synth: user's manual

A 16 + 16 tone phase-coherent synthesizer on the Red Pitaya STEMlab 125-14

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

Red Pitaya `synth` design, version 2.1
October 25, 2026

Abstract

`synth` makes each output of the Red Pitaya the sum of 16 sine waves, each with its own frequency, amplitude and phase, all referred to one zero time shared by both outputs. Changes happen on one exact sample, ramps follow smooth shapes and land exactly on their targets, and timed sequences run in the FPGA, started by a trigger and repeated by the FPGA itself if you wish. This manual is for the lab: the first signals, what every setting means, phase coherence, sequences for acousto-optic deflectors and pulse experiments, amplitude modulation from an input, the board calibration, and the checks the board makes on itself through its inputs. Each chapter starts with a worked example and then gives the exact rules and limits. The figures marked "measured" were recorded on the board used here; the others are computed. The command reference is in appendix A; the FPGA internals are in `docs/THEORY.md` and `docs/REGISTERS.md`, the test records in `docs/VERIFICATION.md`.

Contents

1 What synth is **3**
1.1 How a tone is made, in one page . . . 4

2 Before you connect equipment **4**

3 Setting up **5**

4 Getting started **5**
4.1 A first signal in five lines . . . 5
4.2 The four ways to control synth . . . 6
4.3 The notebook panel . . . 7

5 Tones **7**
5.1 Example: a five-tone comb . . . 7
5.2 Headroom and crest factor . . . 8
5.3 Droop correction and the output response . . . 8
5.4 All settings . . . 9
5.5 Presets . . . 9

6 Phase and the zero time **10**
6.1 Example: coherent switching . . . 10
6.2 Phases at the connectors . . . 10

7 Sequences **11**
7.1 Example: a step, a chirp and a smooth turn-off . . . 11
7.2 Steps and ramps . . . 11
7.3 Shaped ramps . . . 12
7.4 Triggers and markers . . . 13
7.5 Sequences repeated by the FPGA . . . 13
7.6 Limits . . . 14
7.7 Reference: the sequence dictionary . . . 15
7.8 Long sequences . . . 15
7.9 During and after a sequence . . . 15

8 Amplitude modulation from an input **15**
8.1 Example . . . 15

9 Calibration **16**
9.1 DC calibration . . . 16
9.2 Output frequency response . . . 16
9.3 Output delay (OUT1–OUT2 skew) . . . 17

10 Measuring with the inputs **17**
10.1 Tones . . . 17
10.2 Spectra . . . 17

11 Recipes **19**
11.1 Sweeping the phase between the outputs . . . 19
11.2 Turning an RF amplifier on and off gently . . . 19
11.3 A stepped frequency sweep with a marker per step . . . 19
11.4 A slow linear sweep for spectroscopy, repeated . . . 19
11.5 An amplitude staircase for a linearity check . . . 19
11.6 Multitone noise . . . 20
11.7 A two-tone test of an amplifier . . . 20
11.8 Frequency-shift keying . . . 20
11.9 What synth cannot do . . . 20

12 Using synth from experiment software **20**
12.1 From Python . . . 20
12.2 From other languages . . . 21
12.3 Building sequences in code . . . 21
12.4 Several programs at once . . . 21

13 Application notes **22**
13.1 Acousto-optic deflectors and optical tweezer arrays . . . 22
13.2 Pulse sequences and triggers . . . 23
13.3 Phase-coherent RF . . . 24
13.4 Measuring a filter or an amplifier . . . 25

14 Specifications **25**

15 Troubleshooting **26**

16 Quick reference **27**

A Command reference **27**
A.1 Python methods 27
A.2 TCP commands 28
A.3 Command-line options 28

1 What synth is

`synth` turns a Red Pitaya STEMlab 125-14 into a **multi-tone signal source**. Each output is

$$\mathrm{OUT}n(t) = \mathrm{offset}_n + (1 + d_n V_{\mathrm{in}}(t)) \sum_{k=1}^{16} A_{n,k} \sin 2\pi \Big(f_{n,k}\,(t - t_0) + \frac{\varphi_{n,k}}{360^\circ} \Big), \qquad n = 1, 2, \tag{1}$$

evaluated at every sample (125 MSa/s) and rounded once to the 14-bit DAC. The factor $1 + d_n V_{\mathrm{in}}$ is the optional amplitude modulation from an input (section 8); it is 1 unless you switch it on.

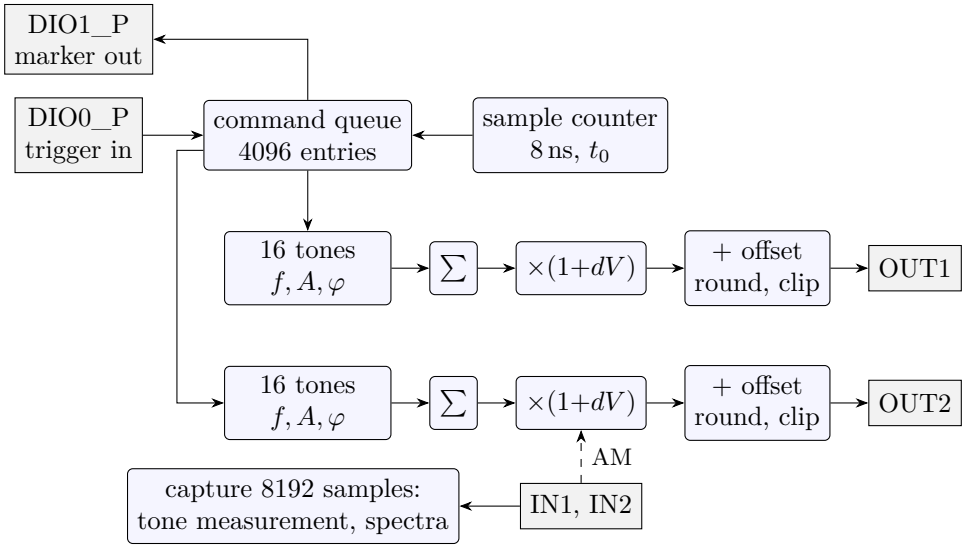


Figure 1: What is inside. Every block runs on one 125 MHz sample clock: both outputs, the queue that changes them, the trigger, the marker and the input capture share one time base. Amplitude modulation (dashed) can come from either input on either output.

connector	signal
OUT1, OUT2	16 tones each, on one shared time base
DIO0_P (E1)	trigger input (3.3 V logic, pull-down): starts a waiting sequence
DIO1_P (E1)	marker output (3.3 V logic): a pulse when a marked step takes effect
IN1, IN2	capture of 8192 samples on the same time base (tone measurement, spectra, calibration), or the source of amplitude modulation

What sets it apart:

- **Phases mean something.** Every phase refers to one zero time t_0, the same for both outputs. Switching a tone $f_1 \to f_2 \to f_1$ returns it to the phase it would have had without the excursion, and equal settings on OUT1 and OUT2 are in phase sample for sample.
- **Changes are atomic.** Any set of tones on both outputs changes on one sample.
- **Fine steps.** Frequency in steps of $125\,\mathrm{MHz}/2^{48} = 0.44\,\mu\mathrm{Hz}$, amplitude in 2^{-17} of full scale, phase in 2^{-16} of a cycle.
- **Sequences in the FPGA.** Steps and ramps (linear, $\cos^2$, Blackman, minimum jerk) at exact sample times from a 4096-entry queue, started by a trigger and repeated by the FPGA itself if you wish; longer sequences are streamed by the board's service.

- **Real volts.** Amplitudes and offsets are volts at the SMA connectors, through the board calibration shared by every design in the repository.

It was inspired by the Simon lab's "Multi-Chromatic DDS" for the same board (2017), which has 10 tones on OUT1 only, 32-bit frequency words and no phase control.

1.1 How a tone is made, in one page

Each tone is a **direct digital synthesizer** (DDS). A 48-bit phase accumulator adds the frequency word $F = f \cdot 2^{48}/f_s$ at every sample (8 ns); its upper bits are the phase in cycles, and a sine table with a first-order correction turns the phase into the sine's value, accurate to about half a step of 2^{-17} (figure 2). The value is multiplied by the tone's amplitude, the 16 tones of an output are added exactly, and the sum is rounded *once* to the 14-bit DAC, which holds it for 8 ns.

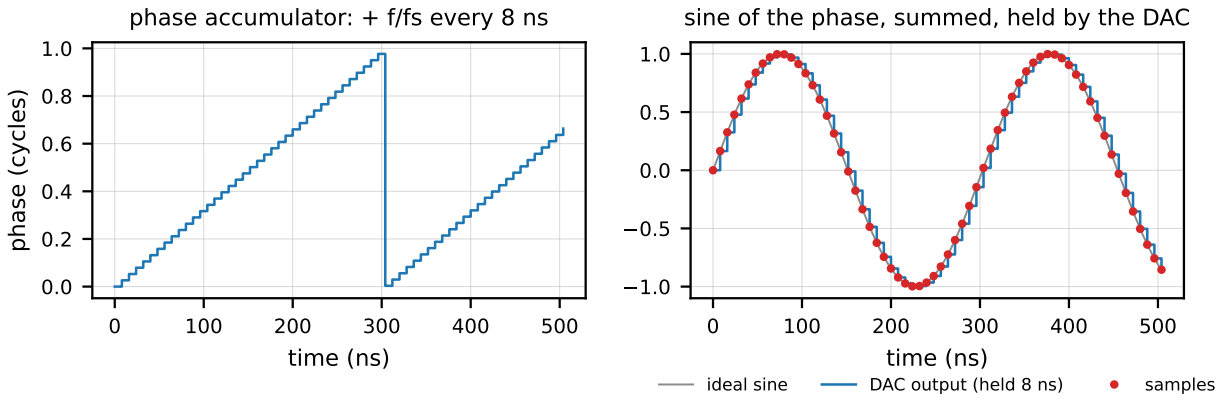


Figure 2: Computed: a 3.3 MHz tone. Left: the phase accumulator, advancing by f/f_s of a cycle every sample and wrapping. Right: the sine of the phase at each sample, and the DAC's output, which holds each value for 8 ns.

Three consequences follow. **Phases are exact:** the accumulator is an integer, so the phase after any number of samples is known to the bit; setting it to $F(n - t_0)$ when a tone changes makes its phase relative to t_0 exact, which is what coherent mode does. **Frequencies are exact relative to the board clock,** in steps of $f_s/2^{48}$. **Changes are exact in time:** a command queue in the FPGA changes any set of tones on one chosen sample, ramps add a fixed increment every sample for a given number of samples, and the sample counter that times them is the same one that numbers the input samples. The details are in `docs/THEORY.md`.

2 Before you connect equipment

Read the repository's section "Connecting to lab equipment: read first" in the root `README.md`. For synth in particular:

Levels

The set amplitude is the level **into 50 Ω**. The outputs behave like a 50 Ω source, so into a high-impedance load (a scope at 1 MΩ) you get about twice it. The largest single tone is about 1.13 V (+11 dBm) into 50 Ω on the board used here. By default the driver refuses any setting whose worst case $|\text{offset}| + \sum A$ leaves the DAC range; with `allow_clip` it accepts it, and the output saturates on peaks. Connect only devices that can take +11 dBm, or about ±2.3 V into high impedance.

set amplitude (V peak)	into 50 Ω (dBm)	into 50 Ω (V_{pp})	into 1 MΩ (V_{pp}, about)
1.13	+11.1	2.26	4.5
0.5	+4.0	1.0	2.0
0.25	−2.0	0.5	1.0
0.1	−10.0	0.2	0.4
0.05	−16.0	0.1	0.2
0.01	−30.0	0.02	0.04

In general $P\,[\mathrm{dBm}] = 20\log_{10}(A/\mathrm{V}) + 10$ into 50 Ω. With several tones the powers add: 16 tones of 0.05 V carry $-16 + 12 = -4$ dBm in total.

- **Loading synth leaves the outputs quiet:** all tones off, outputs at 0 in converter units (a few mV from 0 V before calibration). After a reboot the control service restores the last configuration, tones included.
- **Steps are steps.** A change without a ramp happens on one sample. Use ramps for devices that dislike sudden changes (acousto-optic deflectors, amplifiers near compression).
- **The extension-connector pins are 3.3 V LVCMOS** and not 5 V tolerant. Drive DIO0_P from a 3.3 V source (or through a divider); never drive DIO1_P from outside.
- **Images.** Each sample is held for 8 ns, so a tone at f also appears at 125 MHz − f, filtered only by the board's analog front end. Add a low-pass filter for the band you use, especially above 40 MHz.

Safe state

`SAFE` (Python `synth.safe()`) mutes every tone on the same sample, stops any sequence and puts both outputs at the calibrated 0 V. The notebook panel has a **SAFE** button.

3 Setting up

The board must be prepared once as described in `common/SETUP.md`: the PYNQ image, the network (here 192.168.0.149), and, on a Linux computer, the repository. Then, in `synth/`:

```
make upload            # the FPGA design, the Python package and the notebook
make install-service   # the control service (TCP port 5025), with synth active
make test-hw           # a loopback self-test: OUT1 -> IN1 and OUT2 -> IN2 cabled
```

`make test-hw` checks amplitudes, phases, coherent switching, sequences, amplitude modulation and `SAFE` with the board's own inputs, and prints one PASS or FAIL line per check. Building the FPGA design yourself needs Vivado and is described in the design's `README.md`; the steps above need only the files in the repository.

4 Getting started

4.1 A first signal in five lines

On the board (Jupyter, or Python over SSH):

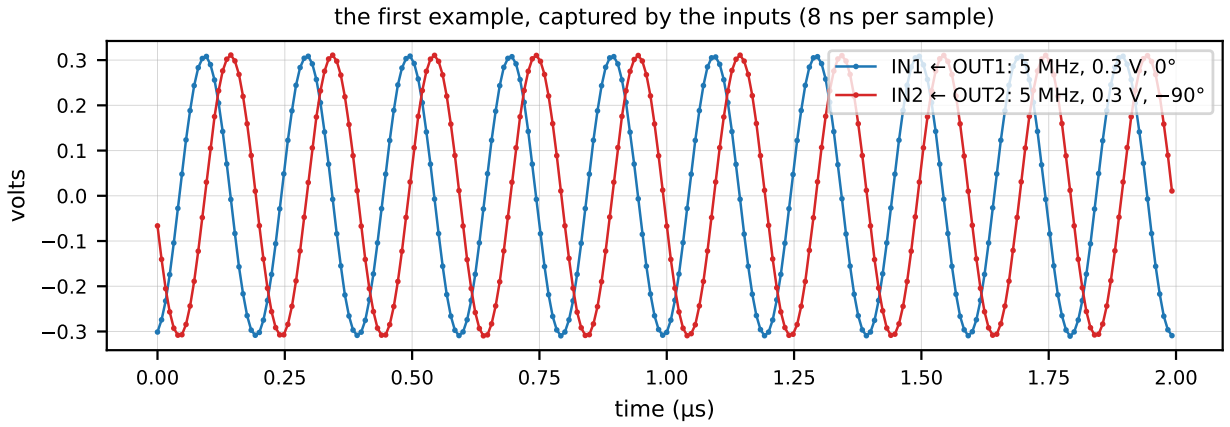


Figure 3: Measured: the example above, cabled OUT1 → IN1 and OUT2 → IN2 through 50 Ω terminators and captured by the board's inputs. The lock-in reads 0.309 V and 0.310 V (the loop gain at 5 MHz is slightly above 1, figure 5) and 89.83° between the inputs.

```
import rpfpga                                   # must come first
from pynq.overlays.synth import DRIVER
synth = DRIVER()                                # loads synth if the board runs another design
synth.set_tone(1, 1, 5e6, 0.3)                  # OUT1 tone 1: 5 MHz, 0.3 V
synth.set_tone(2, 1, 5e6, 0.3, -90)             # OUT2 tone 1: 5 MHz, 90 degrees behind OUT1
print(synth.describe())
```

TCP: `TONE 1.1 5e6 0.3`
TCP: `TONE 2.1 5e6 0.3 -90`

The two outputs are now in quadrature: OUT2 lags OUT1 by exactly a quarter period on every sample (figure 3). `describe()` prints the values actually applied:

```
{1: 'OUT1 = 1 tone: 5000000 Hz 0.3 V +0.00 deg', 2: 'OUT2 = 1 tone: 5000000 Hz 0.3 V -90.00 deg'}
```

4.2 The four ways to control synth

Jupyter

Open `http://192.168.0.149:9090` (password `xilinx`), then `synth/synth.ipynb`. The control panel (`make_gui(synth)`) has the tabs **Tones** (two tables of 16 tones, offsets, dither, amplitude modulation, phase mode, **Zero time**, **SAFE**), **Presets** and **Calibration**.

Python on a PC

The client needs only the Python standard library:

```
import sys; sys.path[:0] = ["common/python", "synth/control"]   # from the repository root
from synth import SynthClient
with SynthClient("192.168.0.149") as synth:
    synth.set_tone(1, 1, 5e6, 0.3)
    print(synth.tones())
```

It has the same methods as the driver; every example in this manual works with either.

TCP

Anything that can open a TCP connection: `nc 192.168.0.149 5025`, then one command per line, for example `TONE 1.1 10e6 0.25`, `TONES?`, `HW?`, `SAFE`. Replies are `OK <value>` or `ERR <message>`; values are JSON.

Command line

The repository's common command line:

```
PYTHONPATH=common/python:synth/control python3 -m rpfpga.cli --host 192.168.0.149 \
    --tone 1.1=5e6,0.3 --tone 2.1=5e6,0.3,-90
```

Several programs may control the board at once (a notebook, a script, the service): each change is one atomic transaction, and every reader sees the settings actually applied.

4.3 The notebook panel

`make_gui(synth)` shows three tabs.

Tones
: Two tables, OUT1 and OUT2, of 16 rows: frequency (Hz), amplitude (V) and phase (degrees). Below each table: the offset (V), **Dither**, **AM by** (off, in1, in2) with its **depth** (1/V), and **Apply OUT**n, which sends the edits of that output together, on one sample. Under both tables: the phase mode (coherent or continuous), **Zero time** (t_0 := the next change), **SAFE (all off)** and **Refresh from hardware**, which reads what the board runs (for example after a script changed it). The line below shows the hardware counters.

Presets
: Load a built-in or saved preset; save the present settings under a name.

Calibration
: The board calibration of all designs: input ranges, the loopback calibration with an optional DMM reading, and the stored constants.

The panel and any script may be used at the same time; **Refresh from hardware** brings the panel up to date.

5 Tones

5.1 Example: a five-tone comb

```
synth.apply({"outputs": {"1": {"offset": 0.0, "tones": {
    str(k): {"frequency": k * 1e6, "amplitude": 0.1, "phase": 0.0} for k in range(1, 6)}}}})
print(synth.config()["outputs"]["1"]["tones"]["3"])
```

TCP: `CONFIG <json>, with the same dictionary as JSON`

A configuration holds, per output, 16 tones (`frequency` in Hz, `amplitude` in volts peak, `phase` in degrees), an `offset` (volts), `dither` and `am`; and, for both outputs, `phase_mode`, `t0`, `droop_correction`, `allow_clip`, `apply_delay` and `marker_width`. `apply()` takes any part of it and keeps the rest (`replace=True` starts from all tones off).

setting	range	resolution
frequency	0 to 62.5 MHz; keep below about 50 MHz for clean spectra	0.44 μHz
amplitude	0 to about 1.13 V (the calibrated full scale)	2^{-17} of full scale (8.6 μV)
phase	any; reported in $-180° \ldots 180°$	0.0055°
offset	the DAC range minus the tones' peak	one DAC step (138 μV)

Returned values are the realized ones: the frequency of the 48-bit word, the amplitude of the 17-bit code, the phase of the 16-bit word. Use them in your analysis.

Frequency accuracy is that of the board's 125 MHz crystal, which is not disciplined: the realized frequencies are exact relative to it. Its absolute accuracy and phase noise were not measured here.

5.2 Headroom and crest factor

The driver checks that $|\text{offset}| + \sum_k A_k$ stays within the DAC range, the worst case when all tones peak together. Sixteen equal tones are then limited to about 70 mV each. Their actual peak depends on the phases (figure 4): with equal phases they do add up once per period; with **Schroeder phases**, $\varphi_k = -180°k(k-1)/16$ (the preset `schroeder16`), the peak is only about 1.91 times the RMS instead of 4.22. Set `allow_clip` to use the room this gives; the hardware then saturates (never wraps) any rare peak and counts the saturated samples (`hardware()["counts"]["clipped_samples"]`).

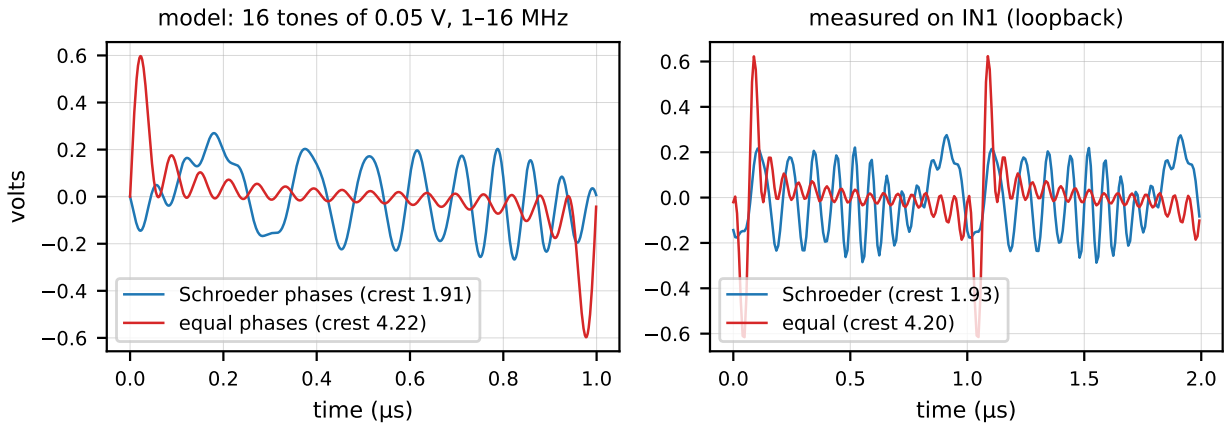


Figure 4: Sixteen tones of 0.05 V from 1 to 16 MHz. Left: computed, one period. Right: measured on IN1 in loopback, crest factors 1.93 (Schroeder) and 4.20 (equal phases). The measured equal-phase peak is lower than the computed one because the loop's response and delay differ slightly between the tones.

5.3 Droop correction and the output response

Each DAC sample is held for 8 ns, which weakens high frequencies by $\text{sinc}(f/f_s)$: −0.36 dB at 20 MHz, −2.4 dB at 50 MHz. **Droop correction** (on by default) divides each amplitude by it. The output amplifier and filter add their own response; figure 5 shows the measured loop OUT → IN. If you store a measured output response in the board calibration (section 9.2), synth corrects it as well.

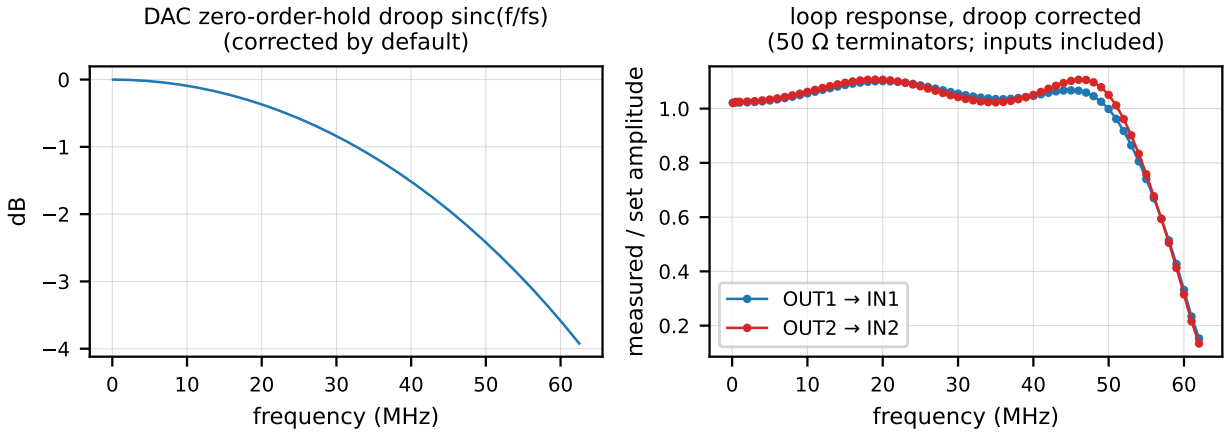


Figure 5: Left: the DAC's zero-order-hold droop, corrected by default. Right: measured amplitude over set amplitude through the loop OUT → IN with 50 Ω terminators, droop corrected (OUT1: 1.02 at 1 MHz, 1.10 at 20 MHz, 1.00 at 50 MHz, 0.33 at 60 MHz). The inputs' response is included; the fall above 50 MHz is probably mostly theirs.

Dither adds a triangular random signal of ±1 DAC step before the single rounding to 14 bits. Without it the rounding error of a tone at a simple fraction of 125 MHz (10 MHz is 2/25 of it) repeats

with a short period and forms spurs, here −96 dBc; with dither they become a flat floor, below −113 dBc per bin (figure 6). Both are far below what the analog outputs add (section 10), so dither matters only for the cleanest narrow-band work. At most other frequencies the error does not repeat and spreads out by itself.

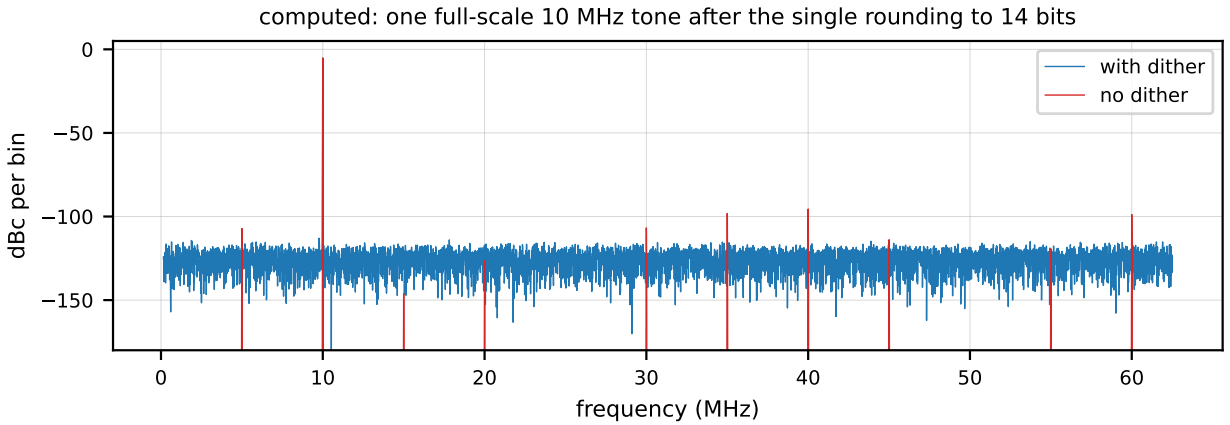


Figure 6: Computed with the bit-exact model: the output codes of one full-scale 10 MHz tone (2^{15} samples, Kaiser window), without dither (spurs) and with it (a flat floor). The analog output adds its own, larger distortion (figure 15).

5.4 All settings

key	default	meaning
per output (`"outputs": {"1": {...}, "2": {...}}`)		
`tones."k".frequency`	0	Hz, 0 to 62.5 MHz
`tones."k".amplitude`	0	volts peak into 50 Ω
`tones."k".phase`	0	degrees, relative to t_0
`offset`	0	volts
`dither`	false	triangular dither of ±1 DAC step
`am.source`, `am.depth`	`"off"`, 0	amplitude modulation by `"in1"` or `"in2"`, depth in 1/V
both outputs		
`phase_mode`	`"coherent"`	or `"continuous"`
`t0`	0	the zero time, a sample index (8 ns each)
`droop_correction`	true	correct sinc droop and the stored output response
`allow_clip`	false	accept settings whose worst-case peak leaves the DAC range
`apply_delay`	0.01	seconds from `apply()` to the change (coherent mode)
`marker_width`	10^{-6}	seconds of the DIO1_P marker pulse

Tones are numbered 1 to 16 (keys `"1"` … `"16"`); a tone with amplitude 0 is off. `config()` returns all of it, `describe()` a one-line summary per output.

5.5 Presets

name	outputs
`off`	all tones off, 0 V (the default)
`10mhz`	10 MHz, 0.5 V on both outputs, in phase
`quadrature`	1 MHz, 0.5 V on both; OUT2 90° behind OUT1
`comb5`	OUT1: 1, 2, 3, 4, 5 MHz, 0.1 V each, in phase at t_0
`schroeder16`	OUT1: 1 to 16 MHz, 0.12 V each, Schroeder phases; `allow_clip`

`synth.load_preset("comb5")`; save your own with `synth.save_preset("name")`.

TCP: `PRESET comb5`

TCP: `PRESET SAVE myname`

6 Phase and the zero time

6.1 Example: coherent switching

```
synth.set_tone(1, 1, 10e6, 0.3)
synth.set_tone(1, 1, 11e6)            # 11 MHz, with the phase 11 MHz would have had since t0
synth.set_tone(1, 1, 10e6)            # back: exactly the phase of an uninterrupted 10 MHz
```

Every tone's phase is $\varphi + 360°\,f\,(t - t_0)$, where t_0 is a sample index of the FPGA's 125 MHz counter (8 ns per sample), shared by both outputs.

- **Coherent mode** (default). Each `apply()` takes effect `apply_delay` (10 ms) after the call, on one sample, and sets every tone's phase from t_0 exactly. Whatever a tone did before, its phase afterwards is as if it had always run at its present frequency (figure 7).
- **Continuous mode** (`"phase_mode": "continuous"`). Changes take effect at once and the tones' phases continue: no phase jumps at frequency changes, but phases relative to t_0 lose their meaning.
- **Zero time.** `synth.zero_time()` makes the next change's sample the new t_0: every tone then starts with exactly its phase setting there. $t_0 = 0$ is the FPGA's reset; after a reload of the FPGA the counter starts again from 0.

  TCP: `ZERO`

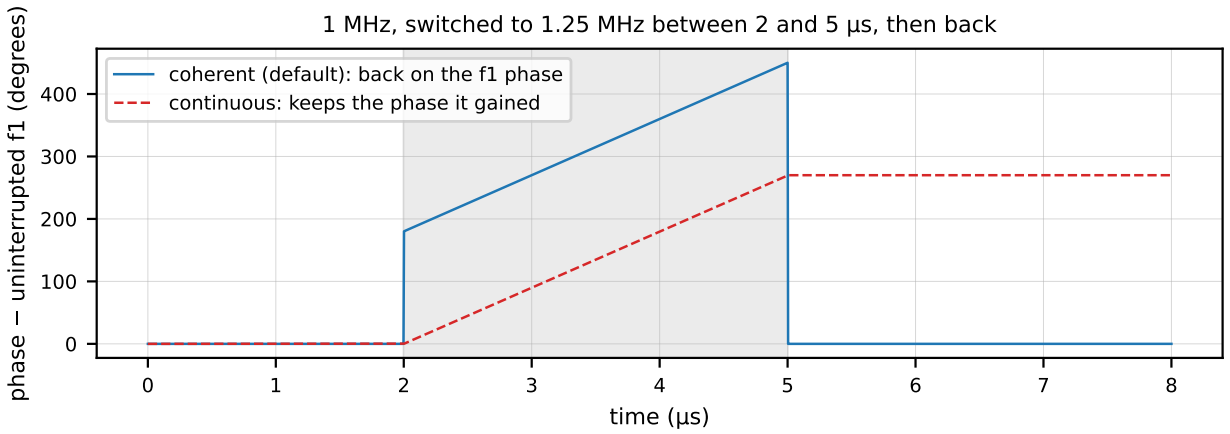


Figure 7: Computed: the phase of a 1 MHz tone relative to an uninterrupted 1 MHz, when it is switched to 1.25 MHz for 3 μs. In coherent mode it returns to the phase it would have had; in continuous mode it keeps the 270° it gained.

6.2 Phases at the connectors

Digitally, equal settings on OUT1 and OUT2 are in phase on every sample. At the connectors the DAC and the output amplifiers add a small relative delay τ, which shows as $-360°\,f\tau$. On the board used here OUT2 is 0.14 to 0.20 ns *earlier* than OUT1 between 1 and 40 MHz (figure 8, right). Its calibration holds −0.165 ns for OUT2, so synth corrects every phase on OUT2 by $360°\,f\tau$; the residual is about ±0.03 ns (±0.1° at 10 MHz). Section 9.3 shows how to measure it.

How it is verified

`make test-hw` checks phase steps of +90°, −45° and +170° (within 0.003° on 2026-10-22), returning to a frequency after an excursion (within 0.005°), and the phases of a five-tone comb against each tone alone (within 0.08°).

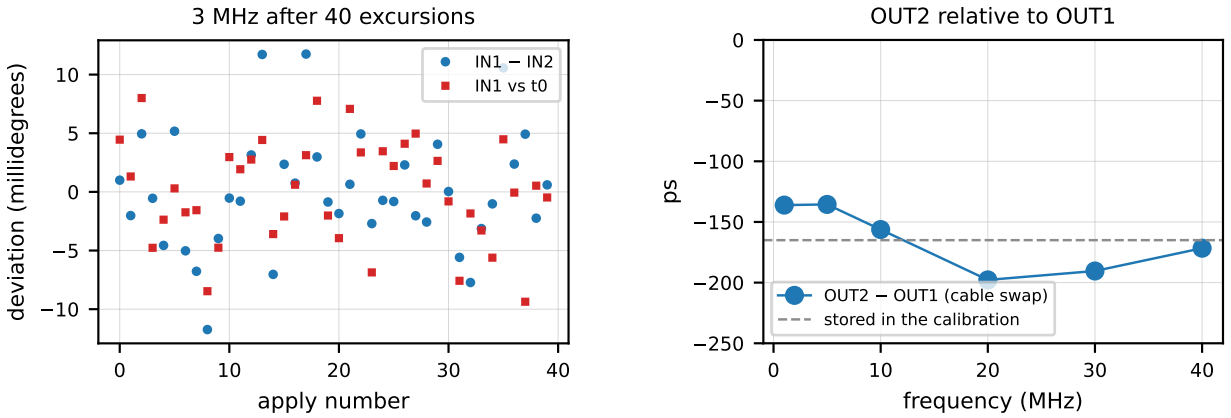


Figure 8: Measured. Left: a 3 MHz tone on both outputs (OUT2 set 90° behind), re-applied 40 times, each after an excursion to another frequency. The phase relative to t_0 scatters by 4.3 m° RMS and the IN1 − IN2 difference by 4.9 m°: the inputs' noise. Right: how much later OUT2 is than OUT1, by the cable-swap method of section 9.3.

7 Sequences

7.1 Example: a step, a chirp and a smooth turn-off

```
synth.play({"steps": [
    {"at": 0.0,  "tones": {"1.1": {"frequency": 40e6, "amplitude": 0.2}}},
    {"at": 1e-3, "ramp": 2e-4, "tones": {"1.1": {"frequency": 41e6}}, "marker": True},
    {"at": 2e-3, "ramp": 1e-4, "shape": "cos2", "tones": {"1.1": {"amplitude": 0.0}}}]})
print(synth.sequence_status())
```

TCP: `PLAY <json>, with the same dictionary as JSON`

`play()` compiles the steps into entries of the FPGA's command queue and starts them a few milliseconds after the call. From then on the FPGA alone decides when each step happens, to the sample.

7.2 Steps and ramps

- Tone keys are `"<output>.<tone>"`; settings a step does not name keep their values. `"outputs"` changes an output's `offset`, `dither` and `am` at the step.
- Times (`at`, seconds) count from the sequence's start, or from the trigger edge.
- A step without `ramp` changes on one sample. In coherent mode it sets the phase relative to t_0, exactly as `apply()` does.
- A **ramp** moves frequency and amplitude in `ramp` seconds, phase-continuously, and lands **exactly** on the new values: at its end the frequency and amplitude words equal the targets to the last bit. A phase given in a ramped step is a phase step at its start. Figure 9 shows a measured chirp.
- `"marker": true` pulses DIO1_P on the sample where the step takes effect (`marker_width`, 1 μs by default).
- Other tones may change during a shaped or long ramp (its pieces are queued in time order with the other steps); the ramping tone itself not until the ramp's end.
- `stop()` leaves a running ramp running. If a new sequence's first change of such a tone is a ramp, the tone is first stepped to the old ramp's target, because the new ramp is computed from there.

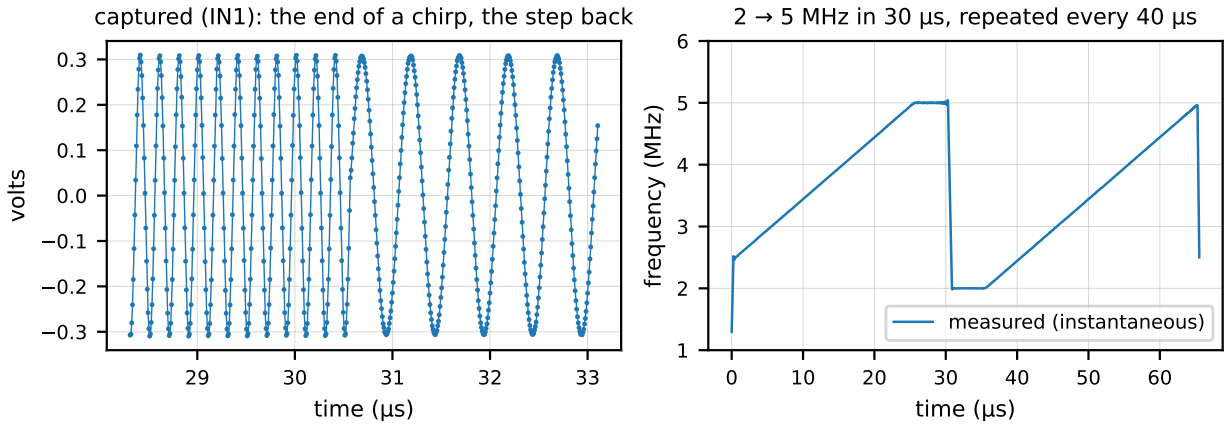


Figure 9: Measured: a chirp from 2 to 5 MHz in 30 μs, then a step back to 2 MHz, repeated every 40 μs by the FPGA (section 7.5). Right: the instantaneous frequency of the captured signal; the ramp is straight and phase-continuous, the step back is immediate.

7.3 Shaped ramps

`"shape"` is `linear` (default), `cos2`, `blackman` or `min-jerk` (figure 10). A shaped ramp is made of `segments` (16 by default, at most 256) linear pieces that land exactly on the curve. With 16 pieces the curve is followed within 0.24 % ($\cos^2$), 0.28 % (minimum jerk) and 0.41 % (Blackman) of the total change. Each piece is one APPLY, which needs about 0.5 μs plus 8 ns per changed word; short ramps therefore get fewer pieces automatically (about 0.6 μs per piece for one tone).

`cos2`
: $(1 - \cos \pi x)/2$: the usual smooth edge for pulses; zero slope at both ends.

`blackman`
: the integral of a Blackman window: still smoother edges, the lowest spectral sidelobes of the four (figure 17).

`min-jerk`
: $x^3(10 - 15x + 6x^2)$: zero velocity and acceleration at both ends, the standard profile for moving trapped atoms with optical tweezers.

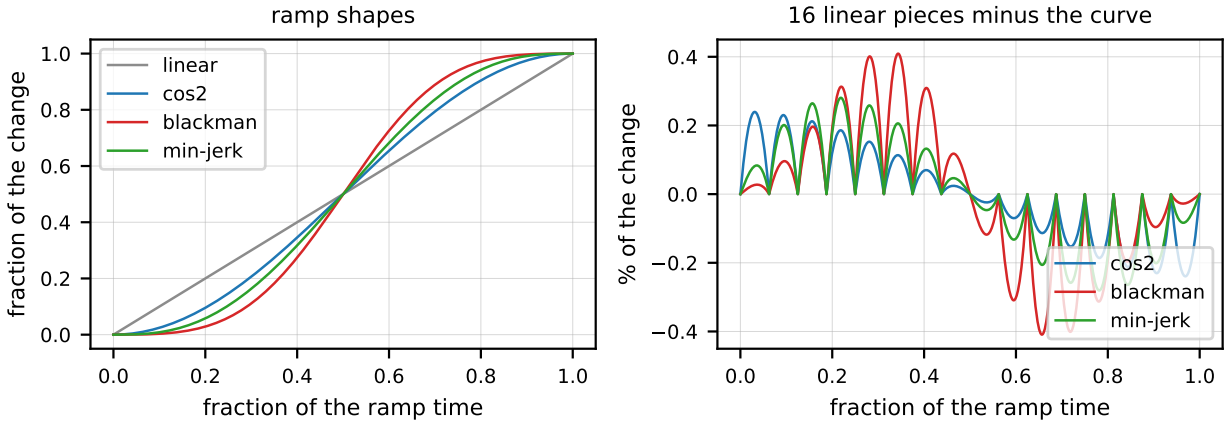


Figure 10: Computed: the four ramp shapes, and how far the 16 linear pieces that the FPGA plays deviate from the true curves.

7.4 Triggers and markers

With `"trigger": true` the sequence waits for a rising edge on DIO0_P (or `synth.trigger()`), and the step times count from the edge:

```
synth.play({"trigger": True, "steps": [
    {"at": 100e-6, "tones": {"2.1": {"frequency": 15e6, "amplitude": 0.4}}, "marker": True},
    {"at": 120e-6, "tones": {"2.1": {"amplitude": 0.0}}}]})
synth.trigger()                       # or a 3.3 V edge on DIO0_P
```

TCP: `TRIG`

The edge is also the sequence's t_0: every run starts with the same phases relative to the trigger, within the 8 ns sampling of the asynchronous input. The first step must be at least about 0.15 μs after the edge. Figure 11 shows the timing.

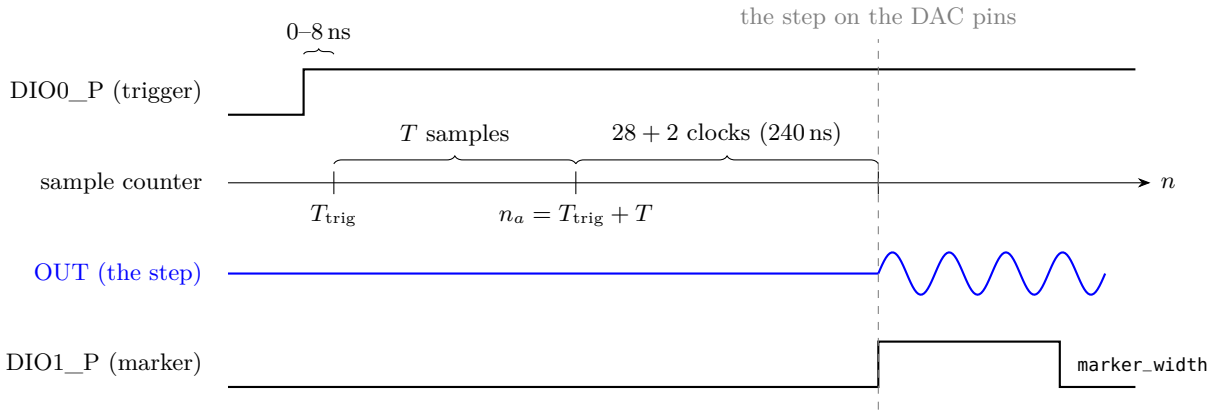


Figure 11: Trigger and marker timing (not to scale). T_{trig} is the first sample clock that sees the edge. A step at T samples after the trigger reaches the DAC pins $T + 30$ clocks after T_{trig}; the marker rises on the same DAC clock edge. Wired back to DIO0_P, the marker of a step at sample n_a is seen as a trigger at $n_a + 31$ (measured, every time, for markers from 8 ns to 1 μs).

7.5 Sequences repeated by the FPGA

`"repeat": K` plays the steps K times (0: until `stop()`), `"period"` seconds apart:

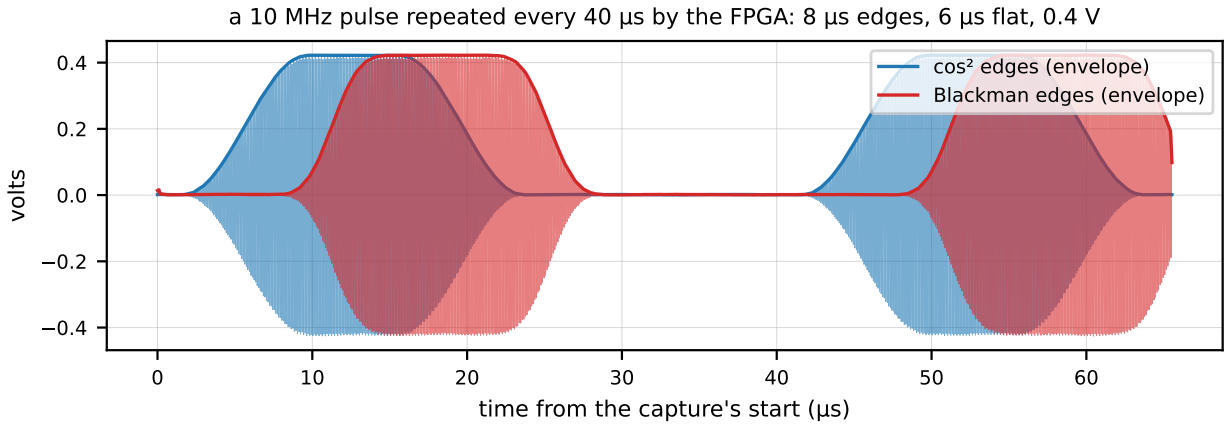


Figure 12: Measured: a 10 MHz pulse of 0.4 V with 8 μs edges and a 6 μs flat top, repeated every 40 μs by the FPGA, once with cos^2 and once with Blackman edges. The thick lines are the envelopes of the captured signals.

```
synth.play({"repeat": 1000, "period": 100e-6, "steps": [
    {"at": 0.0, "tones": {"1.1": {"frequency": 20e6, "amplitude": 0.0}}},
    {"at": 1e-6, "ramp": 4e-6, "shape": "cos2", "segments": 4,
     "tones": {"1.1": {"amplitude": 0.3}}, "marker": True},
    {"at": 7e-6, "ramp": 4e-6, "shape": "cos2", "segments": 4,
     "tones": {"1.1": {"amplitude": 0.0}}}]})
```

The queue rewinds by itself, so there is no host traffic and no jitter between repetitions: repetition r starts exactly r periods after the first. In coherent mode the phases refer to the start of each repetition, so all repetitions are identical (figure 12). With a trigger, each repetition waits for its own edge. The whole sequence must fit in the queue (at most 4095 entries), every tone's first change in it must be a step (so that each repetition starts from the same state), and the period must exceed the last step.

7.6 Limits

quantity	limit
step spacing	0.5 μs plus 8 ns per changed word (a one-tone change: 2–6 words)
first step after a trigger	about 0.15 μs
chirp rate	119 kHz/μs; faster changes must be steps
amplitude ramp	full scale in 8 μs at the fastest
ramp length	up to 3600 s (split into pieces of at most 134 ms automatically)
queue	4096 entries; a one-tone step costs about 6, so about 680 steps
repeated sequences	must fit in the queue (4095 entries); first change of each tone a step
streaming (longer sequences)	about 60 000 entries/s, i.e. one-tone steps every 0.1 ms (measured for 3 s); about 20 000/s while another client polls `STATUS?` without pause
compiling	about 0.85 ms per step on the board (`play()` of 6000 steps: 5 s)

7.7 Reference: the sequence dictionary

key	default	meaning
`trigger`	false	wait for a DIO0_P edge; times count from it, and it is t_0
`phase_mode`	`"coherent"`	steps set phases from t_0; `"continuous"`: phases run on
`repeat`	1	play the steps this many times; 0: until `stop()`
`period`	0	seconds between repetitions (without a trigger)
`steps`	—	the list of steps, in time order
each step		
`at`	—	seconds from the start (or the trigger, or the repetition's start)
`ramp`	0	seconds; 0 is a step
`shape`	`"linear"`	`"cos2"`, `"blackman"`, `"min-jerk"`
`segments`	16	linear pieces of a shaped ramp (1 to 256; fewer if too short)
`marker`	false	pulse DIO1_P when the step takes effect
`tones`	{}	`{"o.k": {"frequency", "amplitude", "phase"}}`; missing settings keep their values
`outputs`	{}	`{"o": {"offset", "dither", "am"}}`

7.8 Long sequences

The queue holds 4096 entries. Longer sequences need the control service (`make install-service`): its worker pushes the rest as space frees up. If the host falls behind, the late step **halts** the sequence (state `halted`) and the outputs keep the last applied settings: nothing is ever played at a wrong time. A long `play()` holds the board while it compiles; other clients wait meanwhile. Over TCP, sequences above 32 kB of JSON are sent compressed (`PLAYZ`) by the client; very long ones (above about 6000 steps) are best played on the board itself.

7.9 During and after a sequence

`sequence_status()` reports `none`, `queued`, `streaming`, `done`, `stopped` or `halted` (TCP `SEQ?`). `config()` always reads the settings the tones run on, also during a sequence and after `stop()` (TCP `STOP`), so a later `apply()` continues from what the outputs carry. `apply()` during a sequence stops it; `play()` right after `apply()` starts after the pending change. `hardware()` (TCP `HW?`) shows the queue level, the counts of APPLYs, late APPLYs, triggers and clipped samples.

8 Amplitude modulation from an input

8.1 Example

```
synth.set_tone(1, 1, 25e6, 0.2)
synth.set_am(1, "in2", depth=1.0)       # OUT1 = (1 + 1/V * V_IN2) * the 0.2 V tone
synth.set_am(1, "off")
```

TCP: `AM 1 IN2 1.0`
TCP: `AM 1 OFF`

Each output's tones can be multiplied by $1 + d\,V_{\text{in}}(t)$, where V_{in} is IN1 or IN2 in calibrated volts and d the depth in 1/V (equation 1). The offset is not modulated. Figure 13 shows it measured: the factor follows the input linearly, with a measured slope of 1.000/V for $d = 1/\text{V}$.

- The factor stays within 0 to 2 while $|d\,V_{\text{in}}| \le 1$; beyond, it saturates at 0 or 2. The depth's range is about ±1.8/V on LV and ±0.1/V on HV.
- The headroom check uses the largest factor the input can produce.
- Inside the FPGA an input sample reaches the output 8 samples (64 ns) later; the converters and the analog input and output paths add their own delays (not measured).

- Sequences can switch it per step: `"outputs": {"1": {"am": {"source": "in2", "depth": 0.3}}}`.

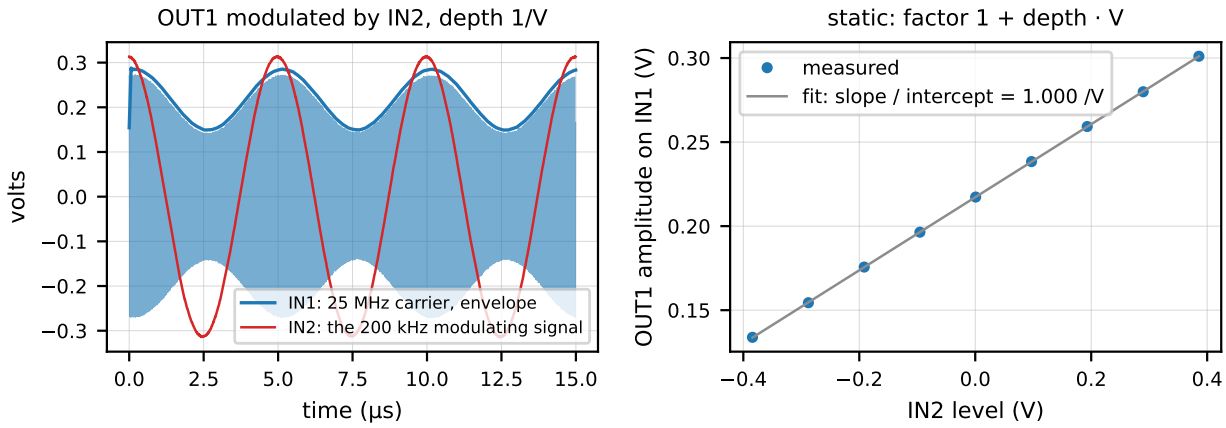


Figure 13: Measured: OUT1 (25 MHz, 0.2 V) modulated by IN2 with $d = 1/\mathrm{V}$, cabled OUT1 → IN1 and OUT2 → IN2. Left: OUT2 carries a 200 kHz tone of 0.3 V into IN2; the carrier's envelope follows it. Right: OUT2 at DC levels from −0.4 to +0.4 V; the measured carrier amplitude is a straight line with slope over intercept 1.000/V.

Uses: an intensity noise eater (a photodiode signal on the input, a negative depth), an amplitude servo driven by an external controller, a modulated test signal.

9 Calibration

All designs in this repository share one **board calibration** (`common/CALIBRATION.md`): the gains and offsets of both inputs and outputs, measured in loopback (optionally with a DMM), and two optional AC fields per output: the frequency response and the delay. synth can measure all of them.

9.1 DC calibration

Cable OUT1 → IN1 and OUT2 → IN2 (LV jumpers), then run `synth.calibrate()`. For absolute accuracy, hold the reference (`synth.hold_reference()`), read both outputs with a DMM, and pass the readings: `synth.calibrate(dmm={"out1": V1, "out2": V2})`. The Calibration tab of the notebook guides through the steps. Without a calibration, nominal constants are used, which on the board used here are about 14 % off in amplitude.

TCP: `CAL LOOPBACK`

TCP: `CAL?`

9.2 Output frequency response

The DC calibration makes a 0.5 V setting right at low frequencies; at higher frequencies the output amplifier and filter roll off. A response table corrects that:

```
for n in (1, 2):
    table = synth.measure_response(n)          # [[Hz, gain], ...], 1 to 60 MHz, in loopback
    synth.set_output_response(n, table)        # stored in the board calibration
```

TCP: `CAL RESPONSE OUT1 [[1e6, 1.0], [2e7, 1.1], [6e7, 0.33]]`

`measure_response` measures the loop OUTn →INn, so the table includes the input's own response (figure 5), as the DC calibration includes the input's gain. On the board used here it was *not* stored, because most of the fall above 50 MHz is probably the inputs'. With a power meter or a calibrated scope you can measure the outputs alone and store your own table the same way; `set_output_response(n, [])` removes it.

9.3 Output delay (OUT1–OUT2 skew)

Use two equal cables:

```
from synth import measure                          # on the board: from pynq.overlays.synth import
    ↪ measure
straight = synth.phase_difference(10e6)            # OUT1 -> IN1, OUT2 -> IN2
# now swap the cables at the OUTPUTS: OUT1 -> IN2, OUT2 -> IN1
swapped = synth.phase_difference(10e6)
lag = measure.skew(straight, swapped)              # seconds: how much later OUT2 is than OUT1
cal = synth.calibration()
synth.set_output_delay(2, cal["out2"].get("delay_s", 0.0) + lag)
```

TCP: `CAL DELAY OUT2 -1.65e-10`

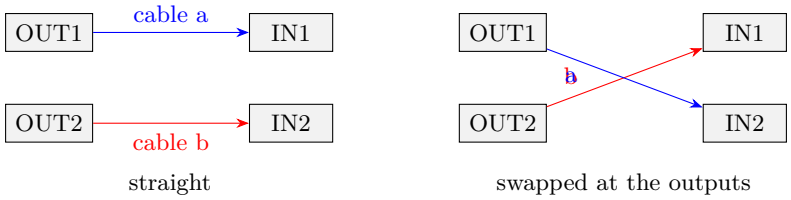


Figure 14: The cable swap. Each cable stays on its input; only the outputs change places.

With OUTi delayed by τ_i and the paths into INj by ϵ_j (cables and inputs), the two phase differences IN1 − IN2 are $-360°\,f(\tau_1 - \tau_2 + \epsilon_1 - \epsilon_2)$ and $-360°\,f(\tau_2 - \tau_1 + \epsilon_1 - \epsilon_2)$; their difference is $720°\,f(\tau_2 - \tau_1)$, free of the inputs and cables. synth then adds $360°\,f\tau_n$ to each phase of output n, so equal phase settings are in phase at the connectors. The result is unambiguous within ±25 ns at 10 MHz; check it at a few frequencies (figure 8, right).

10 Measuring with the inputs

The inputs record 8192 samples (65.5 µs) on the same time base as the tones. In loopback the board checks its own output.

10.1 Tones

```
result = synth.measure_tones()
for ch, data in result["inputs"].items():
    for k, t in data["tones"].items():
        print(ch, k, "%.4f V (set %.4f), %+.2f deg (set %+.2f)" % (
            t["amplitude"], t["set_amplitude"], t["phase"], t["set_phase"]))
```

TCP: `MEASURE?`

`measure_tones()` measures every configured tone with a lock-in (Blackman–Harris window): amplitude, and phase relative to t_0, per input (OUT1 → IN1 and OUT2 → IN2 by default). The phase differs from the setting by the loopback delay, $-360°\,f\tau$ (about 349.2 ns with short cables and terminators here): a constant for each frequency. Repeat after any change: the difference stays the same, which shows phase coherence (figure 8). Tones closer than about 100 kHz bias each other. Once a step of a trigger sequence has run, the phases refer to the trigger edge; in a repeating sequence, to the start of the repetition in which the capture starts; after the next coherent `apply()`, to t_0 again.

10.2 Spectra

```
from synth import measure
f = measure.bin_frequency(10e6)                  # on the FFT grid: no leakage of its own
synth.set_tone(1, 1, f, 0.4)
lines = measure.analyze(synth.spectrum("in1"), [f])
print(lines["fundamentals"], lines["harmonics"], lines["spur"], lines["floor_dbc"])
```

`spectrum()` returns the input's amplitude spectrum in dB re 1 V peak per 15.26 kHz bin, and `analyze()` reads the fundamentals, their harmonics, the largest other line and the floor. Figure 15 shows three measured spectra.

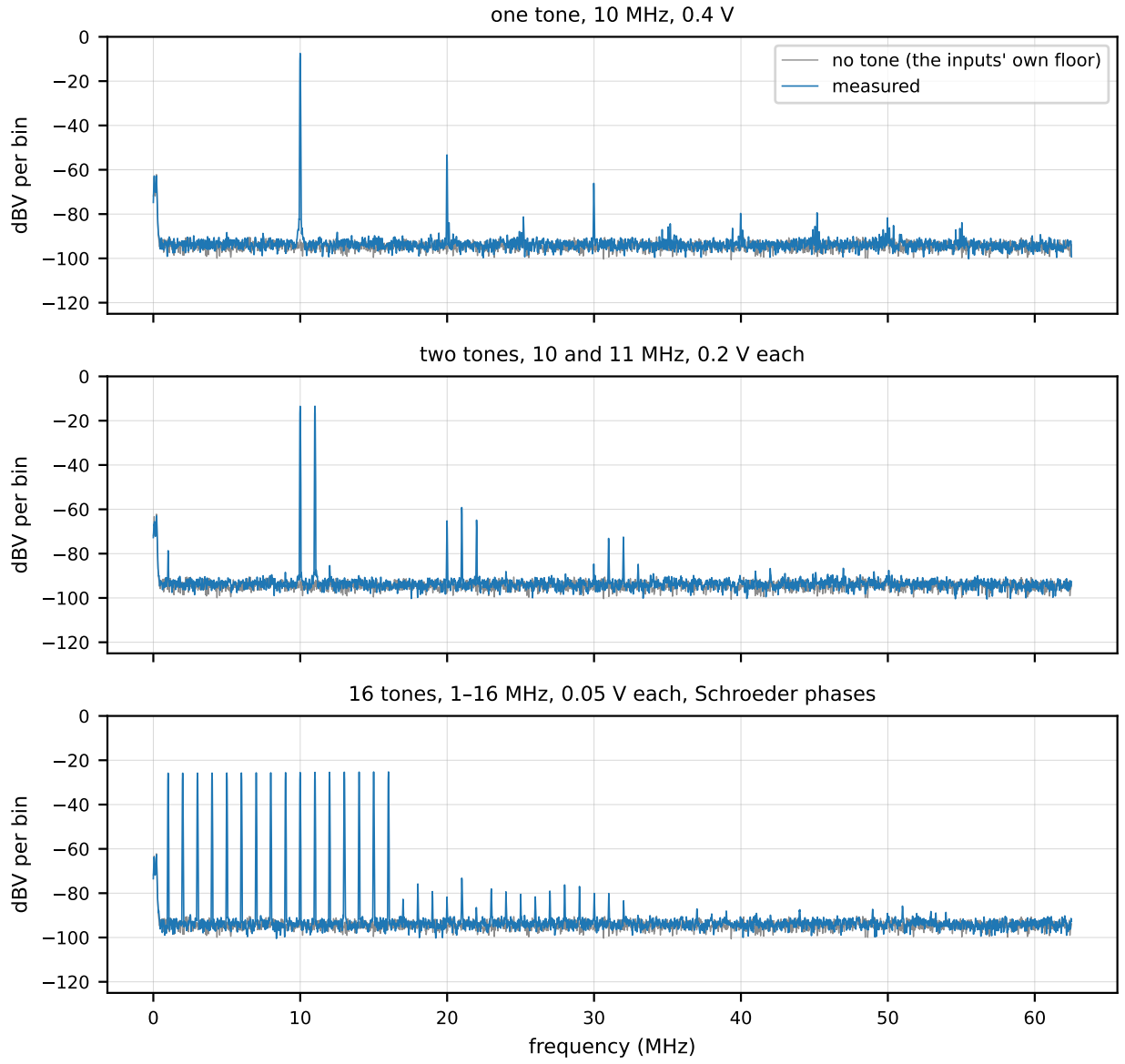


Figure 15: Measured through the loop OUT1 → IN1 (8 averages). Top: 10 MHz at 0.4 V, second harmonic −46 dBc, third −59 dBc. Middle: 10 and 11 MHz at 0.2 V each; third-order intermodulation −72 dBc, the sum frequency 21 MHz −46 dBc. Bottom: 16 tones of 0.05 V. Grey: no tone, the inputs' own floor (about −94 dBV per bin), with a line at 214 kHz present even without any signal.

What the inputs can show

The ADC's own distortion and noise are part of every spectrum, so these figures are upper bounds for the outputs: spurs below about −75 dBc, the DAC images near 125 MHz − f and phase noise need a spectrum analyzer. The digital synthesis itself is much cleaner: spurs below −95 dBc after the 14-bit rounding (computed).

11 Recipes

Short, complete examples for common tasks. Each runs as it is, on the board or through the client.

11.1 Sweeping the phase between the outputs

To calibrate a phase shifter or a phase detector, step OUT2's phase relative to OUT1:

```
f = 10e6
synth.set_tone(1, 1, f, 0.3)
for phase in range(0, 360, 30):
    synth.set_tone(2, 1, f, 0.3, phase)
    # ... read your detector here ...
```

Each step changes only OUT2's phase; OUT1 runs on undisturbed, and the step is exact to 0.0055°.

11.2 Turning an RF amplifier on and off gently

```
synth.play({"steps": [
    {"at": 0.0, "tones": {"1.1": {"frequency": 25e6, "amplitude": 0.0}}},
    {"at": 1e-3, "ramp": 1e-3, "shape": "cos2", "tones": {"1.1": {"amplitude": 0.5}}}]})
# later:
synth.play({"steps": [{"at": 0.0, "ramp": 1e-3, "shape": "cos2", "tones": {"1.1": {"amplitude":
    ↪ 0.0}}}]})
```

A 1 ms $\cos^2$ edge avoids the broadband splash of a hard switch.

11.3 A stepped frequency sweep with a marker per step

For a scope or a DAQ that records one point per frequency:

```
freqs = [1e6 + 0.2e6 * i for i in range(96)]          # 1 to 20 MHz
synth.play({"steps": [{"at": i * 50e-6, "marker": True,
                       "tones": {"1.1": {"frequency": f, "amplitude": 0.2}}}
                      for i, f in enumerate(freqs)]})
```

Each marker rises on the sample where the new frequency starts; trigger your acquisition on DIO1_P and record one point per pulse.

11.4 A slow linear sweep for spectroscopy, repeated

```
synth.play({"repeat": 0, "period": 2.0, "steps": [
    {"at": 0.0, "marker": True, "tones": {"1.1": {"frequency": 10e6, "amplitude": 0.2}}},
    {"at": 0.01, "ramp": 1.0, "tones": {"1.1": {"frequency": 20e6}}}]})
```

A 1 s sweep from 10 to 20 MHz every 2 s, until `synth.stop()`, with a marker at each start. The ramp is split into eight 125 ms pieces automatically and is still exactly linear in the frequency word.

11.5 An amplitude staircase for a linearity check

```
levels = [0.01, 0.02, 0.05, 0.1, 0.2, 0.5]
synth.play({"steps": [{"at": i * 10e-3, "marker": True,
                       "tones": {"1.1": {"frequency": 5e6, "amplitude": a}}}
                      for i, a in enumerate(levels)]})
```

The amplitudes are set in calibrated volts with 2^{-17} resolution; at 5 MHz the loop response is flat within 3 % (figure 5).

11.6 Multitone noise

Sixteen equal tones with random phases make a band-limited noise-like signal with a Gaussian-looking amplitude distribution; drawing new phases gives a new realization:

```
import random
tones = {str(k + 1): {"frequency": 2e6 + k * 0.25e6, "amplitude": 0.05,
                      "phase": random.uniform(-180, 180)} for k in range(16)}
synth.apply({"outputs": {"1": {"tones": tones}}})
```

Its spectrum is 16 exact lines, so it is a deterministic noise: ideal for repeatable tests of a servo or a filter.

11.7 A two-tone test of an amplifier

Two equal tones close together show an amplifier's third-order intermodulation at $2f_1 - f_2$ and $2f_2 - f_1$:

```
from synth import measure                    # on the board: from pynq.overlays.synth import
    ↪ measure
f1, f2 = measure.bin_frequency(10e6), measure.bin_frequency(10.5e6)
synth.apply({"outputs": {"1": {"tones": {"1": {"frequency": f1, "amplitude": 0.1},
                                         "2": {"frequency": f2, "amplitude": 0.1}}}}})
```

Measure the amplifier's output with a spectrum analyzer (or, above about −70 dBc, with `synth.spectrum("in1")` through an attenuator). The board's own IMD3 is −72 dBc at 0.2 V per tone (figure 15).

11.8 Frequency-shift keying

A bit pattern as two frequencies, 1 μs per bit, phase-continuous:

```
bits = [1, 0, 1, 1, 0, 0, 1, 0]
synth.play({"phase_mode": "continuous", "steps": [
    {"at": i * 1e-6, "tones": {"1.1": {"frequency": 11e6 if b else 10e6, "amplitude": 0.2}}}
    for i, b in enumerate(bits)]})
```

In continuous mode the frequency changes without a phase jump; in coherent mode each frequency would instead resume its own phase from t_0.

11.9 What synth cannot do

- **Frequencies above 62.5 MHz** (the Nyquist limit), and more than 16 tones per output.
- **An external clock.** The board runs on its own crystal; two boards are not phase-locked to each other, and synth's phases are exact only relative to its own clock.
- **Real-time control from Python.** Calls take about 0.1 s; anything timed must be a sequence.
- **Frequency or phase modulation from an input.** Only the amplitude can follow an input.
- **Long captures.** The inputs record 65.5 μs at a time.

12 Using synth from experiment software

12.1 From Python

The client (`SynthClient`) is the simplest way to drive synth from an experiment's control program: it needs only the Python standard library and has the same methods as the driver. Open one connection per program and keep it:

```
synth.load_preset("off")
synth.apply({"outputs": {"1": {"tones": {"1": {"frequency": 20e6, "amplitude": 0.2}}}}})
state = synth.hardware()
if state["counts"]["late"]:
    print("a change came late:", state["counts"])
```

Errors come back as exceptions with the board's message (`RemoteError` on a PC, `ValueError` or `RuntimeError` on the board): a refused setting changes nothing.

What is real-time and what is not

Every call travels over the network and takes of the order of 0.1 s through the service; an `apply()` then takes effect `apply_delay` (10 ms) after it reaches the board. Anything that must happen at a defined time relative to something else belongs in a **sequence**: its steps are timed by the FPGA to 8 ns, and a trigger from your experiment controller starts it. Use calls for settings between runs.

12.2 From other languages

The TCP protocol is plain text, one command per line. Any language with sockets works; for example in Python without the client:

```
import socket
with socket.create_connection(("192.168.0.149", 5025)) as s:
    f = s.makefile("rw")
    f.write("TONE 1.1 10e6 0.25\n"); f.flush()
    print(f.readline())            # OK {"frequency": 10000000.0, "amplitude": 0.25, ...}
```

and in MATLAB (not tested here):

```
t = tcpclient("192.168.0.149", 5025);
writeline(t, "TONE 1.1 10e6 0.25"); disp(readline(t));
writeline(t, "PLAY " + jsonencode(seq)); disp(readline(t));
```

A sequence is the same dictionary as in Python, sent as JSON after `PLAY`. Lines are limited to 64 kB; the Python client compresses longer sequences (`PLAYZ`) by itself.

12.3 Building sequences in code

Sequences are ordinary lists, so loops and numpy build them easily. A frequency comb that steps through 50 settings, 200 μs each, with a marker at every step:

```
steps = []
for i in range(50):
    spacing = 0.5e6 + 0.02e6 * i
    steps.append({"at": i * 200e-6, "marker": True, "tones": {
        "1.%d" % (k + 1): {"frequency": 10e6 + k * spacing, "amplitude": 0.05} for k in range(8)}})
info = synth.play({"steps": steps})
print(info["entries"], "queue entries; starts at sample", info["start"])
```

`play()` returns the number of queue entries and the start sample: compare the entries with the queue's 4096 to know whether the sequence is streamed. `sequence_status()` says when it is done.

12.4 Several programs at once

A notebook, your experiment script and the command line may all be connected. Every change is one atomic transaction, and every reader sees the settings actually applied, also during sequences. To make sure that nobody changed the settings between your reading and your writing, pass the revision you read: `synth.apply(patch, expected_revision=revision)` is refused if the configuration changed in between (`common/README.md`, "Sessions and concurrent control"). Save the configurations your experiment needs as presets (`synth.save_preset("mot")`) and load them by name.

13 Application notes

13.1 Acousto-optic deflectors and optical tweezer arrays

Goal. An AOD makes one deflected beam per RF tone; the beam angle follows the frequency, its power the tone's power. Several tones make an array of beams (tweezers), which can be moved smoothly by ramping the frequencies, for example to rearrange trapped atoms.

Rearrangement moves. Eight tones at 30–37 MHz, moved by 2 MHz in 300 μs along a minimum-jerk profile and back, with a marker at the start of the move:

```
base = {"1.%d" % (k + 1): {"frequency": 30e6 + k * 1e6, "amplitude": 0.08} for k in range(8)}
moved = {"1.%d" % (k + 1): {"frequency": 32e6 + k * 1e6} for k in range(8)}
back = {"1.%d" % (k + 1): {"frequency": 30e6 + k * 1e6} for k in range(8)}
synth.play({"steps": [
    {"at": 0.0, "tones": base},
    {"at": 1e-3, "ramp": 300e-6, "shape": "min-jerk", "tones": moved, "marker": True},
    {"at": 2e-3, "ramp": 300e-6, "shape": "min-jerk", "tones": back}]})
```

All eight tones start each of the 16 pieces on the same sample (figure 16). The peak chirp rate of this move is 12.5 kHz/μs, well below the limit of 119 kHz/μs. On the board used here (2026-10-22) the sequence was 1705 queue entries, `play()` took 208 ms, and all 33 APPLYs ran on time. After each move the tones read 0.081 to 0.083 V in loopback, and their frequencies sat on the target words, within 0.2 μHz of the nominal values.

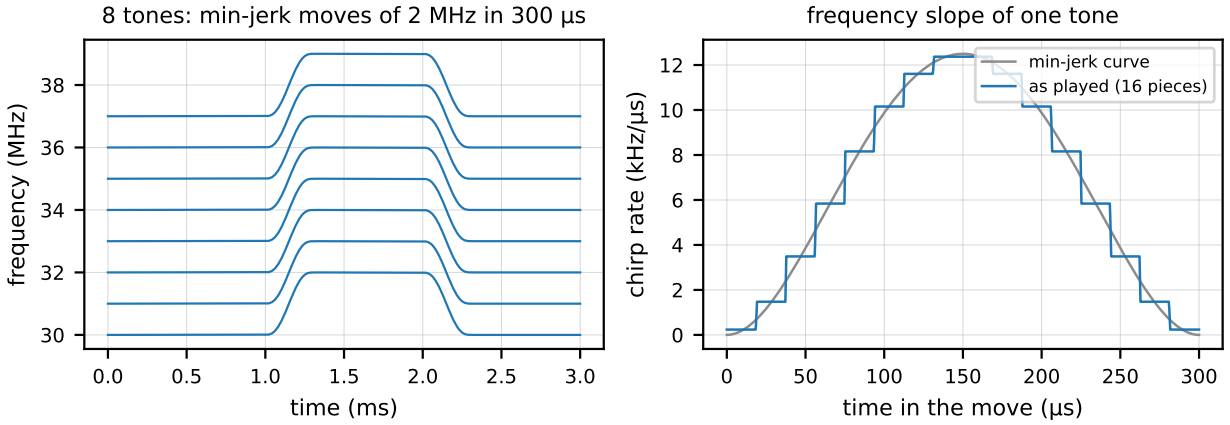


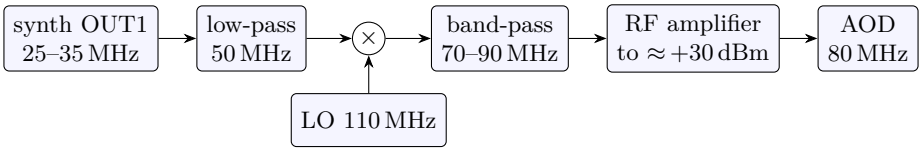
Figure 16: Computed: the eight tones' frequencies during the moves (left), and the chirp rate of one tone during a move: the minimum-jerk curve and the 16 linear pieces the FPGA plays (right).

Equal beams. The diffraction efficiency of an AOD and the RF chain's response vary with frequency, so equal RF amplitudes rarely give equal beams. Measure each beam (a camera, or a photodiode while the tones are switched on one at a time) and scale the amplitudes; iterate two or three times. The same loop on the board's own inputs equalizes the electrical amplitudes:

```
target = 0.08
tones = {str(k + 1): {"frequency": 30e6 + k * 1e6, "amplitude": target} for k in range(8)}
synth.apply({"outputs": {"1": {"tones": tones}}})
for _ in range(3):
    found = synth.measure_tones()["inputs"]["in1"]["tones"]
    for k, t in found.items():
        tones[k]["amplitude"] *= target / t["amplitude"]
    synth.apply({"outputs": {"1": {"tones": tones}}})
```

Headroom with many tones. N equal tones need NA of headroom in the worst case. Spreading the phases (Schroeder, section 6) keeps the real peak near $2\sqrt{N/2}\,A$; with `allow_clip` the driver then accepts larger amplitudes. Keep the amplifier in its linear range: intermodulation between the tones makes extra beams at $2f_i - f_j$.

Upconversion to 80 MHz AODs. The board's band is DC to 62.5 MHz. For an AOD centred at 80 MHz, mix up with a local oscillator, filter and amplify:



With the LO above the band ($f_{\mathrm{AOD}} = f_{\mathrm{LO}} - f$) the frequencies are mirrored: a move up in synth is a move down at the AOD. The low-pass removes the DAC images ($125\,\mathrm{MHz} - f$); the band-pass removes the sum products and the LO leakage. Relative phases between the tones survive the mixing. A tone of 0.08 V is −12 dBm; allow 6–7 dB conversion loss in the mixer, and set the amplifier's gain for the AOD's rated RF power.

13.2 Pulse sequences and triggers

Goal. Shaped RF pulses at exact times relative to an experiment, for example Raman or microwave-style pulses, with phases you control and a marker for the data acquisition.

Pulse shape and spectrum. A rectangular pulse has a sinc-shaped spectrum whose sidelobes fall slowly: it excites transitions far from resonance (here −33 dB between 1 and 2 MHz from the carrier). Smooth edges lower the sidelobes by tens of dB (figure 17). The FPGA plays a shaped edge as straight pieces (13 for an 8 µs edge of one tone), and their corners set the far sidelobes: −66 dB for $\cos^2$ and −63 dB for Blackman edges as played, against −84 and −108 dB for the ideal curves. Longer edges get more pieces (up to `segments`) and come closer to the ideal. The price of smooth edges is a longer pulse for the same area.

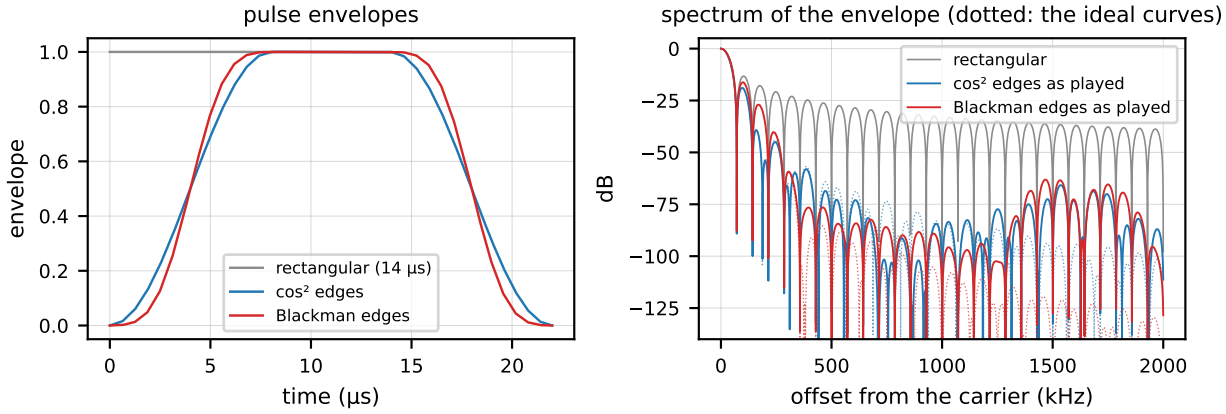


Figure 17: Computed: pulses with 8 µs edges and a 6 µs flat top as the FPGA plays them, and a rectangular pulse (left); the spectra of their envelopes around the carrier (right), with the ideal curves' spectra dotted.

A Ramsey sequence started by the experiment. Two $\pi/2$ pulses with 4 µs $\cos^2$ edges and a 4 µs flat top, 200 µs apart, the second with a phase of 90°; every trigger edge on DIO0_P starts it again, with a marker at the first pulse:

```
f = 20e6
pulse = lambda at, phase: [
    {"at": at, "tones": {"1.1": {"frequency": f, "amplitude": 0.0, "phase": phase}}},
    {"at": at + 1e-6, "ramp": 4e-6, "shape": "cos2", "tones": {"1.1": {"amplitude": 0.3}}},
    {"at": at + 9e-6, "ramp": 4e-6, "shape": "cos2", "tones": {"1.1": {"amplitude": 0.0}}}]
steps = pulse(10e-6, 0.0) + pulse(210e-6, 90.0)
steps[0]["marker"] = True
synth.play({"trigger": True, "repeat": 0, "steps": steps})
```

Because the trigger edge is the sequence's t_0, the carrier's phase in both pulses is the same in every run, to within the 8 ns sampling of the trigger (7.2° per ns at 20 MHz: up to 58° for 8 ns). If your experiment needs better, derive the trigger from the board's clock domain, or use a free-running coherent tone instead of a triggered phase. The 90° of the second pulse is relative to the first, exactly.

Markers and timing. The marker rises on the DAC clock edge of the marked step's first sample (figure 11): use it to trigger a scope or a DAQ, or to time other equipment from synth instead of the other way round. To keep a train running without a controller, use `"repeat": 0` without a trigger and a `period` (figure 12).

13.3 Phase-coherent RF

IQ and quadrature. Equal frequencies on both outputs with phases 0° and −90° give an I/Q pair (figure 3); the preset `quadrature` does it at 1 MHz. With the calibrated output delay, the phase difference at the connectors is the set one within about 0.1° at 10 MHz.

Beat notes and offsets. Two tones at f and $f + \delta f$ beat at exactly δf, with δf resolved to 0.44 μHz: a 1 Hz offset between the outputs is exact relative to the board clock. Both outputs share the clock, so their beat has no noise of its own.

Coherent frequency hopping. A sequence of steps hops between frequencies, and in coherent mode each frequency returns with its own uninterrupted phase:

```
hops = [10e6, 12e6, 15e6, 10e6, 12e6, 15e6]
synth.play({"steps": [{"at": i * 20e-6, "tones": {"1.1": {"frequency": f, "amplitude": 0.2}}}
                      for i, f in enumerate(hops)]})
```

A receiver that tracks each frequency separately sees continuous phases; this is the basis of stepped-frequency measurements of delays and distances.

Test signals with exact modulation. A carrier with two sidebands is an amplitude- or a phase-modulated signal, depending only on the sidebands' phases, and synth sets those exactly. With a carrier A_c at f_c and sidebands A_s at $f_c \pm f_m$: sidebands in phase with the carrier give AM with index $m = 2A_s/A_c$, $(1 + m\cos 2\pi f_m t)\, A_c \sin 2\pi f_c t$; both sidebands 90° ahead of it give narrow-band PM with index $\beta = 2A_s/A_c$, $A_c \sin(2\pi f_c t + \beta \cos 2\pi f_m t)$ to first order in β (all phases relative to t_0):

```
fc, fm, ac, a_s = 20e6, 10e3, 0.3, 0.015          # m = beta = 0.1
am = {"1": {"frequency": fc, "amplitude": ac},
      "2": {"frequency": fc + fm, "amplitude": a_s, "phase": 0.0},
      "3": {"frequency": fc - fm, "amplitude": a_s, "phase": 0.0}}
pm = {"1": {"frequency": fc, "amplitude": ac},
      "2": {"frequency": fc + fm, "amplitude": a_s, "phase": 90.0},
      "3": {"frequency": fc - fm, "amplitude": a_s, "phase": 90.0}}
synth.apply({"outputs": {"1": {"tones": am}, "2": {"tones": pm}}})
```

OUT1 then carries 10 % AM and OUT2 0.1 rad PM of the same carrier, both exact to the amplitude and phase steps: a calibration source for demodulators, phase detectors and lock-ins.

A reference for a lock-in. Drive the experiment from OUT1 and the lock-in's reference input from OUT2 at the same frequency. Any change on OUT1 (frequency hops, amplitude steps) keeps the two coherent, and the reference's phase can be set to compensate the experiment's delay: `synth.set_tone(2, 1, f, 0.5, phase)`. Over many re-applies the phase difference repeats within a few millidegrees (figure 8).

13.4 Measuring a filter or an amplifier

Goal. The board can act as a simple network analyzer: OUT1 drives the device, IN1 measures it, and `measure_tones()` gives amplitude and phase at each frequency. Measure a "through" first (the device replaced by a cable) and divide:

```
import math
freqs = [k * 2e6 for k in range(1, 21)]
def sweep():
    result = []
    for f in freqs:
        synth.apply({"outputs": {"1": {"tones": {"1": {"frequency": f, "amplitude": 0.2}}}}},
                    replace=True)
        t = synth.measure_tones()["inputs"]["in1"]["tones"]["1"]
        result.append((t["amplitude"] / t["set_amplitude"], t["phase"]))
    return result
through = sweep()                       # OUT1 -> IN1 through a cable
dut = sweep()                           # ... and again with the device in the path
for f, (a0, p0), (a1, p1) in zip(freqs, through, dut):
    print("%5.1f MHz  %+6.2f dB  %+7.2f deg" % (f / 1e6, 20 * math.log10(a1 / a0),
                                               (p1 - p0 + 180) % 360 - 180))
```

Because every tone's phase is referred to t_0, the phases of separate measurements are directly comparable: no reference channel is needed. Figure 18 shows the through itself on the board used here: a loop delay of 349.2 ns (30 samples, 240 ns, are the FPGA's latency and the DAC interface; the rest the converters, the analog front ends and the cables) and the response of the outputs and inputs together.

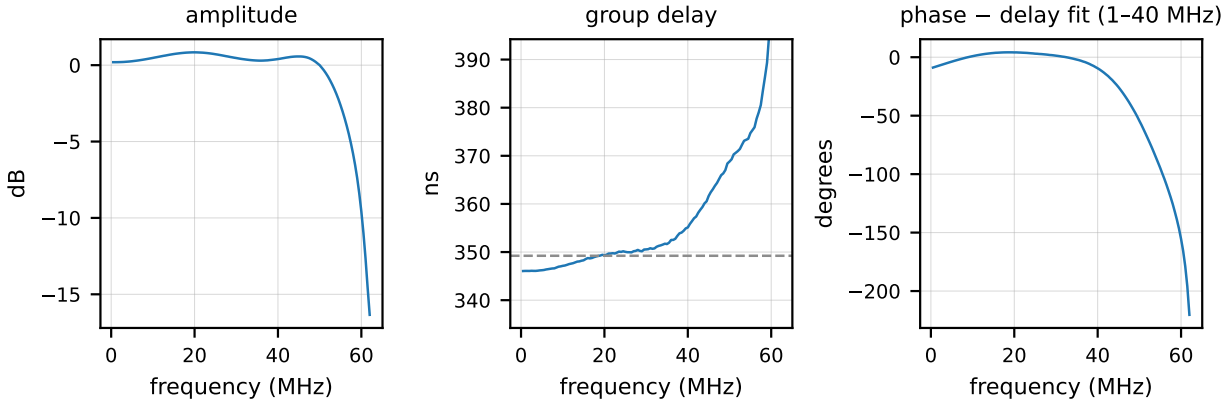


Figure 18: Measured: the loop OUT1 → IN1 through a 50 Ω terminator, from 0.5 to 62 MHz in 124 steps. Left: amplitude. Middle: group delay; dashed, the delay fitted between 1 and 40 MHz. Right: the phase that remains after removing that delay.

Keep the device's level within the inputs' range (±1 V on LV), and remember that the inputs are high-impedance: terminate a 50 Ω device's output at IN1.

14 Specifications

Figures for the board used here (calibrated 2026-10-04; recordings of 2026-10-22). "Measured" values are records in `docs/VERIFICATION.md` and this manual's data (`docs/manual/data/`); "computed"

ones come from the bit-exact model; the rest are design values.

quantity	value
tones	16 per output, both outputs on one 125 MSa/s time base
frequency	0 to 62.5 MHz, steps of 0.44 μHz, exact relative to the board clock
amplitude	0 to about 1.13 V (+11 dBm) into 50 Ω, steps of 8.6 μV; droop corrected
phase	steps of 0.0055°; measured: steps within 0.003°, repeatability 4.3 m° RMS after excursions
OUT1–OUT2 skew	measured: OUT2 0.14–0.20 ns earlier (1–40 MHz); corrected by the calibration to about $\pm$0.03 ns
loop response	measured, OUT→IN through 50 Ω: 1.02 at 1 MHz, 1.10 at 20 MHz, 1.00 at 50 MHz, 0.74 at 55 MHz, 0.33 at 60 MHz (inputs included)
digital spurs	sine core below −120 dBc; after the 14-bit rounding below −95 dBc; with dither a flat floor (computed)
analog spectrum	measured through the ADC (upper bounds): 0.4 V at 10 MHz: H2 −46 dBc, H3 −59 dBc; two tones of 0.2 V: IMD3 −72 dBc
updates	any set of tones on one sample; `apply()` takes effect 10 ms after the call
ramps	linear or shaped, exact endpoints, up to 3600 s; chirps up to 119 kHz/μs
sequences	4096-entry queue; repeated by the FPGA; streamed at about 60 000 entries/s (measured)
trigger	DIO0_P, 3.3 V, pull-down; edge to output $(T + 30) \times 8$ ns + 0–8 ns
marker	DIO1_P, 3.3 V, on the DAC edge of the step's first sample; back on DIO0_P 31 samples (248 ns) after the step (measured)
amplitude modulation	by IN1 or IN2; depth up to about $\pm$1.8/V (LV); 64 ns from input to output in the FPGA; measured slope 1.000 for 1/V
input capture	8192 samples of IN1 and IN2 on the same time base
not measured	phase noise and absolute frequency (the board's crystal), DAC images, output spurs below the ADC's, the analog delay of the AM path

15 Troubleshooting

"can leave the DAC range" on apply()
: The worst-case peak exceeds the range: lower the amplitudes or the offset, or set `allow_clip` (and watch the clip counts).

A sequence is `halted`
: A step came late: the host or the worker could not keep up, or a step was too early after the previous one. The outputs keep the last applied settings. Make the dense part shorter than the queue, or slower; avoid polling `STATUS?` without pause from another program while streaming.

play() refuses a long sequence
: Sequences longer than the queue need the control service (`make install-service`); its worker streams them. Over TCP, very long sequences exceed one command: play them on the board.

A repeating sequence is refused
: It must fit in the queue, every tone's first change must be a step, and the period must exceed the last step.

A chirp is refused
: It is faster than 119 kHz/μs: make the ramp longer, or use a step.

A trigger sequence does not start
: Check `hardware()["trigger_input_high"]` and the trigger count; the edge must rise after the sequence started waiting (earlier edges are counted as missed). 3.3 V logic only.

Phases differ from the settings in loopback
: That is the loopback delay, $-360° f\tau$: constant for a given frequency.

Amplitudes twice the setting
The load is high-impedance: the setting is the level into 50 Ω.

`hardware()` shows late APPLYs after apply()
The host was slower than `apply_delay`; phases are then off by f times the lateness. Apply again, or raise `apply_delay`.

Other clients wait for seconds
A long `play()` holds the board while it compiles (about 0.85 ms per step).

16 Quick reference

task	Python (TCP)
one tone	`synth.set_tone(1, 1, 10e6, 0.25, 30)` (`TONE 1.1 10e6 0.25 30`)
many settings at once	`synth.apply({"outputs": {...}})` (`CONFIG <json>`)
what is on	`synth.describe()`, `synth.tones()` (`DESCRIBE?`, `TONES?`)
everything off	`synth.safe()` (`SAFE`)
phases from now	`synth.zero_time()` (`ZERO`)
a timed sequence	`synth.play({"steps": [...]})` (`PLAY <json>`)
start it by a trigger	`"trigger": True`; edge on DIO0_P or `synth.trigger()` (`TRIG`)
repeat it in the FPGA	`"repeat": K, "period": seconds` (0: until stopped)
stop it, its state	`synth.stop()`, `synth.sequence_status()` (`STOP`, `SEQ?`)
amplitude modulation	`synth.set_am(1, "in1", 0.5)` (`AM 1 IN1 0.5`)
counters, late steps	`synth.hardware()` (`HW?`)
measure in loopback	`synth.measure_tones()` (`MEASURE?`)
spectrum of an input	`measure.analyze(synth.spectrum("in1"), [f])`
presets	`synth.load_preset("comb5")`, `save_preset("name")` (`PRESET ...`)

A step dictionary:

```
{"at": 1e-3, "ramp": 2e-4, "shape": "cos2", "segments": 16, "marker": True,
 "tones": {"1.3": {"frequency": 10e6, "amplitude": 0.1, "phase": 90.0}},
 "outputs": {"2": {"offset": 0.0, "dither": False, "am": {"source": "off", "depth": 0.0}}}}
```

number to remember	value
sample	8 ns (125 MSa/s)
frequency step	0.44 μHz
largest tone	1.13 V into 50 Ω (+11 dBm)
step spacing	0.5 μs + 8 ns per changed word
fastest chirp	119 kHz/μs
queue	4096 entries (about 680 one-tone steps)
trigger to output	$(T + 30) \times 8$ ns + 0–8 ns
capture	8192 samples (65.5 μs)

A Command reference

A.1 Python methods

The driver (`Synth`, on the board) and the client (`SynthClient`, on a PC) have the same methods. The common ones (`config`, `apply`, `describe`, `status`, `safe`, `presets`, `load_preset`, `save_preset`, `calibration`, `calibrate`, `ranges`, `set_input_range`) are those of every design in the repository (`common/README.md`).

method	action
`set_tone(o, k, f, A, phase=None)`	one tone; returns its applied settings
`tones()`	the tones that are on
`zero_time(delay=None)`	t_0 := the next change's sample
`set_am(o, source, depth)`	amplitude modulation (`"off"`, `"in1"`, `"in2"`; depth in 1/V)
`play(sequence)`	play a sequence; returns start, entries, trigger, ...
`stop()`, `sequence_status()`	stop a sequence; its state
`trigger()`	software trigger
`hardware()`, `clear_flags()`	counters, queue and flags; clear them
`capture(delay)`	8192 samples of IN1 and IN2
`measure_tones(t0=None)`	the configured tones measured on the inputs
`spectrum(channel, averages)`	an input's amplitude spectrum (numpy)
`measure_response(o, frequencies)`	the loop response table
`phase_difference(f)`	IN1 − IN2 with one tone on both outputs
`set_output_response(o, points)`	store an output's response in the calibration
`set_output_delay(o, seconds)`	store an output's delay in the calibration

`measure.skew(a, b)`, `measure.analyze(spectrum, f)` and `measure.bin_frequency(f)` are functions of the module `synth.measure`.

A.2 TCP commands

Port 5025, one command per line, keywords case-insensitive; replies `OK <value>` or `ERR <message>`.

command	reply / action
`TONE <o>.<k> <Hz> <V> [<deg>]`	set a tone; JSON as applied
`TONE <o>.<k>?`, `TONES?`	one tone; the tones that are on
`AM <o> OFF|IN1|IN2 [<1/V>]`	amplitude modulation
`ZERO [<s>]`	t_0 := the next change's sample
`TRIG`	software trigger
`PLAY <json>`, `PLAYZ <base64>`	play a sequence (PLAYZ: zlib-compressed)
`STOP`, `SEQ?`	stop; sequence state
`HW?`, `CLEAR`	counters and flags; clear them
`CAPTURE? [<s>]`, `MEASURE? [<t0>]`	capture; tone measurement
`CONFIG <json>`, `CONFIG?`, `DESCRIBE?`	apply a (partial) configuration; read it
`PRESET <name>`, `PRESET SAVE <name>`	presets
`SAFE`	the safe state
`CAL?`, `CAL LOOPBACK`, `CAL RESET`	board calibration
`CAL RESPONSE OUT<n> <json>`	store an output's response table
`CAL DELAY OUT<n> <s>`	store an output's delay
`STATUS?`, `HELP`, `*IDN?`	status, help, identity

A.3 Command-line options

`python3 -m rpfpga.cli -host 192.168.0.149 [options]` (on a PC), or `python -m rpfpga.cli synth [options]` on the board:

option	action
`-tone O.K=HZ,V[,DEG]`	set a tone (repeatable)
`-offset1 V`, `-offset2 V`	output offsets
`-am1 SRC[,DEPTH]`, `-am2 SRC[,DEPTH]`	amplitude modulation
`-phase-mode coherent|continuous`	the phase mode
`-zero`	t_0 := the next change's sample
`-preset NAME`, `-safe`, `-load`, `-reset`	common options
`-calibration`, `-json`	print the calibration; JSON output