



# Pulse Streamer 8/2 Documentation

*Release 1.7*

**Swabian Instruments**

**Jun 03, 2023**



# CONTENTS

<b>1</b>	<b>Getting Started</b>	<b>1</b>
1.1	Software installation . . . . .	1
1.2	Generate simple pulse pattern . . . . .	2
1.3	Firmware update . . . . .	5
<b>2</b>	<b>Hardware</b>	<b>7</b>
2.1	Output Channels . . . . .	7
2.2	Trigger Input . . . . .	9
2.3	External Clock Input . . . . .	13
2.4	Status LEDs . . . . .	14
<b>3</b>	<b>Network Connection</b>	<b>17</b>
3.1	Assign a static IP with the MAC address and DHCP . . . . .	17
3.2	Permanent static IP: 169.254.8.2 . . . . .	18
3.3	Modify the network settings . . . . .	18
<b>4</b>	<b>Programming interface</b>	<b>21</b>
4.1	Overview . . . . .	21
4.2	Module level functions . . . . .	26
4.3	PulseStreamer . . . . .	27
4.4	Sequence . . . . .	35
4.5	OutputState . . . . .	39
4.6	Advanced (Beta) features . . . . .	40
<b>5</b>	<b>Low-level RPC interface</b>	<b>43</b>
5.1	Pulse Sequences . . . . .	43
5.2	Resetting Pulse Streamer to constant outputs . . . . .	43
5.3	Setting constant outputs . . . . .	44
5.4	Running pulse sequences . . . . .	44
5.5	More features . . . . .	45
5.6	Communicating with the instrument . . . . .	46
5.7	JSON-RPC Interface . . . . .	46
5.8	gRPC Interface . . . . .	47
<b>6</b>	<b>Changelog</b>	<b>49</b>
6.1	2023-06-01 . . . . .	49
6.2	2023-04-03 . . . . .	49
6.3	2023-03-08 . . . . .	49
6.4	2023-02-27 . . . . .	50
6.5	2023-02-16 . . . . .	50
6.6	2022-10-05 . . . . .	50

6.7	2022-05-02	50
6.8	2022-02-28	50
6.9	2021-12-20	51
6.10	2021-08-31	51
6.11	2021-08-23	51
6.12	2021-07-28	52
6.13	2021-05-20	52
6.14	2021-03-12	52
6.15	2021-02-12	52
6.16	2020-11-12	53
6.17	2020-08-17	53
6.18	2020-07-27	53
6.19	2020-01-20	54
6.20	2019-08-07	54
6.21	2019-05-10	54
6.22	2019-04-23	55
6.23	2019-03-01	55
6.24	2018-12-17	57
6.25	2018-11-09	57
6.26	2018-10-10	58
6.27	2018-01-05	58
6.28	2017-05-07	58
6.29	2016-04-08	59
6.30	2016-03-17	59
6.31	2016-03-07	59
6.32	2016-03-03	59
6.33	2016-02-02	59
<b>7</b>	<b>Previous versions</b>	<b>61</b>
7.1	Version 1.x	61
7.2	Version 0.x	61
<b>8</b>	<b>Indices and tables</b>	<b>63</b>
	<b>Index</b>	<b>65</b>

## GETTING STARTED

### 1.1 Software installation

The Pulse Streamer does not require any driver installation. It uses a 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 is supported as well.

#### 1.1.1 Client software

Please visit the [Software Downloads](#) section of our website and download the client and example files for any of the supported programming languages. Alternatively, the packages for Python, Matlab and LabVIEW are available via the PyPi, Matlab Add-on Explorer and VI Package Manager respectively.

The MATLAB toolbox can be installed with [MATLAB Add-on Explorer](#) or by downloading and double-clicking the toolbox package file (Windows). The toolbox is compatible with MATLAB versions starting from 2014b.

The LabVIEW package installation requires free [JKI VI Package Manager](#), which is often installed alongside LabVIEW. You can find the “Pulse Streamer” package in the community repository if you search for it in the VI Package Manager.

The Python module for Pulse Streamer can be installed either from the [www.pypi.org](http://www.pypi.org) with the following command

```
pip install pulsestreamer  
  
# or if you want to use gRPC client then  
# install the pulsestreamer with optional "grpc" support  
pip install pulsestreamer[grpc]
```

or from a local package file using the command

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

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

## 1.1.2 Graphical user interface

The Pulse Streamer Application provides basic configuration functionality and allows for generation of simple signals.

1. Download and install the most recent [Pulse Streamer Windows App](#) from our downloads site.
2. Start the Pulse Streamer (GUI) from the Windows start menu.
3. The GUI-window should show up on your screen.

---

**Note:** If you have already installed Pulse Streamer App v1.0.2, you have to uninstall the old version before installing version  $\geq$ v1.5.0.

---

## 1.2 Generate simple pulse pattern

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

### 1.2.1 Client software

Python

MATLAB

LabVIEW

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

# 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()

# 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)
```

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

% A pulse with 10µs HIGH and 30µs LOW levels
```

(continues on next page)

(continued from previous page)

```

pattern = {10000, 1; 30000, 0};

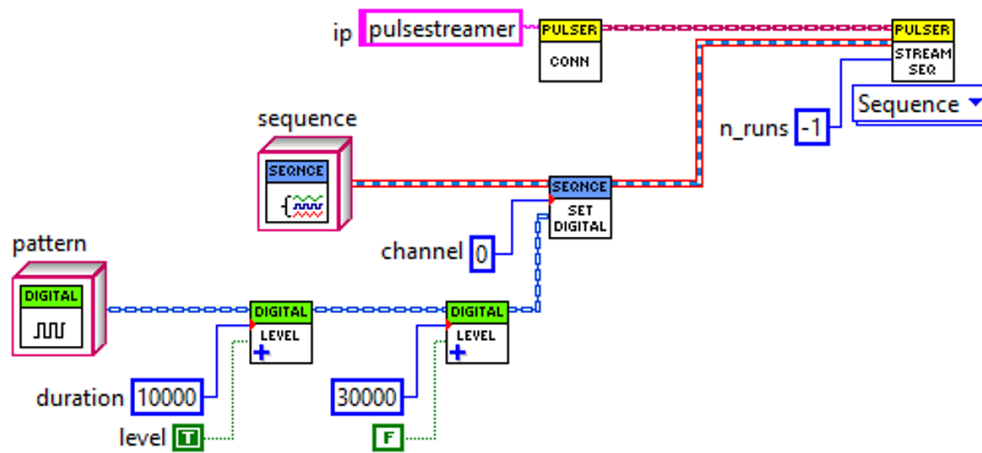
% Connect to Pulse Streamer
ip = 'pulsestreamer';
ps = PulseStreamer(ip);

% 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_INFINITY; % endless streaming
ps.stream(sequence, n_runs);

```



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

## 1.2.2 Graphical user interface

After startup, the Pulse Streamer Application scans the network for connected Pulse Streamers and shows detailed information on all discovered devices.

If your device is not shown, make sure it is correctly connected to the network and press the button **Auto Discover**. Devices with firmware version v1.0.x can only be discovered with **Network Scan**.

1. Choose your device and click **Connect**.
2. On the right side, change the drop down field **DIGITAL 0** from constant to **Pulse**.
3. Set **period** to 40000 ns and **Duration** to 10000 ns.
4. Click the **Play** button.

**Note:** The Pulse Streamer application provides easy-to-use device setup functionality, such as network configuration and firmware update. It currently only provides basic functionality concerning sequence generation and streaming. Essentially, it allows setting simple pulse repetitions or square waves for demonstration issues. Particularly its capabilities of merging different channel patterns are limited. Therefore putting individual pulse patterns on different digital

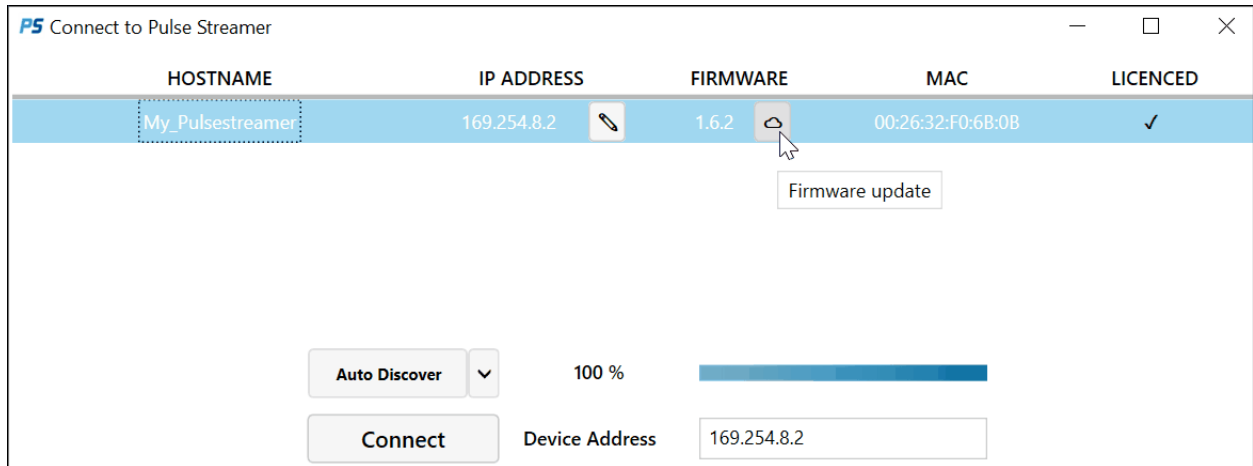


Fig. 1: Pulse Streamer Application: Start screen

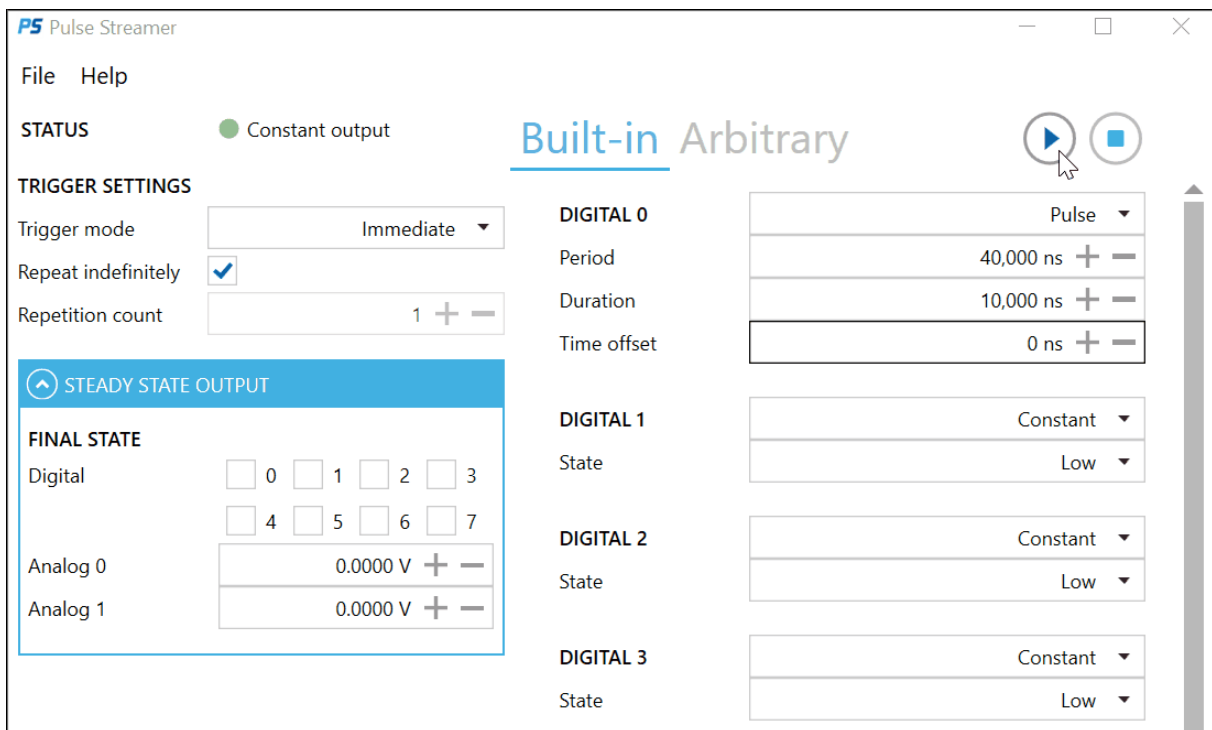


Fig. 2: Pulse Streamer Application: Sequence configuration



channels with only one or a few repetition counts will always preserve the desired frequencies (set via `Period`, `Time offset` and `Duration`) but might lead to deviation of the `Repetition count` set in the GUI window. We recommend using our API to benefit from the full functionality of the Pulse Streamer 8/2.

### 1.3 Firmware update

We recommend continuously updating all devices to our latest Pulse Streamer 8/2 firmware. The Pulse Streamer GUI will inform you about available updates and guide you through firmware update process. On the startup-screen with the listed devices, press the button `Update firmware` behind the field with the firmware version to open the firmware-updater-window and follow the further instructions.

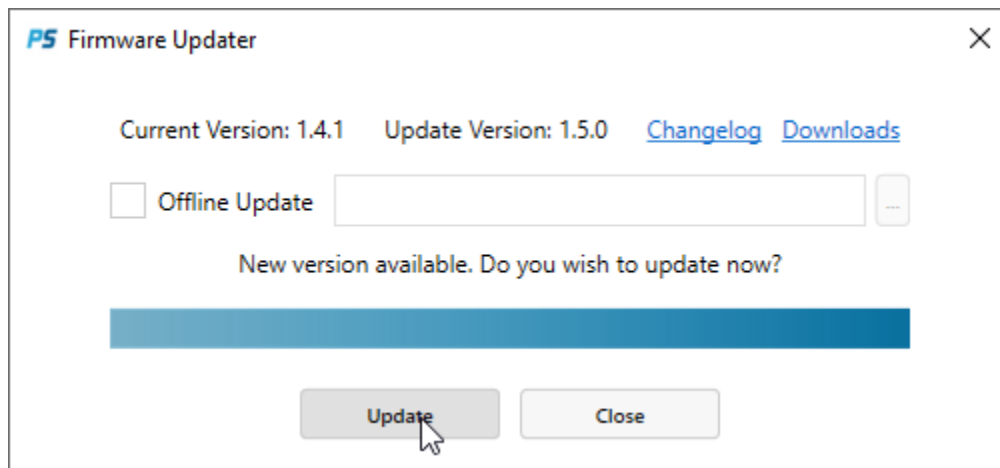


Fig. 3: Pulse Streamer Application: Firmware Updater

In case you need to update a device located in a network without internet access, you can manually download the `updater` file from our downloads site. After that, you can perform the firmware update process independently by using the offline mode and specifying the path to the updater.

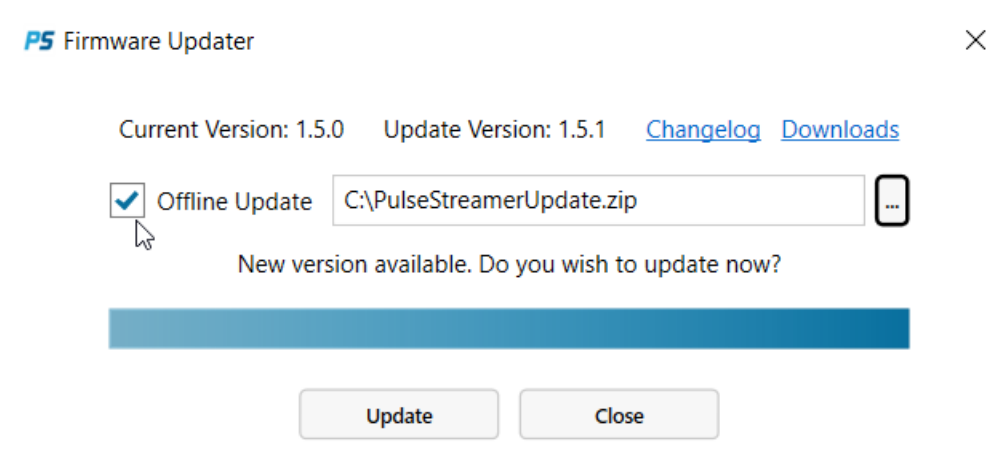


Fig. 4: Pulse Streamer Application: Firmware Updater offline mode

If you face any problems or if your device is not equipped with firmware v1.0.1 or later, please contact [support@swabianinstruments.com](mailto:support@swabianinstruments.com). We will provide a customized firmware update for you.



## HARDWARE

### 2.1 Output Channels

The *Pulse Streamer 8/2* has 8 digital and 2 analog output channels. The electrical characteristics are tabulated below.

#### 2.1.1 Digital Output

Hardware Version 3.x

Hardware Version  $\leq 2.x$

Property	Value
Output into 50 Ohm	0 and 2.6 V
Output impedance <sup>1</sup>	~ 13 Ohm
Sampling rate	1 GHz
Rise/fall time (20%-80%)	< 300 ps
Minimum pulse width	2 ns
RMS jitter	< 50 ps

Property	Value
Output into 50 Ohm	0 and 3 V
Output impedance <sup>1</sup>	~ 5 Ohm
Sampling rate	1 GHz
Rise/fall time (20%-80%)	< 1.1 ns
Minimum pulse width	3 ns
RMS jitter	< 50 ps

<sup>1</sup> The Pulse Streamer expects a 50 Ohm termination to avoid reflections. All output voltages imply the 50 Ohm load termination. Without termination, you will get a slightly higher output voltages due to the output impedance  $> 0$  Ohm.

## 2.1.2 Analog Output

Hardware Version 3.x

Hardware Version 3.1/3.0

Hardware Version <= 2.x

Property	Value
Sampling rate	125 MHz
Output into 50 Ohm	-1.0 to 1.0 V
Output impedance <sup>1</sup>	~ 2 Ohm
Bandwidth (-3db)	50 MHz
Resolution	14 bit
Offset error (into 50 Ohm load)	< 2 mV
Gain error (into 50 Ohm load)	< 1 %
Rise/fall time (20%-80%)	< 7 ns
Step response overshoot (typ.)	25 %
Output settling time (1%)	< 100 ns
Crosstalk (analog)	< -45 dB
Crosstalk (digital)	< -55 dB

Property	Value
Sampling rate	125 MHz
Output into 50 Ohm	-1.0 to 1.0 V
Output impedance <sup>Page 7, 1</sup>	~ 2 Ohm
Bandwidth (-3db)	50 MHz
Resolution	14 bit
Offset error (into 50 Ohm load)	< 2 mV
Gain error (into 50 Ohm load)	< 1 %
Rise/fall time (20%-80%)	< 7 ns
Step response overshoot (typ.)	25 %
Output settling time (1%)	< 100 ns
Crosstalk (analog)	< -45 dB
Crosstalk (digital)	< -55 dB

**Warning:** Around 20s after power-cycling, the analog outputs have a short pulse (duration ~30ns) with a voltage level of -0.5V.

Property	Value
Sampling rate	125 MHz
Output into 50 Ohm <sup>2</sup>	-1.0 to 1.0 V
Output impedance <sup>Page 7, 1</sup>	~ 2 Ohm
Bandwidth (-3db)	50 MHz
Resolution	14 bit
Accuracy <sup>3</sup>	±5 mV
Rise/fall time (20%-80%)	< 7 ns
Crosstalk (analog)	< -45 dB
Crosstalk (digital)	< -55 dB

**Note:**

**Warning:** During power-up, the analog outputs have an undefined output voltage value in the range of -1V to 1V.

**Note:**

## 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. The sequence can be retriggered after the sequence has finished and the retrigger dead-time has passed. Triggers that come early are discarded. Information about how to configure the trigger functionality of the Pulse Streamer can be found in the *Running pulse sequences* section

Electrical characteristics:

Hardware Version 3.4

Hardware Version <= 3.3

Hardware Version 2.x

Property	Value
Termination	50 Ohm
Max. voltage range (no damage)	-0.3 to 5.3 V
Input voltage range	0 to 5 V
Trigger level	0.5 V
Minimum pulse width (rising/falling) <sup>4</sup>	4 ns
<i>TriggerToData</i> (rising/falling, typ.) <sup>Page 9, 4</sup>	65/66 ns
<i>TriggerToData</i> jitter	±4 ns
Retrigger dead-time <sup>6</sup>	< 50 ns

**Note:**

<sup>2</sup> Some devices may have the actual full range smaller by up to 30 mV.

<sup>3</sup> Accuracy is specified with 50 Ohm load. Pulse Streamer 8/2 shipped with firmware version v1.3.0 or later include calibration data for the analog outputs. Devices with previous firmware versions require calibration to achieve the specified accuracy. You can do the calibration yourself. Please follow the instructions in the *Calibrating the analog outputs* section.

<sup>4</sup> Measured with a trigger signal amplitude of 0 to 2.6 V

<sup>6</sup> The minimum time gap required between the end of the streamed sequence to trigger the next sequence.

Property	Value
Termination	50 Ohm
Max. voltage range (no damage)	-0.3 to 5.3 V
Input voltage range	0 to 5 V
Trigger level	0.5 V
Minimum pulse width (rising/falling) <sup>5</sup>	4/8 ns
<i>TriggerToData</i> (rising/falling, typ.) <sup>Page 10, 5</sup>	65/68 ns
<i>TriggerToData</i> jitter	±4 ns
Retrigger dead-time <sup>Page 9, 6</sup>	< 50 ns

**Note:**

Property	Value
Termination	50 Ohm
Max. voltage range (no damage)	0 to 3.3 V
Input voltage range	0 to 3.3 V
Low-level range	0 to 0.8 V
High-level range	2.0 to 3.3 V
Minimum pulse width	< 2 ns
<i>TriggerToData</i> (typ.)	60 ns
<i>TriggerToData</i> jitter	±4 ns
Retrigger dead-time <sup>Page 9, 6</sup>	< 50 ns

**Note:**

### 2.2.1 TriggerToData

The trigger to data delay depends on the phase of the incoming trigger event relative to the clock of the Pulse Streamer (125 MHz). The trigger to data signal path shows for the default case (internal clock signal) a delay that is equally distributed between 61 and 69 ns as shown in the following figure.

The reason for this distribution is that all output clocks are derived from the Pulse Streamer clock signal (125 MHz). The positive clock edge samples the trigger events which results in an accuracy of 8 ns. This situation is visualized in the following figure.

<sup>5</sup> Measured with a trigger signal amplitude of 0 to 2.6 V

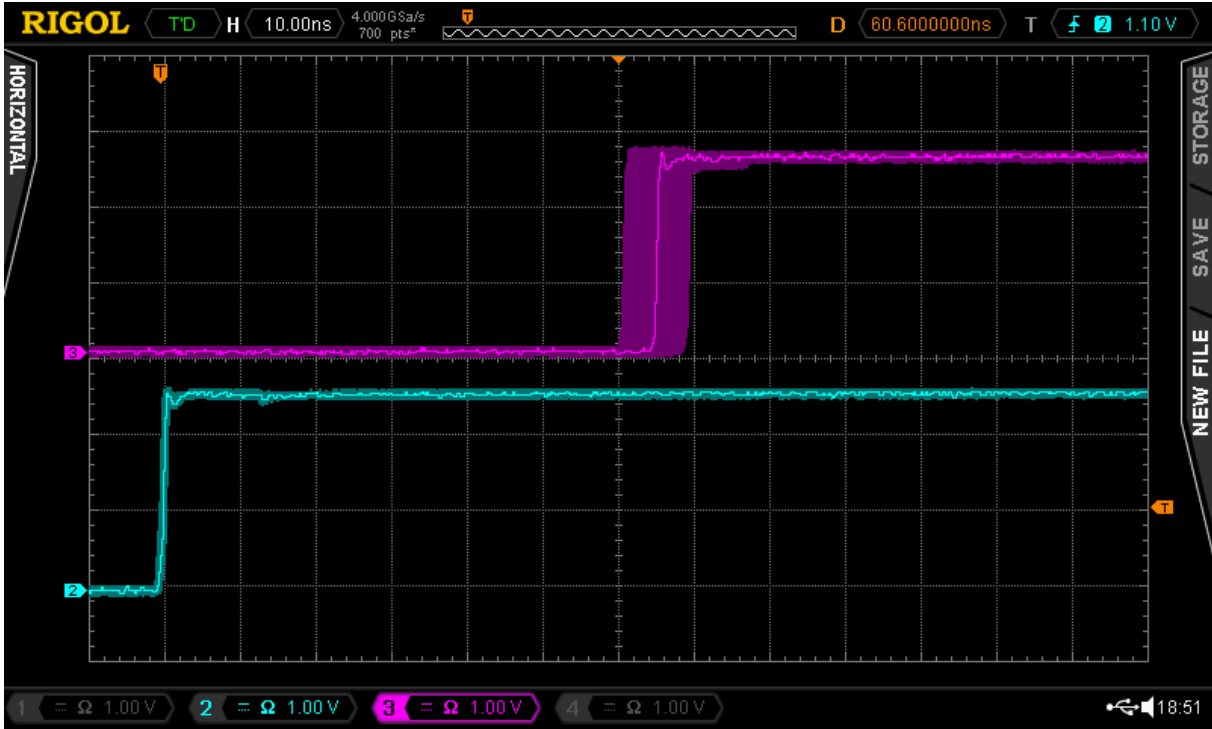


Fig. 1: TriggerToData: Trigger signal (bottom) and the data-out signal range (top)

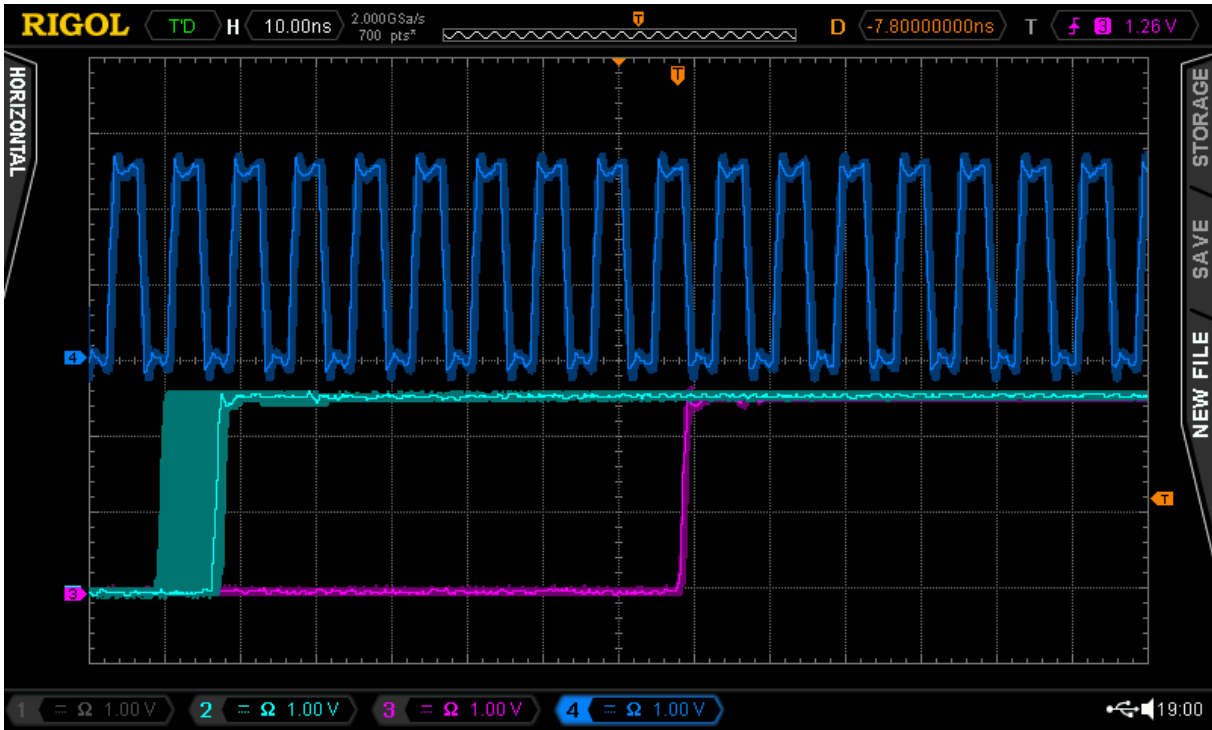


Fig. 2: Pulse Streamer clock signal (top, 125MHz), trigger signal (bottom left) and the data-out signal range (bottom right)

## How to avoid the trigger to data jitter

One way to circumvent the TriggerToData jitter is to use the Pulse Streamer as a master device. When the other devices have a lower TriggerToData jitter, this can solve the issue.

Another possibility is to phase-align the the Pulse Streamer clock and the trigger signal.

### 2.2.2 Synchronization of Trigger and Pulse Streamer clock

The jitter of the TriggerToData can be avoided by phase aligning the trigger signal with the Pulse Streamer clock. You can achieve synchronization by using the external 125 MHz clock input capability of the Pulse Streamer 8/2 (see [Using an external clock](#)). All internal clocks related to the Pulse Streamer output stages will be derived from the signal fed to the clock input.

If the external trigger and the clock of the Pulse Streamer 8/2 are phase-aligned, it will lead to a fixed TriggerToData value between 61-69 ns. The exact value depends on the trigger's phase position and the positive edge of the clock signal as shown in the following figure:

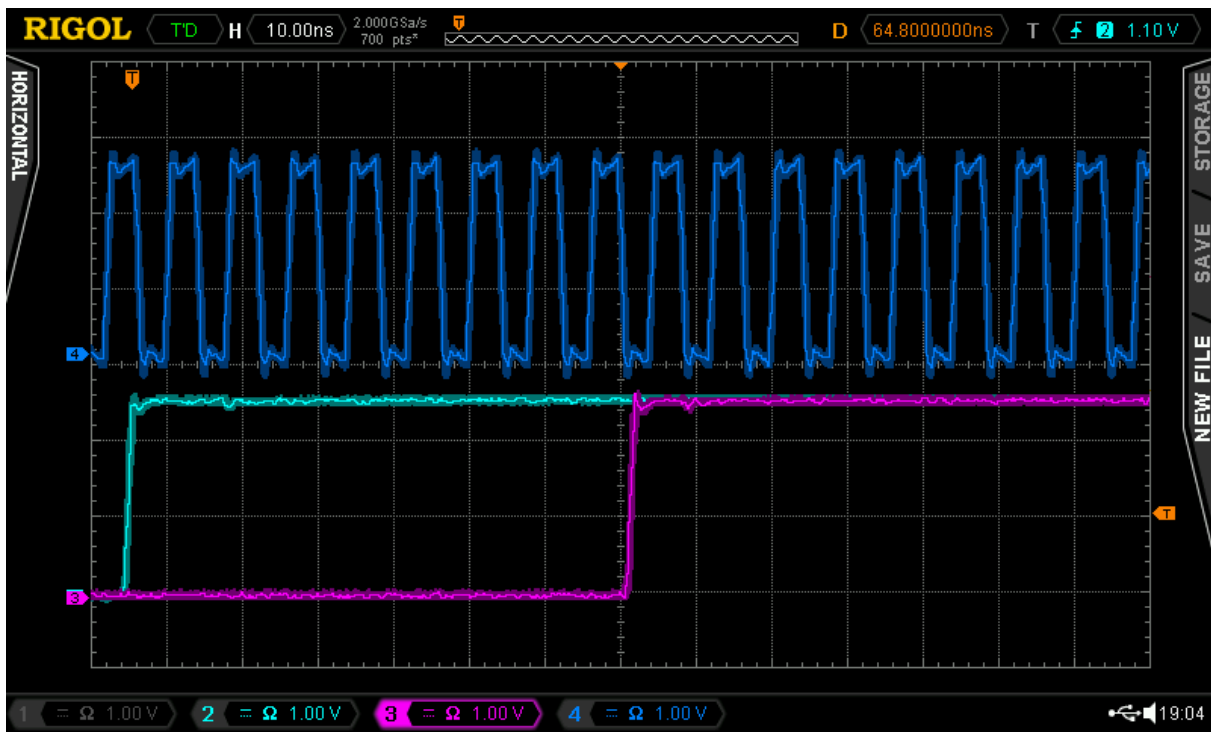


Fig. 3: Pulse Streamer clock (top, 125MHz), trigger signal (bottom left) and the data-out signal range (bottom center)

**Note:** When the trigger is exactly at the position of the sampling clock, the TriggerToData of each trigger will be randomly +0ns or +8ns as illustrated in the figure below:



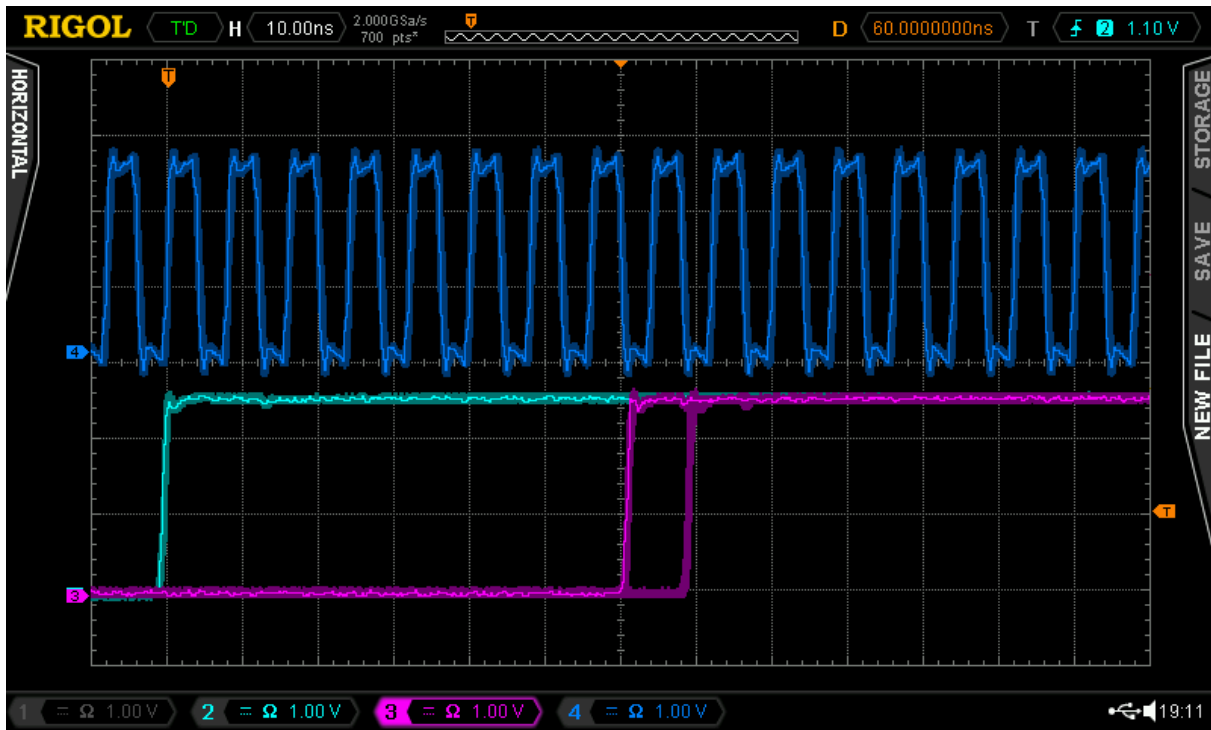


Fig. 4: Clock signal (top), trigger signal (bottom left) at the critical position with  $dt=0$  showing the discrete 8 ns output jitter (bottom right)

When this situation occurs, please shift your signal ideally by half the clock period (4ns), e.g., with a longer or shorter cable on the clock or trigger signal line.

**Note:** In case you are using 10MHz as an external reference clock for the Pulse Streamer, phase alignment with the 125 MHz is not possible.

## 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.

Electrical characteristics:

Hardware Version 3.x

Hardware Version 2.x

Hardware Version  $\leq$  2.1

Property	Value
Termination	50 Ohm
Coupling	AC coupled
Amplitude range	0.2 - 5 Vpp
Accepted frequencies	10 or 125 MHz

Property	Value
Termination	50 Ohm
Input voltage range	0 to 3.3 V
Low-level range	0 to 0.8 V
High-level range	2.0 to 3.3 V
Accepted frequencies	10 or 125 MHz

**Warning:** Due to hardware limitations, there is a 100mV ripple on the digital outputs if an external clock source is connected to the Pulse Streamer 8/2. The analog channels are not affected.

Property	Value
Termination	50 Ohm
Input voltage range	0 to 3.3 V
Low-level range	0 to 0.8 V
High-level range	2.0 to 3.3 V
Accepted frequencies	10 or 125 MHz

**Warning:** Due to hardware limitations, there is a 100mV ripple on the digital outputs if an external clock source is connected to the Pulse Streamer 8/2. The analog channels are not affected.

**Note:** These hardware revisions of the Pulse Streamer 8/2 have ambiguous input port labels. The correct input port for the external clock is the second one on the left side labeled with **\*GP In\*** or **\*SlowDigital 1\***.

---

## 2.4 Status LEDs

The Pulse Streamer 8/2 has two LEDs showing information about the status of the device and the network connection.

Device status LED:

green	Pulse Streamer successfully booted
blinking green-orange	sequence is streaming
orange	waiting for trigger/retrigger
blue	sequence finished - retrigger disable
blinking blue	no valid license
continuous red	error

Network LED:

red	no configuration/connection
blinking green-red	setting DHCP - no connection
green	setting DHCP - connection found
blinking blue-red	setting static IP - no connection
blue	setting static IP - connection found



## NETWORK CONNECTION

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')
```

Starting from the Pulse Streamer firmware v1.2, you can discover all accessible Pulse Streamers in the network and their IP addresses using *findPulseStreamers()*.

```
# Example
# query the network for all connected Pulse Streamers
devices = findPulseStreamers()

# query the network for a Pulse Streamer with specific serial number
devices = findPulseStreamers("00:26:32:F0:3B:1B")
```

### 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 Streamer's MAC address. In this way, you know the IP that the Pulse Streamer will receive by DHCP. You find the MAC address of your Pulse Streamer on the bottom label of the device. It is the same as the 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 establish a 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

By default, the Pulse Streamer 8/2 will attempt to acquire an IP address via DHCP. If you want to assign a specific IP address to your device, you can disable DHCP and configure a static IP instead. We recommend using our Pulse Streamer Application (Windows only) to modify the Pulse Streamer 8/2 network configuration. The graphical user interface will guide you through the network configuration. You can enable/disable DHCP and set a specific IP address, the netmask, and the default gateway for a static IP configuration. You can test the new network settings before deciding to apply the configuration permanently in a second step.

Requirements:

- network access to your Pulse Streamer (at least via permanent static fallback 169.254.8.2)
- Pulse Streamer 8/2 firmware version 1.5.0 or later
- Pulse Streamer Application [Software Downloads](#)

Configuring the network settings:

1. On the startup-screen with the listed devices, click the button `Edit network configuration` behind the IP-address field.
2. Choose your settings in the pop-up window.
3. Click `Test Settings` to check if the device is reachable via the new settings (power-cycling resets the changes).
4. Click `Apply` to set the configuration permanent.

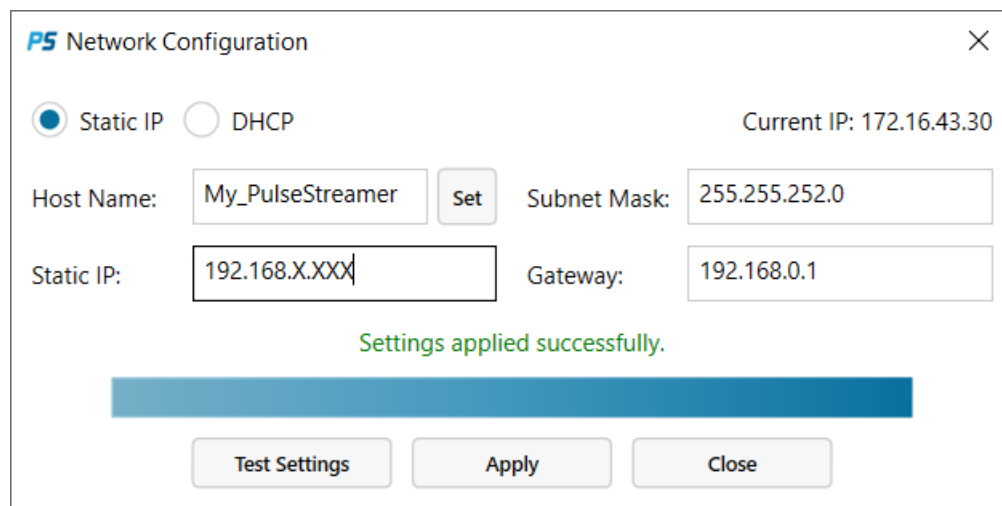


Fig. 1: Pulse Streamer Application: Network Configuration

Alternatively, you can configure the device's network settings via our client software interfaces. For more information, see *Modify the network configuration* section.

If you need assistance with the network configuration, please contact [support@swabianinstruments.com](mailto:support@swabianinstruments.com).





## PROGRAMMING INTERFACE

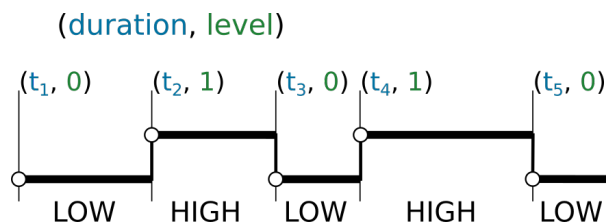
### 4.1 Overview

This section defines terminology used in this documentation and provides an overview of how signals can be generated with the Pulse Streamer API.

#### 4.1.1 Pulse pattern

The pulse pattern is a sequence of levels, defining the signal to generate. 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. 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 an analog pattern definition.

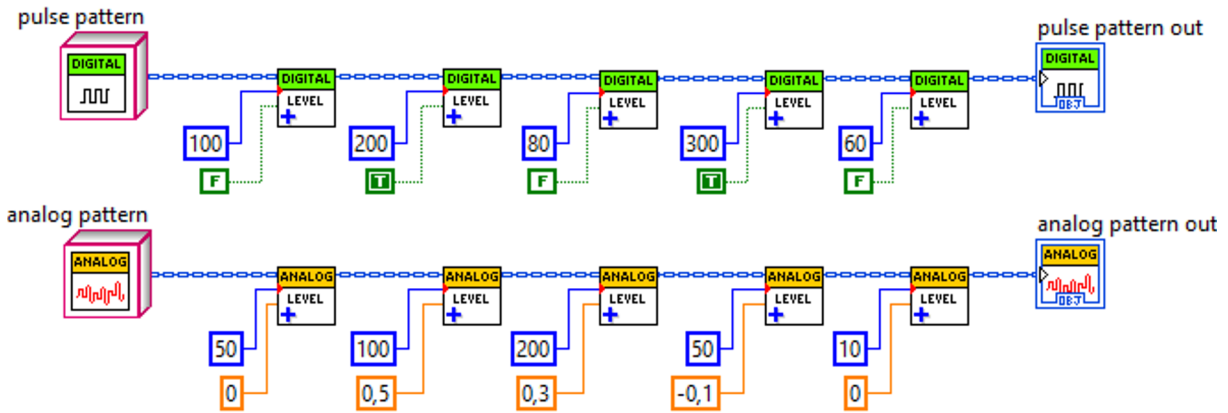
Python

Matlab

LabVIEW

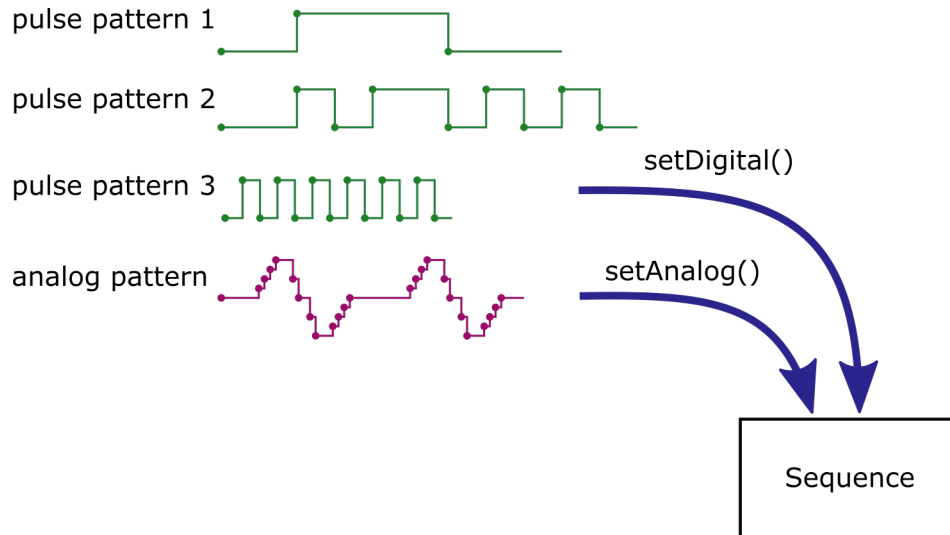
```
pulse_patt = [(100, 0), (200, 1), (80, 0), (300, 1), (60, 0)]  
analog_patt = [(50, 0), (100, 0.5), (200, 0.3), (50, -0.1), (10, 0)]
```

```
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};
```



### 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  
 Matlab  
 LabVIEW

```

from pulsestreamer import PulseStreamer

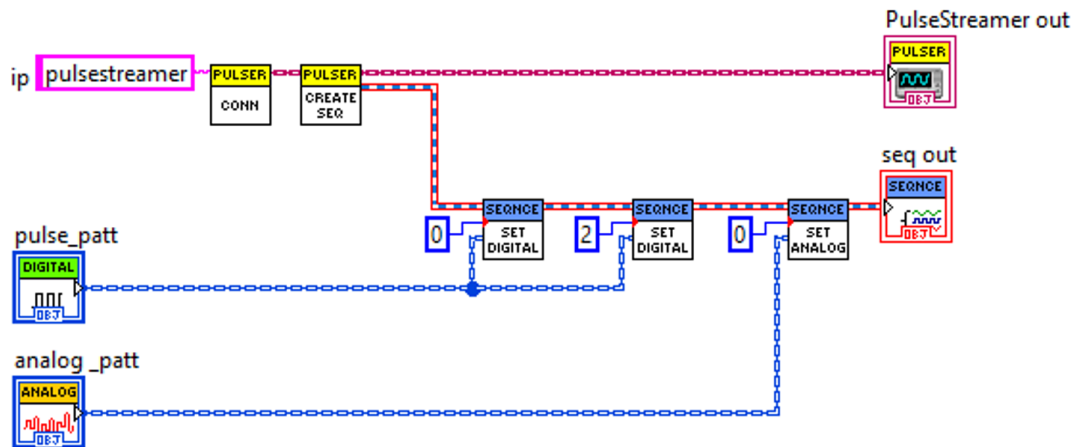
ps = PulseStreamer('pulsestreamer')

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

```
import PulseStreamer.PulseStreamer;

ps = PulseStreamer('pulsestreamer');

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



### 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. Let's 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

Matlab

LabVIEW

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

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

(continues on next page)

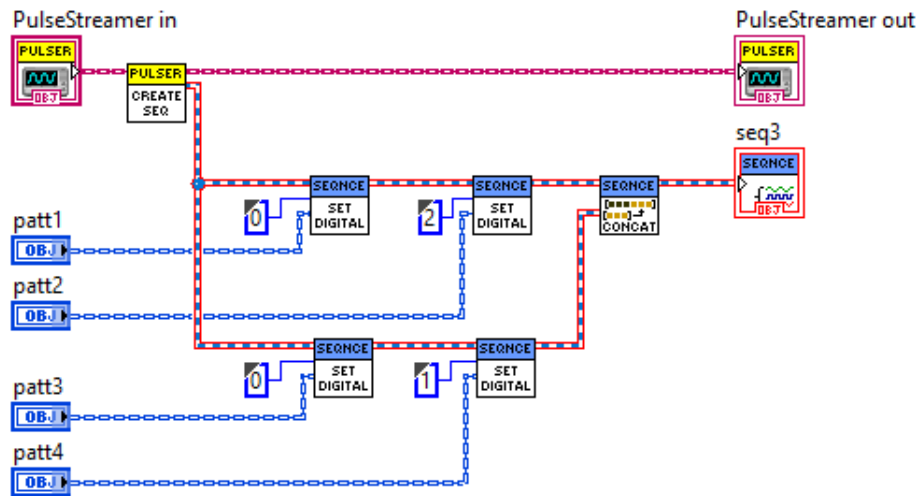
(continued from previous page)

```
seq3 = seq1 + seq2  # concatenation
```

```
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
```



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 the 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  $seq$ .

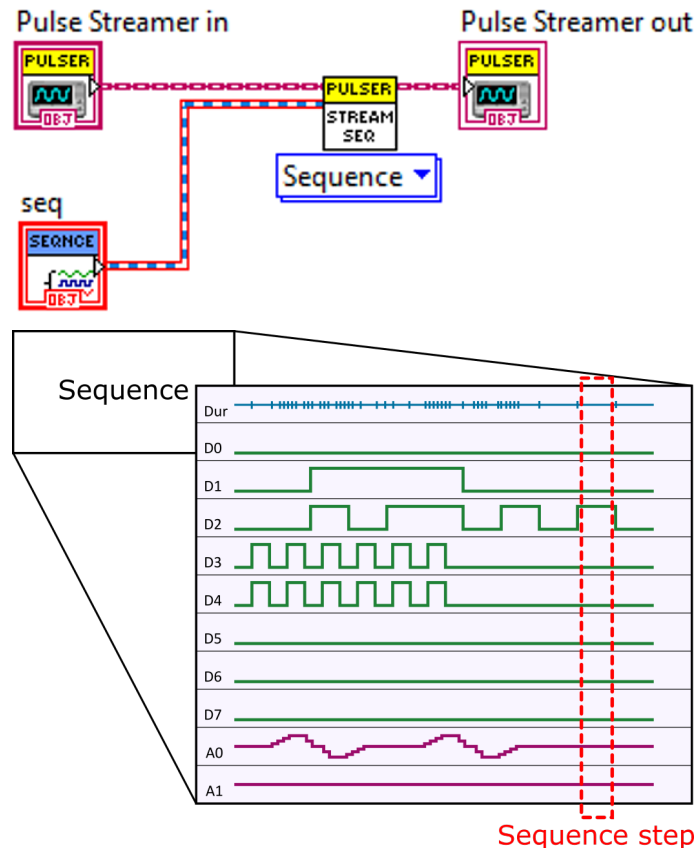
Python

Matlab

LabVIEW

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

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



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.

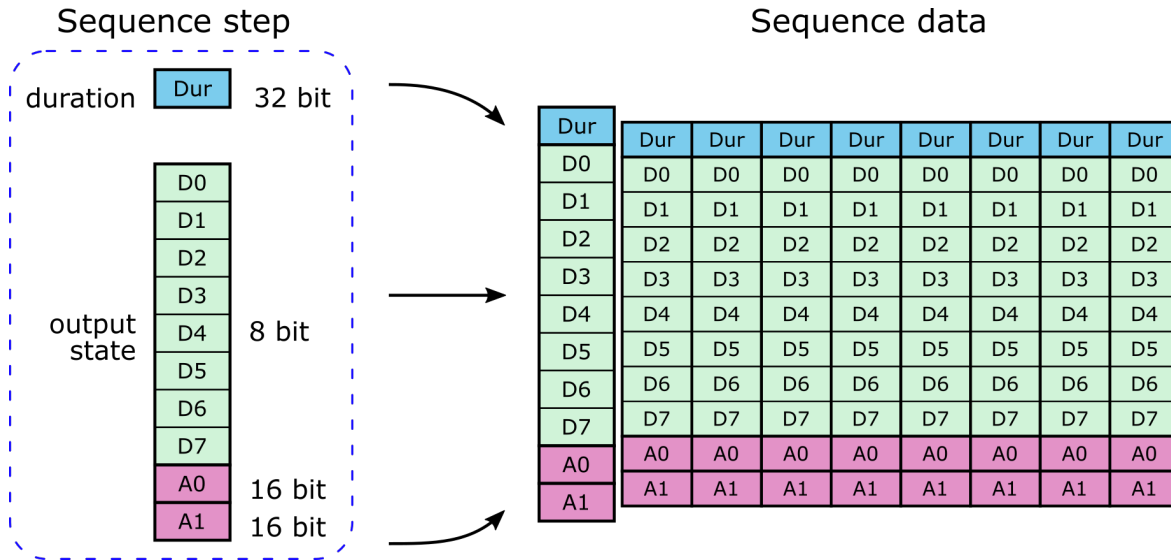
**Note:** Internally, the Pulse Streamer hardware is always splitting the sequence data into 8 nanosecond long chunks. When a sequence is shorter than 8 ns or its length is not an exact multiple of 8 ns the extra time will be padded to complete the last chunk. You can observe the effects of such padding if you try to stream a short pulse repetitively.

Example 1. Your sequence consists of 3 ns high-level and 2 ns low-level and you stream it with infinite repetitions, the resulting signal will have 3 ns high-level but 5 ns low-level. Therefore, the actual pulse frequency will be 125 MHz instead of 200 MHz. For continuous periodic signals, you can solve this problem by creating a sequence of repetitive pulses that has a duration which is multiple of 8 ns. One easy way to guarantee that sequence duration is a multiple of 8 ns is to repeat it 8 times using `Sequence.repeat()` method, which will repeat the sequence data in PC memory before sending it to the Pulse Streamer hardware.

Example 2. You want to stream a sequence that is 12345 ns long and you want to repeat it infinitely by setting `n_runs=-1`. Since this sequence duration is not a multiple of 8 ns ( $12345 \text{ ns} / 8 \text{ ns} = 1543.125$ ) the Pulse Streamer will allocate 1544 chunks, and the actual sequence duration will be  $1544 * 8 \text{ ns} = 12352 \text{ ns}$ , or 7 ns longer.

### 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 the 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 Module level functions

`findPulseStreamers(search_serial="")`

**Parameters**

`search_serial` (*str*) – Pulse Streamer serial number as a string.

**Returns**

List of *DeviceInfo* objects.

This function searches and returns basic information about discovered Pulse Streamers. If non-empty `search_serial` string is provided, then information is returned only for a specific Pulse Streamer unit.

The returned value is a list of *DeviceInfo* objects containing the IP address and basic information.

**class DeviceInfo**

This class contains read-only information about the discovered Pulse Streamer.

Property name	Example data	Description
ip	“192.168.0.2”	Device IP address
serial	“00:26:32:f0:3b:1b”	Device serial number
hostname	“pulsestreamer”	Pulse Streamer hostname
model	“Pulse Streamer 8/2”	Pulse Streamer model name
fpgaid	“123456789ABCD”	FPGA ID number
firmware	“1.2.0”	Firmware version
hardware	“1.3”	Hardware version

The discovery algorithm sends Pulse Streamer specific query packets overall available and active network interfaces and listens for responses from the connected Pulse Streamers.

---

**Note:** The `findPulseStreamers()` is capable of finding the devices and reporting their IP addresses even in the networks without dynamic IP assignment by a DHCP server or an improper IP address configuration. Therefore, it might happen that the reported Pulse Streamer IP is not accessible from your network. For example, when the reported IP is `169.254.8.2` (static fallback) and your PC is configured as `192.168.1.2`, you will not be able to connect to the Pulse Streamer. This is because of the way how IP networks operate. However, you will still be able to discover this Pulse Streamer and learn its IP, which is very helpful for identifying network connection problems.

---

## 4.3 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 an incorrect value or the 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, and 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.

#### `reboot()`

Perform a soft reboot of the device without power-cycling.

#### `createSequence()`

Use this method to create a new hardware-specific `Sequence` object. A hardware-specific sequence object has 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. The generic *Sequence* object's normal behavior is to check hardware limits only on the call to *PulseStreamer.stream()* method.

#### Returns

Hardware-specific *Sequence* object.

### 4.3.1 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 retrigger 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)`.

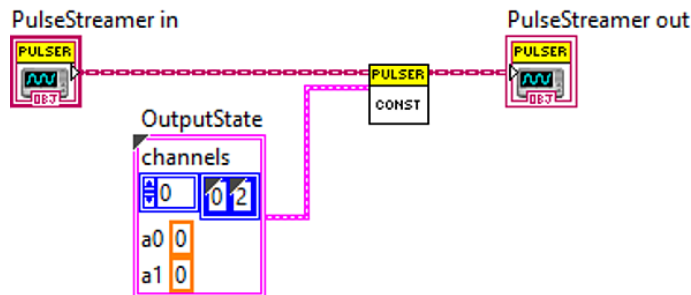
Python

Matlab

LabVIEW

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

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



#### Parameters

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



### 4.3.2 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_INFINITY[,
                        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: (*duration\_ns*, [*channels\_to\_set\_HIGH*], *analogV\_0*, *analogV\_1*)

Python

Matlab

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

```
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_INFINITY=-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=TriggerStart.HARDWARE\_RISING* (rising edge on the trigger input), *start=TriggerStart.HARDWARE\_FALLING* (falling edge on the trigger input) or *start=TriggerStart.HARDWARE\_RISING\_AND\_FALLING* (both edges are active).

If the automatic rearm functionality is enabled (*rearm=TriggerRearm.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=TriggerRearm.MANUAL*.

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

#### Parameters

- **start** (`TriggerStart`) – Defines the source of the trigger signal
- **rearm** (`TriggerRearm`) – Enables or disables trigger automatic rearm.

**PulseStreamer.getTriggerStart()**

Queries the hardware for the currently active trigger start condition.

**Returns**

Returns *TriggerStart* value.

**PulseStreamer.getTriggerRearm()**

Queries the hardware for the currently active rearming method.

**Returns**

Returns *TriggerRearm* value.

**PulseStreamer.startNow()**

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

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

If the *start* is *TriggerStart.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.

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

### 4.3.3 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.

### 4.3.4 Using an external clock

The Pulse Streamer can be clocked from three different clock sources. By default, the internal clock of the device is used. It is also possible use 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.

**Parameters**

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

`PulseStreamer.getClock()`

Returns a `ClockSource` element with the clock source currently used by the Pulse Streamer.

**Returns**

`ClockSource` Current clock source

Also, you can apply a continuous square wave of 125MHz to the dedicated Pulse Streamer output channels as an external clock signal for other devices to be synchronized with the Pulse Streamer.

`PulseStreamer.setSquareWave125MHz(channels=[])`

Set a persistent square wave with a frequency of 125 MHz to the selected digital outputs. The 125 MHz will remain and will not be affected by any other settings applied to this channel unless the corresponding channel is deselected via `setSquareWave125MHz()` or the method `reset()` is called. A call to this method without a parameter will disable the 125 MHz signal on all channels it was enabled before.

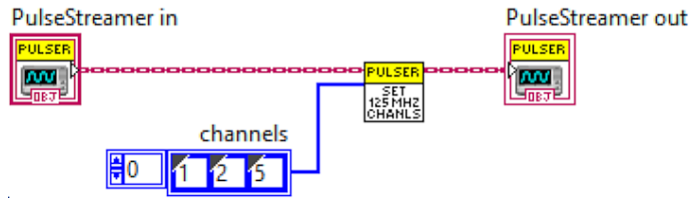
Python

Matlab

LabVIEW

```
ps.setSquareWave125MHz(channels=[1, 2, 5])
```

```
ps.setSquareWave125MHz([1, 2, 5])
```



#### Parameters

**channels** (*list*) – defines to which channels the 125 MHz square wave should be applied.

### 4.3.5 Hardware identification

`PulseStreamer.getSerial()`

#### Returns

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

`PulseStreamer.getFPGAID()`

#### Returns

String containing FPGA ID number.

`PulseStreamer.getFirmwareVersion()`

#### Returns

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

`PulseStreamer.getHardwareVersion()`

#### Returns

String containing the hardware revision number of the connected Pulse Streamer.

`PulseStreamer.setHostname(hostname)`

#### Parameters

**hostname** (*str*) – Hostname string to set for the connected Pulse Streamer.

Set hostname of the connected Pulse Streamer.

---

**Note:** Depending on your network environment, this setting may not affect how your Pulse Streamer is identified in the network. However, the stored hostname will be returned when you call `getHostname()`.

---

`PulseStreamer.getHostname()`

#### Returns

String containing the hostname of the connected Pulse Streamer.

### 4.3.6 Calibrating the analog outputs

Pulse Streamer 8/2 devices shipped with firmware version v1.3.0 or later (published in July 2020) already include calibration data for analog outputs, and no further user action is required. Devices shipped with older firmware require analog output calibration in order to achieve specified accuracy, see *Hardware*. You can perform this calibration yourself by following the steps described below. The calibration requires a sufficiently accurate multimeter (not an oscilloscope) connected to the analog outputs (can be done one channel at a time).

#### Calibration procedure

1. Connect the multimeter to the analog output of the Pulse Streamer. The measurement has to be performed at 50 Ohm load, so you will need to attach 50 Ohm termination.
2. Using the Pulse Streamer API, set the analog output to several values and record multimeter readings. This has to be done at least for -0.9 and +0.9 output values.

3. Calculate the slope

$$\text{slope} = \frac{\text{voltage}_{+0.9V} - \text{voltage}_{-0.9V}}{1.8}$$

4. Calculate the offset

$$\text{offset} = \text{voltage}_{+0.9V} - \text{slope} * 0.9$$

5. Perform the steps 1 to 4 for each analog output.
6. Call the `setAnalogCalibration()` and specify both offsets and slopes.
7. Reboot the Pulse Streamer to apply the new calibration data (from firmware v1.5.0 on, the device is rebooted automatically).

**Warning:** If you perform repeated calibration, you have to reset the slope and offset to values 1 and 0, respectively. Failing to do so will lead to invalid calibration data.

```
PulseStreamer.setAnalogCalibration(dc_offset_a0=0, dc_offset_a1=0, slope_a0=1,
                                   slope_a1=1)
```

Sends the DC-offset and slope of each analog channel to the Pulse Streamer and stores it to internal memory. These values will be applied after reboot. With firmware version v1.5.0 or later, this is done automatically. If you use a former firmware of the Pulse Streamer 8/2, you will have to power cycle your device.

#### Parameters

- **dc\_offset\_a0** – The DC offset of analog channel 0
- **dc\_offset\_a1** – The DC offset of analog channel 1
- **slope\_a0** – The gradient of the transfer function of analog channel 0
- **slope\_a1** – The gradient of the transfer function of analog channel 1

If you need help during the calibration procedure, please contact [support@swabianinstruments.com](mailto:support@swabianinstruments.com).

```
PulseStreamer.getAnalogCalibration()
```

Returns the stored calibration values of your Pulse Streamer. These values will be rounded according to the DAC resolution. If you call this method immediately after `setAnalogCalibration()`, the returned data will not reflect the actual calibration state.

**Returns**

structure of the four calibration values rounded to the DAC resolution.

### 4.3.7 Modify the network configuration

By default, the Pulse Streamer 8/2 will attempt to acquire an IP address via DHCP. If you want to assign a specific IP address to your Pulse Streamer, you can disable DHCP and configure a static IP instead. We recommend using our graphical user interface (Windows only) to modify the Pulse Streamer 8/2 network configuration. For more information, please have a look at *Network Connection*. Alternatively, the following methods allow you to configure the device's network settings via our client software interfaces.

`PulseStreamer.setNetworkConfiguration(dhcp, ip="", netmask="", gateway="", testmode=True)`

Enables DHCP or sets static IP address, Netmask and Standard Gateway. If `testmode=True`, the configuration is applied temporarily. Power-cycling will restore the former configuration. If `testmode=False`, the configuration will be applied permanently and the device is rebooted.

**Parameters**

- **dhcp** (*bool*) – DHCP enable/disable True/False
- **ip** (*str*) – static IP address (If `dhcp=True`, this value is ignored)
- **netmask** (*str*) – Netmask for static IP address configuration (If `dhcp=True`, this value is ignored)
- **gateway** (*str*) – Standard gateway for static IP address configuration (If `dhcp=True`, this value is ignored)
- **testmode** (*bool*) – If True, the configuration is applied temporarily. Power-cycling will restore the former configuration. If False, the configuration will be applied permanently and the device is rebooted.

`PulseStreamer.getNetworkConfiguration(permanent=False)`

**Parameters**

**permanent** (*bool*) – If True, the method returns network settings stored in the device's configuration file. If False, the method returns the current network settings of the device.

**Returns**

structure of the current or stored network settings

`PulseStreamer.applyNetworkConfiguration()`

Applies current (and successfully tested) network configuration permanently and the device is rebooted.

### 4.3.8 Enumerations

`class ClockSource(enumeration)`

This enumeration describes the selectable clock sources of the Pulse Streamer

**INTERNAL**

Use internal clock generator (default)

**EXT\_125MHZ**

Use external 125 MHz clock signal

**EXT\_10MHZ**

Derive clock from external 10 MHz reference signal

**class TriggerStart**(*enumeration*)

This enumeration describes the selectable start modes of the Pulse Streamer

**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 the falling edge.

**HARDWARE\_RISING\_AND\_FALLING**

External trigger on rising and falling edges.

**class TriggerRearm**(*enumeration*)

This enumeration describes the rearm functionality of the Pulse Streamer

**AUTO**

Trigger immediately after a sequence is uploaded. (default)

**MANUAL**

Trigger once only and do not rearm automatically. Rearm manually via the `rearm()` method.

## 4.4 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, successive mappings to the same channel will 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** (*list[int]*) – Digital channel number(s) as labeled on the Pulse Streamer connectors panel.
- **pattern** (*list*) – A pattern to be assigned to the *channel*.

**invertDigital**(*channels*)

Inverts level values in the stored pattern for the specified *channel*.

```
sequence.setDigital(1, [(10, 0), (20, 1), (80, 0)])
sequence.invertDigital(1)
# is equivalent to
sequence.setDigital(1, [(10, 1), (20, 0), (80, 1)])
```

**Parameters**

**channel** (*int*) – Digital channel number.

**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** (*list[int]*) – Analog channel number(s) as labeled on the Pulse Streamer connectors panel.
- **pattern** (*list*) – A pattern to be assigned to the *channel*.

**invertAnalog**(*channel*)

Inverts level values in the pattern for the specified *channel*.

```
sequence.setAnalog(0, [(100, -0.1), (200, 0), (800, 0.5)])
sequence.invertAnalog(0)
# is equivalent to
sequence.setAnalog(0, [(100, 0.1), (200, 0), (800, -0.5)])
```

**Parameters**

**channel** (*int*) – Analog channel number.



### 4.4.1 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 an *OutputState* 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.

### 4.4.2 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

Matlab

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

```
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 object, `sequence1` and `sequence2`, 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 the concatenation operator. This allows for transparent use of any function of these languages that implicitly use concatenation. See the example code for each language.

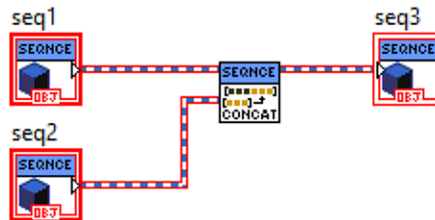
Python

Matlab

LabVIEW

```
seq3 = seq1 + seq2
```

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



### Returns

Returns a new `Sequence` object with concatenated data.

## 4.4.3 Visualization

`Sequence.plotDigital()`

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

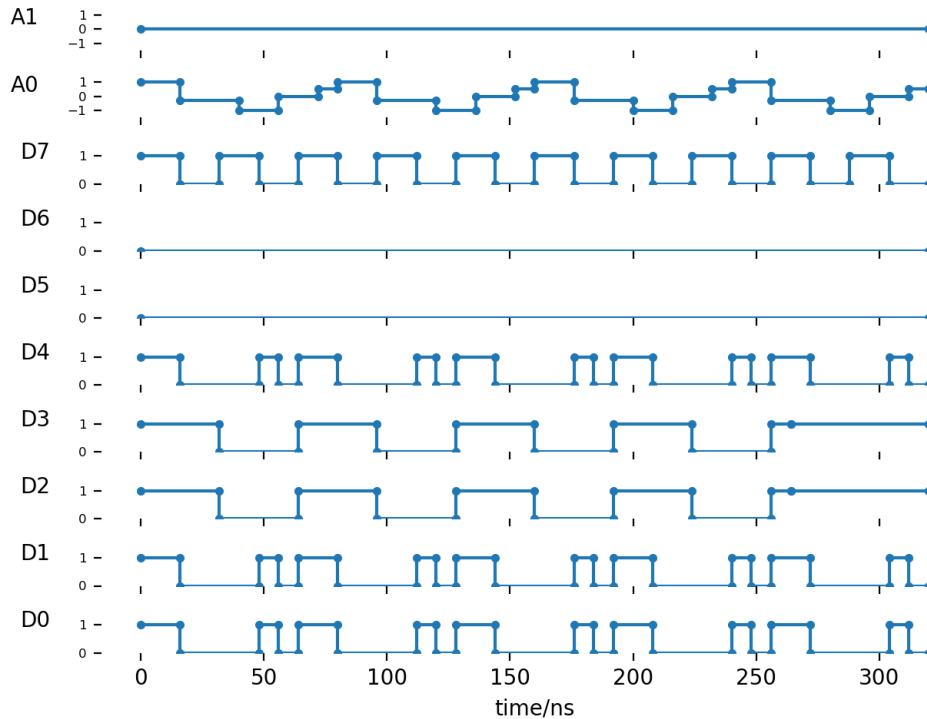
`Sequence.plotAnalog()`

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.




---

**Note:** (Python only)

Since the Python Client release v1.6.1, importing the module `matplotlib` for `plot()` is optional. If you want to use visualization with Python, please ensure you have the package `matplotlib` installed.

---

## 4.5 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)`.

## 4.6 Advanced (Beta) features

### 4.6.1 Synchronized Pulse Streamer (Python only)

With our [programming examples](#) in Python, we provide the class `SyncPulseStreamer` as a wrapper class for the client interface of the Pulse Streamer 8/2. It combines two Pulse Streamer 8/2 (requires firmware version v1.4.0 or later) and is currently only available for the Python client.

#### Synchronization concept and setup

One Pulse Streamer master generates the clock signal and trigger for one Pulse Streamer slave. The only necessary preparation is that digital channel 6 of the master must be connected to the external clock input of the slave as well as digital channel 7 of the master must be connected to the trigger input of the slave. To avoid race conditions between the trigger and trigger-sampling clock-edge, we recommend using cables of equal length.

The features of the resulting `SyncPulseStreamer` object can be described as follows:

- 14 digital channels (6 of the master, 8 of the slave)
- 4 analog channels (2 in each case master/slave)
- The slave is delayed by a constant time offset of  $\sim 70\text{ns}$  (internal + cable)
- Increased ( $\sim x * \sqrt{2}$ ) RMS jitter of the 8 digital channels of the slave ( $< 75\text{ ps}$ )
- In this configuration, devices with hardware Version 2.x (devices before 2021) have a 100mV ripple on the digital channels of the slave due to the external clock signal. The analog outputs are not affected.

#### Usage and sequence generation

The Pulse Streamer synchronization wrapper offers the same API-structure as the original Pulse Streamer client if possible. For a detailed description of the Pulse Streamer 8/2 API, please have a look at the [Programming interface](#) section and the provided [Python example](#).

When it comes to sequence generation, unlike the original method `createSequence()`, the equivalent method of the sync-wrapper returns two sequence objects, one for the Pulse Streamer master and one for the Pulse Streamer slave. You can set the digital and analog channels of both sequences independently. Setting Channel 6 (clock signal) of the master will be ignored, and channel 7 (trigger) of the master will be overwritten by the stream method).

If you want to use a parameter `n_runs > 1` with the `stream()` function, you should ensure that the two sequences are of equal duration. Therefore the `stream()` prints a warning message if the duration of the two sequences differs.

To compensate for the delay of the Pulse Streamer slave, you can add an empty pulse as a first pulse to each channel of the master sequence, as it is shown in the Python code example. In that case, you have to take into account that if you want to use `n_runs > 1`, you have to subtract the delay from the last pulses of the master-sequence, or you can accordingly pad the slave-sequence. The drawback is that this either requires the same state at the beginning and the end of the channel pattern, or you have to deal with the padding. Furthermore, you have to take into account that when a sequence is given to the method `stream()` of the Pulse Streamer or to the transformation-methods (see [Transformation](#)) of the `Sequence` class, all channels are padded with the last value to the longest channel duration. For further information, please have a look at the sections [Creating sequences](#) and [Sequence](#).

Just to give you the complete information of the padding issue: Due to the internal design of the Pulse Streamer, the device itself will pad the last sequence step to a duration that the sequence duration as a whole is a multiple of 8ns (which means a prolongation between 0 and 7ns). This step is executed before the parameter `n_runs` is applied (see [Streaming](#)).

### Further Limitations

1. If you use the synchronized Pulse Streamer with a fast external trigger, it is possible that the Pulse Streamer master is ready for retriggering and the slave is not and vice versa. Therefore, you should always poll the method `hasFinished()` before retriggering the Pulse Streamer with the external trigger.
2. If you stream a second pulse pattern into an already running sequence, you previously should set the Pulse Streamer to a constant state by using the methods `constant()` or `reset()`. Otherwise, the new sequence of the Pulse Streamer slave might already be triggered by the still running stream of the Pulse Streamer master.



## LOW-LEVEL RPC INTERFACE

This section introduces the low-level RPC interface of the Pulse Streamer. You can use this information to understand how to control the Pulse Streamer hardware or to develop your own communication programs. We recommend making use of the API described in the section *Programming interface* that significantly simplifies the pulse pattern generation.

### 5.1 Pulse Sequences

---

**Note:** The code examples provided in this section are in C++ and naming conventions can differ from the *Programming interface* section. The data format given here applies only to the *Pulse Streamer 8/2*.

---

Pulse sequences are represented as a 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` are 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 0 V, 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 0 V. If you set the device to a constant output, an eventually currently streamed sequence is stopped. It is not possible to retrigger 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}
            )
```

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 value of tick is 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 retrigger 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 retrigger 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 finishing 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 set in the current sequence, the output of Pulse Streamer will change to (or stay in) the last known constant state.

## 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, you can call the method

```
std::string getSerial()
```

The method returns a string containing the serial number/MAC-address of your Pulse Streamer 8/2.

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

```
std::string getFPGAID()
```

The method returns a hexadecimal string containing the ID-number of the built-in FPGA.

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 straightforward “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 Google’s 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. You can find more information 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 Google's 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 of the 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;
    }
}
```

(continues on next page)

(continued from previous page)

```
    EXT_10MHZ = 2;
}
Clocking clock_source = 1;
}

message PulseStreamerReply {
    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 (VoidMessage) returns (PulseStreamerStringReply) {}
    rpc getFPGAID (VoidMessage) returns (PulseStreamerStringReply) {}
}
```

## CHANGELOG

### 6.1 2023-06-01

#### 6.1.1 Firmware update v1.7.2

This firmware brings updates of third-party components such as Linux kernel etc. \* Increased upload performance

#### 6.1.2 User interface v1.7.0

- Fixed a rare bug that made Pulse Streamer Application fail when discovering devices
- Fixed a rare bug that made Pulse Streamer Application fail when establishing a connection

#### 6.1.3 Python v1.7.0 & Matlab v1.7.0 & LabView v1.7.0

- Support firmware v1.7.x

### 6.2 2023-04-03

#### 6.2.1 User interface v1.6.4

- Added optional logging support
- In case of failure, Pulse Streamer Application sends an error message to Swabian Instruments
- Fixed bug in loading arbitrary data from CSV-file

### 6.3 2023-03-08

#### 6.3.1 Python Client update v1.6.2

- Fixed a bug that connection fails due to the post-release segment of Python package v1.6.1.post1

## 6.4 2023-02-27

### 6.4.1 User interface v1.6.3

- Fixed a bug that Pulse Streamer Application fails to connect without an internet connection.

## 6.5 2023-02-16

### 6.5.1 User interface v1.6.2

- Pulse Streamer Application shows proper messages in the case of an error and will controllably shut down.
- Package and .NET Framework update

### 6.5.2 Python Client v1.6.1

- Fixed a bug in the sequence creation process that, under some conditions, resulted in missing pulses of a sequence with pulses with a maximum duration value of a 32 Bit word.
- Updated package dependency of package protobuf
- Changed import of module matplotlib to optional

## 6.6 2022-10-05

### 6.6.1 Matlab Client update v1.6.2

- Fixed a bug that, under some conditions, resulted in fixed pulse duration of ~4.2 s when the requested pulse duration was longer than ~2.1 s.

## 6.7 2022-05-02

### 6.7.1 Firmware update v1.6.3

- Support hardware revision v3.3

## 6.8 2022-02-28

### 6.8.1 User interface v1.6.1

- Pulse Streamer Application handles an incorrect setting of sequence parameters with warning

## 6.9 2021-12-20

### 6.9.1 Firmware update v1.6.2

This update overcomes the recommended maximum limit of the retrigger frequency and brings automatic check and repair of sd-card partitions during bootup.

- No limitation of the trigger frequency except a fix retrigger dead-time (<50 ns)
- solved: Very rare boot failure due to sd-card partition damages automatically handled by fsck-tools during bootup
- solved: A sporadic case of missing firmware/hardware version information in Pulse Streamer discovery server information
- solved: Incorrect handling of empty sequences and n\_runs=0 in v1.6.0 (internal release only)

### 6.9.2 Clients

#### Python v1.6.0

- Increased performance of sequence generation
- solved: zoom-in/panning within the sequence plot window

#### Matlab v1.6.1 & LabView v1.6.1

- Support firmware v1.6.2

## 6.10 2021-08-31

### 6.10.1 Firmware update v1.5.2

- solved: Server-side JSON-RPC request-id handling

This release resolves the packet dependency (tinyrpc<=1.1.0) of the Python client

## 6.11 2021-08-23

### 6.11.1 Client update v1.5.2

#### Python

- Fix packet version dependency: tinyrpc <=1.1.0 for Pulse Streamer 8/2 firmware <=1.5.1

## 6.12 2021-07-28

### 6.12.1 User interface v1.5.3

- solved: Correct firmware v1.0.x identification via Pulse Streamer Application

## 6.13 2021-05-20

### 6.13.1 User interface v1.5.2

- solved: Firmware v0.9 detection via Pulse Streamer Application
- solved: Detection of non-licensed devices via Pulse Streamer Application

## 6.14 2021-03-12

### 6.14.1 Firmware update v1.5.1

- solved: extraordinarily long pulse on analog channels of few devices during bootup with hardware version 3.1 and firmware version v1.5.0

#### User interface v1.5.1

- solved: Pulse Streamer Application crashes when trying to update the firmware without an internet connection available
- Firmware update can also be performed on a manually downloaded local updater file
- Pulse Streamer Application brings the opportunity to reconnect to different devices without restarting the application.

## 6.15 2021-02-12

### 6.15.1 Firmware/Client update v1.5.0

This update brings an improved Pulse Streamer Application (GUI).

- API has been supplemented by the methods: `reboot()`, `setNetworkConfiguration()`, `getNetworkConfiguration()`, `applyNetworkConfiguration()`
- solved: rare error in case of a sequence with one sequence step, RLE-value  $\leq 8$ ns and `n_runs=INFINITE`
- Matlab and LabVIEW clients are now available from online repositories via *Matlab Addon Explorer* and *JKI VI Package Manager*, respectively.



## User interface

- Pulse Streamer Application makes use of the full API extensions since version v1.0.2
- Pulse Streamer Application makes use of the device discovery functionality
- Pulse Streamer Application provides functionality for network configuration
- Pulse Streamer Application performs firmware update process

## 6.16 2020-11-12

### 6.16.1 Firmware/Client update v1.4.0

This update brings some functionality as output port enabling after power-cycling or automatic hardware version detection for the new hardware version v3.1 of the Pulse Streamer 8/2.

- API has been supplemented by the methods: `getHardwareVersion()`, `setSquareWave125MHz()`
- New trigger input stage leads to a typical TriggerToData of 65 ns (hardware version 3.1) respectively 60 ns (hardware version  $\leq 2.3$ )

## 6.17 2020-08-17

### 6.17.1 Bug-fix in LabView client

- Internal sequence data for binary protocol uses little-endian encoding compared to big-endian of the JSON-RPC. This resulted in incorrect signal generation with binary protocol enabled firmware versions.

In order to fix the problem, you have to update the LabView client to the latest version.

## 6.18 2020-07-27

### 6.18.1 Firmware/Client update v1.3.0

This update brings the opportunity to calibrate the analog outputs to increase its accuracy. Devices shipped with firmware version v1.3.0 or later come with calibrated outputs. Devices which has been shipped with a previous firmware version can manually be calibrated by the user with a dedicated method.

- API has been supplemented by the methods: `setAnalogCalibration()`, `getAnalogCalibration()`
- solved: sequences with sequence steps longer than  $\sim 0.5 \mu\text{s}$  and a number of sequence steps near to the maximum limit could raise `std::bad_alloc()`

## 6.19 2020-01-20

### 6.19.1 Firmware/Client update v1.2.0

This update brings new device discovery functionality that greatly simplifies finding and connecting to the Pulse Streamer. Moreover, new getter methods are added to form a more complete set of functions that allows you to set and query the device state. Existing functionality has also received an upgrade in performance. Now sequence upload happens 2x faster thanks to a new binary communication protocol that works along the JSON-RPC.

- new network device discovery functionality, `findPulseStreamers()`
- sequence upload performance increased by factor two (requires both firmware and client interface update)
- API has been supplemented by the methods: `setHostname()`, `getHostname()`, `getTriggerStart()`, `getTriggerRearm()`, `getClock()`
- recommended maximum external retrigger frequency increased to 1 kHz
- minimum trigger pulse width reduced to < 2ns
- TriggerToData increased to 64.5/65.5 ns (mean value, rms jitter 2.3 ns)
- unified version numbers for the Pulse Streamer 8/2 firmware and the client interfaces that now share the first two numerals

## 6.20 2019-08-07

### 6.20.1 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)

## 6.21 2019-05-10

### 6.21.1 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.

## 6.22 2019-04-23

### 6.22.1 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](http://www.pypi.org). You can now install it with `pip install pulsestreamer`.
- Matlab: minor code cleanup.

## 6.23 2019-03-01

### 6.23.1 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 backward-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_INFINITY = -1` for infinite sequence repetition.
- `Sequence` object now applies padding to the pattern data and 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.
- 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 the 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 a cleaner layout.
- `PulseStreamer.stream` method accepts `Sequence` object as input parameter directly.

## 6.24 2018-12-17

### 6.24.1 Firmware update v1.0.2

- solved: occasionally missed external trigger

## 6.25 2018-11-09

### 6.25.1 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

### 6.25.2 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 the 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 the new way of building sequences

## 6.26 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

## 6.27 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

## 6.28 2017-05-07

clients

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

documentation

- added 'Getting Started' section

## 6.29 2016-04-08

Matlab client

- added links to the Matlab client examples

## 6.30 2016-03-17

static sequence beta 0.9

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

## 6.31 2016-03-07

provide network configuration

- added section on network configuration

## 6.32 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

## 6.33 2016-02-02

static sequence alpha





## 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: v1.6.x - firmware: v1.6.x
- client: v1.5.x - firmware: v1.5.x
- client: v1.4.x - firmware: v1.4.x
- client: v1.3.x - firmware: v1.3.x
- client: v1.2.x - firmware: v1.2.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



## INDICES AND TABLES

- genindex
- search



## A

applyNetworkConfiguration() (*PulseStreamer* method), 34  
 AUTO (*TriggerRearm* attribute), 35

## B

built-in function  
   findPulseStreamers(), 26

## C

ClockSource (*built-in class*), 34  
 concatenate() (*Sequence static method*), 37  
 constant() (*PulseStreamer method*), 28  
 createSequence() (*PulseStreamer method*), 27

## D

DeviceInfo (*built-in class*), 26

## E

EXT\_10MHZ (*ClockSource attribute*), 34  
 EXT\_125MHZ (*ClockSource attribute*), 34

## F

findPulseStreamers()  
   built-in function, 26  
 forceFinal() (*PulseStreamer method*), 30

## G

getAnalogCalibration() (*PulseStreamer method*), 33  
 getClock() (*PulseStreamer method*), 31  
 getData() (*Sequence method*), 37  
 getDuration() (*Sequence method*), 37  
 getFirmwareVersion() (*PulseStreamer method*), 32  
 getFPGAID() (*PulseStreamer method*), 32  
 getHardwareVersion() (*PulseStreamer method*), 32  
 getHostname() (*PulseStreamer method*), 32  
 getLastState() (*Sequence method*), 37  
 getNetworkConfiguration() (*PulseStreamer* method), 34  
 getSerial() (*PulseStreamer method*), 32  
 getTriggerRearm() (*PulseStreamer method*), 30

getTriggerStart() (*PulseStreamer method*), 30

## H

HARDWARE\_FALLING (*TriggerStart attribute*), 35  
 HARDWARE\_RISING (*TriggerStart attribute*), 35  
 HARDWARE\_RISING\_AND\_FALLING (*TriggerStart* attribute), 35  
 hasFinished() (*PulseStreamer method*), 31  
 hasSequence() (*PulseStreamer method*), 31

## I

IMMEDIATE (*TriggerStart attribute*), 35  
 INTERNAL (*ClockSource attribute*), 34  
 invertAnalog() (*Sequence method*), 36  
 invertDigital() (*Sequence method*), 36  
 isEmpty() (*Sequence method*), 37  
 isempty() (*Sequence method*), 37  
 isStreaming() (*PulseStreamer method*), 31

## M

MANUAL (*TriggerRearm attribute*), 35

## O

OutputState (*built-in class*), 39  
 OutputState() (*OutputState method*), 39

## P

plot() (*Sequence method*), 38  
 plotAnalog() (*Sequence method*), 38  
 plotDigital() (*Sequence method*), 38  
 PulseStreamer (*built-in class*), 27  
 PulseStreamer() (*PulseStreamer method*), 27

## R

rearm() (*PulseStreamer method*), 30  
 reboot() (*PulseStreamer method*), 27  
 repeat() (*Sequence static method*), 37  
 reset() (*PulseStreamer method*), 27

## S

selectClock() (*PulseStreamer method*), 31

Sequence (*built-in class*), 35  
Sequence() (*Sequence method*), 35  
setAnalog() (*Sequence method*), 36  
setAnalogCalibration() (*PulseStreamer method*), 33  
setCallbackFinished() (*PulseStreamer method*), 30  
setDigital() (*Sequence method*), 35  
setHostname() (*PulseStreamer method*), 32  
setNetworkConfiguration() (*PulseStreamer method*), 34  
setSquareWave125MHz() (*PulseStreamer method*), 31  
setTrigger() (*PulseStreamer method*), 29  
SOFTWARE (*TriggerStart attribute*), 35  
startNow() (*PulseStreamer method*), 30  
stream() (*PulseStreamer method*), 29

## T

TriggerRearm (*built-in class*), 35  
TriggerStart (*built-in class*), 35

## Z

ZERO (*OutputState attribute*), 39