spin: user's manual

A pulsed RF transceiver for teaching pulsed NMR, NQR and ESR and for qubit readout, on the Red Pitaya STEMlab 125-14

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

Red Pitaya `spin` design, version 1.1 (build 0x33bbab94)
November 12, 2026

Abstract

`spin` turns the Red Pitaya into a pulsed RF transceiver for two-level systems. A pulse program on the board drives two transmitters up to about 50 MHz (phase-coherent frequencies, phases and amplitudes in volts, envelope shapes and chirps, on two outputs or an I/Q pair), receives with the same oscillators, averages the echoes on the board, measures one I/Q value per shot and decides on the board, on a pin or in the program itself. A built-in emulator, a spin ensemble or a qubit, answers the transmitters, so every experiment runs without a magnet. Templates provide the classic experiments (free induction decays, echoes, CPMG and spin-locked trains, relaxation, nutation, field sweeps; Rabi, Ramsey, single-shot readout, active reset) with their fits. This manual is for the lab: the first run, every part of the instrument, the experiments with the board's own recordings, real setups, and the reference. Figures marked "measured" were recorded on the board used here (build 0xc9ca505b, whose RTL differs from 1.1's only in the gates at a guard trip; 1.1 moved the averager into common/ unchanged), on the emulator or in loopback. The physics behind the experiments, with exercises, is in the theory manual (`theory_manual.pdf`). Behaviour with real probes, bridges, spins and qubits has not been measured: no such hardware was at hand.

Contents

1 Introduction **3**
1.1 What it is for . . . 3
1.2 Connectors . . . 3
1.3 This manual . . . 4

2 Getting started **4**
2.1 The board and the service . . . 4
2.2 A first experiment, without wiring . . . 4
2.3 The control panel . . . 5
2.4 A loopback check . . . 5

3 The pulse program **6**
3.1 Instructions . . . 6
3.2 Timing . . . 7
3.3 Waits and triggers . . . 7
3.4 Shots and markers . . . 7

4 Transmitters and outputs **8**
4.1 Frequencies, phases and amplitudes . . . 8
4.2 Shapes and chirps . . . 8
4.3 Two outputs or an I/Q pair . . . 8
4.4 Protecting an amplifier, a probe and a sample . . . 8
4.5 Digital outputs . . . 9
4.6 Slow outputs . . . 10

5 Receivers, windows and averaging **10**
5.1 Receivers . . . 10
5.2 Windows . . . 10
5.3 The receiver's phase . . . 10
5.4 The averager . . . 11
5.5 Weighted results . . . 11
5.6 The raw capture . . . 12

6 Decisions **12**
6.1 A window's state . . . 12
6.2 Branches . . . 12
6.3 Latencies . . . 13
6.4 State outputs . . . 13
6.5 Active reset and repeat-until-success . . . 13

7 The emulator **14**
7.1 What it computes . . . 15
7.2 Using it . . . 15

8 Experiments **16**
8.1 Templates . . . 16
8.2 The FID and its spectrum . . . 17
8.3 The Hahn echo . . . 17
8.4 T_2 and echo trains . . . 17
8.5 T_1: inversion and saturation recovery . . . 18
8.6 Nutation . . . 19
8.7 An echo-detected field sweep . . . 19
8.8 Qubit: Rabi, Ramsey and T_1 . . . 20
8.9 Single-shot readout . . . 21

9 Real setups **21**
9.1 Pulsed NMR with a probe . . . 21
9.2 NQR: a frequency search . . . 22
9.3 Pulsed ESR at an intermediate frequency . . . 22
9.4 Qubit readout at an intermediate frequency . . . 23
9.5 Triggers and other instruments . . . 23

10 Reference **23**
10.1 Presets . . . 23
10.2 TCP commands . . . 23
10.3 The Python interface . . . 24
10.4 Specifications . . . 25
10.5 How it is tested . . . 25
10.6 Files . . . 25

1 Introduction

`spin` is a design for the Red Pitaya STEMlab 125-14 that turns the board into a *pulsed RF transceiver* for two-level systems: nuclear spins in a magnet, quadrupolar nuclei in zero field, electron spins behind a microwave bridge, or a qubit read out through a resonator. It does what a pulsed spectrometer's console does:

- a **pulse program** on the board (1024 instructions, four nested loops, four tables, waits for a trigger, a timer or software, and branches) times everything in 8 ns steps, from 16 ns to 34 s per instruction;
- two **transmitters** make RF pulses up to about 50 MHz: eight phase-coherent frequencies, eight phases and eight amplitudes each (in volts at the connector), envelope shapes and linear chirps, on two outputs or on an I/Q pair for an IQ mixer;
- two **receivers** demodulate the inputs with the transmitters' own oscillators, so the signal stays coherent with the pulses, filter and decimate it (8 ns to 131 µs per point), and **average** it on the board, phase-cycled, without dead time;
- each window of the receivers can give one **weighted value per shot** (a matched filter), a **state** from a threshold, on a digital output or as a **branch** of the pulse program;
- eight **digital outputs** gate power amplifiers and switches, four **slow outputs** step a field or an attenuator from the program;
- a built-in **emulator**, a spin ensemble or a qubit, answers the transmitters at the receivers, so every experiment runs with no magnet, no probe and no qubit.

Python *templates* build the classic experiments (section 8): free induction decays, Hahn echoes, CPMG and spin-locked trains, inversion and saturation recovery, nutation, echo-detected field sweeps, and for qubits Rabi, Ramsey, T_1, single-shot readout, active reset and repeat-until-success, with fits of their results.

1.1 What it is for

`spin` is meant for

- teaching pulsed NMR and NQR, with a probe, a duplexer and amplifiers (section 9);
- pulsed ESR at an intermediate frequency, with an external microwave bridge;
- reading out a qubit at an intermediate frequency, with external mixers;
- and, through the emulator, teaching all of these without the hardware: the theory manual's exercises run on it.

Its numbers were measured on the board in loopback (a cable from an output to an input) and on the emulator. **No probe, bridge, spin sample or qubit was used**: the behaviour of amplifiers, switches, probes and samples is not measured here, and the manual says so where it matters.

1.2 Connectors

Before connecting equipment

Read the equipment-safety section of the repository's README. In short: the E1 pins are 3.3 V and not 5 V tolerant (a 5 V TTL source needs a 50 Ω terminator at the board); the outputs D1–D8 are in high impedance until enabled, so **fit a pull-down on every line that unblanks a power amplifier or switches a T/R switch**, or the amplifier may open while the board boots; **never connect an amplifier's output to an input**; before connecting an amplifier, set the outputs' limits and the transmitters' duty guard (section 4.4) and lock them.

Connector	Pin	Use
SMA	OUT1, OUT2	the transmitters: TX1 and TX2, or the I/Q pair OUT1 + i OUT2 = TX1 + TX2
SMA	IN1, IN2	the receivers' inputs (or the pair IN1 + i IN2)
E1	DIO0_P	TRIG: a WAIT continues at its rising edge, time-stamped to 1 ns
E1	DIO1_P, DIO2_P	COND A, COND B: levels for branches
E1	DIO0_N–DIO7_N	outputs D1–D8: TTL bits, amplifier gates, markers, states
E2	AO0–AO3	slow outputs, 0 … 1.8 V, set by the program

Table 1: What `spin` uses. DIO3_P–DIO7_P and the SATA connectors are not used.

1.3 This manual

Section 2 runs a first experiment on the emulator and a loopback check. Sections 3–6 describe the instrument part by part: the pulse program, the transmitters and outputs, the receivers and averaging, the decisions. Section 7 describes the emulator, section 8 the experiments with the board's own recordings, section 9 real setups, and section 10 is the reference: presets, commands, the Python interface, pins and specifications. The theory manual explains the physics and the signal processing, with exercises. Every setting is in volts, seconds and hertz at the connectors (through the board's calibration, `common/CALIBRATION.md`); the definitions of every register and clock are in `spin/docs/THEORY.md` section 2.

2 Getting started

2.1 The board and the service

The board is set up as described in `common/SETUP.md` (or booted from the project's SD card image) and reachable on the network, here at 192.168.0.102. A control service, `rp-overlay`, runs on it: it loads the last active design at boot, restores its configuration, and answers commands on TCP port 5025. Switch the board to `spin` with

```
python3 -m rpfpga.cli --host 192.168.0.102 --load spin      # from a PC (common/python on PYTHONPATH)
    ↪
echo "LOAD spin" | nc 192.168.0.102 5025                     # or with netcat
```

Loading reprograms the FPGA with every output off: both transmitters off, D1–D8 in high impedance, the slow outputs at their idle levels, the program a STOP. The board's calibration and input-range settings are kept (they belong to the board, not to a design).

From a PC, `SpinClient` speaks to the service; on the board, the driver `Spin` has the same methods. The examples in this manual use `dev` for either, `X` for the templates and `A` for the analysis:

```
import sys; sys.path[:0] = ["common/python", "spin/control"]
from spin import SpinClient, experiments as X, analysis as A
dev = SpinClient("192.168.0.102")
```

2.2 A first experiment, without wiring

The emulator answers the transmitters inside the FPGA, so the first experiment needs no cable. The preset `fid-emulator` sets up an emulated spin ensemble at 10.02 MHz with a Gaussian line of 20 kHz, $T_1 = 1$ ms and $T_2 = 300$ µs, and a program of 90° pulses of 0.25 µs at 10 MHz followed by the free induction decay (FID), phase-cycled; the template `X.fid` builds the same experiment and `X.acquire` runs it and returns the averaged trace in volts:

```
dev.load_preset("fid-emulator")
p = X.pulses(f=10e6, amplitude=0.8, t90=0.25e-6, dwell=0.5e-6, points=256, tr=5e-3, scans=64)
r = X.acquire(dev, X.fid(p))                # APPLY, empty the averager, run, collect
print(A.fit_fid(r["t"], r["traces"][0]))    # T2*, the offset (20 kHz), the amplitude
```

Figure 5 (section 8.2) shows the result recorded on the board. The run takes 1.3 s: 64 scans of four phase-cycle steps, 5 ms apart. Nothing in it depends on the PC's timing: the pulse program runs on the board, and the averager adds every scan in the FPGA.

2.3 The control panel

In Jupyter (on the board at port 9090, or on a PC) the panel opens with

```
import spin
spin.make_gui(dev)
```

It shows the run's state with run, stop and trigger buttons; the program as a strip of instructions (pulses of TX1 and TX2, windows, TTL bits, loops and branches); the emulator's settings; an experiment panel that builds any template from its parameters, applies it, acquires it and shows the traces, the spectrum or the arrayed points with their fit; a window's live average; the I/Q points of the last RESULT records with the decision line; the digital outputs; and the presets.

2.4 A loopback check

A coax cable from OUT1 to IN1 (no terminator) checks the transmitter and the receiver against each other. The preset `pulse-loopback` sends four 1 µs pulses of 0.2 V at 10 MHz, 100 µs apart, with TX1's phase stepping through 0°, 90°, 180° and 270° and the receiver's phase following (the CYCLOPS cycle, section 5.3); it captures IN1 from the first pulse and averages window 0 (200 points of 64 ns at TX1's oscillator) over the four pulses:

```
dev.load_preset("pulse-loopback")
dev.collect(); dev.clear_average()          # earlier sums out
dev.run()
cap = dev.capture()["in1"]                  # 2048 samples of IN1 in volts, 8 ns apart
dev.swap(); dev.collect()                   # the half with the four pulses, into the host's sums
avg = dev.window_average(0)                 # 200 points [re, im] in volts at TX1's LO
```

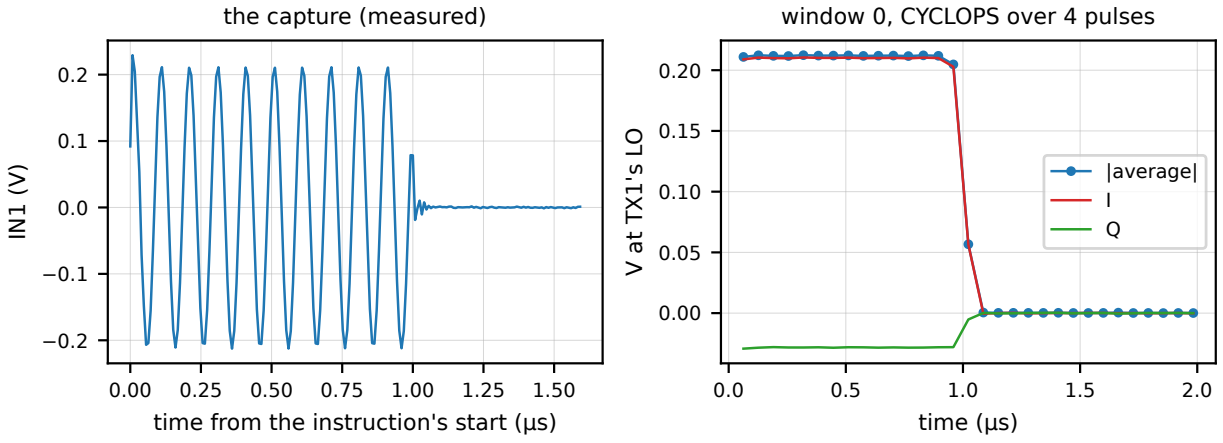


Figure 1: Loopback OUT1 → IN1 (measured, 2026-11-08). Left: the capture of IN1 from the first pulse's instruction: the 1 µs pulse at 10 MHz starts at the first sample. Right: window 0 averaged over the four phase-cycled pulses: 0.2120 V while the pulse lasts (ripple 0.2 mV rms), then zero.

Figure 1 shows what the board recorded. The capture starts at the IN connector at the instruction's start (the receivers and the capture are aligned to the connectors through the calibration's

input delay), so the pulse begins at the first sample. The window's average is the pulse's amplitude at TX1's oscillator, 0.2120 V for the set 0.2 V: the board's analog path from OUT1 to IN1 at 10 MHz reads about 6 % high (the board calibration corrects the DC gains; on this board the loop reads 3.5 % low at DC, 2.6 % high from 0.25 to 2 MHz and up to 10 % high near 20 MHz: the converters' and amplifiers' frequency response, which the calibration does not hold). The four pulses' phases cancel in the sum unless the receiver follows them, which is what the receiver phase is for.

Volts

Every amplitude, threshold and result is in volts at a connector, through the board's calibration (calibrated 2026-10-04 on the board used here). With no calibration the nominal one is used, and the numbers are then nominal, not measured. Times are referred to the connectors: a pulse, a window, a capture and a digital output that start in the same instruction start together at the connectors (within the calibration's accuracy: about 1 ns for the pins).

3 The pulse program

Everything `spin` does in time comes from its pulse program: a list of up to 1024 instructions in the FPGA, each lasting from 16 ns to 34 s in 8 ns steps, played without gaps. An instruction says what the transmitters, the digital outputs, the receivers' windows and the slow outputs do while it lasts; loop, wait and branch instructions shape the flow.

3.1 Instructions

An instruction is a dictionary; every key is optional:

Key	Meaning
`op`	`cont` (default), `loop`, `end`, `wait`, `branch`, `jump`, `stop`
`duration`	seconds (16 ns … 34.36 s, rounded to 8 ns), or `{"table": j}`
`tx1`, `tx2`	a pulse: `on`, `freq`, `phase`, `amp` (register numbers), or their values from tables (`freq_table`, `phase_table`, `amp_table`), `shape`, `chirp`, `keep`; without the key the transmitter is off and its oscillator runs on
`window`, `rx_phase`	a receiver window starts (0–7), with the receiver's phase in 90° steps
`ttl`, `shot`, `capture`	eight TTL bits, a shot boundary, the raw capture's start
`ao`	a slow output's new level, from an AO register or a table
`level`, `count`	a loop's nesting level (0–3) and count (a number, a count register or a table)
`wait`, `reref`	a WAIT's source (`trigger`, `timer`, `software`); `reref` refers the phases to the event
`target`, `if`, `not`	a JUMP's or BRANCH's target and a BRANCH's condition (section 6.2)
`offset`	added to every table index of the instruction

A LOOP is the first instruction of its body and an END the last; both play their duration, so a loop costs no extra time. Four tables of 1024 words each are indexed by a loop counter (`tables.counters` chooses which level indexes each table), so an instruction's duration, a pulse's frequency, phase or amplitude, a loop's count or a slow output's level can change on every pass: a phase cycle, an array of delays, a sweep. This program plays a 1 µs pulse at 10 MHz whose phase steps through a table, every 100 µs, with a TTL marker:

```
tx1 = {"frequencies": [10e6] + [0.0] * 7, "amplitudes": [0.2] + [0.0] * 7}
dev.apply({"tx": {"1": tx1},
           "tables": {"counters": [0, 0, 0, 0], "data": [[0, 16384, 32768, 49152], [], [], []]},
           "program": [{"op": "loop", "count": 4, "duration": 1e-6, "ttl": 1, "shot": True,
                        "tx1": {"on": True, "phase_table": 0}},
                       {"op": "end", "duration": 99e-6},
                       {"op": "stop"}]})
print(dev.config()["program"][0])           # as applied: durations rounded to 8 ns
```

Table words are raw: phases in 2^{-16} turn (16384 = 90°), frequencies in steps of 2^{16} frequency units (29.1 mHz), durations in clocks of 8 ns, slow-output levels as codes (`dev.ao_code(output,`

`volts`)). The driver checks every instruction and refuses a program it cannot play (and writes nothing); `dev.config()["program"]` returns the program as applied.

3.2 Timing

- **Run control**: `dev.run()` starts the program at instruction 0 (a running program restarts); its first instruction begins 192 ns later. `dev.stop()` ends it; `dev.trigger()` is a software trigger. `dev.state()` reports the run: running or waiting, the current instruction, the shots, an error, missed triggers and timer ticks, the averager, missed windows, lost records and the duty guard.
- **Bursts**: an engine reads each instruction (4 clocks) and a player plays them from a FIFO. Instructions of 32 ns or more play back to back without limit; about 30 instructions of 16 ns fit back to back before the FIFO runs dry. A program that outruns the engine stops with the error *underflow*; timing never shifts silently.
- **Phases** are referred to the run's start, t_0: a pulse at frequency register r starts with the phase it would have had if it had run since t_0, so returning to a frequency restores its phase (phase-coherent switching), and every scan starts with the same phases.
- **Connector time**: a pulse, a window, a capture and a digital output that start in the same instruction start together at the connectors; the receivers and the capture see the IN connector at that moment, through the calibration's delays.

3.3 Waits and triggers

A WAIT holds its outputs until an event: a rising edge on TRIG (DIO0_P), a tick of the timer (every `timer` seconds, at least 256 ns) or a software trigger; the next instruction starts 192 ns after the event. TRIG is sampled every nanosecond: the edge's time is known to 1 ns. With `reref` the phase reference t_0 moves to the edge, including its nanosecond within the 8 ns clock, so the RF that follows is locked to the external event:

```
dev.apply({"program": [{"op": "wait", "wait": "trigger", "reref": True},
                       {"duration": 1e-6, "tx1": {"on": True}, "shot": True},
                       {"duration": 50e-6}, {"op": "stop"}]})
```

Against an asynchronous trigger from another instrument the RF's phase follows each edge with 0.28 ns rms (1.0 ns peak to peak, measured with an oscilloscope); a pulse reaches the OUT connector 377–384 ns after the TRIG edge (one 8 ns clock of spread, measured). The timer makes a program repeat at a fixed rate whatever its instructions last; `state()` counts the ticks a program missed.

3.4 Shots and markers

A `shot` marks the start of an acquisition. The emulator draws its per-shot detuning there, the averager counts scans and swaps halves at shots, and RESULT records carry the shot number. The eight `ttl` bits of each instruction go to any digital output in mode `ttl` (section 4.5), with the same timing as the pulses.

How it is verified

`tests/tb_seq.sv` compares every DAC word, pin word and slow-output pin of eleven scenarios clock by clock with the bit-exact model (`control/spin/model.py`): loops with counts 0, 1 and many, tables on every loop level, waits on all three sources with re-referencing, bursts of 16 ns instructions up to an underflow, branches at their shortest durations. The board test (`scripts/test_hw.py`) checks run control, the timer, an underflow, and with a jumper or another instrument the trigger's nanosecond.

4 Transmitters and outputs

4.1 Frequencies, phases and amplitudes

Each transmitter (TX1, TX2) has a 48-bit oscillator (steps of 0.44 µHz) and eight registers each of frequency, phase and amplitude:

```
dev.apply({"tx": {"1": {"frequencies": [10e6, 10.5e6] + [0.0] * 6,   # Hz
                        "phases": [0.0, 90.0, 180.0, 270.0] + [0.0] * 4,   # degrees
                        "amplitudes": [0.2, 0.1] + [0.0] * 6}}})         # volts at the connector
```

A pulse names a register of each (`{"on": True, "freq": 1, "phase": 2, "amp": 0}`) or takes the value from a table (`freq_table`, `phase_table`, `amp_table`). The frequency registers are *phase-coherent*: each keeps the phase it would have had running since the run's start, so a pulse at a frequency always starts in step with every earlier pulse at that frequency, however the program switched in between. A frequency from a table is phase-continuous instead (it continues from the oscillator's present phase), and the receivers still demodulate with the same oscillator, so transmitter and receiver stay coherent either way.

Amplitudes are volts at the connector: the driver converts them with the board calibration's DC gain, its response at the transmitter's `carrier` (default: frequency register 0) and the DAC's sinc droop. Above DC the calibration holds no measured response on the board used here, so amplitudes at RF are nominal (the loopback reads about 6 % high at 10 MHz, section 2.4).

4.2 Shapes and chirps

A pulse's envelope is rectangular by default. Up to 31 shapes share a table of 4096 points: amplitude (0 … 1) and phase at points spread over the shape's duration, interpolated linearly at every sample, from 1/65536 to 16 points per sample:

```
import math
gauss = [math.exp(-0.5 * ((k - 32) / 10) ** 2) for k in range(65)]
dev.apply({"shapes": {"1": {"amplitude": gauss, "duration": 4e-6}},
           "tx": {"1": {"frequencies": [5e6] + [0.0] * 7, "amplitudes": [0.3] + [0.0] * 7,
                        "chirps": [2e11] + [0.0] * 6}},          # chirp register 1: Hz/s
           "program": [{"duration": 4e-6, "tx1": {"on": True, "shape": 1, "chirp": 1}},
                       {"op": "stop"}]})
```

The shape starts with the instruction; a following instruction with `keep` continues the pulse (its oscillator, chirp and envelope) instead of starting it anew. A chirp register holds a frequency slope (3.5 mHz/s … 4.9×10^{14} Hz/s); the pulse's frequency ramps from its start, for adiabatic inversion or a frequency sweep. The pulse's phase, the envelope's phase and the chirp's phase all add before the one sine lookup of each output.

4.3 Two outputs or an I/Q pair

In `"mode": "real"` OUT1 carries TX1 and OUT2 TX2: two independent channels (a drive and a readout, or two nuclei). In `"mode": "iq"` the outputs are an I/Q pair for an external IQ mixer, OUT1 + i OUT2 = TX1 + TX2: each transmitter appears as a single sideband at its frequency (positive or negative) from the mixer's LO, so two frequencies share one mixer. Each transmitter has its own Q-path gain and phase (`q_gain`, `q_phase`) to null the mixer's image at its own frequency, and each output a DC level (`analog.n.dc`) to null the LO leakage.

4.4 Protecting an amplifier, a probe and a sample

Two limits stop an experiment from damaging what is connected. **Output limits** clamp every sample of an output to a window in volts; the **duty guard** of each transmitter stops a run whose pulse is longer than `max_pulse` or whose mean duty, through a leaky bucket that allows a full pulse of `burst` seconds from an empty bucket, exceeds `max_duty`:

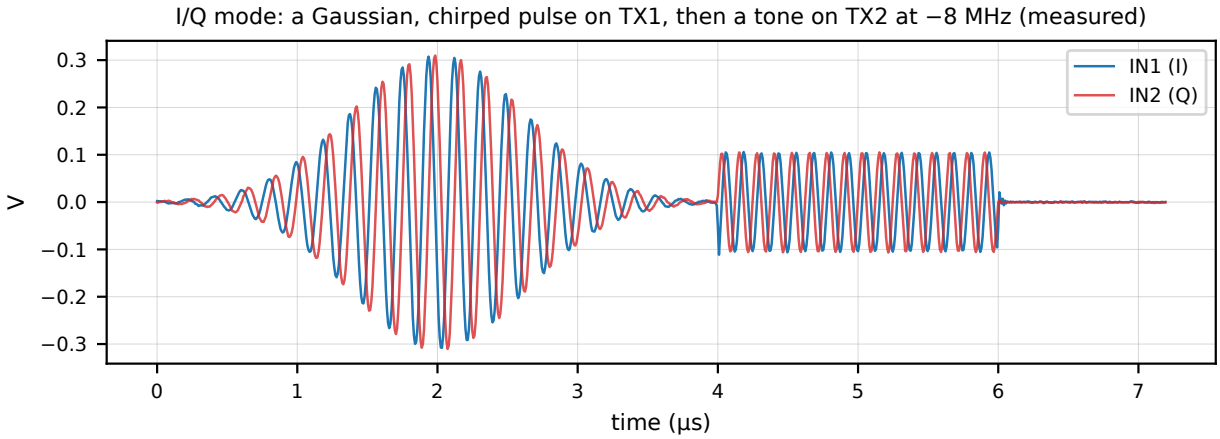


Figure 2: I/Q mode (measured, OUT1 → IN1, OUT2 → IN2): a Gaussian 4 µs pulse on TX1 at 5 MHz with a chirp of 2×10^{11} Hz/s (peak 0.312 V for 0.3 V), then a tone on TX2 at −8 MHz (0.105 V for 0.1 V): I and Q in quadrature, the chirp's frequency rising through the pulse.

```
dev.apply({"analog": {"1": {"dc": 0.0, "limits": {"low": -0.3, "high": 0.3, "protect": True}}},
           "guard": {"1": {"max_pulse": 20e-6, "max_duty": 0.02, "burst": 100e-6}}})
```

A trip blanks the transmitter's RF from that very sample, closes the gate outputs, stops the run 3 clocks later, writes an ERROR record, and `state()["guard"]` says which check tripped. The gates end as after any pulse: each closes its `lag` (and 32 ns) after the cut RF's end, so a receiver protection switch stays on through the probe's ring-down. `protect` (or `dev.protect(n)`, TCP `PROTECT n`) locks output n's limits and transmitter n's guard until the FPGA is reloaded: no configuration, preset or SAFE can change them. Set them from the amplifier's and the probe's ratings, not from the experiment. Unset, nothing but the DAC's range (about ±1.1 V) limits a pulse.

4.5 Digital outputs

D1–D8 on DIO0_N–DIO7_N are in high impedance until a mode is set and after SAFE. Each output has a source, a delay (0 … 2 µs in 2 ns steps) and an inversion:

Mode	Source
`off`, `low`, `high`	high impedance, or a static level
`pattern`	four 2 ns bits repeated every 8 ns
`ttl`	TTL bit `bit` of the program's instructions
`gate1`, `gate2`, `gate`	TX1's, TX2's or either's pulses, with a lead and a lag (`gates`)
`capture`, `shot`, `run`	the raw capture's window, a SHOT marker, the run
`state`	window `bit`'s last single-shot state (section 6)

A gate output opens `lead` seconds before the pulse (up to about 130 ns, from the FIFO's look-ahead) and closes `lag` after it (up to 524 µs), both rounded up to whole clocks (8 ns: they are minimums; the applied configuration shows the values, 100 ns becomes 104 ns): a power amplifier is unblanked before the RF arrives and a receiver protection switch stays closed while the probe rings down:

```
dev.apply({"outputs": {"1": {"mode": "gate1"}, "2": {"mode": "gate1", "invert": True}},
           "gates": {"tx1": {"lead": 100e-9, "lag": 2e-6}}})
```

The outputs are aligned with OUT1 at the connectors through the calibration's pin delay (measured on D7: 1.5 ns early, within ±1 ns of the 2 ns grid; a gate on D1 with lead 96 ns and lag 200 ns opened and closed within 2.5 ns of them).

4.6 Slow outputs

AO0–AO3 on E2 give 0 … 1.8 V through the board's RC filters (PWM at 488 kHz with a second-order ΔΣ modulator, 16 bits). Outside a run they hold their idle levels; an instruction sets an output from one of eight AO registers or from a table, so a program can step a magnet's field supply, an attenuator or a bias between scans:

```
dev.apply({"ao": {"values": [0.0] * 4, "registers": [0.9] + [0.0] * 7},
           "program": [{"duration": 1e-3, "ao": {"output": 0, "register": 0}}, {"op": "stop"}]})
```

The ESR field sweep (section 8.7) steps AO0 through a table, its levels converted to codes through the calibration. On the test board AO0's mean was within 28 mV of the set level from 0.1 to 1.7 V (an oscilloscope's mean). The board's first-order filter leaves the 488 kHz ripple large at the pin: 0.23 V peak to peak at 0.1 V, 0.91 V at 0.9 V. A field supply's control input or a bias needs a low-pass filter of its own.

5 Receivers, windows and averaging

5.1 Receivers

Two receivers, RX1 and RX2, each take IN1, IN2, the pair IN1 + i IN2 (for an IQ demodulator) or the emulator (`rx.n.input`). The input's calibrated offset is subtracted, then a complex mixer multiplies it by the oscillator of TX1 or TX2 (chosen per window), so the receiver turns with the transmitter: a signal at the transmitter's frequency is a constant, its phase fixed by the pulse, and the receiver is coherent with the pulses by construction, through phase cycles, frequency switches and chirps. An order-2 CIC filter then averages and decimates: a point every R samples (the *decimation* $R = 1 \ldots 16384$, the dwell $8\,\text{ns} \times R = 8\,\text{ns} \ldots 131\,\mu\text{s}$), each point a triangle-weighted mean over $2R - 1$ samples centred at $R - 1 + kR$ samples from the window's start. Its response across the band, sinc^2-like, is divided out by `A.spectrum`. In `raw` mode the mixer is bypassed: averaged RF waveforms.

5.2 Windows

A window is started by an instruction (`window`: 0–7) and takes `points` points of its receiver; eight window descriptors set what each does:

```
win = {"rx": 1, "lo": 1, "raw": False, "average": True, "weigh": False, "points": 200,
       "decimation": 8, "region": 0, "region_table": None, "weights": None, "phase": 0.0,
       "threshold": 0.0}
dev.apply({"windows": {"0": win}, "rx": {"1": {"input": "in1"}}})
```

A receiver runs one window at a time (4 samples apart); a window started while its receiver is busy is not taken, and `state()["windows_missed"]` counts it. Each window has an overload flag (an input at the converter's limit during it). Windows start at the IN connector at their instruction's start, through the calibration's input delay.

5.3 The receiver's phase

An instruction's `rx_phase` (0–3) turns the window's points by 0°, 90°, 180° or 270°, exactly (by swapping and negating I and Q). Stepping the transmitter's phase and the receiver's together (the

CYCLOPS cycle 0°/90°/180°/270°, or 0°/180°) adds the signal in every step while the receiver's offsets and, with an I/Q pair, its gain and phase errors cancel in the sum: the cycle is subtracted on the board, before the average.

5.4 The averager

Every window with `average` adds its points into the averager: per receiver two halves of 2048 complex points (35-bit sums, saturation flagged). A window's points go to a *region*: its base `region` plus a table entry (`region_table`) indexed by a loop counter, so each array point, phase-cycle step or 2-D increment has its own place. While one half fills, the other can be read: at a SHOT after `average.scans` scans, or when asked (`dev.swap()`), the halves swap without dead time (a released half is cleared in 16.4 µs first). The service reads each frozen half into exact 64-bit sums on the host every 0.1 s (a half in about 0.15 s, measured), so averaging can run for hours without the PC:

```
dev.collect(); dev.clear_average()          # earlier sums out
dev.run()
while dev.state()["running"]:
    time.sleep(0.01)
dev.swap(); dev.collect()                   # the last half in
avg = dev.window_average(0)                 # {"scans", "blocks", "dt", "t0", "volts": [[re, im],
    ↪ ...]}
```

`window_average(w, offset)` gives the points of window w from region `offset` on, in volts: a tone's amplitude and phase at the oscillator (in raw mode, the input's volts). The templates' `X.acquire` does all of this.

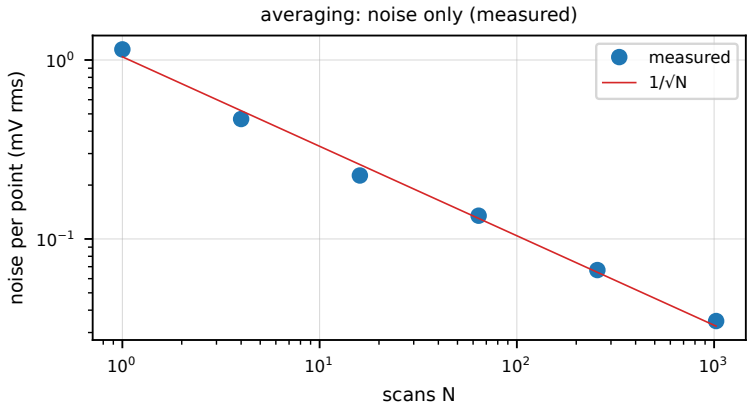


Figure 3: Averaging (measured, the emulator with 10 mV rms of noise and no signal, 128 points of 0.5 µs): the noise per point falls with the number of scans N with a slope of −0.491 on these logarithmic axes (−1/2 expected), from 1.146 mV for one scan to 0.0348 mV for 1024.

Figure 3 shows the averager's gain: noise falls as $1/\sqrt{N}$. One scan's noise per point, 1.042 mV (the mean of $\sqrt{N}$ times the measured rms), is the input's 10 mV per sample reduced by the CIC filter's noise gain $\sqrt{\sum h_j^2}/\sum h_j$ for $R = 62$: 1.037 mV.

5.5 Weighted results

A window with `weigh` also sums its points with complex weights into one value per shot, $S = \sum_k w_k Z_k$, at the full rate (8 ns points allowed): a matched filter, a boxcar (`weights`: null) or a baseline correction (negative weights). The driver normalizes the weights so that a steady tone of amplitude A at the oscillator reads $S = A$ (in volts) and turns them by the window's `phase`; the window's *state* is $\mathrm{Re}\, S >$ `threshold`. Each receiver holds 1024 weights, shared in window order by its weighed windows. Every weighed window writes a RESULT record (shot, time, S, state, overload) into a ring in the board's

memory; the service keeps the last 100 000 (`dev.results(n)`) and streams them (`RECORDS STREAM`, section 10): 100 000 RESULTs per second to a PC without loss (measured over 0.2 s).

5.6 The raw capture

An instruction with `capture` starts the raw capture: 1 … 4096 samples of IN1 and IN2 at 8 ns, from the IN connector at the instruction's start (`dev.capture()`: volts, the delay used and the residual below one clock). It shows pulses, ring-down and saturation as they are, before any mixing.

How it is verified

`tb_seq` compares the receivers' points, the averager's frozen halves and the RESULT records with the model clock by clock, `tb_avg` the averager's saturation after 32 769 full-scale additions and its clearing, `tb_rec` the records' priority and loss counting. On the board a receiver and the capture agree within 0.04 %, the points' phase within 0.2° (loopback); in CYCLOPS a phase-cycled TX1 adds while an uncycled TX2 at the same frequency cancels to below 0.2 %; and the emulator's averages equal the model's bit for bit.

6 Decisions

6.1 A window's state

A window with weights decides at its end: its state is 1 when $\mathrm{Re}(S\,\mathrm{e}^{-\mathrm{i}\varphi})$ exceeds its `threshold` (volts), φ its `phase` (the weights are turned by $-\varphi$). The core keeps each window's last state (0 after a RUN) for branches and digital outputs, and every RESULT record carries it. For a readout, φ lies along the difference of the two states' mean values and the threshold at their midpoint; `X.decision` computes both from the averaged readouts of the ground and the excited state (the references every qubit template records):

spin/control/spin/experiments.py

```
def decision(ground, excited):
    """(phase in degrees, threshold in volts) of a decision window from the averaged readouts of
    the ground and the excited state (complex volts at TX2's LO): the state is 1 (excited) when
    Re(S e^(-i phase)) > threshold, the midpoint."""
    import cmath
    import math
    d = complex(excited) - complex(ground)
    ph = cmath.phase(d)
    mid = (complex(ground) + complex(excited)) / 2
    return math.degrees(ph), (mid * cmath.exp(-1j * ph)).real
```

In use:

```
ref = X.acquire(dev, X.rabi(q, [0.5e-6]))           # regions: the pulse, then ground, excited
phase, threshold = X.decision(ref["traces"][1][0], ref["traces"][2][0])
dev.apply({"windows": {"0": {"phase": phase, "threshold": threshold}}})
```

Section 8.9 shows two such blobs of single shots and the threshold that separates them.

6.2 Branches

A BRANCH plays like an ordinary instruction (its duration, TTL bits and pulses; it starts no window) and then continues at its `target` if its condition holds, else at the next instruction; `not` inverts the condition:

The engine waits at a BRANCH until `bvalid`, then fetches the target or the next instruction (`sp_seq`: `bsrc` the condition's source, `pend` each window's pending starts, `carm`/`ccnt` COND's read 23 clocks after the branch's start, `li[k][0]` the loop counter's lowest bit, `ir4[31]` the inversion):

spin/src/sp_seq.v

```
    wire bvalid = bsrc == 2'd0 ? pend[bwin] == 8'd0 : bsrc == 2'd3 ? 1'b1 : (carm && ccnt == 5'd0);
    wire bval = bsrc == 2'd0 ? w_state[bwin] : bsrc == 2'd1 ? cond[0] : bsrc == 2'd2 ? cond[1] : li[
    ↪ k][0];
    wire [9:0] bdec = (bval ^ ir4[31]) ? target : pc + 10'd1;
```

The conditions:

Condition	True when
`{"state": w}`	window w's last state is 1 (once every window w started before has ended)
`{"cond": "a"}`, `"b"`	COND A or B (DIO1_P, DIO2_P) is high at the branch's start
`{"parity": k}`	loop counter i_k is odd

This program reads a qubit out in window 0, waits for the decision and applies a π pulse only when the state is 1 (an active reset):

```
win = {"rx": 1, "lo": 2, "weigh": True, "average": False, "points": 16, "decimation": 4,
       "threshold": 0.0}
dev.apply({"windows": {"0": win},
           "program": [{"duration": 0.5e-6, "window": 0, "tx2": {"on": True}},     # the readout
                       {"op": "branch", "if": {"state": 0}, "target": 3, "duration": 2e-6},
                       {"op": "jump", "target": 4, "duration": 0.5e-6},           # state 0: no
    ↪ pulse
                       {"duration": 0.5e-6, "tx1": {"on": True}},                 # state 1: pi
                       {"duration": 1e-6}, {"op": "stop"}]})
```

The decision takes time, and a branch that ends before its condition is known would leave the player without an instruction (an underflow, which stops the run). So the driver refuses a branch on COND shorter than 256 ns (the input is read 23 clocks after the branch's start; 29 clocks are the least), and a branch on a window's state unless the instructions from the window's start to the branch's end last at least $(K+1)R + D + 27$ clocks (K points of decimation R, D the receiver's delay: 89 clocks with the emulator). These least durations were measured on the board to the clock (THEORY 2.2). Durations taken from tables are not checked.

6.3 Latencies

From a window's last sample at the IN connector, a branch can start the chosen pulse at the OUT connector $D + 28$ clocks later: 512 ns with the nominal 18-clock converter loop, 488 ns on the test board ($D = 33$ from its calibrated 120 ns loop). A state output changes $D + 5.5$ clocks plus the board's pin delay (`pins_delay_s`) later: 332 ns nominal, 279 ns computed for the test board. An oscilloscope confirmed both on the test board with DC steps (a transmitter at 0 Hz; a window reads twice a DC level, its gain being set for tones). The chosen pulse came 800 ns after the window's input step, as programmed. The state output came 281 ns after the window's last sample, or 277 ns referred to the connectors (THEORY 2.1, VERIFICATION, 2026-11-08).

6.4 State outputs

A digital output in mode `state` shows a window's state for another instrument (a counter, a second board, a scope): `{"mode": "state", "bit": w}`. It has the output's own delay only, no alignment shift.

6.5 Active reset and repeat-until-success

The templates `X.active_reset` and `X.repeat_until_success` put these together on a qubit. An active reset prepares a state (a $\pi/2$ pulse: half excited), reads it out in window 1, applies a π pulse if the state is 1, and reads out again in window 0; repeat-until-success prepares and reads out until the state is 1 (at most `attempts` times):

```
q = dict(q, phase=phase, threshold=threshold)        # the decision from the references
for exp in (X.active_reset(q, reset=False), X.active_reset(q), X.repeat_until_success(q, attempts=8))
    ↪ :
    z = [tr[0] for tr in X.acquire(dev, exp)["traces"]]
    print(exp.name, A.population([z[0]], z[1], z[2])[0])   # the excited population after it
```

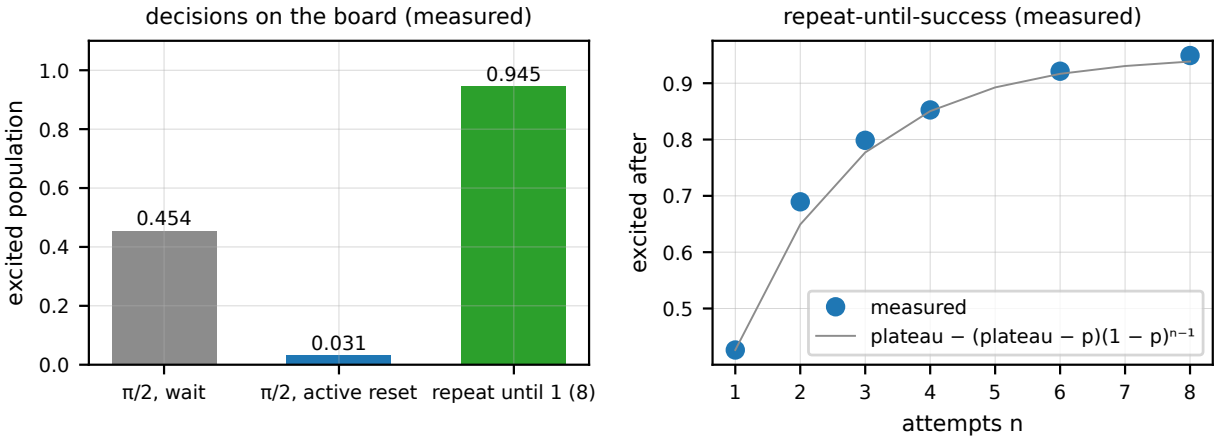


Figure 4: Decisions on the emulated qubit ($T_1 = 20\,\mu s$; measured, 2000 shots each). Left: after a $\pi/2$ pulse 0.454 of the qubit is excited; an active reset leaves 0.0311; repeat-until-success with up to 8 attempts reaches 0.9446. Right: repeat-until-success against the number of attempts, from 0.426 after one.

Figure 4 shows the board's results. The decision used $\varphi = -90.0°$ and a threshold of $-0.40\,mV$ from the references. The reset's residual excitation, 0.0311, comes mostly from decay between the readout's projection and the π pulse: a qubit read as 1 that decays before the π pulse arrives is excited by it (about $1.5\,\mu s$ at $T_1 = 20\,\mu s$ for a half-excited qubit: a few per cent); the bit-exact model expects 0.023 for these settings. The theory manual (section on qubit readout) works out this error budget.

How it is verified

`tb_seq` plays branches on a parity, on a window while it still runs, on a window whose start was missed, on a window without weights, on COND A and inverted COND B driven by the bench, all at their least durations, and one clock short (an underflow) for COND and for a state. On the board (`test_hw`): a parity loop, the least durations to the clock, seeded active-reset and repeat-until-success runs equal to the bit-exact model shot for shot. With a jumper from D8 to COND A the test also checks a branch following the output's level and a state output carried to a branch: both passed, twice (62 of 62 checks, 2026-11-08).

7 The emulator

The emulator is a spin, or a qubit, inside the FPGA. It is driven by TX1's pulses exactly as a real spin would be (their amplitude after the envelope, their phase, their chirp), and it answers at the receivers whose input is `emulator`: every experiment, its timing, its phase cycle and its analysis run unchanged, with no magnet, probe or qubit. It is a teaching tool and the board test's reference: its averages equal the Python model's bit for bit.

```
dev.apply({"emulator": {"mode": "ensemble", "frequency": 10.02e6, "rabi": 1.25e6, "t1": 1e-3,
                        "t2": 300e-6, "signal": 0.1, "noise": 0.002,
                        "detuning": {"shape": "gaussian", "width": 20e3}, "seed": None},
           "rx": {"1": {"input": "emulator"}}})
```

Setting	Meaning
`mode`	`off`, `ensemble` (spins in a magnet) or `qubit` (a qubit read out through a resonator)
`frequency`	the spin's (or qubit's) frequency in Hz; TX1 within 3 MHz of it
`rabi`	the Rabi frequency per volt of TX1's amplitude (Hz/V): 1.25 MHz/V gives 1 MHz at 0.8 V
`t1`, `t2`	relaxation times (s, or null). `t2` at most 2 `t1`; null with a T_1 means $2T_1$
`excited`	the thermal excited population (M_z relaxes to $1 - 2$ `excited`)
`detuning`	a line: `none`, `gaussian` (rms width) or `lorentzian` (half width), in Hz; an offset is drawn from it at every SHOT
`ao`	Hz per volt: AO0 moves the spins' frequency (an emulated field sweep)
`signal`	ensemble: the signal at the input for $\|\boldsymbol{M}\| = 1$; qubit: the readout's amplitude per volt of TX2 (V)
`chi`, `ringdown`	qubit: the readout's phase $\pm\chi$ for ground and excited (degrees), the resonator's time constant (s)
`noise`	Gaussian noise added at the input, rms per quadrature (V)
`reset`, `seed`	each run starts from equilibrium; a seed makes the draws repeat (null: free-running)

7.1 What it computes

One Bloch vector is updated every 128 ns (a *slot* of 16 clocks) by an exact rotation for that slot's drive and offset, with T_1 and T_2 folded in. The vector lives in TX1's rotating frame, so the offset is the spin's frequency minus TX1's (plus the shot's drawn offset and AO0's shift), within ±3.125 MHz; TX1 at full amplitude can drive up to 1.95 MHz of Rabi frequency. Its answer reaches the receivers 89 clocks after the drive (the driver sets the receivers' delay to that), so windows line up with the pulses exactly as with a real input.

Ensemble mode gives the transverse magnetization in TX1's frame. A line is emulated by drawing one offset per SHOT: each shot is *one* isochromat, which does not dephase within the shot, and the average over shots shows the line, its T_2^* and its echoes. With N shots the average keeps a residue of about $1/\sqrt{N}$ of the signal where a real ensemble would have dephased completely (figure 5).

Qubit mode reads the qubit out through TX2: at the start of each readout pulse an outcome is drawn from the excited probability $(1 - M_z)/2$, the vector collapses to that state and decays with T_1 during the readout, and TX2's pulse returns through a one-pole resonator at $+\chi$ (ground) or $-\chi$ (excited) in TX2's frame, with the noise: read it with TX2's oscillator (`"lo": 2`). `dev.emulator_state()` gives the vector and the last outcome.

7.2 Using it

- Keep `t2` ≤ 2 `t1`: no spin has a longer T_2, and the emulator's vector would grow beyond 1. Leave `t2` null for a qubit without extra dephasing.
- TX1's oscillator must run before a run's first pulse: after a RUN its step is 0 until TX1's first coherent start, and a pulse in the same 128 ns slot as such clocks turns about a tilted axis. The templates' 256 ns preludes start TX1 without output; a hand-written program should do the same (`"tx1": {"on": False, "freq": 0}` in an instruction before the first pulse).

spin/control/spin/experiments.py

```
def _prelude(extra):
    """An acquisition's prelude (PRELUDE clocks, with `extra`: its SHOT): TX1's NCO starts at
    ↪ its
    frequency without output, so the emulator's frame follows TX1 from the slot of the first
    pulse on (after a START TX1's step is 0 until its first coherent start: a pulse in a slot
    with such clocks would turn about a tilted axis, THEORY 2.8)."""
    return dict({"duration": PRELUDE, "tx1": {"on": False, "freq": 0}}, **extra)
```

- With a `seed`, `X.simulate(dev, exp, start=r["start"])` reproduces a board run exactly (the slots follow the board's clock, so the model starts at the run's START clock that `acquire` returns).
- A readout must start at least 256 ns after the last drive (the outcome is drawn from the state at the end of the last slot before the readout's); the qubit templates do so.

How it is verified

`tb_emu` compares the emulator's output at every clock and its vector after every update with `emulator.py` (three scenarios, all 55 mutants of the RTL fail it). On the board, seeded runs equal the model's averages bit for bit, and free-running runs reproduce the physics: T_2^*, T_2 by CPMG, T_1 by inversion recovery, Rabi and Ramsey frequencies within a few per mille (section 8). With physical T_1 and T_2 the emulator agrees with the float Bloch equations (`bloch.py`) within 0.1 %.

8 Experiments

`spin.experiments` builds the classic experiments as configurations to apply: TX1's pulses, TX2's readout, a receiver window, the pulse program with its phase cycle, and the tables that array it. Every figure in this section was recorded on the board, on the emulator, with `tools/record_manual_data.py`, and analysed with `spin.analysis` (`tools/manual_figures.py` draws them).

8.1 Templates

The ensemble templates (NMR, NQR, ESR) take a `pulses` specification:

```
p = X.pulses(f=10e6,            # TX1's frequency and the receiver's oscillator (Hz)
             amplitude=0.8,     # TX1's amplitude (V); t90 turns by 90 degrees, 2 t90 by 180
             t90=0.25e-6,       # (s)
             dwell=0.5e-6,      # the receiver's dwell (s): 8 ns x the decimation
             points=256,        # per trace (all traces together at most 2048)
             dead=2e-6,         # after a pulse, before a window (s)
             tr=5e-3,           # the repetition time (s)
             scans=64, dummies=0, cycle="cyclops", input="emulator")
exp = X.fid(p)
print(exp)                      # its traces, points and run time
```

Template	Experiment (each array value in its own averager region)
`fid(p)`	a 90° pulse and the free induction decay
`echo(p, tau)`, `t2(p, taus)`	Hahn echoes 90x–τ–180y–τ, the window centred on the echo
`cpmg(p, tau, n)`	CPMG: n echoes of 180°y pulses 2τ apart, each echo its own region
`slse(p, tau, n, flip)`	NQR's spin-locked echo train: CPMG with `flip`-degree refocusing pulses
`inversion_recovery(p, delays)`	180–t–90, the FID's start against t
`saturation_recovery(p, delays)`	a comb of 90° pulses, t, 90
`nutation(p, lengths)`	a pulse of each length, the FID's start: the Rabi frequency
`field_sweep(p, tau, volts)`	ESR: a slow output's level per point, then a Hahn echo

The qubit templates take a `qubit` specification: TX1's drive (`f`, `amplitude`, `t_pi`), TX2's readout (`f_read`, `amp_read`, `t_read`), `tr`, `scans`, and a decision's `phase` and `threshold`: `rabi`, `ramsey`, `t1_qubit`, `readout`, `active_reset`, `repeat_until_success`. Their last two regions are references (no drive: the ground state; a π pulse: excited), from which `A.population` maps the others. Each acquisition starts with a SHOT and a 256 ns prelude; phase cycles are unrolled with the receiver's phase; dummy scans can precede the scans.

`X.acquire(dev, exp)` applies the configuration, empties the averager, runs and collects, and returns `{"name", "axis", "t", "traces", "scans", "start"}`: a trace per region (complex volts at the oscillator, one value per point), the points' times from the experiment's reference (the pulse's centre, the echo's top) and the run's START clock. `X.simulate(driver, exp)` runs the same experiment in the bit-exact model on a PC. The fits are `A.fit_fid`, `fit_decay`, `fit_recovery`, `fit_oscillation`, `spectrum`, `population` and `readout`.

8.2 The FID and its spectrum

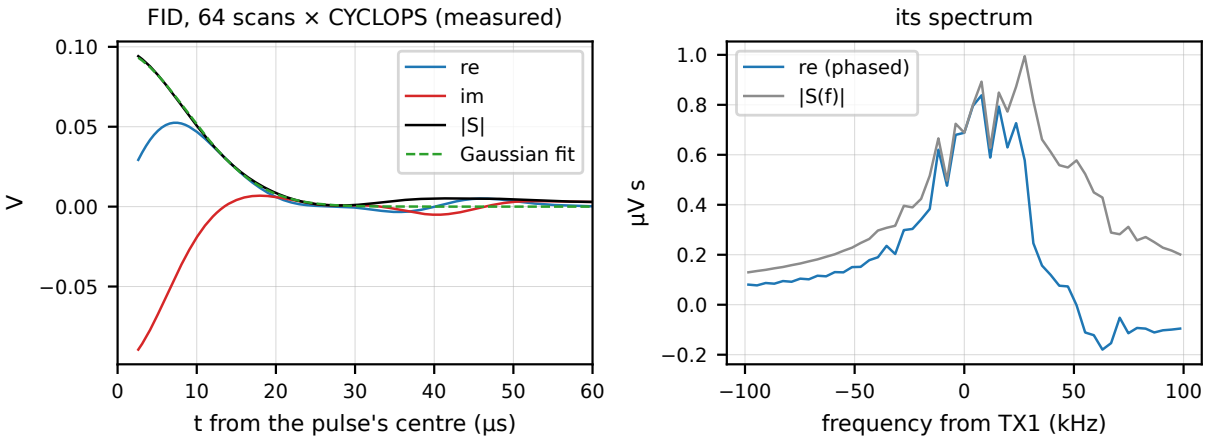


Figure 5: The FID of the emulated ensemble (10.02 MHz, a 20 kHz Gaussian line, $T_2 = 300$ µs; measured, 256 shots: 64 scans of CYCLOPS). Left: the averaged trace and a Gaussian fit to its first 20 µs; right: its spectrum (first point halved, zero-filled, the CIC's droop divided out).

Figure 5 is the experiment of section 2.2. The trace turns at the spins' offset from TX1: `A.fit_fid` finds 19.09 kHz for the set 20 kHz (and an exponential T_2^* of 8.23 µs, a single number for a Gaussian decay). A Gaussian line of rms width σ makes the FID's envelope decay as $\exp(-(2\pi\sigma t)^2/2)$; fitted over the first 20 µs it gives $\sigma = 17.61$ kHz and an amplitude of 0.0980 V (0.1 set). The emulator's line is a draw of one offset per shot, so a run of 256 shots samples it: such runs scatter by 1.3 kHz in this fit (5–95 %: 17.7–21.8 kHz, simulated), and after the decay a residue of about $1/\sqrt{N}$ remains (here 2.9 % of the start after 60 µs), where a real sample's spins would have dephased to zero. For the same reason the spectrum is the histogram of the drawn offsets, broadened by T_2: its highest point (27.6 kHz) moves from run to run within the line.

8.3 The Hahn echo

```
r = X.acquire(dev, X.echo(dict(p, points=128, scans=16), 100e-6))   # 90x - 100 us - 180y - 100 us
```

The FID has decayed long before 2τ (figure 6); the 180° pulse reverses the phases the offsets accumulated and the spins refocus at 2τ: the echo's top is 0.0510 V, the signal reduced by T_2 alone ($0.1\,\mathrm{e}^{-2\tau/T_2} = 0.0513$ V). The FID at 2τ keeps only the $1/\sqrt{N}$ residue of its 64 shots (6.54 mV).

8.4 T_2 and echo trains

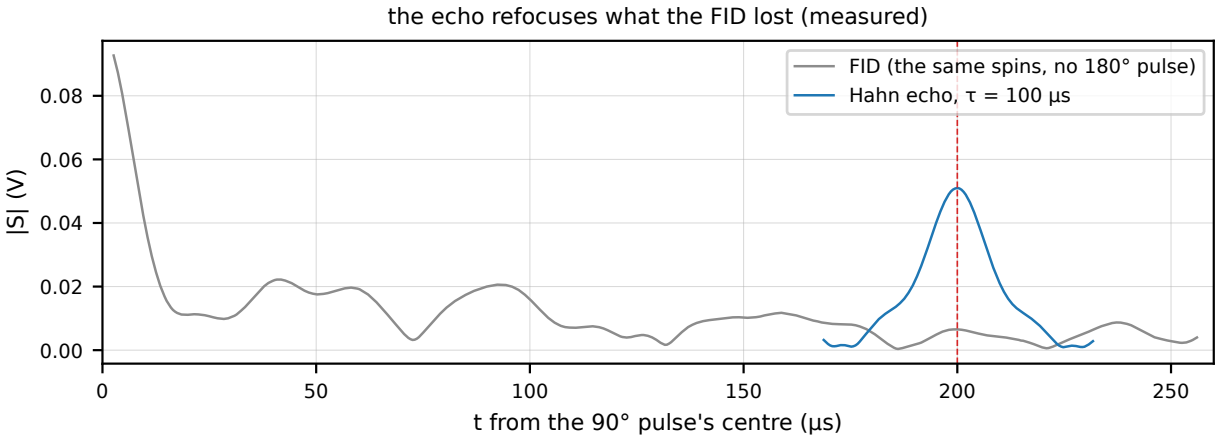


Figure 6: A Hahn echo of the emulated ensemble at $2\tau = 200\,\mu s$ against the FID of the same spins (measured, 64 shots each).

```
p = X.pulses(f=10e6, amplitude=0.8, t90=0.25e-6, dwell=0.5e-6, points=256, tr=5e-3, scans=64)
taus = [50e-6 * k for k in range(1, 13)]
exp = X.t2(dict(p, points=16, dwell=1e-6, scans=8), taus)          # echo tops against 2 tau
trains = [X.cpmg(dict(p, points=16, scans=256), 10e-6, 32), X.slse(dict(p, points=16, scans=256), 10
    ↪ e-6, 32)]
print(exp, trains)
```

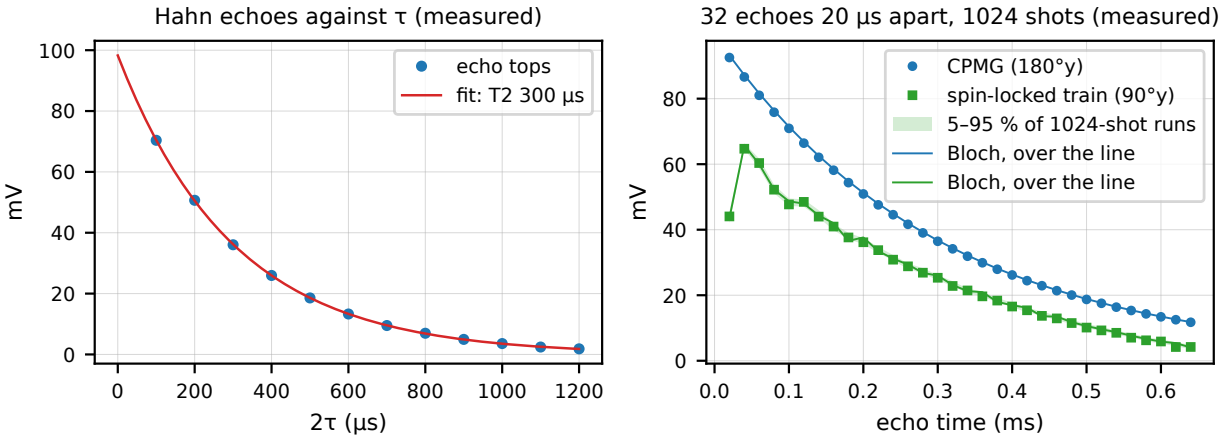


Figure 7: Left: Hahn echoes against 2τ and their exponential fit (measured). Right: 32 echoes in one scan, CPMG (180°y) and the spin-locked train (90°y), 1024 shots, with the float Bloch equations averaged over the line and the 5–95 % spread of 1024-shot runs (measured; curves computed).

The echo tops decay with $T_2 = 300.3 \pm 0.6\,\mu s$ (figure 7, left; the emulator's 300 µs). A CPMG train takes every echo in one scan; its tops give $300.5 \pm 0.1\,\mu s$. The spin-locked train with 90° refocusing pulses (`slse`) starts at about half the signal and, on this spin-1/2 line 20 kHz off resonance, decays faster than CPMG (its last echo 4.1 mV expected against CPMG's 11.8 mV). Its echoes depend strongly on each isochromat's offset, so the board's run is compared with the Bloch equations averaged over the line, and with the spread that runs of 1024 shots have: 91 % of its echoes lie inside that band. The long trains of NQR come from quadrupolar spins in a powder, which the emulator (a spin 1/2) does not model.

8.5 T_1: inversion and saturation recovery

```
p = X.pulses(f=10e6, amplitude=0.8, t90=0.25e-6, dwell=0.5e-6, points=256, tr=5e-3, scans=64)
delays = [50e-6, 100e-6, 200e-6, 400e-6, 700e-6, 1e-3, 1.5e-3, 2e-3, 3e-3, 5e-3]
ir = X.inversion_recovery(dict(p, points=4, dwell=1e-6, scans=16, cycle="2"), delays)
sr = X.saturation_recovery(dict(p, points=4, dwell=1e-6, scans=16, cycle="2", tr=0.5e-3), delays)
print(ir, sr)
```

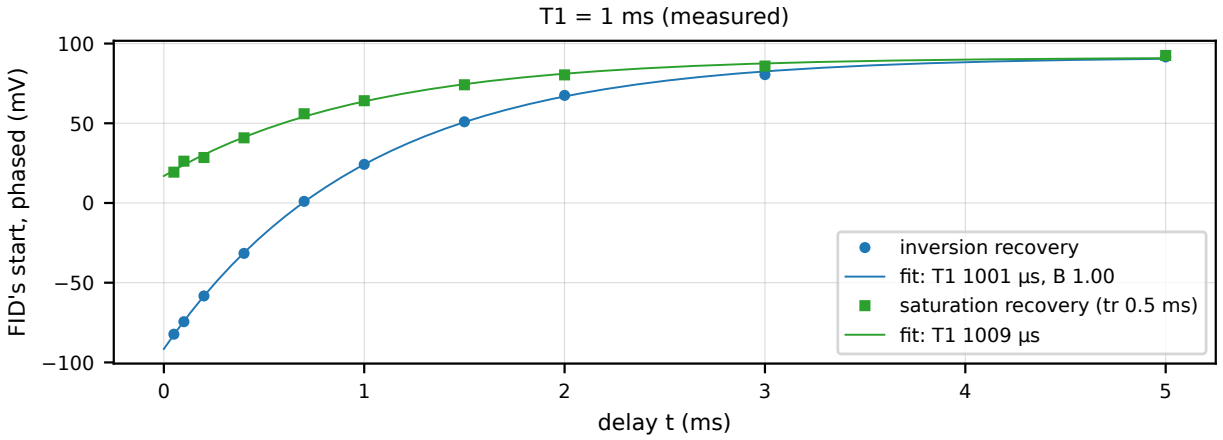


Figure 8: Inversion and saturation recovery of the emulated ensemble ($T_1 = 1\,\text{ms}$; measured).

Inversion recovery (figure 8) gives $T_1 = 1001 \pm 14\,\mu\text{s}$ with an inversion $B = 0.999$ ($B = 1$ is perfect); it needs a repetition time of about $5\,T_1$ between scans. Saturation recovery starts each scan with a comb of 90° pulses instead, so its scans can be 0.5 ms apart: $T_1 = 1009 \pm 62\,\mu\text{s}$.

8.6 Nutation

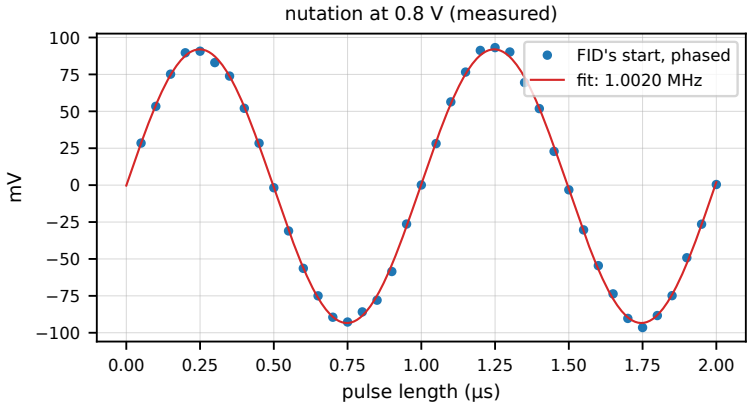


Figure 9: Nutation at 0.8 V: the FID's start against the pulse's length (measured).

The FID's start follows $\sin\theta$ with the pulse's flip angle $\theta = 2\pi f_\text{R} t_p$ (figure 9): the fit gives $f_\text{R} = 1.0020\,\text{MHz}$ (the emulator's 1 MHz at 0.8 V; the lengths, 50 ns steps, are rounded to the 8 ns clock), so $t_{90} = 249.5\,\text{ns}$. With a real probe, B_1 inhomogeneity damps the oscillation; the first maximum is the 90° pulse, the first zero the 180°.

8.7 An echo-detected field sweep

```
volts = [0.7 + 0.02 * k for k in range(41)]
soft = X.pulses(f=10e6, amplitude=0.1, t90=2e-6, dwell=1e-6, points=4, dead=2e-6, tr=60e-6, scans=32,
    ↪
                cycle="2")
exp = X.field_sweep(soft, 10e-6, volts, output=0, settle=2e-6)   # AO0 from a table, then an echo
print(X.config_for(dev, exp)["tables"]["data"][2][:3])          # the levels as codes (calibration)
```

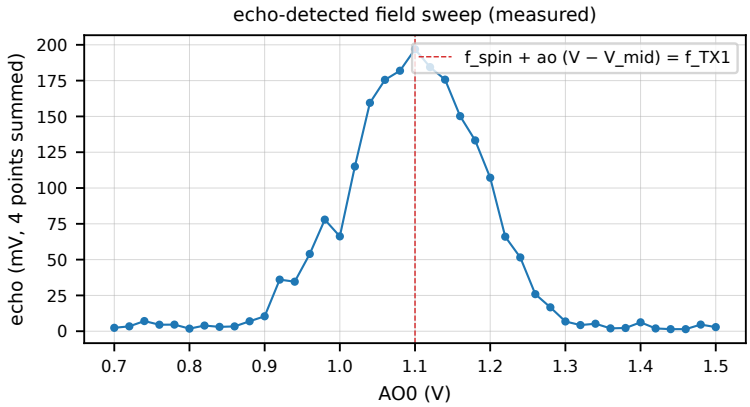


Figure 10: An echo-detected field sweep (measured): the emulator's spins move with AO0 at $-1\,\mathrm{MHz/V}$ across TX1; a 50 kHz line, soft pulses ($t_{90} = 2\,\mu\mathrm{s}$).

In pulsed ESR the line is broad against the pulses' band and the FID is lost in the dead time, so the echo is recorded against the magnetic field. Here AO0 plays the field (figure 10): the line's centroid lies at 1.098 V, where the spins' frequency equals TX1's (1.100 V); its width at half height, 0.18 V, is the line convolved with the soft pulses' excitation band.

8.8 Qubit: Rabi, Ramsey and T_1

```
q = X.qubit(f=10e6, amplitude=0.8, t_pi=0.5e-6, f_read=12e6, amp_read=0.2, t_read=0.5e-6, tr=100e-6,
            scans=300)
rabi = X.rabi(q, [0.05e-6 * k for k in range(1, 41)])
print(rabi.axis[-2:])                          # the two references
```

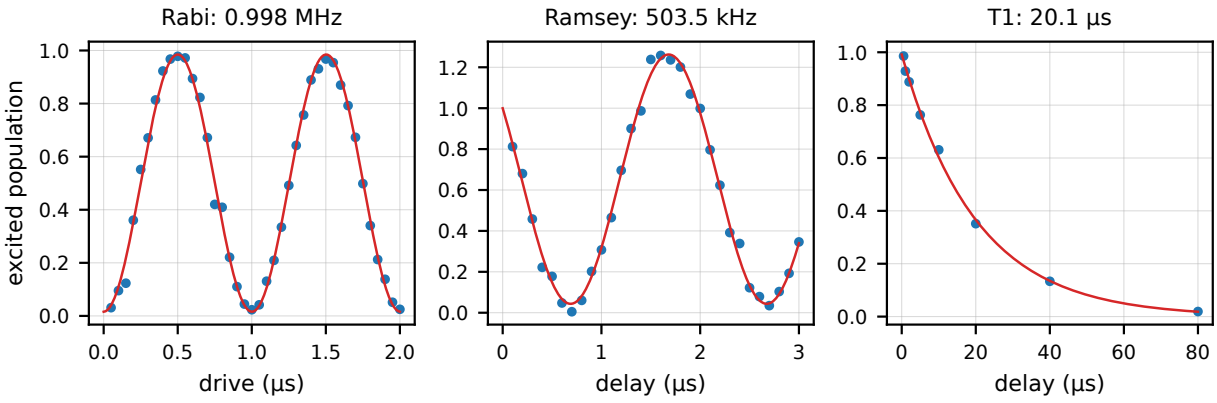


Figure 11: The emulated qubit (10 MHz, $T_1 = 20\,\mu\mathrm{s}$, readout at 12 MHz through a resonator at $\pm 60°$; measured, 300 shots per point): Rabi oscillations, Ramsey fringes 500 kHz off resonance, and T_1.

Each point is a readout's average, mapped to the excited population by the two references (figure 11): the Rabi frequency 0.9979 MHz (1 MHz set), Ramsey fringes at 503.5 kHz (500 kHz), $T_1 = 20.07 \pm 0.71\,\mu\mathrm{s}$ (20 µs).

8.9 Single-shot readout

```
ro = X.readout(dict(q, tr=100e-6), 2000)      # 2000 pairs: no drive, then a pi pulse
dev.apply(ro.config)
dev.run()
shots = dev.results(4000)                     # RESULT records: S in volts per shot
```

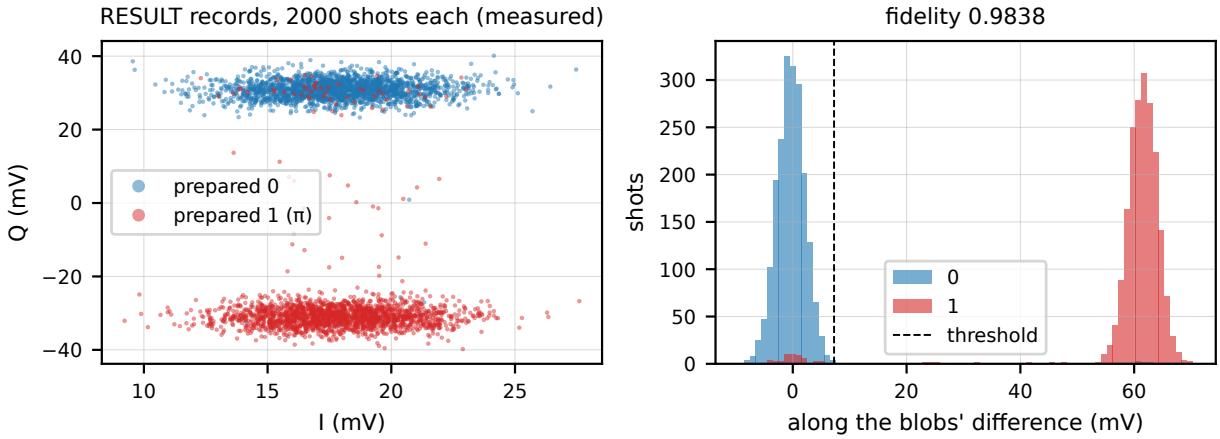


Figure 12: Single shots of the emulated qubit (measured, 2000 prepared in each state, 20 mV of noise): the RESULT records' I/Q points and their histogram along the blobs' difference with the threshold of fewest errors.

The two states give two blobs in the I/Q plane (figure 12); `A.readout` finds the axis between their means, the threshold with the fewest errors and the assignment fidelity $F = 1 - [P(1|0) + P(0|1)]/2 = 0.9838$ (a separation of 7.60 standard deviations). The errors are not symmetric: 2.70 % of the excited shots read 0, because the qubit can decay during the 0.5 µs readout (the tail from the excited blob towards the ground), and 0.55 % of the ground shots read 1, mostly from excitation left by the previous pair's π pulse after 100 µs (e^{-5} of it). The threshold therefore sits near the ground blob.

9 Real setups

The recipes below connect `spin` to real equipment. None of them was tried with real hardware for this manual: the probe, duplexer, amplifier, bridge and sample behaviour (ring-down, recovery, image rejection, sensitivity) is unmeasured, and the numbers are starting points. Protect the equipment first (section 4.4). The same recipes are in `spin/RECIPES.md`.

9.1 Pulsed NMR with a probe

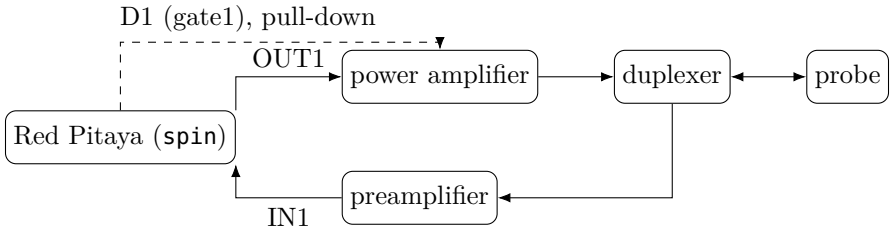


Figure 13: A pulsed NMR setup. D1 unblanks the amplifier around TX1's pulses (with a pull-down at its input); the duplexer (crossed diodes and quarter-wave lines, or a T/R switch, possibly driven by D2 with a longer lag) sends the pulse to the probe and the probe's signal to the preamplifier.

```
dev.load_preset("nmr-fid-in1")                # gated amplifier on D1, OUT1 limited, the duty guard
p = X.pulses(f=21.3e6, amplitude=0.2, t90=1e-6, dwell=1e-6, points=1024, dead=5e-6, tr=2.0,
             scans=8, cycle="cyclops", input="in1")
nut = X.acquire(dev, X.nutation(dict(p, points=8), [0.5e-6 * k for k in range(1, 25)]))
```

1. Find the resonance with a long repetition time and few scans: the FID's beat is its offset from TX1 (`A.fit_fid`'s `df`); move TX1's frequency onto it.
2. Calibrate the 90° pulse by nutation (the first maximum of the FID's start), at a safe amplitude first.
3. Set `dead` from the probe's ring-down and the preamplifier's recovery: watch the first points of a FID with no sample, and the capture (`capture`) right after the pulse.
4. Measure T_2 (`t2`, `cpmg`) and T_1 (`inversion_recovery`, `saturation_recovery`), then the spectrum of a FID with `A.spectrum`.

At low fields (protons below a few hundred kHz) the board's input high-pass (a pole-zero near 21 kHz on the test board) bends the baseline of a long FID; it is not in the calibration.

9.2 NQR: a frequency search

Pure quadrupole resonance needs no magnet but must be found: the lines are narrow and their frequencies drift with temperature. Step TX1's frequency by less than a pulse's band ($\approx 1/t_p$) and record a spin-locked echo train at each step, summing its echoes:

```
p = X.pulses(f=3.6e6, amplitude=0.3, t90=10e-6, dwell=2e-6, points=32, dead=20e-6, tr=0.5,
             scans=16, cycle="2", input="in1")
found = []
for f in [3.55e6 + 10e3 * k for k in range(11)]:
    r = X.acquire(dev, X.slse(dict(p, f=f), 300e-6, 64))            # 64 echoes 600 us apart
    found.append((f, sum(abs(tr[16]) for tr in r["traces"])))        # the summed echo tops
```

In a powder the nutation is not a sine: calibrate the pulse by the largest echo, not by a null.

9.3 Pulsed ESR at an intermediate frequency

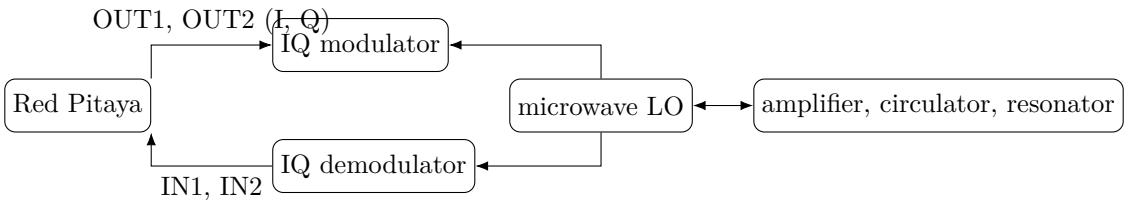


Figure 14: Pulsed ESR at an IF: `spin`'s I/Q pair drives the bridge's IQ modulator, so TX1's frequency is the offset from the microwave LO; the echo returns through the demodulator to IN1 + i IN2. (Simplified: the modulator's output goes through the amplifier and circulator to the resonator, the resonator's signal back through the circulator to the demodulator.)

```
dev.load_preset("esr-iq-echo")                # I/Q mode, Hahn echoes on IN1 + i IN2, the gate on D1
dev.apply({"tx": {"1": {"q_gain": 1.0, "q_phase": 0.0}},             # null the image here
           "analog": {"1": {"dc": 0.0}, "2": {"dc": 0.0}}})           # and the LO leakage
```

Trim the image with a CW tone (`q_gain`, `q_phase` of the transmitter at its frequency) and the LO leakage with the outputs' DC levels, watching the bridge's spectrum. ESR's echoes are short: use dwell times of 8–16 ns, short τ, and `field_sweep` with AO0 on the field supply's control input (its step response sets `settle`).

9.4 Qubit readout at an intermediate frequency

TX1 (or an I/Q pair) drives the qubit through a mixer, TX2 makes the readout pulse at the resonator's IF, and the down-converted readout returns to IN1 (or IN1 + i IN2). Calibrate in this order: the readout's frequency and amplitude (the two states' averages furthest apart), the references and the decision (`X.decision`), the readout's weights (a matched filter: the difference of the two states' mean traces, as `weights`), the fidelity from single shots (`X.readout`, `A.readout`), then Rabi, Ramsey and T_1, and active reset. The decision's latency (section 6) is the time a qubit has to decay between its readout and a conditional pulse.

9.5 Triggers and other instruments

A WAIT on TRIG locks a program to an external event within a nanosecond (section 3.3): a magnet's field cycle, a laser pulse, or `awg` on a second board (its markers on DIO1_P/DIO2_P of its own E1, to `spin`'s DIO0_P). State outputs and COND inputs connect decisions across instruments. The pin maps, levels, cables and the time and frequency across boards are in the guide *Several rpInstruments designs together* (`guides/combining.pdf`).

10 Reference

10.1 Presets

`dev.load_preset(name)` (TCP `PRESET name`) applies a whole configuration; `RUN` starts it. Saved presets (`dev.save_preset(name)`) keep a configuration; limits locked by PROTECT survive every preset.

Preset	What it sets up
`off`	every output in high impedance, both transmitters off, the program a STOP
`pulse-loopback`	four 1 µs pulses of 0.2 V at 10 MHz with CYCLOPS, the gate on D1, IN1 captured and averaged (OUT1 → IN1)
`fid-emulator`	the emulated ensemble (10.02 MHz, a 20 kHz line, T_1 1 ms, T_2 300 µs): FIDs, CYCLOPS, 64 scans
`echo-emulator`	the same: Hahn echoes, $\tau = 100$ µs
`cpmg-emulator`	the same: CPMG, $\tau = 100$ µs, 32 echoes
`nutation-emulator`	the same: pulses of 0.05 … 1 µs
`rabi-qubit`	the emulated qubit (10 MHz, T_1 20 µs, readout at 12 MHz, ±60°): Rabi with references
`ramsey-qubit`	the qubit 500 kHz below TX1: Ramsey over 0.1 … 3 µs
`readout-qubit`	1000 single shots of each state, a RESULT record per shot
`nmr-fid-in1`	a probe on IN1 at 10 MHz: 1 µs pulses of 0.2 V, the amplifier's gate on D1, OUT1 limited to ±0.5 V, the duty guard at 50 µs and 5 %
`esr-iq-echo`	an IQ bridge: I/Q mode, Hahn echoes on IN1 + i IN2, the gate on D1, the same limits

10.2 TCP commands

The service (port 5025) takes the common commands of every design (`*IDN?`, `STATUS?`, `CONFIG?`, `APPLY {json}`, `PRESET`, `SAFE`, `LOAD`, `CAL ...`) and, while `spin` runs, these:

Command	Reply
`RUN`, `STOP`, `TRIGGER`	start (restart) the program, end it, a software trigger
`STATE?`	the run: running, waiting, instruction, FIFO, error, shots, missed triggers and ticks, start and trigger times, the averager, missed windows, lost records, the guard
`CAPTURE? [N]`	the last capture in volts
`PINS? [CLEAR]`	TRIG, COND_A, COND_B: level and edges seen
`HARDWARE?`	the register layout, the samplers' PLL, the ring writer, locked outputs, the slow outputs' levels
`AVERAGE? w [o]`, `AVERAGE CLEAR`	window w's average from the host sums (from region o); forget the sums
`SWAP`, `COLLECT`	swap the averager's halves; read a frozen half now
`RESULTS? [N]`	the last N RESULT records with S in volts
`WORKER?`, `EMULATOR?`	the record worker's counters; the emulator's vector and last outcome
`RECORDS STREAM`	a stream of the records (RESULT, BLOCK, ERROR, RUN, TIME)
`PROTECT n`	lock OUT n's limits and TX n's duty guard until the FPGA is reloaded

The full configuration (`APPLY`, `CONFIG?`) is listed in `spin/docs/PROTOCOL.md`; lists and `shapes` are replaced as a whole, other dictionaries merge.

10.3 The Python interface

Method	
`apply(cfg)`, `config()`, `load_preset(n)`, `safe()`	configure (the common `rpfpga.Overlay` interface)
`run()`, `stop()`, `trigger()`, `state()`	run control
`collect()`, `swap()`, `clear_average()`, `window_average(w, o)`	the averager
`results(n)`, `capture()`, `pins(clear)`	RESULT records, the raw capture, the inputs
`protect(n)`, `emulator_state()`, `hardware()`	locks, the emulator, the hardware
`ao_code(output, volts)`	a slow output's level as a table code (driver)
`make_gui(dev)`	the Jupyter panel (`import spin`)

`spin.experiments`: `pulses`, `qubit`, the templates of section 8, `decision`, `config_for`, `acquire`, `simulate`. `spin.analysis`: `fit`, `phase`, `phased`, `spectrum`, `population`, `fit_decay`, `fit_fid`, `fit_recovery`, `fit_oscillation`, `readout`. `spin.model` is the bit-exact model of the core, `spin.emulator` of the emulator, `spin.bloch` the float Bloch equations.

10.4 Specifications

Program	1024 instructions, 16 ns … 34 s each in 8 ns steps; 4 nested loops; 4 tables of 1024 words; instructions of 32 ns or more without limit, about 30 of 16 ns back to back
Triggers	TRIG time-stamped to 1 ns; the next instruction 192 ns after the edge; RF phase referred to the edge (0.28 ns rms, measured); a pulse at the connector 377–384 ns after the edge (measured); a timer; software
Transmitters	2 × 8 phase-coherent frequency registers (0.44 µHz), 8 phases (0.0055°) and 8 amplitudes each, or tables; 7 chirps each; 31 envelope shapes in a 4096-point table
Outputs	two real outputs or an I/Q pair with per-transmitter Q gain and phase; DC levels; lockable limits; a duty guard per transmitter
Digital outputs	D1–D8: 2 ns bits, delays 0 … 2 µs; TTL, gates (whole clocks, rounded up: lead ≤ 130 ns, lag ≤ 524 µs), markers, states; aligned with OUT1 (D7: 1.5 ns early; a gate on D1 within 2.5 ns; measured)
Slow outputs	AO0–AO3, 0 … 1.8 V, 16 bits; idle levels, registers, tables; AO0 within 28 mV, a 488 kHz ripple of up to 0.91 V p-p at the pin (test board, measured)
Receivers	2 complex receivers on IN1, IN2, IN1 + i IN2 or the emulator; order-2 CIC, 8 ns … 131 µs per point; exact 90° receiver phases; aligned to the IN connector
Averaging	2 halves × 2048 complex points per receiver, swapped without dead time; 64-bit host sums
Weighted results	1024 complex weights per receiver at the full rate; a state per shot; RESULT records, 100 000/s to a PC without loss (measured)
Decisions	branches on a state, COND A/B or a parity; least durations measured to the clock; to a chosen pulse $D + 28$ clocks (512 ns nominal, 488 ns on the test board), to a state output 281 ns (measured)
Emulator	a spin ensemble or a qubit, one Bloch vector per 128 ns, offsets within ±3.125 MHz, Rabi ≤ 1.95 MHz; bit-exact with its model, within 0.1 % of the Bloch equations
Resources	96.1 % of the slices, 13 867 LUTs, 47.5 block RAMs, 71 DSP48 of the XC7Z010

10.5 How it is tested

- **Simulation**: six XSim testbenches with a `PASS:` receipt; `tb_seq` compares the whole core clock by clock with the bit-exact model in eleven scenarios, `tb_emu` the emulator update by update; one-token mutants of new RTL must fail a bench.
- **Host tests**: the driver, protocol and client against the model; the templates through the driver and the model with the emulator against the float Bloch equations; this manual's examples and figures (`tests/test_manual.py`).
- **Board** (`make test-hw`, 62 checks, twice for each release, OUT1 → IN1, OUT2 → IN2 and a jumper D8 → COND A; 58 without the jumper): run control, the timer and underflows, the records and the averager, loopback timing, volts and phases, I/Q mode, shapes, chirps, limits and the duty guard, the receivers, CYCLOPS, weighted results, the emulator bit-exact and against the physics, decisions and their timing, active reset, every preset. Dated records and receipts are in `spin/docs/VERIFICATION.md` and `spin/docs/data/`.

10.6 Files

`spin/src/`	the RTL (`sp_core`, `sp_seq`, `sp_tx`, `sp_rx`, `sp_avg`, `sp_emu`, …)
`spin/control/spin/`	the driver, client, protocol, models, templates, analysis, GUI
`spin/RECIPES.md`	worked setups
`spin/docs/`	THEORY (physics and definitions), REGISTERS, PROTOCOL, VERIFICATION, this manual
`spin/tools/`	host tools: the board recordings and figures of the manuals
`spin/tests/`	the host tests, the testbenches and their vectors

The design builds on the shared framework (`common/`) and the converter interface of dspsandbox's FPGA-Notes-for-Scientists and Pynq-Redpitaya-125 (`spin/NOTICE.md`).

References