atomic_clock: user's manual

A 10 MHz clock simulator on the Red Pitaya STEMlab 125-14

Rok Žitko
Jozef Stefan Institute, Ljubljana, Slovenia
rok.zitko@ijs.si

Red Pitaya `atomic_clock` design, version 3.2
October 28, 2026

Abstract

This manual takes you from an empty SD card to a working clock simulator: setting up the board and a Linux computer, building and installing the design, the first signals, the controls, and recipes for common tasks: simulating an oscillator from its data sheet, phase steps and frequency offsets, replaying a recorded clock, PPS work and stability measurements. Application notes show five uses with other equipment: checking a counter, an NTP server on the simulated PPS, a replayed GNSS satellite clock, holdover tests and teaching the noise types. A live dashboard estimates the noise while the data arrive. The command reference is `docs/PROTOCOL.md`; how the signal is made is in the theory manual, how it is measured in the testing manual.

Contents

1 What atomic_clock is **2**

2 Safety **3**

3 Setting up from an empty SD card **3**
3.1 The PYNQ image . . . 3
3.2 Network and first login . . . 4
3.3 SSH keys and the sudo rule . . . 4
3.4 Scheduled jobs and the system clock . . . 4
3.5 The computer: repository and Python . . . 4
3.6 Building the design . . . 5
3.7 Installing on the board . . . 5
3.8 Calibration . . . 5
3.9 Checks . . . 5

4 Getting started **5**
4.1 A first measurement in loopback . . . 6
4.2 Presets, reset and the safe state . . . 7

5 Controlling the simulator **7**
5.1 Settings . . . 7
5.2 TCP commands . . . 8
5.3 Python methods . . . 8
5.4 Noise levels from Allan deviations . . . 8

6 Recipes **9**
6.1 Simulate an oscillator from its data sheet . . . 9
6.2 A phase step, smooth or abrupt . . . 9
6.3 A frequency offset . . . 10
6.4 Replay a recorded clock . . . 10
6.5 PPS and time of day . . . 10
6.6 Measure stability over hours or days . . . 11

7 Oscillator catalogue **11**
7.1 Bad clocks for demonstrations . . . 12

8 Applications **14**
8.1 Checking a frequency counter . . . 15
8.2 A PPS-disciplined NTP server . . . 16
8.3 Replaying a GNSS satellite clock . . . 16
8.4 Holdover test . . . 18
8.5 Teaching the noise types . . . 19

9 Watching the noise live **20**
9.1 Starting it . . . 21
9.2 Reading the results . . . 21
9.3 The meter stream . . . 22

10 Where next **23**

1 What atomic_clock is

`atomic_clock` turns a Red Pitaya STEMlab 125-14 into a **10 MHz source that behaves like a real clock**. Equipment that expects an atomic-clock reference (a counter, a phase comparator, a disciplined oscillator, a time-transfer receiver) can be fed a signal with a known, programmed imperfection, and its response compared with what was programmed.

connector	signal
OUT1	the **simulated clock**: $A_1 \sin 2\pi(10\,\mathrm{MHz} \cdot t + \varphi(t)) + \mathrm{offset}_1$
OUT2	the **reference**: the same 10 MHz without any programmed phase
DIO1_P (E1)	1 PPS of the simulated clock (3.3 V logic)
DIO2_P (E1)	1 PPS of the reference, optionally on UTC seconds
IN1, IN2	the built-in **phase meter** (10 MHz I/Q); in loopback it measures OUT1 against OUT2

The phase $\varphi(t)$ of OUT1, in cycles of the carrier (1 cycle = 100 ns), is the sum of

- a **set phase**, reached smoothly at a limited rate (`phase`, `phase_slew`);
- a **frequency offset**, integrated exactly (`frequency_offset`);
- **noise** of the IEEE 1139 power-law model [1]: white PM (h_2), flicker PM (h_1), white FM (h_0), flicker FM (h_{-1}), random-walk FM (h_{-2}), plus a linear frequency drift and a sinusoidal phase modulation;
- or a **recorded time error** $x(t)$ of a real clock, played back.

The carrier is exactly 2/25 of the board's 125 MHz clock, so the simulator adds no drift of its own; its absolute accuracy is that of the board's crystal (not measured). Compare OUT1 against OUT2 to see the programmed phase alone. Amplitudes and offsets are **real volts** at the SMA connectors, through the board calibration.

The other designs. The same repository holds `mult_add` (a two-input analog computer, also used to measure the board calibration), `analog_echo` (an ADC-to-DAC echo with gain and offset), `synth` (16 tones per output), `awg` (a function and arbitrary waveform generator) and the stand-alone `LED_blink` example. They share the board setup, the control service and the Python framework (`common/`); the service runs one design at a time and switches on request (`LOAD atomic_clock`).

2 Safety

Read the section *Connecting to lab equipment: read first* of the repository's `README.md` before connecting anything that must not see an unplanned voltage. For `atomic_clock`:

- **Power-up and reload.** For about two minutes after power-on the FPGA is not configured and the outputs are undefined. Whenever the design is loaded (power-up, `LOAD`, a reload by a driver or by `make test-remote`), *both outputs start a 10 MHz sine* of about 0.57 V amplitude (0.5 V uncalibrated). After a reboot the service then restores the last settings, *including noise and playback*.
- **Limits.** Outputs reach about ±1.13 V at DC; there is no clamp. The set amplitude is the 10 MHz level into 50 Ω: 0.5 V gave 1.00 Vpp into a 50 Ω feed-through but 2.12 Vpp into a 1 MΩ scope input (measured 2026-10-13), so a high-impedance load sees about twice the setting (up to about ±2.4 V). The offset (DC) is right into any load. Inputs: at most about ±1 V (LV jumpers).
- **PPS pins** are 3.3 V LVCMOS outputs, not 5 V tolerant. Connect them only to high-impedance logic or counter inputs; never drive them.
- **The phase moves by itself** when noise or playback is on. A phase step (`phase_slew` = 0) can make equipment locked to the 10 MHz lose lock.
- **Safe state:** `SAFE` turns both sines, the noise and the playback off; both outputs sit at the calibrated 0 V.

3 Setting up from an empty SD card

You need: a Red Pitaya STEMlab 125-14 with its power supply, a micro-SD card of at least 16 GB, an Ethernet connection to your local network, SMA cables (two equal ones for loopback), and a Linux computer. Building the FPGA design needs Vivado 2025.1 on a Linux machine (§3.6); everything else runs with Python 3.

3.1 The PYNQ image

The board runs the PYNQ 3.0.1 image for the STEMlab 125-14 (Ubuntu with Python, Jupyter and the PYNQ FPGA library) built by the `Pynq-Redpitaya-125` project, https://github.com/dspsandbox/Pynq-Redpitaya-125. Its README links the ready-made image `Pynq-Redpitaya-125-14-3.0.1.img` (a download from Google Drive) and describes how it was built. Use exactly this image: the design, the sudo rule and the service paths below assume PYNQ 3.0.1 with its virtual environment in `/usr/local/share/pynq-venv`.

Write the image to the card on the Linux computer. Find the card's device name with `lsblk` (insert the card and see which device appears, e.g. `/dev/sdX` or `/dev/mmcblk0`); writing to the wrong device destroys its contents.

```
lsblk                                        # identify the SD card, e.g. /dev/sdX
sudo dd if=Pynq-Redpitaya-125-14-3.0.1.img of=/dev/sdX bs=4M status=progress conv=fsync
sync
```

A graphical alternative is balenaEtcher. Insert the card into the board, connect Ethernet, and power it up.

3.2 Network and first login

The board obtains an address from your network's DHCP server. Find it in the router's list of clients, or log in on the serial console (micro-USB console port of the board, 115 200 baud: `screen /dev/ttyUSB0 115200`) and run `ip addr`. Give the board a fixed address (a DHCP reservation in the router is simplest); this manual uses `192.168.0.149`, the address of the lab's board. If yours differs, put `BOARD_HOST = <address>` into `atomic_clock/config.mk` (copy `config.mk.example`) and use your address wherever this manual writes `192.168.0.149`. The PYNQ user is `xilinx`, password `xilinx`; change it with `passwd` if the network is not private. The control service has no authentication: keep the board on a trusted network.

3.3 SSH keys and the sudo rule

The workflow logs in by SSH without passwords. On the computer (create a key with `ssh-keygen` first if you have none):

```
ssh-copy-id xilinx@192.168.0.149
ssh -o BatchMode=yes xilinx@192.168.0.149 true      # must succeed without a prompt
```

Loading an FPGA design needs root, so the PYNQ Python must run under sudo without a password. On the board (`ssh -t xilinx@192.168.0.149`):

```
sudo visudo -f /etc/sudoers.d/rpfpga
```

add the line

```
xilinx ALL=(root) NOPASSWD: /usr/local/share/pynq-venv/bin/python
```

save, and run `sudo chmod 0440 /etc/sudoers.d/rpfpga && sudo visudo -c`. If `sudo -l` lists an `(ALL : ALL) ALL` rule *after* this one, move the line to the very end of `/etc/sudoers` (sudo uses the last matching rule). Check from the computer; it must print `0 3.0.1`:

```
ssh -o BatchMode=yes xilinx@192.168.0.149 \
  'sudo -n /usr/local/share/pynq-venv/bin/python -c "import os, pynq; print(os.geteuid(), pynq.
    ↪ __version__)"'
```

3.4 Scheduled jobs and the system clock

Ubuntu on the board runs maintenance jobs (`apt-daily`, `apt-daily-upgrade`, `man-db`) that load the CPU and the SD card for a minute or more. That can stall the control service for seconds, which interrupts the noise playback and long recordings. Disable them (and update by hand when convenient, `sudo apt update && sudo apt upgrade`):

```
sudo systemctl disable --now apt-daily.timer apt-daily-upgrade.timer man-db.timer
```

For PPS on UTC seconds the board's clock must be synchronized by NTP: `timedatectl` should report `System clock synchronized:  yes` (the board needs access to an NTP server, by default on the internet).

3.5 The computer: repository and Python

On the Linux computer you need Bash, GNU Make, OpenSSH, `flock` (util-linux), git and Python 3. Clone the repository:

```
git clone https://github.com/rokzitko/redpitaya.git
cd redpitaya
```

The Python client needs only the standard library. The stability tools (`atomic_clock/tools/`) need numpy, scipy, matplotlib and allantools [2], for example in a conda environment:

```
conda create -n base-tools python numpy scipy matplotlib
conda activate base-tools && pip install allantools
```

3.6 Building the design

The FPGA files (`.bit`, `.hwh`, `.tcl`, `.json` in `atomic_clock/overlay/atomic_clock/`) are built from the sources with Vivado; they are not stored in the repository. On the build machine (it can be the same computer):

- install **Vivado 2025.1** with Zynq-7000 support (no GUI is needed);
- install the Red Pitaya board definition `redpitaya.com:redpitaya-125-14:part0:1.0` (the folder `Vivado/board_files/redpitaya-125-14` of the same `Pynq-Redpitaya-125` repository) in Vivado's `data/boards/board_files/`, or point `BOARD_REPO_PATH` at it;
- load Vivado into the shell: `source <Vivado>/2025.1/Vivado/settings64.sh`.

Then build:

```
cd atomic_clock
make                  # creates the project, synthesizes, implements, exports overlay/
```

`make help` lists all targets. The build is checked by its timing summary and DRC report (see `common/README.md`).

3.7 Installing on the board

```
make upload install-service     # copy the design and the Python packages; install the service
make test-remote                # program the board, check identity and power-up state
```

`make upload` copies the overlay, the driver package and the notebook to the board and installs the framework into the PYNQ virtual environment. `install-service` installs and starts `rp-overlay.service`, the TCP control service on port 5025, with `atomic_clock` active; from now on the board loads the design about two minutes after every power-up and restores the last settings. `make test-remote` loads the design, verifies its identity and build, and applies the `reference` preset.

3.8 Calibration

Volts are converted with the board calibration, measured once per board with `mult_add`, the loopback cables and a DMM, as described in `common/CALIBRATION.md`. Until then nominal converter constants are used, which on the lab's board were about 14 % off. The calibration is stored on the board and shared by all designs. Match the software's LV/HV input-range settings to the physical jumpers: they select the calibration used to convert the phase meter's amplitudes to volts.

3.9 Checks

```
echo '*IDN?' | nc -q 1 192.168.0.149 5025     # identity of the running design
make test                                     # offline tests on the computer
make test-hw                                  # loopback test on the board (OUT1->IN1, OUT2->IN2)
```

`make test-hw` needs the two loopback cables; it measures phase steps, the slew, the frequency offset, white and flicker FM and the PPS counters with the board's own meter. Results of the checks done on the lab's board are in `docs/VERIFICATION.md`.

4 Getting started

Connect a scope to OUT1 and OUT2 through 50 Ω feed-throughs (the set amplitude; at 1 MΩ you see about twice it) and trigger on OUT2. There are four ways to control the board; all act on the same settings.

TCP, from anything. One text command per line, one reply line (`OK ...` or `ERR ...`); keywords are case-insensitive:

```
nc 192.168.0.149 5025
PRESET quadrature           # OUT1 leads OUT2 by a quarter cycle (25 ns)
DESCRIBE?
PHASE 0.1                   # OUT1 moves to 10 ns ahead, smoothly (in 15 ms)
FREQ 1e-9                   # then gains 1 ns per second
STATE?
RESET                       # back to the default: in phase, no offset, no noise
```

Python on the computer through the service (standard library only), from the repository root:

atomic_clock/README.md

```
import sys; sys.path[:0] = ["common/python", "atomic_clock/control"]
from atomic_clock import AtomicClockClient
with AtomicClockClient("192.168.0.149") as clock:   # the board must run atomic_clock
    clock.set(1, amplitude=0.3, phase=-1e-6)        # 1e-6 cycle = 100 fs behind OUT2
    clock.set_noise(h_1=1e-26, h0=2e-24, seed=1)    # flicker FM + white FM
    clock.set_playback(source="model")              # played by the board's service
    print(clock.playback_status())
```

If the board runs another design, switch first:

```
rpfpga.client.connect("192.168.0.149").load("atomic_clock")
```

Jupyter on the board: open `http://192.168.0.149:9090` (password `xilinx`), then `atomic_clock/atomic_clock.ipynb`. Its control panel has tabs for the outputs, the noise (with a preview of the resulting Allan deviation and $\mathcal{L}(f)$), the meter and PPS, presets, and calibration.

Command line, from the repository root:

```
PYTHONPATH=common/python:atomic_clock/control python3 -m rpfpga.cli --host 192.168.0.149 --phase
    ↪ 0.25
```

4.1 A first measurement in loopback

Cable OUT1 → IN1 and OUT2 → IN2 with two equal cables, both connected the same way (a 50 Ω terminator on one input only shifts its phase by about 0.1 cycle). In direct loopback the meter reads about 0.67 V for the 0.5 V setting: the input loads the output, which at 10 MHz behaves like a 50 Ω source. Then:

```
PRESET reference
MEAS? 100                   # average of 100 windows of 10 ms
PHASE 0.001                 # 100 ps
MEAS? 100
```

`MEAS?` returns, as JSON, the amplitude (V) and phase (cycles) of each input and the `difference` IN1 − IN2. The difference at `PRESET reference` is the fixed difference of the two cable and channel paths; after `PHASE 0.001` it changes by 10^{-3} cycle. White PM above the meter's window rate (100 Hz by default) averages out and is not seen.

The service averages n consecutive worker windows for `MEAS? n`, also while recording, logging or streaming. Measurements share one acquisition slot: a second measurement gets `BUSY`, a superseding control/calibration/range change cancels the first (`CANCELLED`), and a reload makes it `STALE`. Use a separate client connection to control while measuring. A direct board driver may use a cached single window; multiwindow measurements during a recording go through the service.

4.2 Presets, reset and the safe state

`PRESET reference`	both outputs 10 MHz, 0.5 V, in phase, no noise, playback off (the default)
`PRESET quadrature`	as `reference`, OUT1 leading by a quarter cycle (25 ns)
`PRESET off`	both sines off, outputs at 0 V
`RESET`	the default configuration (as `reference`)
`SAFE`	sines, noise and playback off; outputs at the calibrated 0 V

Your own settings can be saved on the board with `save_preset(name)` and recalled with `PRESET <name>`. Setting a phase, a preset, `RESET` or `SAFE` puts OUT1 at exactly the set phase relative to OUT2, discarding the phase accumulated so far by an offset or noise; it moves there at `phase_slew`. A preset or `RESET` sets the default slew (10^{-6}) itself: `SLEW 0` sent right after it finishes the move at once. When playback is off in the new configuration, it stops and clears the queue in the same hardware commit. RESET therefore stops queued rates before realigning the phase, without waiting for the service worker.

5 Controlling the simulator

The complete reference, with every method, command and option, is `docs/PROTOCOL.md`. The tables below are the subset used day to day.

5.1 Settings

setting	meaning	range
`outputs.<n>` (n = 1, 2; phase, slew and frequency offset for OUT1 only)		
`amplitude`, `offset`	volts at the connector: amplitude into 50 Ω (about 2.1× into 1 MΩ), offset into any load	offset ± amplitude within ≈ ±1.13 V
`enabled`	sine on or off (off: the offset alone)	
`phase`	OUT1 phase relative to OUT2, cycles; unwrapped; positive: OUT1 leads	$\pm 2^{31}$
`phase_slew`	largest fractional frequency a phase move may cause; 0: steps	$0 \dots 10^{-3}$
`frequency_offset`	fractional frequency offset y	$\lvert y \rvert \le 10^{-3}$
`noise`		
`h2`, `h0`	white PM, white FM (in the FPGA, always active)	$h_2 \le 1.6 \cdot 10^{-23}$
`h1`, `h_1`, `h_2`	flicker PM, flicker FM, random-walk FM (need playback)	
`drift_per_day`	linear frequency drift per day (needs playback)	$\lvert D \rvert \le 10^{-6}$
`pm_sine_amplitude`	sinusoidal PM, cycles (needs playback)	
`pm_sine_frequency`	its frequency, Hz	$\le f_{\text{seg}}/10$
`seed`	the same seed gives the same noise	$0 \dots 2^{64} - 1$
`dither`	OUT1 dither before the DAC rounding; use whenever the phase moves	
`playback`		
`source`	`off`, `model` or `replay`	
`replay`	the replay record's name	
`replay_loop`, `_scale`	repeat the record; multiply it	
`segment_rate`	software noise reaches half of it in offset frequency	$1 \dots 10^4$ /s
`pps`, `meter`		
`pps.width_us`, `utc`	PPS pulse width; keep the PPS on UTC after reboots	
`meter.window`	phase meter window (s)	200 ns … 1 s

5.2 TCP commands

command	action
`AMP<n> V`, `OFFS<n> V`, `OUT<n> ON|OFF`	amplitude, offset, sine on/off of output n (1 or 2); with `?` reads back
`PHASE c`, `SLEW y`, `FREQ y`	OUT1 phase (cycles), phase slew, frequency offset; `?` reads back
`SET1 JSON {...}`	several output settings in one commit
`NOISE JSON {...}`, `NOISE?`	noise settings, e.g. `NOISE JSON {"h0":2e-24,"seed":1}`
`PLAY MODEL`, `PLAY REPLAY name`, `PLAY OFF`, `PLAY?`	playback source; status
`MEAS? [n]`, `MEAS WINDOW s`	phase meter, averaged over n windows; window length
`MEAS STREAM`	every meter window, pushed as it is measured (Section 9.3)
`PPS?`, `PPS UTC`, `PPS WIDTH us`	PPS status; align to UTC; pulse width
`REC START name [JSON {...}]`, `REC?`, `REC STOP`	stability recordings
`STATE?`, `DESCRIBE?`, `STATUS?`, `*IDN?`	the applied values; a description; versions and build
`PRESET name`, `RESET`, `SAFE`	presets and safe state

Values out of range are refused with `ERR` and an explanation, and nothing is written. Every reply reports the *applied* value after rounding.

5.3 Python methods

On the client `AtomicClockClient` (computer) and the driver (board; the two marked "client" only on the computer):

method	action
`set(n, amplitude=, offset=, enabled=, phase=, phase_slew=, frequency_offset=)`	patch output n in one commit; for OUT1 also phase, slew and frequency offset
`set_phase(c)`, `set_frequency_offset(y)`	OUT1 phase (cycles), frequency offset
`set_noise(**h)`, `set_playback(**s)`	noise and playback settings
`playback_status()`, `measure(windows)`	playback status; phase meter
`pps_status()`, `align_utc()`	PPS status; align the PPS to UTC
`start_recording(name, duration_h=)`	a stability recording (also `recording_status()`, `stop_recording()`)
`download_recording(name, folder)`	copy a recording to the computer (client)
`upload_replay(name, path)`	upload a replay record (client)
`load_preset(name)`, `save_preset(name)`	presets
`reset()`, `safe()`	default configuration; safe state
`state()`, `describe()`, `status()`	applied values; description; versions and build

5.4 Noise levels from Allan deviations

The noise is set by the IEEE 1139 coefficients h_α. Data sheets usually give Allan deviations $\sigma_y(\tau)$ or phase noise $\mathcal{L}(f)$; the module `atomic_clock.noise` converts:

atomic_clock/README.md

```
from atomic_clock import noise           # on the board: from pynq.overlays.atomic_clock import
    ↪ noise
noise.h_from_adev("h0", 1e-12, 1.0)      # white FM with ADEV 1e-12 at 1 s  -> 2e-24
noise.h_from_adev("h_1", 1e-14, 1.0)     # flicker floor of 1e-14            -> 7.2e-29
noise.h2_from_dbc(-150)                  # white PM floor of -150 dBc/Hz     -> 2e-29
```

The relations behind it ($f_h = 62.5\,\mathrm{MHz}$ for the FPGA's white PM):

noise type	$\sigma_y^2(\tau)$	set from
white PM, h_2	$3 f_h h_2/(2\pi\tau)^2$	a phase-noise floor: $h_2 = 2 \cdot 10^{\mathcal{L}/10}/f_0^2$
white FM, h_0	$h_0/(2\tau)$	$h_0 = 2\tau\sigma_y^2$
flicker FM, h_{-1}	$2\ln 2\, h_{-1}$	$h_{-1} = \sigma_{\mathrm{floor}}^2/(2\ln 2)$
random-walk FM, h_{-2}	$(2\pi)^2\tau h_{-2}/6$	$h_{-2} = 6\sigma_y^2/((2\pi)^2\tau)$
drift D per day	$D_s^2\tau^2/2$ ($D_s = D/86\,400\,\mathrm{s}$)	aging from the data sheet

White PM and white FM are made in the FPGA at every 8 ns sample and need nothing else. The others are *played back*: computed by the board's service and fed to the FPGA as a sequence of rates (theory manual, "Played-back noise"); they need `PLAY MODEL` and reach up to half the segment rate (500 Hz by default). `PLAY?` shows whether playback runs; the service logs the programmed phase to `/home/xilinx/rpfpga/atomic_clock/playback/`. The same model and seed continue where they stopped, also after a reboot; changing a setting starts the model anew, always from the present phase without a jump.

Dither

Turn the dither on whenever OUT1's phase moves (noise, playback, frequency offset): `NOISE JSON {"dither": true}`. Without it the 14-bit rounding of the DAC adds 1.5–3× in Allan deviation at short τ (measured). The price is a white floor of roughly −156 dBc/Hz at 0.5 V (estimated).

6 Recipes

All recipes start from `PRESET reference` and are given as TCP lines; each has a Python equivalent in `docs/PROTOCOL.md`. OUT2 stays the clean reference throughout.

6.1 Simulate an oscillator from its data sheet

Data sheets give stability as Allan deviations $\sigma_y(\tau)$ at a few averaging times, phase noise $\mathcal{L}(f)$ at a few offsets, an aging rate and a temperature coefficient. Each figure fixes one setting (Table 1). Example: a typical single-oven OCXO with $\sigma_y = 10^{-12}$ white FM at 1 s, a flicker floor of 10^{-12}, random-walk FM reaching 10^{-12} at one day, an aging of $2 \cdot 10^{-10}$ per day and a daily temperature cycle of 10^{-11} amplitude.

All settings in one command:

```
NOISE JSON {"h0": 2e-24, "h_1": 7.2e-25, "h_2": 1.76e-30, "drift_per_day": 2e-10, "pm_sine_amplitude
    ↪ ": 1.375, "pm_sine_frequency": 1.1574e-5, "dither": true, "seed": 1}
PLAY MODEL
PLAY?
```

Section 7 gives ready settings for eleven oscillator types, from a bare crystal to a hydrogen maser.

6.2 A phase step, smooth or abrupt

```
SLEW 1e-9                  # moves at most 1e-9 in frequency
PHASE 0.5                  # a 50 ns step taken as a ramp over 50 s
SLEW 0
PHASE 0                    # an abrupt step back, on one sample
```

specification (example)	relation	h parameter / setting
white FM: $\sigma_y(\tau) = 10^{-12}$ at $\tau = 1\,\mathrm{s}$, falling as $\tau^{-1/2}$	$\sigma_y^2 = h_0/(2\tau)$, so $h_0 = 2\tau\,\sigma_y^2$	`h0` $= 2 \cdot 10^{-24}$
flicker floor: $\sigma_y = 10^{-12}$, flat in τ	$\sigma_y^2 = 2\ln 2\, h_{-1}$, so $h_{-1} = \sigma_y^2/(2\ln 2)$	`h_1` $= 7.2 \cdot 10^{-25}$
random-walk FM: $\sigma_y = 10^{-12}$ at $\tau = 1\,\mathrm{day}$, rising as $\tau^{1/2}$	$\sigma_y^2 = (2\pi)^2 h_{-2}\tau/6$, so $h_{-2} = 6\sigma_y^2/((2\pi)^2\tau)$	`h_2` $= 1.76 \cdot 10^{-30}$
aging: $D = 2 \cdot 10^{-10}$ per day	$y(t) = D\,t/\mathrm{day}$; adds $\sigma_y = D\tau/(\sqrt{2}\,\mathrm{day})$	`drift_per_day` $= 2 \cdot 10^{-10}$
daily temperature cycle: $y = A\sin(2\pi t/\mathrm{day})$, $A = 10^{-11}$	phase $\varphi = -\dfrac{f_0 A}{2\pi f_m}\cos 2\pi f_m t$ cycles, $f_m = 1/\mathrm{day}$	`pm_sine_amplitude` $= 1.375$, `pm_sine_frequency` $= 1.1574 \cdot 10^{-5}$ (a sine; only the phase of the cycle differs)
white PM floor: $\mathcal{L} = -150\,\mathrm{dBc/Hz}$ (not in this example)	$\mathcal{L} = S_\phi/2 = h_2 f_0^2/2$, so $h_2 = 2 \cdot 10^{\mathcal{L}/10}/f_0^2$	`h2` $= 2 \cdot 10^{-29}$
phase noise $\mathcal{L}(f)$ at offset f, on a slope $f^{\alpha-2}$	$S_\phi = h_\alpha f_0^2 f^{\alpha-2}$, so $h_\alpha = 2f^{2-\alpha}\,10^{\mathcal{L}(f)/10}/f_0^2$	$\alpha = 1$: `h1` (slope f^{-1}); $\alpha = 0$: `h0` (f^{-2}); $\alpha = -1$: `h_1` (f^{-3})

Table 1: From data-sheet figures to settings ($f_0 = 10\,\mathrm{MHz}$, $1\,\mathrm{day} = 86\,400\,\mathrm{s}$). The Allan-deviation relations are the standard asymptotes (IEEE 1139 [1]); the total $\sigma_y(\tau)$ is the root of the sum of the squares of all terms, so take each figure where its term dominates. `noise.h_from_adev` and `noise.h2_from_dbc` compute the first rows and the white PM floor.

A move of $\Delta\varphi$ cycles takes $\Delta\varphi/(\mathrm{slew} \cdot 10\,\mathrm{MHz})$. Whole cycles count: `PHASE 3` looks like `PHASE 0` on a scope but requests a 300 ns PPS advance, rounded to its 8 ns edge grid.

6.3 A frequency offset

`FREQ 1e-11` makes OUT1 gain 10 ps per second (0.86 µs per day) on OUT2, integrated exactly (resolution $6.8 \cdot 10^{-19}$, no drift). Changing only the offset keeps the phase accumulated so far; `PHASE 0` realigns OUT1 but preserves `FREQ`, so the phase ramp continues from there.

6.4 Replay a recorded clock

Write the time error of a real clock (seconds, positive = ahead) against time (seconds) as a two-column CSV file, e.g. `lab.csv`, and upload it (at most 50 MB):

atomic_clock/README.md

```
with AtomicClockClient("192.168.0.149") as clock:
    clock.upload_replay("lab", "lab.csv")           # TCP: REPLAY PUT ... / REPLAY DONE
```

Then `PLAY REPLAY lab`. The record is interpolated linearly; after its end it is held, or with `PLAY JSON {"replay_loop": true}` it repeats, continuing smoothly. `PLAY JSON {"replay_scale": 10}` multiplies it. Recorded data reach the output up to half the segment rate; the FPGA's white noise can be added on top (`h2`, `h0`).

6.5 PPS and time of day

DIO1_P carries the simulated clock's PPS, DIO2_P the reference's (ground on the same connector E1). At $\varphi = 0$ both rise on the same 8 ns sample. `PHASE 1` requests a 100 ns advance, realized as 96 ns with a fixed +4 ns remainder; that remainder does not average away. To put the reference on UTC seconds:

```
PPS UTC                     # arm both PPS on the next UTC second (NTP, ms accuracy)
PPS JSON {"utc": true}      # re-align by itself after every reload or reboot
PPS?                        # time of day, board_error_s, simulated_error_s, board drift
```

After arming, the PPS follow the board's crystal: `PPS?` reports the board time minus UTC and, after some hours, the crystal's fitted frequency offset. A time-interval counter comparing the edges measures their time difference: DIO1_P − DIO2_P is $-\varphi/f_0$ plus the reported grid remainder, so a positive phase gives an earlier edge. The remainder is normally below 8 ns but can exceed one sample during jumps. Digital alignment through the vendor DAC interface is verified; analog alignment at the connectors is unmeasured.

6.6 Measure stability over hours or days

The service records the phase meter on the board while you are away (every 10 ms window for the first hour, one window per second for the whole run). With the loopback cables, in `atomic_clock/tools/` on the computer:

```
python stability.py start run1 --duration-h 24      # REC START run1 JSON {"duration_h": 24}
python stability.py status
python stability.py fetch run1                      # -> recordings/run1/
python stability.py analyze recordings/run1 -o run1.pdf
```

The PDF report has ADEV, MDEV, Hadamard and time deviations with confidence intervals, the expected curves of the configured noise and a fit of the h_α. The testing manual explains the measurement, the floor of the chain and a series of tests that can be run in sequence with `run_tests.py`. The same tools analyze phase or frequency files from other instruments (`stability.py analyze-file`), compare MTIE and TDEV with ITU-T masks, and separate two external clocks and the board's own (`stability.py hat`); see its section "Further estimators and tools". To watch the deviations and the h_α grow while the run goes on, see Section 9.

The slow interval must be an integer multiple of the quantized fast window (200 ns steps). Fast and slow data retain gap flags independently, including a discontinuity between slow samples. Duration is measured by the board clock even when no usable signal arrives. A meter-window change ends the recording and preserves the new setting; write failures remain in `header.json` and status as `write_error` and a failed stop reason. The host runner checks cadence, row counts and at least 95 % acquisition coverage before analysis. A configured-noise check with too little resolved information, or a phase step across a broken unwrapping branch, is *inconclusive*.

7 Oscillator catalogue

The board can play the noise of 11 real oscillator types, from a bare crystal to a hydrogen maser [3, 4]. Table 2 lists their typical values (orders of magnitude from manufacturers' data sheets; a particular unit can differ several-fold), Table 3 the settings that simulate them: put a row's keys and values into `NOISE JSON {...}`, then `PLAY MODEL`. From Python the same settings are `OSCILLATORS[i].settings()` in `tools/clocklib/plan.py`.

From values to settings. Each power-law term is fixed by its Allan deviation at one τ (white FM at 1 s, the flicker floor, the random walk at one day), as in Table 1; the catalogue converts through the board's meter model, which agrees with the standard formulas there to about 1 %. Aging is `drift_per_day`; a daily temperature cycle $y = A\sin(2\pi t/\text{day})$ is a sinusoidal PM of amplitude $f_0 A\,\text{day}/2\pi$ cycles. Turn the dither on (`"dither": true`).

The GPS-disciplined OCXO is not a sum of power laws: a type-2 phase-locked loop (time constant 1000 s, damping 0.7) steers the OCXO's flicker, random walk, aging and daily cycle towards GPS time, whose error is white, 10 ns rms per second. It is a time-error record at 1 s, generated

by `tools/clocklib/gpsdo.py`, uploaded with `upload_replay` and played with `PLAY REPLAY gpsdo`, while the FPGA adds the OCXO's white FM (`h0`).

oscillator	noise, as ADEV				aging per day	daily cycle A
	white PM at 1 s	white FM at 1 s	flicker floor	random walk		
bare crystal oscillator (XO)	–	1.0×10^{-10}	1.0×10^{-10}	–	3.0×10^{-9}	5.0×10^{-8}
MEMS oscillator	–	1.0×10^{-11}	5.0×10^{-12}	–	1.0×10^{-9}	5.0×10^{-9}
TCXO	–	5.0×10^{-11}	3.0×10^{-11}	–	1.0×10^{-9}	1.0×10^{-8}
OCXO	–	1.0×10^{-12}	1.0×10^{-12}	1.0×10^{-12} at 1 d	2.0×10^{-10}	1.0×10^{-11}
double-oven OCXO	–	1.0×10^{-13}	1.0×10^{-13}	1.0×10^{-13} at 1 d	1.0×10^{-11}	1.0×10^{-13}
chip-scale atomic clock (CSAC)	–	3.0×10^{-10}	1.0×10^{-11}	–	3.0×10^{-11}	1.0×10^{-11}
rubidium	–	2.0×10^{-11}	3.0×10^{-13}	–	2.0×10^{-12}	–
caesium beam	–	1.2×10^{-11}	1.0×10^{-14}	–	–	–
caesium beam, high performance	–	5.0×10^{-12}	1.0×10^{-14}	–	–	–
active hydrogen maser	1.5×10^{-13}	2.0×10^{-14}	1.0×10^{-15}	–	1.0×10^{-16}	–
GPS-disciplined OCXO	–	1.0×10^{-12}	–	–	–	–

Table 2: The oscillator types and the typical values that define them.

Table 3: Settings of each oscillator type (keys and values for `NOISE JSON`; the GPS-disciplined OCXO also plays its replay record).

model	settings
M-xo	`h0` 2.0×10^{-20}, `h_1` 7.2×10^{-21}, `drift_per_day` 3.0×10^{-9}, `pm_sine_amplitude` 6.875×10^{3}, `pm_sine_frequency` 1.157×10^{-5}
M-mems	`h0` 2.0×10^{-22}, `h_1` 1.8×10^{-23}, `drift_per_day` 1.0×10^{-9}, `pm_sine_amplitude` 6.875×10^{2}, `pm_sine_frequency` 1.157×10^{-5}
M-tcxo	`h0` 5.0×10^{-21}, `h_1` 6.5×10^{-22}, `drift_per_day` 1.0×10^{-9}, `pm_sine_amplitude` 1.375×10^{3}, `pm_sine_frequency` 1.157×10^{-5}
M-ocxo	`h0` 2.0×10^{-24}, `h_1` 7.2×10^{-25}, `h_2` 1.8×10^{-30}, `drift_per_day` 2.0×10^{-10}, `pm_sine_amplitude` 1.375, `pm_sine_frequency` 1.157×10^{-5}
M-docxo	`h0` 2.0×10^{-26}, `h_1` 7.2×10^{-27}, `h_2` 1.8×10^{-32}, `drift_per_day` 1.0×10^{-11}, `pm_sine_amplitude` 1.375×10^{-2}, `pm_sine_frequency` 1.157×10^{-5}
M-csac	`h0` 1.8×10^{-19}, `h_1` 7.2×10^{-23}, `drift_per_day` 3.0×10^{-11}, `pm_sine_amplitude` 1.375, `pm_sine_frequency` 1.157×10^{-5}
M-rb	`h0` 8.0×10^{-22}, `h_1` 6.5×10^{-26}, `drift_per_day` 2.0×10^{-12}
M-cs	`h0` 2.9×10^{-22}, `h_1` 7.2×10^{-29}
M-cs-hp	`h0` 5.0×10^{-23}, `h_1` 7.2×10^{-29}
M-hmaser	`h2` 3.9×10^{-27}, `h0` 8.0×10^{-28}, `h_1` 7.2×10^{-31}, `drift_per_day` 1.0×10^{-16}
M-gpsdo	`h0` 2.0×10^{-24}, replay `gpsdo`

7.1 Bad clocks for demonstrations

The catalogue's clocks are realistic, and the good ones are subtle: an OCXO needs hours on the board's meter to show its character. For teaching, demonstrations and checking a measurement chain, six deliberately bad clocks show their fault within minutes (Table 4, Figure 2). They are illustrations, not data-sheet values. Play one on OUT1, without recording, with

```
python stability.py play                  # list the clocks
python stability.py play popcorn          # uploads its replay record if it has one
python stability.py play off              # back to RESET
```

in `atomic_clock/tools/`, or record each for one to two hours with `python run_tests.py --tier`

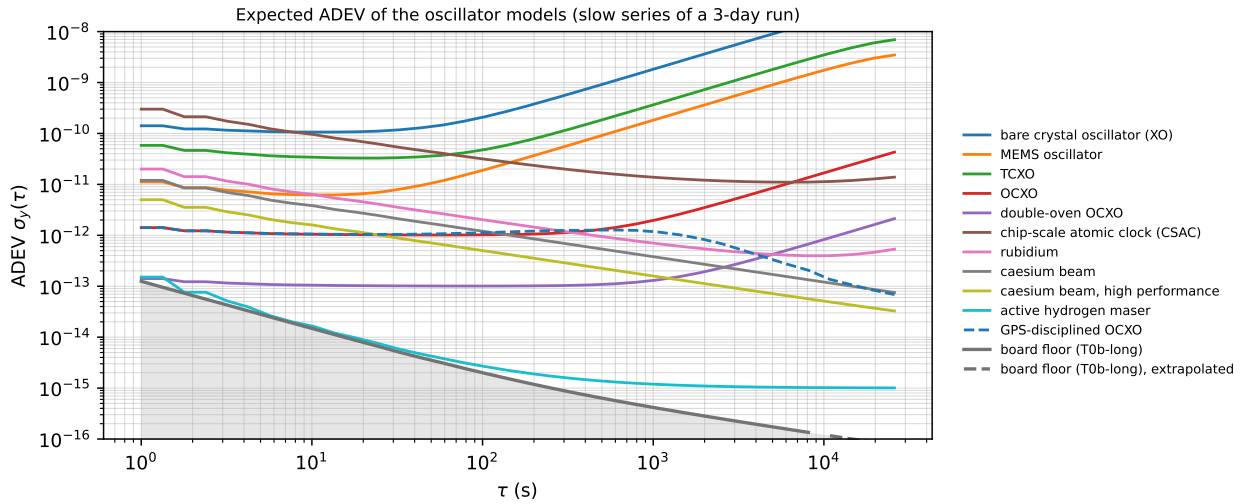


Figure 1: Expected Allan deviation of each oscillator type, and the floor of the board's own measurement (OUT1 against OUT2 in loopback): solid where measured, dashed where extrapolated. Below the floor (shaded) the board's meter cannot show the simulated clock; an external instrument with a lower floor can.

`demos` (a one-hour floor D0, then the runs `D-<key>`). `D-mains` uses 5 ms windows, 200 samples/s, to keep the 50 Hz modulation strictly below Nyquist.

clock	fault
jittery clock buffer	a cheap PLL or clock buffer: white PM at the hardware's largest setting, about 5 ns rms, visible as fuzz on a scope
crystal next to a hot CPU	an uncompensated crystal on a busy board: 1e-9 noise, fast aging, and a 15-minute thermal cycle from the load or the fan
50 Hz supply ripple	phase modulation of 1 ns at the mains frequency through the supply
popcorn frequency noise	random telegraph noise: the frequency flips between ±5e-10, 60 s mean dwell
frequency jumps	random frequency steps of 1e-9 rms about every 10 minutes (a failing rubidium lamp, a cracked crystal)
badly installed GPSDO	the GPS-disciplined OCXO with a poor antenna (100 ns rms) and a 100 s, underdamped loop: it hunts

run	settings
D-jitter	`h2` 1.3×10^{-23}
D-hot-xo	`h0` 2.0×10^{-18}, `h_1` 7.2×10^{-19}, `drift_per_day` 1.0×10^{-6}, `pm_sine_amplitude` 4.297×10^{2}, `pm_sine_frequency` 1.111×10^{-3}
D-mains	`h0` 2.0×10^{-22}, `pm_sine_amplitude` 1.000×10^{-2}, `pm_sine_frequency` 5.000×10^{1}
D-popcorn	`h0` 2.0×10^{-22}, replay `popcorn`
D-jumps	`h0` 2.0×10^{-22}, replay `jumps`
D-gpsdo-bad	`h0` 2.0×10^{-24}, replay `gpsdo-bad`

Table 4: The bad clocks and their settings (keys and values for `NOISE JSON`; a replay record is generated by `tools/clocklib/replays.py` and uploaded by `play` or the runner).

What each shows.

Jittery clock buffer White PM at the largest setting, about 5 ns rms per sample: a fuzzy edge on a scope triggered by OUT2, ADEV falling as τ^{-1} from 10^{-9} at 10 ms. The meter's 10 ms windows average it down to a few ps.

Crystal next to a hot CPU A 15-minute thermal cycle of $3 \cdot 10^{-7}$ moves OUT1 by ±43 µs against OUT2: on a scope the sine slides through hundreds of cycles; ADEV has a bump near 7 minutes on top of 10^{-9} noise and fast aging.

50 Hz supply ripple 1 ns of phase modulation at 50 Hz: a wobble on the scope. The `D-mains` run's 5 ms windows sample four times per period, so its amplitude can be fitted after allowing for the window's sinc response. At 10 ms the ripple lies exactly at Nyquist: an alternating trace depends on the modulation's phase and cannot determine its amplitude. A 100 Hz ripple would average away completely in 10 ms windows: choose a shorter window (`MEAS WINDOW`) to see it.

Popcorn frequency noise The frequency flips between $\pm 5 \cdot 10^{-10}$: the time error zigzags by 30 ns per minute; ADEV has a broad maximum of $3 \cdot 10^{-10}$ near the one-minute dwell time and falls on both sides.

Frequency jumps A random step of 10^{-9} every ten minutes or so: the time error changes slope at each; ADEV rises like random-walk FM.

Badly installed GPSDO Noisy GPS (100 ns) through a short, underdamped loop: the time error hunts by tens of ns with a period of minutes; ADEV has a broad bump at 50–300 s.

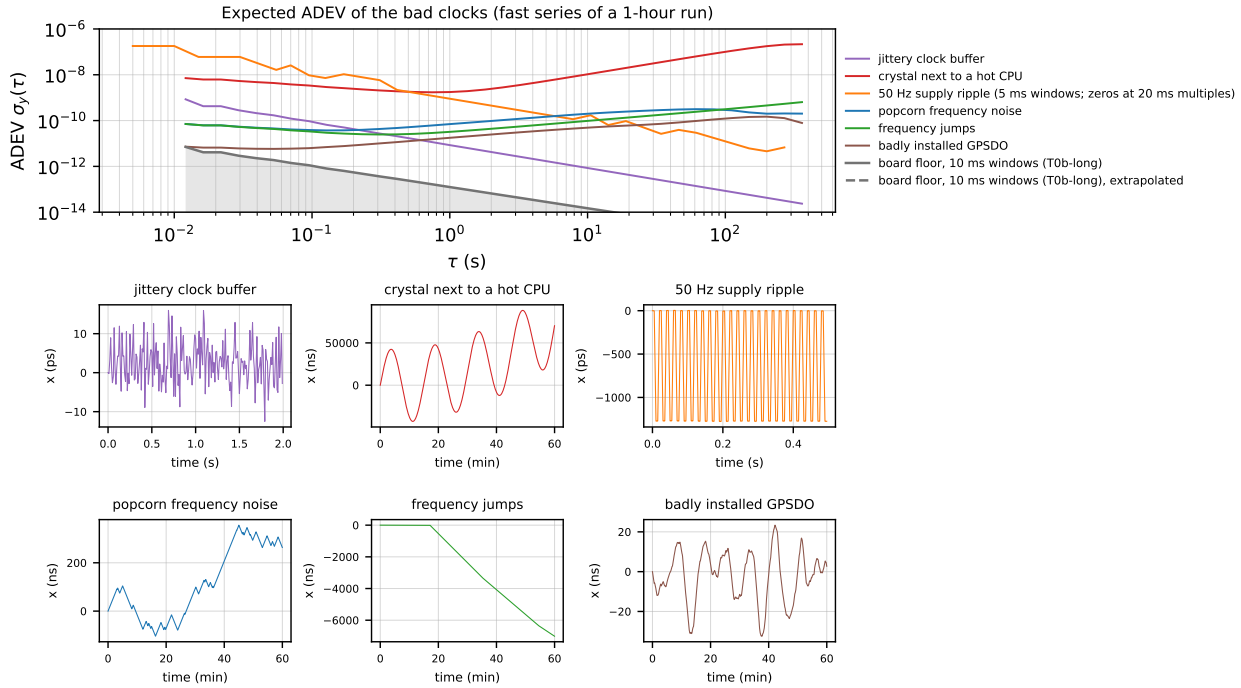


Figure 2: Synthetic. Top: expected ADEV of each bad clock in an hour's run (fast series, from 10 ms, or 5 ms for D-mains), with the board floor measured using 10 ms windows. Below: one simulated stretch of each as the meter records it (10 ms windows for the jitter, 5 ms for the ripple, 1 s otherwise). The shorter `D-mains` window and 200 samples/s cadence resolve the ripple's amplitude.

8 Applications

Five uses of the simulated clock with your own equipment, each with the cabling, the commands and the result to expect. The figures are *synthetic*: computed with the simulator's noise models, the oscillator catalogue (Section 7) and the measured floor of the board, with fixed seeds (`tools/manual_figures.py`, figures `app_*`). Nothing in this section is a measurement. Throughout, OUT2 is the clean reference and everything simulated is on OUT1 and its PPS (DIO1_P).

The board's own crystal. OUT1, OUT2 and both PPS outputs come from the board's 125 MHz crystal. Equipment that compares OUT1 or DIO1_P with OUT2 or DIO2_P sees only the simulated time error, exactly. Equipment that compares them with its own clock, GPS or NTP sees the simulated clock *plus the board's crystal*: an uncompensated crystal of order 10^{-10} at 1 s, a few ppm

off frequency and drifting (not measured; the testing manual's three-cornered hat measures it). For such tests, simulate clocks worse than that crystal, or frequency offsets, steps and other deterministic behaviour, which are exact relative changes.

8.1 Checking a frequency counter

Idea. A counter that reads a clock with known noise and known frequency offsets is checked end to end: its readings, its Allan deviation and its own resolution.

Setup. OUT1 to the counter's input, OUT2 to its external 10 MHz reference input (the counter's own time base is then not used, and the board's crystal cancels). Most counters need more than 0.5 V at 50 Ω: `AMP1 1`, `AMP2 1`. Use frequency mode with a 1 s gate and gap-free (back-to-back) readings: dead time between readings biases the Allan deviation [5]. A time-interval counter between the two 10 MHz signals is ambiguous every 100 ns, so use it only while the time error stays small.

Steps.

```
PRESET reference
NOISE JSON {"h0": 2e-22, "dither": true, "seed": 1}   # white FM, ADEV 1e-11 at 1 s
FREQ 1e-10                    # the mean reading moves by 1e-10 (1 mHz at 10 MHz)
NOISE JSON {"h0": 0}          # no noise: the counter's own floor
```

Record an hour per setting and analyze the readings with `stability.py analyze-file readings.txt --kind frequency --tau0 1` (or `adev1.py`, after summing them to phase). Each decade in ADEV is a factor 100 in `h0`: $h_0 = 2\sigma_y^2(1\,\mathrm{s})$.

What to look for (Figure 3). The ADEV of the readings is the setting's $\sigma_y(\tau) = \sqrt{h_0/2\tau}$ and the counter's own white phase noise, $\sqrt{3}\,\sigma_x/\tau$ for a single-shot resolution σ_x, added in quadrature. A counter with 20 ps resolution hides a 10^{-11} clock at 1 s and shows it from about 10 s on. The run without noise measures the counter itself: the board's outputs add a white phase noise of a few ps (estimated), below most counters. The means of the readings follow `FREQ` exactly; a counter that reports 10^{-10} as something else has a problem with its reference input or its gate.

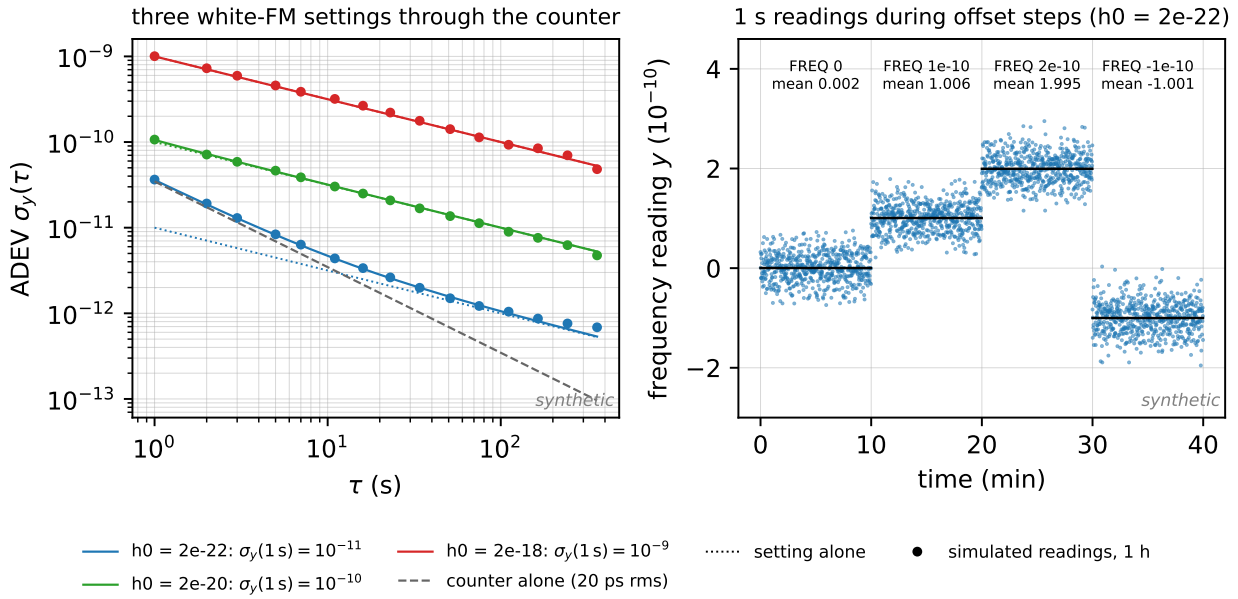


Figure 3: Synthetic. Left: ADEV of one hour of 1 s readings (points) for three white-FM settings, with a counter of 20 ps rms single-shot resolution; dotted: the setting alone, solid: setting and counter, dashed: the counter alone. Right: the readings during 10-minute steps of `FREQ`; the means are within 10^{-12} of the steps.

8.2 A PPS-disciplined NTP server

Idea. A time server (`chrony` on a Raspberry Pi or a PC) disciplined by a PPS is fed the simulated PPS. Frequency offsets, frequency steps, phase steps and poor oscillators can then be applied at will, and the server's reaction read from its logs: how fast it follows, how it handles a step, what it reports.

Setup. DIO1_P (3.3 V LVCMOS, 20 µs pulses) to the server's PPS input, ground from connector E1 (README, "Connecting to lab equipment"). On a Raspberry Pi: a GPIO pin with `dtoverlay=pps-gpio,gpiopin=18` in `config.txt`, which gives `/dev/pps0`. The PPS only marks the second; chrony numbers the seconds with another source, here NTP servers [6]:

```
# /etc/chrony/chrony.conf on the server under test
pool pool.ntp.org iburst           # which second it is
refclock PPS /dev/pps0 refid PPS poll 4 trust prefer
log tracking refclocks statistics
```

On the board, `PPS UTC` puts both PPS on the UTC second (to the board's NTP, about a millisecond).

Steps. `FREQ 1e-8` for two hours, then `SLEW 0` and `PHASE 0`: a frequency step of 0.010 ppm, followed by an abrupt phase reset after gaining 72 µs (Figure 4, left). The edge-time error is $-\varphi/f_0$: the PPS moves earlier during the positive frequency offset, returns at the phase reset, then resumes the same ramp because `PHASE 0` preserves `FREQ`. `chronyc tracking` should show a 0.010 ppm frequency change within a few loop time constants, with a transient at the phase reset rather than a programmed return to the old frequency. The phase step tests chrony's step and slew policy (`makestep`, `maxslewrate`).

What to look for. The server sees the PPS with its time-stamping noise, of order 1 µs for a GPIO interrupt (measure yours: `chronyc sourcestats`), and the simulated clock on top of the board's crystal (Figure 4, right). Within an hour, only offsets, steps and noise above the crystal's (XO, 10^{-10} and up) are visible; a simulated OCXO is hidden below the crystal. The PPS edges lie on the 8 ns grid (normally less than 8 ns late, with larger remainders possible on jumps), far below the time stamps' noise during steady operation.

Pitfalls. The board's PPS follows its crystal, a few ppm from UTC: in hours it walks milliseconds away from the pool's time (`PPS?` reports `board_error_s`). `trust` keeps chrony on the PPS while the pool disagrees; numbering the seconds needs the disagreement to stay well below half a second, so re-arm with `PPS UTC` before long runs. Clients of this server inherit the board's crystal; use an isolated network.

8.3 Replaying a GNSS satellite clock

Idea. The International GNSS Service (IGS) publishes the clock of every GPS satellite against its time scale (its multi-GNSS analysis centres also Galileo, GLONASS and BeiDou), in RINEX clock files [7]: final products every 30 s or 5 min, about two weeks after the day. Replayed on OUT1, a satellite's rubidium or hydrogen-maser clock drives your equipment.

Data. The files (e.g. `IGS0OPSFIN_20240010000_01D_30S_CLK.CLK.gz`) are at NASA's CDDIS archive (free registration, Earthdata login) and at other analysis centres. Each satellite record is one line:

```
AS G01       2017 03 11 00 00  0.000000  2    0.175309377613E-08  0.183422207046E-10
```

with the satellite, the epoch (year to seconds, GPS time), the number of values, the clock bias in seconds and its sigma. A short script writes one satellite as a replay record, with the clock's offset and rate removed:

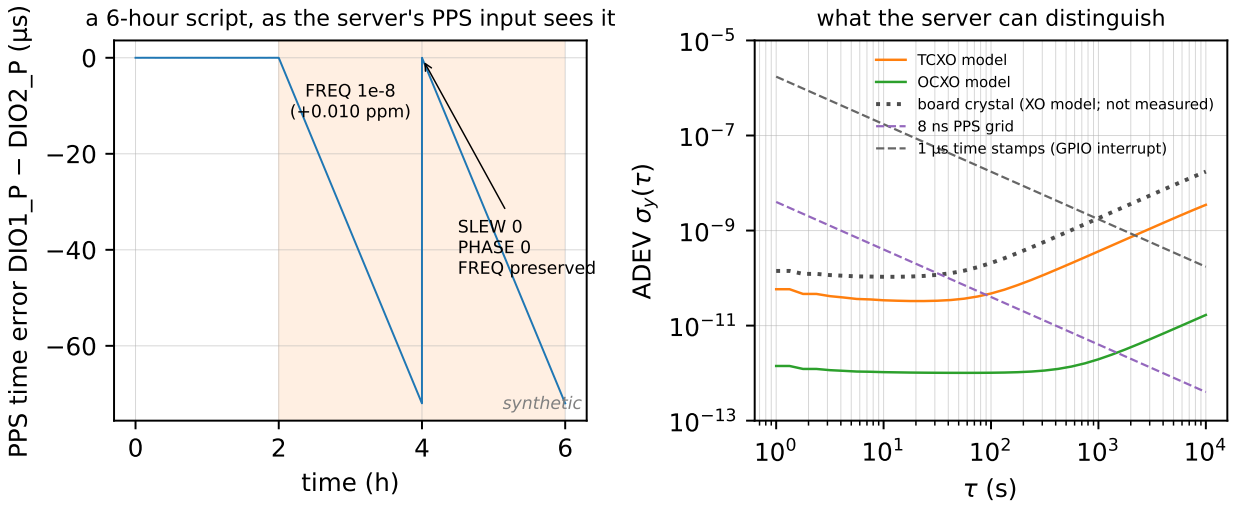


Figure 4: Synthetic. Left: edge-time error of the simulated PPS against the reference PPS, $-\varphi/f_0$ plus the grid remainder; the ramp continues after `PHASE 0`. Right: ADEV of the PPS time error for two catalogue oscillators, the board's crystal (an XO model, not measured), the 8 ns edge grid and 1 µs time stamps as white phase noise. The server sees the sum of the simulated clock, the board's crystal and its time stamps.

```python
import sys
from datetime import datetime, timezone
import numpy as np

def satellite(path, sat):
    t, x = [], []
    with open(path) as f:
        for line in f:                                  # skip the header
            if "END OF HEADER" in line:
                break
        for line in f:
            p = line.split()
            if len(p) >= 10 and p[0] == "AS" and p[1] == sat:
                y, mo, d, h, mi = map(int, p[2:7])
                epoch = datetime(y, mo, d, h, mi, tzinfo=timezone.utc).timestamp()
                t.append(epoch + float(p[7]))
                x.append(float(p[9]))                   # clock bias, s
    return np.array(t), np.array(x)

t, x = satellite(sys.argv[1], sys.argv[2])              # e.g. IGS0OPSFIN_...CLK G01
t -= t[0]
x -= np.polyval(np.polyfit(t, x, 1), t)                 # offset and rate removed
np.savetxt(sys.argv[2] + ".csv", np.column_stack([t, x]), fmt="%.3f,%.6e")
```

Then `clock.upload_replay("G01", "G01.csv")` (Section 6, "Replay a recorded clock") and `PLAY REPLAY G01`, with `PLAY JSON {"replay_loop": true}` for more than a day.

What to look for (Figure 5). The record shows the clock's noise and periodic terms at one and two cycles per orbit (about 12 and 6 h for GPS). From 30 s on, the replayed clock, measured by the board at 1 s, has the record's Allan deviation. Below 30 s the record holds no information: the replay interpolates linearly, which is smoother than any clock. Add the white FM that the satellite's clock data sheet gives (e.g. `h0` $= 2 \cdot 10^{-24}$, 10^{-12} at 1 s) to fill that range.

Pitfalls. The clock values contain the analysis's own estimation noise, of order 20 ps, which is white phase noise in the record and dominates the first points. Gaps (a satellite not observed, a clock jump) are interpolated straight across; split the file there. Keep the offset and rate removed unless

they are wanted: a satellite clock's bias can be up to a millisecond and its rate tens of picoseconds per second.

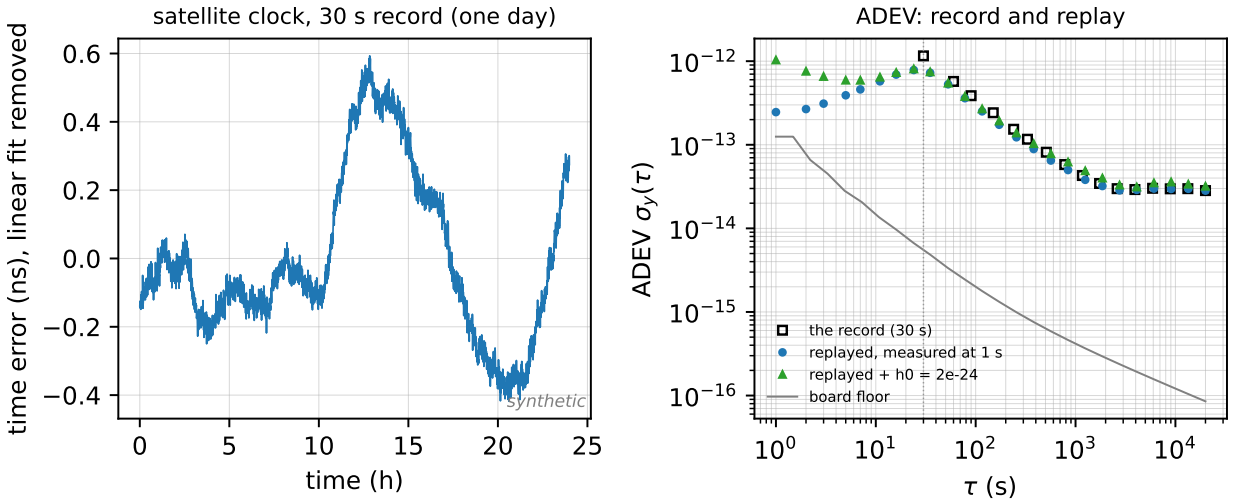


Figure 5: Synthetic: a rubidium-like satellite clock with periodic terms of 0.2 and 0.1 ns and 20 ps of estimation noise. Left: the 30 s record, linear fit removed. Right: its ADEV, and the replay measured at 1 s by the board, without and with added white FM.

8.4 Holdover test

Idea. Equipment locked to a reference (a base station, a PTP grand master, a GPS-disciplined oscillator) keeps time on its own oscillator when the reference is lost: *holdover*. With OUT1 as that oscillator, a holdover is replayed on demand: the clean reference while "locked", then at the loss of lock a catalogue oscillator with a residual frequency error. Equipment downstream of OUT1 shows when its time error leaves its budget (for 4G/5G time-division radio, $|x| \leq 1.5\,\mu s$, ITU-T G.8271 class 4 [8]).

Steps. The settings of the OCXO of Table 1; the frequency error at the loss of lock is drawn from the oscillator's ADEV at the disciplining loop's time constant, here 1000 s (10^{-12}), and the sign of the aging at random:

```
import random, time
from atomic_clock import AtomicClockClient
OCXO = {"h0": 2e-24, "h_1": 7.2e-25, "h_2": 1.76e-30, "drift_per_day": 2e-10,
        "pm_sine_amplitude": 1.375, "pm_sine_frequency": 1.1574e-5}
with AtomicClockClient("192.168.0.149") as clock:
    clock.reset()
    clock.set_noise(dither=True)
    time.sleep(3600)                                  # locked: a clean reference
    OCXO["drift_per_day"] *= random.choice([-1, 1])
    clock.set(1, frequency_offset=random.gauss(0, 1e-12))
    clock.set_noise(seed=random.getrandbits(63), **OCXO)
    clock.set_playback(source="model")                # holdover starts here
```

What to look for (Figure 6). In 200 simulated holdovers per oscillator, 95 % stay within 1.5 µs for about 3 minutes with a TCXO (its daily temperature cycle dominates), 9 hours with the OCXO (aging, which grows the time error as t^2) and the whole day with rubidium (0.2 µs). One run on the board is one of these curves; repeat with new seeds for the spread.

Pitfalls. Real holdover performance depends on the temperature during the test and on how well the loop had learned the oscillator's frequency and aging before the loss, which the catalogue cannot know. The budget usually has to cover the network as well, so the holdover gets only part of it.



Figure 6: Synthetic: $|x|$ after the loss of lock, 200 runs per catalogue oscillator (frequency error at the loss from the ADEV at 1000 s, aging of either sign, daily cycle at a random phase). Solid: 95 % of runs below, dashed: the median; dots: where the 95 % curve crosses 1.5 µs.

8.5 Teaching the noise types

Idea. The five power-law noises and drift are hard to tell apart in a time series and easy in an Allan-deviation plot. A script plays each for an hour; students measure with a counter, compute ADEV and modified ADEV, read off the slopes and identify the noise (Table 5).

Setup and steps. The counter as in Section 8.1. The segment rate of 1/s limits the played noise to 0.5 Hz, the Nyquist frequency of 1 s readings, so that it does not alias; white PM comes from the FPGA at every sample and is read by the counter as jitter:

```
import time
from atomic_clock import AtomicClockClient
SEGMENTS = [("white PM", {"h2": 5.7e-24}), ("flicker PM", {"h1": 3.1e-17}),
            ("white FM", {"h0": 5e-18}), ("flicker FM", {"h_1": 1.8e-19}),
            ("random-walk FM", {"h_2": 3.8e-21}), ("drift", {"drift_per_day": 1e-6})]
OFF = dict.fromkeys(("h2", "h1", "h0", "h_1", "h_2", "drift_per_day"), 0.0)
with AtomicClockClient("192.168.0.149") as clock:
    clock.reset()
    clock.set_playback(source="model", segment_rate=1)
    for name, setting in SEGMENTS:
        clock.set_phase(0)                              # each segment from x = 0
        clock.set_noise(**dict(OFF, **setting), dither=True, seed=1)
        print(time.strftime("%H:%M:%S"), name)
        time.sleep(3600)
```

The levels give about $5 \cdot 10^{-10}$ in ADEV at 10 s; drift at the largest setting reaches it at about a minute. The white-PM level as read depends on the counter's input bandwidth.

What to look for (Figure 7). One hour of readings gives slopes within about 0.1 of the table, with two exceptions: flicker PM's ADEV falls a little more slowly than τ^{-1} (a logarithmic factor), and the counter's own white PM flattens the drift's slope at short τ. Questions for students: which

noise	ADEV slope	MDEV slope	typical source
white PM	-1	$-3/2$	thermal noise of amplifiers, the counter itself
flicker PM	-1	-1	amplifiers, multipliers, cables
white FM	$-1/2$	$-1/2$	atomic standards (shot noise of the resonance)
flicker FM	0	0	quartz resonators: the flicker floor
random-walk FM	$+1/2$	$+1/2$	environment: temperature, vibration
drift	$+1$	$+1$	aging

Table 5: Slopes of $\sigma_y(\tau)$ on log-log axes [1, 5]. White and flicker PM have the same ADEV slope; the modified ADEV separates them.

segment could not be told apart in ADEV alone, why does random-walk FM wander so far in the time series, and what does the drift look like in the readings themselves?

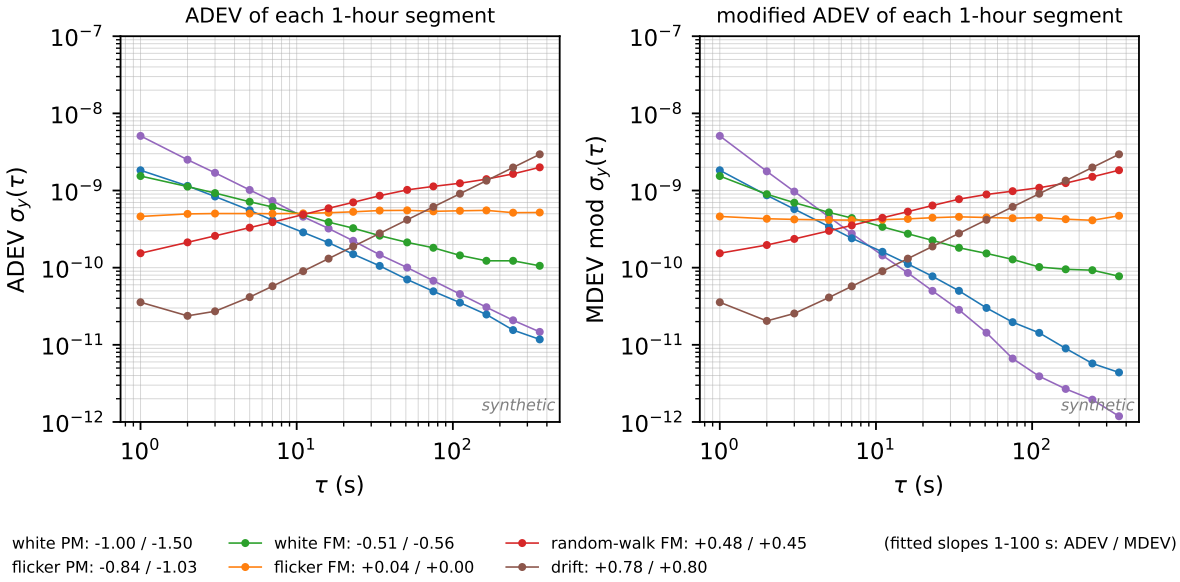


Figure 7: Synthetic: one hour of 1 s counter readings (20 ps resolution) per segment; ADEV (left) and modified ADEV (right), with the slopes fitted between 1 and 100 s.

9 Watching the noise live

A stability recording is analyzed after it is downloaded. `rustle`, a program on the computer, does the same analysis *while the data arrive*. It shows a dashboard in the browser with:

- the Allan, modified Allan, Hadamard and time deviations, with confidence intervals that tighten as the run goes on;
- the fitted coefficients $h_2 \dots h_{-2}$ (IEEE 1139), the frequency offset and the linear drift, each with an uncertainty;
- the power spectral density, a $\sigma_y(\tau)$ table at adev1's τ values, and the phase and frequency over the run.

It suits a first look before a long run, demonstrations and teaching. The recorded tests of the testing manual remain the reference.

9.1 Starting it

In `atomic_clock/tools/` on the computer, with the conda environment `base-tools` (numpy, scipy, allantools, fastapi, uvicorn):

```
python -m rustle --board 192.168.0.149     # the board's phase meter, live
python -m rustle --demo ocxo --speed 1000  # a simulated OCXO, 1000 s of data per second
python -m rustle                           # choose the source in the browser
```

Then open `http://127.0.0.1:6767`. With `--bind 0.0.0.0` other computers can open it too. There is no password, so use this on the lab network only. Several browsers may watch one session, for example a projector and a laptop. Any of them can change it, and the last change wins.

The sources are:

Board every meter window as it is measured (`MEAS STREAM`, Section 9.3). The window (`MEAS WINDOW`, 10 ms by default) is the sample spacing. The stream goes on during a recording, so a long run can be watched as it is recorded. When the connection drops, `rustle` reconnects and continues with a new segment.

Demo a simulated clock pair: any clock of the catalogue (Section 7), a bad clock for demonstrations, or your own h values, with an offset, a drift and a window of 1 ms to 1 s. The data look as the meter would measure them. The speed runs from 1× to 1000× or as fast as the computer can go: a day of 10 ms data passes in under a minute and a half.

Recording a downloaded recording, its fast or slow series (`--recording recordings/run1 --series slow`).

File phase or frequency data of another instrument, read as by `stability.py analyze-file` (`--file counter.txt --data frequency --units ppb --tau0 1`).

`--headless` prints the tables instead of serving the dashboard, and `--json` saves them.

9.2 Reading the results

Deviations. The same values as `stability.py` and allantools. Each point's shaded band is a confidence interval (68.3 % by default; 90 and 95 % in the settings). The plot and fit use a fixed 1–2–5 plus octave grid, independent of the table's choice of τ. Stop, end of file and errors finish the pending MDEV terms.

Coefficients. Each h_α comes with two uncertainties. *Fit* is taken from the fit itself and is conservative (2–6× too large in the checks). *Bootstrap* is the spread of 20 simulated copies of the fitted clock. Through 2^{21} timeline samples, including gaps, `segment-exact` preserves the segments and each estimator's term counts, including pending MDEV tails. Above that limit, `edf-rescaled` is an explicitly labelled approximation using shorter records: it reproduces marginal scatter approximately, without the joint covariance of deviations and trend. Earlier checks agreed within about 20 % for resolved terms; this is not a guarantee for sparse, gap-dominated runs. The table also gives the range of τ in which each term dominates. A term without such a range is not measured, only bounded.

Offset and drift. A least-squares fit of the phase, with the offset given at mid-run. Their uncertainties allow for the noise that was fitted: with random-walk FM they are much larger than a plain least-squares error. They use actual segment endpoints and sample counts; even two-sample segments constrain a slope.

Gaps and flags. Lost windows, windows without signal and setting changes on the board (phase, slew, frequency offset, playback restarts) start a new segment. The deviations pool segments by their numbers of overlapping terms, with correspondingly weighted degrees of freedom; the offset fit keeps one phase intercept per segment. A change of the meter window ends the run.

Each analysis iterates the fit, model-dependent confidence intervals and drift-noise correction to a consistent result, including the first headless snapshot. Changing *fit drift* cancels old bootstrap work and clears its errors; bootstrap rounds track the model revision and estimator counts as well as the number of samples.

Export. The download button saves JSON with the normalized source configuration and model kernels, actual segment endpoints/counts, analysis options and model revision, deviations and their intervals, coefficients and both errors, spectrum and strip chart. The strip chart includes the original block-mean phases, segment IDs and samples per point, as well as detrended data; it is compressed display data, so retain the source recording for full-resolution reanalysis.

Limits

- In short runs (10 minutes to an hour), h_{-2} comes out 10–40 % low: a little flicker FM takes its place. Judge random-walk FM from runs of several hours.
- The meter cannot follow OUT1 when its phase moves more than about a quarter cycle in one window: the window's amplitude is lost (by |sinc| of the cycles moved), and the board drops it. With 10 ms windows that means a frequency offset of at most about $2.5 \cdot 10^{-6}$. For faster motion, shorten the window.
- The meter's own floor is not a sum of power laws. For board data, give a floor recording (`--floor recordings/T0b`, outputs looped back, no noise). Its deviations are then subtracted in the fit. With a floor, the bootstrap errors are a lower bound: use the larger fit/bootstrap error. The dashboard's significance rule does so when a measured floor is supplied.
- Periodic terms (a thermal cycle, mains ripple) and steps are not modelled. The fit absorbs them into the nearest power law, and its χ^2 rises.

9.3 The meter stream

`rustle` reads the board through `MEAS STREAM`, which any program can use. On a connection of its own, it sends every meter window: its index and time, both inputs' unwrapped phases (cycles) and amplitudes (V), and a flag that marks where the phase is not continuous with the previous row. From Python:

```
with MeterStream("192.168.0.149") as s:
    rows, lost = s.read()       # every interval: [(index, t_s, in1, in2, amp1, amp2, window_s,
    ↪ flags)]
```

The board unwraps IN1 against OUT1's predicted phase (frequency offset, played-back noise, slew), which it knows exactly. Windows without signal are not sent. The stream runs beside recordings; it raises the CPU load of the board's service from 39 to 67 % (measured, with model playback). The details are in `docs/PROTOCOL.md`, "Meter stream"; the dashboard's own documentation, with its options, is `tools/rustle/README.md`.

10 Where next

document	contents
`docs/manual/theory_manual.pdf`	how the signal is made: exact carrier, phase word, noise generators, playback, output stage, PPS, limits
`docs/manual/testing_manual.pdf`	the phase meter, stability recordings and statistics, the tests, oscillator models, measured results
`docs/PROTOCOL.md`	every setting, method, TCP command and command-line option
`tools/rustle/README.md`	the live dashboard: sources, options, what is computed, API
`docs/THEORY.md`	the theory in short, including the phase meter
`docs/VERIFICATION.md`	what was tested, when, with what result
`README.md`, `common/SETUP.md`, `common/CALIBRATION.md`	overview and specifications; board setup; calibration

References

[1] IEEE standard definitions of physical quantities for fundamental frequency and time metrology — random instabilities. IEEE Std 1139-2022, 2022. Earlier editions 1988, 1999, 2008.

[2] A. E. E. Wallin et al. AllanTools. Python library for Allan deviation and related statistics, https://github.com/aewallin/allantools.

[3] F. Riehle. *Frequency Standards: Basics and Applications*. Wiley-VCH, Weinheim, 2004.

[4] J. Levine. Introduction to time and frequency metrology. *Rev. Sci. Instrum.*, 70(6):2567–2596, 1999.

[5] W. J. Riley. *Handbook of Frequency Stability Analysis*. NIST Special Publication 1065. National Institute of Standards and Technology, 2008. The practical reference for the deviations, their confidence and noise identification.

[6] chrony: a versatile implementation of the Network Time Protocol. https://chrony-project.org. Documentation of `chrony.conf`: directives `refclock`, `pool`, `makestep`.

[7] J. Ray, W. Gurtner, and M. J. Coleman. RINEX extensions to handle clock information, version 3.04. International GNSS Service, https://files.igs.org/pub/data/format/rinex_clock304.txt, 2017.

[8] Time and phase synchronization aspects of telecommunication networks. ITU-T Recommendation G.8271/Y.1366, 2020.