Pulse Streamer 8/2 Documentation

Release 1.1

Swabian Instruments

Aug 13, 2019

CONTENTS

1 Getting Started **1**
- 1.1 Software installation . . . 1
- 1.2 Generate simple pulse pattern . . . 1

2 Hardware **3**
- 2.1 Output Channels . . . 3
- 2.2 Trigger Input . . . 3
- 2.3 External Clock Input . . . 4

3 Network Connection **5**
- 3.1 Assign a static IP with the MAC address and DHCP . . . 5
- 3.2 Permanent static IP: 169.254.8.2 . . . 5
- 3.3 Modify the network settings . . . 5

4 Programming interface **7**
- 4.1 Overview . . . 7
- 4.2 PulseStreamer . . . 12
- 4.3 Sequence . . . 16
- 4.4 OutputState . . . 19

5 Low-level RPC interface **21**
- 5.1 Pulse Sequences . . . 21
- 5.2 Resetting Pulse Streamer to constant outputs . . . 21
- 5.3 Setting constant outputs . . . 22
- 5.4 Running pulse sequences . . . 22
- 5.5 More features . . . 23
- 5.6 Communicating with the instrument . . . 24
- 5.7 JSON-RPC Interface . . . 24
- 5.8 gRPC Interface . . . 24

6 Changelog **27**
- 6.1 2019-08-07 . . . 27
- 6.2 2019-05-10 . . . 27
- 6.3 2019-04-23 . . . 27
- 6.4 2019-03-01 . . . 28
- 6.5 2018-12-17 . . . 29
- 6.6 2018-11-09 . . . 29
- 6.7 2018-10-10 . . . 30
- 6.8 2018-01-05 . . . 31
- 6.9 2017-05-07 . . . 31
- 6.10 2016-04-08 . . . 31

6.11 2016-03-17 . . . 31
6.12 2016-03-07 . . . 31
6.13 2016-03-03 . . . 31
6.14 2016-02-02 . . . 32

7 Previous versions **33**
7.1 Version 1.x . . . 33
7.2 Version 0.x . . . 33

8 Indices and tables **35**

CHAPTER

ONE

GETTING STARTED

1.1 Software installation

The Pulse Streamer does not require any driver installation. It uses standard Ethernet interface for communication, so all drivers are provided with your operating system. The only requirement is that the Pulse Streamer can be added to your network, see the *Network Connection* section for further information. Additionally, the direct connection to the network card of your PC supported as well.

1.1.1 Client software

Please visit the Software Downloads section of our web-site and download the client and example files for any of the supported programming languages.

MATLAB toolbox can be manually installed with MATLAB Add-on Manager by double-clicking the toolbox package file (Windows). The toolbox is compatible with MATLAB versions starting from 2014b.

LabVIEW package installation requires free JKI VI Package Manager which is often installed alongside with LabView.

Python module for Pulse Streamer can be installed either from the www.pypi.org with the following command

```
pip install pulsestreamer
```

or from a local package file using the command

```
pip install path/to/packagefile.whl
```

Replace `path/to/packagefile.whl` with actual path of the package file.

1.2 Generate simple pulse pattern

This section shows simple example on how to generate a simple signal on digital output 0 of the Pulse Streamer. The signal will consist of a single pulse which is repeated infinite number of times. While this example is extremely simple, yet it shows very typical way of defining and generating various signals.

Python

```
# import API classes into the current namespace
from pulsestreamer import PulseStreamer, Sequence

# A pulse with 10ţs HIGH and 30ţs LOW levels
pattern = [(10000, 1), (30000, 0)]
```

(continues on next page)

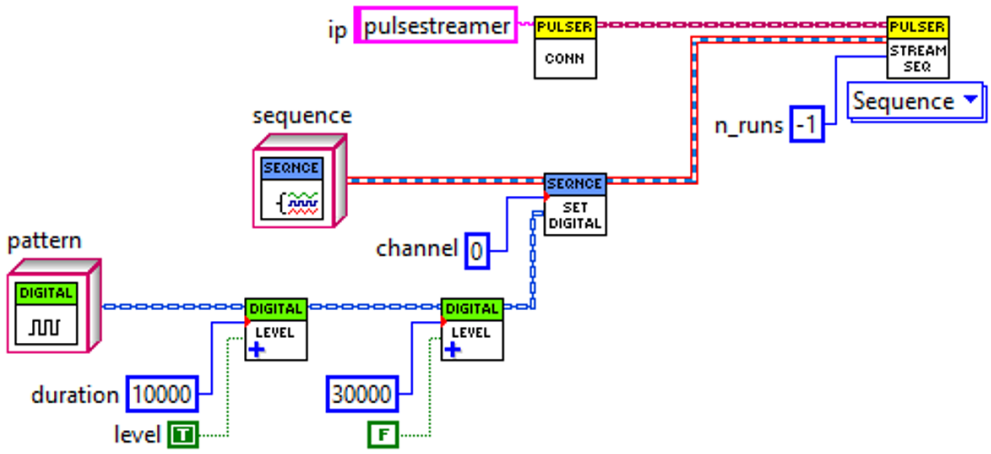
(continued from previous page)

```
# Connect to Pulse Streamer
ip = 'pulsestreamer'
ps = PulseStreamer(ip)

# Create a sequence object
sequence = ps.createSequence()

# Create sequence and assign pattern to digital channel 0
sequence.setDigital(0, pattern)

# Stream the sequence and repeat it indefinitely
n_runs = PulseStreamer.REPEAT_INFINITELY
ps.stream(sequence, n_runs)
```

MATLAB

```
% import API classes into the current namespace
import PulseStreamer.*

% A pulse with 10µs HIGH and 30µs LOW levels
pattern = {10000, 1; 30000, 0};

% Connect to Pulse Streamer
ip = 'pulsestreamer';
ps = PulseStreamer(ip);

% Create a sequence object
sequence = ps.createSequence();

% Assign pulse pattern to digital channel 0
sequence.setDigital(0, pattern);

% Stream the sequence and repeat it indefinitely
n_runs = PulseStreamer.REPEAT_INFINITELY; % endless streaming
ps.stream(sequence, n_runs);
```

LabVIEW



Detailed information about the programming interface of the Pulse Streamer and the API can be found in the *Programming interface* section.

CHAPTER

TWO

HARDWARE

2.1 Output Channels

The *The Pulse Streamer 8/2* has 8 digital and two analog output channels (SMA-connectors). The electrical characteristics are tabulated below.

2.1.1 Digital Output

Property	Value
Voltage level	0 to 3 V
Output drive	50
Sampling rate	1 GHz
Rise/fall time	1.6/1.1 ns
Bandwidth	~300 MHz

2.1.2 Analog Output

Property	Value
Output Voltage Range	-1.0 to 1.0 V
Resolution	14 bit
Output drive	50
Sampling rate	125 MHz
Rise/fall time	8.8/9.6 ns
Bandwidth	~35 MHz

2.2 Trigger Input

The Pulse Streamer 8/2 has one external trigger input, which can be enabled by software. By default, the Pulse Streamer is automatically rearmed after a sequence with a finite number of n_runs has finished. That means if another trigger occurs after a sequence has finished, the sequence will be re-triggered again. On the contrary, if a trigger occurs while a sequence is streamed, it will be discarded. We recommend, not to exceed 100 Hz as an external re-trigger frequency. Information about how to configure the trigger functionality of the Pulse Streamer can be found in the *Running pulse sequences* section.

Electrical characteristics:

Property	Value
Termination	50
Input voltage range	-0.3 to 3.45 V
Low level range	-0.3 to 0.8V
High level range	2.0 to 3.45 V
Minimum pulse width	~8ns
TriggerToData(min/max)	50/58ns

2.3 External Clock Input

The Pulse Streamer 8/2 has one input that can receive an external 125MHz or 10 MHz reference clock. Further information about how to set the clock-source of the Pulse Streamer can be found in the *Using an external clock* section. Before using the external clock the first time, please contact support@swabianinstruments.com

Electrical characteristics:

Property	Value
Termination	50
Input voltage range	-0.3 to 3.45 V
Low level range	-0.3 to 0.8V
High level range	2.0 to 3.45 V
Accepted frequencies	10/125 MHz

CHAPTER

THREE

NETWORK CONNECTION

In order to communicate with the Pulse Streamer, you need to know its IP address. By default, the device will attempt to acquire an IP address via DHCP. There is also a preconfigured second permanent IP address that allows direct connection to the PC, see *Permanent Static IP*.

```
# Example:
#  using default hostname
ps = PulseStreamer('pulsestreamer')

#  using fallback IP
ps = PulseStreamer('169.254.8.2')
```

3.1 Assign a static IP with the MAC address and DHCP

You can configure your DHCP server or router to assign a static DHCP IP to the Pulse Streamers MAC address. In this way you know the IP that the Pulse Streamer will receive by DHCP. The MAC address of your Pulse Streamer is specified on the bottom label of the device and is the same as serial number.

To verify your network configuration, open a terminal and enter

```
[user@host~] arp

Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.1.108            ether   00:26:32:f0:09:30   C                     wlp1s0
router                   ether   18:83:bf:c1:1f:67   C                     wlp1s0
```

In this example the first line is the Pulse Streamer and the second line is the router.

3.2 Permanent static IP: 169.254.8.2

The Pulse Streamer is always reachable via a permanent second static IP-address 169.254.8.2 (netmask 255.255.0.0). This address allows you to estabish connection when the Pulse Streamer is connected directly to your computer with an Ethernet cable. This shall work out-of-the-box on both Windows and Linux (requires avahi/zeroconf). In some cases, however, you may need to reboot your computer to detect the Pulse Streamer, if there has been a DHCP-connection before.

3.3 Modify the network settings

Requirements:

- network access to your Pulse Streamer (for example using 169.254.8.2 permanent static fallback)
- ssh / putty

If you wish to assign a certain IP address to your Pulse Streamer, you can disable DHCP and configure a static IP instead. We provide a tool for doing so on our website www.swabianinstruments.com. To modify your network settings:

- Download the network configuration script
- Start windows shell (press Windows+R, type `cmd` and hit Enter)
- In the Windows shell select the folder with the `setNetworkConf.bat` file
- Run the script with hostname or IP address as parameter

```
> setNetworkConf.bat pulsestreamer
```

or in case the IP address should be used

```
> setNetworkConf.bat IP_ADDRESS
```

where IP_ADDRESS is the address of the Pulse Streamer, e.g. 169.254.8.2

The tool will guide you through the network configuration. You can edit the IP address, the netmask and the standard gateway. Then you can test the new network settings and subsequently decide, if you want to set the static IP permanent. For assistance please contact support@swabianinstruments.com.

CHAPTER

FOUR

PROGRAMMING INTERFACE

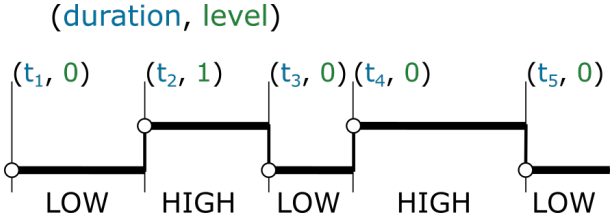
4.1 Overview

This section defines terminology used in this documentation and provides overview of how signals are defined with the Pulse Streamer API.

4.1.1 Pulse pattern

The pulse pattern is a sequence of levels that make up the signal to be produced. It is defined as an array of *(duration, level)* tuples, in other words, using Run-Length Encoding (RLE). In contrast to defining pulse patterns as an array of values with equal time durations, the RLE encoded pattern is more memory efficient, especially for patterns that consist of levels of both short and long durations.

The *duration* is always specified in nanoseconds and the *level* is either 0 or 1 for digital output or a real number between -1 and +1 in Volt for analog outputs. See the *Hardware* section for more details on the electrical properties of the generated signals.



The following code shows how to define a pulse pattern similar to the one shown in the image above. In addition, it shows an example of analog pattern definition.

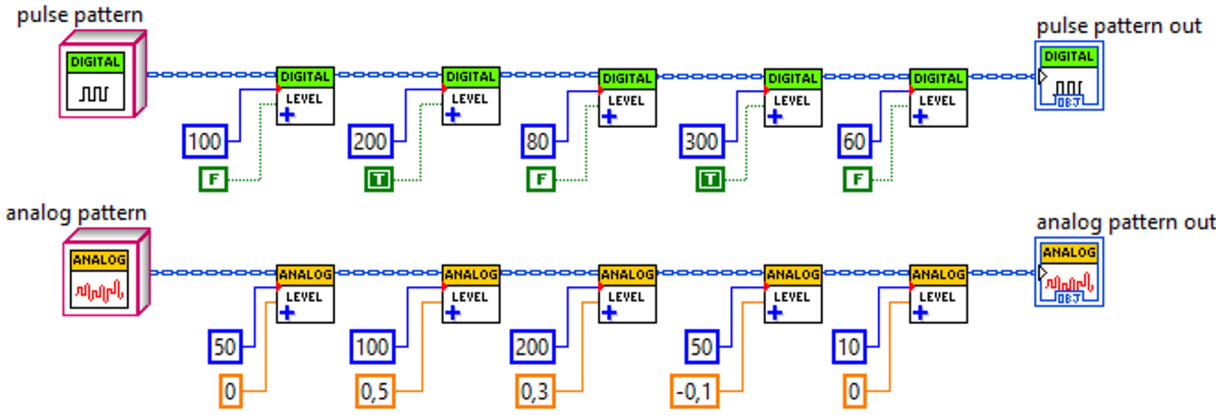
Python

```
pulse_patt = [(100, 0), (200, 1), (80, 0), (300, 1), (60, 0)]
analog_patt = [( 50, 0), (100, 0.5), (200, 0.3), (50, -0.1), (10, 0)]
```

Matlab

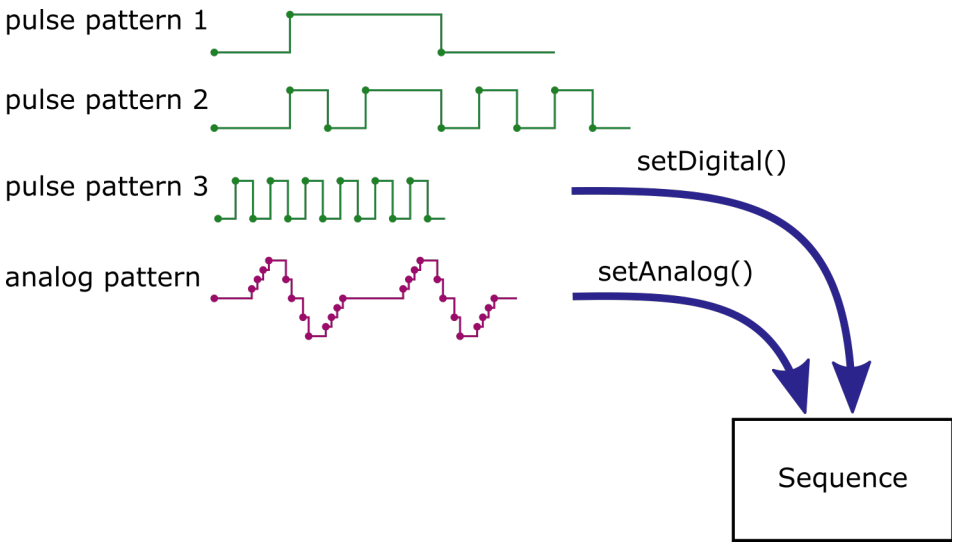
```
pulse_patt = {100, 0; 200, 1; 80, 0; 300, 1; 60, 0};
analog_patt = {50, 0; 100, 0.5; 200, 0.3; 50, -0.1; 10, 0};
```

LabVIEW



4.1.2 Creating sequences

Before a pattern can be sent for streaming to the Pulse Streamer outputs, they have to be mapped to the output channels. All these steps are performed with the *Sequence* object which is created with *PulseStreamer.createSequence()*. The digital and analog channel assignment is done with the *setDigital()* and *setAnalog()* methods, respectively.



Python

```
from pulsestreamer import PulseStreamer

ps = PulseStreamer('pulsestreamer')

seq = ps.createSequence()
seq.setDigital(0, pulse_patt)
seq.setDigital(2, pulse_patt)
seq.setAnalog(0, analog_patt)
```

Matlab

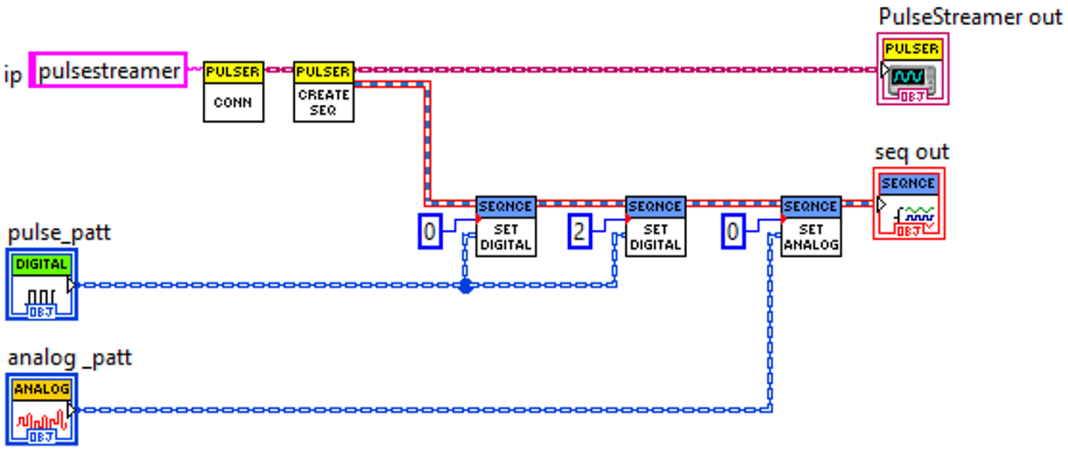
```
import PulseStreamer.PulseStreamer;

ps = PulseStreamer('pulsestreamer');
```

(continues on next page)

(continued from previous page)

```
seq = ps.createSequence();
seq.setDigital(0, pulse_patt);
seq.setDigital(2, pulse_patt);
seq.setAnalog(0, analog_patt);
```

LabVIEW



4.1.3 Sequence transformation

Sequence transformation methods enable the creation of complex sequences from simpler sub-sequences. The sequence data can be repeated or combined with another sequence. These operations while inherently simple have a few edge cases that are important to know. Concatenation and repetition operations are non-destructive, in other words, they preserve original sequence objects (immutability). The result is stored in a newly created sequence object. Internally, the sequence stores a map of the channel number and the pattern data with the pattern data unmodified. In general, this results in a sequence that consists of patterns having different durations. On concatenation or repetitions however, it is intuitively expected that a sequence is treated as a solid unit with every pattern of the same duration. Therefore, before concatenating the sequence data, the pattern durations are padded to the common duration.

When two sub-sequences being concatenated have a different set of mapped channels the resulting sequence will include them all. This is explained with the following example. Lets assume we have two sequences, *seq1* and *seq2*, which we want to concatenate. The *seq1* has patterns mapped to channels *(0,2)* and *seq2* has channels *(0,1)*, as shown in the code below.

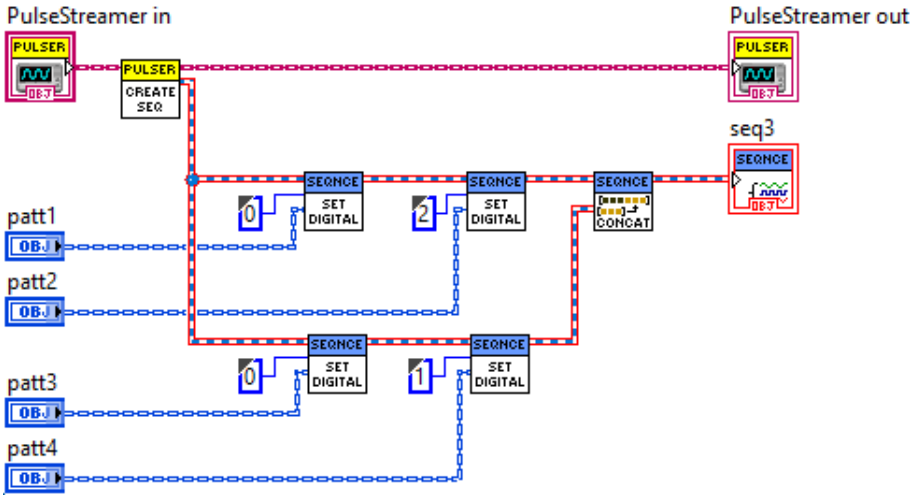
Python

```
seq1 = ps.createSequence()
seq1.setDigital(0, patt1)
seq1.setDigital(2, patt2)

seq2 = ps.createSequence()
seq2.setDigital(0, patt3)
seq2.setDigital(1, patt4)

seq3 = seq1 + seq2    # concatenation
```

Matlab

```
seq1 = ps.createSequence();
seq1.setDigital(0, patt1);
seq1.setDigital(2, patt2);

seq2 = ps.createSequence();
seq2.setDigital(0, patt3);
seq2.setDigital(1, patt4);

seq3 = [seq1, seq2];    % concatenation
```

LabVIEW



During the concatenation, the channel lists of the two sequences are compared and the output sequence *seq3* will include them all *(0,1,2)*. As a first step, a new sequence object *seq3* will be created as a copy of *seq1*, and an empty pattern will be assigned to the channel *1*. Next, all patterns in *seq3* will be padded to the duration of the longest one, which is essentially the sequence duration. Finally, the pattern data from *seq2* will be appended to the corresponding patterns of the *seq3*.

The duration padding is always performed with the value of a last element in the pattern when there is no previous element, the default value is used. The repetition process behaves similarly and can be qualitatively understood as multiple concatenations of the object with itself.

4.1.4 Streaming

Now, any of the sequence objects created before can be sent for streaming by calling the *PulseStreamer.stream()* method as shown in the following example for the sequence *seq*.

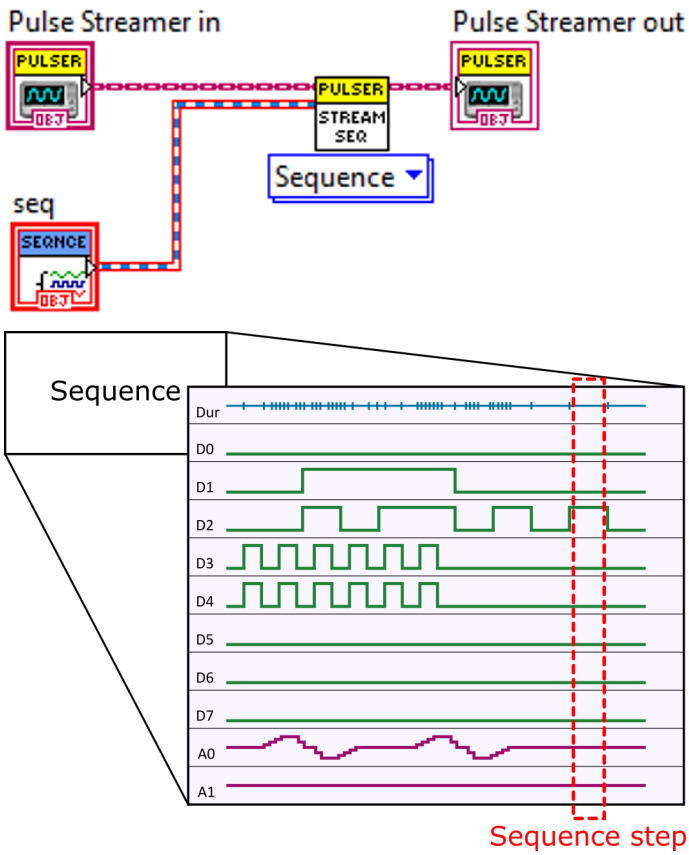
Python

```
ps.stream(seq)
```

Matlab

```
ps.stream(seq);
```

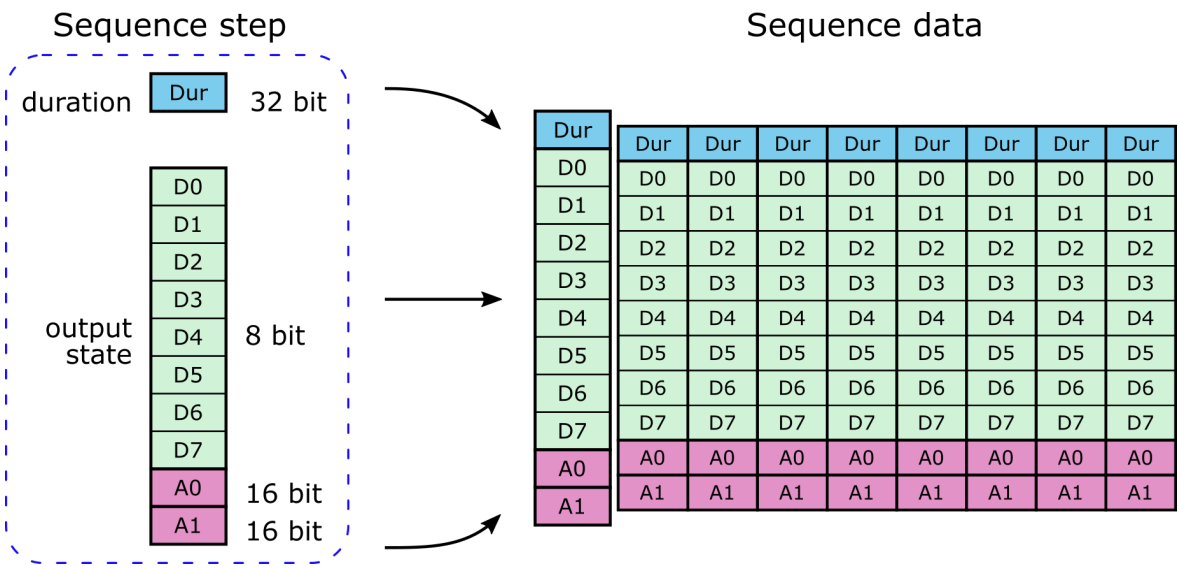
LabVIEW



On streaming, the sequence object is converted to a hardware specific run-length encoded data block which can be understood as an array of *Sequence steps*. Every step defines the state of all channels and the duration to hold the state. Sequences support a number of useful methods, like repetition, concatenation, preview plotting, etc. With this basic set of methods, complex sequences can be built from smaller and simpler sub-sequences.

4.1.5 Sequence step

The sequence step is the smallest element of a sequence that contains information on the state of every output of the Pulse Streamer and the *duration* to hold this state. The image below explains the relation between *Sequence step*, `Sequence`, and `OutputState` objects.



Warning: In a typical use of the client API, the user does not have to care about how to create or operate on the sequence data directly. All necessary functionality is enclosed within the API presented in this article. The description of the *Sequence data* corresponds to the RAW data as it is required by the hardware. The internal API data structures are implemented slightly differently for each programming language, aiming at optimization of the client performance. Furthermore, the RAW sequence data format is hardware dependent and future Pulse Streamer models are likely to use a different format. See also: *Low-level RPC interface*.

4.2 PulseStreamer

The `PulseStreamer` class is a wrapper for the RPC interface provided by the Pulse Streamer hardware. It handles the connection to the hardware and exposes all available methods. This class is implemented in various supported programming languages with consistently named methods. However, in some languages, additional functionality, common to that language is also implemented, like callback functions in MATLAB.

class PulseStreamer

PulseStreamer(*ip*)
The class constructor accepts a single string argument which can be either the IP address or a hostname by which the Pulse Streamer can be reached in the network. The constructor fails when the *ip* has incorrect value or device is not reachable. The Pulse Streamer hardware has a static fallback address 169.254.8.2. This allows operation when the Pulse Streamer is directly connected to a PC network card and without requiring any additional configuration.

Parameters **ip** (*str*) – IP address or hostname of the Pulse Streamer.

reset()
Reset the Pulse Streamer device to the default state. All outputs are set to 0V and all functional configurations are set to default. The automatic rearm functionality is enabled, the clock source is the internal clock of the device. No specific trigger functionality is enabled, which means that each sequence is streamed immediately when its upload is completed.

createSequence()
Use this method to create new hardware specific `Sequence` object. A hardware specific sequence object has exactly the same functionality as `Sequence` and implements early checks for hardware limits. For

example, an attempt to assign a pattern to a non-existing channel or setting analog voltage outside the DAC range will result in an error. While normal behavior of generic *Sequence* object is to check hardware limits only on the call to *PulseStreamer.stream()* method.

Returns Hardware specific *Sequence* object.

4.2.1 Setting constant output state

`PulseStreamer.`**`constant`**(*state=OutputState.ZERO*)

Set the outputs to a constant state. Calling the method without a parameter will result in the default output state with all output set to 0V. If you set the device to a constant output an eventually currently streamed sequence is stopped. It is not possible to re-trigger the last streamed sequence after setting the Pulse Streamer constant. *OutputState.ZERO* is a constant equal to `OutputState([],0,0)`.

Alternatively, the *state* parameter can be specified as a tuple consisting of three elements `([],0,0)`.

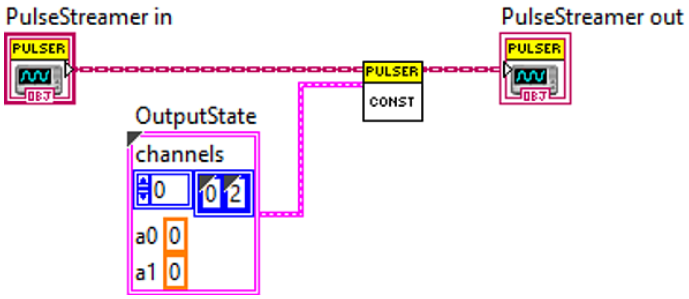
Python

```
ps.constant(OutputState([1, 2, 5], 0, 0))
# or
ps.constant(([1, 2, 5], 0, 0))
```

Matlab

```
ps.constant(OutputState([1, 2, 5], 0, 0));
% or
ps.constant({[1, 2, 5], 0, 0});
```

LabVIEW



Parameters **`state`** (`OutputState`) – *OutputState* object that defines the state of outputs or a tuple.

4.2.2 Running pulse sequences

This section describes methods that allow to run sequences and configure trigger properties. You can repeat a pulse sequence infinitely or an integer number of times. If you want to stop a running sequence and force it to the *final*, specified in the function call, you can do this by calling the method `forceFinal()`.

`PulseStreamer.`**`stream`**(*sequence*[, *n_runs=PulseStreamer.REPEAT_INFINITELY*[, *final=OutputState.ZERO*]])

Sends the sequence to the Pulse Streamer. After the sequence has been repeated the given *n_runs*, the constant state *final* will be reached. All parameters except *sequence* have default values and can be omitted. By default, the sequence is started immediately. Alternatively, you can select when

the output of the sequence starts with *setTrigger()* to a software-based trigger started with *startNow()* or to an external hardware trigger.

If the sequence is empty then the *final* state will set immediately.

The *sequence* parameter of the *stream()* method also accepts an RLE sequence defined as a list of 4 element tuples of the following format:

Python

```
ps.stream([(100, [1, 2], 0, 0), (10, [2], 0, 0), (5, [], 0, 0)])
```

Matlab

```
ps.stream([{100, [1, 2], 0, 0}, {10, [2], 0, 0}, {5, [], 0, 0}]);
```

Parameters

- **sequence** (Sequence) – Sequence object or a list of tuples.
- **n_runs** (*int*) – Number of times to repeat the sequence. Infinite repetitions if *n_runs<0*. There is also a symbolic constant *REPEAT_INFINITELY=-1*
- **final** (OutputState) – OutputState object which defines the constant output after the sequence has finished.

PulseStreamer.**setTrigger**(*start*[, *rearm=TriggerRearm.AUTO*])

Defines how the uploaded sequence is triggered.

If you want to trigger the Pulse Streamer by using the external trigger input of the device you have to set the *start* parameter to one of the following values: *start=HARDWARE_RISING* (rising edge on the trigger input), *HARDWARE_FALLING* (falling edge on the trigger input) or *HARDWARE_RISING_AND_FALLING* (both edges are active).

If the automatic rearm functionality is enabled (*rearm=AUTO*), which is the default power-on state, you can re-trigger a sequence which is finished, by providing an appropriate trigger signal, depending on *start* argument. You can disable the automatic rearm by passing *rearm=MANUAL*.

If the automatic rearm functionality is disabled, you can manually rearm the Pulse Streamer by calling the method *rearm()*

***TriggerStart* (enumeration)**

- *IMMEDIATE*: Trigger immediately after a sequence is uploaded. (default)
- *SOFTWARE*: Trigger by calling *startNow()* method.
- *HARDWARE_RISING*: External trigger on the rising edge.
- *HARDWARE_FALLING*: External trigger on falling edge.
- *HARDWARE_RISING_AND_FALLING*: External trigger on rising and falling edges.

***TriggerRearm* (enumeration)**

- *AUTO*: Trigger is rearmed automatically. (default)
- *MANUAL*: Trigger once only and do not rearm automatically. Rearm manually via the *rearm()* method.

Parameters

- **start** (*TriggerStart*) – Defines the source of the trigger signal

- **rearm** (*TriggerRearm*) – Enables or disables trigger automatic rearm.

PulseStreamer.**startNow**()
Starts streaming the sequence present in the Pulse Streamers memory. The behavior of this method depends on the trigger configuration performed with *setTrigger()* method.

If the *start* is *IMMEDIATE*, *n_runs*>=0 and the sequence has finished then *startNow()* will trigger the sequence again.

If the *start* is *SOFTWARE* then the sequence is started every time the *startNow()* is called. In case of the *rearm=TriggerRearm.MANUAL* the method *startNow()* will trigger the sequence only once. Call *rearm()* to manually rearm the trigger.

If the *start* is set to one of the hardware sources then this method does nothing.

PulseStreamer.**rearm**()
Rearm the trigger in case the Pulse Streamer has reached the final state of the current sequence and the trigger rearm method was set to *TriggerRearm.MANUAL*. Returns True on success.

Returns bool True or False

PulseStreamer.**forceFinal**()
Interrupts the sequence and sets the final state. This method does not modify the output state if the sequence has already finished and the Pulse Streamer is in the final state.

In case the Pulse Streamer is streaming a sequence and has not reached the final state, the upload speed of a new sequence with *stream()* is reduced by about 20%. This can be avoided by calling *forceFinal()* before uploading the sequence.

If no final state was declared in the current sequence, the output of Pulse Streamer will change to (or stay in) the last known constant state.

The recommended way to stop the Pulse Streamer streaming is to set its output to a constant value via the method *constant()*, described above.

PulseStreamer.**setCallbackFinished**(*callback_func) (MATLAB only*)
Allows to set up a callback function which will be called after the sequence streaming has finished. The callback function will be called with the following signature *callbackFunction(pulseStreamer_obj)*. An example of such a function is shown below.

Matlab

```
function callbackFunction(pulseStreamer)
    % this is an example of a callback function

    disp('hasFinishedCallback - Pulse Streamer finished.');

end
```

4.2.3 Requesting the streaming state

The following methods allow you to request whether the Pulse Streamer has a sequence in memory, whether a sequence is currently being streamed or if it has already finished.

PulseStreamer.**hasSequence**()
Returns *True* if the Pulse Streamer memory contains a sequence.

Returns True or False.

PulseStreamer.**isStreaming**()
Returns *True* if the Pulse Streamer is currently streaming a sequence. When the sequence has finished and the device remains in the final state, this method returns *False* again.

Returns True or False.

PulseStreamer.**hasFinished**()
Returns *True* if the Pulse Streamer remains in the final state after having finished a sequence.

Returns True or False.

4.2.4 Using an external clock

The Pulse Streamer can be clocked from three different clock sources. By default, the clock source is the internal clock of the device. It is also possible to feed in the system by an external clock of 125MHz (sampling clock) or an external 10MHz reference signal. You can choose the clock source via *selectClock()*. For more information on the required electrical parameters of an external clock signal please see the section *Hardware*.

PulseStreamer.**selectClock**(*source*)
Set the hardware clock source.

ClockSource (enumeration)

- INTERNAL: Use internal clock generator (default)
- EXT_125MHZ: Use external 125 MHz clock signal
- EXT_10MHZ: Derive clock from external 10 MHz reference signal

Parameters **source** (*ClockSource*) – Specifies clock source for the Pulse Streamer hardware.

4.2.5 Hardware identification

PulseStreamer.**getSerial**()

Returns String containing the serial number which is the same as MAC address of the Ethernet interface.

PulseStreamer.**getFirmwareVersion**()

Returns String containing the firmware version number of the connected Pulse Streamer.

4.3 Sequence

The Sequence contains information about the patterns and channel assignments. It also handles the mapping of patterns (see *Pulse pattern*) to output channels and has a number of built-in methods that operate on the whole sequence, like concatenation, repetitions, visualization, etc.

Warning: While the same pattern can be mapped to one or more channels, however successive mappings to the same channel will simply replace the previous mapping.

class Sequence

Sequence()
Class constructor. The *Sequence* is a generic class without early hardware limit checks. Since this class is not aware of hardware specific limitations, like available channels or analog range, the validation will be performed only during an attempt to stream this Sequence object.

Note: If you want to have early limit checks, channel number validation, please create a hardware specific Sequence object with *PulseStreamer.createSequence()* method. This, however, requires an active connection to the hardware.

---

setDigital(*channels*, *pattern*)
Assign a pattern to a digital output. The same pattern can be assigned to one or more channels simultaneously.

```
sequence.setDigital(0, patt1)
sequence.setDigital([1,2,6], patt2)
```

Parameters

- **channels** (*int_array*) – Digital channel number(s) as labelled on the Pulse Streamer connectors panel.
- **pattern** (*array*) – A pattern to be assigned to the *channel*.

setAnalog(*channels*, *pattern*)
Assign a pattern to an analog output. The same pattern can be assigned to one or more channels simultaneously.

```
sequence.setAnalog([0,1], patt2)
```

Parameters

- **channels** (*int_array*) – Analog channel number(s) as labelled on the Pulse Streamer connectors panel.
- **pattern** (*array*) – A pattern to be assigned to the *channel*.

4.3.1 Properties

Sequence.**isEmpty**()
Sequence.**isempty**() *(in MATLAB)*
Returns True if the sequence is empty.

Sequence.**getDuration**()
Returns sequence duration in nanoseconds.

Sequence.**getLastState**()
Returns the last state in the sequence as a OuputState object.

Returns *OutputState* Last state of the sequence.

Sequence.**getData**()
Returns the run-length encoded (RLE) sequence data. This method is called automatically by the *PulseStreamer.stream()* method.

Returns Run-length encoded data.

4.3.2 Transformation

(static) Sequence.repeat(sequence, n_times)

Returns the sequence data duplicated *n_times*. The sequence data in the original object remains unmodified. In case the *Sequence* object consists of patterns with different durations they will be padded to the longest one, which defines sequence duration as returned by *Sequence.getDuration()* method. In this context, a Sequence shall be understood as a set of patterns all of the same duration.

In Python, you can repeat sequences similarly to lists by multiplying it by a number.

Python

```
# The following three lines are fully equivalent
seq1 = Sequence.repeat(seq, 5)
seq1 = seq * 5
seq1 = 5 * seq
```

Matlab

```
seq1 = repeat(seq, 5)
```

Parameters

- **sequence** (*Sequence*) – Sequence object to be repeated.
- **n_times** (*int*) – Number of times the sequence is repeated.

Returns Returns a new *Sequence* object with data duplicated *n_times*.

(static) Sequence.concatenate(sequence1, sequence2)

Creates a new *Sequence* object with a sequence of *sequence2* object appended at the end of this sequence. Both, this object *sequence1* and *sequence2* objects remain unmodified. In case the *sequence1* sequence consists of patterns with different durations they will be padded to the longest one, which defines sequence duration as returned by *Sequence.getDuration()* method. In this context, a Sequence shall be understood as a set of patterns all of the same duration.

This method is exposed directly in LabVIEW, however, in MATLAB and Python it is exposed only as the override of concatenation operator. This allows for transparent use of any function of these languages that implicitly uses concatenation. See the example code for each language.

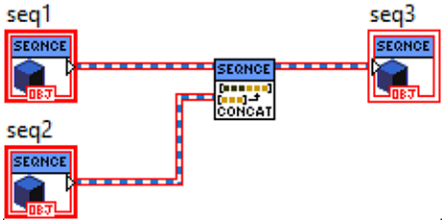
Python

```
seq3 = seq1 + seq2
```

Matlab

```
seq3 = [seq1, seq2];
```

LabVIEW



Returns Returns a new *Sequence* object with concatenated data.

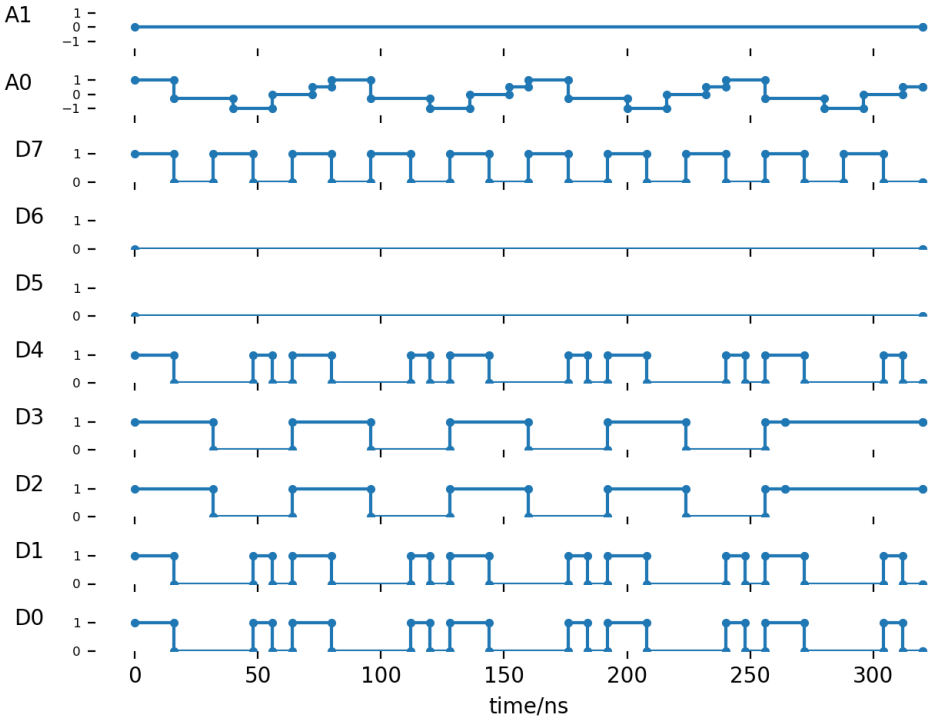
4.3.3 Visualization

`Sequence.`**`plotDigital`**`()`
Plots the sequence data for digital outputs. Plotting is done into the current axes. (Only in MATLAB and LabView)

`Sequence.`**`plotAnalog`**`()`
Plots sequence data for analog outputs. Plotting is done into the current axes. (Only in MATLAB and LabView)

`Sequence.`**`plot`**`()`
Calls *`plotDigital()`* and *`plotAnalog()`* and shows the results as subplots in a single figure. Plotting is done in the current figure.

An example of the *`plot()`* output is shown in the image below.



4.4 OutputState

The *OutputState* is a simple value class that contains information on the state of every output of the Pulse Streamer.

`class OutputState`

`OutputState``(`*channels*, *A0*, *A1*`)`
Class constructor. Input parameters specify the state of each output of the Pulse Streamer. Digital and analog output values are specified differently. In order to set a HIGH level at the digital channel, add the channel number to the *channels* list, for example `channels=[0,2,3]` will set HIGH level on the channels *0*, *2* and *3*. All other digital channels will be set to LOW. Output values at each of two analog outputs are specified with corresponding parameters *A0* and *A1*.

Parameters

- **channels** (*list*) – List of digital channels to be set HIGH.
- **A0** (*float*) – Analog output 0 voltage in Volt.
- **A1** (*float*) – Analog output 1 voltage in Volt.

ZERO

This is a helper constant equal to `OutputState([], 0, 0)`.

CHAPTER

FIVE

LOW-LEVEL RPC INTERFACE

This section is an introduction into the low-level RPC interface of the Pulse Streamer. You can use this information for more deeper understanding of how to control Pulse Streamer hardware or to develop your own communication programs. We recommend to make use API described in the section *Programming interface* that greatly simplifies the pulse pattern generation.

5.1 Pulse Sequences

Note: The code examples given in this paragraph assume C++ programming language and use name conventions as in the firmware code. The naming may deviate from that used by high-level programming interface described in the section *Programming interface*. The data format given here applies only to the *Pulse Streamer 8/2*.

Pulse sequences are represented as one dimensional array of `Pulse` structures. Each `Pulse` specifies its duration and the states of the digital and analog output channels. The C++ data type is:

```
struct Pulse {
    uint_t32 ticks; // duration in ns
    uint_t8 digi; // bit mask
    int_t16 ao0;
    int_t16 ao1;
};
```

The duration `ticks` is specified in nanoseconds.

The lowest bit in the digital bit mask `digi` corresponds to channel 0, the highest bit to channel 7. A channel is high when its corresponding bit is 1 and low otherwise.

The analog values span the full signed 16 bit integer range, i.e. -1.0 V corresponds to -0x7fff and 1.0 V corresponds to 0x7fff. Note that the DAC resolution is 12 bits, i.e., the 4 LSB are ignored.

5.2 Resetting Pulse Streamer to constant outputs

You can reset the Pulse Streamer by using the method

```
void reset()
```

All outputs are set to 0V and all functional configurations are set to default. The automatic rearm functionality is enabled, the clock source is the internal clock of the device. No specific trigger functionality is enabled, which means that each sequence is streamed immediately when its upload is completed.

5.3 Setting constant outputs

You can set the outputs to a constant state.

The method is:

```
void constant(Pulse pulse=CONSTANT_ZERO)
```

CONSTANT_ZERO is a symbolic constant for a pulse with the value {0,0,0,0} Calling the method without a parameter will result in the default output state with all outputs set to 0V. If you set the device to a constant output an eventually currently streamed sequence is stopped. It is not possible to re-trigger the last streamed sequence after setting the Pulse Streamer constant.

5.4 Running pulse sequences

Running a pulse sequence corresponds to a single function call where you pass your pulse sequence as an argument.

You can repeat a pulse sequence infinitely or an integer number of times. A sequence run will start from the current constant output state.

The method to run a pulse sequence is:

```
void stream(std::vector<Pulse> sequence,
            int_t64 n_runs=INFINITE,
            Pulse final=Pulse{0,0,0,0},
            start_t start=IMMEDIATE
            )
```

sequence represents the pulse sequence

After the sequence has been repeated the given n_runs , the final output state will be reached.

The sequence is repeated infinitely if n_runs < 0 and a finite number of repetitions otherwise. INFINITE is a symbolic constant with the value -1. final represent the constant output after the sequence is finished (the tick values are ignored).

All parameters except sequence have default values and can be omitted.

By default, the sequence is started immediately. Alternatively, you can tell the system to wait for a later software start command or for an external hardware trigger applied via

```
void setTrigger(start_t start, trigger_mode_t mode=NORMAL)
```

start is an enum with the mapping {IMMEDIATE:0, SOFTWARE:1, HARDWARE_RISING:2, HARDWARE_FALLING:3, HARDWARE_RISING_AND_FALLING:4} specifying how the stream should be started. If you have passed start=SOFTWARE, you can start the sequence using the method

```
void startNow()
```

If you want to trigger the Pulse Streamer by using the external trigger input of the device you have to pass HARDWARE_RISING (rising edge is the active trigger flank), HARDWARE_FALLING (falling edge is the active trigger flank) or HARDWARE_RISING_AND_FALLING (both edges are active) to the start argument. mode is an enum with the mapping{NORMAL:0, SINGLE:1}. If automatic rearm functionality is enabled (mode=NORMAL) you can re-trigger a successfully finished sequence, by the trigger mode you selected with the start argument. You can disable the automatic rearm by passing SINGLE to the mode argument.

If automatic rearm functionality is disabled, you can manually rearm the Pulse Streamer by using the method

```
void rearm()
```

After that you can re-trigger a successfully finished sequence exactly one time, by the trigger mode you selected with the start argument.

You can check whether a sequence is available in memory by the following method

```
bool hasSequence()
```

The method returns true if a previous sequence is available.

You can check whether the Pulse Streamer is currently streaming by calling the method

```
bool isStreaming()
```

The method returns true if the Pulse Streamer is streaming a sequence. When the sequence is finished and the device remains in the final state, this method returns false again.

You can check whether the last sequence has finished successfully by calling the method.

```
bool hasFinished()
```

This method returns true if the Pulse Streamer remains in the final state after having finished a sequence.

The recommended way to stop the Pulse Streamer streaming is to set its output to a constant value via the method constant(), described above. However, if you want to stop a running sequence and force it to the dedicated final state, you can do this by calling the method

```
void forceFinal()
```

If no final state was declared in the current sequence, the output of Pulse Streamer will change to (or stay in) the last known constant state. Furthermore, if upload-performance is crucial to your application, you can call this function directly before streaming the next sequence. This will increase the upload-performance by about 20 percent.

5.5 More features

The Pulse Streamer can be fed in with three different clock sources. By default, the clock source is the internal clock of the device. It is also possible to feed in the system by an external clock of 125MHz (sampling clock) or an external 10MHz reference clock. You can choose the clock source via

```
void selectClock(clocking_t clock_source)
```

clock_source is an enum with {INTERNAL:0, EXT_125MHZ:1, EXT_10MHZ:2}.

If you want to get the serial number of your device or the ID-number of the built-in FPGA, you can call the method

```
std::string getSerial(serial_t serial=ID)
```

The method returns a hexadecimal string containing either the serial number/MAC-address or the ID-number of the FPGA depending of the value of the argument serial. Serial is an enum with the mapping {ID:0; MAC:1}.

If you want to get the version number of the current firmware you can call the method

```
std::string getFirmwareVersion()
```

5.6 Communicating with the instrument

Your Pulse Streamer 8/2 contains an embedded operating system. You connect to the embedded system over LAN through straight forward Remote Procedure Calls (RPC). Requests to the system are directly converted into C++ calls. This architecture gives you direct control over the system.

You can connect to the device via two RPC interfaces. (i) a JSON-RPC interface that is based on the well-established JSON data format (http://www.json.org/) (ii) a Google RPC interface (gRPC) that is based on Googles data exchange format (https://developers.google.com/protocol-buffers/). Both RPC interfaces provide the same functionality, but the default and recommended communication protocol between the PC and the Pulse Streamer is JSON-RPC.

5.7 JSON-RPC Interface

JSON-RPC libraries are available for most software languages. More information can be found on the official website https://www.jsonrpc.org/ and on Wikipedia https://en.wikipedia.org/wiki/JSON-RPC.

The JSON-RPC URL of the Pulse Streamer is `http://<pulse_streamer_ip>:8050/json-rpc`, where <pulse_streamer_ip> is the IP address of your Pulse Streamer.

5.7.1 Sending Data over JSON-RPC

There is no native format for sending array data over JSON-RPC. Therefore, the pulse sequence is sent as a binary string. Since the HTTP transport layer requires string data to be base64 encoded, one conversion step is needed before sending a sequence. The JSON-RPC interface call can be performed with an array

```
{base64 string sequence,
 int_t64 n_runs,
 {uint_t32 ticks, uint_t8 digi, int_t16 ao0, int_t16 ao1},
}
```

or with named parameters

```
{"sequence":base64 string sequence,
 "n_runs":int_t64 n_runs,
 "final":{uint_t32 ticks, uint_t8 digi, int_t16 ao0, int_t16 ao1},
}
```

sequence is the array data as per above C++ data format definition packed into a binary string and converted to a base64 string. Please check out the Python example for connecting to the JSON-RPC server `random_pulses_json.py`. All other arguments are self-explanatory as per the above sections.

Other remote call methods are one to one correspondences to the C++ interface.

5.8 gRPC Interface

gRPC (https://grpc.io/) is a new RPC interface that is based on Googles well established data exchange format called Protocol Buffers (https://developers.google.com/protocol-buffers/). There are gRPC libraries available for most programming languages. Note that gRPC requires the new Protobuf3 standard.

The gRPC server of the Pulse Streamer is `<pulse_streamer_ip>:50051`, where `<pulse_streamer_ip>` is the IP address of your Pulse Streamer.

5.8.1 Sending Data over gRPC

In gRPC, data types are defined by generic, language independent templates. The language specific implementation automatically takes care about conversion to native data types.

The Pulse Streamer interface looks like this.

```
package pulse_streamer;

message VoidMessage {}

message PulseMessage {
    uint32 ticks = 1;
    uint32 digi = 2;
    int32 ao0 = 3;
    int32 ao1 = 4;
}

message SequenceMessage {
    repeated PulseMessage pulse = 1;
    int64 n_runs = 2;
    PulseMessage final = 3;
}

message TriggerMessage {
    enum Start {
        IMMEDIATE = 0;
        SOFTWARE = 1;
        HARDWARE_RISING = 2;
        HARDWARE_FALLING = 3;
        HARDWARE_RISING_AND_FALLING = 4;
    }
    Start start = 1;
    enum Mode {
        NORMAL = 0;
        SINGLE = 1;
    }
    Mode mode = 2;
}

message ClockMessage {
    enum Clocking {
        INTERNAL = 0;
        EXT_125MHZ = 1;
        EXT_10MHZ = 2;
    }
    Clocking clock_source = 1;
}

message GetSerialMessage{
    enum Serial{
        ID=0;
        MAC=1;
    }
    Serial serial =1;
}

message PulseStreamerReply {
```

(continues on next page)

(continued from previous page)

```
    uint32 value = 1;
}

message PulseStreamerStringReply {
    string string_value=1;
}

service PulseStreamer {
    rpc reset (VoidMessage) returns (PulseStreamerReply) {}
    rpc constant (PulseMessage) returns (PulseStreamerReply) {}
    rpc forceFinal (VoidMessage) returns (PulseStreamerReply) {}
    rpc stream (SequenceMessage) returns (PulseStreamerReply) {}
    rpc startNow (VoidMessage) returns (PulseStreamerReply) {}
    rpc setTrigger (TriggerMessage) returns (PulseStreamerReply) {}
    rpc rearm (VoidMessage) returns (PulseStreamerReply) {}
    rpc selectClock (ClockMessage) returns (PulseStreamerReply) {}
    rpc isStreaming (VoidMessage) returns (PulseStreamerReply) {}
    rpc hasSequence (VoidMessage) returns (PulseStreamerReply) {}
    rpc hasFinished (VoidMessage) returns (PulseStreamerReply) {}
    rpc getFirmwareVersion (VoidMessage) returns (PulseStreamerStringReply) {}
    rpc getSerial (GetSerialMessage) returns (PulseStreamerStringReply) {}
}
```

CHAPTER

SIX

CHANGELOG

6.1 2019-08-07

6.1.1 Firmware update v1.0.3

- solved: at rare intervals occurring server crashes
- solved: channel analog0 shows increased jitter (observed only on very few devices)
- solved: extremely rarely missed internal trigger (sequence dependent)

6.2 2019-05-10

6.2.1 Client update v1.1.2

Matlab

- Bug-fix in the `PulseStreamer.debug.PulseStreamer_RPCLogger` class.
- `PulseStreamer_RPCLogger` class now stores log-file snapshots on RPC errors.

6.3 2019-04-23

6.3.1 Clients update v1.1.1

- Python: corrected overflow error for sequence durations above 4 seconds on some systems.
- Python client is now available at www.pypi.org. You can now install it with `pip install pulsestreamer`.
- Matlab: minor code cleanup.

6.4 2019-03-01

6.4.1 Client API update v1.1.0

This update brings homogenized API for PulseStreamer clients in all supported languages. From now on, the signatures of all currently present functions and methods are frozen and will remain stable over the future minor updates and releases. In the future, we plan to add any new functionality in a backwards compatible way with no user code modifications required.

- The API was slightly redesigned and homogenized in all supported languages.
- The use of high-level clients is now a recommended way of programming and streaming the pulse sequences with the Pulse Streamer.
- New methods `PulseStreamer.createSequence` and `PulseStreamer.createOutputState` that create hardware specific `Sequence` and `OutputState` objects.
- New method `PulseStreamer.getFPGAID`.
- New `OutputState` class for defining the state of Pulse Streamer outputs in some methods.
- New `OutputState.ZERO` constant.
- New named constant `PulseStreamer.REPEAT_INFINITELY = -1` for infinite sequence repetition.
- `Sequence` object now applies padding to the pattern data and previous previous levels on concatenation.
- Renamed enum `TriggerMode` to `TriggerRearm`, also renamed enumeration values.
- Renamed enumeration values in enums `TriggerStart`.
- Modified signature of the `PulseStreamer.getSerial()` method which now has no input parameter and always returns hardware serial number.

Matlab

- The client code is now distributed as a packaged Matlab Toolbox.
- Now the PulseStreamer client is placed into its own namespace `PulseStreamer` in order to prevent possible collisions in function names. You can use `import PulseStreamer.*` to shorten the class names.
- Moved compatibility functions for FW v0.9 to `PulseStreamer.compat` sub-package.
- `Sequence.setDigital` and `Sequence.setAnalog` allows for overwriting mapped pulse patterns even after concatenation or repetition.
- `PulseStreamer.stream()` method now supports `Sequence` object and `[{duration,chan_list,a0,a1}; {...}]` cell array as input.
- Functionality of `PSSequenceBuilder` and `PSSequence` classes is now combined and moved to `Sequence` class.
- Renamed `PSTriggerMode` to `TriggerRearm`.
- Renamed `PSTriggerStart` to `TriggerStart`.
- Renamed `PSSequence` to `Sequence`.
- Removed `PSSerial` enum.
- `Sequence.plot` method plots the sequence data exactly as defined by user without resampling to common time.

- `Sequence` is completely decoupled from `PulseStreamer` class. Use `PulseStreamer.createSequence()` method to create a `Sequence` object that does early channel number validation.
- Helper classes like `PulseStreamer_Dummy` and `PulseStreamer_RPCLogger` located in a sub-package `PulseStreamer.debug`.
- Solved problem with multiple timers created on repeated script runs with long sequences. Only one timer can exist for a given device.

LabView

- Pulse Streamer client code for LabView is now distributed as a VIPM package.
- Functionality of `SequenceBuilder` and `Sequence` classes is now combined into `Sequence` class.
- Added `Sequence:plot.vi` which plots the pulse patterns according to user input with no resampling to common time.
- `PulseStreamer:stream.vi` method is a polymorphic VI with wrappers to handle implementation VIs with dynamic inputs.
- `PulseStreamer:stream.vi` supports `Sequence` object, old `Pulse` array, and `RLEdata` cluster as inputs.
- Renamed `Digital Pattern.lvclass` to `PulsePattern.lvclass`.
- Renamed `Analog Pattern.lvclass` to `AnalogPattern.lvclass`.

Python

- PulseStreamer client is distributed as wheel package `pulsestreamer`.
- PulseStreamer now uses standard `tinyrpc` package instead of previously used customized version `tinyrpc3.py`. Use `pip install tinyrpc` to install the package, if missing.
- Some changes in parameter names. Please see *Programming interface*.
- New `concatenate`, `repeat` and `` `plot `` methods are added to `Sequence` class.
- PulseStreamer client is organized into a python module with cleaner layout.
- `PulseStreamer.stream` method accepts `Sequence` object as input parameter directly.

6.5 2018-12-17

6.5.1 Firmware update v1.0.2

- solved: occasionally missed external trigger

6.6 2018-11-09

6.6.1 Firmware update v1.0.1

- API has been supplemented by method rearm() and forceFinal()
- second permanent IP 169.254.8.2 added

- network configuration file on user partition -> static IP can be configured via RPCs
- login password changed

6.6.2 Clients

Python

- adapted to new API
- class Sequence added as handy sequence-builder
- channel_map {ch0:0, ch1:1} no longer supported - use channel_list e.g. [0,1,3,7]

Matlab

- adapted to new API
- large changes in the way sequences are created and manipulated
- new classes for sequence creation: `PSSequenceBuilder` and `PSSequence`
- classes `P` and `PH` are modified and labeled as deprecated
- added compatibility function `convert_PPH_to_PSSequence` that converts sequences created with `P` or `PH` objects into `PSSequence`
- added examples that show how to migrate old code to version 1.0
- code examples completely reworked to reflect new way of building sequences

LabView

- adapted to new API
- large changes in the way sequences are created and manipulated
- new classes for sequence creation: `SequenceBuilder` and `Sequence`
- client code is now contained in a LabView library.
- slightly modified and renamed classes for signal pattern creation
- code examples completely reworked to reflect new way of building sequences

6.7 2018-10-10

firmware update v1.0

- underflows do not occur any more -> getUnderflow() returns 0 always
- API changes (see API-migration-doc for details)
- substantial changes in the embedded Linux-operation system
- no network configuration file - only DHCP and fallback IP available

Clients

- Python, Matlab and LabVIEW adapted to new API

6.8 2018-01-05

user interface

- added a GUI to determine the IP address of the Pulse Streamer and to create simple pulses (beta release)

clients

- improved Python client

6.9 2017-05-07

clients

- added LabVIEW client
- improved Matlab client
- improved Python client

documentation

- added Getting Started section

6.10 2016-04-08

Matlab client

- added links to the Matlab client examples

6.11 2016-03-17

static sequence beta 0.9

- enums in RPCs
- API name changes
- rising and falling edges on external trigger

6.12 2016-03-07

provide network configuration

- added section on network configuration

6.13 2016-03-03

static sequence alpha

- initial, final, underflow states

- software start
- external trigger
- rerun sequence
- separate underflow flags for digital and analog
- optional values in jRPC

6.14 2016-02-02

static sequence alpha

CHAPTER

SEVEN

PREVIOUS VERSIONS

Here you can find an archive of documentation for previous versions of PulseStreamer clients.

Generally the versions of client software and firmware versions are not the same.

7.1 Version 1.x

- client: v1.1.x - firmware: v1.0.x
- client: v1.0.x - firmware: v1.0.x

7.2 Version 0.x

- client: v0.9 - firmware: v0.9

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- search

INDEX

constant()PulseStreamer method, 13
createSequence()PulseStreamer method, 12

forceFinal()PulseStreamer method, 15

getData()Sequence method, 17
getDuration()Sequence method, 17
getFirmwareVersion()PulseStreamer method, 16
getLastState()Sequence method, 17
getSerial()PulseStreamer method, 16

hasFinished()PulseStreamer method, 16
hasSequence()PulseStreamer method, 15

isEmpty()Sequence method, 17
isempty()Sequence method, 17
isStreaming()PulseStreamer method, 15

OutputStatebuilt-in class, 19
OutputState()OutputState method, 19

plot()Sequence method, 19
plotAnalog()Sequence method, 19
plotDigital()Sequence method, 19
PulseStreamerbuilt-in class, 12
PulseStreamer()PulseStreamer method, 12

rearm()PulseStreamer method, 15
reset()PulseStreamer method, 12

selectClock()PulseStreamer method, 16
Sequencebuilt-in class, 16
Sequence()Sequence method, 16
setAnalog()Sequence method, 17
setCallbackFinished()PulseStreamer method, 15
setDigital()Sequence method, 17
setTrigger()PulseStreamer method, 14
startNow()PulseStreamer method, 15
stream()PulseStreamer method, 13

ZEROOutputState attribute, 20