Pulse Streamer 8/2 Documentation

Release 0.1

Swabian Instruments

February 23, 2016


Swabian instruments

CHAPTER

ONE

HARDWARE

1.1 Output Channels

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

1.1.1 Digital Output

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

1.1.2 Analog Output

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

1.2 Trigger Input

The Pulse Streamer 8/2 has one trigger input that can be applied to GPIO 3.

1.3 General purpose IO (available upon request)

The device is ready to be equipped with SMA connectorized general purpose digital IO ports and slow analog input and output ports. These can be used to implement custom features such as special fast input or output triggers, enable / disable gates, software controllable input and output lines, etc.. Please contact us for custom designs.

CHAPTER

TWO

PROGRAMMING INTERFACE

2.1 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 {
    unsigned int ticks; // duration in ns
    unsigned char digi; // bit mask
    short ao0;
    short 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.

2.2 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 indefinitely or an integer number of times. In the latter case, you can additionally specify the output state after the execution of the sequence.

By default, the sequence is executed immediately. Alternatively, you can tell the system to wait for an external trigger applied to GPIO channel 3.

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

```
void stream(std::vector<Pulse> sequence,
            unsigned long n_runs=0,
            Pulse final=Pulse{0,0,0,0},
            Pulse underflow=Pulse{0,0,0,0},
            bool triggered=False
            )
```

sequence represents the pulse sequence

the sequence is repeated indefinitely if n_runs <=0 and a finite number of repetitions otherwise.

final represents the final output state after completing the sequence (the tick value is ignored).

underflow specifies the output state that the instrument should enter in case a buffer undeflow occurs (the tick value is ignored).

If trigger is true, the system waits for an external trigger before starting the sequence.

2.3 Data streaming and underflows

The total sampling rate of the pulse streamer is higher than its internal data transfer rate (~6.4 Gbit / s). Thus, the instrument will run into buffer underflow conditions when the pulse sequence cannot be represented in a compressed form. This is the case e.g. for very short digital pulses with arbitrary channel masks or for arbitrary analog sequences with the full sample rate. However, in most use cases the pulse sequence can be represented in a compressed form, where the actual rate of transfered information is smaller than the sampling rate. Buffer underflows will not occurr when the average repeat value of the corresponding low level pulses is larger or equal to 3.

To treat buffer underflows gracefully, the instrument will halt the output data stream and set the output levels to a user defined state.

If you are streaming sequences at the edge of the cappability, it is good practice to check for buffer underflows on a regular basis using the programming interface.

2.4 Checking for buffer underflows

You can check for buffer underflows with a single function call. The underlying C++ function is

```
bool getUnderflow()
```

The function returns true when the system has entered the underflow state.

2.5 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 throught 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.

2.6 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 Wikpedia https://en.wikipedia.org/wiki/JSON-RPC.

The JSON-RPC URL of the Pulse Streamer is http://192.168.1.100:8050/json-rpc.

2.6.1 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 is

```
{base64 string sequence,
 unsigned long n_runs,
 {unsigned int ticks, unsigned char digi, short ao0, short ao1},
 {unsigned int ticks, unsigned char digi, short ao0, short ao1},
 bool triggered}
```

"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 exaple for connecting to the JSON-RPC server random_pulses.py. All other arguments are self explanatory.

2.7 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 and is in a beta development stage.

The gRPC server of the Pulse Streamer is 192.168.1.100:50051.

2.7.1 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. Please check out the source file pulse_streamer.proto.

```
syntax = "proto3";
package pulse_streamer;

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

message SequenceMessage
{
    repeated PulseMessage pulse = 1;
    int64 repeat = 2;
    PulseMessage final = 3;
    PulseMessage underflow = 4;
    bool triggered = 5;
}

message PulseStreamerReply
{
    string message = 1;
}

service PulseStreamer
{
    rpc stream (SequenceMessage) returns (PulseStreamerReply) {}
}
```

Please check out the python example for connecting to the gRPC interface random_pulses.py.

CHAPTER

THREE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*