awg: user's manual

A two-channel function and arbitrary waveform generator with eight digital outputs, on the Red Pitaya STEMlab 125-14

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

Red Pitaya `awg` design, version 3.0 (build 0x97cda2f3)
October 25, 2026

Abstract

`awg` turns the Red Pitaya into a bench signal generator in three parts. A **function generator** on each output: sine, square, pulse, ramp, PRBS and table shapes with sub-sample edges, a second generator, Gaussian noise of adjustable bandwidth, AM with FM, PM, PWM, FSK or BPSK, sweeps, and bursts and gates started by triggers with nanosecond timestamps. A **true arbitrary waveform generator**: point pairs for both outputs from the board's memory at any rate up to 125 MSa/s, in sequences with repeats, waits and gapless jumps on events, at baseband or upconverted as I/Q. And **eight digital outputs** that play patterns, counts or a PRBS aligned with the analog outputs to the sample. Everything refers to one time base: phases are relative to one zero time, and a change of both outputs lands on one sample. This manual is for the lab: the first signals, what every setting means and how exactly it behaves, the timing of triggers and markers, sequences, the digital outputs, calibration, the board's self-checks, worked applications, specifications and references. Figures marked "measured" were recorded on the board used here, through its own inputs in loopback; the others are computed with the design's bit-exact model. 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 awg is **4**
1.1 What makes it different . . . 4
1.2 How the output is made, in one page . . . 5

2 Before you connect equipment **5**

3 Setting up **6**

4 Getting started **6**
4.1 A first signal . . . 6
4.2 The ways to control awg . . . 7

5 Time, phase and how changes happen **8**
5.1 The sample counter and the zero time . . . 8
5.2 Applies . . . 8
5.3 What the outputs show while a change is applied . . . 8
5.4 Refused settings . . . 8

6 Waveforms **9**
6.1 Example: the shapes . . . 9
6.2 Sine . . . 10
6.3 Square, pulse, ramp and triangle: edges . . . 10
6.4 PRBS . . . 10
6.5 Tables . . . 11
6.6 Generator 2 . . . 11

7 Noise **11**
7.1 Example . . . 11

8 Modulation **12**
8.1 Example: AM and FM by generator 2 . . . 12
8.2 Sources . . . 13
8.3 AM and DSB . . . 13
8.4 FM, PM and PWM . . . 14
8.5 FSK and BPSK . . . 14

9 Sweeps **14**
9.1 Example . . . 14

10 Bursts and gates **15**
10.1 Example . . . 15

11 Triggers **16**
11.1 Sources . . . 16
11.2 Timing . . . 17
11.3 Thresholds on the inputs . . . 17
11.4 Timers . . . 17
11.5 The other output . . . 17

12 Markers **18**

13 Limits, protection and dither **18**
13.1 Voltage limits . . . 18
13.2 Dither . . . 19

14 Arbitrary waveforms **19**
14.1 Example: a chirp . . . 19
14.2 Segments . . . 20
14.3 The sequencer . . . 21
14.4 Filters . . . 21
14.5 Sequences . . . 22
14.6 Memory and streaming . . . 23
14.7 Applies, restarts and status . . . 23

15 I/Q **23**
15.1 Baseband I/Q . . . 23
15.2 Upconversion in the FPGA . . . 23

16 Digital outputs **24**
16.1 Example . . . 24
16.2 The settings . . . 24
16.3 Programs . . . 25
16.4 Starting and stopping . . . 25

16.5 Patterns that follow a sequence . . . 25
16.6 Timing and alignment . . . 27

17 Calibration **27**
17.1 Input ranges . . . 27
17.2 DC calibration . . . 27
17.3 Output response and droop . . . 28
17.4 Skew between the outputs . . . 28
17.5 Delay of the digital pins . . . 28

18 Measuring with the inputs **28**

19 Presets **29**

20 The notebook panel **29**

21 Using awg from experiment software **30**
21.1 From Python . . . 30
21.2 Status and counters . . . 30
21.3 From other languages . . . 31
21.4 The command line . . . 31
21.5 After a reboot, and the service . . . 31

22 Application notes **31**
22.1 A pulse experiment with gates for other instruments . . . 31
22.2 Adiabatic passage: a shaped chirp . . . 32
22.3 I/Q pulses for a modulator or a qubit drive . . . 32
22.4 Testing a receiver: data, noise and a PRBS reference . . . 33
22.5 Event-driven sequences . . . 33
22.6 Clocks and frames for other instruments . . . 34
22.7 What awg cannot do . . . 34

23 Specifications **34**

24 Troubleshooting **35**

25 Quick reference **36**

A Command reference **36**
A.1 The configuration . . . 36
A.2 Python methods . . . 37
A.3 TCP commands . . . 37
A.4 Command-line options . . . 38

1 What awg is

`awg` is an FPGA design for the Red Pitaya STEMlab 125-14 together with the software that controls it. With it the board becomes a two-channel signal generator with the features of a bench function and arbitrary waveform generator, plus eight logic outputs. Each analog output is the sum

$$\mathrm{OUT}n(t) = \mathrm{offset}_n + A_1\, g_1(t)\,(1 + d\, m(t)) + A_2\, g_2(t) + \mathrm{noise}_n(t) + A_{\mathrm{arb}}\, p_n(t), \qquad n = 1, 2, \quad (1)$$

computed at every sample (125 MSa/s, one sample every 8 ns), clamped to the output's limits and rounded once to the 14-bit DAC. The terms are:

g_1, generator 1
: the main waveform: sine, square, pulse, ramp or triangle (with adjustable rise and fall times), a pseudo-random bit sequence, a table shape, or off. Its frequency and phase may be modulated (FM, PM, PWM, FSK, BPSK) or swept, and it may run continuously or in bursts.

m, the modulation
: amplitude modulation (AM, or double-sideband with suppressed carrier) by generator 2, by an input (IN1 or IN2), or by the other output.

g_2, generator 2
: a second waveform of any shape, added to the output with its own amplitude, and the internal modulation source.

noise
: Gaussian noise with a flat spectrum up to an adjustable bandwidth.

p_n, arbitrary points
: the output's half of a stream of point pairs (OUT1, OUT2) played by the sequencer from the board's memory, reconstructed by a step, linear or band-limited filter (in place of the noise). With *upconversion* the pairs are I and Q, and the output is $I\cos\omega t - Q\sin\omega t$.

Eight **digital outputs** D0–D7 play 8-bit words, one per sample, from patterns, counters or a PRBS; two **marker outputs** show the state of an output (its sync, a burst, a sweep marker, a start pulse, or marker bits stored with the arbitrary points). Triggers come from two pins timestamped to 1 ns, four event pins, thresholds on the inputs, two timers, the software and the other output.

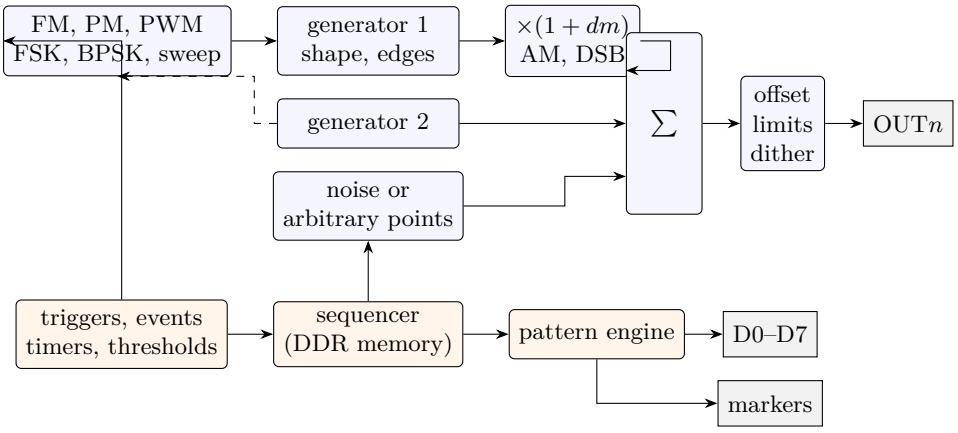


Figure 1: One output of awg (the other is the same) and the parts both share: the sequencer of the arbitrary points, the triggers, and the pattern engine of the digital outputs. Everything runs on one 125 MHz sample clock.

1.1 What makes it different

- **One time base, one zero time.** Both outputs share one sample counter. Phases are relative to a common zero time t_0, and a configuration change of both outputs happens on one sample

(section 5). Two outputs at the same frequency stay locked with the phase you set; going back to a frequency restores its phase.

- **Sub-sample edges.** Square, pulse, ramp and PRBS edges are computed from the exact phase, so their timing moves in steps of a fraction of a nanosecond: a period that is not a whole number of samples shows no 8 ns jitter (section 6.3).
- **Triggers timed to 1 ns.** Edges on TRIG A and TRIG B are timestamped to 1 ns, and a triggered burst or sequence starts at that time plus a fixed delay, to 1/256 of a sample (31 ps) in the computed phase (section 11).
- **True arbitrary waveforms.** Every point is played once, at the rate you set, from up to 16 million point pairs in the board's memory; sequences jump between segments without gaps (section 14).
- **Digital outputs aligned to the analog ones.** Patterns start with a burst, a sequence step or an apply, on the sample of the analog output they belong to (section 16).
- **Real volts.** All voltages are volts at the SMA connectors (into 50 Ω), through the board calibration shared by every design of the repository (section 17).

1.2 How the output is made, in one page

Each generator has a 48-bit **phase accumulator** that advances by the frequency word $F = \mathrm{round}(2^{48} f/f_\mathrm{s})$ every sample: the frequency resolution is $f_\mathrm{s}/2^{48} = 0.44\,\mu\mathrm{Hz}$. The phase selects the waveform's value: a sine from a table with interpolation, a trapezoid (square, pulse, ramp, triangle) whose edges are computed from the exact phase, a 4096-point table, or the next bit of a shift register (PRBS). The values of both generators, the noise or arbitrary points, the AM factor and the offset are summed with 27 fractional bits and rounded once to the 14-bit DAC code; nothing is rounded in between. Because the DAC holds every sample for 8 ns, the output's spectrum is that of the samples multiplied by $\mathrm{sinc}(f/f_\mathrm{s})$ ("droop": −0.37 dB at 20 MHz, −2.4 dB at 50 MHz), with images at $125\,\mathrm{MHz} - f$. awg corrects the droop of sine amplitudes (setting `droop_correction`); a low-pass filter after the output removes the images. Keep frequencies below about 50 MHz.

2 Before you connect equipment

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

Levels

Amplitudes are set **into 50 Ω**: the outputs behave like 50 Ω sources, so into a high-impedance load (a scope at 1 MΩ) you get about twice the set amplitude. The output range is about ±1.13 V into 50 Ω on the board used here. By default the driver refuses a setting whose worst case (|offset| plus all amplitudes, AM peaks and 4 σ of noise) leaves the DAC range; with `allow_clip` it accepts it and the output saturates on peaks. Set **voltage limits** for a sensitive load (section 13).

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

A sine of peak amplitude A carries $P\,[\mathrm{dBm}] = 20\log_{10}(A/\mathrm{V}) + 10$ into 50 Ω; noise of σ volts rms carries $20\log_{10}(\sigma/\mathrm{V}) + 13$.

- **Loading awg leaves the outputs quiet:** both outputs off at 0 in converter units (a few mV from 0 V before calibration), the digital outputs high impedance. After a reboot the control service restores the last configuration.
- **Steps are steps.** A change happens on one sample. Bursts start and stop where their phase says, which may be a step from the idle level.
- **The extension-connector pins are 3.3 V LVCMOS** and **not 5 V tolerant**. DIO0_P (TRIG A), DIO3_P (TRIG B) and DIO4_P–DIO7_P (EVENT 0–3) are inputs with pull-down resistors; DIO1_P and DIO2_P are the marker outputs, DIO0_N–DIO7_N the digital outputs D0–D7. Never drive an output pin from outside, and never connect a driving signal to a digital output you enable.
- **Images and harmonics.** A tone at f also appears at 125 MHz $- f$; sharp edges (squares, pulses, PRBS, arbitrary points with the step filter) contain harmonics up to the images. Filter the output for the band you use.

Safe state

`SAFE` (Python `awg.safe()`, TCP `SAFE`, the panel's **SAFE** button) mutes both outputs at once (to 0 V, or to the nearest protected limit), then switches everything off, stops any sequence and makes the digital outputs high impedance.

3 Setting up

The board is 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 `awg/`:

```
make upload            # the FPGA design, the Python package and the notebook
make install-service   # the control service (TCP port 5025), with awg active
make test-hw           # the loopback self-test (wiring below)
```

None of these needs Vivado: the FPGA design is in the repository. Rebuilding it is described in the design's `README.md`.

The loopback self-test uses the board's own inputs as the measuring instrument and prints one PASS or FAIL line per check (57 checks). Wire:

- OUT1 → IN1 and OUT2 → IN2, through equal cables with a 50 Ω terminator at each input; the input jumpers on LV;
- on the extension connector E1: DIO1_P (marker 1) to DIO0_P (TRIG A), and DIO0_N (D0) to DIO3_P (TRIG B). Without these two wires the trigger and digital checks report the missing wire.

It checks amplitudes and phases, shapes, edges and their sub-sample positions, noise (level, spectrum, statistics), bursts, sweeps (their law), every modulation, limits, triggers and thresholds, markers, arbitrary waveforms (playback, rates, waits, jumps, gates, I/Q, streaming at 125 MSa/s), the digital outputs and `SAFE`. It restores your configuration at the end. The results of the board used here are in `docs/VERIFICATION.md`.

4 Getting started

4.1 A first signal

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

```
import rpfpga                                       # must come first
from pynq.overlays.awg import DRIVER
awg = DRIVER()                                      # loads awg if the board runs another design
awg.set_output(1, function="square", frequency=1e6, amplitude=0.3, duty=30.0,
               rise=20e-9, fall=20e-9)
awg.set_output(2, function="sine", frequency=1e6, amplitude=0.3, phase=-90.0)
print(awg.describe())
```

TCP: `FUNC 1 square 1e6 0.3`
TCP: `OUT 1 {"duty":  30, "rise":  2e-8, "fall":  2e-8}`
TCP: `FUNC 2 sine 1e6 0.3 -90`

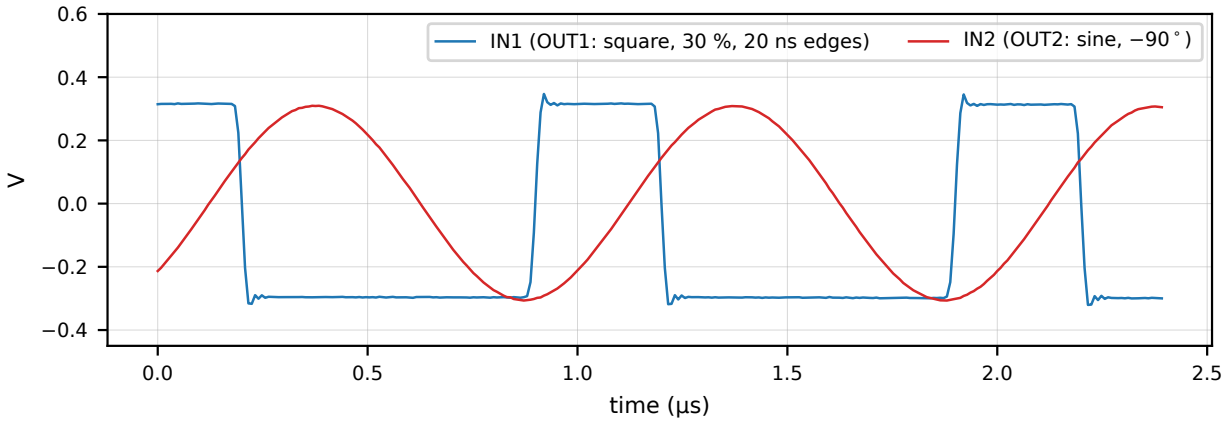


Figure 2: Measured: the example above, cabled OUT1 → IN1 and OUT2 → IN2 through 50 Ω terminators and captured by the board's inputs. The square's duty cycle at its 50 % points reads 30.05 %; the sine 0.3072 V. The overshoot of the square's edges is the analog output and input path (the DAC's reconstruction and the inputs' front end), not the computed waveform.

`describe()` prints what is actually applied, one line per output and one for the digital outputs:

```
{1: 'OUT1 = square 1000000 Hz 0.3 V 30 %', 2: 'OUT2 = sine 1000000 Hz 0.3 V -90.00 deg',
 'D': 'D0-D7 = off (high impedance)'}
```

Every setting method returns the applied settings: the frequency on its 0.44 µHz grid (here 999999.9999998899 Hz), the amplitude of the 24-bit word, an edge time rounded to what the hardware makes.

4.2 The ways to control awg

Jupyter on the board

Open `http://192.168.0.149:9090` (password `xilinx`), then `awg/awg.ipynb`: worked examples and the control panel (section 20).

Python on a PC

The client needs only the Python standard library and the repository's `common/python` and `awg/control`:

```
import sys; sys.path[:0] = ["common/python", "awg/control"]   # from the repository root
from awg import AwgClient
with AwgClient("192.168.0.149") as awg:          # the board must run awg
    awg.set_output(1, function="sine", frequency=1e6, amplitude=0.4)
    print(awg.output(1)["frequency"])
```

It has the same methods as the driver (and a few shortcuts such as `awg.function(1, "sine", 1e6, 0.4)`); 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 `FUNC 1 square 1e6 0.4`, `OUT 1 {"duty": 25}`, `HW?`, `SAFE`. Replies are `OK <value>` or `ERR <message>`; values are JSON. Appendix A.3 lists all commands.

Command line

The repository's common command line:

```
PYTHONPATH=common/python:awg/control python3 -m rpfpga.cli --host 192.168.0.149 \
    --func 1=sine,1e6,0.4 --offset1 0.05
```

Several programs may control the board at once (the service, a notebook, a script): each change is one atomic transaction, and every reader sees the settings actually applied. If the board runs another design, switch with `rpfpga.client.connect("192.168.0.149").load("awg")` (TCP `LOAD awg`).

5 Time, phase and how changes happen

5.1 The sample counter and the zero time

The FPGA counts samples from the moment it was configured: sample n is at time $n \cdot 8\,\text{ns}$. The configuration holds a **zero time** t_0, a sample index, and every phase is relative to it: a sine of frequency f and phase φ is $\sin(2\pi f\,(t - t_0) + \varphi)$ whatever happened before. Two outputs with the same frequency therefore keep the phase difference you set; a frequency changed and changed back returns to the same phase; and two applies at different times with the same settings give the same waveform. `awg.zero_time()` (TCP `ZERO`) sets t_0 to the sample of the next change: phases then count from that change.

5.2 Applies

A change does not happen when the register is written but at an **apply**: the settings of both outputs, staged in pending registers, take effect together on one sample n_a, each at its own pipeline stage, so that the output samples from n_a on all follow the new settings. With the default `phase_mode "coherent"` the driver schedules the apply `apply_delay` (10 ms by default, 1 ms to 1 s) after the call and loads every phase accumulator so that its phase at n_a is $F\,(n_a - t_0)$ plus the set phase: the change is phase-coherent with t_0. With `"continuous"` it applies at once (within about $0.2\,\mu\text{s}$ of the write) and the accumulators run on: a frequency change is phase-continuous, without a jump.

Which to use

Coherent mode for anything that must keep phase relations (two outputs, I/Q, phase steps you set, repeated identical applies). Continuous mode for interactive tuning and frequency changes that must not jump in phase; the absolute phase then depends on when you changed.

5.3 What the outputs show while a change is applied

Nothing intermediate: the old settings up to sample $n_a - 1$, the new ones from n_a. A change that does not touch an output's waveform (for example only the other output, or a marker) leaves its phase running. Bursts, gates and triggered sweeps restart from idle at an apply; a running arbitrary-waveform sequence and a running digital pattern continue unless their own settings changed (sections 14.7 and 16.6).

5.4 Refused settings

A setting that cannot be made (a frequency above the Nyquist frequency, an edge shorter than one sample at that frequency, a peak beyond the output range, a sweep too fast, an unknown table)

is refused with a message and **changes nothing**: the outputs keep their previous settings. The messages name the setting and the limit:

```
ValueError: outputs.1.sweep is too fast: at most 953.7 Hz per sample (log: 1/2048 octave per sample)
```

6 Waveforms

6.1 Example: the shapes

```
awg.set_output(1, function="pulse", frequency=200e3, amplitude=0.25, offset=0.25,  # 0 V to 0.5 V
               width=1e-6, rise=50e-9, fall=200e-9)
awg.set_output(2, function="ramp", frequency=50e3, amplitude=0.4, symmetry=90.0)
```

TCP: `OUT 1 {"function": "pulse", "frequency": 2e5, "amplitude": 0.25, "offset": 0.25, "width": 1e-6, "rise": 5e-8, "fall": 2e-7}`

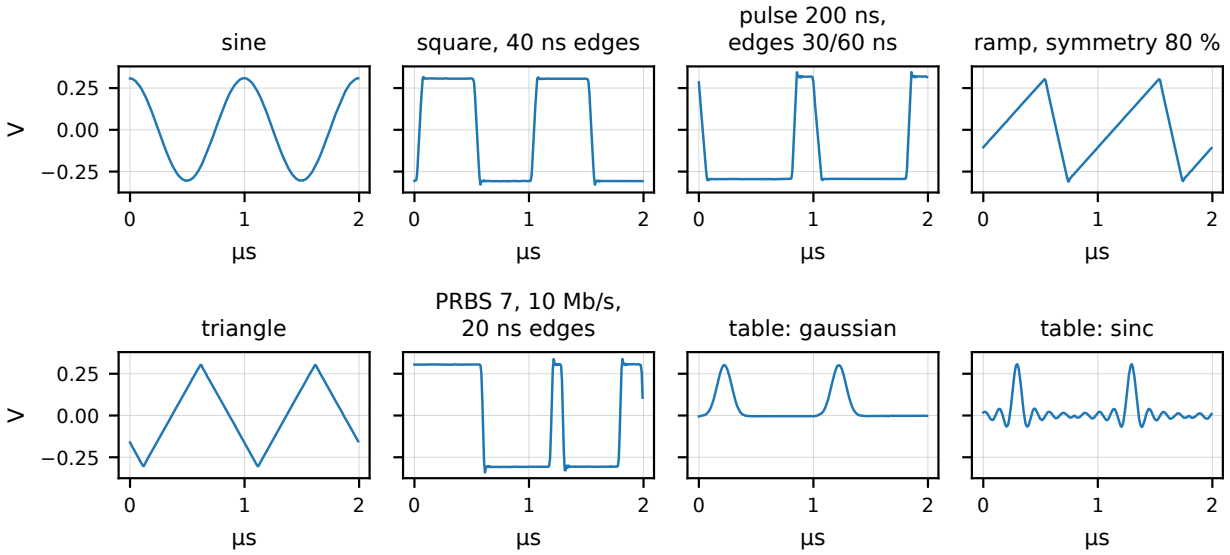


Figure 3: Measured: every shape of generator 1 at 1 MHz and 0.3 V, in loopback (OUT1 → IN1). The PRBS runs at 10 Mb/s, so its pattern repeats every 12.7 μs; the square's and pulse's edges have the set rise and fall times. Sharp corners show the overshoot of the analog path.

Generator 1 makes the output's main waveform (figure 3). The settings of one output, all optional in a patch:

key	default	meaning
`function`	`"off"`	`off`, `sine`, `square`, `pulse`, `ramp`, `triangle`, `prbs`, `arb` (a table shape), `dc` (the offset only)
`frequency`	1e6	Hz, 0 … 62.5 MHz (PRBS: the bit rate); steps of 0.44 μHz
`amplitude`	0.0	V, peak into 50 Ω: the waveform swings between offset ± amplitude
`offset`	0.0	V, DC added to everything
`phase`	0.0	degrees relative to t_0; in bursts, the start phase
`duty`	50.0	square: % of the period high, at the 50 % points
`width`	100e-9	pulse: s, at the 50 % points
`rise`, `fall`	0.0	s, 10–90 % edge times of square, pulse and PRBS; 0: the shortest edge (one sample)
`symmetry`	50.0	ramp: % of the period rising (triangle: always 50)
`order`	7	PRBS: 7, 9, 11, 15, 20, 23 or 31 bits
`table`	`"gaussian"`	function `arb`: a built-in or saved table

A phase of 0 puts the 50 % point of the rising edge (square, pulse, ramp, PRBS) or the start of the period (sine, table) at t_0, so phase differences between outputs are what you set whatever the shape. The `dc` function shows the offset only (use it for a level that must not move with amplitude settings).

6.2 Sine

The sine comes from a quarter-wave table of 1024 entries with first-order interpolation: its error is below 1.1 of the 18-bit shape's LSB, and its spurs are below −100 dBc before the 14-bit DAC rounding (computed). The DAC's rounding then sets the floor: about −86 dBc of quantization noise spread over the band for a full-scale sine (computed: $6.02 \cdot 14 + 1.76$ dB), less with dither (section 13.2). Sine amplitudes are corrected for the DAC's sinc droop and, if the board calibration has one, for the measured output response (`droop_correction`, on by default).

6.3 Square, pulse, ramp and triangle: edges

These shapes are trapezoids: a rising edge, a high part, a falling edge and a low part, each a fraction of the period. `rise` and `fall` are 10–90 % times as an oscilloscope measures them; the full edge (0–100 %) is 1/0.8 of it. The shortest edge is one sample (8 ns full, 6.4 ns 10–90 %), set by `rise` 0. Ramps have one edge of `symmetry` % of the period and one of the rest (at least one sample each).

The edges are computed from the exact phase: the sample that falls on an edge gets a value in proportion to where the edge is within it. After the DAC and any band-limiting, the edge moves continuously with the phase, in steps of 2^{-32} cycle, not in 8 ns steps. A square of 3.3 MHz (37.88 samples per period) therefore has the same period in every cycle, and a pulse's position can be set to a fraction of a nanosecond (figure 4). An edge shorter than one sample at the given frequency cannot be made this way and is refused.

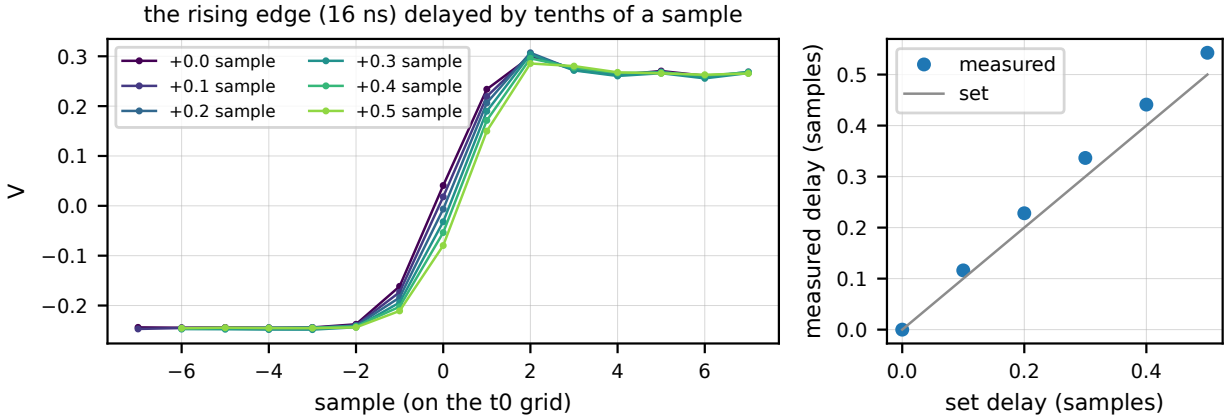


Figure 4: Measured: the rising edge of a pulse (125 kHz, 1000 samples per period, 16 ns edges) captured at six phase settings, each a tenth of a sample (0.8 ns) later. Left: the samples on the t_0 grid; right: the 50 % crossings (interpolated from the captured samples) against the set delay. They move 1.085 samples per sample set, within 0.005 samples of a line. The excess of about 8 % was also seen by the self-test (0.110 per 0.1 step); its cause, the interpolation of the captured samples or the analog path, is not established.

Duty, width and edges

The duty cycle and the pulse width are measured at the 50 % points, independently of the edge times: a 30 % square with slow edges is still high 30 % of the period at half amplitude. A width or duty that leaves no room for the edges is refused.

6.4 PRBS

`prbs` plays a pseudo-random bit sequence from the shift register $x^n + x^m + 1$ of ITU-T O.150 ($n = 7$, 9, 11, 15, 20, 23 or 31 bits, period $2^n - 1$ bits): one bit per period of `frequency` (the bit rate), the

levels offset ± amplitude, edges with the set `rise` (both edges alike; `fall` is not used), at sub-sample positions as above. In coherent mode the register is loaded at an apply with the state the sequence has at that time counted from t_0, so the bit pattern is coherent with t_0 just as a phase is: re-applying the same settings continues the same sequence. A burst restarts it from all ones. Generator 2 may be a PRBS too, as a modulation source or added noise-like signal.

6.5 Tables

Function `arb` plays one period of a 4096-point table, linearly interpolated between its points, at `frequency`. Built-in tables:

`gaussian`	a Gaussian pulse, centred, $\sigma = 1/16$ of the period
`lorentzian`	a Lorentzian pulse, centred, half width 1/32 of the period
`sinc`	$\sin x/x$, centred, zeros every 1/16 of the period
`exp_rise`, `exp_fall`	$1 - e^{-t/\tau}$, $e^{-t/\tau}$ with $\tau = 1/5$ of the period
`haversine`	$(1 - \cos 2\pi t)/2$, a raised-cosine pulse

Your own table is any list of at least 2 values in $[-1, 1]$ over one period; it is resampled to 4096 points (linear, periodic) and stored by name on the board:

```
import math
awg.save_table("two_bumps", [math.exp(-((i - 30) / 6) ** 2) - 0.5 * math.exp(-((i - 70) / 4) ** 2)
                             for i in range(100)])
awg.set_output(1, function="arb", table="two_bumps", frequency=20e3, amplitude=0.4)
```

TCP: `TABLE two_bumps [0.0, 0.1, ...]`

The table value 1 is `amplitude` volts above the offset. Tables are also the shapes of generator 2, so a table can be a modulation source (an arbitrary AM envelope or frequency profile). They are short and repeat with the period; for long, non-repeating waveforms use the arbitrary waveforms of section 14. `awg.table_names()` (TCP `TABLES?`) lists the built-in and saved tables.

6.6 Generator 2

Every output has a second generator, `aux`, with the same shapes and settings (`function`, `frequency`, `amplitude`, `phase` and the shape keys). It is

- **added** to the output with its own amplitude: two-tone tests, a sine on a square, a PRBS on a carrier;
- the **internal modulation source** (`"aux"`) of AM, FM, PM, PWM, and the key of FSK and BPSK, at its full scale whatever its amplitude.

Set its amplitude to 0 to use it only as a modulation source:

```
awg.set_output(1, function="sine", frequency=5e6, amplitude=0.3,
               aux={"function": "sine", "frequency": 7.5e6, "amplitude": 0.1})   # two tones
```

Generator 2 has no bursts and no sweeps; it runs continuously, coherent with t_0.

7 Noise

7.1 Example

```
awg.set_output(1, noise={"rms": 0.1, "bandwidth": 10e6})              # noise only
awg.set_output(2, function="sine", frequency=1e6, amplitude=0.3,
               noise={"rms": 0.02, "bandwidth": 2e6})                 # a sine with noise
```

TCP: `NOISE 1 0.1 10e6`

Each output adds Gaussian noise of `rms` volts with a flat spectrum up to `bandwidth` (the −3 dB frequency, 1 Hz to 47.7 MHz). The noise is made from Gaussian random numbers (a 64-bit generator and the inverse normal distribution, to 7.1 σ) at a point rate of bandwidth/0.382, interpolated to the sample rate by a 12-tap band-limiting filter: flat within ±0.5 dB to 0.31 of the point rate, −3 dB at the bandwidth, below −60 dB from 1.6 times the bandwidth (computed). Because the output is a weighted sum of independent Gaussian numbers it is Gaussian at any bandwidth.

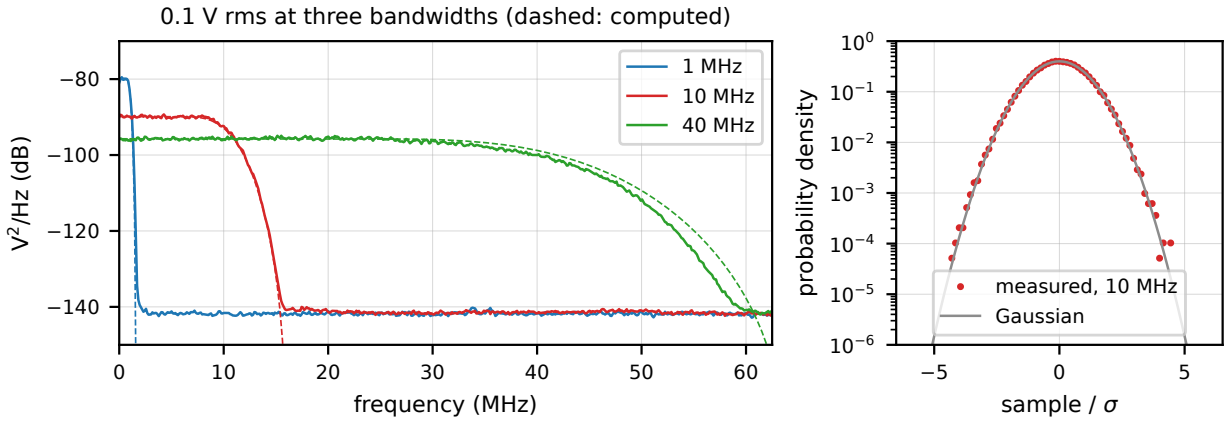


Figure 5: Measured: noise of 0.1 V rms at bandwidths of 1, 10 and 40 MHz, in loopback. Left: the spectrum (16 captures, 1024-point Hann segments), dashed: the computed filter shape. The −3 dB points read 1.00, 10.2 and 36.9 MHz; at 40 MHz the board's analog bandwidth (about 60 MHz) takes part. The floor near −142 dB is the inputs' own noise (1.27 mV rms over the band). Right: the distribution of 131,072 samples at 10 MHz against the Gaussian; kurtosis 3.033 (Gaussian: 3).

Noise and headroom

The driver counts noise at 4 σ when it checks that a setting stays within the output range: 0.25 V rms of noise uses ±1 V. With `allow_clip` you may set more and accept that the rare peaks are clipped. Noise is off on an output that plays arbitrary points (they share the filter).

Uses: noise injection into a control loop, dithering a modulator, testing a detector's noise rejection, a calibrated noise source for a spectrum analyzer: its density is σ^2/B per Hz below the bandwidth B (the filter's equivalent noise bandwidth equals B within 0.1 %, computed), so 0.1 V rms at 10 MHz is 10^{-9} V²/Hz, or 31.6 μV/$\sqrt{\text{Hz}}$.

8 Modulation

8.1 Example: AM and FM by generator 2

```
awg.set_output(1, function="sine", frequency=1e6, amplitude=0.3,
               aux={"function": "sine", "frequency": 50e3},              # the modulation source
               am={"mode": "am", "source": "aux", "depth": 0.5})          # 50 % AM
awg.set_output(2, function="sine", frequency=5e6, amplitude=0.3,
               aux={"function": "sine", "frequency": 100e3},
               angle={"mode": "fm", "source": "aux", "deviation": 500e3})  # +-500 kHz FM
```

TCP: `OUT 1 {"am": {"mode": "am", "source": "aux", "depth": 0.5}}`

Each output has one **amplitude** modulation (`am`) and, at the same time, one **angle** modulation (`angle`): FM, PM, PWM, FSK or BPSK. Both act on generator 1.

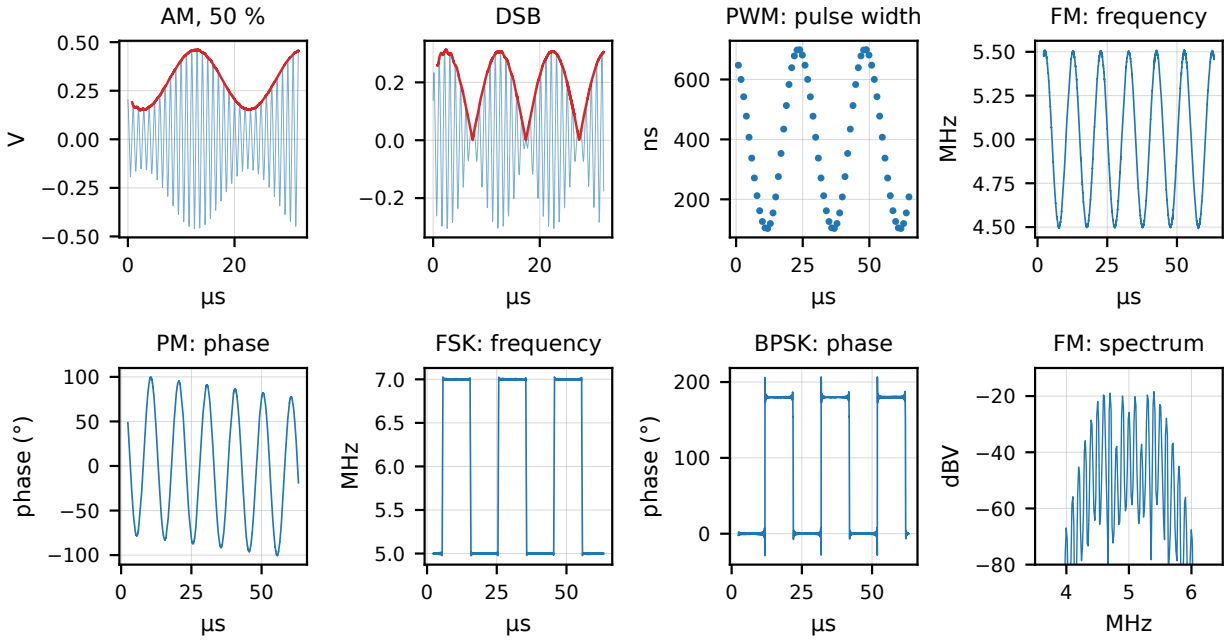


Figure 6: Measured in loopback, all by generator 2. Top: AM of 50 % (1 MHz carrier, 50 kHz modulation; red: the envelope), DSB with suppressed carrier, the width of PWM pulses (400 ns ± 300 ns at 40 kHz: from 103 to 699 ns), the instantaneous frequency of FM (5 MHz ± 500 kHz at 100 kHz: fitted 499.5 kHz). Bottom: the phase of PM (±90°: fitted 90.00°), the frequency of FSK between 5 and 7 MHz (5.00013 and 6.99995 MHz), the phase of BPSK (a step of 179.0°), and the spectrum of the FM signal (modulation index 5). The AM sidebands give a depth of 50.0 %.

8.2 Sources

`"aux"`
: generator 2 of the same output, at its full scale (its own amplitude does not matter): depth and deviation are reached at its peaks.

`"in1"`, `"in2"`
: an input, per volt: a depth of 0.5 with `in1` is 50 % AM for 1 V at IN1. The input's calibrated offset is removed, and input sample $n - 7$ modulates output sample n (56 ns). On the LV jumper the inputs reach about ±1.1 V.

`"other"`
: the other output's final value (after its sum and limits), per volt: OUT2 modulated by OUT1's waveform, 26 samples (208 ns) later.

8.3 AM and DSB

`am: {"mode": "off"|"am"|"dsb", "source": ..., "depth": ...}`.

- **AM**: generator 1 multiplied by $1 + d\,m$, with m the source (±1 at generator 2's peaks, or volts): depth 1 is 100 % AM. The driver counts the AM peak ($1 + |d|$ times the amplitude) in the headroom check.
- **DSB**: multiplied by $d\,m$ (suppressed carrier): a depth of 1 makes the amplitude's peaks at the source's peaks. With a source at frequency f_m and a carrier f_c the output contains $f_c \pm f_m$ only.

A burst's idle level is modulated too. AM by an input makes a voltage-controlled amplitude (an analog VGA with 14-bit resolution and a 56 ns delay).

8.4 FM, PM and PWM

`angle`: `{"mode": "fm"|"pm"|"pwm", "source": ..., "deviation": ...}`.

- **FM**: the frequency is $f + \Delta f\, m$, `deviation` Δf in Hz at the source's full scale (or per volt). Any deviation up to the Nyquist frequency; the phase stays continuous. Coherence with t_0 is kept only on average (the modulated phase depends on the modulation's history).
- **PM**: the phase is $\varphi + \Delta\varphi\, m$, `deviation` in degrees (up to $\pm 360°$).
- **PWM**: square and pulse only: the high part's length changes by `deviation` seconds at full scale. The falling edge moves (with its fall time); the rising edge stays where the phase puts it.

8.5 FSK and BPSK

`angle`: `{"mode": "fsk"|"bpsk", "key": ..., "hop": ..., "bpsk_phase": ...}`.

- **FSK**: the frequency is `hop` (Hz) while the key is on, the output's frequency otherwise; the switch is phase-continuous.
- **BPSK**: the phase is advanced by `bpsk_phase` (degrees, 180 by default) while the key is on.
- The **key** is `"aux"`, generator 2's first half cycle (its frequency is the keying rate: 50 kHz keys 10 μs on, 10 μs off), or the level of a pin or a threshold: `trig_a`, `trig_b`, `event0` ... `event3`, `in1`, `in2` (section 11.3). An external data signal on a pin thus keys the carrier directly; the self-test keys it with digital output D0.

Combinations

AM and one angle modulation work together (AM of an FM carrier, for example). A sweep excludes the angle modulations and bursts on the same output (the sweep drives the same frequency input). Arbitrary-waveform upconversion uses both modulators for the carriers and excludes them.

9 Sweeps

9.1 Example

```
awg.set_output(1, function="sine", amplitude=0.3,
               sweep={"mode": "continuous", "type": "log", "start": 1e3, "stop": 1e6,
                      "time": 10e-3, "hold": 1e-3, "return": 2e-3, "marker": 10e3})
awg.apply({"markers": {"1": {"source": "sweep", "channel": 1}}})   # DIO1_P: high below 10 kHz
```

A sweep changes generator 1's frequency in time. `sweep` holds:

key	default	meaning
`mode`	`"off"`	`continuous` (repeats) or `triggered` (once per trigger of the output, then holds the last frequency)
`type`	`"linear"`	`linear`, `log` (equal octaves in equal times), `list`, `profile`
`start`, `stop`	1e3, 1e6	Hz (linear and log)
`time`	1e-3	s from start to stop
`hold`	0.0	s at the stop frequency
`return`	0.0	s back to the start (a ramp of the same type); 0: a jump
`marker`	None	Hz: the sweep marker (marker source `"sweep"`) is high from the start up to this frequency
`points`	[ ]	list: `[[Hz, s], ...]`, each frequency for its dwell time
`profile`	[ ]	profile: `[[s, Hz], ...]`, straight lines between the points

The sweep is built in hardware from up to 512 straight pieces of the frequency (linear) or of its logarithm (log, through an exponential table with a relative error below 10^{-7}), each landing exactly on its end frequency. The frequency changes every sample, so a sweep has no steps. Limits: a linear

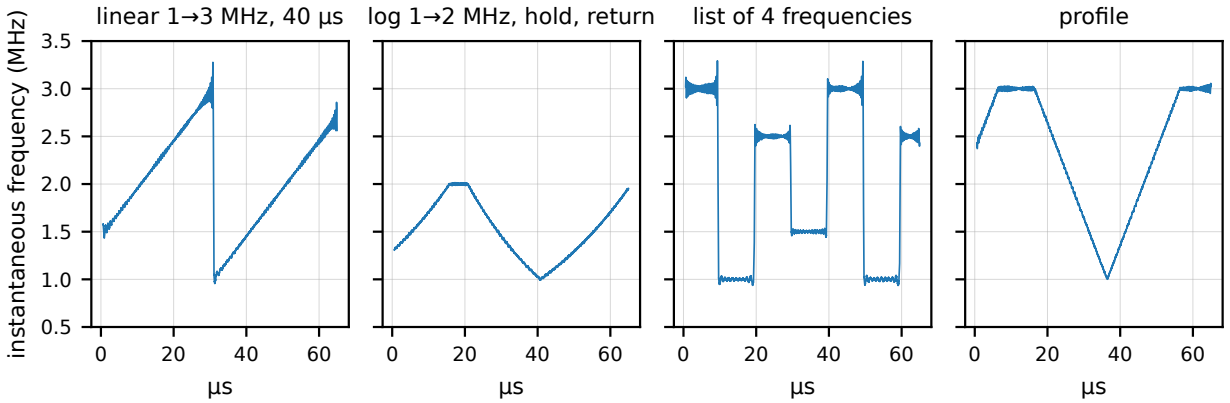


Figure 7: Measured: the instantaneous frequency of four repeating sweeps (from the analytic signal of the loopback capture). Linear $1 \to 3$ MHz in 40 μs: the rate is 5.0005×10^{10} Hz/s (set: 5) within 6.7 kHz rms of a line; logarithmic $1 \to 2$ MHz in 25 μs with 5 μs hold and a 20 μs logarithmic return; a list of four frequencies of 10 μs each (the spikes are the analytic signal at the phase-continuous jumps); a profile of three straight pieces.

piece changes at most 0.95 kHz per sample (119 GHz/s), a logarithmic one 1/2048 octave per sample (an octave in 16.4 μs or more); faster settings are refused with the limit in the message. The phase is continuous through the whole sweep, the jumps of a list included.

The sweep marker (`marker`) splits the forward sweep at that frequency: markers with source `"sweep"` are high from the start of the sweep up to it, so the marker's falling edge is the moment the frequency passes the marker frequency, for example to trigger an oscilloscope at a resonance. Invert the marker for the opposite sense.

Triggered sweeps start at the output's trigger (section 11), with the trigger's timing, run once, and hold the last frequency until the next trigger. A trigger during the sweep is missed. The phase runs on from where it was (a sweep changes only the frequency).

10 Bursts and gates

10.1 Example

```
awg.set_output(1, function="sine", frequency=1e6, amplitude=0.3, phase=90.0,
               burst={"mode": "triggered", "cycles": 3, "stop_phase": 180.0,
                      "idle": "dc", "idle_level": -0.1},
               trigger={"source": "timer1"})
awg.apply({"timers": {"1": {"period": 20e-6, "on": True}}})
```

In burst mode generator 1 is idle until a trigger, then plays from the start phase (`phase`) for a number of cycles and stops at the stop phase. `burst`:

key	default	meaning
`mode`	`"off"`	`off` (continuous), `triggered` (cycles per trigger), `gated` (while the gate is open)
`cycles`	1	triggered: 1 … $2^{32} - 1$ cycles, 0: endless (until the next apply)
`stop_phase`	None	degrees: the burst ends this far into the cycle after the last whole one (None: at the start phase, i.e. after whole cycles)
`idle`	`"start"`	between bursts: `start` (the value at the start phase), `last` (the value where it stopped), `dc` (`idle_level`)
`idle_level`	0.0	V above the offset, within ± the amplitude
`gate`	`"trig_b"`	gated: the level source: `trig_a`, `trig_b`, `event0`–`event3`, `in1`, `in2`
`polarity`	`"high"`	gated: open while the source is high (or low)
`finish`	True	gated: at closing, finish the cycle (to the stop phase) or stop at once

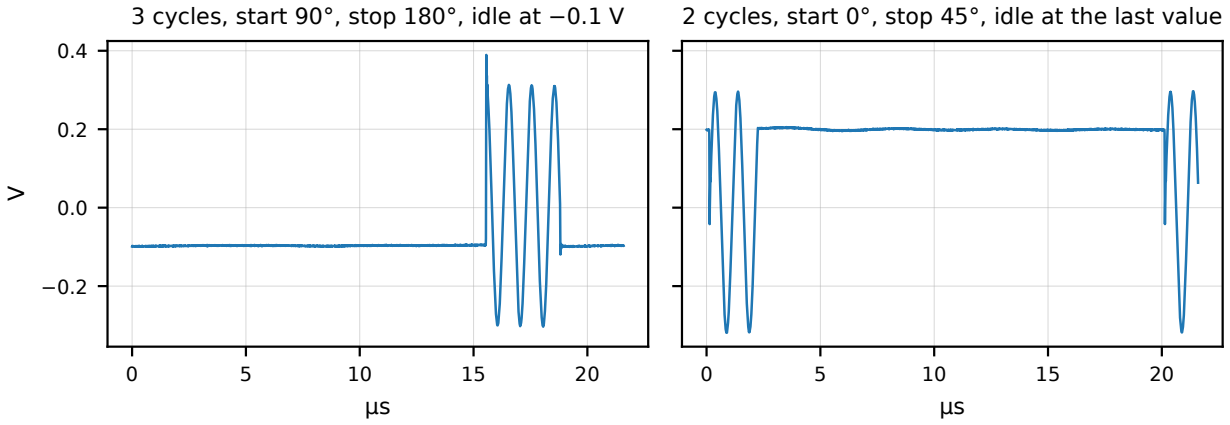


Figure 8: Measured: bursts of a 1 MHz sine started by timer 1 every 20 μs. Left: the example above: 3 cycles from a start phase of 90° (the sine's peak) to a stop phase of 180°, idling at −0.1 V between bursts. Right: 2 cycles from 0° to 45°, idling at the last value. The ringing at the start is the analog path's response to the step from the idle level.

The burst counts in phase, not in time: `cycles` 3 is exactly three periods of the (possibly modulated) frequency, so an FM or FSK burst also ends on its phase. With `idle start` the waveform starts without a step (the idle level is the start value); with `dc` or `last` it may step at the start. Generator 2, the noise and the arbitrary points are not burst; AM continues on the idle level.

Gates. In `gated` mode the burst runs while the gate source is at `polarity`: an opening edge starts it like a trigger (with the trigger's delay and timing), a closing edge stops it at its next stop phase (`finish`) or at once. A gate already open at an apply starts the burst at the apply.

11 Triggers

11.1 Sources

Each output's `trigger` is `{"source": ..., "edge": ..., "delay": ...}`. It starts the output's bursts and triggered sweeps (and, for OUT1, the arbitrary-waveform sequencer if that is triggered by OUT1's settings: section 14.3).

source	meaning
`trig_a`, `trig_b`	DIO0_P, DIO3_P (3.3 V, pull-down): edges timestamped to 1 ns
`event0` … `event3`	DIO4_P–DIO7_P (3.3 V, pull-down): edges on the sample grid (8 ns)
`in1`, `in2`	the input crossing a threshold (section 11.3): on the sample grid
`software`	`awg.trigger(n)` (TCP `TRIG n`), at the clock of the register write
`timer1`, `timer2`	periodic events, period 16 ns to years, aligned to t_0
`other`	the other output's start (its burst, gate or sweep start)

`edge` is `rising`, `falling` or `both`; `delay` adds 0 to days in steps of 1/256 sample (31.25 ps). A trigger that comes while the burst runs (or the triggered sweep runs or waits for its start) is **missed** and counted: `awg.hardware()` reports per output the accepted `triggers`, the `missed` ones and the last start time.

```
awg.set_output(1, function="square", frequency=2e6, amplitude=0.2,
               burst={"mode": "triggered", "cycles": 10},
               trigger={"source": "trig_a", "edge": "rising", "delay": 2.5e-6})
hw = awg.hardware()["outputs"]["1"]
print(hw["triggers"], hw["missed"], hw["last_start"])     # starts, missed triggers, last start (
    ↪ samples)
```

11.2 Timing

Every event has a time τ in ticks of 1/256 sample. A burst (or sweep, or sequence) starts at

$$T = \tau + 20 \text{ samples} + \text{delay}, \tag{2}$$

exactly: its first sample is the first one at or after T, and generator 1 starts there at the phase it would have had at T. The output's time origin is therefore T to 31 ps, and a start triggered on TRIG A or TRIG B has only the 1 ns quantization of its timestamp (0.29 ns rms for edges uniformly distributed in time, computed). EVENT pins and the thresholds are seen on the 8 ns sample grid.

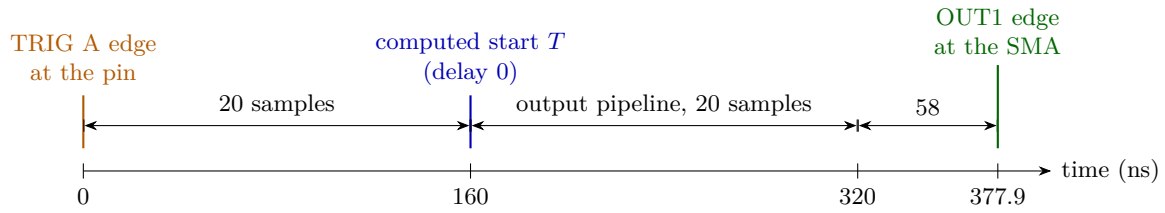


Figure 9: Measured: the trigger-to-output latency with delay 0 is 377.9 ns from the TRIG A pin (at 1.65 V) to OUT1's edge at its connector (oscilloscope, 2026-10-24): 160 ns to the computed start, 160 ns through the output pipeline, and about 58 ns for the input pin, the DAC interface, the converter and the analog output stage. Set a trigger delay to place an output after its trigger; a negative delay is not possible.

Jitter

Against a trigger from outside, the start inherits the timestamp's 1 ns quantization (computed). With a trigger made by awg itself (a marker or digital output wired to a trigger pin) the edge always falls at the same point of the sample, and the start repeats to the timestamp's resolution: the self-test reads the same start time in every read. The jitter against an asynchronous external trigger has not been measured.

11.3 Thresholds on the inputs

`inputs`: `{"in1": {"high": 0.1, "low": -0.1}, "in2": {...}}` in volts at the input. An input's level becomes high when the input rises above `high` and low when it falls below `low`: the difference is the hysteresis against noise. The level is a trigger source (`in1`, `in2`: its rising or falling edges), a gate source and an FSK/BPSK key.

```
awg.apply({"inputs": {"in2": {"high": 0.1, "low": -0.1}},
           "outputs": {"1": {"function": "sine", "frequency": 2e6, "amplitude": 0.2,
                             "burst": {"mode": "triggered", "cycles": 2},
                             "trigger": {"source": "in2", "edge": "rising"}}}})
```

A slow input crosses the threshold with its noise: a 20 kHz sine of 0.4 V moves 0.4 mV per sample against about 1 mV of input noise, which moves the decision by a few samples. Use hysteresis and fast edges where the timing matters.

11.4 Timers

`timers`: `{"1": {"period": 1e-3, "on": False}, "2": {...}}`. A timer makes an event every `period` (16 ns to years, in ticks of 31 ps), on the grid $t_0 + k\cdot$ period: its events keep their phase relative to t_0 through applies, like the generators. Two outputs triggered by the same timer start together; a timer period that is a multiple of a generator's period gives bursts of identical phase.

11.5 The other output

Source `other` triggers an output by the other one's start: a burst of OUT2 that follows each burst of OUT1, a fixed delay later (`delay`), or a burst started by OUT1's sweep.

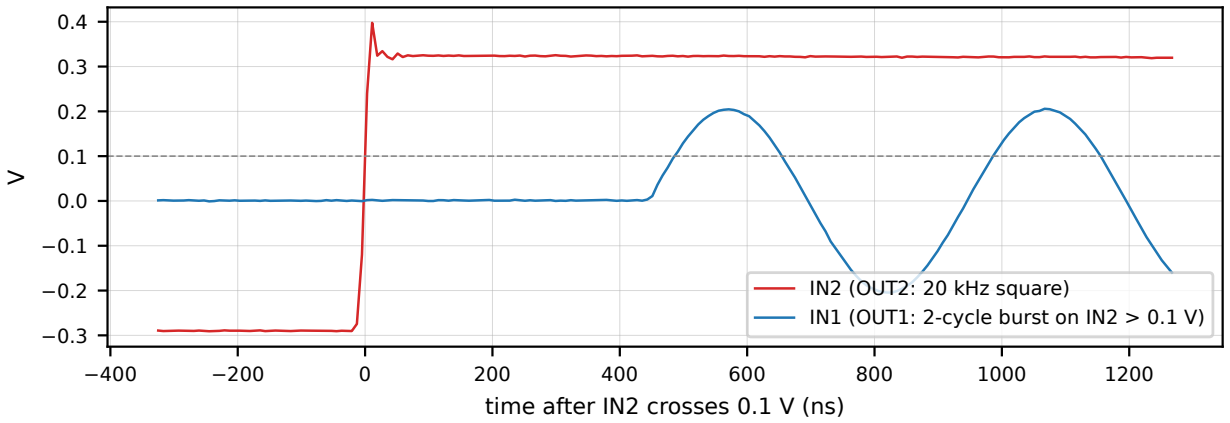


Figure 10: Measured: OUT2 (a 20 kHz square) into IN2 triggers 2-cycle bursts of OUT1 on its threshold at 0.1 V. In loopback the burst appears 467 ns after IN2 crosses the threshold (the comparator, the trigger, the output pipeline and both analog paths). The self-test counts one burst per edge (0.99989 of the edges) with the same delay to 0.005 sample.

12 Markers

Two 3.3 V outputs, MARKER 1 (DIO1_P) and MARKER 2 (DIO2_P), show a source of one output. `markers: {"1": {"source": ..., "channel": 1, "invert": False, "delay": 0.0, "width": 1e-6}, "2": {...}}`.

source	high while …
`low`, `high`	never, always (a static level)
`sync`	the first half of generator 1's cycle (a square at its frequency, phase 0 at the rise)
`run`	the burst runs (always in continuous mode); arbitrary points: a played (not idle) point
`sweep`	the sweep is below the marker frequency (section 9)
`start`	a pulse of `width` (8 ns to 34 s) from each start: a burst, a sweep, a sequence, or a step with its marker flag
`arb1`, `arb2`	the marker bits 1, 2 stored with the output's arbitrary points

`channel` is the output (1 or 2) whose source is shown, `invert` inverts it.

Delay and alignment. With `delay` 0 a marker changes with the analog sample it belongs to: the start pulse with the burst's first sample at the connector, a point marker with its point. `delay` moves it in 8 ns steps, from about −160 ns to +3.9 µs. Inside the FPGA the marker is aligned with the sample; at the connectors the analog path's delay is corrected by the board calibration's `pins_delay_s` (section 17.5). On the board used here the pins lead OUT1 by 28.6 ns; the driver shifts them by four samples, so a marker at delay 0 changes 3.3 ns after the analog edge, the closest the 8 ns grid allows. The applied delay (`awg.config()["markers"]`) reports this residual.

Markers as triggers

A marker wired to a trigger pin is a trigger with a fixed delay: the self-test wires MARKER 1 to TRIG A and starts OUT1 46.25 samples after OUT2's start (the marker's position, the pin and input paths, and the 20-sample trigger delay), the same in every read.

13 Limits, protection and dither

13.1 Voltage limits

```
awg.set_output(1, function="sine", frequency=1e6, amplitude=0.4,
               limits={"low": -0.15, "high": 0.25})
```

Each output clamps every sample to `limits` `low` … `high` (V; None: the DAC's range), after all sums, modulation, noise, arbitrary points and dither: nothing the generators do can go beyond them. Clamped samples are counted (`awg.hardware()["outputs"]["1"]["clipped_samples"]`). Set the limits from the range a connected device tolerates.

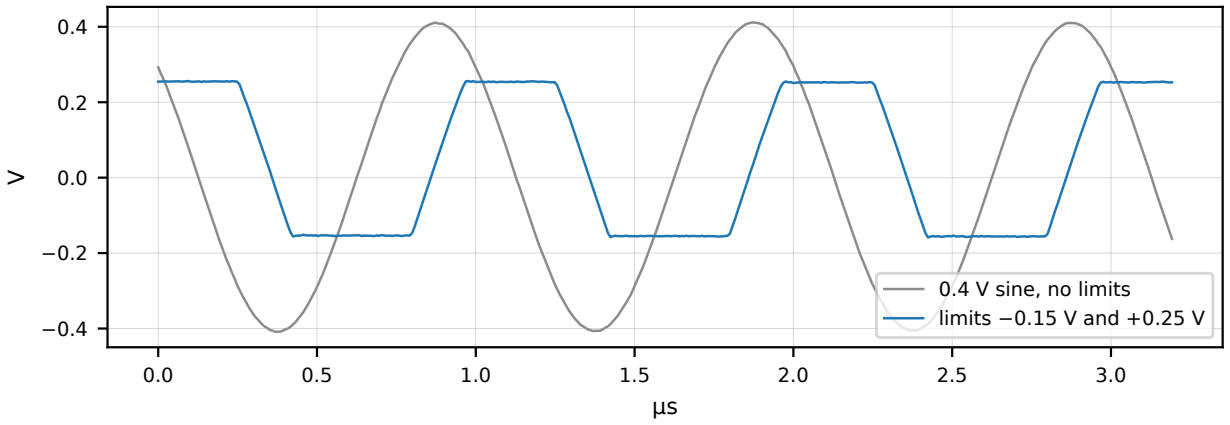


Figure 11: Measured: a 0.4 V sine with limits at −0.15 and +0.25 V. The clamped levels read −0.157 and 0.256 V (1st and 99th percentiles). The loopback itself reads a DC level 3.6 % low and a sine 2.5–3 % high on this board, so a clipped peak (fast edges) reads above the limit; the clamp holds the DAC code exactly.

Protection. `awg.protect(n)` (TCP `PROTECT n`, or `protect` True in the limits) locks output n's limits until the FPGA is reloaded: no configuration, preset, `SAFE` or calibration can widen them, and a setting with other limits is refused. Use it to protect an expensive device from mistakes, including those of other programs. `SAFE` mutes to 0 V, or to the nearest limit if 0 V is outside a protected window.

Headroom. Without limits the driver refuses settings whose worst case leaves the DAC's range (`allow_clip` False); `awg.status()["outputs"]` reports each output's peak and remaining headroom in volts.

13.2 Dither

`dither` True adds triangular random dither of ±1 DAC step (about ±138 µV) to every sample before rounding. Without dither, the DAC's rounding error is correlated with the waveform (harmonics of a small sine, steps in a slow ramp, a DC offset off by up to half a step); with dither it becomes uncorrelated noise (about 0.5 DAC step rms in total, against 0.29 without, computed) and averages to the exact value: DC levels and slow ramps between DAC steps, and small signals without quantization harmonics.

14 Arbitrary waveforms

14.1 Example: a chirp

```
import math
n = 4000
chirp = [math.sin(2 * math.pi * 40 * (i / n) ** 2) * math.sin(math.pi * i / n) ** 2 for i in range(n)
    ↪ ]
awg.save_segment("chirp", chirp, markers=[1 if i < 40 else 0 for i in range(n)])  # marker 1 at the
    ↪ start
awg.play(sequence="chirp", rate=100e6 / 1.37)            # every point once, 73 MSa/s
awg.set_output(1, arb={"on": True, "amplitude": 0.3, "filter": "smooth"})
```

```
TCP: SEGMENT chirp (binary data)
TCP: PLAY {"sequence":  "chirp", "rate":  72992700}
TCP: ARB 1 {"on":  true, "amplitude":  0.3}
```

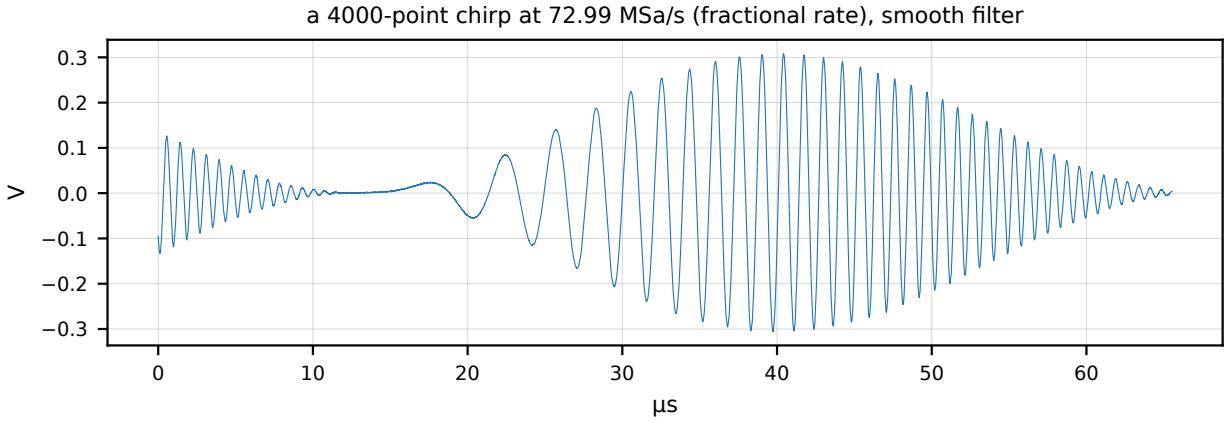


Figure 12: Measured: the example's 4000 points at 72.9927 MSa/s, a rate that is not a fraction of the sample rate; the segment repeats every 54.8 μs. Every point is played once, and the smooth filter places the points between the samples at their exact times.

Arbitrary waveforms in awg are **points played at a rate**, not a table looked up by a phase: every point you store is played exactly once per repetition, in order, at the **point rate** you set (any rate up to 125 MSa/s, in steps of 2^{-32} of the sample rate, 0.03 Sa/s), and reconstructed between the samples by a filter. The points come in **pairs**, one for OUT1 and one for OUT2, played by one **sequencer** from the board's memory (16 million pairs). Each output takes its half of the pairs if you switch its `arb` on; it then plays them in place of its noise, with its own amplitude, offset and filter, summed with its generators.

14.2 Segments

A **segment** is a named list of at least 64 pairs (up to $2^{25} - 1$), stored on the board's SD card:

```
import math
awg.save_segment("ramp", [i / 100 - 0.64 for i in range(128)])          # OUT2 gets the same
awg.save_segment("iq", [math.cos(2 * math.pi * k / 16) for k in range(1600)],
                 values2=[math.sin(2 * math.pi * k / 16) for k in range(1600)])  # OUT1 I, OUT2 Q
print(awg.segment_names(), awg.segment_values("ramp", 0, 3)["1"]["values"])
```

- **Values** are in $[-1, 1]$, stored with 14 bits (± 8191): the step is 1.2×10^{-4} of full scale. The output's `amplitude` (V) is the voltage of the value 1.
- `values2` are OUT2's points (default: the same as OUT1's).
- `markers` and `markers2`: per point an integer, bit 0 marker 1 and bit 1 marker 2: the marker sources `arb1` and `arb2` show them, aligned with the point (section 12).
- Lists or numpy arrays: with numpy on the board, 4 million pairs take about 5 s to convert; plain lists take about 34 μs per pair.

- From a file on the board: `awg.load_segment_file(name, path)` reads `.npy` or CSV (columns out1 [, out2 [, markers1 [, markers2]]]). From a PC, the client sends the pairs as binary data (TCP `SEGMENT`): 8–12 MB/s over the network, a million pairs (4 MB) in about half a second.
- `segment_names()`, `segment_values(name, start, count)`, `delete_segment(name)` (a segment, or a sequence, that the applied configuration plays is not deleted: turn the arbitrary waveform off or play another sequence first).

A segment name also works where a sequence is expected: it plays forever.

14.3 The sequencer

The top-level `arb` holds the sequencer's settings (`awg.play(**settings)`, TCP `PLAY`):

key	default	meaning
`sequence`	`""`	a saved sequence, or a segment (played forever)
`rate`	125e6	pairs per second (every pair once), up to 125e6
`run`	`"continuous"`	`continuous` (starts 20 samples + delay after the apply), `triggered` (each trigger starts the sequence and starts waiting steps), `gated` (plays while the gate is open)
`trigger`	software, rising, 0	source, edge and delay of the starts (as an output's trigger, section 11)
`gate`	trig_b, high, finish	gated: level source, polarity, and whether a closing gate finishes the repetition
`jump`	event0, rising	the source and edge of the steps' jump events

The sequencer uses OUT1's trigger hardware: while it plays, OUT1 has no bursts or sweeps of its own (its generators and its noise, when it does not take points, are free). OUT2 may play its half alone.

A start (trigger, gate, or apply) at $T = \tau + 20$ samples + delay plays pair k of the first step at time $T + k/r$ exactly (to 31 ps): the arbitrary waveform has the trigger's sub-sample timing like a burst.

14.4 Filters

Each output reconstructs its points with a `filter`:

`step`
: each point held for $1/r$: a staircase, no delay. For digital-like signals and for points at the full rate (125 MSa/s, where all filters are equal).

`linear`
: straight lines between points, a delay of one point.

`smooth`
: the band-limiting filter of the noise: flat to 0.31 of the rate, −3 dB at 0.382, below −60 dB from 0.6 (computed): images of the point rate are removed, and the points sit at their exact times between the samples. A delay of six points. The points' markers are delayed alike, so they stay with their points.

The output's settings (`arb` in the output, `awg.set_output(n, arb={...})`, TCP `ARB n`): `on`; `filter`; `amplitude` (V at value 1); `point_offset` (−1 … 1, added to every point, the idle ones too); `idle` and `idle_level`: what the output shows before the sequence starts and after it ends: `start` (the first point), `last` (the last point played) or `dc` (`idle_level`); and `upconvert` (section 15). Noise and sweeps are off on an output with `arb` on; generator 1 and generator 2 stay summed.

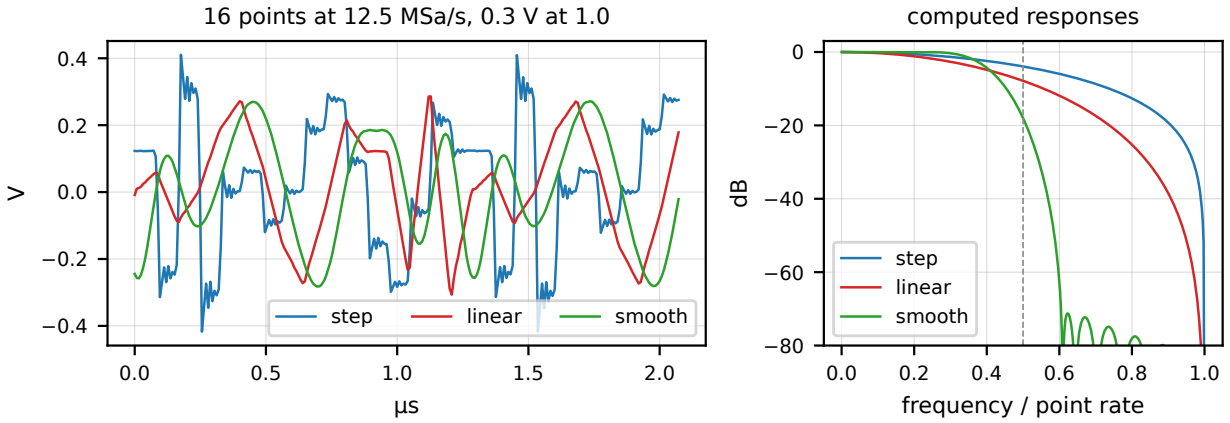


Figure 13: Left, measured: the same 16 points at 12.5 MSa/s (10 samples per point) with each filter (traces shifted in time by the filters' delays; the step filter's edges ring in the analog path). Right, computed: the filters' amplitude responses against frequency in units of the point rate; the dashed line is half the point rate, above which the step and linear filters pass images.

14.5 Sequences

A **sequence** is a list of up to 512 steps; each plays a segment:

```
import math
awg.save_segment("idle", [0.0] * 512)
awg.save_segment("tone", [math.sin(2 * math.pi * i / 64) for i in range(6400)])
awg.save_sequence("demo", [
    {"segment": "idle", "repeat": 0, "on_event": "end", "event_target": 1},  # until a jump event
    {"segment": "tone", "repeat": 3, "marker": True, "next": "end"}])         # 3 tones, then idle
awg.play(sequence="demo", rate=125e6, run="triggered", trigger={"source": "trig_a"},
         jump={"source": "event0", "edge": "rising"})
awg.set_output(1, arb={"on": True, "amplitude": 0.3, "filter": "step"})
```

TCP: `SEQUENCE demo [{"segment": "idle", "repeat": 0, ...}, ...]`

key	default	meaning
`segment`		the segment's name
`repeat`	1	times ($1 \ldots 2^{32}-1$); 0: until an event jumps away, or forever
`wait`	False	wait for the sequencer's trigger before the step
`wait_level`	0.0	both outputs while waiting ($-1 \ldots 1$), or `"hold"` (the last points)
`marker`	False	a start pulse on markers with source `start` at the step's first point
`next`	`"follow"`	then: the next step, `"end"` (idle; a trigger restarts at step 0), or a step index (goto)
`on_event`	`"none"`	on a jump event: `"end"` jump at the end of the repetition, `"now"` at once
`event_target`	0	the step to jump to
`digital`	`"none"`	start a pattern program (by name) or `"stop"` at the step's first point (section 16.5)

The last step needs `next "end"` or a goto. A plain segment name is a step with the defaults.

The rules of jumps. A jump event (from `jump`'s source) counts for the step that plays when it arrives; the jump happens at the end of the repetition (`end`) or at the next point (`now`), but at least 16 samples (128 ns) after the step's first point. There is **no gap**: the target's first point follows the last point played, at the same rate, because the sequencer keeps the first points of every event target ready. A sequence may jump to at most 3 different steps besides step 0.

The hot cache. The first step and the event targets play their first 512 pairs from a cache in the FPGA while the memory reads restart behind them; a step of at most 512 pairs plays from the

cache in all its repetitions. The memory must deliver the rest within the cached pairs' time: 4.1 μs at 125 MSa/s, ample at lower rates.

Waiting steps. A step with `wait` holds both outputs at `wait_level` (or their last points) until the sequencer's trigger, then starts with the trigger's sub-sample timing. In a `gated` sequence the wait flags are ignored.

14.6 Memory and streaming

The pairs are read from a 64 MB buffer in the processor's memory (16 million pairs; 32 or 16 MB if less contiguous memory is free) through a 64-bit port, into a FIFO of 4096 pairs (33 μs at 125 MSa/s). At 125 MSa/s both outputs read 500 MB/s: measured on the board, both outputs streamed a 16 MB segment for 10 s without a gap, also while the processors copied memory, the FIFO never below 491 of its 2048 words (7.9 μs of reserve). Segments are placed in the buffer at the first apply that uses them (a 16 MB segment in about 2.5 s) and stay there: a later apply that keeps them takes about 0.10 s. If a pair ever came late the sequence would stop and `awg.hardware()["arb"]["underflow"]` would report it.

14.7 Applies, restarts and status

An apply that changes the sequence, its settings, or an output's point settings restarts the sequence (at step 0, waiting for its trigger if it is triggered). An apply that changes anything else (an amplitude, a generator, a marker) lets it play on without a gap. `awg.hardware()["arb"]` reports the sequencer's `state` (`idle`, `waiting`, `playing`), the `step` being read, `underflow` and the FIFO's fill.

15 I/Q

15.1 Baseband I/Q

Put I in the pairs' OUT1 values and Q in their OUT2 values (`values2`): OUT1 plays I and OUT2 plays Q, sample-synchronous (they share the sequencer's position), each with its own `amplitude` (the gain balance) and offsets (`point_offset` for the I/Q offsets at the mixer, the output offset for DC). This drives an external I/Q modulator.

15.2 Upconversion in the FPGA

```
import math
n = 4000
awg.save_segment("ssb", [math.cos(2 * math.pi * k / 8) for k in range(n)],
                 values2=[math.sin(2 * math.pi * k / 8) for k in range(n)])
awg.play(sequence="ssb", rate=1e6)                                  # a 125 kHz tone as I + jQ
awg.set_output(1, arb={"on": True, "filter": "smooth", "amplitude": 0.3,
                       "upconvert": {"on": True, "frequency": 125e6 / 64}})
awg.set_output(2, arb={"on": True, "filter": "smooth", "amplitude": 0.0})  # (Q's half, unused here)
```

With `upconvert` on, output n is

$$\mathrm{OUT}n = A\,(I\cos(2\pi f t + \varphi) - g\,Q\sin(2\pi f t + \varphi + \varepsilon)), \tag{3}$$

with I its own half of the pairs, Q the other output's half (which must be `on`, with amplitude 0 if you do not want it on its own output), f `frequency` (0 to 50 MHz), φ `phase`, g `q_gain` and ε `q_phase` (degrees). The carriers are generators 1 and 2 of the output, phase-coherent with t_0; AM and the angle modulations are not available then. A single-sideband signal at $f + f_m$ results from $I = \cos 2\pi f_m t$, $Q = \sin 2\pi f_m t$ (figure 14).

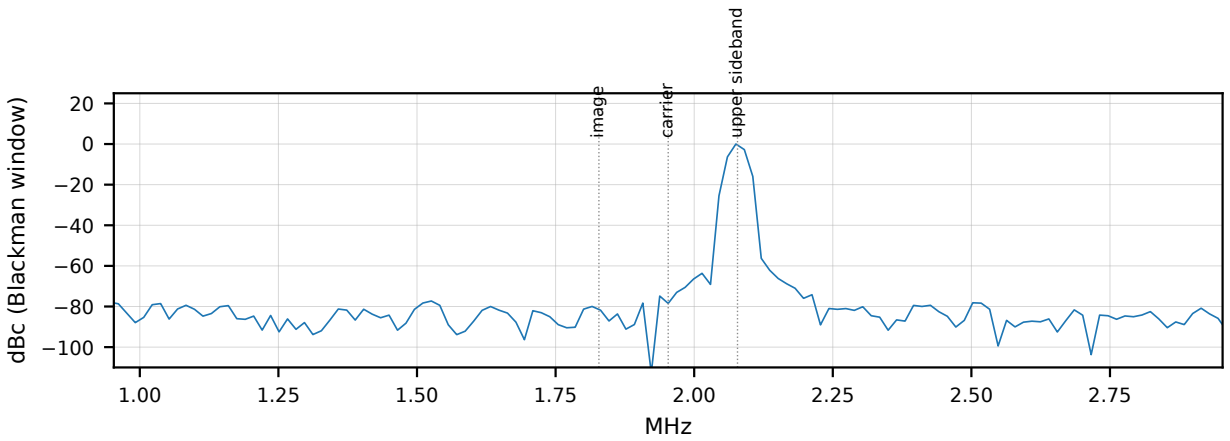


Figure 14: Measured: the example's single-sideband signal (a 125 kHz tone upconverted to 1.953 MHz), spectrum of a loopback capture. The image at $f - f_m$ is −89.7 dBc and the carrier −90.6 dBc, both at the capture's noise floor (the self-test: −91.6 and −91.5 dBc); the wanted sideband reads 0.3069 V for 0.3 V.

Correcting an external modulator. With baseband I/Q into an external mixer, the classic impairments are corrected here: the gain imbalance by the two outputs' amplitudes, the I and Q DC offsets (carrier leakage) by `point_offset` or the output offsets, and a quadrature error by mixing a little I into Q in the points themselves. With the internal upconversion, `q_gain` and `q_phase` trim the carriers' balance.

16 Digital outputs

16.1 Example

```
awg.save_pattern("steps", [0x01, 0x03, 0x07, 0x0F, 0x1F, 0x3F, 0x7F, 0xFF])
awg.set_digital(run="continuous", program="walk", enable=0xFF,
                programs={"walk": {"mode": "memory", "pattern": "steps", "rate": 12.5e6}})
print(awg.describe()["D"])
```

TCP: `PATTERN steps (binary data)`

TCP: `DIGITAL {"run": "continuous", "program": "walk", "enable": 255, "programs": {...}}`

Eight 3.3 V outputs, **D0**–**D7** on DIO0_N–DIO7_N (the second row of the extension connector E1), show one 8-bit **word** per sample: bit k drives Dk. A **pattern engine** plays **programs**: a saved pattern, a binary or Gray count, or a PRBS, at 125 MHz divided by an integer, for a number of passes or forever. A program starts at an apply, at each start of an output, or on the first point of a sequence step; its words change on the sample of the analog output they belong to.

16.2 The settings

`digital` (`awg.set_digital(**settings)`, TCP `DIGITAL`):

key	default	meaning
`run`	`"off"`	`off` (the idle word), `continuous` (`program` from the apply on), `sequencer` (programs started and stopped by sequence steps), `out1`, `out2` (`program` at each start of that output)
`program`	`""`	the program of `continuous`, `out1`, `out2`
`programs`	{}	up to 8 programs by name; a patch replaces the whole set
`idle`	0	the word when no program plays (0 … 255)
`enable`	0	the pins driven (bit k: Dk); the others are high impedance
`delay`	0.0	s relative to the analog outputs, 8 ns steps

High impedance

A pin not in `enable` is not driven (high impedance, no pull resistor): it floats unless the connected input has a pull resistor. Loading awg, `SAFE` and the default configuration leave all eight pins high impedance; enable only pins whose other end is an input.

16.3 Programs

A program:

key	default	meaning
`mode`	`"memory"`	`memory` (a saved pattern), `binary`, `gray` (counts), `prbs`
`pattern`	`""`	memory: the pattern's name
`start`	0	memory: the index of the first word; counts: the first value (0 … 255)
`length`	0	words per pass; 0: memory to the pattern's end, counts 256, PRBS one period
`rate`	125e6	words per second: 125 MHz divided by an integer (the applied value is exact)
`repeats`	0	passes; 0: forever
`end`	`"idle"`	after the last pass: `idle` (the idle word) or `hold` (the last word)
`order`	7	PRBS: 7, 9, 11, 15, 20, 23, 31

`memory`
: plays the pattern's words from `start`, `length` of them per pass, each pass from `start` again. **Patterns** are saved lists of 1 to 32768 words (`awg.save_pattern(name, values)`, integers 0 … 255 or bytes; TCP `PATTERN` with binary data, 32 kB in about 14 ms); the programs in use share the 32768-byte pattern memory. A pattern a program uses is not deleted.

`binary`, `gray`
: count from `start`, one count per word, `length` counts per pass, wrapping at 255. In Gray code exactly one bit changes per word, also at the wrap. A binary count makes divided clocks: Dk is a square wave of rate/2^{k+1}.

`prbs`
: the PRBS of the PRBS shape (ITU-T O.150), one step per word, starting from all ones: D0 is the bit sequence, Dk the same sequence k words earlier (D1–D7 are delayed copies, for testing a receiver's bit alignment or a deserializer). Bits at and above the order stay 0 (PRBS 7 leaves D7 low).

16.4 Starting and stopping

`continuous`
: the program starts on the apply's sample and plays its passes. The driver restarts it at an apply only if the run mode, the program or the programs changed: enabling a pin or changing the idle word leaves it running.

`out1`, `out2`
: the program starts at each start of that output: a burst or gate opening, a sweep start, the start of an arbitrary-waveform sequence or of a step with its marker flag (the `start` marker's pulses). A start while it plays starts it again from its first word.

`sequencer`
: programs are started and stopped by sequence steps.

`off` the idle word.

16.5 Patterns that follow a sequence

A sequence step's `digital` names a program to start, or `"stop"`, on the step's first point:

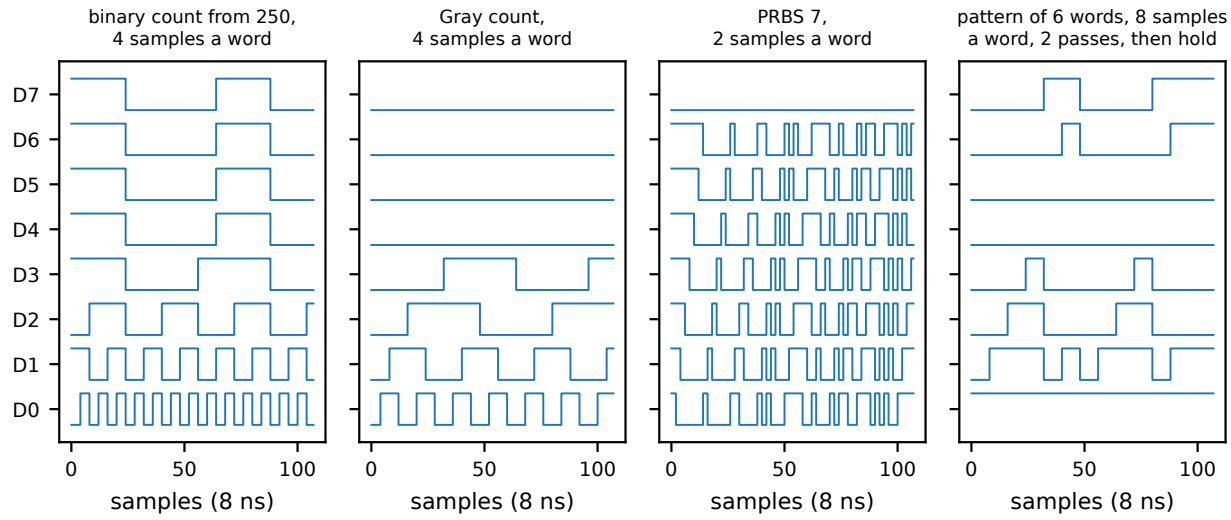


Figure 15: Computed with the model: the words of four programs started at an apply. A binary count from 250 wraps to 0 after 255; a Gray count changes one bit per word; in the PRBS each line is the line below delayed by one word; a pattern of 6 words, 2 passes, then holding its last word.

```
awg.save_segment("ramp2", [i / 32 - 1 for i in range(64)])
awg.save_segment("level", [0.6] * 64)
awg.save_sequence("with_count", [{"segment": "ramp2", "repeat": 2, "digital": "count"},
                                 {"segment": "level", "digital": "stop", "next": 0}])
awg.apply({"arb": {"sequence": "with_count", "rate": 62.5e6},
           "outputs": {"1": {"arb": {"on": True, "amplitude": 0.3, "filter": "step"}}},
           "digital": {"run": "sequencer", "enable": 0x0F,
                       "programs": {"count": {"mode": "binary", "rate": 125e6 / 8}}}})
```

The action takes effect on the sample whose analog output shows the step's first point, after the filter's delay of the output it follows (OUT1 if OUT1 plays points, else OUT2): words and points stay together at any rate and with any filter (figure 16). A jump into a step performs its action, a repetition of the step does not.

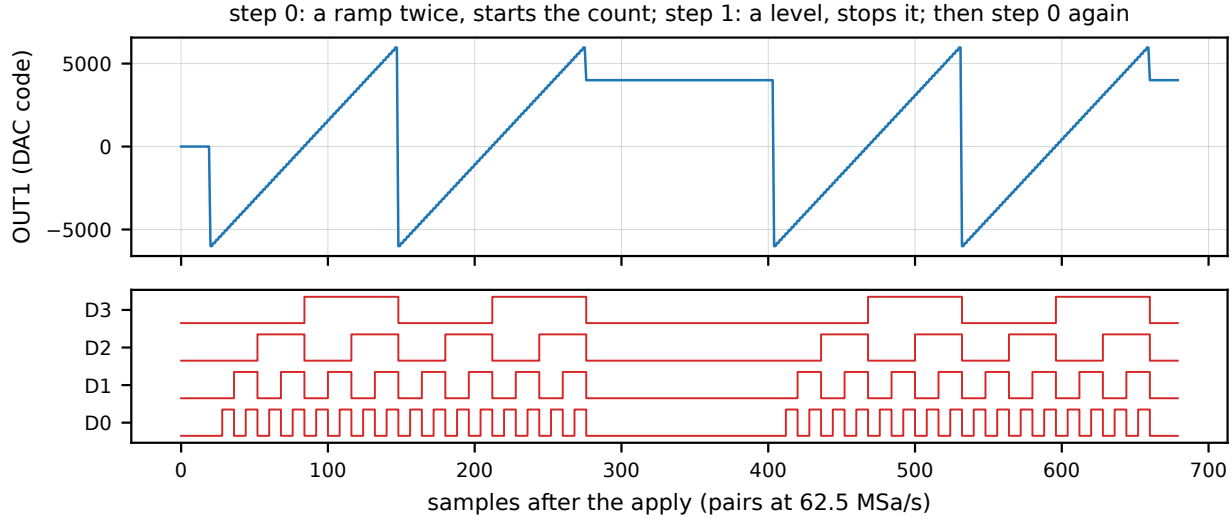


Figure 16: Computed with the model: the sequence above. Step 0 (a ramp, twice) starts a binary count of 8 samples per word on its first point; step 1 (a level) stops it on its first point (the words return to the idle word 0); then step 0 starts it again.

16.6 Timing and alignment

The words change on the 8 ns grid. With `delay` 0 a word changes with the analog sample it belongs to (inside the FPGA, to the sample); `delay` moves the outputs in 8 ns steps from about −140 ns to +3.9 µs. At the connectors the board calibration's pins delay is corrected as for the markers (section 17.5): on the board used here the words change 3.3 ns after the analog edge. Measured on the board: D0 wired to TRIG B started a burst at the same time, to the pins' 1 ns resolution, as MARKER 1 wired to TRIG A (0.0 ns), and a delay step of 8 ns moved it by one sample. D1–D7 were not measured at the pins; they are the same logic (and the simulation compares all eight pins with the model).

Programs and applies. The driver writes new programs into the half of the program table that is not playing and switches at the apply, so a running program is never changed under it. Patterns stay in the pattern memory at the place they were first put; if new patterns do not fit next to them, the memory is laid out anew, which may disturb a pattern playing at that moment.

Status. `awg.hardware()["digital"]` reports the engine's `state` (`idle`, `running`, `holding`), the index of its program (programs are numbered by sorted name) and the number of `starts`. The self-test counted 1667 starts in 0.213 s from sequence steps, each one followed by a trigger through D0.

Electrical. 3.3 V LVCMOS, 8 mA drive, slow slew rate (the same as the markers); not 5 V tolerant. The rise time at the pins was not measured; drive short lines into high-impedance inputs, and terminate long ones at the source with a series resistor of about 33 Ω if reflections matter.

17 Calibration

All voltages in awg are volts at the SMA connectors. The conversion from volts to DAC codes (and from ADC codes to volts for the inputs) uses the **board calibration**: one set of constants per board, stored on the board and shared by every design of the repository (`common/CALIBRATION.md`). Without a calibration the nominal constants are used, about 14 % off in amplitude on the board used here.

17.1 Input ranges

Each input has a jumper: LV (±1 V nominal, ±1.13 V on this board) or HV (±20 V nominal). Tell awg which one is set, so that input volts (thresholds, AM and FM by an input, captures) are right:

```
awg.set_input_range("in2", "hv")              # IN2's jumper is on HV
print(awg.ranges())
```

TCP: `RANGE IN2 HV`

The HV gain is nominal unless calibrated with a voltmeter on HV.

17.2 DC calibration

awg can measure the calibration itself, in loopback (OUT1 → IN1, OUT2 → IN2 with the 50 Ω terminators): it steps both outputs through 9 levels, reads the inputs and fits gain and offset of each output and input. A voltmeter reading of the reference level makes the gains absolute:

```
report = awg.calibrate()                    # loopback, nominal input gain
print(report)
```

TCP: `CAL LOOPBACK`
TCP: `CAL HOLD`

(hold the reference level for a voltmeter) The notebook panel's **Calibration** tab runs the same steps with a form for the voltmeter readings, and shows the stored constants. `CAL?` prints them; `CAL RESET` returns to nominal values. The calibration drives the outputs (to about ±0.8 V): disconnect sensitive equipment first.

17.3 Output response and droop

Sine amplitudes are corrected for the DAC's $\mathrm{sinc}(f/f_s)$ droop and, if the calibration holds a measured output response (`set_output_response`, measured for example with synth's network tools), for that response too: a set 0.5 V comes out as 0.5 V at every frequency of the table. `droop_correction` False switches both off (the amplitude words are then the plain DAC scale). Other shapes are not corrected: their harmonics would need a filter, not a factor.

17.4 Skew between the outputs

The calibration's `delay_s` of OUT2 relative to OUT1 (on this board OUT2 is 0.17 ns early, measured with synth's cable-swap method) corrects the phases of sines and carriers: two outputs set to the same phase are in phase at the connectors.

17.5 Delay of the digital pins

The markers and the digital outputs are aligned with the analog samples inside the FPGA, but at the connectors the analog path (DAC, filter, amplifier) is slower than the pins. The calibration's `pins_delay_s` is the time a pin changes after OUT1 shows the sample it is aligned with; awg subtracts it, in whole samples, from the delays you set. On the board used here the pins lead OUT1 by 28.6 ns (oscilloscope, 2026-10-24): stored as −28.6 ns, the markers and digital outputs at delay 0 land 3.3 ns after the analog edge.

To measure it on another board (`common/CALIBRATION.md`): play a step on OUT1 with a marker on the same sample, for example a step-filtered arbitrary square with marker bits on its first high points; trigger an oscilloscope on OUT1 (into 50 Ω) and read the time from OUT1's 0 V crossing to the marker's 1.65 V crossing; subtract the probe's delay relative to the coax (measure it by probing the OUT1 connector with both). Store it:

```
awg.set_pins_delay(-28.6e-9)                # the pins change 28.6 ns before OUT1
```

TCP: `CAL PINS -2.86e-8`

18 Measuring with the inputs

The inputs are sampled on the same clock as the outputs. `awg.capture(delay)` records 8192 samples (65.5 µs) of IN1 and IN2, starting `delay` seconds after the call (after a scheduled apply has taken effect), and returns codes, volts (through the calibration) and the sample index of the first sample:

```
data = awg.capture()
x = data["volts"]["in1"]
print(data["start"], max(x) - min(x))
```

TCP: `CAPTURE?`

A capture is not triggered by events; to look at a burst or a sequence, make it repeat (a timer, a continuous sweep, a looping sequence) so any capture contains it, as the self-test and this manual's measured figures do. With the loopback cables the board checks itself: amplitudes, frequencies, phases (lock-in over whole cycles), edge positions to a tenth of a sample, noise spectra, modulation by demodulation. The capture's start index is on the t_0 grid of the outputs, so positions are absolute (figure 4).

Loopback levels

The board's input path has its own response: on the board used here a DC level reads 3.6 % low on IN1 and a sine at 1–2 MHz about 2.5–3 % high, so the self-test compares like with like (DC with DC, sines with sines) or uses ratios. The outputs themselves are calibrated at DC.

19 Presets

name	outputs
`off`	everything off, 0 V (the default)
`sine`	OUT1: 1 MHz sine, 0.5 V
`square`	OUT1: 1 MHz square, 0.5 V, 50 %, 10 ns edges
`pulse`	OUT1: 100 kHz, 200 ns pulses from 0 V to 0.5 V
`iq`	OUT1 and OUT2: 1 MHz, 0.5 V, OUT2 90° behind
`noise`	OUT1: Gaussian noise, 0.1 V rms, 10 MHz
`am`	OUT1: 1 MHz carrier, 0.4 V, 80 % AM by a 10 kHz sine
`sweep`	OUT1: 1 kHz to 1 MHz linear sweep in 10 ms, repeating
`burst`	OUT1: 5-cycle 1 MHz bursts every 1 ms (timer 1)

```
awg.load_preset("burst")
awg.set_output(1, frequency=2e6)
awg.save_preset("my_burst")                    # the present configuration, by name
print(awg.presets())
```

TCP: `PRESET burst`

TCP: `PRESET SAVE my_burst`

TCP: `PRESETS?`

A preset is a whole configuration: loading one replaces everything except protected limits.

20 The notebook panel

In the notebook `awg.ipynb`, `make_gui(awg)` shows three tabs:

Outputs

For each output: the function, frequency, amplitude, offset, phase and the shape settings; the noise and its bandwidth; burst mode, cycles, trigger source and delay; the sweep (mode, linear or log, start, stop, time); AM, angle modulation and their source; the limits; the arbitrary waveform (on, filter, amplitude); and **Apply OUT**n, which sends that output's edits together. Below: the **arbitrary-waveform sequencer** (sequence, rate, run mode, trigger, **Apply sequencer**); the **digital outputs** (run mode, program, idle word, the pins driven, **Apply digital**); **Trigger OUT1/OUT2/both**, **Zero time**, **SAFE (all off)** and **Refresh from hardware**. A line shows t_0, the sample counter, each output's state and counters, and the sequencer's and pattern engine's states.

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

Calibration
: The board calibration of all designs: input ranges, the loopback calibration with optional voltmeter readings, the stored constants.

Settings not in the panel (generator 2's shapes, list and profile sweeps, a sweep's hold, return and marker, markers, timers, input thresholds, pattern programs, sequences) are set from code in the same notebook; the panel shows them after **Refresh from hardware**. The panel and any script may be used at the same time.

21 Using awg from experiment software

21.1 From Python

On the board the driver, on a PC the client: the same methods (appendix A.2). The whole configuration is a JSON-compatible dictionary (appendix A.1): `awg.config()` returns it with the applied values, `awg.apply(patch)` merges a patch and applies everything on one sample, `awg.apply(config, replace=True)` replaces it. A typical experiment script sets the configuration once and then changes a few values:

```
awg.apply({"outputs": {"1": {"function": "sine", "frequency": 10e6, "amplitude": 0.2}},
           "markers": {"1": {"source": "sync", "channel": 1}}}, replace=True)   # the rest: defaults
for f in (9.9e6, 10.0e6, 10.1e6):
    applied = awg.set_output(1, frequency=f)          # coherent: lands 10 ms later, phase from t0
    print(applied["frequency"])
```

Each coherent change returns when it is scheduled (about 10 ms before it takes effect, the `apply_delay`); a measurement that must see it should wait that long. A configuration change through the service takes about 90 ms from a PC.

Several programs at once. Every change is one transaction; the configuration has a **revision** that changes with every change. A program that edits a draft (as the panel does) can pass `expected_revision` to `apply` and gets a conflict error instead of overwriting someone else's change. `awg.snapshot()` returns the configuration, its revision, the description and the status in one consistent read.

21.2 Status and counters

`awg.hardware()` (TCP `HW?`) reads the hardware's state in one call:

`now`, `last_apply`, `apply`
: the sample counter, the last apply's sample, whether an apply is pending, and the count of applies (and of late ones: scheduled less than 21 samples ahead).

`levels`
: the present level of each pin and threshold (`trig_a` ... `in2`).

`outputs`
: per output: `armed`, `running`, `sweeping`; the counts of accepted `triggers`, `missed` ones and `clipped_samples`; the `last_start` time in samples (to 1/256); whether the limits are `protected`.

`arb` the sequencer's `state`, `step`, `underflow`, a read outside the buffer (`window_error`), the FIFO's fill and its lowest fill.

`digital`
: the pattern engine's `state`, `program_index`, `starts`.

muted, clock_locked
: SAFE's mute; the trigger pins' clock.

`awg.clear_counts()` (TCP `CLEAR`) sets the counts to 0. Start times are on the t_0 grid of the sample counter: the difference of two outputs' `last_start` is their relative timing to 1/256 sample, which the self-test uses to time the trigger pins.

21.3 From other languages

Anything that opens a TCP connection to port 5025 and sends lines:

```
printf 'FUNC 1 sine 1e6 0.3\nOUT 1?\n' | nc -q1 192.168.0.149 5025
```

In MATLAB, `t = tcpclient("192.168.0.149", 5025); writeline(t, "FUNC 1 sine 1e6 0.3"); readline(t)`; in LabVIEW, TCP Open/Write/Read with a line feed. Every command answers one line, `OK ...` or `ERR ...`; JSON values (`OUT 1 {...}`, `CONFIG {...}`) carry any setting of this manual. Binary data (segments, patterns) need the framing of `common/README.md` ("Binary data"); the Python client does it for you.

21.4 The command line

```
PYTHONPATH=common/python:awg/control python3 -m rpfpga.cli --host 192.168.0.149 \
    --func 1=square,1e6,0.3 --offset1 0.1 --noise2 0.05,10e6
python3 -m rpfpga.cli --host 192.168.0.149 --segment chirp chirp.npy --play chirp,10e6 --arb 1=0.3,
    ↪ smooth
python3 -m rpfpga.cli --host 192.168.0.149 --pattern clk clk.txt \
    --digital '{"run": "continuous", "program": "c", "enable": 1, "programs": {"c": {"pattern": "clk
    ↪ "}}}'
```

Without options it prints the present state (`-json` for all of it). All options are in appendix A.4.

21.5 After a reboot, and the service

The control service (`rp-overlay.service`) keeps the last applied configuration on the SD card and restores it when the board starts (segments, sequences, tables and patterns are files there too). Restoring the arbitrary waveforms reloads their segments into the memory buffer, so the first apply after a start takes a few seconds with large segments.

22 Application notes

The examples combine several parts of awg. They are complete: each runs as written (on the board, or from a PC with the client).

22.1 A pulse experiment with gates for other instruments

An experiment fires a laser at an external trigger, then needs a 2 μs RF pulse 1.5 μs later, a gate for a detector around it, and a camera trigger. One trigger on TRIG A does it all: OUT1 plays the RF burst with the trigger's 1 ns timing, marker 1 gates the detector (the burst's `run` state), and the digital outputs give the camera trigger and a frame counter, started by the same burst.

```
awg.save_pattern("camera", [0b01] * 4 + [0b00] * 4)           # D0: 4 words high, then low
awg.apply({
    "outputs": {"1": {"function": "sine", "frequency": 40e6, "amplitude": 0.5,
                      "burst": {"mode": "triggered", "cycles": 80},           # 2 us at 40 MHz
                      "trigger": {"source": "trig_a", "edge": "rising", "delay": 1.5e-6}}},
    "markers": {"1": {"source": "run", "channel": 1, "invert": False, "delay": -40e-9, "width": 1e
    ↪ -6}},
    "digital": {"run": "out1", "program": "cam", "enable": 0x01,
                "programs": {"cam": {"mode": "memory", "pattern": "camera", "rate": 125e6 / 25,
                                     "repeats": 1, "end": "idle"}}}})
```

The RF starts 1.5 μs plus the fixed 377.9 ns after the laser's edge (figure 9); the detector gate opens 40 ns earlier, and D0 gives the camera a 0.8 μs pulse aligned with the burst's start. A trigger during the burst is counted as missed.

22.2 Adiabatic passage: a shaped chirp

A frequency chirp with a smooth amplitude envelope, played once per trigger with the trigger's timing: the arbitrary waveform holds the waveform itself, so the envelope and the frequency law can be anything.

```
import math
n = 20000                                              # 160 us at 125 MSa/s
f0, f1 = 5e6, 15e6
pts = []
for i in range(n):
    t = i / 125e6
    phase = 2 * math.pi * (f0 * t + (f1 - f0) * t * t / (2 * n / 125e6))
    pts.append(math.sin(math.pi * i / n) ** 2 * math.sin(phase))
awg.save_segment("arp", pts, markers=[1] * 8 + [0] * (n - 8))
awg.save_sequence("arp_once", [{"segment": "arp", "wait": True, "wait_level": 0.0, "next": 0}])
awg.play(sequence="arp_once", rate=125e6, run="triggered", trigger={"source": "trig_b"})
awg.set_output(1, arb={"on": True, "amplitude": 0.4, "filter": "smooth"})
```

Each trigger on TRIG B plays the chirp once (the step waits for the trigger, plays, and goes back to waiting), starting $T = \tau + 160$ ns after the trigger to 31 ps; marker bit 1 marks its first 8 points. At 125 MSa/s use the step filter if you want no smoothing delay; the smooth filter is the better choice at lower rates.

22.3 I/Q pulses for a modulator or a qubit drive

Gaussian pulses with a derivative correction on Q (as used against leakage in qubit control), upconverted to a carrier inside the FPGA, with a marker at each pulse:

```
import math
n, sigma, beta = 400, 40.0, 0.3
gauss = [math.exp(-((k - n / 2) / sigma) ** 2 / 2) for k in range(n)]
deriv = [-(k - n / 2) / sigma ** 2 * g for k, g in enumerate(gauss)]
awg.save_segment("pulse_i", gauss + [0.0] * 600, values2=[beta * d for d in deriv] + [0.0] * 600,
                 markers=[1] * 4 + [0] * 996)
awg.play(sequence="pulse_i", rate=125e6)
awg.set_output(1, arb={"on": True, "amplitude": 0.3, "filter": "step",
                       "upconvert": {"on": True, "frequency": 20e6, "phase": 0.0}})
awg.set_output(2, arb={"on": True, "amplitude": 0.0})
```

The pulses repeat every 8 μs (1000 pairs). Change `phase` for the pulse's axis, and `q_gain` and `q_phase` to trim the I/Q balance. For an external modulator, leave `upconvert` off and give OUT2 the same amplitude: OUT1 plays I, OUT2 plays Q.

22.4 Testing a receiver: data, noise and a PRBS reference

A 5 MHz carrier, BPSK-modulated by a PRBS of 1 Mb/s from generator 2, with Gaussian noise of a chosen signal-to-noise ratio, and the same bit sequence on the digital outputs as a reference for the bit-error count:

```
awg.set_output(1, function="sine", frequency=5e6, amplitude=0.2,
               aux={"function": "prbs", "frequency": 1e6, "order": 15, "amplitude": 0.0},
               angle={"mode": "bpsk", "key": "aux", "bpsk_phase": 180.0},
               noise={"rms": 0.02, "bandwidth": 20e6})
```

Generator 2 here keys BPSK by its first half cycle at 1 MHz, a regular pattern; for a data-like key, drive the key from a pin (`key "trig_b"`) with a PRBS from the digital outputs, wired D0 to TRIG B (the self-test's wiring), or from your data source:

```
awg.set_output(1, function="sine", frequency=5e6, amplitude=0.2,
               angle={"mode": "bpsk", "key": "trig_b", "bpsk_phase": 180.0},
               noise={"rms": 0.02, "bandwidth": 20e6})
awg.set_digital(run="continuous", program="data", enable=0x03,
                programs={"data": {"mode": "prbs", "order": 15, "rate": 125e6 / 125}})   # 1 Mb/s on
    ↪ D0
```

D0 keys the carrier through TRIG B; D1 carries the same sequence one bit later, a reference for the receiver's recovered data. The signal-to-noise ratio of the example is $20\log_{10}((0.2/\sqrt{2})/0.02) = 17$ dB in 20 MHz.

22.5 Event-driven sequences

A sequence that waits in a holding pattern until something happens in the experiment, then jumps without a gap. The event can be a pin (EVENT 0 here), a threshold on an input (a detector's signal crossing a level), or the software:

```
import math
awg.save_segment("hold", [0.1 * math.sin(2 * math.pi * k / 64) for k in range(640)])
awg.save_segment("burst", [math.sin(2 * math.pi * k / 16) for k in range(1600)])
awg.save_sequence("react", [
    {"segment": "hold", "repeat": 0, "on_event": "now", "event_target": 1},
    {"segment": "burst", "repeat": 1, "next": 0, "digital": "flag"}])
awg.apply({"arb": {"sequence": "react", "rate": 125e6, "jump": {"source": "in1", "edge": "rising"}},
           "inputs": {"in1": {"high": 0.2, "low": 0.1}},
           "outputs": {"1": {"arb": {"on": True, "amplitude": 0.3, "filter": "step"}}},
           "digital": {"run": "sequencer", "enable": 0x01,
                       "programs": {"flag": {"mode": "binary", "start": 1, "length": 1, "repeats": 1,
    ↪
                                             "rate": 125e6 / 100}}}})        # D0 high for 0.8 us
```

When IN1 rises through 0.2 V the sequence leaves the holding pattern at once (at least 16 samples after the holding step's first point) and plays the burst; D0 flags the burst on its first point; then it returns to the holding pattern. The jump's latency from the threshold is fixed (the event's 8 samples plus the output pipeline).

22.6 Clocks and frames for other instruments

A binary count makes divided clocks on all eight outputs, phase-locked to the analog outputs:

```
awg.set_digital(run="continuous", program="clk", enable=0xFF,
                programs={"clk": {"mode": "binary", "rate": 125e6 / 4}})    # D0 15.6 MHz ... D7 122
    ↪ kHz
```

Dk is a square wave of $31.25\,\mathrm{MHz}/2^{k+1}$, all edges on the sample grid and aligned with the analog outputs. A pattern with one word in N high makes a frame or sync pulse; an output's start (run `out1`) restarts the clocks with each burst, so the clock phase is fixed relative to the burst.

22.7 What awg cannot do

- **Frequencies above about 50 MHz** cleanly: the DAC holds each sample for 8 ns, so images at $125\,\mathrm{MHz} - f$ grow as f approaches 62.5 MHz.
- **Independent arbitrary sequences on the two outputs**: one sequencer plays both (pairs); OUT1 and OUT2 always play the same step. Give each output its own segment content in the pairs, and use the generators for independent periodic signals.
- **Bursts or sweeps on OUT1 while the sequencer plays** (it uses OUT1's trigger hardware); OUT2 keeps all of them.
- **Sweeps faster than 0.95 kHz per sample** (linear) or an octave in 16 μs (logarithmic); use an arbitrary waveform for faster chirps.
- **Edges shorter than one sample** (6.4 ns 10–90 %), and timing finer than the digital outputs' and markers' 8 ns grid.
- **Triggered captures**: the inputs record 8192 samples at a time, started by software; make the measured signal repeat.
- **More than 8 pattern programs at once**, more than 32768 pattern words in use, more than 512 steps in a sequence or 3 event targets besides step 0.

23 Specifications

"Measured" values were measured on the board used here (`docs/VERIFICATION.md`), "computed" ones with the design's bit-exact model; board-wide figures (input ranges, bandwidth, noise) are in the root `README.md`.

quantity	value
outputs	2, 125 MSa/s, 14 bits, one time base; changes of both outputs on one sample
frequency	0 to 62.5 MHz, steps of 0.44 μHz; keep below about 50 MHz (images)
phase	relative to t_0, steps of 2^{-32} cycle
amplitude, offset	amplitudes 24-bit words (about 0.07 μV steps), offset steps of 2^{-16} DAC step; ± 1.13 V into 50 Ω on this board
sine	spurs below -100 dBc before the DAC's rounding (computed); droop corrected
edges	from one sample (6.4 ns 10–90 %), sub-sample positions; measured 1.085 samples per sample of phase (figure 4)
shapes	sine, square, pulse, ramp, triangle, PRBS 7–31, tables of 4096 points, DC
noise	Gaussian to 7.1 σ; -3 dB at the set bandwidth (1 Hz to 47.7 MHz); measured at 10 MHz: -3 dB at 10.10 MHz, kurtosis 3.003, 0.272 % beyond 3 σ

quantity	value
modulation	AM/DSB with FM, PM, PWM, FSK or BPSK; sources generator 2, IN1, IN2, the other output; measured: FM 499.5 kHz for 500, PM 89.999° for 90, BPSK 179.2° for 180
sweeps	linear, log, list, profile; up to 512 pieces; 0.95 kHz/sample linear, 1/2048 octave/sample log; measured rates 0.99985 and 0.99999 of the set
bursts	1 to $2^{32} - 1$ cycles or endless, start and stop phase, gates, idle start/last/DC
triggers	TRIG A/B 1 ns timestamps, EVENT 0–3 and thresholds 8 ns, software, timers, the other output; delay 0 to days in 31 ps steps
trigger to output	377.9 ns at the connectors with delay 0 (measured)
trigger jitter	1 ns quantization of TRIG A/B (computed); against an external trigger not measured
markers	2: sync, run, sweep, start pulse, point markers; delay −160 ns to +3.9 µs in 8 ns steps; 3.3 ns after the analog edge at delay 0 (measured)
limits	per output, every sample, lockable until the FPGA is reloaded
arbitrary waveforms	16 M pairs (64 MB), segments of 64 to $2^{25} - 1$ pairs, rates to 125 MSa/s in 2^{-32} steps, step/linear/smooth filters, 512-step sequences, gapless jumps; both outputs streamed at 125 MSa/s without a gap (measured)
I/Q	baseband, or upconverted to 0–50 MHz; measured image −91.6 dBc and carrier −91.5 dBc (at the measurement's floor)
digital outputs	8, 3.3 V, enabled per pin (else high impedance); 8 programs: patterns of 32768 words, binary and Gray counts, PRBS 7–31; 125 MHz/N; aligned with the analog outputs; D0 within 1 ns of the markers (measured)
uploads	segments 8–12 MB/s over the network; a configuration change about 90 ms (measured)
resources	13,869 LUTs (79 %), 58 of 60 block-RAM tiles, 48 DSP48; timing met by 0.003 ns (build 0x97cda2f3)

24 Troubleshooting

"ERR ... can leave the output range"
The worst case of offset, amplitudes, AM and 4 σ of noise exceeds ±1.13 V. Lower something, or set `allow_clip` (the peaks then saturate) or limits.

"an edge of ... is shorter than one sample"
At this frequency the edge would be shorter than 8 ns; set `rise`/`fall` 0 (the shortest) or lower the frequency.

"the duty cycle leaves no room for the edges"
Duty or width too close to 0 or 100 % for the edges at this frequency.

"sweep is too fast"
Longer `time` or `return`, or an arbitrary waveform.

The output does nothing in burst mode
No trigger, or triggers missed: `awg.hardware()["outputs"]["1"]` shows `triggers` and `missed`. Check the source, edge and wiring; for gates, the polarity. Software triggers count only with source `software`.

An arbitrary waveform does not start
The sequencer must have a sequence (`awg.play(sequence=...)`) before an output's `arb` can be on; `hardware()["arb"]` shows its state: `waiting` means a trigger is due.

"underflow"
A step entered by a jump or start read its points from memory too late (the sequencer's `underflow` in `hardware()`): avoid very short steps right after a jump target at the full rate, or lower the rate.

The digital outputs show nothing
: They are high impedance until `enable` has their bits; then check `run` and the `digital` part of `hardware()`.

Timing differs from the scope
: The markers and digital outputs are corrected by the board's `pins_delay_s`; without it they lead the analog outputs by about 29 ns (section 17.5).

After switching designs
: Loading a design reloads the FPGA: handles to the old design are invalid (`Stale`); connect again.

"Permission denied (publickey)" from `make`
: Your shell has no working SSH agent for the board's key; start one, or set `SSH_AUTH_SOCK` to the desktop's agent.

25 Quick reference

to …	do
make a sine	`awg.set_output(1, function="sine", frequency=1e6, amplitude=0.3)`
lock OUT2 90° behind	`awg.set_output(2, function="sine", frequency=1e6, amplitude=0.3, phase=-90)`
add noise	`awg.set_output(1, noise={"rms": 0.05, "bandwidth": 10e6})`
modulate	`am={"mode": "am", "source": "aux", "depth": 0.5}, aux={"function": "sine", "frequency": 1e4}`
sweep	`sweep={"mode": "continuous", "type": "log", "start": 1e3, "stop": 1e6, "time": 1e-2}`
burst on a pin	`burst={"mode": "triggered", "cycles": 5}, trigger={"source": "trig_a"}`
trigger by software	`awg.trigger(1)`
mark the starts	`awg.apply({"markers": {"1": {"source": "start", "channel": 1}}})`
play points	`awg.save_segment(name, values)`, `awg.play(sequence=name, rate=...)`, `awg.set_output(1, arb={"on": True, "amplitude": 0.3})`
digital pattern	`awg.save_pattern(name, words)`, `awg.set_digital(run="continuous", program="p", enable=0xFF, programs={"p": {"pattern": name}})`
protect a device	`limits={"low": -0.2, "high": 0.2}`, then `awg.protect(1)`
everything off	`awg.safe()`

A Command reference

A.1 The configuration

`awg.config()` returns, and `awg.apply(patch)` merges:

```
{"outputs": {"1": OUTPUT, "2": OUTPUT},
 "inputs": {"in1": {"high": 0.1, "low": -0.1}, "in2": {...}},     # thresholds (V), hysteresis
 "timers": {"1": {"period": 1e-3, "on": False}, "2": {...}},      # aligned to t0
 "markers": {"1": {"source": "low", "channel": 1, "invert": False, "delay": 0.0, "width": 1e-6}, "2":
    ↪ {...}},
 "arb": SEQUENCER,                # section "The sequencer"
 "digital": DIGITAL,              # section "Digital outputs"
 "phase_mode": "coherent",        # or "continuous"
 "t0": 0,                         # zero time, a sample index
 "droop_correction": True,
 "allow_clip": False,
 "apply_delay": 0.01}             # coherent changes this long after the call (1 ms ... 1 s)
```

`OUTPUT` has the keys of the tables in sections 6 (waveform), 6.6 (`aux`), 7 (`noise`), 8 (`am`, `angle`), 9 (`sweep`), 10 (`burst`), 11 (`trigger`), 13 (`limits`, `dither`) and 14.3 (`arb`). The full list with defaults

is `awg.default_config()` and `docs/PROTOCOL.md`.

A.2 Python methods

method	meaning
`config()`, `apply(patch, replace=False)`	the configuration; apply a patch (or replace)
`describe()`, `status()`, `snapshot()`	one line per output; status with headroom; all at once
`set_output(n, **settings)`	patch output n; returns its applied settings
`function(n, name, Hz, V, deg)` (client)	the function shortcut
`trigger(n=None)`	software trigger of output n, or both
`zero_time(delay=None)`	t_0 := the next apply sample; re-apply
`protect(n)`	lock output n's limits until the FPGA is reloaded
`safe()`	mute, then everything off, digital outputs high impedance
`presets()`, `load_preset(name)`, `save_preset(name)`, `delete_preset(name)`	presets
`table_names()`, `save_table(name, values)`	tables
`hardware()`, `clear_counts()`	counters, states, levels, last starts; zero the counts
`capture(delay=0.001)`	8192 samples of IN1 and IN2
`save_segment(name, values, markers, values2, markers2)`	a segment of pairs
`segment_names()`, `segment_values(name, start, count)`, `delete_segment(name)`	segments
`load_segment_file(name, path)` (driver)	a segment from a `.npy` or CSV file on the board
`save_sequence(name, steps)`, `sequence(name)`, `sequence_names()`, `delete_sequence(name)`	sequences
`play(**settings)`	the sequencer's settings
`set_arb(n, **settings)` (client)	output n's `arb`
`save_pattern(name, values)`, `pattern_names()`, `pattern_values(name)`, `delete_pattern(name)`	patterns
`set_digital(**settings)`	the digital outputs
`calibration()`, `calibrate(dmm=None)`, `reset_calibration()`	the board calibration
`set_output_response(n, points)`, `set_output_delay(n, s)`, `set_pins_delay(s)`	its AC fields

A.3 TCP commands

Port 5025, one command per line, replies `OK <value>` or `ERR <message>`. awg's own:

command	reply
`OUT <o> <json>`, `OUT <o>?`	patch output o; its applied settings
`FUNC <o> <function> [<Hz> [<V> [<deg>]]]`	function, frequency, amplitude, phase
`FREQ`, `VOLT`, `OFFS`, `PHAS <o> <value>`	one setting
`NOISE <o> <V rms> [<Hz>]`	noise
`TRIG [<o>]`, `ZERO [<s>]`, `PROTECT <o>`	trigger; t_0; lock the limits
`TABLES?`, `TABLE <name> <json>`, `TABLEZ <name> <base64>`	tables
`SEGMENT <name>` + binary, `SEGMENT? <name> [<start> <count>]`, `SEGMENTS?`, `SEGDEL <name>`	segments
`SEQUENCE <name> <json>`, `SEQUENCE? <name>`, `SEQUENCES?`, `SEQDEL <name>`	sequences
`PLAY <json>`, `PLAY?`, `ARB <o> <json>`	the sequencer; an output's `arb`
`PATTERN <name>` + binary, `PATTERN? <name>`, `PATTERNS?`, `PATDEL <name>`	patterns
`DIGITAL <json>`, `DIGITAL?`	the digital outputs
`HW?`, `CLEAR`, `CAPTURE? [<s>]`	hardware state; counts to 0; a capture

The common commands of every design: `*IDN?`, `DESIGN?`, `LOAD <design>`, `CONFIG?`, `CONFIG <json>`, `CONFIG REPLACE <json>`, `DESCRIBE?`, `STATUS?`, `SNAPSHOT?`, `PRESETS?`, `PRESET <name>`, `PRESET SAVE <name>`, `SAFE`, `RANGE?`, `CAL?`, `CAL LOOPBACK`, `CAL PINS <s>` and the others of `common/README.md`.

A.4 Command-line options

`python3 -m rpfpga.cli [-host H] [common options] [awg options]`; awg's:

option	meaning
`-func O=F[,HZ[,V[,DEG]]]`	output O's function, frequency, amplitude, phase (repeatable)
`-offset1 V`, `-offset2 V`	offsets
`-noise1 RMS[,HZ]`, `-noise2 ...`	noise
`-play NAME[,RATE]`	play a sequence or segment
`-arb O=V[,FILTER]`	output O plays its half of the pairs (`O=off`: stop)
`-segment NAME FILE`	a segment from a `.npy` or CSV file on this machine
`-pattern NAME FILE`	a pattern from a text file (decimal, `0x` or `0b` words)
`-digital JSON`	patch the digital outputs
`-trigger [O]`, `-zero`	software trigger; $t_0 :=$ the next apply

Common options: `-preset`, `-safe`, `-reset`, `-load`, `-range`, `-calibration`, `-json`.