pdq Documentation

Release 3.0.dev

Robert Jördens

Feb 27, 2018

Contents

CHAPTER 1

Contents

1.1 PDQ Overview

A pretty darn quick interpolating arbitrary waveform generator.

1.1.1 Build

Requirements:

- Migen (https://github.com/m-labs/migen)
- MiSoC (https://github.com/m-labs/misoc)
- Xilinx ISE (a WebPack license 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 currently depends on Python 3.5 (see https://github.com/m-labs/migen/issues/62). 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/asyncserial#egg=asyncserial
$ 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. Instead of using `pip`, the conda packages can be installed:

```
$ conda install -c m-labs/label/dev migen misoc
```

Install PDQ using pip:

```
$ pip install -e .
```

Then to build the gateware:

```
$ pdq_make -c 3
```

where `-c 3` determines the number of channels (see also `pdq_make -h` for help).

The HTML documentation can be built with:

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

1.1.2 Flashing

For each PDQ board, power it with 5V (only the 5V digital power is required). Connect a JTAG adapter (e.g. Xilinx Platform Cable USB II) to the board. Use Xilinx Impact to write the flash to the board. Some example scripts are included in the `misc` directory.

1.1.3 Programming

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

See the *pdq.host.usb.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 *pdq.host.usb.PDQ* can be used is the command line test interface to the PDQ in *pdq.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`

1.1.4 Testbenches

There are multiple testbenches for the individual gateware and software components. Some gateware tests are included in the respective sources in `pdq/gateware/`, others are in `pdq/test/` or have been moved into more suitable places like the `migen`, `misoc`, or `artiq` packages. `nose` can be used to conveniently run the included tests:

```
$ nosetests -v
```

1.1.5 Examples

Some initial example usage of the PDQ code is located in this repository in `examples/`.

1.1.6 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

1.2 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/SPI device. Number of PDQ stacks limited by maximum number of USB/SPI 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.

1.2.1 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.

1.2.2 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.

1.2.3 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 32 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*.

1.3 Reference Manual

1.3.1 Protocol

A PDQ stack provides two different channels for data communication apart from the hardware trigger signal. Both SPI and USB can be used to configure the device, write registers and write to memory. The SPI bus provides a read-back mechanism to verify correct communication and read out status. The USB bus is read-only.

Note: Both SPI and USB are active at the same time. They can both be used to access the device. But care should be taken not to use both methods at the same time. In that case SPI has precedence and will interrupt and corrupt any ongoing USB transfers.

1.3.2 Messages

Each communication with the PDQ over SPI or USB forms a message. Each message starts with a one-byte header determining the address of the board to access, the address of the register or memory to access and the action to perform.

Name	Length (Bits)	Description
`adr`	2	Channel memory or register address
`is_mem`	1	Flag signaling a channel memory access
`board`	4	Board address (the selector switch on the PDQ board). `0xf == 15` signaling the broadcast address to access all boards.
`we`	1	Write-enable. Access is a (register or channel memory) write.

For example, `0b0_1111_0_00` signals a read from register 0 on any board. Since data reads can only be performed over SPI and since only one board can drive the MISO line this will read register 0 from the master board.

As another example, `0b1_0011_1_01` signals a write to the second channel memory of board number 3.

The data following the header byte then depends on the action performed. The following table defines the data format both to (MOSI/USB) and from the device:

Target	Access	MOSI/USB	MISO
Register	read	`HEAD dummy dummy`	`dummy dummy DATA`
Register	write	`HEAD DATA`	`dummy ....`
Memory	read	`HEAD ADDR_LO ADDR_HI dummy dummy ....`	`dummy dummy dummy DATA_LO DATA_HI ....`
Memory	write	`HEAD ADDR_LO ADDR_HI DATA0_LO DATA0_HI DATA1_LO DATA1_HI ....`	`dummy ....`

Registers

Name	Register address (`adr`)	Description
`config`	0	Configuration register
`crc`	1	Data checksum register
`frame`	2	Frame selection register

Configuration

The configuration register is used to reset the device, configure its clock source, enable and disable it, perform a soft trigger over USB or SPI and to configure the behavior of the AUX/F5 TTL.

Name	Length (bits)	Description
`reset`	1	Reset the boards. Self-clearing. Reset the FPGA registers. Does not reset memories. Does not reload the bitstream. Does not reset the USB interface.
`clk2x`	1	Choose 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.
`enable`	1	Enable the channel data parsers and spline interpolators. Disabling also aborts parsing of a frame and forces the parser to the frame jump table. Any currently active line will also be aborted.
`trigger`	1	Soft trigger. Logical or with the hardware trigger.
`aux_miso`	1	If set, drive the SPI MISO data on the AUX/F5 TTL port of each bord. If cleared, drive the masked logical OR of the DAC channels' aux data on AUX/F5.
`aux_dac`	8	Mask for AUX/F5. Each bit represents one channel.

AUX/F5 is therefore: `aux_f5 = aux_miso ? spi_miso : (aux_dac & Cat(_.aux for _ in channels) != 0)`.

Examples of messages (register writes with header and data):

- `0b1_1111_0_00 0b000_0_0_0_0_1` resets all boards.
- `0b1_0000_0_00 0b000_1_0_1_1_0` enables board 0, 100 MHz clock, and MISO on AUX/F5.
- The sequence of two configuration register writes `0b1_1111_0_00 0b000_1_1_1_1_0` and `0b1_1111_0_00 0b000_1_0_1_1_0` performs a short trigger over SPI.

Checksum

When receiving message bytes (USB framing and escape bytes are ignored; see below *USB Protocol*) from either SPI/MOSI or USB, the checksum register is updated with a new value. This can be used to ensure and verify correct data transfer by computing the checksum on the sending end and then reading it back and comparing.

The checksum algorithm used is a 8-bit cyclic redundancy check (CRC) with a polynomial of `0x07`. This polynomial is also commonly known as CRC-CCITT and implemented both in gateware on the PDQ and in the host side code. Given some example input it behavoes as follows:

```
crc8([1,2,3,4,5,6,7,8,9]) == 0x85
```

The checksum register can be set to initialize it with a known value and read to obtain the current value.

Examples:

- `0b1_1111_0_01 0x00` clears the checksum register on all boards.
- `0b0_1111_0_01 0x00 0x00` reads the checksum register on the board connected to MISO.

Frame

The frame selection register determines the currently executed frame for all channels on the addressed board(s). There are currently 32 frames (5 bits) supported. The unused bits are ignored (wrap around on the value) when written and zero when read.

Examples:

- `0b1_1111_0_10 0x13` selects frame 0x13 on all connected boards.

Memory access

The payload data of the message is interpreted as a 16 bit memory address (in the channel memory) followed by a sequence of 16 bit values (two bytes little-endian).

Warning:

- No length check or address verification is performed.
- Overflowing the address counter will wrap around to the first address.
- Non-existent or invalid combinations of board address and/or channel number are silently ignored or wrapped.

Examples:

- `0b1_0001_1_10 0x03 0x04 0x05 0x06 0x07 0x08` writes `0x0605 0x0807` to the memory locations including and following address `0x0403` of channel `0b10` on board `0b0001`.

1.3.3 SPI Protocol

The SPI bus provides access to a stack of PDQ boards over four-wire SPI (separate MISO and MOSI lines).

The SPI bus is wired with `CS_N` from the SPI master connected to `F2 IN` on the master PDQ, `CLK` connected to `F3 IN`, `MOSI` connected to `F4 IN` and `MISO` (optionally) connected to `F5 OUT`. `F1 TTL Input Trigger` remains

as waveform trigger input. Due to hardware constraints, there can only be one board connected to the core device's MISO line and therefore there can only be SPI readback from one board at any time.

Messages on the SPI bus are framed using SPI `CS_N`. There can be at most one transaction per SPI `CS_N` cycle. Register writes are performed when the last bit of the data is clocked into the device. Register access messages have fixed length (two bytes for a write and three bytes for a read). Message data after a register access is ignored.

An implementation of the PDQ SPI protocol is the ARTIQ PDQ driver with its documentation.

1.3.4 USB Protocol

The USB 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).

Each message on the USB bus is framed by the ASCII STX (`0x02`) and ASCII ETX (`0x03`) control characters. Control characters are escaped using `0xa5`. Since the escape character can also appear inside a message each `0xa5` within the message is also escaped using `0xa5`. A valid message as sent over the USB connection therefore looks like:

```
0xa5 0x02  <escaped-message> 0xa5 0x03
```

where `<escaped message>` has all occurences of `0xa5` replaced by `0xa5 0xa5`.

1.3.5 Memory Layout

Depending on the bitstream configuration the memory is divided up among the channels. For three channels the memories contain (16, 12, 12) KiB, for two channels, they contain (20, 20) KiB and a single channel has all 40 KiB available. 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).

1.3.6 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.

1.3.7 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]}},
```

(continues on next page)

(continued from previous page)

```
                {"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.

1.4 Code Documentation

1.4.1 Hose side code, USB

`pdq.host.crc` module

class pdq.host.crc.**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)

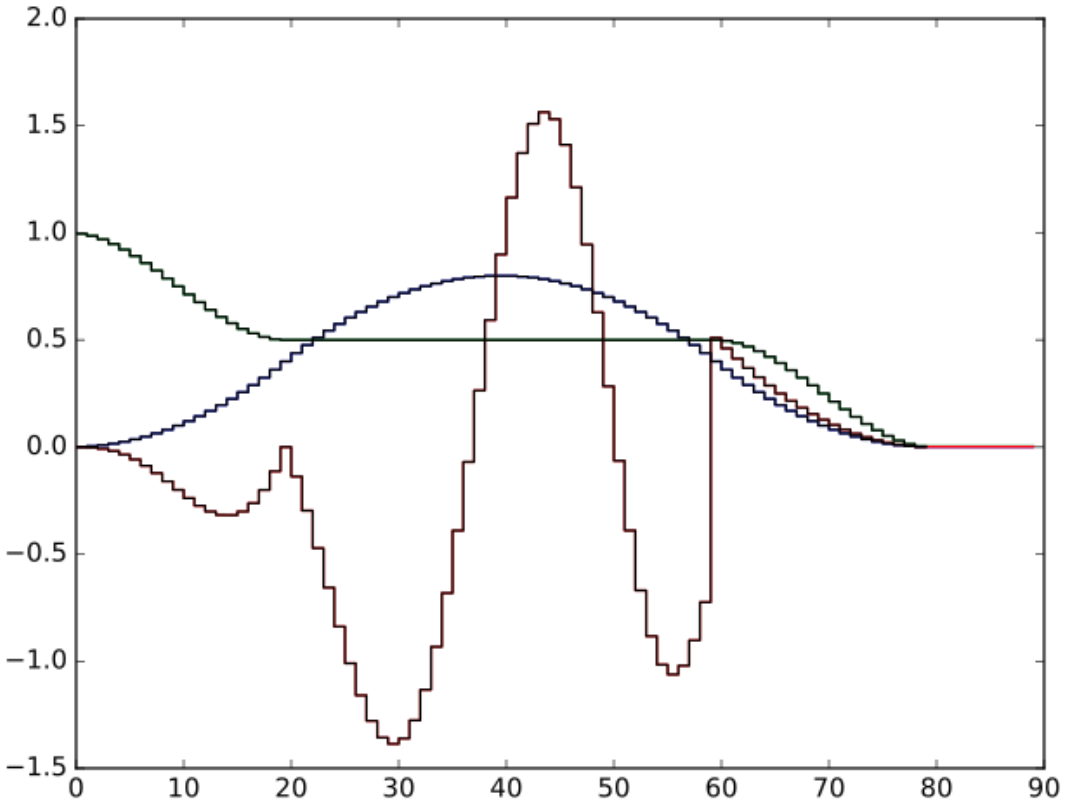

Fig. 1: PDQ and `Synthesizer` outputs for wavesynth test program.
The abcissa is the time in clock cycles, the ordinate is the output voltage of the channel.
The plot consists of six curves, three colored ones from the gateware simulation of the board and three black ones from the `Synthesizer` verification 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`.

- 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)
```

pdq.host.protocol module

class pdq.host.protocol.**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 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 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 bytes

class pdq.host.protocol.**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.

get_config(*board=15*)
Read configuration register.

get_crc(*board=15*)
Read checksum register.

See also:

set_crc()

get_frame(*board=15*)
Read frame selection register.

See also:

set_frame()

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_config(*reset=0*, *clk2x=0*, *enable=1*, *trigger=0*, *aux_miso=0*, *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 reading and execution of waveform data from memory.
- **trigger** (*bool*) – Soft trigger. Logical or with the `F1 TTL Input` hardware trigger.
- **aux_miso** (*bool*) – Drive SPI MISO on the AUX/`F5 OUT` TTL port of each board. If `False`/`0`, drive the masked logical OR of the DAC channels' aux data.
- **aux_dac** (*int*) – DAC channel 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_crc(*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_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.

set_reg(*adr*, *data*, *board*)
Set a register.

Parameters

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

write_mem(*mem*, *adr*, *data*, *board=15*)
Write to channel memory.

Parameters

- **mem** (*int*) – Channel memory to write to.
- **data** (*bytes*) – Data to write to memory.
- **adr** (*int*) – Start address to write data to.

pdq.host.protocol.**PDQ_CMD**(*board*, *is_mem*, *adr*, *we*)
Pack PDQ command fields into command byte.

Parameters

- **board** – Board address, 0 to 15, with `15 = 0xf` denoting broadcast to all boards connected.
- **is_mem** – If `1`, `adr` denote the address of the memory to access (0 to 2). Otherwise `adr` denotes the register to access.
- **adr** – Address of the register or memory to access. (`PDQ_ADR_CONFIG`, `PDQ_ADR_FRAME`, `PDQ_ADR_CRC`).
- **we** – If `1` then write, otherwise read.

class pdq.host.protocol.**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(*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 bytes

pdq.host.protocol.**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.

pdq.host.usb module

class pdq.host.usb.**PDQ**(*url=None*, *dev=None*, ***kwargs*)
Initialize PDQ USB/Parallel device stack.

Note: This device should only be used if the PDQ is intended to be configured using the USB connection and **not** via SPI.

Parameters

- **url** (*str*) – Pyserial device URL. Can be `hwgrep://` style (search for serial number, bus topology, USB VID:PID combination), `COM15` for a Windows COM port number, `/dev/ttyUSB0` for a Linux serial port.
- **dev** (*file-like*) – File handle to use as device. If passed, `url` is ignored.
- ****kwargs** – See `PDQBase` .

close()
Close the USB device handle.

flush()
Flush pending data.

set_reg(*adr*, *data*, *board*)
Set a register.

Parameters

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

write(*data*)
Write data to the PDQ board over USB/parallel.

SOF/EOF control sequences are appended/prepended to the (escaped) data. The running checksum is updated.

Parameters **data** (*bytes*) – Data to write.

write_mem(*mem*, *adr*, *data*, *board=15*)
Write to channel memory.

Parameters

- **mem** (*int*) – Channel memory to write to.
- **data** (*bytes*) – Data to write to memory.
- **adr** (*int*) – Start address to write data to.

pdq.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
-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

pdq.host.cli.**main**(*dev=None*, *args=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.

1.4.2 ARTIQ-facing components

pdq.artiq.spi module

class pdq.artiq.spi.**PDQ**(*dmgr*, *spi_device*, *chip_select=1*, ***kwargs*)
PDQ smart arbitrary waveform generator stack.

Provides access to a stack of PDQ boards connected via SPI using PDQ gateware version 3 or later.

The SPI bus is wired with `CS_N` from the core device connected to `F2 IN` on the master PDQ, `CLK` connected to `F3 IN`, `MOSI` connected to `F4 IN` and `MISO` (optionally) connected to `F5 OUT`. `F1 TTL Input Trigger` remains as waveform trigger input. Due to hardware constraints, there can only be one board connected to the core device's MISO line and therefore there can only be SPI readback from one board at any time.

Parameters

- **spi_device** – Name of the SPI bus this device is on.
- **chip_select** – Value to drive on the chip select lines of the SPI bus during transactions.

get_reg(*adr*, *board*)
Get a PDQ register.

Parameters

- **adr** – Address of the register (`_PDQ_ADR_CONFIG`, `_PDQ_ADR_FRAME`, `_PDQ_ADR_CRC`).
- **board** – Board to access, `0xf` to write to all boards.

Returns Register data (8 bit).

read_mem(*mem*, *adr*, *data*, *board=15*, *buffer=8*)
Read from DAC channel waveform data memory.

Parameters

- **mem** – DAC channel memory to access (0 to 2).
- **adr** – Start address.
- **data** – Memory data. List of 16 bit integers.
- **board** – Board to access (0-15) with `0xf = 15` being broadcast to all boards.

set_reg(*adr*, *data*, *board*)
Set a PDQ register.

Parameters

- **adr** – Address of the register (`_PDQ_ADR_CONFIG`, `_PDQ_ADR_FRAME`, `_PDQ_ADR_CRC`).
- **data** – Register data (8 bit).
- **board** – Board to access, `0xf` to write to all boards.

setup_bus(*write_div=24*, *read_div=64*)
Configure the SPI bus and the SPI transaction parameters for this device. This method has to be called before any other method if the bus has been used to access a different device in the meantime.

This method advances the timeline by the duration of two RTIO-to-Wishbone bus transactions.

Parameters

- **write_div** – Write clock divider.
- **read_div** – Read clock divider.

write_mem(*mem*, *adr*, *data*, *board=15*)
Write to DAC channel waveform data memory.

Parameters

- **mem** – DAC channel memory to access (0 to 2).
- **adr** – Start address.
- **data** – Memory data. List of 16 bit integers.
- **board** – Board to access (0-15) with `0xf = 15` being broadcast to all boards.

pdq.artiq.aqctl_pdq module

PDQ controller.

Use this controller for PDQ stacks that are connected via USB.

```
usage: aqctl_pdq [-h] [-d DEVICE] [--simulation] [--dump DUMP] [-r]
                 [-b BOARDS]
```

Named Arguments

-d, --device	serial port
--simulation	do not open any device but dump data
	Default: False
--dump	file to dump simulation data into
	Default: "pdq_dump.bin"
-r, --reset	reset device [False]
	Default: False
-b, --boards	number of boards [3]
	Default: 3

pdq.artiq.mediator module

exception pdq.artiq.mediator.**ArmError**
Raised when attempting to arm an already armed PDQ, to modify the program of an armed PDQ, or to play a segment on a disarmed PDQ.

exception pdq.artiq.mediator.**FrameActiveError**
Raised when a frame is active and playback of a segment from another frame is attempted.

exception pdq.artiq.mediator.**InvalidatedError**
Raised when attemting to use a frame or segment that has been invalidated (due to disarming the PDQ).

exception pdq.artiq.mediator.**SegmentSequenceError**
Raised when attempting to play back a named segment which is not the next in the sequence.

1.4.3 Gateware

pdq.gateware.pdq module

class pdq.gateware.pdq.**CRG**(*platform*)
PDQ Clock and Reset generator.

Parameters **platform** (Platform) – PDQ Platform.

rst
Signal – Reset input.

dcm_sel
Signal – Select doubled clock. Input.

dcm_locked
Signal – DCM locked. Output.

cd_sys
ClockDomain – System clock domain driven.

cd_sys_n
ClockDomain – Inverted system clock domain driven.

class pdq.gateware.pdq.**Pdq**(**args*, ***kwargs*)
PDQ Top module.

Wires up USB FIFO reader `gateware.ft245r.Ft345r_rx`, clock and reset generator `CRG`, and the DAC output signals. Delegates the wiring of the remaining modules to `PdqBase`.

`pads.g2_out` is assigned the DCM locked signal.

Parameters **platform** (`Platform`) – PDQ platform.

class pdq.gateware.pdq.**PdqBase**(*ctrl_pads*, *mem_depths=(8192*, *8192*, *4096)*)
PDQ Base configuration.

Used both in functional simulation and final gateware.

Holds the three `gateware.dac.Dac` and the communication handler `gateware.comm.Comm`.

Parameters

- **ctrl_pads** (*Record*) – Control pads for `gateware.comm.Comm`.
- **mem_depth** (*list[int]*) – Memory depths for the DAC channels.

dacs
list – List of `gateware.dac.Dac`.

comm
Module – `gateware.comm.Comm`.

pdq.gateware.comm module

class pdq.gateware.comm.**Arbiter**(*width=8*)
Simple arbiter for two framed data streams. Uses end-of-packet (eop) to detect that `sink0` is inactive and yields to `sink1`.

class pdq.gateware.comm.**Comm**(*ctrl_pads*, *dacs*)
USB Protocol handler.

Parameters

- **ctrl_pads** (*Record*) – Control signal pads.
- **dacs** (*list*) – List of `gateware.dac.Dac`.

sink
Endpoint[bus_layout] – 8 bit data sink containing both the control sequencences and the data stream.

Control command handler.

Controls the input and output TTL signals, handles the excaped control commands.

Parameters

- **pads** (*Record*) – Pads containing the TTL input and output control signals
- **dacs** (*list*) – List of `gateware.dac.Dac`.

rg
ResetGen

proto
Protocol

spi
SPISlave

class pdq.gateware.comm.**FTDI2SPI**
Converts parallel data stream from FTDI chip into framed SPI-like data.

It uses the `Unescaper` to to detect escaped start-of-frame SOF and EOF characters.

sink
Endpoint – Raw data from FTDI parallel bus.

source
Endpoint – Framed data stream (eop asserted when there is no active frame).

class pdq.gateware.comm.**Protocol**(*mems*)
Handles the register and memory protocols and reads/writes data in the channel memories.

Parameters **mems** (*list*) – List of memories from `gateware.dac.Dac`.

sink
Endpoint – 8 bit data sink.

source
Endpoint – 8 bit data source for SPI MISO read-back.

board
Signal(4) – Board address.

config
Record – Configuration register.

frame
Signal(max=32) – Selected frame.

class pdq.gateware.comm.**ResetGen**(*n=128*)
Reset generator.

Asserts *reset* for a given number of cycles when triggered.

Parameters **n** (*int*) – number of cycles.

trigger
Signal – Trigger input.

reset
Signal – Reset output. Active high.

pdq.gateware.dac module

class pdq.gateware.dac.**Dac**(*fifo=0*, ***kwargs*)
Output module.

Holds the Memory, the *Parser*, the *Sequencer*, and its two output line executors.

Parameters

- **fifo** (*int*) – Number of lines to buffer between *Parser* and *Sequencer*.
- ****kwargs** – Passed to *Parser*.

parser
The memory *Parser*.

out
The *Sequencer* and output executor. Connect its `data` to the DAC.

class pdq.gateware.dac.**Dds**(*line*, *stb*, *inc*)
DDS spline interpolator.

The line data is interpreted as:

- 16 bit amplitude offset
- 32 bit amplitude first order derivative
- 48 bit amplitude second order derivative
- 48 bit amplitude third order derivative
- 16 bit phase offset
- 32 bit frequency word
- 48 bit chirp

Parameters

- **line** (*Record[line_layout]*) – Next line to be executed. Input.
- **stb** (*Signal*) – Load data from next line. Input.
- **inc** (*Signal*) – Evolve clock enable. Input.

data
Signal[16] – Output data from this spline.

class pdq.gateware.dac.**Parser**(*mem_depth=4096*)
Memory parser.

Reads memory controlled by TTL signals, builds lines, and submits them to its output.

Parameters **mem_depth** (*int*) – Memory depth in 16 bit entries.

mem
Memory – Memory to read from.

source
Endpoint[line_layout] – Endpoint of lines read from memory. Output.

arm
Signal – Allow triggers. If disarmed, the next line will not be read. Instead, the Parser will return to the frame address table. Input.

start
Signal – Allow leaving the frame address table. Input.

frame
Signal[3] – Values of the frame selection lines. Input.

class pdq.gateware.dac.**Sequencer**
Line sequencer.

Controls execution of a line. Owns the executors that evolve the line data and sums them together to generate the output. Also manages the line duration counter, the `2**shift` counter and the acknowledgment of new line data when the previous is finished.

sink
Endpoint[line_layout] – Line data sink.

trigger
Signal – Trigger input.

arm
Signal – Arm input.

aux
Signal – TTL AUX (F5) output.

silence
Signal – Silence DAC clocks output.

data
Signal[16] – Output value to be send to the DAC.

class pdq.gateware.dac.**Volt**(*line*, *stb*, *inc*)
DC bias spline interpolator.

The line data is interpreted as a concatenation of:

- 16 bit amplitude offset
- 32 bit amplitude first order derivative
- 48 bit amplitude second order derivative
- 48 bit amplitude third order derivative

Parameters

- **line** (*Record[line_layout]*) – Next line to be executed. Input.
- **stb** (*Signal*) – Load data from next line. Input.
- **inc** (*Signal*) – Evolve clock enable. Input.

data
Signal[16] – Output data from this spline.

pdq.gateware.platform module

class pdq.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.

pdq.gateware.escape module

class pdq.gateware.escape.**Unescaper**(*layout*, *escape=165*)
Split a data stream into an escaped low bandwidth command stream and an unescaped high bandwidth data stream.

Items in the input stream that are escaped by being prefixed with the escape character, will be directed to the `source_b` output Endpoint.

Items that are not escaped, and the escaped escape character itself are directed at the `source_a` output Endpoint.

Parameters

- **layout** (*layout*) – Stream layout to split.
- **escape** (*int*) – Escape character.

sink
Endpoint[layout] – Input stream.

source0
Endpoint[layout] – High bandwidth unescaped data Endpoint.

source1
Endpoint[layout] – Low bandwidth command Endpoint.

pdq.gateware.ft245r module

class pdq.gateware.ft245r.**Ft245r_rx**(*pads*, *clk=10.0*)
FTDI FT345R synchronous reader.

Parameters

- **pads** (*Record[ft345r_layout]*) – Pads to the FT245R.
- **clk** (*float*) – Clock period in ns.

source
Endpoint[bus_layout] – 8 bit data source. Output.

busy
Signal – Data available but not acknowledged by sink. Output.

class pdq.gateware.ft245r.**SimFt245r_rx**(*data*)

pdq.gateware.spi module

class pdq.gateware.spi.**Debouncer**(*cycles=1*)
Debounce a signal.

The initial change on input is passed through immediately. But further changes are suppressed for *cycles*.

Parameters **cycles** (*int*) – Block furhter level changes for that many cycles after an initial change.

i
Signal – Input, needs a *MultiReg* in front of it if this is an asynchronous signal.

o
Signal – Debounced output.

class pdq.gateware.spi.**SPISlave**(**args*, ***kwargs*)
SPI slave.

- CLK_PHA, CLK_POL = 0,0
- MSB first

Parameters **width** (*int*) – Shift register width in bits.

spi
Record – SPI bus record consisting of *cs_n*, *clk*, *mosi*, *miso*, *oe_s*, and *oe_m*. Use *oe_s* (driven by the slave) to wire up a tristate half-duplex data line. Use *oe_m* on the master side.

data
Endpoint – SPI parallel communication stream.

- ***mosi*: *width* bits read on the mosi line in the previous clock** cycles
- ***miso*: *width* bits to be written on miso line in the next** cycles
- *stb*: data available in mosi and data read from miso.
- ***ack*: in half-duplex mode, drive miso on the combined** miso/mosi data line

cs_n
Signal – use to *s.reset.eq(s.cs_n)* and for framing logic.

class pdq.gateware.spi.**ShiftRegister**(*width*)
Shift register for an SPI slave.

Parameters **width** (*int*) – Register width in bits.

i
Signal – Serial input.

o
Signal – Serial output.

data
Signal(width) – Content of the shift register.

next
Signal(width) – Combinatorial content of the register in the next cycle.

stb
Signal – Strobe signal indicating that *width* bits have been shifted (in and out) and the register value can be swapped.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index