# analog_echo 1.0: reference documentation

**overview, theory, control reference, registers, test records**

Rok Zitko, Jozef Stefan Institute

2026-11-13 (commit 6af1a9b)

## Contents

**analog_echo: gain and offset for both Red Pitaya channels** **1**
- Before connecting equipment . . . 2
- Quick start . . . 2
- Presets . . . 3
- Specifications . . . 3
- Hardware setup . . . 3
- Documentation . . . 4
- Changing the design . . . 4
- Source layout . . . 4
- Verification . . . 4
- Author . . . 4

**Theory of operation** **5**
- Number representation . . . 5
- The gain/offset stage (`src/scale_offset.v`) . . . 5
- Control path . . . 5
- Physical units (the board calibration) . . . 6
- Accuracy and limits . . . 6

**Control reference** **6**
- Configuration . . . 7
- Presets . . . 7
- Python methods . . . 7
- TCP commands . . . 9
- Command line . . . 9
- Where the state is kept . . . 10

**Register map** **10**
- Gain/offset words (axi_gpio_0) . . . 10
- Raw access . . . 11

**Verification record** **11**
- 2026-10-04: consistency and correctness review . . . 11
- 2026-10-04: board calibration applied . . . 11
- 2026-10-04: moved onto the common framework . . . 11
- Earlier (stand-alone version) . . . 12

## analog_echo: gain and offset for both Red Pitaya channels

analog_echo passes IN1 to OUT1 and IN2 to OUT2, sample by sample at **125 MSa/s**, with a software-set **gain and offset** on each channel:

```
OUTn = gain_n · INn + offset_n        (volts at the connectors; saturates at the output limits)
```

Use it to buffer, invert, scale or offset a signal, or as a quick check of the signal chain. Gains (volts out per volt in) and offsets (volts) are **real volts at the SMA connectors**: the software converts them with the **board calibration**, which belongs to the board and is shared by every design. analog_echo applies that calibration but cannot measure

it (it cannot read its inputs); it is measured once with mult_add (../common/CALIBRATION.md). The range is that of the converters: on this board about **±1.13 V** at the inputs (LV jumper setting) and at the outputs. For a far more capable signal processor (products, ratios, rectifiers, output low-pass filters, Schmitt comparators, clamps and meters), see mult_add.

## Before connecting equipment

Read the repository's Connecting to lab equipment: read first section. For analog_echo in particular:

- **Loading the design makes the outputs a passthrough.** Whenever analog_echo is loaded (at power-up, by `LOAD analog_echo`, by creating the driver while another design runs, by `make test-remote`, or by the first access after `make upload` of a new build), it starts at unity gain and zero offset in raw converter units, i.e. without calibration: on this board OUT1 ≈ 1.006·IN1 − 23 mV and OUT2 ≈ 1.005·IN2 − 6 mV (computed from the calibration, not measured). It stays so until a configuration is applied. After a reboot, the service restores the last configuration. `make upload` itself copies/installs files; it does not program the FPGA.
- **There is no clamp.** The outputs saturate at the converter limits, about ±1.13 V on this board. A gain above 1 can drive them to full scale with a small input.
- **Safe state:** `SAFE` (Python `echo.safe()`) sets gain 0 and offset 0, so both outputs sit at the calibrated 0 V.
- **Load:** all voltages refer to a high-impedance load (scope at 1 MΩ, DMM). The output voltage into 50 Ω has not been measured.

## Quick start

The board is already set up (../common/SETUP.md) and reachable at `192.168.0.149`. Connect a function generator to IN1 and IN2 (at most 0.8 V amplitude, input jumpers on LV) and a scope (1 MΩ input) to OUT1 and OUT2. Then use whichever is convenient:

- **Jupyter** on the board: open `http://192.168.0.149:9090` (password `xilinx`), then `analog_echo/analog_echo.ipynb`, and run the cells. The control panel has three tabs:

| Tab | Contents |
|---|---|
| Channels | gain and offset of each channel (slider and text box, **Invert**, **Unity / 0 V**), a warning when a full-scale input would clip, the IN1/IN2 LV/HV selectors, **Refresh from hardware**, **Reset both**, **Mute all** |
| Presets | load, save and delete presets |
| Calibration | the board calibration in use; **Reset to nominal** (measure it with mult_add) |

  Edits apply immediately and patch only the changed field. **Invert** checks the revision of the displayed gain, so it reports a conflict if another program has changed the configuration. **Refresh from hardware** reads a consistent snapshot. Every reload invalidates the panel's handle: reconnect and create a new panel.

- **Python on the board** (in Jupyter, or a script run with `sudo /usr/local/share/pynq-venv/bin/python`):

```
import rpfpga                                   # must come first
from pynq.overlays.analog_echo import DRIVER
echo = DRIVER()                               # loads analog_echo if the board runs something else
echo.set(1, gain=2.0, offset=-0.25)             # returns the applied values
echo.load_preset("inverted")
print(echo.describe())                          # {1: 'OUT1 = -IN1', 2: 'OUT2 = -IN2'}
```

  If analog_echo is already loaded (by the service, another notebook or the command line), `DRIVER()` attaches to it without touching the outputs; otherwise it loads the design first.

- **Python on a PC**, through the board's control service (standard library only). From the repository root:

```
import sys; sys.path[:0] = ["common/python", "analog_echo/control"]
from analog_echo import AnalogEchoClient
with AnalogEchoClient("192.168.0.149") as echo:   # the board must run analog_echo
    echo.set(1, gain=2.0, offset=-0.25)
    print(echo.describe())                        # on this board: {1: 'OUT1 = 2.001·IN1 - 0.25
     ↪ V', ...}
    print(echo.state())
```

  `rpfpga.client.connect("192.168.0.149")` returns the client of whichever design is running. If the

board runs another design, switch with `connect("192.168.0.149").load("analog_echo")` (TCP `LOAD analog_echo`); the outputs then become a passthrough. Reloading even the same design invalidates existing driver/client handles, including the caller. Reconnect explicitly before further calls; see concurrent control.

- **TCP** from anything: `nc 192.168.0.149 5025`, then for example `GAIN1 2`, `OFFS2 -0.25`, `STATE?`, `PRESET inverted`, `SAFE`.
- **Command line:** from a PC (with `common/python` and `analog_echo/control` on `PYTHONPATH`), `python3 -m rpfpga.cli --host 192.168.0.149 --gain1 2 --offset2 -0.25` acts on the running design. On the board itself (e.g. over SSH): `sudo /usr/local/share/pynq-venv/bin/python -m rpfpga.cli analog_echo --gain1 2`.

docs/PROTOCOL.md lists every method, TCP command and command-line option.

## Presets

| Name | Function |
|---|---|
| `passthrough` | OUT1 = IN1, OUT2 = IN2 (calibrated unity gain, 0 V offset) |
| `inverted` | OUT1 = −IN1, OUT2 = −IN2 |
| `notebook` | OUT1 = IN1 − 0.5 V, OUT2 = IN2 + 0.25 V (the offsets of the tutorial this design is based on) |

Save your own with `echo.save_preset("name")` (TCP `PRESET SAVE name`); saved presets are kept on the board. `echo.safe()` (TCP `SAFE`) puts both outputs at 0 V, and `echo.reset()` (TCP `RESET`) sets unity gain and zero offset.

## Specifications

Figures for this board use its calibration (2026-10-04, measured with mult_add and a DMM). Board-wide figures (input impedance, bandwidth, resolution, DC accuracy) are in the root README.

| Quantity | Value |
|---|---|
| Function | OUTn = gain_n · INn + offset_n, per channel, saturating at the output limits |
| Input and output range | about ±1.13 V on this board (LV jumper setting; HV: ±20 V nominal input) |
| Processing delay in the FPGA | 32 ns (4 clock cycles), fixed. The total delay from input to output connector has not been measured. |
| Gain range | ±128 × G·K, where G·K ≈ 1.006 on LV on this board: about ±128.8. On HV, about ±7.3 (the HV gain is nominal, not calibrated). |
| Gain steps | G·K/256: 0.0039 on LV (about 0.4 %). Requested gains are rounded to the nearest step: on this board, gain 1 is realized as 0.9986 (OUT1) and 1.0014 (OUT2), gain 2 as 2.001 (OUT1). |
| Offset steps | one DAC code, about 35 µV on this board. The DAC itself resolves about 138 µV. |
| Offset range | about ±1.13 V, shifted by the DAC offset and the amplified ADC offset; at gain 0, about −1.143 to +1.128 V on OUT1. An out-of-range request is refused with the exact range. |
| DC accuracy of an offset | +0.46 mV and −0.30 mV error at +0.5 V and −0.3 V (DMM check, docs/VERIFICATION.md) |
| Settings after loading | uncalibrated passthrough, OUT ≈ 1.006·IN with an offset of up to a few tens of mV, until a configuration is applied (see ”Before connecting equipment”) |
| Update time | earlier software measured under 1 ms per short on-board call; over TCP, `GAIN1` about 25 ms and `STATE?` about 11 ms; `LOAD` 1.2–1.8 s. The new concurrent-control path is unmeasured on the board (100 ms responsiveness target). |

Changing a setting is not real-time: a feedback loop closed in Python is limited to roughly 10–100 Hz. Gain and offset of one channel change together, in one clock cycle.

## Hardware setup

- **Input jumpers:** LV (about ±1.13 V on this board) or HV (±20 V nominal). After moving a jumper, tell the software: `echo.set_input_range("in1", "hv")`, TCP `RANGE IN1 HV`, the IN1/IN2 selector in the Channels

tab, or the command line's `--range in1=hv`. Gains stay in output volts per input volt; a setting that cannot be realized on the new range is rejected before hardware writes. Runtime/transport failures do not promise rollback. The setting belongs to the board, so every design sees it.

- **Calibration:** measured once with mult_add (../common/CALIBRATION.md). Without it, nominal converter constants are used, which on this board are about 13 % off. `echo.calibration()` (TCP `CAL?`, the Calibration tab) shows what is in use.
- **Outputs:** about ±1.13 V into a high-impedance load, with no clamp; they saturate at the converter limits.
- **A quick check:** with `PRESET notebook` and grounded inputs, OUT1 reads about −0.50 V and OUT2 about +0.25 V.

## Documentation

| Document | Contents |
|---|---|
| docs/PROTOCOL.md | configuration, presets, Python methods, TCP commands, command line |
| docs/THEORY.md | how it works: signal flow, number format, the gain/offset stage, calibration, accuracy |
| docs/REGISTERS.md | the hardware registers (for debugging) |
| docs/VERIFICATION.md | what was tested, when, and the build's resource use and timing |
| ../common/CALIBRATION.md | the board calibration and input ranges, shared by every design |
| ../common/README.md | the shared framework: TCP service, common API, build workflow |
| NOTICE.md | upstream sources |
| AGENTS.md | notes for maintainers and coding agents |

## Changing the design

Rebuilding needs Vivado 2025.1 on a build machine (../common/SETUP.md). From this directory, with Vivado's `settings64.sh` sourced:

```
make                        # create the project (first time), build, export the overlay
make test                   # offline tests: driver, protocol, client, command line, GUI
make sim                    # simulation: common system ID + the ADC/gain-offset/DAC datapath
make test-remote            # upload, program the board, check identity and power-up state
make install-service        # the control service, with analog_echo active
python3 ../common/tests/verify_build.py --design analog_echo
```

`make help` and ../common/README.md describe the common targets and variables. analog_echo has no `make test-hw`: check the analog path with a generator, a scope and a DMM. The design follows the FPGA Notes for Scientists tutorial (NOTICE.md) and is the simplest design on the common framework.

## Source layout

| Path | Contents |
|---|---|
| `design.tcl` | design description for the common recipe: ID "ECHO", RTL, registers and two `scale_offset` cells |
| `src/scale_offset.v` | the gain/offset stage |
| `control/analog_echo/` | Python package: driver, TCP commands, client, command-line options, Jupyter GUI |
| `control/analog_echo.ipynb` | the notebook |
| `tests/` | `tb_datapath.sv` (simulation with the actual converter interfaces) and the Python tests |

## Verification

What was tested, when, with results: docs/VERIFICATION.md.

## Author

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

## Theory of operation

analog_echo passes both inputs to the outputs at **125 MSa/s** (one sample every 8 ns), each through its own gain and offset stage. This page describes how that works inside the FPGA (the programmable logic chip of the Red Pitaya) and how the software turns volts into the numbers the hardware uses.

```
IN1 ── ADC ── scale_offset_1 ── DAC ── OUT1        code_out = sat16(round(code_in · q / 256) + o)
IN2 ── ADC ── scale_offset_2 ── DAC ── OUT2
                    ↑ {q, o}: one 32-bit control word per channel
ARM processor ── AXI bus ─┬─ GPIO block, two 32-bit registers (0x41200000)
                          └─ sysid block (0x40000000, common)
```

- **ADC / DAC:** the analog-to-digital and digital-to-analog converters, 14 bit each, clocked at 125 MHz.
- **scale_offset:** the gain/offset stage, `src/scale_offset.v`, one per channel.
- **AXI bus, GPIO block:** the processor (the board's ARM CPU running Linux and Python) reaches the FPGA through memory-mapped registers on an AXI bus. A Xilinx "AXI GPIO" block provides two 32-bit output registers, one control word per channel (REGISTERS.md).
- **sysid:** a small read-only block, common to every design, that identifies the loaded design ("ECHO", version, build ID) so the software never talks to the wrong one (`../../common/README.md`).

### Number representation

Inside the FPGA, signals are signed 16-bit integers ("codes", −32768 ... 32767):

- The ADC's 14-bit sample sits in the top 14 bits (bits 15:2), so input codes come in steps of 4.
- The DAC uses bits 15:2 and ignores bits 1:0, i.e. it truncates to steps of 4 codes.
- On the LV input range, **1 V is nominally 32768 codes**, for both the ADC and the DAC. The real conversion is the board calibration (below): on this board the LV inputs give about 28800–29050 codes per volt and the DACs about 28700–28900, so the full scale is about ±1.13 V instead of ±1 V.

### The gain/offset stage (`src/scale_offset.v`)

```
code_out = sat16( floor((code_in · q + o · 256 + 128) / 256) )
```

- **Gain code q:** a signed 16-bit number read as having 8 fractional bits (a fixed-point format often written "Q8.8"): the hardware gain is q/256, from −128 to +127.996 in steps of 1/256. `q = 256` is unity; negative values invert.
- **Offset code o:** a signed 16-bit number in output codes (DAC steps of one 16-bit code), nominally 30.5 µV and about 35 µV on this board. Because the DAC truncates to 4 codes, a static output moves in steps of about 138 µV. The truncation drops bits 1:0, 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 0.5 mV accuracy of the calibration.
- **Rounding:** to nearest, halves up. The multiply-add `code_in · q + C`, with `C = o · 256 + 128` folding the offset and the rounding constant together, maps onto one DSP48E1 (the FPGA's built-in multiply-accumulate block).
- **Saturation:** results beyond the 16-bit range are held at the limit (about ±1.13 V at the output on this board) instead of wrapping around; wrapping would flip a clipped signal to the opposite rail. There is no adjustable clamp.
- **Latency:** 4 clock cycles (32 ns) in the stage, fixed. The converter interfaces and analog front ends add more; the total delay from input to output connector has not been measured.
- **Atomic updates:** both halves of the control word are taken on the same clock edge, so one 32-bit write changes gain and offset together.

### Control path

Each channel's word `{q [31:16], o [15:0]}` is one output register of the GPIO block (REGISTERS.md). The registers read back the written value, which the driver uses to verify each write and to recognise changes made by other programs.

The driver (`control/analog_echo/driver.py`):

- converts gains and volts to codes with the board calibration (below);
- rejects values out of range before anything is written;
- writes both registers, reads them back, and stores the applied configuration;

- returns the applied (rounded) values.

Safe loading (never accessing the FPGA before it holds this design, which would freeze the board), identification via sysid and the uniform API come from `rpfpga.Overlay`; see `../../common/README.md`.

**Power-up state:** the registers start at `0x01000000` (q = 256, o = 0) when the design is loaded, i.e. unity gain and zero offset in codes. This is not calibrated: with this board's constants it gives OUT1 ≈ 1.006·IN1 − 23 mV and OUT2 ≈ 1.005·IN2 − 6 mV (computed from the calibration below, not measured). For registers in that state, the driver reports these calibrated values (gain 1 and offset 0 only with nominal constants); the next configuration applied is calibrated.

### Physical units (the board calibration)

The board calibration (`../../common/CALIBRATION.md`) models INn's ADC at its jumper setting and OUTn's DAC as straight lines:

```
ADC:  code_in  = K · V_in + A          K: codes per volt, A: offset in codes
DAC:  V_out    = G · code_out + B      G: volts per code, B: offset in volts
```

With gain code q and offset code o, the stage gives `code_out = q/256 · code_in + o`, so

```
V_out = (q/256 · G·K) · V_in  +  (G · (q/256 · A + o) + B)
          applied gain              applied offset
```

The driver solves this for the request:

- gain: q = round(gain / (G·K) · 256);
- offset: o = round((offset − B − G · q/256 · A) / G).

It reports the applied values above. With the board's constants (G·K = 1.0064 on IN1/OUT1), unity gain becomes q = 254, an applied gain of 0.9986 (on IN2/OUT2, G·K = 1.0053 gives q = 255 and 1.0014). The offset code then cancels both the DAC offset and the amplified ADC offset. `derive()` evaluates the same two expressions for register values written by someone else.

The gain limits follow from q: ±128 · G·K (about ±128.8 on LV on this board). The offset limits follow from o = ±32767: about ±1.13 V on this board, shifted by B + G · q/256 · A; the error message gives the exact range.

### Accuracy and limits

- **Resolution:** 14-bit converters: nominally 122 µV per step on LV (±1 V), about 138 µV on this board (±1.13 V). Gain steps are 1/256 of G·K, about 0.4 %.
- **Gain and offset errors:** corrected by the board calibration, to the accuracy it was measured with. A DMM check of two offsets after calibration showed errors of +0.46 mV and −0.30 mV (VERIFICATION.md). Uncalibrated, the converter gains on this board were off by 11–14 % and the offsets by up to about 15 mV.
- **Bandwidth:** set by the analog front ends (Red Pitaya specifies DC to 60 MHz, not checked here); the stage processes one sample per 8 ns.
- **Input range:** the jumper setting (`RANGE`) selects the HV or LV constants, so gains stay in volts per volt on HV. There K is nominally 20 times smaller, so the gain range is about ±6.4 nominally and ±7.3 on this board (its HV gain is not calibrated; the nominal 20 V full scale is used).

## Control reference

analog_echo is controlled in the same ways as every design in this repository:

- the **driver** `AnalogEcho` (Python on the board, e.g. in Jupyter);
- the **client** `AnalogEchoClient` (Python on a PC), which talks to the board's control service and shares its high-level controls;
- the **TCP protocol** of that service (port 5025, one text command per line);
- the **command line** `python -m rpfpga.cli`.

The parts common to all designs are described in `../../common/README.md` (the uniform driver API, the TCP service and its core commands, the command line). This page gives what a user of analog_echo needs, and everything analog_echo adds. All values are volts at the connectors, converted with the board calibration (`../../common/CALIBRATION.md`).

### Configuration

The configuration is a JSON-compatible dictionary:

```
{"outputs": {"1": {"gain": 1.0, "offset": 0.0},     # OUT1 = gain · IN1 + offset (V)
             "2": {"gain": 1.0, "offset": 0.0}}}
```

`apply(patch)` merges a partial configuration, e.g. `echo.apply({"outputs": {"2": {"offset": 0.1}}})`; `apply(config, replace=True)` starts from the default above. Both channels are recomputed and written within one software transaction; the two GPIO writes are not a simultaneous hardware commit.

**Gain** (output volts per input volt). The hardware multiplies by a signed 16-bit number with 8 fractional bits, i.e. in steps of 1/256 between −128 and +127.996, times G·K: the DAC's volts per code (G) times the input's ADC codes per volt (K), both from the board calibration. The driver uses the gain code

```
q = round(gain / (G·K) · 256)          applied gain = q/256 · G·K
```

- Steps: G·K/256. On this board G·K ≈ 1.006 on LV, so the steps are 0.0039 (about 0.4 %); on HV, K is about 18 times smaller on this board (20 nominally; the HV gain is not calibrated), and so are the steps.
- Range: ±128 · G·K: about ±128.8 on LV and ±7.3 on HV on this board.
- Unity gain is realized as 0.9986 on OUT1 and 1.0014 on OUT2 on this board.

**Offset** (volts). Set in steps of one DAC code, G: about 35 µV on this board (30.5 µV nominal). The range is ±32767 codes, about ±1.13 V on this board, shifted by the DAC offset and by the ADC offset times the gain. The error message gives the exact range for the current gain and calibration, e.g. `Offset must be finite and within [-1.1427, 1.1277] V`.

Values out of range raise `ValueError` (TCP: `ERR ...`), and nothing is written for that validation rejection. A runtime/transport failure can occur after a write takes effect; it does not promise rollback. Getters return the **applied** values, i.e. what the hardware realizes after rounding, not the requested ones.

**After loading** (power-up, `LOAD`, or the driver downloading the design), the hardware runs unity gain and zero offset in raw converter units, which is not calibrated: on this board OUT1 ≈ 1.006·IN1 − 23 mV (computed from the calibration). `config()` and `state()` derive the calibrated interpretation of those raw words; gain 1 and offset 0 are reported only with nominal calibration. Apply any configuration (for example `load_preset("passthrough")`) to get calibrated values on both channels. After a reboot, the service restores the last configuration by itself.

### Presets

| Name | Function |
|---|---|
| `passthrough` | OUT1 = IN1, OUT2 = IN2: unity gain, 0 V (the default configuration) |
| `inverted` | gain −1 on both outputs |
| `notebook` | the upstream tutorial's offsets: OUT1 = IN1 − 0.5 V, OUT2 = IN2 + 0.25 V |

`presets()` returns `{"builtin": {name: description}, "user": [names]}`. `save_preset(name)` stores the current configuration on the board (in `/home/xilinx/rpfpga/analog_echo/presets/`), `load_preset(name)` applies a built-in or saved preset, and `delete_preset(name)` removes a saved one. `safe()` sets gain 0 and offset 0, so both outputs sit at the calibrated 0 V.

### Python methods

These are on the driver `AnalogEcho` (board) and the client `AnalogEchoClient` (PC):

| Method | Description |
|---|---|
| `set(n, gain=None, offset=None)` | patch channel n (1 or 2): either or both values; returns its applied `{gain, offset, word}` in the same transaction and one client round trip |
| `set_gain(n, g)`, `set_offset(n, volts)` | set one value, keeping the other; return `get(n)` |
| `get(n)` | applied values of channel n: `{"gain", "offset", "word"}` (`word` is the raw 32-bit register value) |
| `state()` | `{1: get(1), 2: get(2)}` |
| `reset()` | unity gain, zero offset on both channels (calibrated); returns the resulting state in the same transaction and one client round trip |

| Method | Description |
|---|---|
| `config()`, `apply(patch, replace=False, expected_revision=None)` | the applied configuration; apply a patch, optionally checking the revision of a draft |
| `snapshot()`, `apply_snapshot(patch, replace=False, expected_revision=None)` | atomic config/revision/session/description/status/calibration/ranges; or apply and return that snapshot |
| `describe()` | one line per output, e.g. `{1: 'OUT1 = 2.001·IN1 - 0.25 V', 2: 'OUT2 = IN2'}` |
| `status()` | design name, ID, hardware/driver/rpfpga versions, build ID and date, register words, calibration method, `"calibrates": False`; `rpfpga.version_text(echo)` prints the versions on one line |
| `safe()` | gain 0, offset 0 on both channels |
| `presets()`, `load_preset(name)`, `save_preset(name)`, `delete_preset(name)` | presets (above) |
| `calibration()`, `reset_calibration()` | the board calibration in use; back to nominal constants |
| `ranges()`, `set_input_range("in1", "hv")` | the input jumper settings; record a change (the configuration is re-applied for it) |
| `load()`, `reconnect()` | reprogram (uncalibrated passthrough), invalidating all handles including the caller; explicitly obtain a new handle with reconnect |

The client also has `idn()`, `design()`, `designs()` and `load(design)` to switch the board to another design. Creating an `AnalogEchoClient` fails with a clear message if the board runs another design; `rpfpga.client.connect(host)` returns the client of whichever design is running. analog_echo cannot run the calibration's measuring steps (`measure_input_offsets()`, `hold_reference()`, `calibrate()`): they raise an error, because the design cannot read its inputs. Use mult_add for them.

Raw `registers()`, `read()`, `write()`, `sysid()`, `build_id()` and `expected_build_id()` are driver-only methods. `status()["build_id"]` reports the actual FPGA build; the expected manifest build is not a status field.

#### Concurrent control and reloads

Direct drivers and TCP clients share the board-wide transaction/revision system. `set_gain`/`set_offset` preserve the other field as it exists **inside the transaction**, and their replies cannot accidentally report a later program's edit. This software transaction does not make the two physical GPIO writes a simultaneous sample-level commit.

For a staged edit, capture `base = echo.snapshot()` and call `echo.apply_snapshot(patch, expected_revision=base["revision"])`. A changed revision raises `Conflict` before writing. The GUI sends sparse live patches for gain/offset and uses a revision check for **Invert**, which depends on the displayed gain. **Refresh from hardware** obtains a consistent snapshot.

Every reload, including the same design, invalidates old handles. After a successful `echo.load()`, use `echo = echo.reconnect()` (or create a new driver/client); recreate a stale GUI panel. Accepted edits cancel pending board acquisitions, and a second acquisition is `Busy`, not queued. Stable errors `Busy`, `Cancelled`, `Stale`, `Conflict` are shared by driver/client (`rpfpga.errors`; aliases with `Error` suffix). A transport failure closes the client connection; reconnect explicitly and inspect state, without automatic command replay. Details and the unmeasured 100 ms control-response target are in the common concurrency reference.

```
# on the board (Jupyter, or a script run with sudo /usr/local/share/pynq-venv/bin/python)
import rpfpga                                   # must come first
from pynq.overlays.analog_echo import DRIVER
echo = DRIVER()                                 # attaches, or loads analog_echo if needed
echo.set(1, gain=2.0, offset=-0.25)
print(echo.state())

# on a PC, from the repository root
import sys; sys.path[:0] = ["common/python", "analog_echo/control"]
from analog_echo import AnalogEchoClient
with AnalogEchoClient("192.168.0.149") as echo:
    echo.load_preset("inverted")
    print(echo.describe())
```

### TCP commands

Connect with `nc 192.168.0.149 5025` (or any socket library) and send one command per line; keywords are case-insensitive. Each reply is one line, `OK <value>` or `ERR <message>`, with JSON for structured values.

**analog_echo commands** (n = 1 or 2):

| Command | Reply |
|---|---|
| `GAIN<n> <g>` | `OK <applied gain>` |
| `GAIN<n>?` | `OK <current gain>` |
| `OFFS<n> <volts>` | `OK <applied offset>` |
| `OFFS<n>?` | `OK <current offset>` |
| `SET<n> <g> <volts>` | both values in one write; `OK <applied gain> <applied offset>` |
| `SET<n> JSON <patch>` | atomic partial update, e.g. `SET1 JSON {"offset":0.1}`; `OK {"gain":..., "offset":...,"word":...}` from that mutation |
| `STATE?` | `OK {"1": {"gain": ..., "offset": ..., "word": ...}, "2": {...}}` |
| `RESET` | unity gain, zero offset on both channels; `OK` |
| `RESET?` | perform the reset and return its atomic state: `OK {"1":{...},"2":{...}}`; despite `?`, this changes outputs |

**Core commands** most often needed (the full list is in `../../common/README.md`):

| Command | Reply |
|---|---|
| `*IDN?` | `OK RedPitaya,rpfpga,analog_echo,<version>, <build>` |
| `HELP` | the core and analog_echo commands |
| `DESIGN?`, `DESIGNS?` | the running design; the installed designs |
| `LOAD [design]` | reprogram, or switch to another design (the outputs become a passthrough) |
| `CONFIG?`, `CONFIG <json>`, `CONFIG REPLACE <json>` | read, patch or replace the configuration |
| `SNAPSHOT?`, `SESSION?`, `REQUEST <json>` | consistent readback; connection identity; session/revision-guarded command (common reference) |
| `DESCRIBE?`, `STATUS?` | one line per output; status (JSON) |
| `PRESETS?`, `PRESET <name>`, `PRESET SAVE <name>`, `PRESET DELETE <name>` | presets |
| `SAFE` | both outputs at 0 V |
| `RANGE?`, `RANGE <IN1\|IN2> <LV\|HV>` | the input jumper settings (JSON); record a change |
| `CAL?`, `CAL RESET` | the board calibration (JSON); back to nominal |

Example session:

```
GAIN1 2
OK 2.00103
OFFS1 -0.25
OK -0.250013
DESCRIBE?
OK {"1": "OUT1 = 2.001·IN1 - 0.25 V", "2": "OUT2 = IN2"}
```

(The values are those for this board's calibration.) The service has no authentication: keep the board on a trusted network (see the common README).

The Python client obtains a session on connection and uses `REQUEST` for every command. Short mutations and their replies share a transaction. Unwrapped legacy commands retain their existing replies. Stable concurrency failures are `ERR BUSY ...`, `ERR CANCELLED ...`, `ERR STALE ...` and `ERR CONFLICT ...`.

### Command line

```
# from a PC, through the service (acts on the running design):
PYTHONPATH=common/python:analog_echo/control python3 -m rpfpga.cli --host 192.168.0.149 --gain1 2
 ↪  --offset2 -0.25
# on the board itself (e.g. over SSH):
sudo /usr/local/share/pynq-venv/bin/python -m rpfpga.cli analog_echo --preset inverted
```

On the board, the design name comes first. Options:

| Option | Action |
|---|---|
| `--host H`, `--port P` | control a board through its service (default port 5025) instead of locally |
| `--data-dir D` | board data directory (default `/home/xilinx/rpfpga/analog_echo`) |
| `--load` | reprogram the FPGA first (uncalibrated passthrough) |
| `--range IN=lv\|hv` | record an input jumper setting, e.g. `--range in1=hv` (repeatable) |
| `--reset` | start from the default configuration (unity gain, 0 V) |
| `--preset NAME` | load a built-in or saved preset |
| `--gain1 G`, `--offset1 V`, `--gain2 G`, `--offset2 V` | analog_echo settings |
| `--safe` | finally put both outputs at 0 V |
| `--calibration` | also print the board calibration |
| `--json` | print the state, configuration and descriptions as JSON |

The actions run in this order: `--load`, `--range`, `--reset`, `--preset`, the analog_echo settings, `--safe`. Each call prints one line per output and the applied values with the register words, e.g. with this board's calibration:

```
OUT1 = 2.001·IN1
OUT2 = IN2 - 0.25 V
CH1: gain +2.0010, offset +0.00001 V (0x01fd044d)
CH2: gain +1.0014, offset -0.25000 V (0x00ffe4a0)
```

### Where the state is kept

The applied configuration is kept in `/run/rpfpga/analog_echo/state.json` (in memory, shared by every program on the board) and copied to `/home/xilinx/rpfpga/analog_echo/state.json` on the SD card, from which the service restores it after a reboot. If another program changed the registers, the driver derives the configuration from them. The board calibration and the input jumper settings are in `/home/xilinx/rpfpga/` (`calibration.json`, `board.json`), shared by every design.

## Register map

For debugging only: normal use goes through the driver, client or TCP commands (PROTOCOL.md), which work in volts.

The processor reaches the FPGA through two AXI4-Lite slaves (memory-mapped blocks of 32-bit registers on the processor's AXI bus):

| Base | Size | Block |
|---|---|---|
| 0x40000000 | 4 KiB | common sysid (read-only identification), design ID **"ECHO"**, version 1.0: see `../../common/README.md` |
| 0x41200000 | 64 KiB | `axi_gpio_0`: Xilinx AXI GPIO with two 32-bit output channels |

### Gain/offset words (axi_gpio_0)

| Offset | Channel | Bits 31:16 | Bits 15:0 | After loading |
|---|---|---|---|---|
| 0x0 | OUT1 | gain code q, signed, 8 fractional bits (gain q/256) | offset code o, signed, in DAC codes | `0x01000000` (q = 256, o = 0) |
| 0x8 | OUT2 | gain code q | offset code o | `0x01000000` |

The stage computes `code_out = sat16(round(code_in · q / 256) + o)` (THEORY.md). The power-up word is unity gain and zero offset in codes, i.e. an **uncalibrated** passthrough; the driver computes calibrated q and o from the board calibration.

The data registers read back the written value. Write whole 32-bit words: one write updates gain and offset together.

PYNQ's `AxiGPIO` channel `write(val, mask)` merges `val` into a per-process copy that starts at zero. A fresh

process writing only bits 15:0 would therefore also set the gain to zero, so the driver writes raw 32-bit words instead.

### Raw access

Use the driver, `pynq.overlays.analog_echo.DRIVER()`, rather than raw memory access (PYNQ `MMIO`). The driver loads the design when needed and refuses to touch registers that cannot answer: an access while the FPGA does not hold this design freezes the board (see "Safe loading" in the common README). For debugging, after the driver has loaded the design:

```
import rpfpga                                    # must come first
from pynq.overlays.analog_echo import DRIVER
echo = DRIVER()                                  # safe load / attach
with echo.transaction():
    echo.write(0x0, 0x0200c000)                # q = 512, o = -16384; nominally gain 2, offset -0.5 V
    print(hex(echo.read(0x0)), echo.sysid())
    print(echo.get(1))                           # calibrated interpretation, not nominal equation
# To request physical gain 2 and offset -0.5 V, use echo.set(1, gain=2, offset=-0.5).
```

Raw writes bypass the driver's state file. The driver notices and derives the configuration, in calibrated volts, from the registers.

## Verification record

What was tested on analog_echo, when, and with what result, newest first. Only results that were actually obtained are listed; anything not yet done is said so. The build's resource use and timing are in the record of the build they belong to (currently "moved onto the common framework", below).

How to repeat the checks: `make test`, `make sim`, `make`, `python3 ../common/tests/verify_build.py --design analog_echo` and `make test-remote` (see "Changing the design" in ../README.md). analog_echo has no `make test-hw`, because it cannot read its inputs: check the analog path with a generator, an oscilloscope and a DMM.

### 2026-10-04: consistency and correctness review

- No hardware change (build 0xf98c8ee2 unchanged after the rebuild with the updated build scripts). `make sim` passed (`tb_sysid`, `tb_datapath`).
- `make test-remote` passed. `make test`: 18 tests (now including a calibration that cannot realize the configuration, which is not stored), also on the board with the GUI.

### 2026-10-04: board calibration applied

- The board calibration (`rpfpga.calibration`) is now shared by all designs; analog_echo applies it, and the input ranges, to its gain and offset codes.
- **`make test`:** 17 tests passed offline and on the board, including the GUI with the new range selectors. The new tests use the board's real constants and check the modeled output against the applied values. They also cover the HV range and the refusal to run the measuring steps.
- **On the board:**
  - With the DMM-referenced calibration (measured with mult_add), OUT1 was set to gain 0, offset +0.5 V, and OUT2 to gain 0, offset −0.3 V.
  - The offset codes were 14648 (nominal 16384) and −8669.
  - An independent 6.5-digit DMM read **+0.50046 V** and **−0.30030 V**: errors of +0.46 mV and −0.30 mV. Uncalibrated they would have been about +60 mV and −41 mV (calculated: nominal code −9830 gives $1.1425 \cdot (-0.3\ \text{V}) + 1.7\ \text{mV} = -0.341\ \text{V}$).
  - The remaining error matches the drift seen between two loopback calibrations (about 0.5 mV).

### 2026-10-04: moved onto the common framework

Using Vivado 2025.1, Python 3.12 on the build machine, and PYNQ 3.0.1 on the board at 192.168.0.149.

- **`make sim`:** `tb_sysid` (common) and `tb_datapath` passed.

- **`make test`:** 15 tests passed offline (the GUI test is skipped without ipywidgets) and on the board, including the GUI.
- **Build (0xf98c8ee2):** 4.56% of LUTs (803), 3.33% of registers, 2 DSP48E1. All timing constraints met: WNS +0.835 ns, WHS +0.039 ns. Routed DRC shows one RTSTAT-10 ”no routable loads” warning. `check_stamp` found the build ID in the `.bit`.
- **`python3 ../common/tests/verify_build.py --design analog_echo`:** every stage passed.
  - Recipe-only creation took 344 s; an unchanged rebuild took 22 s.
  - The build ID of the regenerated project equals that of the first build.
- **`make test-remote`:** the board was programmed through the driver; sysid (ECHO 1.0), the build ID and the power-up state were verified, and the `passthrough` preset was applied.
- **Service:** `rp-overlay.service` switched to analog_echo with `LOAD analog_echo` (1.2–1.8 s), and `connect()` returned an `AnalogEchoClient`. Round trips: `GAIN1` about 25 ms, `STATE?` 11 ms.
- **Jupyter:** `control/analog_echo.ipynb` executed headlessly (nbconvert) on the board.
- **Reboot:** the service’s boot path, which is common to all designs, was tested with mult_add as the active design (see its record).
- **Not yet done:** observing the analog outputs with an oscilloscope.

### Earlier (stand-alone version)

- Gain/offset hardware verified in XSim, including mutation tests.
- Timing met; readback of the GPIO words confirmed on the board with PYNQ 3.0.1.
- Analog waveforms were never observed by automated tests.