator that only delivers a few hundred mV of amplitude, sufficient for the MAX2771 but not for the AD9851, which requires a TTL-compatible voltage. Pulling out an old oscillator in a metal case that meets these requirements (Fig. 8), it is clear that several decades of development on oscillators have significantly improved their stability, as illustrated in the figure below on the left. An estimator to quantify the stability of a quantity as a function of integration time is the Allan variance, which observes the standard deviation of sliding averages over various durations τ provided on the x-axis. An oscillator that drifts will always see its Allan variance increase since regardless of the integration time, measurements for large τ will be far from the average value. In contrast, a stable oscillator that fluctuates around an average value will see its Allan variance decrease since averaging reduces noise. Numerous studies analyze the nature of the noise based on the slope at which the Allan variance increases or decreases and the physical cause associated with these noises. The calculation of Allan variance is far from trivial, and rather than using the Python codes from `allantools` by Anders Wallin and his colleagues at `https://github.com/aewallin/allantools`, we rely on the bisontine software suite at `https://gitlab.com/sigmatheta1/` to plot the curves presented in this article (figure below, right).

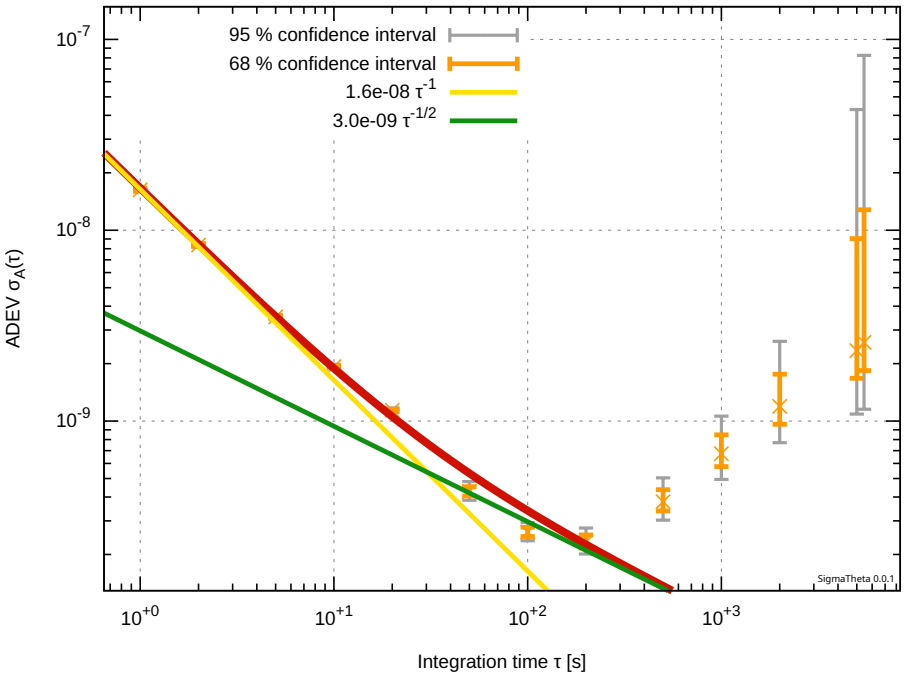
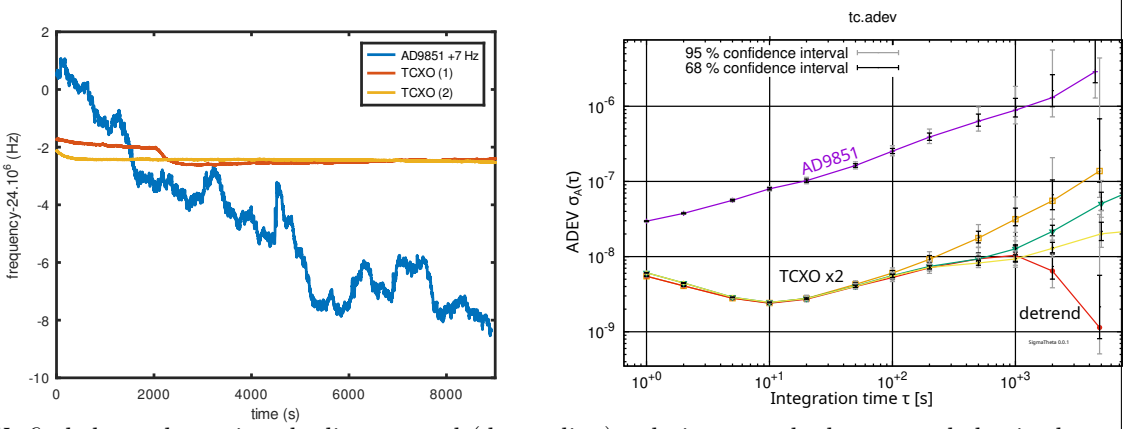


Figure 12: Allan variance indicating the fluctuation of the time offset between local clock and GPS clock as a function of integration time: the minimum of the Allan variance indicates the time constant of the control system below which overall performance would degrade by not fully leveraging the short-term stability of the quartz oscillator, and beyond which thermal drift would hinder long-term stability. This curve results from analyzing the data in Fig. 11 (left).



We find that subtracting the linear trend (detrending) only impacts the long-term behavior but that the short-term behavior is independent of this slow trend. Note that to achieve this result, it is essential to save the pre-processed data by GNU Octave in `save -text` format, which preserves all decimals, since the scientific notation of `save -ascii` retains only 6 decimals, insufficient for studying relative stabilities of a few 10^{-8} at a second. The curve labeled "AD9851" refers to the metal oscillator visible to the right of Fig. 8, while "TCXOx2" refers to the two measurements of the CMS oscillator with which we equipped our development boards for MAX2771, reference Epson TG2520SMN 24.0000M-MCGNNM3.

Knowing that we obtain an estimate of the time error every second and that the optimal time constant is 200 s, we might be tempted to process only one point every 200 seconds, which would obviously

be foolish, depriving us of the ability to average 200 successive observations before applying a correction. A recursive infinite impulse response filter of the form $y_{n+1} = a \cdot x_n + b \cdot y_n$, thus producing a new output as a weighted sum of the previous inputs x_n and outputs y_n, can take various forms depending on the coefficients a and b. Thus, `https://tomroelandts.com/articles/low-pass-single-pole-iir-filter` teaches that a filter of the form

$$y_{n+1} = y_n + c \cdot (x_n - y_n) \Leftrightarrow y+ = c \cdot (x - y)$$

when the `+=` operator exists (taking the old value of the variable and assigning it the sum with the difference between the new input and its old value, weighted by the coefficient c), then the time constant τ is determined by $d = 1 - c$ such that $\tau = -1/\ln(d)$ or $d = \exp(-1/\tau$. If $\tau = 200$ s, then $d = 0.995$ and $c = 5 \cdot 10^{-3}$.

Since we have a separate process in Python for acquiring and processing data capable of measuring the time and frequency deviation from `gnss-sdr`, and on the other hand, we are able to program the AD9851 in Python by sending *Vendor Requests* to the FX2LP, all that remains is to insert the control law we have just determined to periodically correct the oscillator and bring it to its nominal value aiming to correct the frequency bias (Fig. 13).

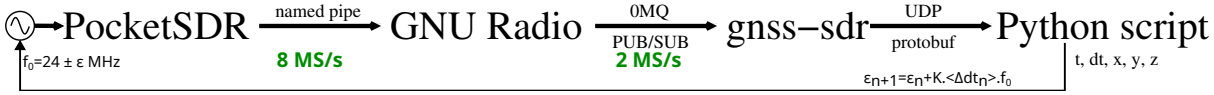


Figure 13: The same acquisition and processing sequence as in Fig. 10, but this time feedback aims to correct the local oscillator in order to compensate for its variations based on environmental conditions, particularly the random fluctuations introduced by temperature, in addition to correcting the systematic bias introduced by the inaccuracy of the quartz oscillator.

During these extended measurements, we discovered an anomaly in the timestamps extracted from the GPS navigation messages every Saturday night: it relates to the end of the GPS week [10] and the transition to the new week, a transition that `gnss-sdr` does not seem to handle correctly. Carles Fernández-Prades, the author of `gnss-sdr`, noted this malfunction as a bug in the GitHub error report at `https://github.com/gnss-sdr/gnss-sdr/issues/787`, but the PVT solver of `gnss-sdr` is a complex piece of software that will certainly not be easy to fix. Furthermore, `gnss-sdr` is supposed to provide in the TOW variable (Time of Week), in milliseconds, the symbol number for which the delay was observed. The symbols of the GPS L1 C/A navigation messages are transmitted every 20 ms, and the delay `user_clk_offset` is only between 0 and 20 ms: it is therefore necessary to know which symbol corresponds to the delay to resolve the ambiguity modulo 20 ms. With time drifting at a rate of 400 ns per second (0.4 ppm), we expect to change the symbol index every $20 \cdot 10^{-3}/400 \cdot 10^{-9} = 50000$ seconds or about 14 hours, but this is not the case. Indeed, during acquisitions much longer than 14 hours, the symbol index stubbornly remains the same. This behavior does not seem normal.

3.2 Frequency control

Our goal is to produce a true stabilized oscillator (and secured against potential spoofing detection), and thus to control the AD9851 locking the MAX2771. This seemingly trivial task—removing the 24 MHz oscillator to replace it with the output of the AD9851—turned out to be much more challenging than expected, since the MAX2771 did produce clock and IQ signals to the FX2LP, but no USB communication reached the PC. After exploring the reference signal power—the technical documentation of the MAX2771 mentions a 0.5 V*pp* sine wave as the "reference input level" while the AD9851 produces barely 0 dBm or 0.6 V*pp* – we verified that the timing is functional with −20 dBm. It turns out that the solution lies in a subtle combination of starting the AD9851 early in the initialization sequence of the FX2LP and filtering out the unwanted signals that allowed us to verify the correct operation of the AD9851 with the PlutoSDR. The timing of starting the oscillator locking the MAX2771 remains a mystery. On the other hand, for filtering, instead of struggling with inductors and capacitors that never behave as expected at radio frequencies, we will repurpose a quartz resonator to use it not as a frequency-selective element in an oscillation loop as it is typically used, but as a transmission filter. It just so happens that a 24 MHz resonator was lying around in our spare parts, allowing to easily test this principle (Fig. 14).

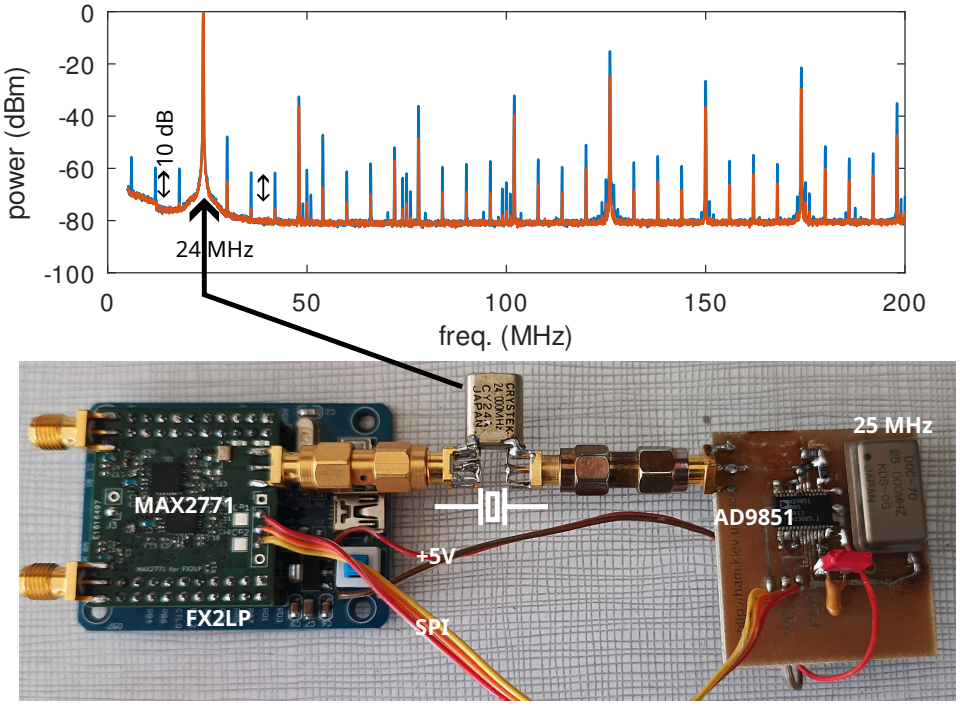


Figure 14: Spectrum at the output of the DDS AD9851 before filtering (blue) and after filtering (red) by the quartz resonator (bottom). Without this filtering, the MAX2771 refuses to communicate its measurements to the USB bus. The vertical arrows indicating 10 dB of attenuation guide the eye towards the effective filtering of the low-frequency spectral components around 24 MHz, but this filtering capability degrades rapidly at higher frequencies.

The bulk acoustic resonator is a quartz slab coated on both flat faces with electrodes: the applied electric potential to the dipole results in a displacement within the crystal lattice of the quartz due to the inverse piezoelectric effect, and this displacement generates a strain in the material that produces charges and thus a current via the direct piezoelectric effect. Thus, from the electronic engineer's perspective, a quartz resonator is a dipole presenting a resonance that is increasingly sharp as the losses in the material are low, at a resonance frequency determined by the thickness of the quartz slab. These two electrodes on either side of the dielectric that is quartz form a parasitic capacitor in parallel with the mechanical resonance, which can be modeled, like any resonance, by a series RLC circuit. This parasitic capacitor explains why the filtering capacity of the resonator degrades at high frequencies (Fig. 15), but we have found that the rejection of the spectral components that disrupt the MAX2771 is sufficient to allow for its proper functioning.

Once the MAX2771 accepts to be clocked by the AD9851 and communicates its measurements, we assemble the entire system with data acquisition and transmission via USB to be received by `app/pocket_dump/pocket_dump` from PocketSDR to the FIFO, GNU Radio which performs frequency transposition to remove the intermediate frequency of 2 MHz and decimates to produce 2 MS/s on its ZeroMQ Publish port, and `gnss-sdr` which produces the position and timing solution transmitted over UDP port and read by the Python script implementing `protobuf`. The time and frequency offset information between the local oscillator—thus this time the output of the AD9851—and the GPS clocks are read: initially, we reprogram the AD9851 to verify its impact on the information from `gnss-sdr`. Already, it is necessary to avoid modifying the frequency too abruptly, or else the control loops of `gnss-sdr` will lose lock and the time to regain the signals from the satellites can be long. In the example of Fig. 16, we began with a nominal frequency of 24 MHz produced by the AD9851 – which is of course not exactly 24 MHz since the AD9851 is clocked by a quartz oscillator that is itself inaccurate – and then we increment by 3 Hz every 200 seconds or so. We observe (Fig. 16, top) that `gnss-sdr` indeed detects the frequency variations of the MAX2771 clocking, visible as the slope of the time offset to the 1-PPS GPS

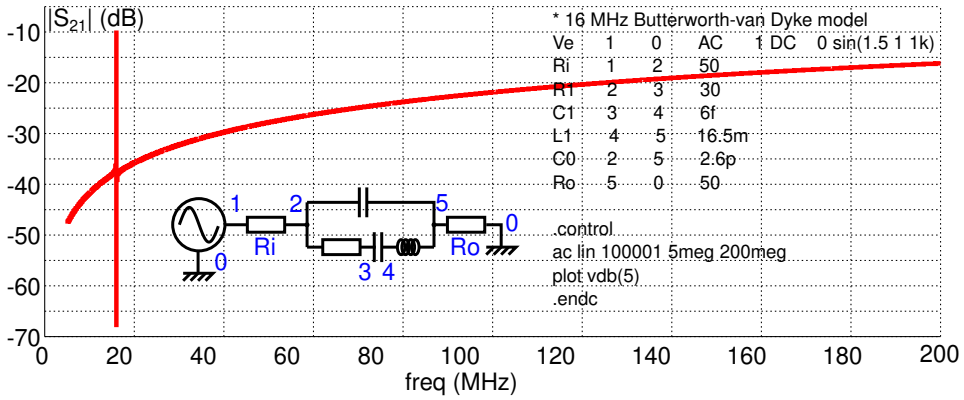


Figure 15: SPICE simulation (inset, top right for `ngspice`) of the transmission transfer function of a quartz resonator modeled by its equivalent circuit known as Butterworth-VanDyke, placing a circuit $R_1L_1C_1$ in parallel to simulate the mechanical resonance (motion branch) with the static capacitance C_0 formed by the two electrodes covering the quartz slab (electric branch). We observe that the resonator attenuates the signal little at its resonance (here 16 MHz) but filters poorly at high frequency when the impedance of C_0 decreases.

(1 pulse per second representing GPS time), and that the addition of 6 Hz almost cancels the time drift. In the second half of the experiment, after the 800 s mark, we abruptly subtracted 9 Hz to return to the nominal 24 MHz, illustrating the loss of lock of the control loops and the time required for `gnss-sdr` to regain information from the GPS constellation. Below, the frequency offset from the nominal value follows the same trend, since frequency is the derivative of phase and time resembles a phase, thus each slope on the time drift (top) corresponds to steps at different levels of frequency offset. Since GPS is clocked by atomic clocks, we can trust that cancelling this frequency offset allows us to obtain a signal at an exact frequency this time, as we observe by measuring the output frequency of the DDS on a counter referenced to a hydrogen maser considered to be accurate (Fig. 16, inset).

The final control code consists of initializing various Python modules for communication with the peripherals and control constants

```
#!/usr/bin/env python3

import socket
from monitor_pvt_pb2 import MonitorPvt
import usb
from sys import platform as _platform
import binascii
import struct
import sys
import time

VID = 0x04B4
PID = 0x1004
fclk=150e6
f=24e6
ftw0=((f/fclk)*2**32)   # 687194767
y=0
c=5e-3
Kp=.01
Ki=Kp/3
```

followed by opening the communication interfaces via USB and UDP

```
dev = usb.core.find(idVendor = VID, idProduct = PID)
msg=struct.pack('>I',int(ftw0))
msg+=bytes([1])
```

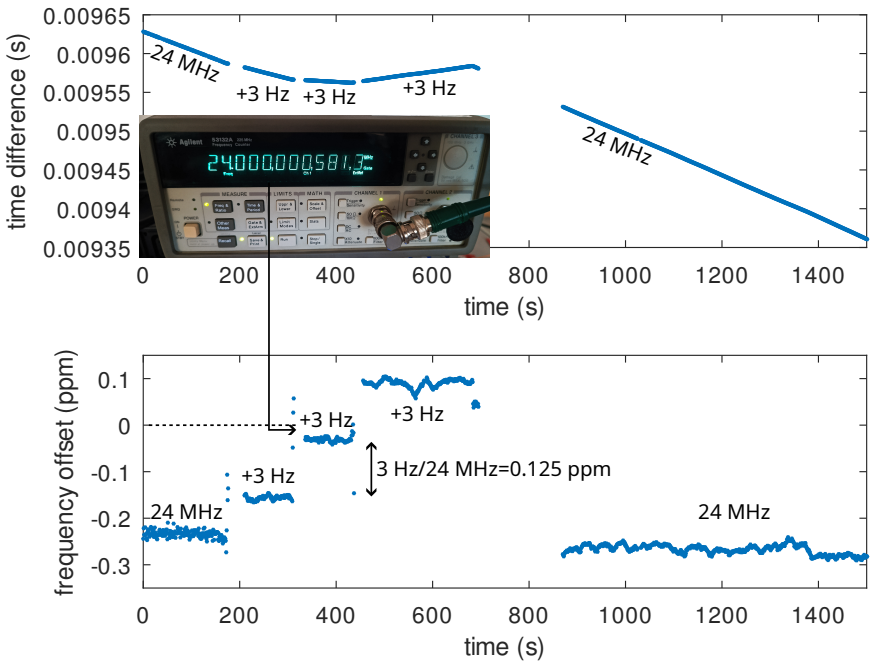


Figure 16: Top: time offset (or phase) between the oscillator (DDS) locking the MAX2771 and GPS time. Bottom: frequency deviation between the signal from the DDS and the nominal frequency determined by GPS. When the DDS is programmed to24 MHz+6 Hz, the frequency error is almost compensated (dashed line), as confirmed by a frequency counter referenced to a presumed accurate hydrogen maser. The step of 3 Hz added to the output of the DDS corresponds to a relative variation of $3/24 = 0.125$ ppm, in agreement with the information provided by `gnss-sdr` (bottom).

```
dev.ctrl_transfer(0x40 , 0x50, 0x00, 0, msg) # write msg on SPI bus

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind(("127.0.0.1", 1234))
foo = MonitorPvt()
```

The control loop itself is a classic proportional-integral controller, which we will simply take care to reset after a prolonged absence of messages from `gnss-sdr` that would have allowed the free oscillator to drift too much (variable `m`).

```
m=0
yold=0
oldtime=time.time()
while True:
    data, addr=sock.recvfrom(1500)
    foo.ParseFromString(data)
    df=foo.user_clk_drift_ppm*24
    dt=foo.user_clk_offset
    y+=c*(df-y)  # low pass IIR with time constant 1/c
    print(f"{m}: TOW={foo.tow_at_current_symbol_ms} dt={dt} \
␣␣␣␣␣␣x={foo.pos_x} y={foo.pos_y} z={foo.pos_z} df={df} f={f}")
    newtime=time.time()
    if (newtime-oldtime)>200:
        m=0
    oldtime=newtime
    if (m>200):
        dc=Kp*(y-yold)+Ki*y
```

```
        if abs(dc)>1:
            dc=dc/abs(dc)
        yold=y
        f-=dc
        ftw0=((f/fclk)*2**32)
        msg=struct.pack('>I',int(ftw0))
        msg+=bytes([1])
        dev.ctrl_transfer(0x40 , 0x50, 0x00, 0, msg) # write msg on SPI bus
    m=m+1
```

The results over two days are presented in Fig. 17, where we first observe the amnesias of `gnss-sdr`, which sometimes struggles to find a solution (three significant absences around 7 h, 15 h, and from 30 to 40 h): these are the gaps in the two curves at the top during which `gnss-sdr` has no information to provide. During this time, the oscillator driving the MAX2771 is free to drift as it wishes, as observed on the third curve from the top, acquired by an external frequency counter HP53131A. When the solution is found, the control successfully brings the oscillator back to its nominal frequency value of 24 MHz. Finally, at the bottom, the control frequency of the AD9851 aims to absorb environmental fluctuations by keeping its output at the nominal value of 24 MHz relative to the GPS signal: again, in the absence of measurements by `gnss-sdr`, no control is produced, and the curve shows gaps during these amnesic phases.

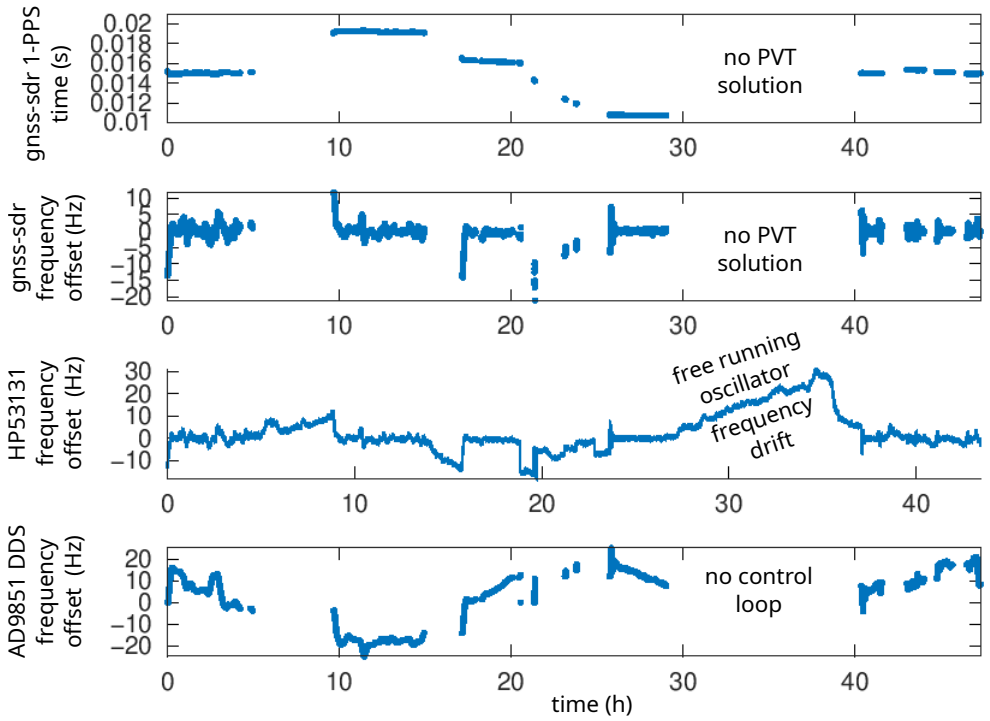


Figure 17: Control of the frequency of the AD9851 based on measurements provided by `gnss-sdr`, resulting from the comparison between the local oscillator and the atomic clocks embedded in GPS satellites. At the top, the time offset provided by `gnss-sdr`, and in the middle, the estimated frequency offset by `gnss-sdr`, with the gaps in the curve indicating the absence of a PVT solution.

The problem of oscillator drift in the absence of a reference signal to lock onto is a classical issue known as "holdover", which any mobile telephony relay (Fig. 18), for example, must be able to manage, justifying the need for "good" oscillators alongside the GPS receiver fitted on these base stations. Furthermore, we note that our correction measures and acts only on frequency, thus leaving a static error on time that will still need to be corrected so that the visible steps on the top curve of Fig. 17 align

with the same setpoint. This is the point we will now address.

3.3 Time (phase) control

We have succeeded in correcting the frequency of the quartz oscillator when `gnss-sdr` identified the signals from the GPS constellation, and in catching up with the drift in the meantime. This already meets a number of needs for radio frequency receivers that are only interested in the frequency of the local oscillator and not in time. However, many applications are interested not only in syntonizing oscillators (same frequency) but also in synchronizing them (same phase). Phase φ and time τ are closely related through the carrier frequency ν of the radio frequency signal since $\varphi = 2\pi\nu\tau$, but above all, frequency is thus the derivative of phase $\nu = \frac{1}{2\pi} \cdot \frac{d\varphi}{dt}$. Control is therefore considerably more complicated since this time the system no longer controls a frequency (DDS) to correct a frequency – thus having a unit gain between the two quantities – but the phase (time): thus, the integral of frequency accumulates errors. Ideally, we would have two controls, one for delay to correct the phase and one for frequency, but that is not the case, so we must rely on the unique degree of freedom that is the frequency controlling the MAX2771 output from the AD9851 to first bring the phase (time) to its setpoint, and then prevent it from drifting by keeping the oscillator at its nominal frequency relative to the frequency provided by GPS, all with a quartz oscillator that drifts with environmental conditions.

The limitation of controlling only the frequency of the DDS faces an additional constraint: one must not stray too far from the nominal value of 24 MHz, lest it no longer allow `gnss-sdr` to compensate for the Doppler shift. Although nominally ±5 kHz, the configuration of `gnss-sdr` allows extending the search range for the frequency offset to compensate for the inaccuracy of the local oscillator, but the more we extend this range, the more computing resources are required. Finally, one must not disturb the oscillator driving the MAX2771 too much, lest it causes the frequency and delay control loops of `gnss-sdr` to drop out, which will then have to find its solution (acquisition) and propagate a solution again (tracking). From these two conditions, we deduce an open-loop initialization phase aimed at roughly reaching the setpoint, as follows:

1. By configuring the oscillator to a nominal 24 MHz, without knowing its accuracy, we let `gnss-sdr` find the GPS signal, hoping that the local oscillator is not "too far off" so that the Doppler-deviated signal related to satellite movements and the oscillator drift remain within the range we have defined in `Acquisition_1C.doppler_max` in the configuration file of `gnss-sdr`,

2. Once the constellation is acquired, `gnss-sdr` provides, on the one hand, an estimate of the frequency offset between the local oscillator and the expected exact frequency, and on the other hand, the time offset between the local clock and GPS time. Depending on the sign of this time offset relative to the setpoint of the middle of a symbol of duration 20 ms, thus set to 10 ms, we shift the frequency of the local oscillator by +300 Hz if the delay is less than 10 ms, or by −300 Hz relative to the nominal 24 MHz if the delay is greater. In doing so, we intentionally allow the local clock time to drift relative to GPS time at a rate of $300/24 \cdot 10^6 = 12.5$ μs/s. Thus, it will take at most $10 \cdot 10^{-3}/12.5 \cdot 10^{-6} = 800$ s to reach the setpoint in the worst-case scenario where the initial point was 10 ms from the setpoint,

3. However, by shifting the oscillator from 24 MHz by 300 Hz, this equates to shifting the GPS L1 carrier at 1575.42 MHz by $300/24 \times 1575.42 \simeq 19$ kHz, thus well beyond the ±5 kHz introduced by the movement of the satellites. Therefore, one must think carefully about adjusting `Acquisition_1C.doppler_max` to more than 25 kHz to reacquire the satellites during the drift of the local clock,

4. Once the setpoint is reached, the local oscillator is reprogrammed to the nominal 24 MHz that we know can be achieved upon receiving information from `gnss-sdr` about the offset of the local oscillator from the nominal frequency, including the ±300 Hz that we have introduced. It should be noted that by doing this, we annoy `gnss-sdr` for the second time, as it "hates" these sudden changes in the frequency of the analog-to-digital converter clock, which causes its frequency and phase lock loops to unlock. The best we can hope for is that the constellation is sufficiently favorable for `gnss-sdr` to reacquire all satellites before time has drifted too much, especially since the oscillator is at its exact value, and the drift has been minimized in open loop.

5. Once the constellation is reacquired, knowing that the delay is close to its setpoint, we start the PI control with very high gains: indeed, compared to the control on frequency, since time t and frequency f are related by $f = 1/t$, thus $df/f = -dt/t$, and $f = 24 \cdot 10^6$, while the time interval for measuring delay is $t = 1$ s, thus $df = dt \cdot 24 \cdot 10^6$ which would cause the control loops to unlock if we attempted to apply such gains during the initialization phase, which must in any case be saturated to avoid losing the satellite signals. We have found that limiting the frequency jump to $df = 0.2$ Hz each second allows `gnss-sdr` to keep the control loops locked without losing phase and signal frequency, and still allows incrementing the local oscillator frequency by 2 Hz in 10 seconds, thus compensating for environmental drifts.

All of this is theory...

Figure 18: Two examples of antennas for receiving satellite navigation signals for the time synchronization of mobile telecommunications base stations, on the left at the scientific campus of the University of Franche-Comté in Besançon, on the right in the suburbs of Clermont-Ferrand (antenna at the top of the tower outlined in red).

In practice, we implemented the control law c_n through its recurrence equation, which starting from the error at date n to the setpoint ε_n produces a proportional term $K_p\varepsilon_n$ and an integral term $K_i \sum_{p=0}^{n} \varepsilon_p$, which indicates that $c_{n+1} - c_n = K_p(\varepsilon_{n+1} - \varepsilon_n) + K_i\varepsilon_{n+1}$. While we thought we could do without the integral term since the system is itself an integrator, we observe that if the error does not evolve, then the setpoint does not evolve either when only retaining the proportional term, but without guaranteeing that the error is cancelled, merely maintained constant. Thus, even with the integrator behavior of the

system, we retain the integral term to eliminate static error.

One of the major concerns we face, which becomes even more dramatic as the phase accumulates error during GPS signal loss phases of "holdover", is that `gnss-sdr` decides to invalidate the PVT solutions even when more than 4 satellites are visible. This is very annoying, especially since we are not interested at all in the position but only in time, even if all these quantities are intimately intertwined. The PVT solution produces two pieces of information: the position dilution of precision $PDOP = \sqrt{\sigma_x^2 + \sigma_y^2 + \sigma_z^2}$, the sum of the variances of the positions, and the geometric dilution of precision $GDOP = \sqrt{\sigma_x^2 + \sigma_y^2 + \sigma_z^2 + \sigma_t^2}$, which is the sum of the variances in position as well as in time. We have used the convention from the literature to denote σ_t, but it should be noted that due to the homogeneity of the quantities, it is actually $c \cdot t$ that is expressed, with c being the speed of light, to convert the delay into distance. Thus, we obviously observe that the fluctuation in time $\sigma_t^2 = GDOP^2 - PDOP^2$, which is called $TDOP^2$, but which is of no interest to anyone since everyone uses GPS for positioning: this estimator is therefore not produced by `gnss-sdr`, but must be calculated. If we aim for a control on the order of hundreds of nanoseconds (10^{-7} relative stability), then the $TDOP$ of $3 \cdot 10^8 \times 10^{-7} = 30$ remains acceptable even if it is the limit that `gnss-sdr` imposes on $GDOP$ to invalidate the PVT solutions, noting that in practice it is often the position that hampers the result. This problem is known since `https://rtklibexplorer.wordpress.com/2019/05/18/a-few-simple-debugging-tips-for-rtklib/` indicates "I have found that the chi square threshold can be too strict in some cases, so in the demo5 code I have changed the error to a warning."

4 Conclusion

We have concluded this exploration of efficient USB bus programming under GNU/Linux with the development of a dedicated board for receiving satellite navigation signals in L band equipped with several synchronous receivers allowing, for example, a measurement of the direction of arrival of the signal. The proper functioning of the system has been demonstrated by decoding the position of the receiver.

More interestingly, the error between the local clock and the remote clock has been measured and then corrected, through a long-term real-time measurement respecting the Unix precepts, namely an efficient tool dedicated to each operation. To do this, `PocketSDR` acquired the samples, GNU Radio read from a FIFO to transpose the frequency and emit the result on a ZeroMQ socket, so that `gnss-sdr` could decode its content and disseminate the result of its calculations on a UDP socket. It should be noted that this approach, implemented here on a single computer, is perfectly suitable for being distributed across multiple computing platforms provided there is sufficient communication bandwidth. Unbeknownst to many, time synchronization via GNSS signals is omnipresent (Fig. 18), and the stakes are such that providing a secured signal, or at least stopping that signal if a jamming source is identified, has been addressed with the direction of arrival analysis of the signal on two spatially separated antennas.

All our discussions have focused on GPS L1 C/A because it is the only signal for which we have the necessary computing power to process in real-time with `gnss-sdr`, a necessary condition for implementing the controls. We have validated the proper functioning of L5, E1, or E5 decoding in post-processing, but the computation time is much longer than the acquisition time, partly due to the higher data rate and partly due to algorithmic complexities that are much more resource-intensive.

The assertion of a generic receiver in the L band is somewhat misleading since the voltage-controlled oscillators (VCOs) that equip each MAX2771 cannot cover the entirety of the 1–2 GHz band, but with the addition of a first local oscillator and mixer to bring the signal of interest into the frequencies accessible to the VCOs, the 44 MS/s measurement bandwidth remains very attractive. In this sense, let us mention Inmarsat, which communicates with ground terminals between 1525 and 1646.5 MHz, while the Thuraya satellite phones (OsmocomGMR project) exploit frequencies between 1525 and 1661 MHz, all of which are at least partially accessible to the MAX2771. However, communications from meteorological satellites via HRPT at 1700 MHz with a bandwidth of 3 MHz are beyond the capabilities of the MAX2771.

Ultimately, these circuits provide an opportunity to discover the specifics of the fascinating world of low-resolution but fast analog-to-digital converters, which have somewhat different statistical properties compared to high-resolution converters. Indeed, [11] repeatedly states, *when the quantization grain size q is fine enough, (i.e., equal to one standard deviation or less), ...*" as a hypothesis for his analyses

of the statistical properties of quantization of sampled signals, e.g., *... the PDF, CF, and moments of the quantization noise are almost indistinguishable from those of uniformly distributed noise. To see differences from the uniform case, the quantization must be extremely rough.*" and therefore probably invalid for converters coded on only 2 or 3 bits, since the coarse quantization mentioned is exactly the case presented here.

The source codes, configuration files, and printed circuit board layouts are available at `https://github.com/jmfriedt/max2771_fx2lp`.

Acknowledgments

Claudio Calosso from the Italian metrology laboratory INRIM (Turin) presented the design of the low-pass filter in the form of an infinite impulse response (IIR) filter and proposed various analyses on the data collected during this study. All the works in the bibliography were obtained from Library Genesis.

These developments were carried out as part of the ELISE Master program at the University of Franche-Comté in Besançon for teaching embedded electronics using free opensource software (`http://electronique.univ-fcomte.fr/elise.php`).

References

[1] J.-M Friedt, *Programmation USB sous GNU/Linux : application du FX2LP pour un récepteur de radio logicielle dédié aux signaux de navigation par satellite (1/2)*, Hackable **57** 88–130 (Nov.-Dec. 2024)

[2] K. Borre, M. D.M. Akos, N. Bertelsen, P. Rinder, S.H. Jensen, *A software-defined GPS and Galileo receiver: a single-frequency approach*,.Springer (2007)

[3] K. Borre, I. Fernández-Hernández, J.A. López-Salcedo, M.Z.H. Bhuiyan, *GNSS software receivers*, Cambridge University Press (2023)

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

[5] J.- M Friedt, *La peinture sur spectre radiofréquence, et l'effet capture de la modulation en fréquence – ou pourquoi les avions communiquent encore en AM*, GNU/Linux Magazine France **216** (Juin 2018)

[6] Tom Rondeau, *GNU Radio for Exploring Signals*, FOSDEM (2016) à `https://archive.fosdem.org/2016/schedule/event/gnu_radio/`

[7] D.S. De Lorenzo *& al. GPS receiver architecture effects on controlled reception pattern antennas for JPALS*, Proc. 17th International Technical Meeting of the Satellite Division of The Institute of Navigation (ION GNSS 2004) 2010–2020) discute de l'annulation de leurrage et brouillage avec des récepteurs munis de convertisseurs analogique-numériques à peu de bits, rendant difficile l'extraction du vrai signal quand le signal interférant est puissant.

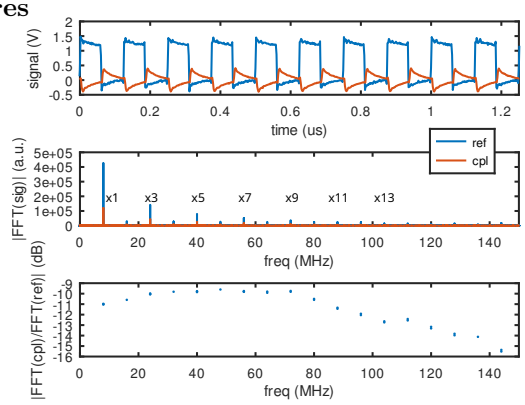
[8] D. Rabus *& al.*, *Generating A timing information (1-PPS) from A software defined radio decoding of GPS signals*, Proc. joint EFTF/IFCS (2021)

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

[10] il se trouve que la semaine GPS a commencé dans la nuit du 5 au 6 janvier 1980, entre un samedi soir et un dimanche matin, d'après `https://www.cnmoc.usff.navy.mil/Our-Commands/United-States-Naval-Observatory/Precise-Time-Department/Global-Positioning-System/GPS-Information-SA-DGPS-Leap-Seconds-etc/GPS-Week-Number-Rollover/`. Le problème de `gnss-sdr` est plus ennuyeux que les récepteurs qui ne savent pas gérer le fait que la semaine GPS n'est codée que sur 10 bits et se réinitialisent tous les 20 ans environ.

[11] B. Widrow, I. Kollár, *Quantization Noise – Roundoff Error in Digital Computation*, Cambridge University Press (2008)

Correction: coupling between adjacent wires



We concluded the previous article on prototyping with the FX2LP to control the MAX2771 evaluation board through an estimate of coupling, using a network analyzer, between adjacent wires carrying fast signals (a few MHz), arguing for the necessity of routing a printed circuit board to avoid these couplings that induce data corruption. As a reminder, a network analyzer emits a signal at a frequency and observes, in the form of the ratio of the amplitude of the received signal to the amplitude of the emitted signal, the reflected and transmitted voltage. The indications in the form of scattering coefficients (noted S_{ij}) are provided in decibels. Since the decibel is a power ratio, while the measurement of S parameters is a voltage ratio, the conversion from one to the other is obtained by $S_{dB} = 20\log_{10}(S_{ij})$. In our previous analysis, we mistakenly converted the dB measurements into amplitude ratios, as pointed out by a reader, Yves Barbin, who is thanked for his corrections.

We therefore revisit the illustration of coupling between adjacent wires to correct the numerical values provided previously and to complement the network analyzer measurements with a time measurement using an oscilloscope. In this demonstration, a square wave generator at 8 MHz drives a 21 cm wire connected to ground through a 50 Ω resistor, while a second parallel wire of the same length connects the input of a radiofrequency oscilloscope (impedance 50 Ω) to ground via a 50 Ω resistor, with the two wires being parallel and separated by a few millimeters.

At the top of this graph, the emitted signal in blue and the induced (coupled) signal in red, whose significant amplitude is the source of erroneous bits transmitted. In the middle, the Fourier transform of the emitted signal (blue) and induced signal (red), predominantly containing the **odd harmonics** whose **amplitude** (and not power as we previously indicated) decreases as the harmonic number increases (Fourier series of the square wave). At the bottom, the ratio of the amplitude of the **induced signal to the emitted signal** (Vo/Vi), converted to decibels by $20\log_{10}(\cdot)$ to account for the square relating amplitude and power.

In the table from the previous article, it should therefore be read, converting dB to a linear scale of voltage ratios by $10^{dB/20}$:

08 MHz: -24.9 dB=0.06
12 MHz: -21.7 dB=0.08
24 MHz: -16.4 dB=0.15
48 MHz: -12.1 dB=0.25

in much better agreement with the observations in the time domain above than those deduced earlier from the spectral domain measurements.