Pulse Streamer 8/2 Documentation

Release 1.0.1

Swabian Instruments

Nov 15, 2018

CONTENTS


Swabian instruments

CHAPTER

ONE

GETTING STARTED

1.1 Software installation

The Pulse Streamer does not require any driver or software installation. The interface of the Pulse Streamer is standard Ethernet, so all drivers are provided with your operating system.

The only requirement is that the Pulse Streamer is available within your network, see the *Network Connection* section for further information.

1.2 Programming guide and examples, basic GUI

To get used to the programming interface (API), various examples for Matlab, LabVIEW and Python are provided within the example folder of the installation directory. PulseStreamer-1.0.1.exe

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

CHAPTER

TWO

HARDWARE

2.1 Output Channels

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

2.1.1 Digital Output

Property	Value
Voltage level	0 to 3.3 V
Output drive	50 Ω
Sampling rate	1 GHz
Bandwidth	300 MHz

2.1.2 Analog Output

Property	Value
Output Voltage Range	-1.0 to 1.0 V
Output drive	50 Ω
Sampling rate	125 MHz
Bandwidth	80 MHz

2.2 Trigger Input

The Pulse Streamer 8/2 has one external trigger input, which can be enabled by software. By default, the Pulse Streamer is automatically rearmed after a sequence with a finite number of n_runs has finished. That means if another trigger occurs after the sequence has finished, the sequence is running again. We recommend, not to exceed 100 Hz as an external retrigger frequency. Information about how to configure the trigger functionality of the Pulse Streamer can be found in the *Programming Interface* section.

2.3 External Clock Input

The Pulse Streamer 8/2 has one input that can be applied to an external 125MHz clock, or to an external reference cock (10MHz). Further information about how to set the clock-source of the Pulse Streamer can be found in the *Program-*

ming Interface section. Before using the external clock the first time, please contact support@swabianinstruments.com

CHAPTER

THREE

NETWORK CONNECTION

In order to communicate with the Pulse Streamer, you need to know its IP. By default, the Pulse Streamer will attempt to acquire an IP address via DHCP.

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 of the device.

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 Use 169.254.8.2/16 permanent static IP

The Pulse Streamer is always reachable via a permanent second static IP-address 169.254.8.2/16. You can connect via this address by directly plug the device to your computer. Maybe you will have to reboot Windows to detect the Pulse Streamer, if there has been a DHCP-connection before.

3.3 Modify the network settings

Requirements:

- network access to your Pulse Streamer (e.g. 169.254.8.2/16 permanent static fallback)
- ssh / putty

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

- Download the network configuration script
- Start windows shell (press Windows+R, type cmd and hit Enter)

- In the Windows shell select the folder with the setNetworkConf.bat file
- Run the script with hostname or IP address as parameter

  > setNetworkConf.bat pulsestreamer

  or in case the IP address should be used

  > setNetworkConf.bat IP_ADDRESS

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

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

CHAPTER

FOUR

PROGRAMMING INTERFACE

4.1 Quick start guide and Tutorials

To get used to the programming interface (API), various examples for Matlab, LabVIEW and Python are provided: PulseStreamer-1.0.1.zip

It is not required to install any drivers or other software due to the Ethernet network interface used to communicate with the Pulse Streamer. Alongside with the examples provided, please have a look at the following sections to get a deeper understanding of the principles the Pulse Streamer works.

4.2 Pulse Sequences

Pulse sequences are represented as one dimensional arrays of pulses. 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 pulse duration is specified in nanoseconds.

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

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

4.3 Resetting Pulse Streamer to constant outputs

You can reset the Pulse Streamer by using the C++ method

```
void reset()
```

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

4.4 Setting constant outputs

You can set the outputs to a constant state.

The C++ method is:

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

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

4.5 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 C++ method to run a pulse sequence is:

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

sequence represents the pulse sequence

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

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

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

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

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

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

```
void startNow()
```

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

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

```
void forceFinal()
```

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

4.6 More features

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

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

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

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

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

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

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

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

4.7 Communicating with the instrument

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

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

4.8 JSON-RPC Interface

JSON-RPC libraries are available for most software languages. More information can be found on the official website http://json-rpc.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.

4.8.1 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.

4.9 gRPC Interface

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

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

4.9.1 Sending Data over gRPC

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

The Pulse Streamer interface looks like this.

```
package pulse_streamer;

message VoidMessage {}

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

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

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

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

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

message PulseStreamerReply {
```

(continues on next page)

(continued from previous page)

```
    uint32 value = 1;
}

message PulseStreamerStringReply {
    string string_value=1;
}

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

You may want to check out the source file `pulse_streamer.proto`.

We recommend to check out the Python example for connecting to the gRPC interface `random_pulses_grpc.py`.

4.10 Advanced Pulse Streamer clients

We provide Pulse Streamer client classes for Python, Matlab and LabView with advanced functionality for convenient communication with the device. These classes cover all methods of the Pulse Streamer class specified above, to call them directly via RPC.

4.10.1 Python

Both in JSON-RPC and gRPC, there are Python-clients available for the Pulse Streamer. To create an instance of the client, you have to import the Pulse Streamer class from the corresponding Python module, together with the required enums for the specified method-parameters. In distinction from the C++-API, in Python we provide a conversion of the values for the digital and anlog channels. As a result, you can use two representation of a pulse

```
(ticks, digi, a0, a1)
```

This representation is equivalent to the C++ pulse-structure. The pulse duration 'ticks' is specified in nanoseconds. The lowest bit in the digital bit mask "digi" corresponds to channel 0, the highest bit to channel 7. A channel is high when its corresponding bit is 1 and low otherwise. The analog values span the full signed 16 bit integer range, i.e. -1.0 V corresponds to -0x7fff and 1.0 V corresponds to 0x7fff.

```
(ticks, channel_list, a0, a1)
```

This structure accepts a 32 bit integer value as ticks, channel_list is a list of channel numbers, which are dedicated HIGH and the analog values are accepted as float values in range of -1.0 to 1.0.

In addition to the Pulse Streamer client classes, we provide a handy Python module to create sophisticated sequences for your Pulse Streamer application. The Sequence class, defined in Sequence.py, enables you to create multiple

sequences and dedicate different and independent wave forms to the eight digital and the two analog channels. If you import the class Sequence from Sequence.py you can use the methods

```
setDigitalChannel(channel, channel_sequence):

setAnalogChannel(channel, channel_sequence):
```

channel is the number of the dedicated digital (0-7) or analog (0-1) channel. channel_sequence are list of tupels

```
(ticks, digital_value) #setDigitaChannel

(ticks, analog_value) #setAnalogChannel
```

The digital value can either be 1 or 0. The analog value is accepted as a float value in range of -1.0 to 1.0.

To get the complete sequence with all merged and combined channel_sequences you can call the method

```
getSequence():
```

It returns the sequences represented by a list of pulses (ticks, digi, a0, a1).

Alternatively, you can also set the sequence as a whole by calling

```
setOutputList(pulse_list):
```

pulse_list is a list of tuples

```
(ticks, channel_list, a0, a1)
```

channel_list is the list of channel numbers, which are dedicated high. The analog values are accepted as float values in range of -1.0 to 1.0.

For each sequence you can either set the channels via the setDigitalChannel()/setAnalogChannel() or via the setOutputList() method.

We recommend to check out the Python wrapper and Sequence class to use the Pulse Streamer in Python `pulse_streamer_jrpc.py`. `pulse_streamer_grpc.py`. `Sequence.py`.

4.10.2 MATLAB

For MATLAB users we provide a set of classes that wrap the communication with the Pulse Streamer and handle all required data conversion. In addition, the client provides high-level functions that simplify sequence creation and manipulation.

With the new approach the sequences can be created and streamed in the following steps:

1. Create desired signal patterns that you wish to output on each channel. A pattern is simply a cell array containing the levels and their durations. For digital signals the level must be either `true` or `false`, while analog values shall be specified in Volts from -1 to 1. Durations are expected to be in nanoseconds. For example:

```
digitalPattern = {20000, true; 50000, false; 10000, true; 10000, false};
analogPattern = {50000, 0.1; 50000, 0.2; 50000, 0.3; 50000, 0.4; 0, 0};
```

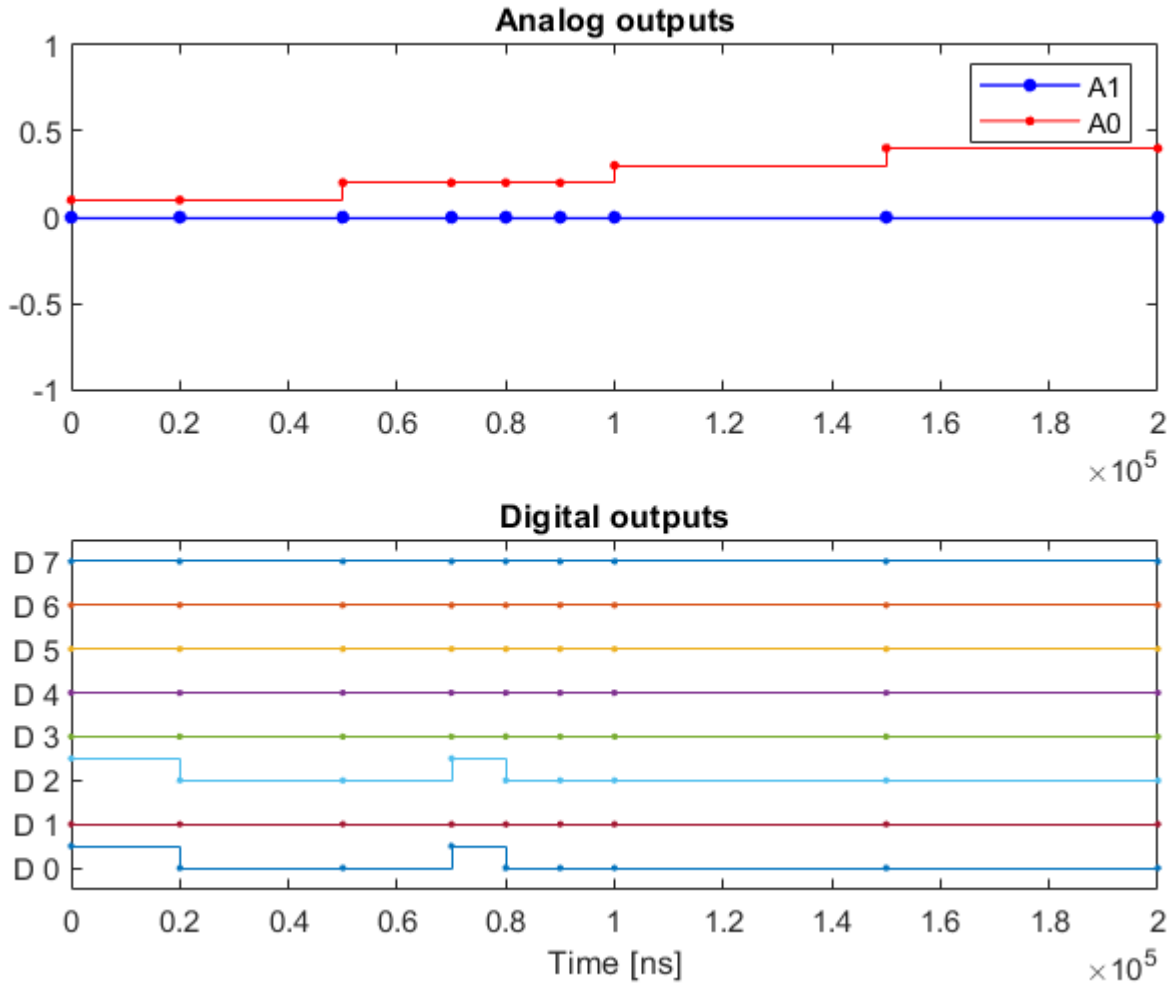
2. Create `PSSequenceBuilder` object and assign the patterns to the channels. The same pattern can be assigned to more than one channel. Pay attention that in case of repetitive assignment of a pattern to the same channel, only the last will be used for sequence creation.

```
% empty IP address allows to create PulseStreamer object
% without connecting to the hardware.
ps = PulseStreamer('');

% Create sequence builder object
builder = PSSequenceBuilder(ps);

% assign patterns to channels D0, D2 and A0
builder.setDigital(0, digitalPattern);
builder.setDigital(2, digitalPattern);
builder.setAnalog(0, analogPattern);
```

3. Build sequence from the assigned patterns. Sequence data can also be visualized for inspection.

```
% Build the sequence. The result is a PSSequence object
sequence = builder.buildSequence();

% You can also visualize the sequence data
plot(sequence); % same as: sequence.plot();
```

The following figure shall be shown



4. Send prepared sequence to the Pulse Streamer

```
nRuns = -1; % number of times to stream the sequence. -1: means repeat indefinitely
finalState = OutputState(0,0,0); % final state after the sequence streaming has␣
↪completed

% In the default state, the Pulse Streamer will start generating signals
% immediately after calling the "stream()" method.
ps.stream(sequence, nRuns, finalState);
```

Migration from earlier versions

The way how sequences are generated has changed in the v1.0. The `P` and `PH` classes are deprecated and their use is discouraged as they will be removed in the future. For easier migration to v1.0, we provide modified versions of these classes along with a function that converts `P` and `PH` objects to a `PSSequence` object.

To put it short. In order to make your code working with v1.0, you have to

1. Download and use the newest version of the client from www.swabianinstruments.com.
2. Set appropriate trigger settings using `setTrigger()` method of PulseStreamer object.
3. Before you can stream the sequence created with `P` or `PH` classes, you have to convert it to PSSequence object using conversion function `convert_PPH_to_PSSequence`. This function takes the old format sequences and converts into a new `PSSequence` object.
4. Stream!

Please take a look into migration example files

- "Example2_QuickStart_migration.m"
- "Example4_LargeSequences_and_Repetitions_migration.m"

4.10.3 LabView

Client code for LabView closely follows the principles of sequence construction and streaming as in the MATLAB client. Please look into the examples.

4.11 Programming examples

Further examples for Matlab, LabVIEW and Python are provided with the following download: PulseStreamer-1.0.1.zip

CHAPTER

FIVE

CHANGELOG

5.1 2018-11-09

5.1.1 Firmware update v1.0.1

- API has been supplemented by method rearm() and forceFinal()
- second permanent IP 169.254.8.2/16 added
- network configuration file on user partition -> static IP can be configured via RPCs
- login password changed

5.1.2 Clients

Python

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

Matlab

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

LabView

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

5.2 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

5.3 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

5.4 2017-05-07

clients

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

documentation

- added 'Getting Started' section

5.5 2016-04-08

Matlab client

- added links to the Matlab client examples

5.6 2016-03-17

static sequence beta 0.9

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

5.7 2016-03-07

provide network configuration

- added section on network configuration

5.8 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

5.9 2016-02-02

static sequence alpha

CHAPTER

SIX

INDICES AND TABLES

- genindex
- search