pdq Documentation

Release 2.5.1

Robert Jördens

Jul 14, 2017

Contents

1 Contents **1**
- 1.1 PDQ Overview . . . 1
- 1.2 Architecture . . . 2
- 1.3 Reference Manual . . . 4
- 1.4 Code Documentation . . . 11

2 Indices and tables **19**

Python Module Index **20**

CHAPTER 1

Contents

PDQ Overview

A pretty darn quick interpolating arbitrary waveform generator.

Build

Requirements:

- Migen (https://github.com/m-labs/migen)
- MiSoC (https://github.com/m-labs/misoc)
- Xilinx ISE (WebPack is sufficient; development uses ISE 14.7)

Installation of Migen and MiSoC differs depending on what packaging system is used (or if one is used at all). Migen and MiSoC can be installed using `pip`:

```
$ pip install -e git://github.com/m-labs/migen.git#egg=migen
$ pip install -e git://github.com/m-labs/misoc.git#egg=misoc
```

M-Labs also provides conda packages for Migen and MiSoC under the `main` and `dev` labels. Then to build the gateware:

```
$ python make.py
```

The HTML documentation can be built with:

```
$ pip install -r doc/requirements.txt
$ make -C doc html
```

Programming

Once the device has been programmed with the gateware and powered up, it can be used to generate waveforms.

See the `host.pdq.Pdq` class for how to access a stack of PDQ board programmatically, how to submit commands, and how prepare, serialize, and program segments, frames, and channels.

An example how `host.pdq.Pdq` can be used is the command line test interface to the PDQ in *`host.cli.main()`*.

Individual commands are described in the manual in *USB Protocol*.

The wavesynth format is described with examples in *Wavesynth Format*.

To communicate with the device, run the testbenches and generate the data, the following additional packages are required:

- `pyserial`
- `scipy`

Testbenches

```
$ python3 -m testbench.escape
$ python3 -m testbench.cli
```

References

Arbitrary waveform generator for quantum information processing with trapped ions; R. Bowler, U. Warring, J. W. Britton, B. C. Sawyer and J. Amini; Rev. Sci. Instrum. 84, 033108 (2013); http://dx.doi.org/10.1063/1.4795552 http://tf.boulder.nist.gov/general/pdf/2668.pdf

Coherent Diabatic Ion Transport and Separation in a Multizone Trap Array; R. Bowler, J. Gaebler, Y. Lin, T. R. Tan, D. Hanneke, J. D. Jost, J. P. Home, D. Leibfried, and D. J. Wineland; Phys. Rev. Lett. 109, 080502; http://dx.doi.org/10.1103/PhysRevLett.109.080502 http://tf.boulder.nist.gov/general/pdf/2624.pdf

Architecture

The PDQ is an interpolating, scalable, high speed arbitrary waveform generator.

- Outputs: 16 bit DACs, +- 10V
- Sample rate and interpolation speed: 50 MHz or 100 MHz online selectable.
- Scalability: Up to three DACs per board. Up to 16 boards stackable to provide 48 channels per USB device. Number of PDQ stacks limited by maximum number of USB devices per computer.
- Default designs with one, two, or three channels.
- Memory: 16/12/12 KiB, 20/20 KiB, or 40 Kib per channel. Compact partitionable data format.
- Interpolation: DC bias B-spline: constant, linear, quadratic, or cubic. Selectable for each spline knot, each channel.
- DDS output per channel: 32 bit frequency, 16 bit phase offset, 48 bit frequency chirp. Cubic spline amplitude modulation, aligned with frequency/phase modulator. DDS output added to DC bias spline.
- Digital outputs: One AUX channel per board, synchronous to spline knots.

- External control, synchronization: One TTL trigger control input to trigger the execution of marked spline knots.
- Frame selection: Eight separate frames each describing a waveform. Selectable in hard real-time using SPI or USB frame select register.
- Programmable over USB or SPI using the same data and message format.
- Communications are tracked using checksums to verify correct data transfers.

Spline Interpolation

Many use cases of analog voltages in physics experiments do not continuously need large bandwidth analog signals yet the signals need to be clean and with very small content of spurious frequencies. Either a large bandwidth at very small duty cycle or a very small bandwidth at longer duty cycles is sufficient. It is therefore prudent to generate, represent, transfer, and store the output waveform data in a compressed format.

The method of compression chosen here is a polynomial basis spline (B-spline). The data consists of a sequence of knots. Each knot is described by a duration Δt and spline coefficients u_n up to order k. If the knot is evaluated starting at time t_0, the output $u(t)$ for $t \in [t_0, t_0 + \Delta t]$ is:

$$u(t) = \sum_{n=0}^{k} \frac{u_n}{n!}(t - t_0)^n = u_0 + u_1(t - t_0) + \frac{u_2}{2}(t - t_0)^2 + \dots$$

A sequence of such knots describes a spline waveform. Such a polynomial segment can be evaluated and evolved very efficiently using only iterative accumulation (recursive addition) without the need for any multiplications and powers that would require scarce resources on programmable logic. From one discrete time i to the next $i + 1$ each accumulators $v_{n,i}$ is incremented by the value of the next higher order accumulator:

$$v_{n,i+1} = v_{n,i} + v_{n+1,i}$$

For a cubic spline the mapping between the accumulators' initial values $v_{n,0}$ and the polynomial derivatives or spline coefficients u_n can be done off-line and ahead of time. The mapping includes corrections due to the finite time step size τ.

$$\begin{aligned}
t_i &= t_0 + i\tau \\
v_{n,i} &= u_n(t_i) \\
v_{0,0} &= u_0 \\
v_{1,0} &= u_1\tau + \frac{u_2\tau^2}{2} + \frac{u_3\tau^3}{6} \\
v_{2,0} &= u_2\tau^2 + u_3\tau^3 \\
v_{3,0} &= u_3\tau^3
\end{aligned}$$

The data for each knot is then described by the integer duration $T = \Delta t/\tau$ and the initial values $v_{n,0}$.

This representation allows both very fast transient high bandwidth waveforms and slow but smooth large duty cycle waveforms to be described efficiently.

CORDIC

Trigonometric functions can also be represented efficiently on programmable logic using only additions, comparisons and bit shifts. See `misoc.cores.cordic` (in the MiSoC package) for a full documentation of the features, capabilities, and constraints.

Features

Each PDQ card contains one FPGA that feeds three DAC channels. Multiple PDQ cards can be combined into a stack. There is one data connection and one set of digital control lines connected to a stack, common to all cards, all FPGAs, and all channels in that stack.

Each channel of the PDQ can generate a waveform $w(t)$ that is the sum of a cubic spline $a(t)$ and a sinusoid modulated in amplitude by a cubic spline $b(t)$ and in phase/frequency by a quadratic spline $c(t)$:

$$w(t) = a(t) + b(t)\cos(c(t))$$

The data is sampled at 50 MHz or 100 MHz and 16 bit resolution. The higher order spline coefficients (including the frequency offset c_1) receive successively wider data words due to their higher dynamic range.

The duration of a spline knot is of the form:

$$\Delta t = 2^E T \tau_0 = T\tau$$

Here, T is a 16 bit unsigned integer and E is a 4 bit unsigned integer. The spline time step is accordingly scaled to $\tau = 2^E\tau_0$ where τ_0 = 20 ns or 10 ns to accommodate the corresponding change in dynamic range of the coefficients. The only exception to the scaling is the frequency offset c_1 which is always unscaled. At 100 MHz sampling rate, this allows for knot durations anywhere between up to 655 µs at 10 ns resolution and up to 43 s at 655 µs resolution. The encoding of the spline coefficients and associated metadata is described in *Line Format*.

The execution of a knot can be delayed until a trigger signal is received. The trigger signal is common to all channels of all cards in a stack.

Each channel can play waveforms from any of eight frames, selected by the frame selection register. All frames of a channel share the same memory. The memory layout is described in *Memory Layout*. Transitions between frames happen at the end of frames. Frames can be aborted at the end of a spline knot by disarming the stack.

Each channel also has one digital output *aux* that can be set or cleared at each knot. Each board can route a logical OR of a masked set of its channels or the SPI MISO signal to the AUX/F5 output.

The waveform data is written into the channel memories over a full speed USB link or the SPI bus. Each channel memory can be accessed individually. Data or status messages can be read back through the SPI bus by enabling SPI MISO to be output on AUX/F5.

The data channel also carries in-band control commands to switch the clock speed between 50 MHz and 100 MHz, reset the device, arm or disarm the device, enable or disable soft triggering, and enable or disable the starting of new frames. The USB protocol is described in *USB Protocol*.

The host side software receives waveform data in an easy-to generate, portable, and human readable format that is then encoded and written to the channels attached to a device. This wavesynth format is described in *Wavesynth Format*.

Reference Manual

USB Protocol

The data connection to a PDQ stack is a single, full speed USB, parallel FIFO with byte granularity. On the host this appears as a "character device" or "serial port". Windows users may need to install the FTDI device drivers available at the FTDI web site and enable "Virtual COM port (VCP) emulation" so the device becomes available as a COM port. Under Linux the drivers are usually already shipped with the distribution and immediately available. Device permissions have to be handled as usual through group membership and udev rules. The USB bus topology or the device serial number can be used to uniquely identify and access a given PDQ stack. The serial number is stored in the FTDI FT245R USB FIFO chip and can be set as described in the old PDQ documentation. The byte order is little-endian (least significant byte first).

Control Messages

The communication to the device is one-way, write-only. Synchronization has to be achieved by properly sequencing the setting of digital lines with control commands, control commands, and memory writes on the USB bus.

Control commands apply to all channels on all boards in a stack.

Control commands on the USB bus are single bytes prefixed by the `0xa5` escape sequence (`0xa5 0xYY`). If the byte `0xa5` is to be part of the (non-control) data stream it has to be escaped by `0xa5` itself.

Name	Command	Description
RESET	`0x00`	Reset the FPGA registers. Does not reset memories. Does not reload the bitstream. Does not reset the USB interface.
TRIGGER	`0x02`	Soft trigger. Logical OR with the external trigger control line to form the trigger signal to the spline.
ARM	`0x04`	Enable triggering. Disarming also aborts parsing of a frame and forces the parser to the frame jump table. A currently active line will finish execution.
DCM	`0x06`	Set the clock speed. Enabling chooses the Digital Clock Manager which doubles the clock and thus operates all FPGA logic and the DACs at 100 MHz. Disabling chooses a 50 MHz sampling and logic clock. The PDQ logic is inherently agnostic to the value of the sample clock. Scaling of coefficients and duration values must be performed on the host.
START	`0x08`	Enable starting new frames (enables leaving the frame jump table).

The LSB of the command byte then determines whether the command is a "disable" or an "enable" command.

Examples:

- `0xa5 0x02` is `TRIGGER` enable,
- `0xa5 0x03` is `TRIGGER` disable,
- `0xa5 0xa5` is a single `0xa5` in the non-control data stream.

Memory writes

The non-control data stream is interpreted as 16 bit values (two bytes little-endian). The stream consists purely of writes of data to memory locations on individual channels. One channel/one memory can be written to at any given time. A memory write has the format (each row is one word of 16 bits):

`channel`
`start_addr`
`end_addr`
`data[0]`
`data[1]`
...
`data[length-1]`

The channel number is a function of the board number (selected on the dial switch on each PDQ board) and the DAC number (0, 1, 2): `channel = (board_addr << 4) | dac_number`. The length of the data written is `length = end_addr - start_addr + 1`.

Warning:

- No length check or address verification is performed.
- Overflowing writes wrap.

- Non-existent or invalid combinations of board address and/or channel number are silently ignored or wrapped.
- If the write format is not adhered to, synchronization is lost and behavior is undefined.
- A valid `RESET` sequence will restore synchronization. To reliably reset under all circumstances, ensure that the reset sequence `0xa5 0x00` is *not* preceded by an (un-escaped) escape character.

Control commands can be inserted at any point in the non-control data stream.

Examples:

- `0x0072 0x0001 0x0003 0x0005 0x0007 0x0008` writes the three words `0x0005 0x0007 0x0008` to the memory address `0x0001` of DAC channel 2 (the last of three) on board 7 (counting from 0).
- `0xa5 0x06 0x0000 0x00a5a5 0x00a5a5 0xa5a5a5a5 0xa5 0x02 0xa5 0x04 0xa5 0x08` enables the clock doubler (100 MHz) on all channels, then writes the single word `0xa5a5` to address `0x00a5` (note the escaping and the endianess) of channel 0 of board 0, enables soft trigger on all channels, arms all channels, and finally starts all channels.

Memory Layout

The three DAC channels on each board have 8192, 8192, 4096 words (16 bit each) capacity (16 KiB, 16 KiB, 8 KiB). Overflowing writes wrap around. The memory is interpreted as consisting of a table of frame start addresses with 8 entries, followed by data. The layout allows partitioning the waveform memory arbitrarily among the frames of a channel. The data for frame `i` is expected to start at `memory[memory[i]]`.

The memory is interpreted as follows (each row is one word of 16 bits):

Address	Data
`0`	`frame[0].addr`
`1`	`frame[1].addr`
...	...
`frame[0].addr`	`frame[0].data[0]`
`frame[0].addr + 1`	`frame[0].data[1]`
...	...
`frame[0].addr + N`	`frame[0].data[N]`
...	...
`frame[1].addr`	`frame[1].data[0]`
`frame[1].addr + 1`	`frame[1].data[1]`
...	...
`frame[1].addr + L`	`frame[1].data[L]`
...	...

Warning:

- The memory layout is not enforced or verified.
- If violated, the behavior is undefined.
- Jumping to undefined addresses leads to undefined behavior.
- Jumping to frame numbers that have invalid addresses written into their address location leads to undefined behavior.

Note: This layout can be exploited to rapidly swap frame data between multiple different waveforms (without having to re-upload any data) by only updating the corresponding frame address(es).

Line Format

The frame data consists of a concatenation of lines. Each line has the following format (a row being a word of 16 bits):

`header`
`duration`
`data[0]`
...
`data[length - 2]`

Warning:

- If reading and parsing the next line (including potentially jumping into and out of the frame address table) takes longer than the duration of the current line, the pipeline is stalled and the evolution of the splines is paused until the next line becomes available.
- `duration` must be positive.

Header

The 16 bits of the `header` are mapped:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
`wait`	`clear`	`end`	`shift`				`aux`	`silence`	`trigger`	`typ`		`length`			

The components of the `header` have the following meaning:

- `length`: The length of the line in 16 bit words including the duration but excluding the header.
- `typ`: The output processor that the data is fed into. `typ == 0` for the DC spline $a(t)$, `typ == 1` for the DDS amplitude $b(t)$ and phase/frequency $b(t)$ splines.
- `trigger`: Wait for trigger assertion before executing this line. The trigger signal is level sensitive. It is the logical OR of the external trigger input and the soft TRIGGER.
- `silence`: Disable the DAC sample and synchronization clocks during this line. This lowers the amount of clock feed-through and potentially the noise on the output.
- `aux`: Assert the digital auxiliary output during this line. The board's AUX output is the logical OR of all channel `aux` values.
- `shift`: Exponent of the line duration (see *Features*). The actual duration of a line is then `duration * 2**shift`.
- `end`: Return to the frame address jump table after parsing this line.
- `clear`: Clear the CORDIC phase accumulator upon executing this line. If set, the first phase value output will be exactly the phase offset. Otherwise, the phase output is the current phase plus the difference in phase offsets between this line and the previous line.
- `wait`: Wait for trigger assertion before executing the next line.

Warning:

- Parsing a line is unaffected by it carrying `trigger`. Only the start of the execution of a line is affected by it carrying `trigger`.
- Parsing the next line is unaffected by the preceding line carrying `wait`. Only the start of the execution of the next line is affected by the current line carrying `wait`.

Spline Data

The interpretation of the sequence of up to 14 `data` words contained in each line depends on the `typ` of spline interpolator targeted by `header.typ`.

The `data` is always zero-padded to 14 words.

The assignment of the spline coefficients to the data words is as follows:

typ	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	a0	a1		a2			a3								
1	b0	b1		b2			b3			c0	c1		c2		

If the `length` of a line is shorter than 14 words, the remaining coefficients (or parts of coefficients) are set to zero.

The coefficients can be interpreted as two's complement signed integers or as unsigned integers depending depending on preference and convenience. The word order is the same as the byte order of the USB protocol: little-endian (least significant word first).

The scaling of the coefficients is as follows:

- `a0` is in units of `full_scale/(1 << 16)`.
- `a1` is in units of `full_scale/(1 << (32 + shift))/clock_period`.
- `a2` is in units of `full_scale/(1 << (48 + 2*shift))/clock_period**2`.
- `a3` is in units of `full_scale/(1 << (48 + 3*shift))/clock_period**3`.
- `b0` is in units of `full_scale*cordic_gain/(1 << 16)`.
- `b1` is in units of `full_scale*cordic_gain/(1 << (32 + shift))/clock_period`.
- `b2` is in units of `full_scale*cordic_gain/(1 << (48 + 2*shift))/clock_period**2`.
- `b3` is in units of `full_scale*cordic_gain/(1 << (48 + 3*shift))/clock_period**3`.
- `c0` is in units of `2*pi/(1 << 16)`.
- `c1` is in units of `2*pi/(1 << 32)/clock_period`.
- `c2` is in units of `2*pi/(1 << (48 + shift))/clock_period**2`.
- `full_scale` is 20 V.
- The step size `full_scale/(1 << 16)` is 305 µV.
- `clock_period` is 10 ns or 20 ns depending on the `DCM` setting.
- `shift` is `header.shift`.
- `2*pi` is one full phase turn.
- `cordic_gain` is 1.64676 (see `gateware.cordic`).

Note: With the default analog frontend, this means: `a0 == 0` corresponds to close to 0 V output, `a0 == 0x7fff` corresponds to close to 10V output, and `a0 == 0x8000` corresponds to close to -10 V output.

Note: There is no correction for DAC or amplifier offsets, reference errors, or DAC scale errors.

Note: Latencies of the CORDIC path, the DC spline path, and the AUX path are not matched. The CORDIC path (both the amplitude and the phase spline) has about 19 clock cycles more latency than the DC spline path. This can be exploited to align the DC spline knot start and the CORDIC output change. DC spline path and AUX path differe by the DAC latency.

Warning:

- There is no clipping or saturation.
- When accumulators overflow, they wrap.
- That's desired for the phase accumulator but will lead to jumps in the DC spline and CORDIC amplitude.
- When the CORDIC amplitude `b0` reaches an absolute value of `(1 << 15)/cordic_gain`, the CORDIC output becomes undefined.
- When the sum of the CORDIC output amplitude and the DC spline overflows, the output wraps.

Note: All splines (except the DDS phase) continue evolving even when a line of a different `typ` is being executed. All splines (except the DDS phase) stop evolving when the current line has reached its duration and no next line has been read yet or the machinery is waiting for TRIGGER, ARM, or START.

Note: The phase input to the CORDIC the sum of the phase offset `c0` and the accumulated phase due to `c1` and `c2`. The phase accumulator *always* accumulates at full clock speed, not at the clock speed reduced by `shift != 0`. It also never stops or pauses. This is in intentional contrast to the amplitude, DC spline, and frequency evolution that takes place at the reduced clock speed if `shift != 0` and may be paused.

Wavesynth Format

To describe a complete PDQ stack program, the Wavesynth format has been defined.

- `program` is a sequence of `frames`.
- `frame` is a concatentation of `segments`. Its index in the program determines its frame number.
- `segment` is a sequence is `lines`. The first `line` should be `triggered` to establish synchronization with external hardware.
- `line` is a dictionary containing the following fields:
  - `duration`: Integer duration in spline evolution steps, in units of `dac_divider*clock_period`.
  - `dac_divider == 2**header.shift`

  - `trigger`: Whether to wait for trigger assertion to execute this line.
  - `channel_data`: Sequence of `spline`, one for each channel.
- `spline` is a dictionary containing as key a single spline to be set: either `bias` or `dds` and as its value a dictionary of `spline_data`. `spline` has exactly one key.
- `spline_data` is a dictionary that may contain the following keys:
  - `amplitude`: The uncompensated polynomial spline amplitude coefficients. Units are Volts and increasing powers of `1/(dac_divider*clock_period)` respectively.
  - `phase`: Phase/Frequency spline coefficients. Only valid if the key for `spline_data` was `dds`. Units are `[turns, turns/clock_period, turns/clock_period**2/dac_divider]`.
  - `clear`: `header.clear`.
  - `silence`: `header.silence`.

Note:

- `amplitude` and `phase` spline coefficients can be truncated. Lower order splines are then executed.

Example Wavesynth Program

The following example wavesynth program configures a PDQ stack with a single board, three DAC channels.

It configures a single frame (the first and only) consisting of a single triggered segment with three lines. The total frame duration is 80 cycles. The following waveforms are emitted on the three channels:

- A quadratic smooth pulse in bias amplitude from 0 to 0.8 V and back to zero.
- A cubic smooth step from 1 V to 0.5 V, followed by 40 cycles of constant 0.5 V and then another cubic step down to 0 V.
- A sequence of amplitude shaped pulses with varying phase, frequency, and chirp.

```
wavesynth_program = [
    [
        {
            "trigger": True,
            "duration": 20,
            "channel_data": [
                {"bias": {"amplitude": [0, 0, 2e-3]}},
                {"bias": {"amplitude": [1, 0, -7.5e-3, 7.5e-4]}},
                {"dds": {
                    "amplitude": [0, 0, 4e-3, 0],
                    "phase": [.25, .025],
                }},
            ],
        },
        {
            "duration": 40,
            "channel_data": [
                {"bias": {"amplitude": [.4, .04, -2e-3]}},
                {"bias": {
                    "amplitude": [.5],
                    "silence": True,
                }},
                {"dds": {
```

```
                    "amplitude": [.8, .08, -4e-3, 0],
                    "phase": [.25, .025, .02/40],
                    "clear": True,
                }},
            ],
        },
        {
            "duration": 20,
            "channel_data": [
                {"bias": {"amplitude": [.4, -.04, 2e-3]}},
                {"bias": {"amplitude": [.5, 0, -7.5e-3, 7.5e-4]}},
                {"dds": {
                    "amplitude": [.8, -.08, 4e-3, 0],
                    "phase": [-.25],
                }},
            ],
        },
    ]
]
```

The following figure compares the output of the three channels as simulated by the `artiq.wavesynth.compute_samples.Synthesizer` test tool with the output from a full simulation of the PDQ gateware including the host side code, control commands, memory writing, memory parsing, triggering and spline evaluation.

Code Documentation

`host.cli` module

PDQ frontend. Evaluates times and voltages, interpolates and uploads them.

```
usage: pdq [-h] [-s SERIAL] [-c CHANNEL] [-f FRAME] [-t TIMES] [-v VOLTAGES]
           [-o ORDER] [-a] [-k AUX_DAC] [-u DUMP] [-r] [-m] [-n] [-e] [-d]
```

Named Arguments

-s, --serial	device url [”hwgrep://”]
	Default: “hwgrep://”
-c, --channel	channel: 3*board_num+dac_num [0]
	Default: 0
-f, --frame	frame [0]
	Default: 0
-t, --times	sample times (s) [”np.arange(5)*1e-6”]
	Default: “np.arange(5)*1e-6”
-v, --voltages	sample voltages (V) [”(1-np.cos(t/t[-1]*2*np.pi))/2”]
	Default: “(1-np.cos(t/t[-1]*2*np.pi))/2”
-o, --order	interpolation (0: const, 1: lin, 2: quad, 3: cubic) [3]
	Default: 3

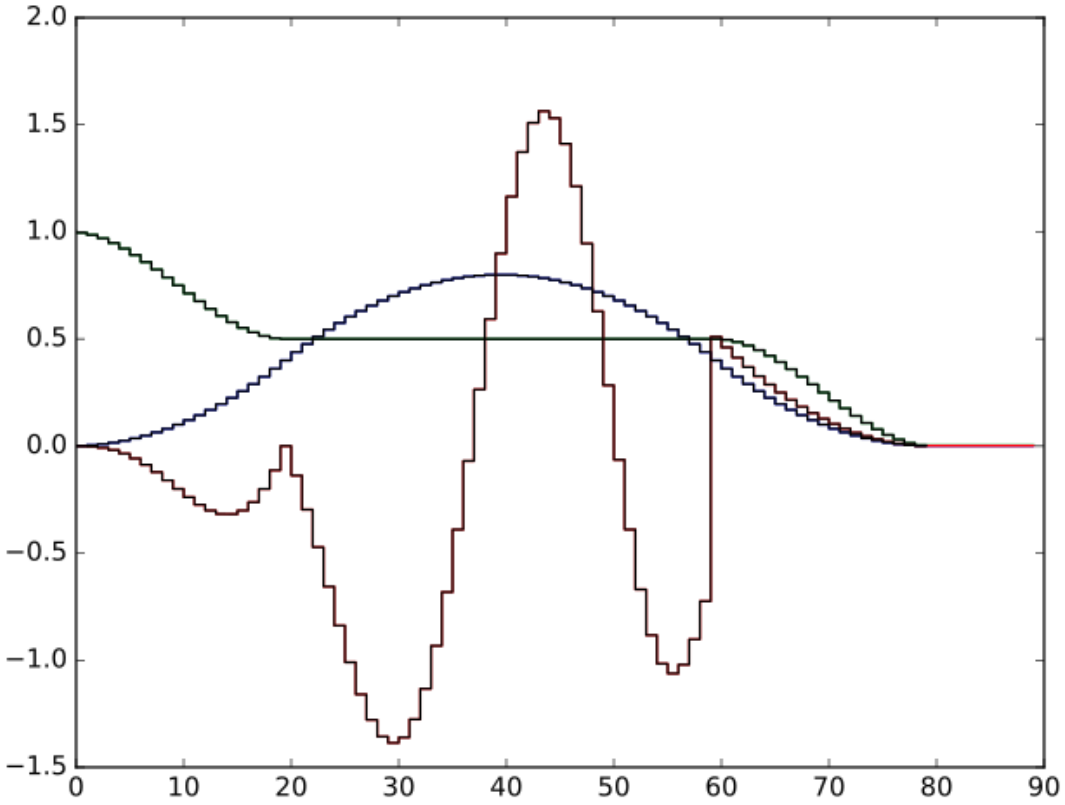

Fig. 1.1: PDQ and `Synthesizer` outputs for wavesynth test program.
The abcissa is the time in clock cycles, the ordinate is the output voltage of the channel.
The plot consists of six curves, three colored ones from the gateware simulation of the board and three black ones from the `Synthesizer` verification tool. The colored curves should be masked by the black curves up to integer rounding errors.
The source of this unittest is part of ARTIQ at `artiq.test.test_pdq.TestPdq.test_run_plot`.

-a, --aux-miso	route MISO to AUX/F5 TTL output [False]
	Default: False
-k, --aux-dac	DAC channel OR mask to AUX/F5 TTL output [0x7]
	Default: 7
-u, --dump	dump to file [None]
-r, --reset	do reset before
	Default: False
-m, --multiplier	100MHz clock [False]
	Default: False
-n, --disarm	disarm group [False]
	Default: False
-e, --free	software trigger [False]
	Default: False
-d, --debug	debug communications
	Default: False

host.cli.**main**(*dev=None*)
Test a PDQ stack.

Parse command line arguments, configures PDQ stack, interpolate the time/voltage data using a spline, generate a wavesynth program from the data and upload it to the specified channel. Then perform the desired arming/triggering/starting functions on the stack.

host.pdq module

class host.pdq.**CRC**(*poly*, *data_width=8*)
Generic and simple table driven CRC calculator.

This implementation is:

- MSB first data
- "un-reversed" full polynomial (i.e. starts with 0x1)
- no initial complement
- no final complement

Handle any variation on those details outside this class.

```
>>> r = CRC(0x1814141AB)(b"123456789")  # crc-32q
>>> assert r == 0x3010BF7F, hex(r)
```

class host.pdq.**Channel**(*max_data*, *num_frames*)
PDQ Channel.

num_frames
int – Number of frames supported.

max_data
int – Number of 16 bit data words per channel.

segments
list[Segment] – Segments added to this channel.

clear()
Remove all segments.

new_segment()
Create and attach a new *Segment* to this channel.

Returns *Segment*

place()
Place segments contiguously.

Assign segment start addresses and determine length of data.

Returns Amount of memory in use on this channel.

Return type *addr* (int)

serialize(*entry=None*)
Serialize the memory for this channel.

Places the segments contiguously in memory after the frame table. Allocates and assigns segment and frame table addresses. Serializes segment data and prepends frame address table.

Parameters **entry** (*list[Segment]*) – See *table()*.

Returns Channel memory data.

Return type *data* (bytes)

table(*entry=None*)
Generate the frame address table.

Unused frame indices are assigned the zero address in the frame address table. This will cause the memory parser to remain in the frame address table until another frame is selected.

The frame entry segments can be any segments in the channel.

Parameters **entry** (*list[Segment]*) – List of initial segments for each frame. If not specified, the first *num_frames* segments are used as frame entry points.

Returns Frame address table.

Return type *table* (bytes)

class host.pdq.**PdqBase**(*num_boards=3*, *num_dacs=3*, *num_frames=32*)
PDQ stack.

checksum
int – Running checksum of data written.

num_channels
int – Number of channels in this stack.

num_boards
int – Number of boards in this stack.

num_dacs
int – Number of DAC outputs per board.

num_frames
int – Number of frames supported.

channels
list[Channel] – List of *Channel* in this stack.

disable(***kwargs*)
Disable the device.

enable(***kwargs*)
Enable the device.

ping()
Ping method returning True. Required for ARTIQ remote controller.

program(*program*, *channels=None*)
Serialize a wavesynth program and write it to the channels in the stack.

The *Channel* targeted are cleared and each frame in the wavesynth program is appended to a fresh set of *Segment* of the channels. All segments are allocated, the frame address tale is generated, the channels are serialized and their memories are written.

Short single-cycle lines are prepended and appended to each frame to allow proper write interlocking and to assure that the memory reader can be reliably parked in the frame address table. The first line of each frame is mandatorily triggered.

Parameters

- **program** (*list*) – Wavesynth program to serialize.
- **channels** (*list[int]*) – Channel indices to use. If unspecified, all channels are used.

program_segments(*segments*, *data*)
Append the wavesynth lines to the given segments.

Parameters

- **segments** (*list[Segment]*) – List of *Segment* to append the lines to.
- **data** (*list*) – List of wavesynth lines.

set_checksum(*crc=0*, *board=15*)
Set/reset the checksum register.

Parameters

- **crc** (*int*) – Checksum value to write.
- **board** (*int*) – Board to write to (0-0xe), 0xf for all boards.

set_config(*reset=False*, *clk2x=False*, *enable=True*, *trigger=False*, *aux_miso=False*, *aux_dac=7*, *board=15*)
Set the configuration register.

Parameters

- **reset** (*bool*) – Reset the board. Memory is not reset. Self-clearing.
- **clk2x** (*bool*) – Enable the clock multiplier (100 MHz instead of 50 MHz)
- **enable** (*bool*) – Enable the channel data parsers and spline interpolators.
- **trigger** (*bool*) – Soft trigger. Logical or with the hardware trigger.
- **aux_miso** (*bool*) – Drive SPI MISO on the AUX/F5 ttl port of each board. If *False*, drive the masked logical or of the DAC channels' aux data.
- **aux_dac** (*int*) – Mask for AUX/F5. Each bit represents one channel. AUX/F5 is: *aux_miso ? spi_miso : (aux_dac & Cat(_.aux for _ in channels) != 0)*

- **board** (*int*) – Board to write to (0-0xe), 0xf for all boards.

set_frame(*frame*, *board=15*)
Set the current frame.

Parameters

- **frame** (*int*) – Frame to select.
- **board** (*int*) – Board to write to (0-0xe), 0xf for all boards.

write_mem(*channel*, *data*, *start_addr=0*)
Write to channel memory.

Parameters

- **channel** (*int*) – Channel index to write to. Assumes every board in the stack has *num_dacs* DAC outputs.
- **data** (*bytes*) – Data to write to memory.
- **start_addr** (*int*) – Start address to write data to.

write_reg(*board*, *adr*, *data*)
Write to a configuration register.

Parameters

- **board** (*int*) – Board to write to (0-0xe), 0xf for all boards.
- **adr** (*int*) – Register address to write to (0-3).
- **data** (*int*) – Data to write (1 byte)

class host.pdq.**Segment**
Serialize the lines for a single Segment.

max_time
int – Maximum duration of a line.

max_val
int – Maximum absolute value (scale) of the DAC output.

max_out
float – Output voltage at *max_val*. In Volt.

out_scale
float – Steps per Volt.

cordic_gain
float – CORDIC amplitude gain.

addr
int – Address assigned to this segment.

data
bytes – Serialized segment data.

bias(*amplitude=[]*, ***kwargs*)
Append a bias line to this segment.

Parameters

- **amplitude** (*list[float]*) – Amplitude coefficients in in Volts and increasing powers of `1/(2**shift*clock_period)`. Discrete time compensation will be applied.
- ****kwargs** – Passed to *line()*.

dds(*amplitude=[]*, *phase=[]*, ***kwargs*)
Append a DDS line to this segment.

Parameters

- **amplitude** (*list[float]*) – Amplitude coefficients in in Volts and increasing powers of `1/(2**shift*clock_period)`. Discrete time compensation and CORDIC gain compensation will be applied by this method.
- **phase** (*list[float]*) – Phase/frequency/chirp coefficients. `phase[0]` in `turns`, `phase[1]` in `turns/clock_period`, `phase[2]` in `turns/(clock_period**2*2**shift)`.
- ****kwargs** – Passed to *line()*.

line(*typ*, *duration*, *data*, *trigger=False*, *silence=False*, *aux=False*, *shift=0*, *jump=False*, *clear=False*, *wait=False*)
Append a line to this segment.

Parameters

- **typ** (*int*) – Output module to target with this line.
- **duration** (*int*) – Duration of the line in units of `clock_period*2**shift`.
- **data** (*bytes*) – Opaque data for the output module.
- **trigger** (*bool*) – Wait for trigger assertion before executing this line.
- **silence** (*bool*) – Disable DAC clocks for the duration of this line.
- **aux** (*bool*) – Assert the AUX (F5 TTL) output during this line. The corresponding global AUX routing setting determines which channels control AUX.
- **shift** (*int*) – Duration and spline evolution exponent.
- **jump** (*bool*) – Return to the frame address table after this line.
- **clear** (*bool*) – Clear the DDS phase accumulator when starting to exectute this line.
- **wait** (*bool*) – Wait for trigger assertion before executing the next line.

static **pack**(*widths*, *values*)
Pack spline data.

Parameters

- **widths** (*list[int]*) – Widths of values in multiples of 16 bits.
- **values** (*list[int]*) – Values to pack.

Returns Packed data.

Return type *data* (bytes)

`host.pdq.`**discrete_compensate**(*c*)
Compensate spline coefficients for discrete accumulators.

Given continuous-time b-spline coefficients, this function compensates for the effect of discrete time steps in the target devices.

The compensation is performed in-place.

gateware.pdq module

gateware.comm module

gateware.dac module

gateware.platform module

class gateware.platform.**Platform**
PDQ Platform.

- Xilinx Spartan 3A 500E in a PQ208 package.
- 50 MHz single ended input clock.
- Single FT245R USB parallel FIFO.
- Three 16 bit LVDS DACs.
- Several TTL control lines.

gateware.escape module

gateware.ft245r module

gateware.spi module

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

g

gateware.platform, 18

h

host.cli, 13
host.pdq, 13

Index

A

addr (host.pdq.Segment attribute), 16

B

bias() (host.pdq.Segment method), 16

C

Channel (class in host.pdq), 13
channels (host.pdq.PdqBase attribute), 14
checksum (host.pdq.PdqBase attribute), 14
clear() (host.pdq.Channel method), 14
cordic_gain (host.pdq.Segment attribute), 16
CRC (class in host.pdq), 13

D

data (host.pdq.Segment attribute), 16
dds() (host.pdq.Segment method), 16
disable() (host.pdq.PdqBase method), 15
discrete_compensate() (in module host.pdq), 17

E

enable() (host.pdq.PdqBase method), 15

G

gateware.platform (module), 18

H

host.cli (module), 13
host.pdq (module), 13

L

line() (host.pdq.Segment method), 17

M

main() (in module host.cli), 13
max_data (host.pdq.Channel attribute), 13
max_out (host.pdq.Segment attribute), 16
max_time (host.pdq.Segment attribute), 16
max_val (host.pdq.Segment attribute), 16

N

new_segment() (host.pdq.Channel method), 14
num_boards (host.pdq.PdqBase attribute), 14
num_channels (host.pdq.PdqBase attribute), 14
num_dacs (host.pdq.PdqBase attribute), 14
num_frames (host.pdq.Channel attribute), 13
num_frames (host.pdq.PdqBase attribute), 14

O

out_scale (host.pdq.Segment attribute), 16

P

pack() (host.pdq.Segment static method), 17
PdqBase (class in host.pdq), 14
ping() (host.pdq.PdqBase method), 15
place() (host.pdq.Channel method), 14
Platform (class in gateware.platform), 18
program() (host.pdq.PdqBase method), 15
program_segments() (host.pdq.PdqBase method), 15

S

Segment (class in host.pdq), 16
segments (host.pdq.Channel attribute), 13
serialize() (host.pdq.Channel method), 14
set_checksum() (host.pdq.PdqBase method), 15
set_config() (host.pdq.PdqBase method), 15
set_frame() (host.pdq.PdqBase method), 16

T

table() (host.pdq.Channel method), 14

W

write_mem() (host.pdq.PdqBase method), 16
write_reg() (host.pdq.PdqBase method), 16