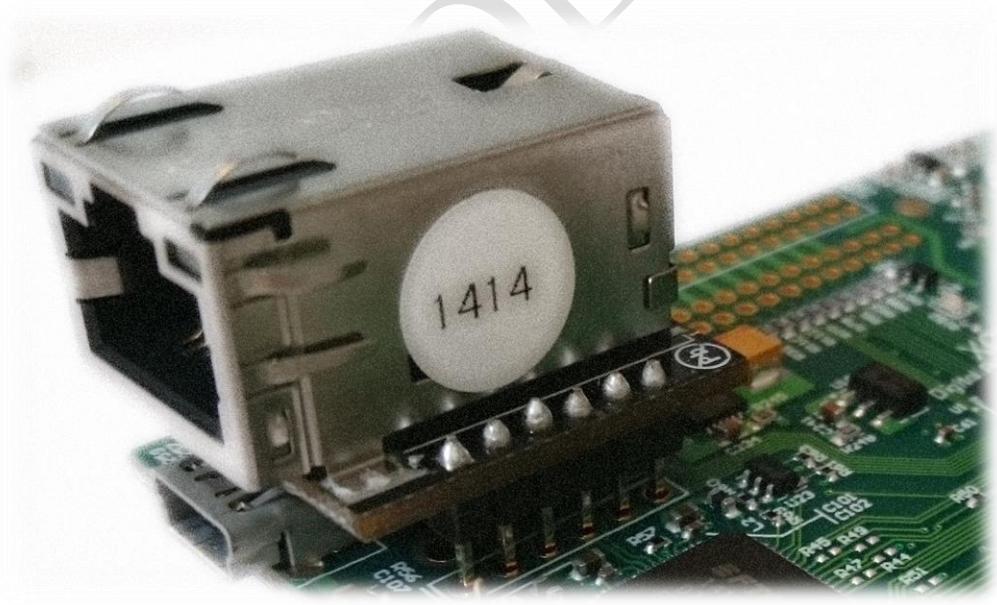
XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)
e-mail: info@xglab.it
ph: +39 02 49660460
P. IVA/C.F. 06557660963
Capitale Sociale 50.000 euro I.V., R.E.A. MI-1899304

Milano, November 29th, 2018
Manual revision: 2.4

OBJECT: DANTE Application Programming Interface



Contents

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

Intended audience

This manual details the use of the library to control DANTE Digital Pulse Processor Systems connected using the USB or TCP-IP Ethernet protocols.

It is intended for software developers needing to integrate DANTE systems in a custom acquisition software through the use of the library.

System requirements

The software is available for Windows Vista, Windows 7, 8, 8.1, 10, both x32 and x64. The library is available also for Ubuntu 64 bit, and can be compiled under other Linux distributions on request.

An USB 2.0 port and an Ethernet connector is required to operate the system.

Linux installation

Under Linux few operation have to be done before using the library. Those mostly involve the USB drivers which have to be manually installed because Linux by default loads a wrong type of driver for USB communication with our hardware.

1) Inside the software package you will find under Drivers a directory named libftd2xx-x86_64-1.4.8. There you firstly will find a guide named AN_220_FTDI_Drivers_Installation_Guide_for_Linux.pdf. Execute the steps described in section 2.1.1 Native Compiling.
2) If not already installed install libusb package with these commands:

   sudo apt-get install libusb-1.0-0-dev

   sudo apt-get install libusb-dev
3) Launch the script ftdi_config.sh that you will find inside the directory libftd2xx-x86_64-1.4.8.
4) In order to automatically load the drivers of the device without root permissions the user has to added to a special group, so launch this command:

   sudo usermod -aG usb <username>

   where <username> it's the name of your Linux user.
5) Reboot.

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

Introduction

The library provided is used to control the main system functions to integrate it into a customer system through a C/C++ program (or any other language that may use a library with an ANSI-C API).

The library is also compatible with Python (2.7.15 or newer 2.x versions, 3.x currently not supported), see 'Python' section for details.

The library is available for dynamic inclusion into the customer software. The supported compilers for Windows are Microsoft C++ and Embarcadero C++ Builder. An import library (.lib) is included in the software for both of them. The supported compiler for Linux is GCC. Other compilers may be used although the export library may not be compatible. It should be always possible to include the library through dynamic loading of the library and declaration of the required functions. Please consider that other compilers are not supported.

This manual is about Ethernet library version 3.0.2.1 and firmware version 3.1.2.

Technical notes:
The calling convention used for the functions is __cdecl. If required for some reason, this may be changed asking us for a specialized version.

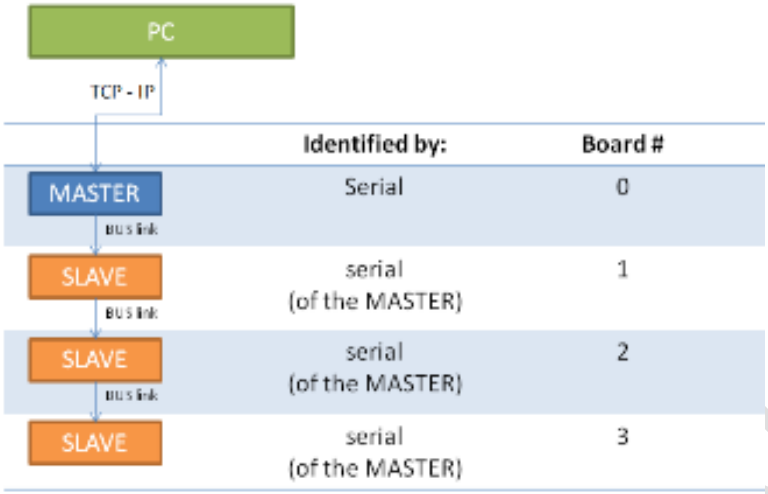
For integers the interface of the library uses the intXX_t types of the standard library, introduced from C++ 11. This avoids any type of misunderstandings about integers size because they are independent on the OS, either if it is a 32 bit or 64 bit distribution.

Size of these types must be respected to correctly interface the library with your software (for example a Boolean value in .NET is 4 bytes and needs to be manualually marshaled).

The library is able to control multiple systems in a LAN or USB connected from the same host PC. Each board is identified through its IP (TCP devices) or serial number (USB devices). For this reason, in most calls the "identifier" (that is the IP or the serial number) is the variable used to identify which of the connected boards (or chains of boards) is the target of the requested operation.

If more than one board is connected in daisy-chain mode, the library will control the full chain through a single identifier that is the "identifier" of the master board. The other boards are identified through a progressive number (starting from '0' for the first board – *see following image*).

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)



The library functions can be divided in two categories: functions directly communicating with the boards (which are asynchronous) and functions communicating only with the library (which are synchronous). It is very convenient indeed in a LAN environment to use asynchronous calls to interact with the device because, if the answer takes long time or in presence of network issues, the caller thread does not stay blocked waiting. Instead other functions will not talk directly with the boards but will write or read internal library data. So in this case the functions will be blocking and will immediately give you data or indicate if the operation was successful.

The asynchronous functions will return immediately a call ID (always incremented) and will give back the answer to the caller in two ways: using callbacks or adding the answers in a queue. In the first case the callback function will have among the parameters the call ID so that the user can combine requests and answers. In the latter case the call ID will be added to the queue too for the same purpose.

We distribute two versions of the library: the Callback library and the Polling library, reflecting these two different behaviors.

The synchronous functions will return a boolean value indicating if the function was successful or not. If the result is FALSE the required operation may not complete correctly. Some functions are used to detect this kind of errors.

The library does not automatically recognizes DANTE devices in the network but it needs to know their IP first. **The IP configuration of DANTE boards can be set through USB so refer to the USB library manual.** The user can add the IP of DANTE boards to a query and then the library will start communicating them, if connected. The library is able to detect hot-plugged devices, without reloading it or the program that uses the library. Errors may occur if a device is removed from the system during a command execution. The library however should be able to recover this type of errors.

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

The callback function

If using the Callback library the user must register, before any other operation, a callback function with this signature:

```
void callback(uint16_t type,
              uint32_t call_id,
              uint32_t length,
              uint32_t* data)
```

The answer of a preceding asynchronous function is given by the library calling this function.

The parameter *type* indicate the type of operation to which the answer refers. It can have three values: 1 if it refers to a read operation, 2 if it refers to a write operation, 0 if an error occurred.

The parameter *call_id* holds the call ID of the corresponding request, so that the user can combine answers with requests.

The parameter *length* tells how many elements there are in the *data* array.

The array *data* contains the read data if it has been a read command or a 1 if it has been a successful write command.

In the descriptions of asynchronous functions you will find how the *data* array is filled for each of them.

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

The answers queue

In the Polling library the answers are inserted in a queue called the answers queue. The data inserted here are the same parameters of the callback function (not present in this version of the library). In particular the parameters are inserted in this order: *call_id*, *type*, *length*, *data*.

In the descriptions of asynchronous functions you will find how the *data* array is filled for each of them.

Acquisition Modes of the DANTE system

The system may work in different acquisition modes. This section contains details about them and shows what functions are relevant for each mode. The system automatically switches between the two modes if controlled by this library. Please refer to functions detailed descriptions for other details.

Normal DPP acquisition mode (single spectrum)

The normal DPP acquisition mode is the default acquisition mode: the system works normally acquiring detected pulses energy and returning the acquired spectrum.

First set the desired DPP parameters with **`configure`** function, start the acquisition with **`start`** function and retrieve data with **`getData`** (together with statistics).

Waveform acquisition mode

The waveform acquisition mode is for debug purposes, allowing the user to acquire the analog input signal of the system.

Start the acquisition with **`start_waveform`** and retrieve waveform samples with **getData.**

List mode

In list mode the DPP returns the energy and the timestamp of each single recorded event.

Configure the DPP with **`configure`** and start the acquisition with **`start_list`**.

Check how many events are available for reading with **`getAvailableData`** and retrieve data with **`getData`.**

List Wave mode

In list-wave mode the DPP returns the energy timestamp and waveform of each single recorded event.

Configure the DPP with **`configure`** and start the acquisition with **`start_listwave`**.

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

Check how many events are available for reading with **getAvailableData** and retrieve data with **getListWaveData.**

Mapping mode (multiple spectra)

In mapping mode the DPP can record multiple spectra of programmable length and return them along with the corresponding statistics.

First set the desired DPP parameters with **configure** function and start the acquisition with **start_map**. Check how many spectra are available for reading with **getAvailableData** in order to allocate enough space. Then retrieve data with **getAllData** (together with corresponding statistics).

Exported functions

Managing controllable systems and library

This functions are used to retrieve information on the available types of systems that this library can control.

In order to use the library it's mandatory to call the InitLibrary function first, otherwise the library will not detect any system and it will not work.

The library supports an arbitrary number of DANTE systems connected to the same PC.

The library assigns a unique name to each system detected. To get the number of connected systems and other information, call **get_dev_number** function and **get_ids** function. The identifiers of the boards, which are their IPs, are used through the whole library to select a system, where the function needs to know on which system to operate. Where this name is not needed the function operates on all the connected systems.

The synchronous library functions return a boolean value (**true**/**false**) to signal if the operation completed successfully or if a problem occurs. To get more information about the error that originate the problem, the **getLastError** function returns the last error detected by the library. The asynchronous functions instead, if a problem occurs, return the error code using the callback function or adding it to the queue. Returned error codes and other details are given later in this text.

The library is thread-safe. If multiple calls by multiple threads happen, the library return an error code **DLL_MULTI_THREAD_ERROR** and only the first call is served.

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

InitLibrary

Declaration:

```
bool InitLibrary ( void )
```

Used to initialize the library. Call this function before any other function, otherwise the other functions will always return false and set **DLL_NOT_INITIALIZED** error.

Parameters:

None

Return value:

A boolean, true, if the library is initialized, false if a problem occurs.

CloseLibrary

Declaration:

```
bool CloseLibrary ( void )
```

Used to close the internal library resources. Call this function before unloading the library from memory, otherwise with some softwares (e.g. with NI LabVIEW) you will not be able to re-initialize the library unless restarting the software itself.

Parameters:

None

Return value:

A boolean, true, if the library closed successfully, false if a problem occurs.

getLastError

Declaration:

```
bool getLastError ( uint16_t& error_code )
```

Returns the last detected error. Note that the error code is not reset between function calls. It always reflects the last detected error. **For example, if a function fails and**

the next function works correctly, the returned error code is the one set by the first function call.

Parameters:

uint16_t& error_code

An integer showing the last error detected by the library. The error_code variable has to be instantiated by the user program.

A very little C example:

```
#include "XGL_DPP.h" // Include the library header.
uint16_t error_code = DLL_NO_ERROR;
bool result = false;
result = InitLibrary(); // Initialize the library.

if (result) {
          // Do some wrong stuff here...
          // Detect error:
          result = getLastError(error_code); // Get last error code.
          if (result) {
                    switch (error_code) {
                    case DLL_MULTI_THREAD_ERROR:
                    // Do something...
                    break;
                    case DLL_CLOSED:
                    // etc...
}}}
```

The possible error code names exported and their corresponding integer values are:

Error code	Value	Description
DLL_NO_ERROR	0	No errors occured.
DLL_MULTI_THREAD_ERROR	1	Another thread has a lock on the library functions.
DLL_NOT_INITIALIZED	4	The library is not initialized. Call InitLibrary before calling anything else.
DLL_CHAR_STRING_SIZE	5	Supplied char buffer size is too short. Return value updated with minimum length.
DLL_ARRAY_SIZE	6	An array passed as parameter to a function has not enough space to contain all the data that the function should return.
DLL_ALREADY_INITIALIZED	42	The library has been already initialized.
DLL_COM_ERROR	57	Communication error or timeout.
DLL_ARGUMENT_OUT_OF_RANGE	60	An argument supplied to the library is out of valid range.
DLL_WRONG_SERIAL	62	The supplied serial is not present in the system.
DLL_TIMEOUT	64	An operation timed out. Result is unspecified.

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

DLL_CLOSING	67	Error during library closing.
DLL_RUNNING	68	The operation cannot be completed while the system is running.
DLL_WRONG_MODE	69	The function called is not appropriate for the current mode.
DLL_NO_DATA	70	No data to be read.
DLL_DECRYPT_FAILED	71	An error occured during decryption.
DLL_INVALID_BITSTREAM	72	Trying to upload an invalid bistream file.
DLL_FILE_NOT_FOUND	73	The specified file hasn't been found.
DLL_INVALID_FIRMWARE	74	Invalid firmware detected on one board. Upload a new one with load_firmware function.
DLL_UNSUPPORTED_BY_FIRMWARE	75	Function not supported by current firmware of the board. Or firmware on the board is not present or corrupted.
DLL_THREAD_COMM_ERROR	76	Error during communication between threads.
DLL_MISSED_SPECTRA	78	One ore more spectra missed.
DLL_MULTIPLE_INSTANCES	79	Library open by multiple processes.
DLL_THROUGHPUT_ISSUE	80	Download rate from boards at least 15% lower than expected, check your connection speed.
DLL_INCOMPLETE_CMD	81	A communication error caused a command to be truncated.
DLL_MEMORY_FULL	82	The hardware memory became full during the acquisition. Likely the effective throughput is not enough to handle all the data.
DLL_SLAVE_COMM_ERROR	83	Communication error with slave board
DLL_SOFTWARE_MEMORY_FULL	84	Allowed memory for the library became full. Likely some data/event have been discarded.

If one of the following errors are returned, contact us for further investigation:

Error code	Value	Description
DLL_WIN32API_FAILED_INIT	3	Win32 errors - Debugging.
DLL_WIN32API_GET_DEVICE	7	Win32 errors - Debugging.
DLL_WIN32API	41	Generic WIN32 API error.
DLL_WIN32API_INIT_EVENTS	43	Win32 errors - Debugging.
DLL_WIN32API_RD_INIT_EVENTS	44	Win32 errors - Debugging.
DLL_WIN32API_REPORTED_FAILED_INIT	45	Win32 errors - Debugging - Possible hardware/driver error.
DLL_WIN32API_UNEXPECTED_FAILED_INIT	46	Win32 errors - Debugging.

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

DLL_WIN32API_MULTI_THREAD_INIT_SET	47	Win32 errors - Debugging.
DLL_WIN32API_LOCK_HMODULE_SET	48	Win32 errors - Debugging.
DLL_WIN32API_HWIN_SET	49	Win32 errors - Debugging.
DLL_WIN32API_WMSG_SET	50	Win32 errors - Debugging.
DLL_WIN32API_READ_SET	51	Win32 errors - Debugging.
DLL_WIN32API_GET_DEVICE_SIZE	52	Win32 errors - Debugging.
DLL_WIN32API_DEVICE_UPD_LOCK_G	53	Win32 errors – Debugging.
DLL_FT_CREATE_IFL	54	FT errors - Debugging.
DLL_FT_GET_IFL	55	FT errors - Debugging.
DLL_CREATE_DEVCLASS_RUNTIME	56	Library errors - Debugging - Possible hardware/driver error.
DLL_CREATE_DEVCLASS_ARGUMENT	58	Library errors - Debugging.
DLL_CREATE_DEVCLASS_COMM	59	Library errors - Debugging.
DLL_RUNTIME_ERROR	61	Generic runtime error.
DLL_WIN32API_HMODULE	65	Win32 errors - Debugging.
DLL_WIN32API_DEVICE_UPD_LOCK_F	66	Win32 errors - Debugging.
DLL_INIT_EXCEPTION	77	Library errors - Debugging.

Return values:

A boolean, true, if the returned error_code is valid. Returns false and **DLL_MULTI_THREAD_ERROR** if **getLastError** is called while another thread is working with the library.

resetLastError

Declaration:

```
bool resetLastError ( void )
```

Resets the "last error" variable to **DLL_NO_ERROR**.

Parameters:

None

Return values:

A boolean, true, if the function succeeds.

libVersion

Declaration:

```
bool libVersion ( char* version,
```


XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

```
                    uint32_t& version_size )
```

Returns the library version string.

Parameters:

char* version

A pointer to a null-terminated C string (array of chars). The memory for this array has to be initialized by the user program.

uint32_t& version_size

An integer passed by reference, set to the maximum length of the version string. If the length is not enough the function returns with false and sets the correct size in this variable.

C example:

```
// Initialize library first!

uint32_t size = 20; // Enough space for the string.
char* version = new char[size];
bool result = false;

result = libVersion(version,size);

If (result) {
      // Output library version to the command prompt:
      printf("Library version is: %s\r\n",version);
}

delete[] version; // Free memory.
```

Outputs: "**Library version is: 2.0.2**".

Return values:

True if the returned value is correct, false if a problem occurs (probably string size is not enough).

add_to_query

Declaration:

```
bool add_to_query ( char* address )
```

Add an IP address to the query.

Parameters:

char* address

A string containing the IP to be added (for example **"192.168.1.120"**).

Return values:

True if the parameter is correct, false if a problem occours.

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

remove_from_query

Declaration:

```
Bool remove_from_query ( char* address )
```

Remove an IP address from the query.

Parameters:

char* address

A string containing the IP to be removed.

Return values:

True if the parameter is correct, false if a problem occurs.

get_dev_number

Declaration:

```
bool get_dev_number ( uint16_t& devs )
```

Get the number of connected devices (chain masters only).

Parameters:

uint16_t& devs

The number of master devices connected.

Return values:

True if the returned value is correct, false if a problem occurs.

get_ids

Declaration:

```
bool get_ids ( char* identifier,
               uint16_t& nb,
               uint16_t& id_size)
```

Returns devices serial from progressive number 'nb' of connected devices.

Parameters:

char* identifier

A string that contains the IP of the selected connected device.

uint16_t& nb

The progressive number of connected device; nb starts from '0'.

uint16_t& id_size

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

An integer passed by reference, set to the maximum length of the serial string. If the length is not enough the function returns with false and sets the correct size in this variable.

Return values:

True if all the parameters are filled with correct values, false otherwise.

getFirmware

Declaration:

```
uint32_t getFirmware ( const char* identifier,
                       uint16_t Board)
```

Ask information about the firmware of a specific system. Asynchronous call.

Parameters:

const char* identifier

A null-terminated string with the identifier of the system to query.

uint16_t Board

It contains the progressive number of the board to query, starting from '0' for the MASTER of the chain.

Return values:

The call ID if the parameters are filled with correct values, 0 otherwise.

Asynchronous answer data:

In the data array of the callback function or in the answers queue you will find, in this order, the Major, Minor, Build of the firmware. Otherwise, if the call was not successful, the type field will be 0.

get_boards_in_chain

Declaration:

```
bool get_boards_in_chain ( const char* identifier,
                           uint16_t& devs )
```

Get the number of devices in the chain controlled by the specified master board.

Parameters:

const char* identifier

A null-terminated string with the identifier of the system to query.

uint16_t& devs

The number of devices in the chain.

Return values:

True if all the parameters are filled with correct values, false otherwise.

write_IP_configuration

Declaration:

```
uint32_t write_IP_configuration ( const char* identifier,
                                  const char* IP,
                                  const char* subnet_mask,
                                  const char* gateway)
```

Write a new IP configuration to the specified DPP board.

Parameters:

const char* identifier

A null-terminated string with the identifier of the system to query.

const char* IP

String of the IP.

const char* subnet_mask

String of the subnet mask

const char* gateway

String of the gateway.

Return values:

ID code of the reply.

load_firmware

Declaration:

```
uint32_t load_firmware ( const char* identifier,
                         bool store,
                         const char* filename)
```

Loads a new firmware via USB.

Parameters:

const char* identifier

A null-terminated string with the identifier of the system to query.

bool store

With store to false, it will load to the system a temporary firmware (20 seconds) that will be lost if rebooted. Otherwise whith store to true it will store the firmware on memory and it will persist even after power cycles.

const char* filename

The filename must contain the path in this format:
C:\\ArbitraryDirectory\\firmware_name.bitc

Return values:

ID code of the reply.

Get_load_fw_progress

Declaration:

```
bool get_load_fw_progress (double& progress)
```

Get the progress of the load firmware operation.

Parameters:

Double& progress

Floating number from 0 to 1.

Return values:

A boolean true, if the returned number is correct, false otherwise.

global_reset

Declaration:

```
bool global_reset (const char* identifier,)
```

Resets the communication on the entire chain.

Parameters:

const char* identifier

A null-terminated string with the identifier of the system to query Floating number from 0 to 1.

Return values:

A boolean true, if the returned number is correct, false otherwise.

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

Configuration

This functions are used to configure the system parameters to get the best acquisition results.

configure

Declaration:

```
uint32_t configure ( char* identifier,
                     uint16_t Board,
                     const configuration const cfg)
```

Configures the system with the required acquisition configuration. Asynchronous call.

Parameters:

char* identifier

A null-terminated string with the identifier of the system to query.

uint16_t Board

It contains the progressive number of the board to query, starting from '0' for the MASTER of the chain.

const configuration const cfg

A constant pointer to a constant structure that defines the base system configuration. Memory for this structure has to be allocated and maintained by the caller.

The structure is defined with the following fields:

```
 struct configuration {
       uint32_t fast_filter_thr = 0;
       uint32_t energy_filter_thr = 0;
       uint32_t energy_baseline_thr = 0;
       double max_risetime = 20;
       double gain = 1;
       uint32_t peaking_time = 25;
       uint32_t max_peaking_time = 0;
       uint32_t flat_top = 5;
       uint32_t edge_peaking_time = 4;
       uint32_t edge_flat_top = 1;
       uint32_t reset_recovery_time = 200;
       double zero_peak_freq = 1;
       uint32_t baseline_samples = 64;
       bool inverted_input = 0;
       double time_constant = 0;
       uint32_t base_offset = 0;
       uint32_t overflow_recovery = 0;
       uint32_t reset_threshold = 0;
       double tail_coefficient = 0;
       uint32_t other_param = 0;
};
```

Where:

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

- **fast_filter_thr** is the detection threshold for the fast filter (unit: spectrum BIN, range: 0 to 4096).
- **energy_filter_thr** is the detection threshold for the energy filter (currently not implemented, its value it's ignored).
- **Energy_baseline_**thr is the threshold for the baseline calculation currently not implemented, its value it's ignored).
- **max_risetime** is the maximum expected risetime of the detector, used for pileup rejection (unit: 8 ns samples, range: 0 to 127). Set to '0' to disable pileup rejection.
- **gain** is the main gain (unit: spectrum BIN / ADC's LSB, range: 0 to 1.99).
- **peaking_time** is the main energy filter peaking time (unit: 32 ns samples, range: 1 to 500)
- **max_peaking_time** Max energy filter peaking time (unit: 32 ns samples)
- **flat_top** is the main energy filter flat top time (unit: 32 ns samples, range: 1 to 15)
- **edge_peaking_time** is the peaking time of the fast filter (unit: 8 ns samples, range: 1 to 31).
- **edge_flat_top** is the peaking time of the fast filter (unit: 8 ns samples, range: 1 to 15).
- **reset_recovery_time time** is the reset recovery time (unit: 8 ns samples; range: 0 to 224-1)
- **zero_peak_freq** is the frequency of the zero peak measurement (unit: kcps, range: 1 to 501).
- **baseline_samples** is a setting for baseline correction (unit: 32 ns samples, range: 0,16,32,64,128,256 or 512).
- **inverted_input**
- **time_constant** it's a parameter currently not implemented, it's value its ignored.
- **base_offset** it's a baseline of the continous reset signal
- **overflow_recovery** overflow recovery time
- **reset_threshold** reset detection threshold
- **tail_coefficient** overflow recovery time
- **other_param:**
    - bit 0: enable gating
    - bit 1: edge of gating

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

- bit 5 to bit 2: OCR limit setting. Supported only by ListWave acquisition mode.

Bit 5 to bit 2	OCR limit	Bit 5 to bit 2	OCR limit
x0	Disabled	x8	8 KHz
x1	100Hz	x9	10KHz
x2	200 Hz	xA	20 KHz
x3	400 Hz	xB	40 KHz
x4	800 Hz	xC	80 KHz
x5	1 KHz	xD	100 KHz
x6	2 KHz	xE	200 KHz
x7	4 KHz	xF	Disabled

Return values:

The call ID if the parameters are filled with correct values, 0 otherwise.

Asynchronous answer data:

In the data array of the callback function or in the answers queue you will find a 1 if the operation succeeded. Otherwise, if the call was not successful, the type field will be 0.

configure_timestamp_delay

Declaration:

```
uint32_t configure ( char* identifier,
                     uint16_t Board,
                     int32_t timestamp_delay)
```

Configures the timestamp delay with the required acquisition configuration. Asynchronous call.

Parameters:

char* identifier

A null-terminated string with the identifier of the system to query.

uint16_t* Board

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

It contains the progressive number of the board to query, starting from '0' for the MASTER of the chain.

int32_t timestamp_delay

Delay applied to the clock propagated among the DPP chain. Change the delay value allows to shift the timestamps among DPP connected in daisy-chain. Delay value ≈ 2ns , range: 0 to 15.

Return values:

The call ID if the parameters are filled with correct values, 0 otherwise.

Asynchronous answer data:

In the data array of the callback function or in the answers queue you will find a 1 if the operation succeeded. Otherwise, if the call was not successful, the type field will be 0.

Configure_offset

Declaration:

```
uint32_t configure_offset ( char* identifier,
                            uint16_t Board,
                            const configuration_offset const cfg)
```

Configures the timestamp delay with the required acquisition configuration. Asynchronous call.

Parameters:

char* identifier

A null-terminated string with the identifier of the system to query.

uint16_t Board

It contains the progressive number of the board to query, starting from '0' for the MASTER of the chain.

const configuration_offset const cfg

A constant pointer to a constant structure that defines the base system configuration. Memory for this structure has to be allocated and maintained by the caller.

The structure is defined with this fields:

```
struct configuration_offset {
            uint32_t offset_val1;
            uint32_t offset_val2;
};
```

Where:

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

- offset_val1 is the offset of digpot1 (range: 0 to 255).
- offset_val2 is the offset of digpot1 (range: 0 to 255).

Return values:

The call ID if the parameters are filled with correct values, 0 otherwise.

Asynchronous answer data:

In the data array of the callback function or in the answers queue you will find a 1 if the operation succeeded. Otherwise, if the call was not successful, the type field will be 0.

Acquisition control

These functions are used to control the acquisition state of each system.

isRunning_system

Declaration:

```
uint32_t isRunning_system ( const char* identifier,
                            uint16_t Board )
```

Ask a board if it is running (acquisition in progress). Asynchronous call.

Parameters:

const char* identifier

A null-terminated string with the identifier of the system to query.

uint16_t Board

It contains the progressive number of the board to query, starting from '0' for the MASTER of the chain.

Return values:

The call ID if the parameters are filled with correct values, 0 otherwise.

Asynchronous answer data:

In the data array of the callback function or in the answers queue you will find 1 if the board is in running and 0 if not. Otherwise, if the call was not successful, the type field will be 0.

start

Declaration:

```
uint32_t start ( const char* identifier,
                 const double time,
```


XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

```
const uint16_t spect_depth )
```

Start a single spectrum for a specified amount of time or in free-running mode. Asynchronous call.

Parameters:

const char* identifier

A null-terminated string with the identifier of the system to query.

const double time

The amount of time the acquisition should run, expressed in seconds. The system precision is up to 0.001 secs (1 msec) for measurements from 1 msec to 1 hour. The system stops automatically after the specified time elapses, but the function stop need to be called anyway before starting another measure.

Use 0 if a free-running measure is required (the system never stops). To stop a free running measure call the stop function.

const uint16_t spect_depth

The bins number of each spectrum. This value can be 1024, 2048 or 4096. No other value are allowed. This setting affect also the amount data returned by the getData function.

Return values:

The call ID if the parameters are filled with correct values, 0 otherwise.

Asynchronous answer data:

In the data array of the callback function or in the answers queue you will find 1 if the function succeeded. Otherwise, if the call was not successful, the type field will be 0.

stop

Declaration:

```
uint32_t stop ( const char* identifier )
```

Stops a running acquisition. Asynchronous call.

Parameters:

const char* identifier

A null-terminated string with the identifier of the system to query.

Please note that if the system is not in free-running mode, the system stops automatically after the specified time elapses, but the function stop need to be called anyway before starting another measure.

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

Return values:

The call ID if the parameters are filled with correct values, 0 otherwise.

Asynchronous answer data:

In the data array of the callback function or in the answers queue you will find 1 if the function succeeded. Otherwise, if the call was not successful, the type field will be 0.

clear

Declaration:

```
bool clear ( const char* identifier )
```

Clears acquisition data, run time and statistical data.

Parameters:

const char* identifier

A null-terminated string with the identifier of the system to query.

Return values:

A boolean true if the data is cleared, false otherwise.

start_waveform

Declaration:

```
uint32_t start_waveform ( const char* identifier,
                          uint16_t mode,
                          uint16_t dec_ratio,
                          uint32_ttrig_mask,
                          uint32_t trig_level,
                          const double time,
                          uint16_t length )
```

Start the acquisition in waveform acquisition mode, for a specified amount of time and with additional acquisition settings (trigger, trigger level, decimation ratio, length). Asynchronous call.

Parameters:

const char* identifier

A null-terminated string with the identifier of the system to query.

const uint16_t mode

The waveform acquisition modes are: analog mode (0), digital mode. By now, only the waveform analog mode is usable.

uint16_t dec_ratio

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

An integer indicating the decimation of the acquired samples, from 1 to 32. If set to 1 it is acquired 1 sample every 16 ns, if set to N one sample every N samples is acquired.

`uint32_t trig_mask`

The trigger mask. Every bit of the integer is a specific type of hardware trigger: if set to 1 the trigger is enabled, if set to 0 the trigger is disabled. Implemented triggers are:

- Bit 0: Instant trigger
- Bit 1: Rising slope crossing of trigger level
- Bit 2: Falling slope crossing of trigger level

The system waits for a triggering event for the time specified with time variable.

`uint32_t trig_level`

The trigger level of the acquisition (from 0 to 65535).

`const double time`

The amount of time the system waits for a triggering event, expressed in seconds.

`uint16_t length`

This parameter specify the desired length of the waveform and is expressed in 16384 samples unit. For example if length is set to 1, 16384 samples are acquired; if set to 2, 32768 sample are acquired; etc.

The function stop has to be called before another acquisition is started.

Return values:

The call ID if the parameters are filled with correct values, 0 otherwise.

Asynchronous answer data:

In the data array of the callback function or in the answers queue you will find 1 if the function succeeded. Otherwise, if the call was not successful, the type field will be 0.

start_list

Declaration:

```
uint32_t start_list ( const char* identifier,
                      const double time )
```

Start a list mode acquisition for a specified amount of time or in free-running mode. Asynchronous call.

Parameters:

`const char* identifier`

A null-terminated string with the identifier of the system to query.

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

const double time

The amount of time the acquisition should run, expressed in seconds. The system precision is up to 0.001 secs (1 msec) for measurements from 1 msec to 48 hours. The system stops automatically after the specified time elapses, but the function stop must be called anyway before a new measure is started.

Use 0 if a free-running measure is required (the system never stops). In free-running mode the system is controlled by the host pc and the stop function has to be called.

Return values:

The call ID if the parameters are filled with correct values, 0 otherwise.

Asynchronous answer data:

In the data array of the callback function or in the answers queue you will find 1 if the function succeeded. Otherwise, if the call was not successful, the type field will be 0.

start_listwave

Declaration:

```
uint32_t start_listwave ( const char* identifier,
                          const double time,
                          uint16_t dec_ratio,
                          uint16_t length,
                          uint16_t offset )
```

Start the acquisition in list-wave acquisition mode, for a specified amount of time and with additional acquisition settings (decimation ratio, length, offset). Asynchronous call.

Parameters:

const char* identifier

A null-terminated string with the identifier of the system to query.

uint16_t dec_ratio

An integer indicating the decimation of the acquired samples, from 1 to 32. If set to 1 it is acquired 1 sample every 16 ns, if set to N one sample every N samples is acquired.

const double time

The amount of time the system waits for a triggering event, expressed in seconds.

uint16_t length

This parameter specifies the desired length of the waveform and is expressed in 16ns samples unit. Allowed range is from 8 to 800 samples.

uint16_t offset

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

This parameter specifies the desired offset of the waveform from the trigger of the waveform acquisition. It's expressed in 16ns samples unit. Allowed range is from 0 to 300 samples.

The function stop has to be called before another acquisition is started.

Return values:

The call ID if the parameters are filled with correct values, 0 otherwise.

Asynchronous answer data:

In the data array of the callback function or in the answers queue you will find 1 if the function succeeded. Otherwise, if the call was not successful, the type field will be 0.

start_map

Declaration:

```
uint32_t start_map ( const char* identifier,
                     const uint32_t sp_time,
                     const uint32_t points,
                     const uint16_t spect_depth )
```

Start a map acquisition (multiple spectra) for a specified amount of time for each point. Possibly, the number of points can be indefinite (free-running map) and the end of each point triggered by an external trigger. Asynchronous call.

Parameters:

const char* identifier

A null-terminated string with the identifier of the system to query.

const uint32_t sp_time

The duration of each spectrum, expressed in milliseconds. If it is set to 0 the DPP waits for an external trigger to start a new spectrum.

const uint32_t points

The number of spectra to acquire. If it is set to 0 the DPP continues to acquire indefinitely, until a stop is issued by the user (free-running map).

There is no deadtime between one spectrum to another so the total acquisition time in ms will be sp_time*points.

const uint16_t spect_depth

The bins number of each spectrum. This value can be 1024, 2048 or 4096. No other value are allowed. This setting affect also the amount data returned by the getData function.

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

Return values:

The call ID if the parameters are filled with correct values, 0 otherwise.

Asynchronous answer data:

In the data array of the callback function or in the answers queue you will find 1 if the function succeeded. Otherwise, if the call was not successful, the type field will be 0.

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

Retrieving acquired data

getAvailableData

Declaration:

```
bool getAvailableData ( const char* identifier,
                        uint16_t Board
                        uint32_t& data_number)
```

Get the number of available spectra in map acquisition mode or number of events in list mode. In case of list_wave mode the function returns both the number of event and the waveform. This function should be called before reading data in mapping mode and list mode in order to know how much space it needs to be allocated.

Parameters:

const char* identifier

A null-terminated string with the identifier of the system to query.

uint16_t Board

It contains the progressive number of the board to query, starting from '0' for the MASTER of the chain.

uint32_t& data_number

The number of available spectra in mapping mode or the number of events available for reading while in list mode. In case of list-wave acquisition, every four waveform samples, the data_number value increases by 1.

Return values:

A boolean true, if the system started with the set parameters, false otherwise.

getData

Declaration:

```
bool getData ( const char* identifier,
               uint16_t Board,
               uint64_t* values,
               uint32_t& id,
               statistics& stats,
               uint32_t& spectra_size)
```

Gets the acquired data. It should be used for normal DPP mode (single spectrum), waveform mode and list mode acquisitions. It can be used also during a map acquisition but it will return only the last complete spectrum acquired (use **getAllData** to retrieve all the spectra available).

Parameters:

const char* identifier

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

A null-terminated string with the identifier of the system to query.

uint16_t Board

It contains the progressive number of the board to query, starting from '0' for the MASTER of the chain.

The meaning of the remaining parameters depends on the acquisition mode selected:

Normal acquisition mode (single spectrum) or Mapping mode

uint64_t* values

Array of uint64_t (64 bit wide) containing the values of each bin of the spectrum (the spectrum length is programmable and can be 1024, 2048 or 4096).

uint32_t& id

Each spectrum acquired has a unique progressive identifier returned with this parameter. The id is reset only after a power cycle.

statistics& stats

A structure containing the statistics associated to the acquisition. Memory for this structure has to be allocated and maintained by the caller.

The structure is defined with this fields:

```
struct statistics {
       // Basic statistics.
       uint64_t real_time;
       uint64_t live_time;
       double ICR;
       double OCR;
       // Advanced statistics.
       uint64_t last_timestamp;
       uint32_t detected;
       uint32_t measured;
       uint32_t edge_dt;
       uint32_t filt1_dt;
       uint32_t zerocounts;
       uint32_t baselines_value;
       uint32_t pup_value;
       uint32_t pup_f1_value;
       uint32_t pup_notf1_value;
       uint32_t reset_counter_value;
};
```

In this structure is reported the real time (microseconds), the live time (microseconds), the input and output count rate (kcps) of the acquisition. In addition are given also some advanced statistics about pileup rejection and other aspects. For more information about them contact us.

uint32_t& spectra_size

The size of the spectrum. It must be 1024, 2048 or 4096.

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

List mode acquisition

uint64_t* values

Array of uint64_t (64 bit wide) containing information about each single event recorded by the DPP. Each 64 bit value correspond to an event and is organized in this way:

- bits 63 to 20: Timestamp of the X-Ray arrival time (10 ns samples).
- bits 19 to 16: Energy filter number.
- bits 15 to 0: Energy of the acquired X-Ray.

uint32_t& spectra_size

The number of events the caller want to read. They have to be equal or less the available stored events. Use the function **getAvailableData** in order to know the number of counts available.

uint32_t& id

Each spectrum acquired has a unique progressive identifier returned with this parameter. The id is reset only after a power cycle.

statistics& stats

A structure containing the statistics associated to the acquisition (same structure of normal acquisition mode). Memory for this structure has to be allocated and maintained by the caller.

Return values:

Returns true if the values array is filled with correct values, false otherwise. The function return false also if there is no data available to be read.

Remarks:

In normal acquisition mode (spectrum) negative values of measured energies (actually noise) are summed up at the first bin. Instead measured energies that exceed the spectrum range (i.e. larger than the last bin) are summed up at the last bin.

getLiveDataMap

Declaration:

```
bool getData ( const char* identifier,
               uint16_t Board,
               uint64_t* values,
               uint32_t& id,
               statistics& stats,
               uint32_t& spectra_size)
```

Gets the acquired data on live mapping acquisition: it's necessary to verify live data cquired.

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

Parameters:

const char* identifier

A null-terminated string with the identifier of the system to query.

uint16_t Board

It contains the progressive number of the board to query, starting from '0' for the MASTER of the chain.

The meaning of the remaining parameters depends on the acquisition mode selected:

Normal acquisition mode (single spectrum) or Mapping mode

uint64_t* values

Array of uint64_t (64 bit wide) containing the values of each bin of the spectrum (the spectrum length is programmable and can be 1024, 2048 or 4096).

uint32_t& id

Each spectrum acquired has a unique progressive identifier returned with this parameter. The id is reset only after a power cycle.

statistics& stats

A structure containing the statistics associated to the acquisition. Memory for this structure has to be allocated and maintained by the caller.

The structure is defined with this fields:

```
struct statistics {
	// Basic statistics.
	uint64_t real_time;
	uint64_t live_time;
	double ICR;
	double OCR;
	// Advanced statistics.
	uint64_t last_timestamp;
	uint32_t detected;
	uint32_t measured;
	uint32_t edge_dt;
	uint32_t filt1_dt;
	uint32_t zerocounts;
	uint32_t baselines_value;
	uint32_t pup_value;
	uint32_t pup_f1_value;
	uint32_t pup_notf1_value;
	uint32_t reset_counter_value;
};
```

In this structure is reported the real time (microseconds), the live time (microseconds), the input and output count rate (kcps) of the acquisition. In addition are given also some advanced statistics about pileup rejection and other aspects. For more information about them contact us.

uint32_t& spectra_size

The size of the spectrum. It must be 1024, 2048 or 4096.

List mode acquisition

uint64_t* values

Array of uint64_t (64 bit wide) containing information about each single event recorded by the DPP. Each 64 bit value correspond to an event and is organized in this way:

- bits 63 to 20: Timestamp of the X-Ray arrival time (10 ns samples).
- bits 19 to 16: Energy filter number.
- bits 15 to 0: Energy of the acquired X-Ray.

uint32_t& spectra_size

The number of events the caller want to read. They have to be equal or less the available stored events. Use the function **getAvailableData** in order to know the number of counts available.

uint32_t& id

Each spectrum acquired has a unique progressive identifier returned with this parameter. The id is reset only after a power cycle.

statistics& stats

A structure containing the statistics associated to the acquisition (same structure of normal acquisition mode). Memory for this structure has to be allocated and maintained by the caller.

Return values:

Returns true if the values array is filled with correct values, false otherwise. The function return false also if there is no data available to be read.

Remarks:

In normal acquisition mode (spectrum) negative values of measured energies (actually noise) are summed up at the first bin. Instead measured energies that exceed the spectrum range (i.e. larger than the last bin) are summed up at the last bin.

getListWaveData

Declaration:

```
bool getListWaveData ( const char* identifier,
                       uint16_t Board,
                       uint64_t* values,
                       uint64_t* wave_values,
                       uint32_t& id,
```


XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

```
                    statistics& stats,
                    uint32_t& spectra_size)
```

Gets the acquired data. It should be used for list-wave DPP mode.

Parameters:

`const char* identifier`

A null-terminated string with the identifier of the system to query.

`uint16_t Board`

It contains the progressive number of the board to query, starting from '0' for the MASTER of the chain.

The meaning of the remaining parameters depends on the acquisition mode selected:

Normal acquisition mode (single spectrum) or Mapping mode

`uint64_t* values`

Array of uint64_t (64 bit wide) containing information about each single event recorded by the DPP. Each event word is followed by its related waveform using the following structure:

Energy1 & Timestamp1
Sample1&Sample2&Sample3&Sample4
Sample5&Sample6&Sample7&Sample8
…
Energy2 & Timestamp2

Each 64 bit Energy&Timestamp value corresponds to an event and is organized in this way:

- bits 63 to 20: Timestamp of the X-Ray arrival time (10 ns samples).
- bits 19 to 16: Energy filter number.
- bits 15 to 0: Energy of the acquired X-Ray.
-

`uint64_t* wave_values`

Currently not used

uint32_t& id

Each spectrum acquired has a unique progressive identifier returned with this parameter. The id is reset only after a power cycle.

statistics& stats

A structure containing the statistics associated to the acquisition. Memory for this structure has to be allocated and maintained by the caller.

The structure is defined with this fields:

```
struct statistics {
	// Basic statistics.
	uint64_t real_time;
	uint64_t live_time;
	double ICR;
	double OCR;
	// Advanced statistics.
	uint64_t last_timestamp;
	uint32_t detected;
	uint32_t measured;
	uint32_t edge_dt;
	uint32_t filt1_dt;
	uint32_t zerocounts;
	uint32_t baselines_value;
	uint32_t pup_value;
	uint32_t pup_f1_value;
	uint32_t pup_notf1_value;
	uint32_t reset_counter_value;
};
```

In this structure is reported the real time (microseconds), the live time (microseconds), the input and output count rate (kcps) of the acquisition. In addition are given also some advanced statistics about pileup rejection and other aspects. For more information about them contact us.

uint32_t& spectra_size

The size of the spectrum. It must be 1024, 2048 or 4096. It's not used in list-wave acquisition mode.

Return values:

Returns true if the values array is filled with correct values, false otherwise. The function return false also if there is no data available to be read.

Remarks:

In normal acquisition mode (spectrum) negative values of measured energies (actually noise) are summed up at the first bin. Instead measured energies that exceed the spectrum range (i.e. larger than the last bin) are summed up at the last bin.

getAllData

Declaration:

```
bool getAllData ( const char* identifier,
                  uint16_t Board,
                  uint16_t* values,
                  uint32_t* id,
                  double* stats,
                  uint64_t* advstats,
                  uint32_t& spectra_size,
                  uint32_t& data_number)
```

Get a data_number of spectra during a mapping mode acquisition. This function must not be used while in other acquisition modes, use **getData** instead.

Parameters:

const char* identifier

A null-terminated string with the identifier of the system to query.

uint16_t Board

It contains the progressive number of the board to query, starting from '0' for the MASTER of the chain.

uint16_t* values

Array of uint16_t (16 bit wide) containing the counts of each bin (the spectrum length is programmable and can be 1024, 2048 or 4096) of each spectrum. The spectra are one next to the other starting from the first acquired. So, for example, if 10 spectra are committed, the resulting values length will be 10*spectrum_length.

uint32_t* id

Array of unique progressive identifiers of each spectrum. The first id of the array correspond to the first spectrum acquired (as for the parameter values).

double* stats

Array of basic statistics of each spectrum. The statistics give here are (in this order): real time (microseconds), live time (microseconds), ICR (kcps), OCR (kcps). The first statistics correspond to the first spectrum acquired. So, the size of the vector will be 4*data_number.

uint64_t* advstats

Array of advanced statistics for each spectrum. The size of this array will be 24*data_number. For more information about them contact us.

Index *i * 18 + 13* of *adv_stats* (starting from index 0, *i* spectra number) contains all the gating sampled values if enabled by the configure command

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

uint32_t& spectra_size

The size of the spectrum. It must be 1024, 2048 or 4096.

uint32_t& data_number

The number of spectra requested. It must be equal or less than the number of spectra available. The user should call the **getAvailableData** function first to allocate enough space for spectra and statistics.

Return values:

Returns true if the values array is filled with correct values, false otherwise. The function return false also if there is no data available to be read.

Remarks:

In normal acquisition mode (spectrum) negative values of measured energies (actually noise) are summed up at the first bin. Instead, measured energies that exceed the spectrum range (i.e. larger than the last bin) are summed up at the last bin.

getWaveData

Declaration:

```
bool getData ( const char* identifier,
               uint16_t Board,
               uint16_t* values,
               uint32_t& data_size)
```

Gets the waveform data. It should be used in waveform mode.

Parameters:

const char* identifier

A null-terminated string with the identifier of the system to query.

uint16_t Board

It contains the progressive number of the board to query, starting from '0' for the MASTER of the chain.

uint16_t* values

In waveform mode values it's an array of ADC samples (wich can go from 0 to 65535), starting from the first acquired sample up to spectra_size acquired samples. The sample acquisition frequency is 62.5 MHz / decimation ratio.

uint32_t& data_size

The number of samples the caller want to read. They have to be equal or less the actual waveform length.

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

Return values:

Returns true if the values array is filled with correct values, false otherwise.

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

Getting Asynchronous Calls' Answers

register_callback

Declaration:

```
bool register_callback ( void (*callback)( uint16_t type,
                                           uint32_t call_id,
                                           uint32_t length,
                                           uint32_t* data))
```

Register the callback function pointer to get the answers of asynchronous calls. Available only in the Callback release.

Parameters:

void (*callback)(uint16_t type, uint32_t call_id, uint32_t length, uint32_t* data)

A pointer to a function with this signature:

uint16_t type

The type of operation to which the answer refers to. It can have three values: 1 if it refers to a read operation, 2 if it refers to a write operation, 0 if an error occurred.

uint32_t call_id

It holds the call ID of the corresponding request, so that the user can combine answers with requests.

uint32_t length

It tells how many elements there are in the *data* array.

uint32_t* data

Array that contains the read data if it has been a read command or a 1 if it has been a successful write command.

Return values:

A boolean true, if the registering succeeded, false otherwise.

GetAnswersDataLength

Declaration:

```
bool getAvailableData ( uint32_t& length )
```

Get how many elements there are in the answers queue. Available only in the Polling release.

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

Parameters:

uint32_t& length

The number of elements in the answers queue.

Return values:

A boolean true, if the returned number is correct, false otherwise.

GetAnswersData

Declaration:

```
bool getAvailableData ( uint32_t length,
                        uint32_t* data )
```

Get a variable number of elements from the answers queue.

Parameters:

uint32_t length

The number to be extracted from the answers queue.

uint32_t* data

An array containing the element retrieved from the answers queue.

Return values:

A boolean true, if the returned data is valid, false otherwise.

Python

Introduction

The DANTE library is compatible with Python (2.7.15 tested, newer 2.x version are compatible, 3.x versions instead aren't currently supported). The .dll (Windows) or .so (Linux) files can be renamed to .pyd Python modules and imported in Python like any other module (that is using: *import XGL_DPP*).

All the conversions between C++ types and Python types are handled by the library and the Python user just need to call the functions of the library using standard Python syntax and types.

However the standard interface of the library make extensive use of strings or integer arguments passed by reference, arrays. In Python strings and integers are *immutable* types, and so they cannot (easily) be passed as reference arguments for functions.

So specialized versions of each function has been declared to better match the Python philosophy. For each function has been declared the pyxxx equivalent (for example *pyInitLibrary*) in which the main difference is that arguments are only passed by value, or when a reference is passed it will be a const reference, in the sense that won't be modified (like string inputs). In other words all the arguments are inputs to the function.

The outputs of the function will be returned in the return value, that will be just a bool for some function, while others that needs to return more data types will return a tuple. A tuple is a Python standard type that implements an immutable list. So reference or raw pointers arguments of the standard interface are moved in the return value for Python interface.

In the next sections we will go through each function definition.

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

Managing controllable systems and library

See *Exported Functions – Managing controllable systems and library* section for the introduction of the following set of functions.

pyInitLibrary

Declaration:

```
def pyInitLibrary ():

return bool(ret_val)
```

Used to initialize the library. Call this function before any other function, otherwise the other functions will always return false and set **DLL_NOT_INITIALIZED** error.

Parameters:

None

Return value:

A boolean, true, if the library is initialized, false if a problem occurs.

pyCloseLibrary

Declaration:

```
def pyCloseLibrary ():

return bool(ret_val)
```

Used to close the internal library resources. Call this function before unloading the library from memory, otherwise with some softwares (e.g. with NI LabVIEW) you will not be able to re-initialize the library unless restarting the software itself.

Parameters:

None

Return value:

A boolean, true, if the library closed successfully, false if a problem occurs.

pygetLastError

Declaration:

```
def pygetLastError ():

tuple(bool(ret_val), uint16_t(error_code))
```


XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

Returns the last detected error. Note that the error code is not reset between function calls. It always reflects the last detected error. **For example, if a function fails and the next function works correctly, the returned error code is the one set by the first function call.**

Parameters:

None

Return values:

bool(ret_val)

A boolean, true, if the returned error_code is valid. Returns false and **DLL_MULTI_THREAD_ERROR** if **getLastError** is called while another thread is working with the library.

Uint16(error_code)

An integer containing the error code. See section Exported functions for the complete error code list.

pyresetLastError

Declaration:

```
def pyresetLastError ():

return bool(ret_val)
```

Resets the "last error" variable to **DLL_NO_ERROR**.

Parameters:

None

Return values:

A boolean, true, if the function succeeds.

pylibVersion

Declaration:

```
def pylibVersion ( uint32 version_size ) :

return tuple(bool(ret_val), string(version))
```

Returns the library version string.

Parameters:

uint32_t version_size

An integer set to the maximum length of the version string. If the length is not enough the function returns false and do not set the string return value.

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

Return values:

Bool(ret_val)

True if the returned value is correct, false if a problem occurs (probably string size is not enough).

String(version)

pyadd_to_query

Declaration:

```
def pyadd_to_query ( string(address) ):

return bool(ret_val)
```

Add an IP address to the query.

Parameters:

String(address)

A string containing the IP to be added (for example **"192.168.1.120"**).

Return values:

True if the parameter is correct, false if a problem occours.

pyremove_from_query

Declaration:

```
def pyremove_from_query ( string(address) ):

return bool(ret_val)
```

Remove an IP address from the query.

Parameters:

String(address)

A string containing the IP to be removed.

Return values:

True if the parameter is correct, false if a problem occurs.

pyget_dev_number

Declaration:

```
def pyget_dev_number () :

return tuple(bool(ret_val), uint16(devs))
```


XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

Get the number of connected devices (chain masters only).

Parameters:

None

Return values:

bool(ret_val)

True if the returned value is correct, false if a problem occurs.

uint16(devs)

The number of master devices connected.

pyget_ids

Declaration:

```
def pyget_ids (uint16(nb),
               uint16(id_size)) :
return tuple(bool(ret_val),string(identifier))
```

Returns devices serial from progressive number 'nb' of connected devices.

Parameters:

uint16(nb)

The progressive number of connected device; nb starts from '0'.

uint16(id_size)

An integer set to the maximum length of the serial string. If the length is not enough the function returns with false and do not set the output string.

Return values:

bool(ret_val)

True if all the parameters are filled with correct values, false otherwise.

string(identifier)

A string that contains the IP of the selected connected device.

pygetFirmware

Declaration:

```
def pygetFirmware ( string(identifier),
                    uint16(Board)):

return uint32(call_id)
```

Ask information about the firmware of a specific system. Asynchronous call.

Parameters:

string(identifier)

A string with the identifier of the system to query.

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

uint16(Board)

It contains the progressive number of the board to query, starting from '0' for the MASTER of the chain.

Return values:

The call ID if the parameters are filled with correct values, 0 otherwise.

Asynchronous answer data:

In the data list of the callback function or in the answers queue you will find, in this order, the Major, Minor, Build of the firmware. Otherwise, if the call was not successful, the type field will be 0.

pyget_boards_in_chain

Declaration:

```
def pyget_boards_in_chain ( string(identifier),
                            uint16(devs)):

return tuple(bool(ret_val),uint16(devs))
```

Get the number of devices in the chain controlled by the specified master board.

Parameters:

string(identifier)

A string with the identifier of the system to query.

Return values:

bool(re_val)

True if all the parameters are filled with correct values, false otherwise.

uint16(devs)

The number of devices in the chain.

pywrite_IP_configuration

Declaration:

```
def pywrite_IP_configuration ( string(identifier),
                               string(IP),
                                string(subnet_mask),
                               string(gateway)):

return uint32(call_id)
```

Write a new IP configuration to the specified DPP board.

Parameters:

string(identifier)

A null-terminated string with the identifier of the system to query.

string(IP)

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

String of the IP.

`string(subnet_mask)`

String of the subnet mask

`string(gateway)`

String of the gateway.

Return values:

ID code of the reply.

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

Configuration

This functions are used to configure the system parameters to get the best acquisition results.

pyconfigure

Declaration:

```
def pyconfigure ( string(identifier),
                  uint16(Board),
                  configuration(cfg)):

return uint32(call_id)
```

Configures the system with the required acquisition configuration. Asynchronous call.

Parameters:

string(identifier)

A null-terminated string with the identifier of the system to query.

uint16(Board)

It contains the progressive number of the board to query, starting from '0' for the MASTER of the chain.

configuration(cfg)

An object of configuration class which defines the base system configuration.

The structure is initialized with this syntax:

```
cfg = XGL_DPP.configuration()
cfg.fast_filter_thr = 100
...
```

And it is defined with this fields:

```
class configuration :
       uint32(fast_filter_thr)
      uint32(energy_filter_thr)
       double(max_risetime)
       double(gain)
       uint32(peaking_time)
       uint32(flat_top)
       uint32(edge_peaking_time)
      uint32(edge_flat_top)
       uint32(reset_recovery_time)
      double(zero_peak_freq)
      uint32(baseline_samples)
      bool(inverted_input)
      double(time_constant)
      uint32(other_param)
```

Where:

- **fast_filter_thr** is the detection threshold for the fast filter (unit: spectrum BIN, range: 0 to 4096).
- **energy_filter_thr** is the detection threshold for the energy filter (currently not implemented, its value it's ignored).
- **max_risetime** is the maximum expected risetime of the detector, used for pileup rejection (unit: 8 ns samples, range: 0 to 127). Set to '0' to disable pileup rejection.
- **gain** is the main gain (unit: spectrum BIN / ADC's LSB, range: 0 to 1.99).
- **peaking_time** is the main energy filter peaking time (unit: 32 ns samples, range: 1 to 500)
- **flat_top** is the main energy filter flat top time (unit: 32 ns samples, range: 1 to 15)
- **edge_peaking_time** is the peaking time of the fast filter (unit: 8 ns samples, range: 1 to 31).
- **edge_flat_top** is the peaking time of the fast filter (unit: 8 ns samples, range: 1 to 15).
- **zero_peak_freq** is the frequency of the zero peak measurement (unit: kcps, range: 1 to 501).
- **baseline_samples** is a setting for baseline correction (unit: 32 ns samples, range: 0,16,32,64,128,256 or 512).
- **time_constant** it's a parameter currently not implemented, it's value its ignored.
- **reset_recovery_time** is the reset recovery time (unit: 8 ns samples; range: 0 to 224-1)
- **other_param:**
  - bit 0: enable gating
  - bit 1: edge of gating
  - bit 5 to bit 2: OCR limit setting. Supported only by ListWave acquisition mode.

Bit 5 to bit 2	OCR limit	**Bit 5 to bit 2**	OCR limit
x0	Disabled	x8	8 KHz
x1	100Hz	x9	10KHz
x2	200 Hz	xA	20 KHz
x3	400 Hz	xB	40 KHz

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

x4	800 Hz	xC	80 KHz
x5	1 KHz	xD	100 KHz
x6	2 KHz	xE	200 KHz
x7	4 KHz	xF	Disabled

Return values:

The call ID if the parameters are filled with correct values, 0 otherwise.

Asynchronous answer data:

In the data array of the callback function or in the answers queue you will find a 1 if the operation succeeded. Otherwise, if the call was not successful, the type field will be 0.

pyconfigure_offset

Declaration:

```
def pyconfigure_offset ( string(identifier),
                         uint16(Board),
                         configuration_offset(cfg)):

return uint32(call_id)
```

Configures the timestamp delay with the required acquisition configuration. Asynchronous call.

Parameters:

string(identifier)

A string with the identifier of the system to query.

uint16(Board)

It contains the progressive number of the board to query, starting from '0' for the MASTER of the chain.

configuration_offset(cfg)

An object of a structure that defines the base system configuration.

The structure is defined with this fields:

```
class configuration_offset :
            uint32(offset_val1)
            uint32(offset_val2)
```

Where:

- offset_val1 is the offset of digpot1 (range: 0 to 255).

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

- offset_val2 is the offset of digpot1 (range: 0 to 255).

Return values:

The call ID if the parameters are filled with correct values, 0 otherwise.

Asynchronous answer data:

In the data array of the callback function or in the answers queue you will find a 1 if the operation succeeded. Otherwise, if the call was not successful, the type field will be 0.

Acquisition control

These functions are used to control the acquisition state of each system.

pyisRunning_system

Declaration:

```
def pyisRunning_system ( string(identifier),
                         uint16(Board) ):

return uint32(call_id)
```

Ask a board if it is running (acquisition in progress). Asynchronous call.

Parameters:

string(identifier)

A string with the identifier of the system to query.

uint16(Board)

It contains the progressive number of the board to query, starting from '0' for the MASTER of the chain.

Return values:

The call ID if the parameters are filled with correct values, 0 otherwise.

Asynchronous answer data:

In the data array of the callback function or in the answers queue you will find 1 if the board is in running and 0 if not. Otherwise, if the call was not successful, the type field will be 0.

pystart

Declaration:

```
def pystart ( string(identifier),
              double(time),
              uint16(spect_depth)):

return uint32(call_id)
```

Start a single spectrum for a specified amount of time or in free-running mode. Asynchronous call.

Parameters:

string(identifier)

A string with the identifier of the system to query.

double(time)

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

The amount of time the acquisition should run, expressed in seconds. The system precision is up to 0.001 secs (1 msec) for measurements from 1 msec to 1 hour. The system stops automatically after the specified time elapses, but the function stop need to be called anyway before starting another measure.

Use 0 if a free-running measure is required (the system never stops). To stop a free running measure call the stop function.

uint16(spect_depth)

The bins number of each spectrum. This value can be 1024, 2048 or 4096. No other value are allowed. This setting affect also the amount data returned by the getData function.

Return values:

The call ID if the parameters are filled with correct values, 0 otherwise.

Asynchronous answer data:

In the data array of the callback function or in the answers queue you will find 1 if the function succeeded. Otherwise, if the call was not successful, the type field will be 0.

pystop

Declaration:

```
def pystop ( string(identifier) ):

return uint32(call_id)
```

Stops a running acquisition. Asynchronous call.

Parameters:

string(identifier)

A string with the identifier of the system to query.

Please note that if the system is not in free-running mode, the system stops automatically after the specified time elapses, but the function stop need to be called anyway before starting another measure.

Return values:

The call ID if the parameters are filled with correct values, 0 otherwise.

Asynchronous answer data:

In the data array of the callback function or in the answers queue you will find 1 if the function succeeded. Otherwise, if the call was not successful, the type field will be 0.

pyclear

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

Declaration:

```
def pyclear ( string(identifier) ):

return bool(re_val)
```

Clears acquisition data, run time and statistical data.

Parameters:

string(identifier)

A string with the identifier of the system to query.

Return values:

A boolean true if the data is cleared, false otherwise.

Retrieving acquired data

pygetAvailableData

Declaration:

```
def pygetAvailableData ( string(identifier),
                         uint16(Board):

return tuple(bool(ret_val),uint32(data_number))
```

Get the number of available spectra in map acquisition mode or number of events in list mode. In case of list_wave mode the function returns both the number of event and the waveform. This function should be called before reading data in mapping mode and list mode in order to know how much space it needs to be allocated.

Parameters:

string(identifier)

A string with the identifier of the system to query.

uint16(Board)

It contains the progressive number of the board to query, starting from '0' for the MASTER of the chain.

uint32(data_number)

The number of available spectra in mapping mode or the number of events available for reading while in list mode. In case of list-wave acquisition, every four waveform samples, the data_number value increases by 1.

Return values:

bool(ret_val)

A boolean true, if the system started with the set parameters, false otherwise.

uint32(data_number)

The number of available spectra in mapping mode or the number of events available for reading while in list mode. In case of list-wave acquisition, every four waveform samples, the data_number value increases by 1.

pygetData

Declaration:

```
def pygetData ( string(identifier),
                uint16(Board),
                uint64_t* values,
                uint32_t& id,
                statistics& stats,
                uint32(spectra_size)) :
```


XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

```
return tuple(bool(ret_val),uint64_vector(values),uint32(id),statistics(stats)
```

Gets the acquired data. It should be used for normal DPP mode (single spectrum), waveform mode and list mode acquisitions. It can be used also during a map acquisition but it will return only the last complete spectrum acquired (use **getAllData** to retrieve all the spectra available).

Parameters:

string(identifier)

A null-terminated string with the identifier of the system to query.

uint16(Board)

It contains the progressive number of the board to query, starting from '0' for the MASTER of the chain.

The meaning of the remaining parameters depends on the acquisition mode selected:

Normal acquisition mode (single spectrum) or Mapping mode

uint32(spectra_size)

The size of the spectrum. It must be 1024, 2048 or 4096.

List mode acquisition

uint32(spectra_size)

The number of events the caller want to read. They have to be equal or less the available stored events. Use the function **getAvailableData** in order to know the number of counts available.

Return values:

bool(ret_val)

Returns true if the values array is filled with correct values, false otherwise. The function return false also if there is no data available to be read.

The meaning of the remaining return values depend on the acquisition mode selected:

Normal acquisition mode (single spectrum) or Mapping mode

uint64_vector(values)

Vector of uint64 (64 bit wide) containing the values of each bin of the spectrum (the spectrum length is programmable and can be 1024, 2048 or 4096). The vector can be indexed as a standard Python list.

uint32(id)

Each spectrum acquired has a unique progressive identifier returned with this parameter. The id is reset only after a power cycle.

statistics(stats)

An object of a class containing the statistics associated to the acquisition.

In this structure is reported the real time (microseconds), the live time (microseconds), the input and output count rate (kcps) of the acquisition. In addition are given also some advanced statistics about pileup rejection and other aspects. For more information about them contact us.

The structure is defined with this fields:

```
class statistics :
      // Basic statistics.
      uint64(real_time)
      uint64(live_time)
      double ICR;
      double OCR)
      // Advanced statistics.
      uint64(last_timestamp)
      uint32(detected)
      uint32(measured)
      uint32(edge_dt)
      uint32(filt1_dt)
      uint32(zerocounts)
      uint32(baselines_value)
      uint32(pup_value)
      uint32(pup_f1_value)
      uint32(pup_notf1_value)
      uint32(reset_counter_value)
```

List mode acquisition

uint64_vector(values)

Vector of uint64 (64 bit wide), which can be indexed as a standard Python list, containing information about each single event recorded by the DPP. Each 64 bit value correspond to an event and is organized in this way:

- bits 63 to 20: Timestamp of the X-Ray arrival time (10 ns samples).
- bits 19 to 16: Energy filter number.
- bits 15 to 0: Energy of the acquired X-Ray.

uint32(id)

Each spectrum acquired has a unique progressive identifier returned with this parameter. The id is reset only after a power cycle.

statistics(stats)

An object of a class containing the statistics associated to the acquisition (same structure of normal acquisition mode).

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

Remarks:

In normal acquisition mode (spectrum) negative values of measured energies (actually noise) are summed up at the first bin. Instead measured energies that exceed the spectrum range (i.e. larger than the last bin) are summed up at the last bin.

pygetAllData

Declaration:

```
def getAllData ( const char* identifier,
                 uint16_t Board,
                 uint16_t* values,
                 uint32_t* id,
                 double* stats,
                 uint64_t* advstats,
                 uint32_t& spectra_size,
                 uint32_t& data_number):

return tuple(bool(ret_val), uint16_vector(values), uint32_vector(id),
double_vector(stats), uint64_vector(advstats))
```

Get a data_number of spectra during a mapping mode acquisition. This function must not be used while in other acquisition modes, use **getData** instead.

Parameters:

string(identifier)

A string with the identifier of the system to query.

uint16(Board)

It contains the progressive number of the board to query, starting from '0' for the MASTER of the chain.

uint32(spectra_size)

The size of the spectrum. It must be 1024, 2048 or 4096.

uint32_t(data_number)

The number of spectra requested. It must be equal or less than the number of spectra available. The user should call the **getAvailableData** function first to allocate enough space for spectra and statistics.

Return values:

bool(ret_val)

Returns true if the values array is filled with correct values, false otherwise. The function return false also if there is no data available to be read.

uint16_vector(values)

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

Array of uint16_t (16 bit wide) containing the counts of each bin (the spectrum length is programmable and can be 1024, 2048 or 4096) of each spectrum. The spectra are one next to the other starting from the first acquired. So, for example, if 10 spectra are committed, the resulting values length will be 10*spectrum_length.

`uint32_vector(id)`

Array of unique progressive identifiers of each spectrum. The first id of the array correspond to the first spectrum acquired (as for the parameter values).

`double_vector(stats)`

Array of basic statistics of each spectrum. The statistics give here are (in this order): real time (microseconds), live time (microseconds), ICR (kcps), OCR (kcps). The first statistics correspond to the first spectrum acquired. So, the size of the vector will be 4*data_number.

`uint64_vector(advstats)`

Array of advanced statistics for each spectrum. The size of this array will be 24*data_number. For more information about them contact us.

Index *i * 18 + 13* of *adv_stats* (starting from index 0, *i* spectra number) contains all the gating sampled values if enabled by the configure command

Remarks:

In normal acquisition mode (spectrum) negative values of measured energies (actually noise) are summed up at the first bin. Instead, measured energies that exceed the spectrum range (i.e. larger than the last bin) are summed up at the last bin.

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

Examples

These are simple C/C++ examples/code tips to demonstrate the use of the library.

Please note that this are really simple examples to illustrate the use of the functions.

Initialization

Just call the ***InitLibrary()*** function. If it will return false, the library is not correctly initialized.

```
#include <iostream>

#include "XGL_DPP.h"

if (!InitLibrary()) {
	std::cout << "ERROR: Library not initialized." << std::endl;
}
```

Add IPs to Query

In order to allow the library to detect connected DPPs you have to update the query, which is initially empty.

```
if (add_to_query("192.168.1.120")) {
	std::cout << "Added 192.168.1.120 to the query." << std::endl;
}

if (add_to_query("192.168.1.121")) {
	std::cout << "Added 192.168.1.120 to the query." << std::endl;
}

if (add_to_query("192.168.1.122")) {
	std::cout << "Added 192.168.1.120 to the query." << std::endl;
}
```

The library will then try to connect to this IPs.

Enumerate connected boards and serial numbers

Among the devices in the query, get information about the ones which are connected.

```
uint16_t devices = 0;

get_dev_number(devices);

std::cout << "Found devices: " << devices << std::endl;
```

Then, prepare the array buffer to get the identifiers (that is the IPs) of the connected MASTER boards:

```
char* identifier(new char[16]);
```


XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

```
uint16_t id_size = 16;
```

Ask for each board the number of SLAVES for each master:

```
for (uint16_t i = 1; i <= devices; i++) {
       get_ids(i,identifier,id_size);
       std::cout << "Device: "<< i << " - Identifier: " << identifier << std::endl;
       uint16_t slaves = 0;
       get_boards_in_chain(identifier, slaves);
       std::cout << "Slaves: " << (slaves - 1) << std::endl;
}
```

Here we used ***get_ids()*** with a progressive number to query the identifier for each MASTER (using the array prepared at the previous point). This identifier may be stored in a global user-array to be used later to call the required function on each chain of boards connected to the system.

Then, for each MASTER, we ask the number of boards in the chain, with ***get_boards_in_chain()***. We must subtract '1' from the returned number to get only the number of SLAVES (without the MASTER).

Get information about the boards

We can ask for the firmware number, if needed:

```
uint32_t firm_call_id;

firm_call_id = get_firmware(serial, 0); // Firmware asked to master board.
```

Then we can rapidly continue doing other tasks. The library give us the answer using the callback if using the Callback library:

```
void callback_function (uint16_t type,
uint32_t call_id,
uint32_t length,
uint32_t* data)
{
   if (call_id == firm_call_id)
   {
       if (type == 1 && length == 3) // As should be for getFirmware call.
       {
       uint16_t Major = data[0];
       uint16_t Minor = data[1];
       uint16_t Build = data[2];
       std::cout << "Firmware (MASTER): "<<Major<<"."<<Minor<<"."<<Build;
       }
    }
}
```

Otherwise, if using the Polling release, we have to get the answer when available.

```
uint16_t answers_length = 0;
```


XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

```
do {
	// Other tasks.
	. . .

	GetAnswersDataLength(answers_length);
}
while (answers_length < 6)

// We have some data in the queue.
uint32_t answer[6];
GetAnswersData(6,answer);
if (answer[0] == firm_call_id)
{
	uint32_t type = answer[1];
	uint32_t length = answer[2];
	uint16_t Major = answer[3];
	uint16_t Minor = answer[4];
	uint16_t Build = answer[5];
	std::cout << "Firmware (MASTER): "<<Major<<"."<<Minor<<"."<<Build;
}
```

Configure the DPP

First declare the configuration structure:

```
configuration config;
config.fast_filter_thr = 80;
config.max_risetime = 30;
config.baseline_samples = 64;
config.gain = 0.43;
config.peaking_time = 25;
config.flat_top = 5;
config.edge_peaking_time = 1;
config.edge_flat_top = 1;
config.reset_recovery_time = 300;
```

And then configure the DPP:

```
uint32_t config_call_id = configure(identifier, Board, &config, 0);
```

Get answer (Callback):

```
void callback_function (uint16_t type,
uint32_t call_id,
uint32_t length,
uint32_t* data)
{
    if (call_id == config_call_id)
    {
	if (type == 2 && data[0] == 1) // As should be for successful configure
call.
	{
	std::cout << "DPP configured." << std::endl;
	}
    }
}
```

Get answer (Polling):

```
uint16_t answers_length = 0;
```


XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

```
do {
        // Other tasks.
        . . .

        GetAnswersDataLength(answers_length);
}
while (answers_length < 4)

// We have some data in the queue.
uint32_t answer[4];
GetAnswersData(4,answer);
if (answer[0] == config_call_id && answer[3] == 1)
{
        std::cout << "DPP configured." << std::endl;
}
```

Start an acquisition (Normal acquisition mode)

Here we use the start function to start an acquisition for all the boards in a specific chain:

```
double time = 0.1; // secs.
uint32_t start_call_id = start(identifier,time);
```

Then get the answer in the usual ways. During the acquisition we can check for the boards status, with something like this:

```
bool running = true;
while (running) {
    // Query MASTER board.
    uint32_t run_call_id = isRunning_system(identifier, 0);
    // Get the answer from the callback or the queue and update the running
variable.
    // If answer not ready do something else...
    // Otherwise.
       if (running)
       {
        std::cout << "Chain with MASTER serial “ <<identifier << “ is running...”;
        std::cout << std::endl;
       }
    }
std::cout << "Acquisition ended.“ << std::endl;
```

Read acquired data (Normal acquisition mode)

First of all, prepare the variables to contain acquired data:

```
uint64_t data[4096];
uint32_t spectra_size = 4096;
uint32_t id;
statistics stats; // The statistics structure is declared in the header file.
```

Then, call the function to populate the variables with the real acquired data:

```
for (uint16_t brd = 0; brd < slaves + 1; brd++) {
       getData(serial, brd, &data[0], id, stats, spectra_size);
       // Plot/store the data somewhere...
}
```


XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

Start an acquisition (Waveform mode)

Here we use the start function to start an acquisition for all the boards in a specific chain:

```
uint16_t mode = 0; // The only allowed mode up to now.
uint16_t dec_ratio = 10; // Acquire one 100MHz sample every ten.
uint32_t trig_mask = 2; // Rising edge trigger.
uint32_t trig_level = 30000; // Trigger value.
double time = 1; // Seconds to wait the triggering event.
uint16_t length = 5; // 2^14 samples * 5

uint32_t wave_call_id = start_waveform(serial, 0, dec_ratio, trig_mask,
trig_level, time, length);
```

Then get the answer in the usual ways.

Read acquired data (Waveform mode)

Prepare the variables to contain acquired data, using the length parameter previously set:

```
uint64_t data[16384*length];
uint32_t data_size = 16384*length;
```

These ones are not used in this mode.

```
uint32_t id;
statistics stats;
```

Then, call the function to populate the array with the real acquired data:

```
for (uint16_t brd = 0; brd < slaves + 1; brd++) {
       getData(serial, brd, &data[0], id, stats, data_size);
       // Plot/store the data somewhere...
}
```

Start an acquisition (List mode acquisition)

Here we use the start function to start an acquisition for all the boards in a specific chain:

```
double time = 0.1; // secs.
uint32_t start_call_id = start(identifier,time);
```

Then get the answer in the usual ways. During the acquisition we can check for the boards status, with something like this:

```
bool running = true;
while (running) {
    // Query MASTER board.
    uint32_t run_call_id = isRunning_system(identifier, 0);
    // Get the answer from the callback or the queue and update the running
variable.
    // If answer not ready do something else...
    // Otherwise.
    if (running)
    {
       std::cout << "Chain with MASTER serial “ <<identifier << “ is running...”;
       std::cout << std::endl;
    }
}
```


XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

```
std::cout << "Acquisition ended.“ << std::endl;
```

Read acquired data (List mode acquisition)

Check firstly how many events are available to be read:

```
uint32_t data_number;
getAvailableData(serial, brd, data_number)
```

Basing on the data_number value, allocate an array with the correct length:

```
uint64_t* values = new uint64_t[data_number];
```

Finally, call the function to populate the array with all available event data:

```
getData(serial, brd, values, id, stats, data_number);
```

Start an acquisition (Mapping mode)

Start an acquisition with the desired number of spectra and time per spectrum.

```
double sp_time = 100; // msecs.
uint32_t points = 10;
uint32_t start_call_id = start(identifier,sp_time,points);
```

Then get the answer in the usual ways.

Read acquired data (Mapping mode)

Check firstly how many spectra are available to be read:

```
uint32_t spectra_number;
getAvailableData(serial, brd, spectra_number)
```

Basing on the data_number value, allocate arrays with the correct lengths:

```
uint16_t* values = new uint16_t[spectra_number*4096];
uint32_t* id = new uint32_t[spectra_number];
double* stats = new double[spectra_number*4];
uint64_t* advstats = new uint64_t[spectra_number*24];
```

Finally, call the function to populate the arrays with all available spectra data:

```
getAllData(serial, brd, values, id, stats, advstats, 4096, spectra_number);
```

Using the Library with NI LabVIEW

It's also possible to use the DANTE Library with National Instruments' Labview.

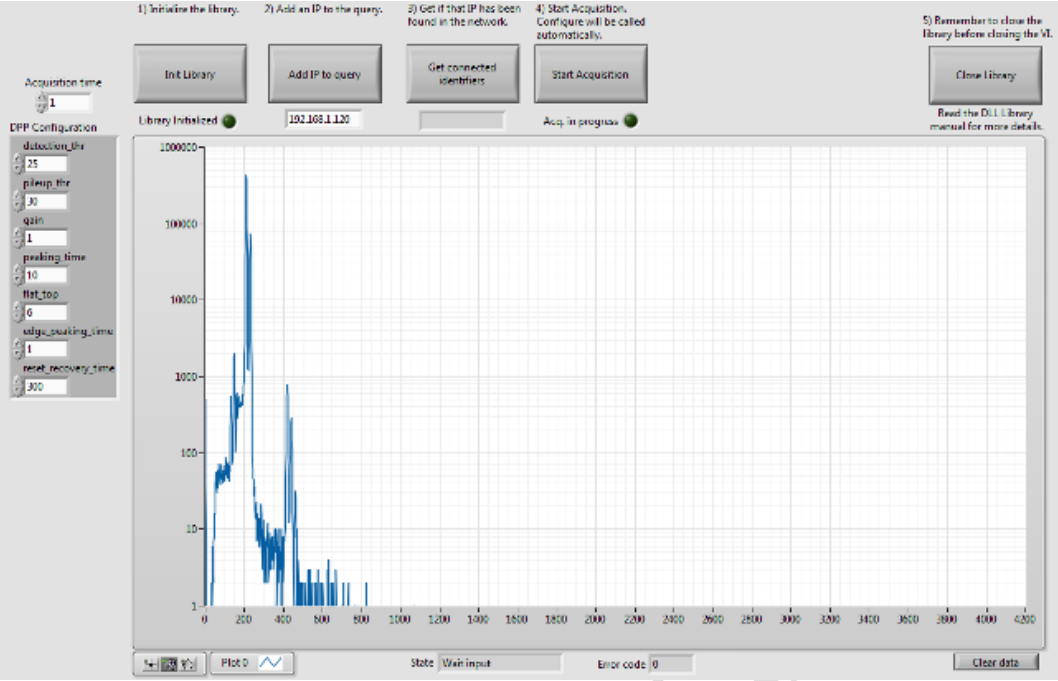
We supply a LabVIEW project ('DANTE – Example.lvproj') that contains: a set of predefined Labview sub-VIs already configured for each library call (in the folder 'VIs') and a basic acquisition example that shows how this subVIs can be used to make a single spectrum acquisition mode (via TCP-IP). When using the Acquisition example the library have to be present in the same folder of the VI. At the moment waveform, list and mapping mode are not present in the example but can be easily added by the user thanks to the subVI library.

This subVIs are simple wrappers of the "Call Library funcion" nodes of LabVIEW, adapted after using the "Import > Shared Library (.dll)" tool. If your version of Labview does not import the example interface provided, you may use this tool with the provided header file (XGL_DPP.h) to re-create this sub-VIs, but please note that this tool does not correctly recognize some parameters of the function calls. In particular:

- The boolean type is traduced as Labview "Unsigned 8-bit integer" and is '0' for false and any other value for true.
- Any pointer, reference and array must be manually set to "Array" or "Pointer to Value", with the appropriate type in each "Call Library Function" node that is used. Labview creates input and output variables for each call. Some "input" variables are useless for some functions.
- Some functions set or get C-style arrays. You must pass to the function as input variable a previously initialized LabVIEW array or the function call may fail. Note that in these cases the function calls also uses some parameters of the call to get the dimension(s) of the array: pay attention for some multi-dimensional arrays used.
- The most important type-mappings of C-types to Labview-types are:
  - C-style char arrays map to Labview's "String" type.
  - C double maps to Labview's "8-byte Double" type.
  - Uint32_t maps to Labview's "Unsigned 32-bit Integer".
  - Uint16_t maps to Labview's "Unsigned 16-bit Integer".
  - As stated above Bool maps to Labview's "Unsigned 8-bit Integer" and must be compared against '0': 0 means false, any other value means true.

The use of the example is very straightforward. It consists of a state machine activated by event of some push buttons.

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)



The first thing the user should do just after the run of the VI is to initialize the VI using the 'Init Library' button. If this or one of the other operations fails the VI will exit immediately reporting the error code in the bottom right.

Then the user can Add an IP to the query and get if that board is connected.

Once set the desired acquisition and DPP configuration in the left, the user can launch an acquisition with the 'Start Acquisition' button and after the data retrieval the spectrum will be showed on the graph.

WARNING:

It is important not to terminate the execution with the stardard 'Abort Execution' button in the VI toolbar but pushing the button 'Close Library'. This will first call the CloseLibrary function of the library and the terminate the execution. The reason of doing this is that, once loaded, the DLL will be unloaded by LabVIEW as soon as the VI is closed (not when the execution is terminated but when the VI window is closed). The DLL uses some resources that need to be closed prior to unload the library otherwise a LabVIEW crash will occurr or the next time InitLibrary is called it could fail.

Keeping this in mind the user can abort the execution of the VI without using the Close Library command and then relaunch it without the need of re-calling InitLibrary, but the Close Library button must be used before closing the VI.

Manual revisions

Revision 2.4:

Added Python support.

Revision 2.3:

Added Linux support.

Revision 2.2:

Changing configuration of timestamp_delay

Revision 2.1:

Added limiting of OCR

Revision 2.0:

Added list-wave mode functions.

Revision 1.9:

Update configure, getData and getAllData to reflect changes for new 3.0.x firmware versions.

Revision 1.8:

Update configure function structure with other_param

Revision 1.7:

Updated configure function structure. For library version 2.1.12.

Revision 1.6:

Added configure_timestamp_delay function and programmable spectrum binning.

Revision 1.5:

Manual layout reviewed.

XGLab S.R.L.
Via Conte Rosso 23, I-20134 Milano (Italy)

Revision 1.4:

Added getWaveData function. For library version 2.1.6.

Revision 1.3:

Removed acquisition status for getData functions and added error codes. For library version 2.1.5.

Revision 1.2:

Added LabVIEW Example. For library version 2.1.2.

Revision 1.1:

Full USB support and new error codes. For library version 2.1.2.

Revision 1.0:

Initial release. For library version 2.0.2.