Red Pitaya lab instruments: board, setup, calibration and framework

overview and safety, setup, calibration, framework and service

Rok Zitko, Jozef Stefan Institute

2026-11-10 (commit 6178370)

Contents

Real-time analog signal processing on a Red Pitaya **1**
What you need . . . 5
Connecting to lab equipment: read first . . . 5
First signal in ten minutes . . . 7
Board specifications . . . 7
What these designs cannot do . . . 9
Documentation . . . 9
Acknowledgements . . . 9
Author . . . 10

Setting up a board and a build machine **10**
A ready SD card (quickest) . . . 10
The board (once) . . . 11
A build machine (for changing or rebuilding designs) . . . 11

Board calibration **12**
What calibration corrects . . . 12
Input ranges . . . 12
Procedure . . . 12
Output frequency response and delay (optional) . . . 14
Input frequency response and loop delay (optional) . . . 15
Delay of the digital pins (optional) . . . 15
Slow analog outputs and inputs (optional) . . . 15
Accuracy to expect . . . 16
For design authors . . . 16

Common infrastructure for Red Pitaya designs **17**
Layout . . . 17
Make targets and variables . . . 17
Optional common RTL . . . 19
System ID . . . 19
Safe loading . . . 20
The uniform driver API . . . 20
The TCP service . . . 22
Record rings . . . 23
Stability tools . . . 25
Command line, client and GUI . . . 25
Adding a new design . . . 26

Real-time analog signal processing on a Red Pitaya

FPGA designs that turn a **Red Pitaya STEMlab 125-14** into lab instruments for analog signals. Two inputs and two outputs are processed sample by sample at 125 MSa/s, with a fixed pipeline delay of a few hundred nanoseconds and optional output filtering in mult_add; atomic_clock instead generates an exact 10 MHz with a programmable

phase, synth sums 16 phase-coherent tones on each output, awg is a two-channel function, pulse, noise and arbitrary waveform generator with eight synchronized digital outputs, lockin is a four-demodulator lock-in amplifier with oscillators, PID controllers and phase-locked loops, and feedback is a lock box built on two of lockin's demodulators with full-rate PIDs, scan-and-relock acquisition and the slow analog inputs and outputs; timetagger timestamps digital edges with picosecond resolution, and ajku analyses the statistics, distributions and spectra of two simultaneous I/Q inputs, such as balanced optical receivers. You control them from Jupyter, Python, or any program that can open a TCP socket.

Design	What it does	Typical uses
mult_add	Each output is a weighted sum of 13 functions of two conditioned inputs X and Y, followed by gain/offset, an optional full-rate low-pass filter and Schmitt comparator, then saturation, mute and a final protectable clamp. Also built-in meters and calibration.	analog multiplier and mixer, filtered lock-in output, laser-power normalization, error-signal conditioning, envelope detector, hysteretic threshold detection, sum and difference
analog_echo	OUTn = gain · INn + offset on each channel.	buffering, inverting and offsetting signals, a quick check of the signal chain
atomic_clock	A 10 MHz sine, exactly 2/25 of the sample clock, on both outputs; OUT1 with a smoothly programmable phase, frequency offset and phase noise (white PM/FM in the FPGA; flicker, random walk, drift or recorded data played back by the board's service); 1 PPS outputs; a 10 MHz phase meter on IN1/IN2.	simulating an atomic clock's 10 MHz and 1 PPS with realistic phase noise from microseconds to a year, against a clean reference; measuring 10 MHz phase
awg	A two-channel function generator: sine, square, pulse, ramp/triangle (adjustable edges with sub-sample timing), PRBS, DC and tables (Gaussian, sinc, Lorentzian, exponential, your own); a second generator added or modulating; Gaussian noise of adjustable bandwidth; AM/DSB with FM, PM, PWM, FSK or BPSK by the second generator, IN1/IN2 or the other output; linear, log, list and profile sweeps with a marker; N-cycle and gated bursts with start and stop phase; triggers from pins (1 ns timestamps), thresholds, timers or software with delays; two markers; lockable voltage limits; arbitrary waveforms from the board's memory (OUT1/OUT2 point pairs at a fractional rate up to 125 MSa/s, smooth/linear/step reconstruction) in sequences with repeats, waits and gapless event jumps; I/Q at baseband or upconverted; eight digital outputs (patterns, counts, PRBS) aligned with the analog outputs, started by bursts or sequence steps.	pulses and bursts for experiments, modulated and swept test signals, noise injection, triggered or gated waveforms, phase-locked pairs, arbitrary and I/Q waveforms, digital patterns synchronized with them
feedback	A lock box: lockin's core with two oscillators and two demodulators, its four frame-rate PIDs (also on IN1/IN2 means and the slow inputs, to the slow outputs), plus two fast PIDs at 125 MSa/s (on IN1, IN2, IN1 − IN2 or a demodulator's mixer product; two low-pass sections, P, I, D, limits; added to an output) with lock acquisition (a triangle scan, engage at a level, relock after a loss); slow analog outputs AO0–AO3 (PWM, delta-sigma) and inputs AI0–AI3 (XADC); four status pins; a TTL reference input (1 ns: PLLs to an external clock, also at N/M of it), a sync output (2 ns edges), a 1 ns capture trigger; PI gains tuned from the measured loop (phase and gain margins, the closed loop measured); the stream, capture, counter, sweeps and fits of lockin.	laser and cavity locks (Pound–Drever–Hall, side of fringe, scan and relock), fast stabilization of a voltage, slow loops on temperature or piezo offsets, PLLs to signals or to a lab clock, lock-in measurements beside a loop

Design	What it does	Typical uses
lockin	A lock-in amplifier: four oscillators (0.44 µHz steps, continuous or coherent phases) summed onto the two outputs; four demodulators on IN1, IN2 or IN1 − IN2 at a harmonic (up to 1023) of an oscillator, optionally with a sideband of another, sine or square reference; 1 to 8 RC sections (τ from 0.5 µs); X, Y, R, θ, unwrapped phase and frequency deviation in volts at the connectors; four PIDs (to an oscillator's frequency or phase, an output's amplitude or offset: PLLs, amplitude and DC loops) with hold and lock detection; monitors; a stream of X/Y rows, a 4-channel capture, a frequency counter; sweeps with resonance fits, spectra, auto-phase; the zoom spectrum and noise density around a reference, a time constant chosen from the signal, Allan deviations; an SR86x-compatible command port (TCP 23) for programs written for a Stanford Research SR860/SR865.	weak signals, Bode plots and resonances (f0, Q), harmonics and intermodulation, locking an oscillator to an external signal or a resonance, amplitude and offset stabilization, continuous logging of demodulated signals
synth	16 tones on each output, each with its own frequency (0.44 µHz steps), amplitude and phase relative to one zero time shared by both outputs; changes on one sample for all tones; linear and shaped ($\cos^2$, Blackman, minimum-jerk) frequency/amplitude ramps and timed sequences in the FPGA, repeated by the FPGA itself, with a TTL trigger and a marker; amplitude modulation by IN1/IN2; loopback capture, tone measurement and spectra on IN1/IN2.	phase-locked signals with defined phase differences, frequency combs, sweeps and chirps, multi-tone RF for acousto-optic deflectors and tweezer moves, triggered or repeated phase-coherent pulse trains
timetagger	Ten-channel time tagger: carry-chain delay lines (160 taps of ~30 ps, 0.98 ps ticks) on DIO0_P..DIO7_P and the SATA S2 pairs, self-calibrated by code density with an on-chip ring oscillator (and live from the channels' own hits); edges, dead time, dividers, offsets; acquisitions started now, at a sample or armed on a PPS; timestamps through a DDR ring and a binary TCP stream to PC programs (g2 correlations, TCSPC, coincidences); a 10 MHz IN2 phase meter, a phase-stepped test signal and an event generator for self-tests.	photon correlations (HBT, TCSPC), coincidences, pulse and clock timing
ajku	CW I/Q analyser for balanced receivers: raw broadband statistics exact through fourth order in every 8.4 ms block (continuous cumulants, kurtosis), full-rate FPGA histograms with configurable windows, continuous FPGA spectra (exact 1024-point Hann spectra of I + iQ to 25 MHz, auto, cross, complex and complementary), finite paired captures (also hardware-timed), PPS and slow-input (AI0–AI3) logging in every statistics block; host raw/FIR spectral matrices with converter-skew correction, distributions/cumulants, Allan deviations, LO-power fits and drift follow-ups. Digitally qualified on the board (1.0, a one-hour workload without loss) with a user's manual on continuous-variable theory and end-user characterization.	calibrated CW quadrature/noise analysis and quantum-optics teaching

Design	What it does	Typical uses
counter	Gated photon counter: eight count inputs on DIO0_P–DIO7_P sampled every 1 ns with a non-paralyzable dead time and a delay per channel, 48-bit totals, RATE records with exact intervals streamed to a PC; shots from TRIG (DIO0_N, an internal timer or software) with four gates each (delay, width, a delay step per point for time traces, or the GATE level), points per shot or per NEXT and sweeps from SEQ (checked), per-point sums collected into 64-bit totals on the board, per-shot records; two on-board decisions per shot (a count window on each of two gates, any logic of the two) on output pins as a level, pulses or early (114 ns simulated from the gate's last ns to the pin), in the records and the sums; a channel's echo on an output; IN1/IN2 as two more count channels (a discriminator at a threshold in volts, positive or NIM-like negative pulses, interpolated to 1 ns, aligned with the pins); two coincidence units of up to four channels (timetagger's clusters, the 15 exclusive patterns in 48-bit totals, one channel delayable for the accidentals, in the RATE records, a herald output); a rate alarm, rate monitors (or test pulses) on OUT1/OUT2, a test photon emitter (pairs, an antibunched source, bright or dark per shot from a table, with a decay); presets, recipes and analysis (contrasts, fits, thresholds and fidelities, g2, α, CHSH). Simulated against a bit-exact model; board-tested without a detector (45 checks: the emitter, OUT1→IN1, a 10 MHz sine on IN2, an output wired back) and with awg on a second board and an oscilloscope (exact counts from awg's patterns, decisions 105 ns after the gate's end, a preselection loop through awg in 356 ns, the analog inputs aligned with the pins within 0.5 ns), and on awg's random patterns with exact expectations (sums, records and coincidences; accidentals, dead time with a known rate, repeat until success). A user's manual covers the instrument, the theory of counting and coincidences, and its use with NV centres, ions and atoms, and photons.	NV pulsed readout, ion/atom state detection, teaching-lab coincidences (with a pulse generator such as awg)
spin	Pulsed RF transceiver (1.1): a pulse program on the board with nested loops, tables, WAITs on a 1 ns trigger that locks the RF phase to the edge, and branches on single-shot states, the COND inputs or a loop counter's parity; two transmitters to about 50 MHz (phase-coherent frequency registers, phases and amplitudes in volts, envelope shapes, chirps) on two real outputs or an I/Q pair with Q corrections; lockable limits and a duty guard; two receivers on the transmitters' oscillators with on-board averaging in swapped halves and full-rate weighted results with a state per shot; eight digital outputs (TTL bits, power-amplifier gates, states), slow outputs AO0–AO3 set by the program; a qubit and spin-ensemble emulator; templates (FID, echoes, CPMG, NQR echo trains, recoveries, nutation, ESR field sweeps, Rabi, Ramsey, T1, readout, active reset) with fits, presets, recipes and a Jupyter panel.	teaching pulsed NMR and NQR, pulsed ESR and qubit readout at an intermediate frequency (behaviour with real samples unmeasured)

Design	What it does	Typical uses
boxcar	Pump–probe boxcar averager and segmented scope (1.0: TTL, analog, timer and software triggers, categories from a chopper or a pattern, points and sweeps, eight timed pulse outputs with an electronic delay table, markers, a snapshot, a test source on OUT1/OUT2 emulating a laser and a sample, dead-time-free averaged traces of IN1/IN2 per category, every shot's raw segment into DDR, eight gates per shot with per-shot normalization by a reference detector, pump-on/off pairs, sums per delay point totalled over sweeps, per-shot records streamed to a PC, a hold for a PC-driven stage, analog boxcar outputs with lockable limits; presets, recipes, a Jupyter panel and a live window for a PC; a user's manual and a theory manual). Board-tested through a cable loopback (ΔT/T of an emulated sample within 0.5 %, every shot's gates at 2 MHz).	pump–probe with amplified (1–10 kHz) and high-repetition (50 kHz–2 MHz) lasers (behaviour with real lasers unmeasured)

All settings are in **volts at the connectors**, using a calibration measured on this board. The board runs one design at a time; switching takes a few seconds.

What you need

- A Red Pitaya STEMlab 125-14 running PYNQ, set up once as described in common/SETUP.md, or from an SD card written with the ready image that `common/scripts/make_sd_image.sh` builds (all designs installed; common/SETUP.md). The current board address is set once in the repository-root config.mk.
- SMA cables or SMA–BNC adapters for IN1, IN2, OUT1 and OUT2.
- A browser for Jupyter, or Python 3 on your computer.

Using the designs needs no FPGA tools. Changing them needs Vivado 2025.1 (common/SETUP.md). On that machine, `make upload` in the repository's top directory builds what is out of date and puts all designs on the board (`make upload NOTEBOOK_OVERWRITE=1` also replaces the notebooks there, keeping backups); it does not change what the board runs.

All design Make workflows inherit the root `config.mk`, including direct `make -C ajku ...` calls and the standalone LED example. A design-local `config.mk` can override it, and command-line assignments take highest priority.

Connecting to lab equipment: read first

The outputs can do unexpected things at the moments listed below. Before connecting anything that must not see an unplanned voltage (laser current drivers, piezo and modulator drivers, cryogenic wiring), plan for each of them.

- **Power-up and reboot.** For about two minutes after power-on, the FPGA is not configured and the output voltage has **not been characterized**: treat it as undefined. Then the control service loads the last used design (from a freshly written ready SD card: the one chosen when the image was built, by default analog_echo, set by `IMAGE_ACTIVE` in the repository-root config.mk; with no design saved on the board, the first installed one, alphabetically). mult_add and analog_echo start as an **uncalibrated unity passthrough** (OUT ≈ IN; on this board about 0.6% gain error and up to about 25 mV offset); atomic_clock starts with an **uncalibrated 10 MHz sine** of about 0.57 V amplitude on both outputs; synth, awg, lockin and feedback start with **all tones (outputs) off**, and timetagger, ajku, counter and spin with **both outputs at 0** (0 in converter units, a few mV; feedback's slow outputs off; timetagger never drives its outputs). A moment later the service **restores the last settings, including offsets**.
- **Loading or switching a design** (`LOAD` or a driver download) starts an **uncalibrated passthrough** (atomic_clock: the uncalibrated 10 MHz sine; synth, awg, lockin, feedback, timetagger, ajku, counter and spin: everything off), with clamps disabled and protection cleared. `LOAD` does not restore settings. `make test-remote` programs the FPGA, checks power-up settings, then applies its selected preset. **`make upload` only copies/installs files**; a subsequent driver attachment may load a changed build. Restarting/installing the service can also load and restore a design.

- **Calibration** drives the outputs on its own. Step 2 holds both outputs at about +0.56 V on this board. Step 3 sweeps them between about −0.83 V and +0.83 V. The reference hold persists until `CAL RELEASE` or a superseding edit. Automatic loopback calibration is refused while any mult_add clamp or awg limit is protected; reference holds outside a protected window are refused before writes. `make test-hw` drives test signals up to about ±0.8 V. Disconnect sensitive equipment first.
- **Uncalibrated volts are wrong.** Without the board calibration, outputs on this board are about 14% larger than requested.
- **Range and limits.** Outputs reach about ±1.13 V at DC. At 10 MHz a high-impedance load sees about twice the set amplitude (see "Load impedance"): up to about ±2.4 V at full scale. mult_add 3.0 can clamp each output to an inward-rounded 14-bit DAC window (`set_clamp(1, vmin=..., vmax=..., protect=True)`). Protection forces the clamp on and locks its bounds until an FPGA bitstream reload, including across ordinary core reset, presets and `SAFE`. Set and read back the window before connecting an actuator. With the FPGA configured and clocks running, the digital stream is bounded from the first DAC capture edge after the commit. **This is not an absolute connector-voltage guarantee:** analog settling, ringing, calibration error, loading, power loss and reconfiguration are outside the clamp's control. The vendor DAC reset independently forces converter data to zero, even for a window excluding zero. awg has per-output voltage limits with the same lock (`protect(n)`), applied to every sample after all sums, modulation, noise and dither; lockin and feedback likewise (`protect(o)`), after the oscillators' sums, the PIDs' terms and the monitor (feedback: also the fast PIDs' terms and scans). spin clamps each output to lockable limits and has a duty guard per transmitter (the same lock, `protect(n)`). analog_echo, atomic_clock, synth, ajku and counter have no clamp (timetagger holds its outputs at 0); synth by default refuses settings whose worst-case peak (|offset| + the sum of its tones' amplitudes) leaves the DAC range.
- **Mute and `SAFE` can be nonzero.** mult_add mute selects the DAC level nearest calibrated 0 V, then the final clamp selects the nearest allowed level. A protected +0.2…+0.8 V window therefore mutes near **+0.2 V**, including on `SAFE`; a wholly negative window behaves analogously. `SAFE` preserves protected clamps but resets unprotected settings. analog_echo `SAFE` sets gain 0 and offset 0 (approximately calibrated 0 V); atomic_clock `SAFE` turns both sines off at calibrated 0 V; synth `SAFE` mutes every tone at once, stops any sequence and sets calibrated 0 V; awg `SAFE` mutes both outputs at once (0 V or the nearest protected limit), then switches everything off; lockin `SAFE` switches both outputs off (0 V or the nearest protected limit) and stops its PIDs; feedback `SAFE` also stops its fast PIDs and scans, switches the slow outputs off and sets the status pins low; timetagger `SAFE` switches every channel, the event generator, the test signal and the ring oscillator off; ajku `SAFE` returns both outputs to calibrated 0 V and aborts an active capture; counter `SAFE` puts DIO4_N–DIO7_N in high impedance, the rate monitors at 0 V and the test emitter off; spin `SAFE` puts DIO0_N–DIO7_N in high impedance. **A lockin or feedback PID moves an output on its own** (amplitude or offset targets) within its limits; feedback's fast PIDs and scans move an output at the full rate, and its slow PIDs may move the slow outputs.
- **Digital pins.** atomic_clock drives two extension-connector pins (DIO1_P, DIO2_P) with 3.3 V PPS pulses; synth reads a trigger on DIO0_P (3.3 V, pull-down) and drives a marker on DIO1_P; awg reads triggers on DIO0_P and DIO3_P and events on DIO4_P–DIO7_P (all with pull-downs), drives markers on DIO1_P and DIO2_P, and drives the digital outputs D0–D7 on DIO0_N–DIO7_N, each only when enabled (high impedance otherwise, after loading and after SAFE); lockin reads a capture trigger on DIO3_P (pull-down); feedback reads it too (sampled to 1 ns), reads a TTL reference on DIO0_P (pull-down), drives a sync output on DIO1_P (a square wave while enabled, low after loading and SAFE) and status pins on DIO0_N–DIO3_N (3.3 V, low after loading and SAFE); timetagger reads DIO0_P–DIO7_P (pull-downs), its markers on DIO0_N–DIO7_N (2.0) and both pairs of the SATA S2 connector, and drives its test signal on both SATA S1 pairs only when enabled (high impedance otherwise, after loading and after SAFE); ajku reads a PPS input on DIO0_P (pull-down) and drives no pin; counter reads DIO0_P–DIO7_P and DIO0_N–DIO3_N (pull-downs) and drives DIO4_N–DIO7_N only when enabled (high impedance otherwise, after loading and after SAFE); spin reads DIO0_P–DIO2_P (pull-downs) and drives DIO0_N–DIO7_N only when enabled (high impedance otherwise, after loading and after SAFE; fit external pull-downs on lines that unblank an RF power amplifier). They are not 5 V tolerant; never drive an output pin from outside. The SATA daisy-chain connectors S1/S2 are 1.8 V FPGA pins without termination, not SATA ports. The other designs leave the connector unused.
- **Slow analog pins (E2).** feedback drives the slow outputs AO0–AO3 (PWM at 488 kHz, 0 … 1.8 V nominal, with about 0.15 V p-p of ripple left by the board's filter: add an external low-pass, e.g. 10 kΩ and 160 nF, before any actuator; feedback/README.md) and reads AI0–AI3 (0 … 7.0 V nominal, through the board's dividers); ajku reads AI0–AI3 the same way and drives no E2 pin. Keep the inputs inside that range. The other designs leave E2 unused (Vivado's default weak pull-down on unused pins).
- **Load impedance.** The calibration was made at DC into a high-impedance load (DMM), and DC levels are right into 1 MΩ and into 50 Ω. At 10 MHz (measured 2026-10-13, atomic_clock, 1 m coax) a set amplitude of 0.5 V gave 1.00 Vpp into 50 Ω (as set) but 2.12 Vpp into a 1 MΩ scope input: at that frequency the outputs behave like a 50 Ω source whose open-circuit voltage is twice the setting. Another Red Pitaya input in direct

loopback sees about 0.63 V for that 0.5 V setting (it loads the output). Other frequencies have not been measured: check the level at the frequency you use before connecting sensitive equipment, and terminate in 50 Ω where the level matters.

First signal in ten minutes

On a board that is already set up:

1. Power the board and wait about two minutes.
2. Open Jupyter at `http://192.168.0.149:9090` (password `xilinx`), then `mult_add/mult_add.ipynb`. Run the cells.
3. Connect a function generator to IN1 and IN2 (sines, at most 0.8 V amplitude, inputs on the LV jumper setting), and a scope to OUT1 and OUT2 (1 MΩ input).
4. In the panel's **Presets** tab, load `passthrough`: OUT1 shows IN1 and OUT2 shows IN2, delayed by a few hundred nanoseconds. Load `multiplier`: OUT1 now shows IN1·IN2/1 V. With 1 MHz on IN1 and 1.1 MHz on IN2 you see a 100 kHz beat (recipe 1).

The same from Python on your computer, through the board's control service:

```
import sys; sys.path[:0] = ["common/python", "mult_add/control"]   # paths in this repository
from rpfpga.client import connect
with connect("192.168.0.149") as dev:        # the client of whichever design is running
    dev.load_preset("multiplier")
    print(dev.describe())                    # {1: 'OUT1 = X·Y/1V', 2: 'OUT2 = Y'}
```

Or over plain TCP: `nc 192.168.0.149 5025`, then type `PRESET multiplier`.

Board specifications

"This board" means values measured on this lab's board (see common/CALIBRATION.md). The others are Red Pitaya's specifications and have not been checked here.

Quantity	Value
Channels	2 inputs, 2 outputs, SMA, DC coupled
Sample rate	125 MSa/s on all four, synchronous
Resolution	14 bit: about 138 µV per step on this board's LV inputs and outputs
Input range	LV jumper: ±1.13 V on this board (nominal ±1 V). HV jumper: ±20 V nominal (gain not calibrated here).
Input impedance	1 MΩ ‖ 10 pF
Input bandwidth	DC to 60 MHz; at 10 MHz the LV inputs read 1.06× a 0.5 V amplitude fed through a 50 Ω terminator (measured 2026-10-13, scope reference)
Output range	±1.13 V at DC on this board, into 1 MΩ and into 50 Ω. At 10 MHz the set amplitude appears into 50 Ω; into 1 MΩ it is about 2.1× (measured 2026-10-13); other frequencies not measured.
Output bandwidth, slew rate	DC to 60 MHz, 2 V per 10 ns
Loopback response (OUT → IN with 50 Ω terminators, droop corrected)	0.960 from DC to 1 kHz with a T and 50 Ω terminator at the input (the DC calibration of 2026-10-04 does not match that cabling by 4 %), a +6.8 % step at a pole-zero near 21 kHz (lockin, 2026-10-25), 1.02 from 0.1 to 2 MHz, 1.06 at 10 MHz, 1.10 at 20 MHz, 1.03–1.07 from 30 to 45 MHz, 1.00–1.05 at 50 MHz, 0.74 at 55 MHz, 0.33 at 60 MHz (measured 2026-10-22 with synth; output and input responses together)
OUT1–OUT2 skew	OUT2 0.14–0.20 ns earlier than OUT1 from 1 to 40 MHz (cable-swap method, 2026-10-22); stored in this board's calibration as a −0.165 ns delay of OUT2

Quantity	Value
Digital pins vs OUT1	an extension-connector output aligned with a sample in the FPGA changes 28.6 ns **before** OUT1 shows that sample at its SMA (the DAC and the analog output path; oscilloscope, probe/coax skew removed, 2026-10-24); stored as `pins_delay_s` = −28.6 ns: awg's markers and digital outputs then land 3.3 ns after the analog edge (the rest of the 8 ns steps; checked); a short jumper from such an output to a 1 ns input pin (`pins_loop_s`): 4.5 ns (±0.5 ns; feedback, 2026-10-26)
Spurs seen by the inputs	in loopback, a 0.4 V tone shows harmonics of −44 to −58 dBc (H2, 1–45 MHz) and two 0.2 V tones IMD3 of −72 to −77 dBc; a line at 214 kHz of about −61 dBV is present on IN1 even with no signal. These include the ADC's own distortion; the outputs alone were not measured with an analyzer.
Sampling artifacts	no anti-aliasing filter beyond the analog front end (about 60 MHz): input signals and noise above 62.5 MHz alias. The DAC holds each sample for 8 ns, so the output droops (−0.37 dB at 20 MHz, −2.4 dB at 50 MHz) and carries images at 125 MHz − f.
DC accuracy after calibration	about 0.5 mV, checked with a 6.5-digit DMM (drifts by a similar amount as the board warms)
Input reading noise	about 1 mV RMS over a 17 ms window, measured in loopback with the built-in meters (a rough figure, not a noise spectrum)
Input noise density	IN1 and IN2 alike, LV, 50 Ω terminated through a loop cable, outputs off (their idle noise included): 5.2 µV/√Hz at 37 Hz, 2.2 at 113 Hz, 1.3 at 317 Hz, 1.1–1.3 from 1 to 10 kHz, 1.6 at 31 kHz, 0.65 at 100 kHz, 84 nV/√Hz at 1 MHz, 90 nV/√Hz at 10 MHz; mains at 50 Hz about 25 µV (lockin's demodulators, 2026-10-25); from the spectral median around the reference (lockin's `noise_density`, 2026-10-27): 1.9 µV/√Hz at 100 Hz, 0.89 at 1 kHz, 1.4 at 30 kHz, 0.85 at 100 kHz, 84 nV/√Hz at 1 MHz (lockin's user's manual)
Crosstalk	OUT2 → IN1 while IN2 carries OUT2: −54 to −55 dB from 100 kHz to 10 MHz (lockin, 2026-10-25)
Converter loop delay	OUT1's DAC port → IN1's ADC port: 119.91 ns, → IN2's: 119.99 ns (cables excluded; a pure delay from 1 to 10 MHz, the analog filters' group delay rising to 129 ns at 36 MHz; lockin's phase slopes, 2026-10-25); stored in this board's calibration
Loopback stability	OUT → IN over 2 h at a die temperature of 71–73 °C: the amplitude wanders 56–59 ppm rms, the same on both outputs and both inputs (Allan deviation 10 ppm at 1 and 10 s, 20 ppm at 100 s; about −100 to −170 ppm/K); the phase 0.0014° rms at 1 MHz and 0.014° at 10.7 MHz, a delay changing by about 15 ps/K (lockin, 2026-10-25)
Memory	512 MB DDR3 (493 MB seen by Linux), 128 MB of it contiguous memory (CMA) for the FPGA: awg's 64 MB buffer of arbitrary points allocates in 0.8 s, also with only 32 MB free (page cache moves out; measured 2026-10-23)
Processing delay	analog_echo: 32 ns in the FPGA. mult_add 3.0: 456 ns, or 608 ns when X/Y is used (57/76 sampling-edge intervals; 58/77 register stages). The mult_add pipeline delay is the same with processing enabled or bypassed; LPF response time, converters and analog front ends add more. Total input-to-output delay has not been measured.

Changing settings from software is not real-time. On the board, each hardware transaction of the concurrent-control path costs about 1.3 ms (its safety check included) and a lockin reading 3.4 ms (2026-10-25); earlier software measured about 10–40 ms per call over the network and 1–6 s to switch designs. Feedback loops closed in Python

are therefore limited to roughly 10–100 Hz. Fast loops must run in the analog path, through the FPGA design itself.

What these designs cannot do

- **Full-rate PIDs only in feedback.** lockin's PIDs run at 976.5625 kHz on its demodulators' readings (loop bandwidths of a few kHz, a PLL up to about 19 kHz); feedback adds two PIDs at 125 MSa/s (about 230 ns from connector to connector, computed: loop bandwidths up to a few hundred kHz) but has only two demodulators. mult_add 3.0 has an optional first-order low-pass filter on each output (time constant about 11.54 ns to 8796 s), and a Schmitt comparator with independently selectable thresholds and levels.
- **No internal oscillator in the signal processors.** In mult_add and analog_echo a lock-in reference must come in on one of the two inputs, so I/Q demodulation with two references in quadrature is not possible there (lockin has its own oscillators and locks them to an external reference). atomic_clock generates a 10 MHz (with programmable phase and noise) and only measures the 10 MHz phase of its inputs; synth generates up to 16 tones per output and uses its inputs only for amplitude modulation, recording and measurement.
- **Meters and output filtering are separate.** The meters average over 8 µs to 1 s for software readout. Each output's optional low-pass filter processes every sample at 125 MSa/s, before the comparator and final output limits.

Multiple programs can control the board together. Configuration edits cancel pending meter/calibration acquisitions; a reload invalidates existing Python handles, which must reconnect explicitly. See the shared sessions and concurrent control reference. The MADD 3.0 build 0xcc5f666c, seven simulation benches and build-workflow verification passed on 2026-10-05; the processing and protected-clamp features have **not yet been deployed or tested on a physical board**.

Documentation

Using the designs:

- mult_add/README.md, then its recipes (worked lab examples) and control reference;
- analog_echo/README.md;
- atomic_clock/README.md;
- synth/README.md, then its recipes and control reference;
- awg/README.md, then its recipes and control reference;
- lockin/README.md, then its recipes and control reference;
- feedback/README.md, then its recipes and control reference;
- common/CALIBRATION.md: calibrating the board, input jumper settings, accuracy;
- common/README.md: the TCP commands every design understands;
- guides/combining/combining.pdf: several designs on two or three boards as one experiment (which pins connect, time across boards, latencies, worked combinations, how to test a combination).

Setting up, changing and adding designs:

- common/SETUP.md: board and build-machine setup, a ready SD card image;
- common/README.md: the shared framework (build workflow, safe loading, service, Python package, "Adding a new design");
- each design's `docs/THEORY.md` and `docs/REGISTERS.md` (how it works inside) and `docs/VERIFICATION.md` (what was tested, when);
- AGENTS.md: rules for maintainers and coding agents.
- site/README.md: the public project page (features, manuals as PDFs, the SD image download) and how to assemble it for upload.

`LED_blink/` is a minimal stand-alone example of the build-and-upload workflow, outside the framework.

Acknowledgements

This project stands on two open-source projects by Pau Gómez (dspsandbox), and we are very grateful for them:

- **FPGA Notes for Scientists**: a wonderful hands-on introduction to FPGA programming for scientists with little or no FPGA experience, with simple, working examples on the Red Pitaya and PYNQ that each add one new concept. This project grew out of its LED blink and Analog echo tutorials; analog_echo still follows the Analog echo tutorial, and every design uses its converter interface: the ADC, DAC and clock IP blocks and the board constraints, included under its MIT license (common/NOTICE.md).

- **PYNQ for the Red Pitaya STEMlab 125-14**: a careful port of PYNQ to the Red Pitaya, with a ready-to-use SD card image, the Vivado board files and a documented build process. The designs run on its PYNQ 3.0.1 image (the ready SD card image adds them to it) and are built with its board files.

Author

Rok Zitko, Jozef Stefan Institute (Ljubljana, Slovenia), rok.zitko@ijs.si.

Setting up a board and a build machine

You need this once per Red Pitaya, and once per computer that builds designs. Using a board that is already set up needs none of it: see the root README for that.

A ready SD card (quickest)

`common/scripts/make_sd_image.sh` writes an SD card image with the steps of the next section done (step 2 with `--ssh-key`), except the calibration: from a fresh card the board starts with all designs installed. It runs on a Linux computer with this repository and the designs' built overlay files (`<design>/overlay/`, from `make` in each design; or pass `--build`, which needs Vivado), and needs no board:

```
common/scripts/make_sd_image.sh --active mult_add
```

It downloads the PYNQ 3.0.1 image once (7.8 GiB, checked by its SHA-256, kept in `~/.cache/rpfpga-sdimage`), copies it to `rpfpga-sd-<date>-<commit>.img` in the current directory, asks for your sudo password to mount that copy, installs the list below and shrinks the copy to 7.0 GB, so that it fits every card sold as 8 GB (`--size` sets another size; the board grows its file system to the whole card at the first boot):

- every design's overlay files, plugin package and notebook, and the `rpfpga` package in PYNQ's Python, as `make upload` would (`--designs "A B"` selects designs; a prototype, version below 1.0, is installed only when named there);
- the control service, enabled: about two minutes after power-up it loads the design given by `--active` (default: `IMAGE_ACTIVE` in the repository-root `config.mk`, analog_echo, a unity passthrough);
- the passwordless sudo rule for PYNQ's Python (step 3 below);
- Ethernet with **DHCP and a fixed address together**: the fixed address (`--static-ip`, default `192.168.2.99/24`, PYNQ's own; `none` for DHCP only) is up at once, while DHCP keeps trying in the background, so the board also takes an address from a DHCP server that answers later (a cable plugged in after power-up, a router started later). The plain PYNQ image has both too, but at boot waits about 11 s for a DHCP server before the fixed address comes up. (Tested 2026-10-27 in an Ubuntu 22.04 container with the image's versions of ifupdown and the DHCP client and PYNQ's client settings: fixed address after 0.1 s; a DHCP server started 15 s later gave an address within 6 s; lease renewals kept it, and the fixed address stayed. Not yet on a board.);
- the scheduled apt/man-db jobs off (step 6; `--keep-apt-timers` keeps them);
- optionally your SSH public key for `xilinx` (`--ssh-key`, step 2; only on a card for yourself, never in an image you give to others), a host name (`--hostname`; the board sends it to the DHCP server) and this board's saved calibration (`--calibration`, a copy of `/home/xilinx/rpfpga/calibration.json`; only for the board it was measured on);
- `/home/xilinx/rpfpga-image.txt`, which records what the card holds (commit, design versions and build IDs, settings).

`--help` lists all options; `--compress` also writes a `.zip` for downloads. Write the image to a card of **8 GB or more** with balenaEtcher, or find the card with `lsblk` (the whole card, e.g. `/dev/sdX`, not a partition) and run `sudo dd if=rpfpga-sd-<...>.img of=/dev/sdX bs=4M conv=fsync status=progress`. On first power-up PYNQ enlarges its file system to the whole card. Log in with `ssh xilinx@<address>` (password `xilinx`; add your key with `ssh-copy-id` unless `--ssh-key` did), then calibrate (step 7). To install into a card already written with the PYNQ image instead, mount its root partition and run the script with `sudo` and `--root <mount point>`.

Needs: Bash, curl, sudo, util-linux (`losetup`, `sfdisk`), e2fsprogs, Python 3 and about 17 GB of free disk (the download and the image).

The board (once)

1. **PYNQ image.** The board runs the PYNQ 3.0.1 image for the Red Pitaya STEMlab 125-14 from dspsandbox/Pynq-Redpitaya-125 (its README links the ready-made `Pynq-Redpitaya-125-14-3.0.1.img`; write it to the SD card with `dd` or balenaEtcher) and is reachable over the network: it takes an address by DHCP and also answers at the fixed `192.168.2.99`. This lab's board is configured in the repository-root `config.mk`, currently `192.168.0.102` (from the lab router's DHCP). Change `BOARD_HOST` there once: every design's Make workflow inherits it. A design-local `config.mk` or an explicit command-line assignment can override it. The manual SSH examples below use the current address; Python clients and host tools take an explicit host.

2. **SSH keys.** From each computer that uploads designs, install your key and accept the board's host key once:

   ```
   ssh-copy-id xilinx@192.168.0.102
   ssh -o BatchMode=yes xilinx@192.168.0.102 true      # must succeed without a prompt
   ```

   The workflow uses `BatchMode=yes`, so it never stops to ask for passwords.

3. **Passwordless sudo for the PYNQ Python.** Loading an FPGA design needs root. On the board:

   ```
   ssh -t xilinx@192.168.0.102
   sudo visudo -f /etc/sudoers.d/rpfpga
   ```

   Add this line, then save:

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

   Then run `sudo chmod 0440 /etc/sudoers.d/rpfpga && sudo visudo -c`. sudo uses the last matching rule: if `sudo -l` lists an `(ALL : ALL) ALL` rule after this one, move the line to the very end of `/etc/sudoers` (`sudo visudo`). The rule lets the `xilinx` user run that Python as root. Nothing in the workflow changes sudoers itself.

4. **Check** from the computer:

   ```
   ssh -o BatchMode=yes xilinx@192.168.0.102 \
     'sudo -n /usr/local/share/pynq-venv/bin/python -c "import os, pynq; print(os.geteuid(),
     ↪  pynq.__version__)"'
   ```

   Expect `0 3.0.1`.

5. **Install the designs and the service** from a computer with this repository:

   ```
   make -C analog_echo upload
   make -C mult_add upload install-service     # the TCP service, with mult_add active
   ```

   `make upload` needs the built design files in `<design>/overlay/`; if they are missing, it builds them, which needs Vivado (below). Upload only copies/installs files; a later driver attachment may program a changed build. `install-service` restarts the service and can load/restore the design. After `install-service`, the board loads the last used design about two minutes after every power-up (equipment safety).

6. **Stop the scheduled maintenance jobs** (recommended). Ubuntu on the board runs `apt-daily` (twice a day), `apt-daily-upgrade` and `man-db` (daily); each loads the CPU and the SD card for a minute or more. That can stall the control service for seconds, which breaks continuous work such as atomic_clock's phase-noise playback and long recordings. On the board:

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

   Update by hand when convenient (`sudo apt update && sudo apt upgrade`), or enable the timers again with `sudo systemctl enable --now <timer>`.

7. **Calibrate** with the loopback cables and a DMM (CALIBRATION.md). Until then, all voltages use nominal converter constants, which were about 13% off on this lab's board.

A build machine (for changing or rebuilding designs)

- Linux with Bash, GNU Make, `flock` (util-linux), OpenSSH and Python 3.
- **Vivado 2025.1** with Zynq-7000 support (`xc7z010clg400-1`); no GUI or X server is needed.
- The Red Pitaya board definition `redpitaya.com:redpitaya-125-14:part0:1.0` (`Vivado/board_files/redpitaya-125-14` in dspsandbox/Pynq-Redpitaya-125), installed in Vivado's `data/boards/board_files/`, or point `BOARD_REPO_PATH` at a directory containing it.

- Before `make`, load Vivado into the shell: `source <Vivado install>/2025.1/Vivado/settings64.sh`, or pass `VIVADO=<path>/bin/vivado` (also settable in `<design>/config.mk`).

Then in a design directory, `make` builds, `make test-remote` uploads and checks the design on the board, and `make help` lists everything (README.md).

Board calibration

The converters and input jumpers belong to the board, not to a design. One calibration, `rpfpga.calibration`, therefore serves every design: measure it once with a design that can read its inputs, and every design uses it. User values are then volts at the connectors. Today mult_add, synth and awg can measure it, and every design applies the calibration.

What calibration corrects

Each converter has a gain and an offset error. The gain error can be large: on the board used here (2026-10-04) the DACs put out 13.5% and 14.3% more than nominal and the LV inputs read 11.3% and 12.0% low. The offsets were up to about 15 mV per converter: ADC −15.3 mV and −7.3 mV, DAC −7.5 mV and +1.7 mV. In loopback the two gain errors nearly cancel (+0.5–0.6%), so only a DMM reference (step 2) reveals them. The calibrated ranges are what the hardware gives: about ±1.13 V for the LV inputs and for the outputs on this board.

The calibration is a DC calibration. Checked with a scope at 10 MHz (atomic_clock, 2026-10-13, 1 m coax): a set amplitude of 0.5 V gave 1.00 Vpp into a 50 Ω feed-through, as set, but 2.12 Vpp into the scope's 1 MΩ input, while DC levels (±0.5 V) were right both ways. At 10 MHz the outputs behave like a 50 Ω source whose open-circuit voltage is twice the setting. The LV input read 1.06× that 0.5 V amplitude through a 50 Ω terminator; in direct loopback (OUT → IN, no terminator) it reads about 0.67 V, since the input loads the output. Other frequencies have not been measured.

Every design streams the converters as signed 16-bit codes, 32768 codes per nominal volt on LV, and models them as:

```
ADC:  code  = K · V_in + A        K ≈ 32768/V (LV) or 1638.4/V (HV), A ≈ 0
DAC:  V_out = G · code + B        G ≈ 1/32768 V, B ≈ 0
```

Each design folds K, A, G and B into its register values. mult_add folds them into input offsets, weights, output offsets, clamp limits, mute codes, and the Schmitt comparator's virtual thresholds and physical DAC14 levels; analog_echo into its gain and offset codes.

Files, in `/home/xilinx/rpfpga/`:

File	Contents
`calibration.json`	K and A per input and jumper setting (`lv`, `hv`), G and B per output, the method and dates. Nominal values (`"method": "nominal"`) until measured.
`board.json`	`{"ranges": {"in1": "lv", "in2": "hv"}}`: how the input jumpers are set. LV if absent.

Input ranges

Set the jumpers on the board, then tell the software, from any design:

```
dev.set_input_range("in1", "hv")      # TCP: RANGE IN1 HV    CLI: --range in1=hv
dev.ranges()                           # TCP: RANGE?
```

The active design re-applies its configuration for the new range. Settings stay in volts. A configuration that cannot be realized on the new range is rejected before hardware writes. Runtime/transport errors do not promise hardware rollback.

Procedure

> **The outputs change during steps 2 and 3:** disconnect equipment that must not see these voltages. Step 2 holds both outputs at the reference code, about +0.56 V on this board. Step 3 sweeps them

between about −0.83 V and +0.83 V (the nominal values are 0.5 V and ±0.73 V; the real ones are larger by the DAC gain error).

Use a design whose `status()["calibrates"]` is true (mult_add, synth, awg). Run the steps from Python (driver or client, here `dev`), TCP or that design's GUI. Each step is a separate action. Offset/loopback measurements restore or re-apply the user configuration on successful completion; **the reference hold remains active until `release_reference()` / `CAL RELEASE` or a superseding edit**. Earlier measurements took roughly a second for a measurement step; the current path has not been timed. Let the board warm up for about 10 minutes first.

Step 1: input offsets (optional, recommended). Disconnect IN1 and IN2, then short them or fit 50 Ω terminators. Don't leave them open.

```
dev.measure_input_offsets()        # TCP: CAL OFFSETS
```

This stores A for each input at its current jumper setting. If you skip it, the ADC offsets count as zero and step 3 attributes the whole loop offset to the DAC. The loop OUT→IN is still exact, but absolute input readings keep the ADC offset (−444.6 and −211.8 codes on the board used here, i.e. −15.3 mV and −7.3 mV with the calibrated gains, −13.6 mV and −6.5 mV with the nominal gain).

Step 2: absolute reference (optional, recommended). Connect a DMM to OUT1 and OUT2 in turn.

```
dev.hold_reference()               # TCP: CAL HOLD — both outputs at code 16384 (≈ +0.56 V here)
# read the DMM on OUT1 and OUT2
dev.release_reference()            # TCP: CAL RELEASE
```

Without a new reference, loopback **retains each input's stored ADC gain K** and calibrates the DAC relative to it. K is nominal only if it has not previously been calibrated (or was reset). Starting from nominal K leaves the inputs mismatched by their true gain difference (0.75% on the board used here); an existing DMM-based gain is not discarded when a later loopback omits `dmm`. See Accuracy to expect. The DMM readings must lie between 0.1 V and 2 V.

Step 3: loopback. Connect OUT1 → IN1 and OUT2 → IN2 with short coax cables, inputs on LV, nothing else on those connectors.

```
report = dev.calibrate()                                    # TCP: CAL LOOPBACK
report = dev.calibrate(dmm={"out1": 0.5601, "out2": 0.57299})    # with step 2's readings
```

The DAC steps through 9 levels from code −24000 to +24000 (about −0.83 V to +0.83 V on this board). At each level the input is averaged over about 17 ms (2^21 samples). A straight-line fit gives IN code = α·DAC code + β per channel. With the model above, α = K·G and β = K·B + A. Given A from step 1 and K (stored, or newly from the DMM: K = (α·16384 + β − A)/V_dmm), the fit solves for G and B.

The report lists, per output:

- the gain error in percent;
- the offset in millivolts;
- the largest fit residual in millivolts: a measure of DAC/ADC linearity and noise, normally < 0.7 mV.

A channel with no response (α ≈ 0) raises an error, usually a missing cable. If the active configuration cannot be realized with the new constants (for example an offset that no longer fits), the new calibration is not stored and the step raises an error ("The new calibration was not stored …"): change the configuration, then repeat.

HV range: run step 1 with the jumpers on HV to calibrate HV offsets. The HV gain uses the nominal 20 V full scale unless you calibrate with a DMM while the jumpers are on HV. In that case the loopback signal uses only 1/20 of the range, so expect about 20× more noise in the HV gain.

`dev.reset_calibration()` (TCP `CAL RESET`) returns to nominal values. `CAL?`, `CAL RESET`, `CAL RESPONSE`, `CAL DELAY`, `CAL PINS`, `RANGE?` and `RANGE` also work with no design loaded. The measuring steps return an error on a design that cannot read its inputs.

Protected output windows

In mult_add 3.0 (protected clamps) and awg (protected limits), protection fixes a digital DAC window and its calibrated voltage interpretation until FPGA bitstream reload:

- Automatic loopback calibration rejects if **either output is protected**, even if a proposed sweep would fit inside its window.

- Reference hold and raw `hold_dac_codes()` requests outside a protected window reject before writes to either output. An allowed hold still passes through the final clamp; unprotected clamps may be temporarily bypassed for calibration.
- DAC gain/offset changes for a protected output are refused. This includes `CAL RESET` if nominal constants would change that output's calibration. Input-offset measurement may proceed because it does not change DAC constants, provided the resulting configuration remains representable.

`CAL RELEASE`, `SAFE`, presets and reconnect do not release protection. Only a deliberate FPGA reload does; it starts an uncalibrated passthrough and invalidates existing handles. Protection bounds the digital stream, with the physical limits described in the root equipment section.

Concurrent edits and processing history

Meter and calibration waits release the shared board transaction lock. Another meter/calibration acquisition receives `Busy`; configuration edits remain available and cancel the acquisition. There is **no reservation** of the board. If another program changes settings or reloads during calibration, the cancelled calibration must not restore its old configuration over that newer state. Cancellation/session failures use `Cancelled`/`Stale` (TCP `CANCELLED`/`STALE`); see sessions and concurrent control.

The reference hold is also not a reservation: an accepted edit supersedes it and merges into the user's configuration, not the temporary raw DAC settings. For designs with stateful output processing, raw calibration levels bypass that processing. Successful measurement completion or explicit reference release restores the user configuration and resets processing from the live signal. Ordinary configuration edits and presets preserve running processing history.

Factory calibration EEPROM

The board's calibration EEPROM is readable on the PYNQ image (`/sys/bus/i2c/devices/0-0050/eeprom`, a 24c64). It holds Red Pitaya's calibration as ID/value records, and the factory copy at 0x1C00 matches the user copy at 0x0000:

- IDs 1–4: LV ADC gain/offset;
- IDs 9–12: HV ADC gain/offset;
- IDs 17–20: DAC gain/offset;
- IDs 33+: input filter coefficients.

The gain encoding is not documented here and was not verified, so the software does not use these values. The loopback calibration plus a DMM reference is the supported method.

Output frequency response and delay (optional)

The constants above are measured at DC. Two optional per-output fields extend the calibration to AC signals, for designs that make them (synth, awg). Both are stored in the same board file and shared by every design:

- **`response`**: the output's measured amplitude gain versus frequency, a table `[[Hz, gain], ...]` with increasing frequencies. Gain is the measured amplitude divided by the set amplitude, after the design's own corrections (for synth, after its sinc droop correction). Between points it is interpolated linearly; beyond the ends the end values hold. A design divides its amplitudes by this gain, so a set 0.5 V comes out as 0.5 V at every tabulated frequency. Without a table the gain is 1.
- **`delay_s`**: the output's delay in seconds, relative to the other output (within ±1 µs). A design that sets phases corrects them by +360°·f·delay, so that a tone's phase is referred to the connector.

Measure them as you would a cable: the response with a calibrated power meter, a spectrum analyzer or a scope into 50 Ω (or with the design's own loopback measurement, which then includes the input's response); the delay by a cable swap (synth's `measure.skew`, `../synth/RECIPES.md`). Then store them:

```
dev.set_output_response(1, [[1e6, 1.000], [10e6, 0.987], [40e6, 0.93]])   # [] removes it
dev.set_output_delay(2, 0.6e-9)
```

TCP: `CAL RESPONSE OUT1 [[1e6, 1.0], [1e7, 0.987]]` and `CAL DELAY OUT2 6e-10`; each replies with the stored calibration. Both also work with no design loaded. Storing re-applies the configuration of the active design. A loopback calibration keeps them; `CAL RESET` removes them. `CAL?` and the calibration summary show them.

Input frequency response and loop delay (optional)

Two optional fields per input **and jumper setting** (`in1.lv`, `in1.hv`, ...) serve designs that measure AC signals and their phases (lockin):

- **`response`**: the input's measured amplitude gain versus frequency, `[[Hz, gain], ...]` with increasing frequencies, gain = the amplitude read divided by the actual amplitude at the connector (interpolated as the output tables). A design divides its readings by it.
- **`delay_s`**: the **converter loop delay** in seconds (0 … 1 µs): the time from a word on OUT1's DAC port, through the connectors joined directly, to the same value on this input's ADC port. A design adds its own digital latencies (known exactly) to refer a measured phase to the OUT1 connector; the difference IN1 − IN2 is the inputs' skew.

Both are measurable in loopback. The delay: cable OUT1 → IN1 (and OUT2 → IN2, whose calibrated skew against OUT1 is taken into account) with cables of known delay, and fit the phase slope versus frequency (lockin: `analysis.measure_input_delay(dev, "in1", cable_delay_s=..., store=True)`; on the board used here 122 ns OUT1 → IN1 including a short cable). The response: against a calibrated source (a loopback measures output and input together). Then store them:

```
dev.set_input_response("in1", [[1e3, 0.96], [1e5, 1.02], [1e7, 1.056]])   # [] removes it
dev.set_input_delay("in1", 121.5e-9)
```

TCP: `CAL RESPONSE IN1 [[1e3, 0.96], [1e5, 1.02]]` and `CAL DELAY IN2 1.2e-7`, at the input's current jumper setting; each replies with the stored calibration and also works with no design loaded. Storing re-applies the active design's configuration. A loopback calibration keeps them; `CAL RESET` removes them. Designs use `adc_response(cal, ranges, ch, f)` and `adc_delay(cal, ranges, ch)`.

Delay of the digital pins (optional)

Designs that align extension-connector outputs with the analog outputs (awg's markers and digital outputs) do so inside the FPGA, to the sample. At the connectors the analog path (DAC, output filter, amplifier) and the digital path (output buffers, traces) add different delays. **`pins_delay_s`** (board-wide, within ±100 ns; 0 until measured) is the time a pin changes after OUT1 shows the sample it is aligned with; such designs subtract it, in whole samples (8 ns), from the delays you set.

Measure it with an oscilloscope (OUT1 into 50 Ω at the scope, a 3.3 V probe on the pin), for example with awg: a single-cycle square burst on OUT1 (`rise` 0, phase 0: its rising edge's 50 % point is on the start sample) every millisecond from timer 1, and marker 1 (DIO1_P) on OUT1's start pulse with delay 0:

```
dev.apply({"outputs": {"1": {"function": "square", "frequency": 1e6, "amplitude": 0.4,
                             "burst": {"mode": "triggered", "cycles": 1}, "trigger": {"source":
                              ↪  "timer1"}}},
           "timers": {"1": {"period": 1e-3, "on": True}},
           "markers": {"1": {"source": "start", "channel": 1, "delay": 0.0, "width": 1e-6}}})
```

Trigger the scope on OUT1's rising edge and read the time from OUT1's 50 % point to the pin's 1.65 V crossing: that is `pins_delay_s` (positive when the pin is later). Store it with `dev.set_pins_delay(seconds)` or TCP `CAL PINS <seconds>` (also with no design loaded); the active design re-applies its configuration. A loopback calibration keeps it; `CAL RESET` removes it.

Designs that also time **input** edges to 1 ns (feedback's TTL reference and trigger, awg's triggers) need a second board quantity, **`pins_loop_s`**: the delay from an output pin to a 1 ns input pin through a short jumper (the output's and the input's buffers and traces, against the designs' simulated timing). With `pins_delay_s` it refers the input timestamps to OUT1: an input pin's edge reaches the samplers `pins_loop_s − pins_delay_s` later, relative to OUT1, than an output pin's leaves. feedback measures it with a jumper DIO0_N → DIO3_P (its status pin's frame pulse, timed by the trigger's samplers): `analysis.measure_pins_loop(dev, store=True)` or TCP `PINSLOOP? STORE` with feedback loaded, or `CAL PINSLOOP <seconds>` to store a value (also with no design loaded). This board: 4.5 ns (±0.5 ns: the samplers' resolution; measured 2026-10-26).

Slow analog outputs and inputs (optional)

The E2 connector's slow outputs AO0–AO3 (a PWM pin and an RC filter on the board) and inputs AI0–AI3 (the Zynq's XADC behind a divider) have their own entries, `slow.ao0` … `slow.ai3`: `volts_per_unit` and `offset_volts` with V = volts_per_unit · value + offset_volts, where the value is the 16-bit code (0 … 65535: an output's duty, an input's left-aligned 12-bit result). Until measured they hold the nominal full scales, 1.8 V for the outputs and 7.0 V for the

inputs (Red Pitaya's specification; AO → AI on this board reads slopes of 0.98–1.00 at the nominal scales), offset 0, and are marked `measured: false` (the calibration summary says so). Designs that use the pins (feedback) read them with `rpfpga.calibration.slow(cal, "ao", k)` → (G, B, measured).

Measure an output with a DMM at two values (e.g. 0.25 and 0.75 of full scale, through the design's own settings at their nominal scale), fit G and B, and store them; with a jumper from that output to an input, read the input at the same two levels and store its constants likewise:

```
dev.set_slow_calibration("ao", 0, 1.795 / 65536, 0.004)    # TCP: CAL SLOW AO0 2.739e-05 0.004
```

`CAL SLOW` also works with no design loaded; the active design re-applies its configuration. A loopback calibration keeps the entries; `CAL RESET` returns them to nominal.

Accuracy to expect

After steps 1 and 3, a value that goes out through OUT and back in through IN is accurate to roughly the meter noise (well under 1 mV over 17 ms).

Absolute accuracy:

- With a DMM reference: limited by the DMM and by drift since the reading.
- Without a new one: limited by the accuracy of the stored ADC gains. If they are still nominal, the true ADC gain error was about 12% on this board.

Match between IN1 and IN2: starting from nominal K without step 2 leaves their true gain difference (0.75% on the board used here). In mult_add, X − Y with IN1 = IN2 = 0.5 V then reads +3.7 mV (a common-mode rejection of about 43 dB) and X + Y reads 0.996 V instead of 1.000 V; with the DMM step, X − Y is within 0.08 mV of 0. Anything that compares IN1 with IN2 (a difference amplifier, ratios between the inputs, the min/max crossover, a test level against IN1) therefore needs step 2. The calibration is at DC: it does not cover the frequency responses of the two front ends, so AC common-mode rejection is not calibrated.

The ADC/DAC resolution is 14 bits: about 138 µV steps on LV on this board (122 µV nominal). The DAC ignores the two lowest bits of the 16-bit code (truncation), so a static output carries 0 to −3 codes of uncorrected bias (about −52 µV on average, at most −104 µV on this board): small next to the other errors. Drift with temperature is not compensated: on the board used here, two loopback calibrations 13 minutes apart differed by about 0.05% in loop gain and 0.5 mV in offset. Recalibrate after warm-up, and whenever you change jumpers or cables.

A design's own quantization adds to this. For example, analog_echo's gain has steps of 1/256 of G·K, about 0.4%: on the board used here, unity gain is realized as 0.9986. The values a design reports back are always the realized ones.

Checked on 2026-10-04: with this calibration, analog_echo outputs set to +0.5 V and −0.3 V read +0.50046 V and −0.30030 V on an independent 6.5-digit DMM.

For design authors

- **Using the calibration (every design):** compile volts to codes with `rpfpga.calibration.adc(cal, ranges, "in1")` → (K, A) and `dac(cal, 1)` → (G, B), from `self.calibration()` and `self.ranges()`. `derive()` applies the inverse. `Overlay` re-applies the configuration whenever the calibration or a range changes. Designs that make AC signals also use `dac_response(cal, n, f)` (divide amplitudes by it) and `dac_delay(cal, n)` (add 360°·f·delay to phases); designs that measure them use `adc_response(cal, ranges, ch, f)` (divide readings by it) and `adc_delay(cal, ranges, ch)` (the converter loop delay).
- **Measuring (optional):** implement the two hooks of `rpfpga.Overlay`.
  - `adc_codes(window_s)` returns the mean raw ADC codes `{"in1", "in2"}` from a window that starts after the call.
  - `hold_dac_codes({1: code, 2: code})` drives raw DAC codes, bypassing ordinary processing settings while respecting any protected final bounds. Validate all channels before writing any of them.

  `can_calibrate()` then reports true, and the procedure above works through the design, unchanged.
- **Digital test signals** generated inside a design should use the codes per volt of a calibrated input, so that they compare directly with real inputs in real volts (mult_add's test level Tn uses INn's LV calibration).
- **GUI and CLI:** `rpfpga.gui.CalibrationPanel` and `RangeSelector`, and the common CLI's `--calibration` and `--range`, show and set all of this for any design.

Common infrastructure for Red Pitaya designs

This page is for people who build, change or add designs. To use the designs, start at the root README; to set up a board, see SETUP.md.

Everything the designs in this repository share lives here:

- the build workflow (Make, Vivado Tcl, SSH upload and remote tests);
- the common block design and system-ID hardware;
- the Python package `rpfpga`: safe loading, the uniform driver API, one TCP service, one command line, a client and test fakes;
- the pinned converter IP and the board constraints.

A design contributes only what makes it different: its RTL, a `design.tcl` describing its block-design cells, a plugin package (driver, protocol commands, client, GUI) and its documentation.

Design	What it does
`analog_echo`	per-channel gain and offset
`mult_add`	two-input analog computer: weighted sums of 13 terms, optional full-rate output low-pass filters, Schmitt comparators and final protectable clamps
`atomic_clock`	exact 10 MHz on both outputs; OUT1 with programmable phase, frequency offset and phase noise (hardware white noise, played-back slower noise); 1 PPS outputs; a 10 MHz phase meter on the inputs
`synth`	16 phase-coherent tones per output, ramps and timed sequences, amplitude modulation by an input, input capture and tone measurement
`awg`	two-channel function and arbitrary waveform generator: shapes, noise, modulation, sweeps, bursts and triggers, sequenced waveforms from DDR, eight digital outputs
`lockin`	four-demodulator lock-in amplifier: four oscillators, X/Y/R/θ, frame-rate PIDs and PLLs, a data stream, capture, sweeps and fits
`feedback`	lock box: lockin's core with two demodulators, two full-rate PIDs, scan-and-relock acquisition, slow analog outputs and inputs (E2), status pins, a TTL reference (1 ns) and sync output, plant-based tuning

`LED_blink` is a stand-alone minimal example outside the framework.

Layout

```
common/
  mk/design.mk            included by every design's Makefile
  scripts/               workflow.sh build.tcl project.tcl create_project.tcl sim.tcl usr_access.tcl
                          upload.sh remote.sh remote_smoke.py install_service.{sh,py} clean.sh
                          make_sd_image.sh (a ready SD card image with every design) ...
  rtl/rp_sysid.v          system-ID block, present in every design
  rtl/optional/           shared RTL built only by designs that list it (rp_xadc.v: AI0..AI3)
  constraints/            board constraints (ADC, DAC, clocks)
  vendor/                 pinned ADC/DAC/clock IP (see NOTICE.md)
  service/                rp-overlay.service (systemd unit template)
  python/rpfpga/          the shared Python package (board and PC)
  tools/stabilitylib/     host-only frequency-stability analysis, shared by the designs
  CALIBRATION.md          the board calibration and input ranges, shared by every design
  SETUP.md                one-time board and build-machine setup
  tests/                  framework tests, tb_sysid.sv, verify_build.py
<design>/
  Makefile                DESIGN := <name>; include ../common/mk/design.mk
  design.tcl              name, ID, version, RTL, testbenches, design_bd proc (design_export)
  src/                    design RTL
  control/<design>/       plugin package; control/<design>.ipynb is the notebook
  scripts/test_hw.py      optional design hardware test (make test-hw)
  tests/                  testbenches and Python tests
  README.md AGENTS.md docs/
```

Make targets and variables

The same targets and variables work in every design directory (`make help`):

Target	Action
`make`	create the project from the recipe if absent, validate, build, export `overlay/<design>/<design>.{bit,hwh,tcl,json}`
`make create`	create the project only
`make upload`	build, then copy the overlay files and plugin package to the board, the notebook if it is not already there (`NOTEBOOK_OVERWRITE=1`: replace it, keeping the old one as `<design>.ipynb.bak-<time>`), and install `rpfpga` in the PYNQ virtualenv
`make test-remote`	preflight, build, upload, program through the design driver, verify sysid/build, check the power-up state, apply `PRESET`
`make test-hw`	the design's own hardware test (`scripts/test_hw.py`), if it has one; no build
`make install-service`	build, upload, install `rp-overlay.service` with this design active
`make sim`	XSim: `tb_sysid` and the design's testbenches
`make test`	the design's offline Python tests
`make rebuild`, `make clean`, `make tcl`	force a rebuild, remove generated products, open a Vivado Tcl console

At the repository root, `make test` runs the framework tests and every design's tests, `make sim` runs every design's testbenches, and `make upload` builds (if needed) and uploads every design, one after another (alphabetically). Variables pass through, e.g. `make upload NOTEBOOK_OVERWRITE=1; DESIGNS="mult_add atomic_clock"` limits any of these to some designs.

`make upload` installs files without programming the FPGA. A later driver attachment can download a newly uploaded build after detecting its changed identity. `make test-remote` explicitly programs and tests; installing/restarting the service can load the selected design and restore its saved settings.

Set the board address once in the repository-root `config.mk`. Every design reads that file, including `LED_blink`, whether invoked through the root Makefile or directly with `make -C <design> ...`. The precedence is:

1. Command-line assignments, e.g. `make upload BOARD_HOST=another-board` (highest).
2. `<design>/config.mk`, for explicit design-local overrides.
3. Repository-root `config.mk`, for shared board/tool settings.
4. The common defaults below.

The design examples leave `BOARD_HOST` commented out so copying one does not silently replace the shared address. Keep design-specific presets in the design directory. Python clients and host tools accept an explicit host; use the current address from `config.mk` for those calls.

Variable	Default	Used by
`VIVADO`, `JOBS`, `BOARD_REPO_PATH`	`vivado`, 3, empty	build
`BOARD_HOST`, `BOARD_USER`, `BOARD_PORT`	root `config.mk` (`redpitaya.local` if absent), `xilinx`, 22	remote targets
`REMOTE_OVERLAY_ROOT`	`/home/xilinx/pynq/overlays`	upload
`REMOTE_NOTEBOOK_DIR`	`/home/xilinx/jupyter_notebooks/<design>`	upload
`REMOTE_PYTHON`, `REMOTE_XRT`	`/usr/local/share/pynq-venv/bin/python`, `/usr`	remote targets; `install-service` (the unit's `XILINX_XRT`)
`SSH_OPTIONS`, `SCP_OPTIONS`	`-o ConnectTimeout=10`, empty	remote targets
`PRESET`	`passthrough` (atomic_clock `reference`; synth, awg, lockin and feedback `off`)	`test-remote`
`HW_CALIBRATE`	`0`	`test-hw` (designs that calibrate)
`NOTEBOOK_OVERWRITE`	`0`	upload (also via test-remote, install-service): `1` replaces the board's notebook, backing up the old one
`SERVER_BIND`, `SERVER_PORT`	`0.0.0.0`, 5025	`install-service`

Build safeguards:

- `design.tcl` is the source of truth: on every build `rp::sync_project` makes an existing project follow it

(adds or removes RTL files, updates the sysid design ID and version), as a newly created project would.
- A lock on the design directory covers every workflow.
- SHA-256 snapshots of all inputs decide what to rebuild.
- A success receipt (`.build/export.ok`) is required before anything is uploaded.

Board setup (once): key-based SSH, plus a passwordless sudo rule for the PYNQ interpreter, as described in SETUP.md. Every remote target uses that one rule. `scripts/make_sd_image.sh` builds an SD card image with that rule, every design and the service installed (SETUP.md).

Optional common RTL

Every `common/rtl/*.v` is part of every design (and of its build ID). Shared blocks that only some designs use live in `common/rtl/optional/`, outside that glob, and a design lists them in `design(rtl)` by an absolute, normalized path, so the project's file list stays stable and no other design's sources change:

```
set design(rtl) [list src/*.v [file normalize [file join [file dirname [file normalize [info
 ↪  script]]] \
    .. common rtl optional rp_xadc.v]]]
```

- `rp_xadc.v`: the slow analog inputs AI0..AI3 of E2 through the XADC in continuous sequence (ADCCLK 25 MHz; the sequence keeps what Linux reads through iio). `ai` holds the four 16-bit words (12-bit result in [15:4]); `strobe` marks each new set. The UNISIM model reads `SIM_MONITOR_FILE` (a `.mem` stimulus in the design's `design(sim_files)`). Used by ajku; feedback keeps its earlier copy (`fb_xadc.v`) until its next rebuild. Convert words with `rpfpga.calibration.slow(cal, "ai", k)`.
- `rp_pins1ns.v`: N extension-connector pins sampled every 1 ns (a PLL at 250 MHz, 0° and 90°, and an ISERDESE2 per pin in oversample mode); `w8[8 p +: 8]` is pin p's 8-bit word of one sample period at the sample clock, bit 0 the earliest, in clock n + 4 for period n. `SIM > 0` is a behavioural sampler for benches. The front end of awg's `aw_trig`, feedback's `fb_pins` and timetagger's `tt_mark` (their copies stay until their next rebuild).
- `rp_oserdes2ns.v`: one output pin with 2 ns bits (OSERDESE2 DDR on the converters' 250 MHz clock) and a tristate through the serializer's T path, into an OBUFT (feedback's `fb_sync` serializer).
- `rp_ring_writer.v`: the AXI3 writer of a ring of 32-bit records in DDR through S_AXI_HP0, with the mailbox, the room check and the watchdog (timetagger's `tt_writer`, with `MAGIC` and `NOP` as parameters); the service side is `rpfpga.ringworker` ("Record rings").
- `rp_sine.v`, `rp_sine_rom.v`: a 24-bit phase to a signed 18-bit sine in 6 clocks (a quarter-wave table of 1024 {value, slope} entries and a first-order Taylor term in one DSP48; one ROM's two ports serve two `rp_sine`). The ROM is generated by `rpfpga.sine` (`python3 python/rpfpga/sine.py`), which is also the bit-exact reference (`sine(p)`); `tests/test_sine.py` checks that the file is current and equals synth's, lockin's, awg's and feedback's copies (they keep them until their next rebuild). Used by spin.
- `rp_ao.v`: the four slow analog outputs AO0..AO3 on E2 (PWM at 125 MHz / 256 with a second-order delta-sigma of each output's 16-bit value: 0 is off, the mean duty v / 65536; pins `ao_o[3:0]` at LVCMOS18 on T10, T11, P15, U13); `rpfpga.ao` is the bit-exact reference (`Ao`, `AoChannel`). feedback's `fb_ao` (its offsets and PID targets summed first) keeps its copy until its next rebuild. Used by spin.
- `rp_avg.v`: a dead-time-free averager of points into block RAM (spin's `sp_avg`, moved for boxcar): two banks, each with two halves of `2^AW` points of (I, Q), `SUM_W`-bit saturating sums (read-modify-write over 4 clocks: a producer keeps 4 clocks between points of the same address); swaps at a shot into a free half (on request or after N scans, gated by `swap_ok`), the closed half's freeze once its producers are done (BLOCK fields with the saturation and the scans, per half or with `BANK_SCANS` per half and bank), the host's reads (three 32-bit words per point; with `AUTO_READ` an auto-increment address for repeated reads of one register), the release and clearing through the write port. `rpfpga.avg` is the bit-exact reference (`Averager`) and the host's decoding (`unpack`). Used by spin (1.1) and boxcar.
- `rp_seg_writer.v`: a finite writer of up to one 32-bit word per clock into a kept DDR allocation through an AXI3 HP port (16-beat bursts from a 2048-beat FIFO, four responses outstanding; the base, count and end checked before the first write; abort; with `gaps_ok` the producer may pause, else a missing word is an error); ajku's `iq_capture` structure. The host side keeps its allocation between captures with `rpfpga.buffers.KeptBuffer`. Used by boxcar (segments on S_AXI_HP1).
- `rp_xoro.v`: xoroshiro128+, one 64-bit word per clock (`PIPE` for 250 MHz).
- `rp_count48.v`: a 48-bit counter in a DSP48 (the slices are usually the scarce resource).

System ID

Every design has `rp_sysid` at **0x40000000** (4 KiB), a read-only AXI4-Lite block:

Offset	Content
0x00	magic `0x52505359` (”RPSY”)
0x04	layout version (1)
0x08	design ID: four ASCII characters, e.g. ”ECHO”, ”MADD”
0x0C	design version: [31:16] major, [15:0] minor
0x10	build ID
0x14	processing clock in Hz

The build ID is the first 32 bits of a SHA-256 over the hardware sources: the design RTL, the common RTL, the constraints, the saved block design and the vendor IP (the `.bd` without Vivado’s volatile `boundary_crc`). Identical sources give the same ID.

- `usr_access.tcl` stamps it into the FPGA’s USR_ACCESS register when the bitstream is written.
- `build.tcl` checks that the stamp is present in the `.bit`.
- `build.tcl` also exports it in `<design>.json`, together with the design ID and version, and the fields of the design’s `design_export` (below).

Safe loading

On the Zynq-7000, a CPU access to an FPGA address **hangs the bus and freezes the whole board** unless the FPGA holds a design that answers there and the PS–PL level shifters are on. After power-up the FPGA is empty and the level shifters are off until a bitstream is loaded.

PYNQ’s own ”loaded” record survives reboots. It is cleared at boot only once Jupyter has started (`clear_pl_statefile.service`, a minute or more on this board), and `Overlay.is_loaded()` trusts it. On 2026-10-04 a boot-time service that trusted it froze the board at every boot.

Therefore every driver (`rpfpga.Overlay`) follows these rules:

1. **Download unless rpfpga loaded this bitstream itself, in this boot, with nothing loaded since.** The record is `/home/xilinx/rpfpga/loaded.json`. It holds the design, the bitfile, the kernel boot ID and PYNQ’s load timestamp, read from the shared global state file, because PYNQ 3.0.1 sets `device.timestamp` only in the process that downloaded.
   - PYNQ 3.0.1 rewrites that file during a download only if the overlay metadata did not come from its cache, while its cold-start check deletes the old file. After a reboot, with a stale record for the same bitstream, a download would leave no record at all. `Overlay` therefore discards the cache flag before downloading (seen in the 2026-10-04 reboot test).
   - `clear_pl_statefile.service` deletes PYNQ’s global state file (`global_pl_state.json`) after Jupyter starts, so rpfpga’s record could no longer be verified against it. The service is therefore ordered after it.
2. **No register access unless the bus is up.** The FPGA manager must report ”operating” and all four level shifters must be enabled. This is checked, together with rule 1’s record, before every operation; the check reads only processor registers, so it is always safe.
   - It takes about 1.3 ms per transaction on the board: the FPGA manager’s state and the level shifters are read every time. The record, the boot ID and PYNQ’s state file are parsed again only when their files change. A file changed less than 0.1 s before it was read is read again the next time, so coarse file times cannot hide a rewrite (`files.read_cached`).
3. **Identify before use.**
   - The build ID must equal the one in `<design>.json`. A mismatch while attaching means new files were uploaded, so the driver loads them first, before the checks below, which would otherwise judge the old build (an upload of a new major version). A mismatch right after a download means the `.bit` and `.json` don’t belong together, which is an error.
   - The sysid magic must be present, the design ID must be the driver’s, and the major version must match.

Within one boot, all clients attach to the same download, so their settings stay in place: the service, Jupyter, the command line and scripts. `tests/test_overlay.py` checks these rules against a fake PYNQ, in which every access a real board would freeze on raises `Hang`.

The uniform driver API

Every design driver subclasses `rpfpga.Overlay`. Its TCP client subclasses `rpfpga.client.Client` and shares the high-level control methods below. Low-level hardware methods and extra measurement controls are driver-only;

service discovery and switching designs are client-specific.

Method	Purpose
`config()`	the applied configuration (a JSON-compatible dict)
`apply(patch, replace=False, expected_revision=None)`	apply a (partial) configuration in one transaction; returns the applied values. An optional revision check rejects an outdated draft. Unknown keys raise `ValueError`
`snapshot()`	consistent `config`, `revision`, `session`, `describe`, `status`, `calibration`, `ranges` (JSON-compatible, string output keys)
`apply_snapshot(patch, replace=False, expected_revision=None)`	apply and return its snapshot under the same transaction
`describe()`	`{output: one-line description}`
`status()`	`design`, `design_id`, `version` (the hardware's, from sysid), `driver_version`, `rpfpga_version`, `build_id`, `built` (the exported build's date), plus design extras
`safe()`	the design's defined safe state; protected output windows take precedence over a request for zero
`presets()`, `load_preset(name)`, `save_preset(name)`, `delete_preset(name)`	built-in and saved presets
`load()`	reprogram the FPGA (power-up configuration); invalidates all existing handles, including the caller
`reconnect()`	explicitly create a new handle; on a PC, selects the currently active design's client; a direct driver attaches to its own design without downloading
`calibration()`, `ranges()`, `set_input_range(ch, rng)`, `reset_calibration()`, `set_output_response(n, points)`, `set_output_delay(n, s)`, `set_input_response(ch, points)`, `set_input_delay(ch, s)`, `set_pins_delay(s)`	the board calibration and input jumpers (CALIBRATION.md); changes re-apply the configuration (`reset_calibration()` returns it)
`measure_input_offsets()`, `hold_reference()`, `release_reference()`, `calibrate(dmm=None)`	measure the board calibration; designs with ADC readback only (`status()["calibrates"]`). `release_reference()` returns the re-applied configuration. A step whose new constants cannot realize the configuration is not stored (`RuntimeError`)
`registers()`, `sysid()`, `build_id()`, `check_power_up()`	low level (driver only)

The measuring client API accepts `calibrate(dmm=None)` and `measure_input_offsets()`; driver-only options are `calibrate(..., levels=9, window_s=0.02)` and `measure_input_offsets(window_s=0.1)`. Raw `adc_codes()` and `hold_dac_codes()` are driver hooks. `build_id()` is also exposed as `status()["build_id"]`; driver-only `expected_build_id()` reads the export manifest and is not a status field. `rpfpga.version_text(device)` turns `status()` into one line, e.g. `atomic_clock: hardware 3.1, build 0xe2be7288 (built 2026-10-06 21:27), driver 3.1, rpfpga 1.1` (a client adds the PC's rpfpga version; a major-version mismatch between hardware and driver is flagged). Every design's notebook prints it after connecting, and every panel shows it (`rpfpga.gui.VersionLabel`).

State and data:

- The live state is on tmpfs, `/run/rpfpga/<design>/state.json`, shared by every process. A copy in `/home/xilinx/rpfpga/<design>/state.json` on the SD card is written in the background and restores the configuration after a reboot. SD writes can stall for seconds, so they stay off the command path.
- Saved presets and design settings live in `/home/xilinx/rpfpga/<design>/`.
- The board calibration (`calibration.json`) and input jumper settings (`board.json`) live in `/home/xilinx/rpfpga/`, shared by every design.
- The state file holds the applied configuration and the register image it belongs to.
- If the registers were changed by someone else, `config()` derives the configuration from the registers instead. On a freshly loaded design it returns the defaults with nominal calibration; on a calibrated board it reports the real (uncalibrated) power-up passthrough, derived from the registers.

Sessions and concurrent control

The service, direct Python drivers and command line share a **board-wide transaction lock** and revision counter. A short transaction includes reading the current settings, merging and validating a patch, writing it and preparing its reply. Thus a sparse edit keeps unrelated settings changed by another program, and the reply belongs to that edit. Hardware sample-level update semantics are design-specific.

Validation/revision rejection occurs before configuration writes. A runtime or transport failure can occur after a write has taken effect: transactions are not hardware rollback, and an error reply or lost reply does not establish that the output stayed unchanged. Reconnect/read back before deciding what to do next.

For an edit prepared from an earlier read, use a revision check (compare-and-set):

```
draft = dev.snapshot()
# Prepare an edit using draft["config"].
applied = dev.apply_snapshot(patch, expected_revision=draft["revision"])
```

If another accepted mutation changes the revision, this raises `Conflict` before applying the draft. Refresh and reconsider it. Without `expected_revision`, a patch is merged into the current configuration.

Long meter/calibration acquisitions release the transaction lock while waiting. They claim one acquisition slot, **not a reservation of the controls**:

- A second acquisition fails promptly with `Busy`; it is not queued.
- An accepted configuration edit, preset, mute, processing reset or board calibration/range change cancels the pending acquisition. It cannot later publish a reading based on the old settings or restore over the newer edit.
- A reload creates a new session, even for the same design and bitstream. All old handles, including the one that called `load()`, become stale. Use `new_dev = dev.reconnect()` explicitly, or construct a new driver/client.
- Each TCP connection handles one request at a time. Use a separate client connection for metering while controlling from another; `client.clone()` creates one pinned to the same session. The GUI does this automatically.
- Transport errors close the Python client's connection. Requests are never automatically replayed; reconnect explicitly and inspect the current state.

Stable exceptions are available from `rpfpga.errors` on both board and PC:

Exception (also available with `Error` suffix)	TCP code	Meaning
`Busy`	`BUSY`	another acquisition is active
`Cancelled`	`CANCELLED`	an acquisition was superseded or its deadline expired
`Stale`	`STALE`	the FPGA session changed; reconnect explicitly
`Conflict`	`CONFLICT`	the supplied configuration revision is no longer current

These inherit from `RemoteError` (`RuntimeError`). The target for control responsiveness while an acquisition waits is **100 ms**, not a measured board guarantee. The older network timings in the design documentation predate this concurrency implementation.

The TCP service

`rp-overlay.service` runs `python -m rpfpga.server` on port 5025 (and, for a design with a compatible protocol, port 23: below).

- **Boot:** it loads the last active design (`/home/xilinx/rpfpga/active.json`) safely and restores its configuration. It starts after `clear_pl_statefile.service`, which waits for Jupyter, so it is ready about two minutes after power-up. The design starts as its power-up configuration (a passthrough for mult_add and analog_echo, a 10 MHz sine for atomic_clock, everything off for synth, awg, lockin and feedback) until the restore, a moment later. `LOAD` does not restore: it leaves the power-up configuration. A `LOAD` that fails leaves no design active (`DESIGN?` replies `none`).
- **Stop:** on SIGTERM (`systemctl stop`) the service exits normally and writes pending state to the SD card first.
- **Background tasks:** a design may run one background task in the service (plugin `WORKER`, e.g. atomic_clock's phase-noise playback). The service starts it with each driver it opens (start-up, `LOAD`, re-attach) and signals the previous one to end; only the service runs it, never a Jupyter or command-line driver.

- **Protocol:** one command per line, keywords case-insensitive. Each reply is one line, `OK <value>` or `ERR <message>`, with JSON for structured values.
- **Concurrency:** short commands are board-wide transactions; meter and calibration waits allow other connections and direct drivers to proceed. The Python client is thread-safe, with one command at a time per connection.

Core command	Reply
`*IDN?`	`RedPitaya,rpfpga,<design>,<version>,<build>`
`HELP`	core and design commands
`DESIGNS?`	installed designs (JSON)
`DESIGN?`	the active design
`LOAD [design]`	switch designs or reprogram (power-up configuration)
`CONFIG?`, `CONFIG <json>`, `CONFIG REPLACE <json>`	read, patch or replace the configuration
`SNAPSHOT?`	atomic configuration, revision, session, descriptions, status, calibration and ranges
`SESSION?`	`{session, design}` for a new connection
`REQUEST <json>`	session-guarded command; optional revision check and atomic post-mutation snapshot (below)
`DESCRIBE?`, `STATUS?`	descriptions, status (JSON)
`PRESETS?`, `PRESET <name>`, `PRESET SAVE <name>`, `PRESET DELETE <name>`	presets
`SAFE`	safe state
`RANGE?`, `RANGE <IN1|IN2> <LV|HV>`	the board's input jumper settings (JSON); the active design re-applies its configuration
`CAL?`, `CAL RESET`	the board calibration (JSON); back to nominal
`CAL OFFSETS`, `CAL HOLD`, `CAL RELEASE`, `CAL LOOPBACK [<dmm json>]`	the measuring steps of CALIBRATION.md, on a design that can read its inputs
`CAL RESPONSE <OUT1|OUT2> <[[Hz, gain], ...]>`, `CAL DELAY <OUT1|OUT2> <s>`	store an output's AC amplitude response or delay (CALIBRATION.md)
`CAL RESPONSE <IN1|IN2> <[[Hz, gain], ...]>`, `CAL DELAY <IN1|IN2> <s>`	store an input's AC amplitude response or converter loop delay at its current jumper setting (CALIBRATION.md)
`CAL PINS <s>`	store the digital pins' delay relative to OUT1 (CALIBRATION.md)
`CAL PINSLOOP <s>`	store the pins' loop delay (an output pin to a 1 ns input pin; CALIBRATION.md)
`CAL SLOW <AO0..AO3|AI0..AI3> <V/unit> <V>`	store a slow analog output's or input's measured constants (CALIBRATION.md)

`RANGE?`, `RANGE`, `CAL?`, `CAL RESET`, `CAL RESPONSE`, `CAL DELAY`, `CAL PINS`, `CAL PINSLOOP` and `CAL SLOW` also work with no design loaded. Each design adds its own commands; see its `docs/PROTOCOL.md`.

The Python client obtains `SESSION?` when connecting, then wraps commands as:

```
REQUEST {"session":"<token>","command":"CONFIG <patch
↪ JSON>","expected_revision":42,"snapshot":true}
```

Here `command` is a single protocol line encoded as a JSON string; `expected_revision` and `snapshot` are optional. A successful short mutation with `snapshot: true` returns `OK <snapshot JSON>` from the same transaction. `LOAD` and long acquisitions cannot request that snapshot form. Acquisitions return their own result. Unwrapped commands keep their usual replies and act on the service's active handle, without a caller-supplied session guard. Concurrency failures are `ERR BUSY ...`, `ERR CANCELLED ...`, `ERR STALE ...` or `ERR CONFLICT ...`; other errors retain `ERR <message>`.

`make install-service`, run from any design, installs and restarts the unit with that design active.

Security: there is no authentication. Keep the board on a trusted network, or install with `SERVER_BIND=127.0.0.1` and use `ssh -N -L 5025:localhost:5025 xilinx@192.168.0.149`.

Record rings

Checked native register reads

`rpfpga` 1.2 adds optional compiled readout of an existing PYNQ register mapping. A worker calls `device.prepare_native_reads()` **outside a hardware transaction**; it compiles `native_read.c` with gcc into

an atomic, source/architecture-keyed cache in the board data directory. Models, unsupported mappings and unavailable compilers keep the Python implementation. `device.read_backend` reports `native` or `python`.

- `read_burst(offsets)` preserves the requested order, including repeated offsets.
- `read_repeated(offset, count)` reads a FIFO/data port repeatedly without building an offset vector. Both native operations are bounded to 2^20 words and validate all offsets before any side-effecting read.
- Every read remains inside the checked board-wide `Overlay.transaction()`, with session/acquisition validation. The lock remains held while ctypes releases the Python GIL. There is no automatic retry after entering the native read loop.
- The helper opens no device, creates no mapping and writes no register. Designs must use the Overlay methods, never the private native reader directly.

This accelerates register readout, not DDR/TCP raw transfer. Compiling does not reserve the hardware or establish a live session; binding the reader is checked after compilation. Tests in `tests/test_native_read.py` cover stale/unprogrammed gates, cancellation, transaction exclusion, fallback and repeated reads.

DDR record pump

`rpfpga.buffers.KeptBuffer(allocate, size, free=None, reserve=0)` keeps one contiguous allocation between finite captures (repeated large allocations fragment the board's CMA region): `take(units)` returns the kept one if large enough (else a new one, at first `reserve` units; if that fails the previous size is allocated again before the error is raised), `give(buffer)` keeps a drained one, `close()` frees it (ajku's pool, for any design; boxcar's segments).

For finite completed buffers, `rpfpga.buffers.immutable_copy(buffer)` (1.3) returns `(bytes, backend)`. On CPython with a writable contiguous buffer, libc `memmove` fills a newly allocated, still-private bytes object while releasing the GIL. This allows other Python control threads to run during a large uncached DDR copy. Read-only buffers or unsupported runtimes use the Python fallback. It creates no mapping and accesses no registers. The caller must establish producer completion, keep the source allocation alive, and recheck its acquisition context before publishing the result. Copy completion is never evidence that DMA has drained. Tests cover ownership/reference lifetime, immutable results and fallback.

A design that writes records into a ring in DDR (an FPGA master on S_AXI_HP0, the ring allocated by its worker; a design may wire further masters to S_AXI_HP1..3 in its `design.tcl`, as boxcar does for its segments) can serve them with the **ring pump**, `rpfpga/ringpump.c`: a small C program that the worker builds on the board with gcc on first use (`rpfpga.ringpump.build`, named by the source's hash, kept in the design's data directory) and runs as its child (`rpfpga.ringpump.Pump`). Python stays out of the data path: the pump maps the ring read-only through /dev/mem (uncached), frames new records and sends them to its TCP clients; it never touches FPGA registers. The worker keeps the registers: every poll it reads the FPGA's written position, passes it to the pump (`written`) and writes back the pump's position (`rd`), which gives the writer its room and its heartbeat.

The worker's generic part is `rpfpga.ringworker.RingWorker` (timetagger's first, now shared): the allocation (96 … 16 MB, the largest that is available), the writer's start with its watchdog (5 s), the pump (or the Python path without a compiler, through the design's `publish`), the poll every 10 ms, a restart of a writer whose watchdog tripped, and the close (the writer stopped and its last bursts waited for before the memory is freed). A design subclasses it with its register offsets (`W_CTRL`, `W_BASE`, `W_SIZE`, `W_RD`, `W_WD`, `W_WR`, `STATUS` with `RUNNING`/`TRIPPED`), the pump's magic and context, and the hooks `prepare`, `opened`, `header`, `publish`, `pump_port` and `poll`; `tests/test_ringworker.py` runs it on a fake ring. The FPGA side is `rtl/optional/rp_ring_writer.v`.

The Python path's stream is `rpfpga.ringhub`: a `Hub(magic, context=None)` that frames the published records as the pump does (a design with a context gives a function of the records and the two context words), each client's queue (32 MB, whole frames lost when it falls behind), and `stream(hub, header, stop)`, the `RECORDS STREAM` generator (it relays the pump's stream while the pump serves the records). On the PC, `rpfpga.client.RecordStream` (standard library) reads the stream from the pump's port (5026) or the service's; a design's subclass sets `MAGIC`. Both are timetagger's first, now shared (counter uses them too).

- **Clients** connect to the pump's port, send one line, get `OK {header JSON}` and then frames of a 24-byte header (little-endian u32: magic, sequence, payload bytes, bytes lost before this frame, two context words) and the payload. Each client has its own queue (16 MB); a client that falls behind loses whole frames, counted in the next frame's header; the ring is never held up.
- **Context** (`context=True`): the pump tracks control records (bits 31..27 = 31) of the record format and puts the context in force at each frame's first record into the header, so a client can decode a frame on its own after a loss (docs/PROTOCOL.md of the design).

- **The tap** (`TAP_TYPES` of the worker; `tap_types` and `on_tap` of `Pump`): for record formats whose header word holds the type in bits 31..27 and the record's words (the header included) in bits 26..20, the pump also hands every whole record of the chosen types to the worker's `publish` (through a SOCK_SEQPACKET socket, the records of each frame at once, in the pump's tap thread), so the service can keep sums of some records while the pump streams them all; a message the worker cannot take at once is dropped (`stats["tap_lost"]` counts the bytes), the ring is never held up. Without `TAP_TYPES` the pump is unchanged.
- **Without a compiler** the worker keeps a Python path (slower); the design's stream command on port 5025 relays the pump when it runs.
- **Speed** (timetagger, 2026-10-28): about 27 MB/s sustained to a PC on Gigabit Ethernet, against about 7 MB/s through Python. `common/tests/test_ringpump.py` checks it on a file.

Stability tools

`tools/stabilitylib` is the host-only frequency-stability analysis that the designs share (numpy, scipy, matplotlib, allantools; not installed on the board). Put `common/tools` on `sys.path` and import `stabilitylib`:

Module	Contents
`model`	the power-law noise model (IEEE 1139: h2 ... h_2, drift, sinusoidal PM): expected ADEV, MDEV, HDEV, TDEV of windowed samples (a phase meter; `wpm_gain` for a reader that sees more white PM than a plain average) or point samples (white PM of bandwidth f_h, its exact sampled covariance), and spectra
`data`	`Series` (equally spaced time error with gaps as segments; optional clocks A and B against one reference) and `Recording` (one or two series and the settings the data should show; subclasses supply their model, notes and checks)
`analysis`	deviations with confidence intervals (allantools, noise identification, Greenhall EDFs), the weighted non-negative fit of h and drift, verdicts, spectra, quadratic and logarithmic drift, a measured floor
`extras`	Totdev and ThéoBR (SP 1065 bias and EDF), MTIE and TIE rms, the three-cornered hat, dynamic ADEV
`masks`	ITU-T G.811 and G.8262 MTIE and TDEV masks
`external`	phase or frequency files of other instruments (units, gaps, time tags) as a `Recording` of point samples
`report`	a PDF report of `analysis.analyze()`

A design subclasses `data.Recording` for its own recordings (atomic_clock: its phase meter's model and board recordings, `tools/clocklib`), or builds an `external.ExternalRecording` from a `Series` of its own (timetagger: the time error of tagged edges, `tools/stability.py`). `tests/test_stabilitylib.py` tests it (skipped without its packages).

Command line, client and GUI

One command line for all designs:

```
sudo /usr/local/share/pynq-venv/bin/python -m rpfpga.cli mult_add --preset multiplier --meters 0.1
 ↪  # on the board
python3 -m rpfpga.cli --host 192.168.0.149 --preset multiplier           # from a PC: the active
 ↪  design
```

- Common options: `--load`, `--range IN=lv|hv`, `--reset`, `--preset`, `--safe`, `--calibration`, `--json`. The design adds its own, listed in `--help`.
- On the board, the design name goes first, before any option.
- From a PC, put `common/python` and `<design>/control` on `PYTHONPATH`.
- `rpfpga.client.connect(host)` returns a client of the active design's class.

Jupyter: each design's notebook does `import rpfpga` and then `from pynq.overlays.<design> import DRIVER, make_gui`. Import `rpfpga` first in scripts run with `sudo`: it sets `XILINX_XRT`, which sudo drops and PYNQ needs. On the board, a design's plugin package *is* its overlay directory, following PYNQ's `pynq.overlays.<name>` convention.

Shared GUI pieces (`rpfpga.gui`): `MessageArea`, `RangeSelector` (input jumpers), `PresetsPanel` and `CalibrationPanel`. The calibration panel shows the measuring steps only on designs that can measure. A design's GUI adds only its own controls.

Panels send only changed fields in live mode. In panels with a staged-edit mode, **Apply** uses the draft's revision; **Refresh** updates applied readback without replacing the draft. A conflicting draft is retained and requires explicit

Discard before starting again. A stale session disables the panel: reconnect and create a new panel. Meter polling uses its own connection and suppresses cancelled or out-of-date results. `ManagedPoller` allows one outstanding UI publication. A slow notebook/plot renderer therefore backpressures polling rather than accumulating callbacks that starve buttons and cleanup. Closing releases its worker even if the notebook event loop has stopped delivering callbacks. This bounds UI publication, not the instrument's independent streaming/recording queues or its hardware acquisition lifetime.

Designs on the lock-in core (lockin, feedback) share two modules. `rpfpga.demod` has their measurements: averaged readings, auto-phase, PLL settings and lock-up, the inputs' loop delay, sweeps, the resonance fit, input spectra, the filters' noise bandwidth and settling time. `rpfpga.rows` has the FIFO row stream: the row fields, the hub, `DATA STREAM`, the service's worker, and `DataStream` and `record()` on a PC. Each design's `analysis`, `stream`, `worker` and client modules re-export or extend them; `tests/test_designs.py` checks that such a design's constants match.

Writing a design's protocol commands (`rpfpga.protocol`): `Command(line)` splits a line into `head`, `args` and `rest` and checks argument counts with `need(n)`. `number()` and `flag()` parse values, and `to_json()` formats replies. A plugin's `handle()` returns `None` for commands that are not its own, and may raise: the server replies `ERR <message>`.

Binary data. Large data (e.g. awg's arbitrary-waveform segments) travel as one `BINARY` request: the line `BINARY <bytes> <request line>` (the request line is usually the client's `REQUEST <json>`), followed by exactly that many raw bytes, at most 64 MiB + 4 KiB. The reply is one line as usual. A plugin lists the heads of its binary commands in `BINARY_COMMANDS` and receives the data as `handle(device, line, payload)`; the server refuses data for other commands (`ERR <head> takes no binary data`) and closes the connection on a malformed size. `Client.binary(command, data)` sends one.

Streaming commands. Some data are best pushed rather than polled. A plugin may declare `STREAM_COMMANDS` (command prefixes, e.g. `("MEAS STREAM",)`) and `stream(device, line, stop)`, a generator of reply lines; atomic_clock's `MEAS STREAM` is the example.

- **Sending:** the client sends such a command unwrapped (not inside `REQUEST`), on a connection of its own.
- **Server:** it runs the generator on its own thread. The first line (the header, or `ERR ...`) is made under the board lock; the rest is sent as it comes.
- **Ending:** the stream ends when the client sends `STOP` or disconnects, or when the generator ends; the connection then closes.
- **A slow client:** at most 16 lines wait for it, and then the generator blocks. The plugin's own queue decides what is lost and must say so in its lines.
- **Generator:** it must never touch the hardware. It reads what the design's worker publishes, because the worker owns the FPGA in the service.
- **Binary data:** after its first (text) line a generator may also yield `bytes`; they are sent unchanged, so the plugin frames them itself (e.g. a length before each block; the demo plugin's `BLOCKS` in `tests/test_cli.py`). This is for bulk data such as timestamp records, where text lines and base64 would cost the board's CPU.

A compatible protocol on a second port. Programs written for another instrument can talk to a design that imitates its command set. The service listens on a second port, `--compat-port` (default 23, an SR860's telnet port; 0 turns it off), and hands each line to the active plugin's `compat(device, line, state)`.

- **Replies** are bare lines, without `OK`; the function returns `None` for no reply. `state` is a dict of the connection's own, for status registers and the like.
- **Errors** reach the client only as the plugin decides (e.g. an instrument's event status bits). An exception in `compat` gives no reply, and the service logs it.
- **Long commands:** heads listed in `COMPAT_LONG` run as acquisitions (as `LONG_COMMANDS`); the others run inside one board transaction.
- **No handler:** with a design that has no `compat`, or none loaded, the port answers nothing.
- **Telnet:** telnet's option bytes (IAC sequences) are dropped from the lines. lockin's SR86x subset is the example (`lockin/docs/PROTOCOL.md`).

Adding a new design

1. **Create the directory** `mydesign/` with:
   - `Makefile`: `DESIGN := mydesign` and `include ../common/mk/design.mk`;
   - `config.mk.example`: copy it from another design.
2. **Write `design.tcl`:**

- `design(name|id|version|rtl|testbenches)`, and optionally `design(xdc)` (e.g. `{constraints/*.xdc}`) for pins of ports the design adds and `design(impl_properties)`, property/value pairs of the implementation run (e.g. `STEPS.PLACE_DESIGN.ARGS.DIRECTIVE AltSpreadLogic_high` for a congested design; set at creation and checked by every build);
- optionally `proc design_export`, run by `build.tcl` on the routed design (`open_run impl_1`) before the export: it returns name/JSON-text pairs that become fields of `<design>.json` next to the build ID (timetagger exports its paths' delays from the timing model; the driver reads them with `build_info()`);
- `proc design_bd`, which creates the datapath between `redpitaya_adc/adc_data_{1,2}` and `redpitaya_dac/dac_data_{1,2}`;
- extra top-level ports (e.g. extension-connector pins) are created in `design_bd` with `create_bd_port` and constrained in the design's own `constraints/*.xdc` (listed in `design(xdc)`; added to the project, the build ID and `verify_build.py`'s source copy). The common constraints keep the extension connector commented out. After changing `design_bd`, regenerate the project (remove `<design>.xpr` and `<design>.srcs`, `make`);
- connect the design's registers to `axi_smc/M01_AXI`, clock them from `redpitaya_clk/clk_125`, reset them from `rst_125/peripheral_aresetn`, and assign their address (not 0x40000000).

3. **Put the RTL in `src/`, and testbenches in `tests/<top>.sv`.** Each testbench prints a unique `PASS: ...` line, listed in `design(testbenches)`.
4. **Write the plugin package `control/mydesign/`.**
   - Its `__init__.py` provides `DRIVER` and `CLIENT`. Optionally it also provides `HELP` and `handle` (TCP commands), `add_arguments`, `apply_arguments`, `readings` and `report` (CLI options), `make_gui` (Jupyter), `WORKER` (a background task run by the service) and `compat` with `COMPAT_LONG` (a compatible protocol on the service's second port). `rpfpga.plugins` supplies defaults for whatever is missing, and takes `DESIGN` from `DRIVER.DESIGN`.
   - A `WORKER` is a class built as `WORKER(device, data_dir)` with `start()` and `stop()`. `stop()` only signals (the caller may hold the board lock); the task uses short transactions, sleeps outside them, ends on `rpfpga.errors.Stale` (the design was reloaded), and keeps its own register writes out of `CONFIG_REGISTERS` unless they are configuration edits.
   - The driver subclasses `rpfpga.Overlay`. It sets `DESIGN`, `DESIGN_ID`, `VERSION`, `REGISTERS`, `REGISTER_BASE`, `CONFIG_REGISTERS`, `POWER_UP_REGISTERS` and `PRESETS`.
   - It implements `default_config`, `compile`, `derive`, `describe_config` and `safe_config`.
   - `compile` and `derive` convert volts with the board calibration (`self.calibration()`, `self.ranges()`; CALIBRATION.md, "For design authors").
   - If the design can read its inputs, it also implements `adc_codes` and `hold_dac_codes`, which enable the measuring steps.
   - Add `control/mydesign.ipynb`.
5. **Write the tests** in `tests/test_*.py`: model-based tests of the driver, the protocol through `rpfpga.server.Server`, and the client over localhost. `rpfpga.testing` provides `FakePynq` (loading), `FakeRegisters` (a register block), `ServerThread` (the TCP server in a thread) and `temp_data_dir()`. The root `Makefile` and the workflow tests find the design by its `design.tcl`. Put a behavior-changing RTL edit in `mydesign/tests/verify_build.json` (`file`, `original`, `replacement`), which `common/tests/verify_build.py` uses. `tests/test_designs.py` then checks the required files and that `design.tcl` and the driver agree.
6. **Write the documentation**, at the level of the existing designs:
   - `README.md` for lab users: overview, equipment safety (linking to the root README), quick start, presets, specifications, hardware setup;
   - `AGENTS.md`, `NOTICE.md`;
   - `docs/THEORY.md`, `docs/REGISTERS.md`, `docs/PROTOCOL.md` and `docs/VERIFICATION.md` (dated records of what was tested), plus `RECIPES.md` (or `docs/RECIPES.md`) and `docs/CALIBRATION.md` where relevant;
   - add the design to the table in the root README.
7. **Verify:** `make test`, `make sim`, `make`, `python3 ../common/tests/verify_build.py --design mydesign`, then `make test-remote` on the board.