Software to work with OWON oscilloscope's binary files

LRDPRDX

November 8, 2019

Contents

1 Introduction

If you have **OWON** oscilloscope you are probably able to use its official software to communicate your scope with PC. That software, among others, allows user to record waveforms from his/her device. Particularly, it allows to record data continuously (event by event). In the latter case data is stored in binary files. However, it seems that **OWON** doesn't provide any information about the structure (protocol) of those binary files, and though user can read the data back by the software which produced it (data), he/she is not able to use it for further (offline) analysis. This software is intended to provide such possibility.

Only oscilloscopes of the following models are verified to be parsed correctly with this software:

- TDS series
  - TDS8204
- SDS series
  - SDS8302
  - SDS6062
  - SDS7072

WARNING: If your oscilloscope is not in the above list you should consider to not use this software: even if it is parsed without errors the content may be incorrect. However, see section 4

2 Installation

Everywhere in the below text it is assumed that the root directory of this software is **<OWON_DIR>**.

2.1 Stand-alone

In order to install the library you should compile dynamic library (shared object) and place it in standard location (**/usr/lib/** in this case). The instructions are the following:

```
cd <OWON_DIR>/make
make compile
sudo make link
make clean
```

Also it is useful to extend the standart include path with the path to the headers. It allows you, for example, not use the include option (**-I**) while compiling. Place the following lines in your **.profile**, or **.bash_profile** if that is what you have.

```
if [[ -n "$CPLUS_INCLUDE_PATH" ]]; then
    CPLUS_INCLUDE_PATH=$CPLUS_INCLUDE_PATH:/path/to/<OWON_DIR>/inc
else
    export CPLUS_INCLUDE_PATH=/path/to/<OWON_DIR>/inc
fi;
```

where **/path/to/<OWON_DIR>** is the absolute path to the **<OWON_DIR>**.

Now logout and login back. That's all!

2.2 ROOT-compatible

Read this section only if you intend to use ROOT CERN framework to create **TTree**s from several binary file. For more details, see Sec. 3.2. At this stage you need to know the location of the following library in your system:

```
libstdc++fs.a
```

You could find it, for example, by using the **find** command:

```
find /usr/lib -name "libstdc++fs.a"
```

In my case it prints

```
/usr/lib/gcc/x86_64-linux-gnu/5/libstdc++fs.a
```

Once you know the location place it in the **<OWON_DIR>/make/CompileROOT.C** file (line 2 in the below code):

```
1 {
2     gSystem->AddLinkedLibs( "/usr/lib/gcc/x86_64-linux-gnu/5/libstdc++fs.a" );//place the correct
          path to the libstdc++fs.a
3     gSystem->AddLinkedLibs( "/usr/lib/libowonparse.so" );
4     gSystem->AddIncludePath( "-I../inc");
5     gROOT->LoadMacro( "../src/ROOT/TreeCreator/TreeCreator.cpp+" );
6     //gROOT->LoadMacro( "../src/ROOT/Drawer/Drawer.cpp+" );
7 }
```

The final step is to compile everything:

```
cd <OWON_DIR>/make
root -l CompileROOT.C
```

3 Usage

In order to prevent name conflicts and keep global scope clean everything is placed into namespace **owon**. In the below code the **using** declaration is *not* assumed so each entity of the library appears with the **owon::** prefix. However, it would be useful to make a namespace alias to type less (see below).

This chapter covers two forms of usage:

- as a stand-alone pure C++ software
- as a part of the software based on the **ROOT CERN** framework

3.1 Stand-alone

This software is used to get two kinds of information about a waveform: *primary* and *secondary*. Primary information consists of data encoded in a binary file itself such as *voltage per division, number of channels, data points, etc.* Secondary information contains results of some analysis of data points: *baseline level, amplitude, integral, etc* and, obviously, based on the primary one.

Terminology

Extracting primary and secondary information from a binary file is called *parsing* and *analysis*, respectively, in this document.

3.1.1 Parsing

To parse a binary file the following algorithm should be executed:

1. Choose a model namespace
2. Create an instance of the **parser** class
3. Call **parser::parse** member function
4. Choose the channel with **parser::set_active_channel** member function
5. Get information via get-methods

Algorithm 1: Parsing of a binary file

Below each step of the above algorithm is explained in detail.

Model namespace. As it was mentioned above everything is contained in the **owon** namespace. The **owon** namespace itself includes separate namespace for each model with a name the same as the model name (lowercase). Such a namespace will be called *a model namespace* in the below text. To choose model namespace corresponding to your scope you include its header which is

```
device/<model_name>.h
```

where **<model_name>** matches the model name with the letters in lower case. Also it is convenient to create a namespace alias to type less. For example, if your oscilloscope is SDS6062 then the following two lines should be in your code (though the second one is not mandatory):

```
#include "device/sds6062.h"

namespace osc = owon::sds6062;//alias
```

The parser class. A model namespace contains the **parser** class — the class that is responsible for the extraction information (parsing) from binary files. The **parser** class is one of the two (along with the **analyzer** class) most important tools in this software. At least one instance of this class must be created:

```
osc::parser p;//create parser
```

The parser::parse member. Once an instance of the **parser** class is created you use its member function **parse** to extract information from a binary. It takes the only argument — path to the binary file (it may be either absolute or relative, with an extension) to be parsed:

```
p.parse( "path/to/binary/file.bin" );//NOTE: the extension is present
```

Setting the active channel and getting information. After successful (see below) execution of the above code the extracted information is accessible through the get-methods of the **parser** class. However, most of these methods are intended to provide information about *individual* channel: *voltage per division, time per division, data points, etc.* This channel is called the *active channel* in the software. Use **parser::set_active_channel** member function to choose what channel is active:

```
p.set_active_channel( owon::CH1 );//select the channel
//returns voltage per division of the active channel --- CH1
p.get_voltage_per_div();

p.set_active_channel( owon::CH2 );//select the channel
//returns voltage per division of the active channel --- CH2
p.get_voltage_per_div();

//iterate over data points of the active channel --- CH2
for( sds::parser::const_point_iterator point = p.cbegin(); point != p.cend(); ++point )
{
    //do something with points...
    std::cout << "Point is ( " << point->time << ", " << point->voltage << ") \n";
}
```

For the full list of availalble get-methods see Sec. 5.2 (**class oscilloscope**).

3.1.2 Analysis

The **owon** namespace also includes the **analyzer** class. It performs simple but often very useful operations on a waveform: calculation of the baseline level, finding the point of maximum amplitude, etc. Use the following algorithm to analyse a waveform:

1. Create an instance of the **parser** class (see Sec. 3.1.1)
2. Parse a waveform (see Sec. 3.1.1)

3. Create an instance of the **analyzer** class
4. Set the channel to be analyzed as the active one (see Sec. 3.1.1)
5. Set the analysis config if needed
6. Call **analyzer::analyze** member function to perform analysis
7. Call get-methods of the **analyzer** class to get the results

Algorithm 2: Waveform analysis

This class is not autonomous — it works in pair with the **parser** class. Before instantiating the **analyzer** an instance of the **parser** class must have been created. Therefore, the first two steps are required.

```
#include "analyzer.h"
#include "device/sds6062.h"

namespace osc = owon::sds6062;

//somewhere in the code
osc::parser p;
p.parse( "path/to/binary/file.bin" )
...
```

The analyzer class. The constructor of the **analyzer** class takes pointer to a **parser** object as an argument:

```
owon::analyzer a( &p );//create analyzer
```

The active channel. The role of the active channel (see Sec. 3.1.1) in the analysis process is simple — all the analysis is performed on the data points of the active channel.

The analysis config. Next step is to set the analysis config

```
a.set_baseline_time( p.get_trigger_time() );//set the range for baseline calculation from the
    beginning of the waveform to its trigger time
a.set_gate( 0.3, 0.8 );//set the integration range from 300 to 800 ns

a.analyze();
```

Getting the results. Use the get-methods of the **analyzer** class to get the results of analysis:

```
a.get_baseline();
a.get_integral();
```

For the full list of the get-methods see Sec. 5.3 (**class analyzer**).

3.1.3 Compilation

Here is the code of a minimal example and how to compile it.

```
#include <iostream>

#include "device/sds6062.h"
#include "analyzer.h"

namespace osc = owon::sds6062;//alias


int main()
{
    osc::parser p;//create parser
    owon::analyzer a( &p );//create analyzer

    p.parse( "path/to/binary/file.bin" );//NOTE: the extension is present

    if( p.get_status() )//if successfully parsed
    {
        p.print();

        a.set_gate( -1, -1 );//see reference guide
        a.set_baseline_time( -1 );//see reference guide

        a.analyze();
        a.print();
    }
    else
    {
        std::cerr << "Something went wrong" << std::endl;
    }

    return 0;
}
```

Provided you have saved the above code to the file named **example.cpp**, and appended the **CPLUS_INCLUDE_PATH** environment variable as it was recommended in Sec. 2 the compilation is as follows:

```
g++ example.cpp -std=c++11 -lowonparse -o example
```

However, it is more convenient to use **Makefile** to compile your code. You could find examples of simple **Makefile**s in the **<OWON_DIR>/examples** directory.

3.2 ROOT-compatible

In this section it will be explained how to use this library together with the ROOT CERN framework, in particular, how to create a **TTree** from several binary files. Provided you have followed the instructions in Sec. 2.2 the following algorithm should be used to create a **TTree** from a set of a binary files:

1. Create an instance of the **parser** class in the model namespace corresponding to the model of an oscilloscope that was used to record waveforms
2. Create an instance of the **TreeCreator** class using previously created **parser**
3. Set analysis config and the active channel using member-functions of the **TreeCreator** class
4. Call the **CreateTree** member-function

5. Compile and run

Algorithm 3: Creation of a **TTree**

Below is the explanation the above algorithm in detail.

The TreeCreator class. This class is responsible for (no surprise) creation of a **TTree** from To create an instance of this class you, firstly, include its header:

```
#include "ROOT/TreeCreator.h"
```

Then you use the only constructor of this class which takes four arguments:

```
TreeCreator( owon::oscilloscope* osc,
             const std::string& pathToDataDir,
             const std::string& pathToTreeFile,
             const std::string& treeFileName )
```

where

osc — pointer to the **oscilloscope** class (which the **parser** class in each model namespace inherits from) that is of the same model that is used to record the binaries

pathToDataDir — path to the directory containing the binaries (see the NOTE below)

pathToTreeFile — path to the location where the resulting **TTree** will be placed

treeFileName — name of a **.root** file (without an extension) with the resulting **TTree**

The meaning of the second argument requires additional explanation. There may be two cases here. The first is the one when the target directory contains no other directories (subdirectories) — only binary files. I.e. the structure is the following:

```
path
|
|_to
  |
  |_data <--target
     |
     |__file1.bin
     |__file2.bin
     |__...
```

In this case, provided the second argument in the **TreeCreator** constructor was **"path/to/data"**, THE ONLY **TTree** named **tree_** would be constructed from all the binaries in the **data** directory. If, on the other hand, the target directory contains other directories, for example:

```
path
|
|_to
  |
  |_data <--target
    |
    |_data1
    | |
    | |__file1.bin
    | |__file2.bin
    | |__...
    |
    |_data2
      |
      |__file1.bin
      |__file2.bin
      |__...
```

then, provided the second argument in the **TreeCreator** constructor was **"path/to/data"**, TWO **TTree**s named **tree_data1** and **tree_data2** would be constructed from all the binaries in the **data1** and **data2** directory (and in its subdirectories), respectively. It means that hierarchy of the subdirectories of the target directory doesn't matter: all the binaries are searched recursively in a subdirectory of the target directory. I.e. the **subsubdata** is expanded when constructing a **TTree** provided the following structure:

```
path
|
|_to
  |
  |_data <--target
    |
    |_data1 <-- matters
    | |
    | |__file1.bin
    | |__file2.bin
    | |__...
    |
    |_data2 <--matters
      |
      |__file1.bin
      |__file2.bin
      |__...
      |__subsubdata <-- DOESN'T matter: expanded
         |__file1.bin
         |__file2.bin
         |__...
```

However, the path (see below) of the every binary is conserved in the resulting **TTree**.

After the instantiating the **TreeCreator** you should set the analysis config and the active channel (see Sec. 3.1.2) which will be used when constructed the **TTree**:

```
tc.SetBaselineTime( -1 );//from the beginning to the trigger time
tc.SetGate( -1, 3 );//from the beginning to 3 us
tc.SetActiveChannel( owon::CH1 );//the default one
```

After that you call the **CreateTree** member-function. The whole code could be:

```
#include "ROOT/TreeCreator.h"

#include "device/tds8204.h"


void CreateTree()
{
    owon::tds8204::parser p;//OWON TDS8204 was used for recording

    TreeCreator tc( &p, "./Data", "./", "myFirstTree" );
    tc.SetBaselineTime( -1 );
    tc.SetGate( -1, 3 );
    tc.SetActiveChannel( owon::CH1 );//the default one

    tc.CreateTree();
}
```

Compile and run. To run the above example create (or append) the **rootlogon.C** file in the directory where you put this example with the following line:

```
{
    gSystem->AddLinkedLibs( "path/to/<OWON_DIR>/src/ROOT/TreeCreator/TreeCreator_cpp.so" );
}
```

where **/path/to/<OWON_DIR>** is the absolute path to the **<OWON_DIR>**.

Then type the following in your working directory (it is assumed you have saved the code in the file **CreateTree.C**):

```
root -l CreateTree.C+
```

After that there will appear the file (along with the others) named **myFirstTree.root** in the current directory with a **TTree**. The full example (with the demo data) see in the **<OWON_DIR>/example/TDS/ROOT/create_tree** directory.

4 Feedback

Please, send bug reports and suggestions to **paradox1859@gmail.com**. In case if the model of your oscilloscope is not in the list of available models (see Sec.1) you can ask me to write a parser for your model. Just send me a letter with this request.

5 Reference guide

This reference guide is not considered to be complete. In this section only **public** members are explained. This is rather a *usage reference guide*.

NOTE: The **owon::** prefix is omitted in the below code

Terminology

Object of each class listed below has *a state*. In this context a state of an object is simply a set of values of its data members.

5.1 namespace owon

Description

The library is entirely contained in the **owon** namespace. The schematics of the library structure is shown in Fig. 1.

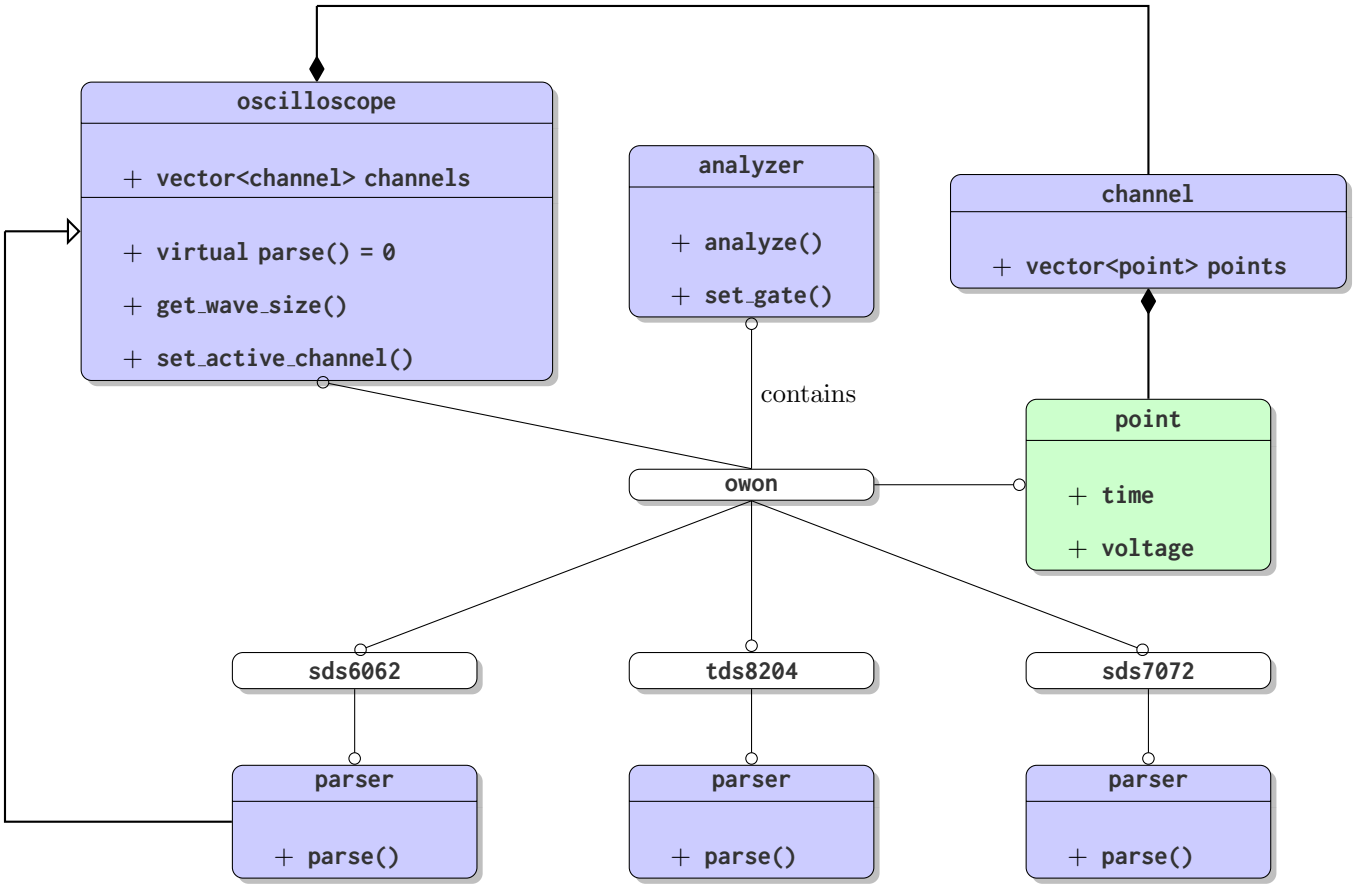

Figure 1: Structure of the library. Colorless rectangles — **namespace**s, green — **structure**s, blue — **class**es. Open triangle arrows — inheritance, diamonds — aggregation, circles — containing

enums

```
enum CHANNEL : unsigned { CH1, CH2, CH3, CH4 };
```

Description:	Represents the list of available channels
Values:	**CH1, CH2, CH3, CH4**

Functions

```
inline std::string ch_to_str( const CHANNEL ch );
```

Description:	Represents the list of available channels
Arguments:	**ch** - channel value
Return:	string representation of the channel

```
inline std::ostream& operator<<( std::ostream& os, const CHANNEL& ch )
```

Description:	Allows user to use stream insertion operator **<<** to print channel values
Usage example:	**std::cout << owon::CH2 << std::endl;**

strusts and classes

```
struct point
```

Description:	Represents a single data point. Each channel (**channel** class) contains a vector of points (**std::vector<point>**) and each oscilloscope (**oscilloscope** class) has an iterator (**const_point_iterator**) that iterates over the points of its active channel
Fields:	**float time** - time mark of a data point (in μs) **float voltage** - voltage mark of a data point (in mV)

```
class channel
```

Description:	Represents an individual channel. User doesn't use this class directly

```
class oscilloscope
```

Description:	Abstract. Base class for all parsers. The **parser** class in every model namespace (see Sec. 3.1.1) derives this class. It has all the get-methods to retrieve the parsed information.

```
class analyzer
```

Description:	Performs some analysis of a waveform

5.2 class oscilloscope

Description

—

public typedefs

```
typedef typename std::vector<owon::point>::const_iterator const_point_iterator;
```

Description:	Should be used to iterate over the data points in a waveform

public members

```
const_point_iterator cbegin() const noexcept
```

Description:	Should be used to get the first point of a waveform (AC)
Return:	Iterator to the beginning
Arguments:	None

```
const_point_iterator cend() const noexcept
```

Description:	Should be used to check if the end of a waveform is reached (AC)
Return:	Iterator to the end
Arguments:	None

```
bool set_active_channel( CHANNEL active_channel )
```

Description:	All the information that is of an individual channel (such as number of data points or voltage per devision) is returned for so called active channel (see Sec.3.1.1). Therefore, in general getting the information has two stages: setting the active channel and then using the get-methods. NOTE: setting the active channel may fail in case when user tries to set unavailable (not existing) channel as the active one. For example, setting the channel **CH3** as the active one on the **sds6062** model is meaningless because that model has only two channels. In such a case the active channel defaults to **CH1**
Return:	**true** if channel setting succeeded. **false** otherwise
Arguments:	**CHANNEL active_channel** - channel to be the active one

```
std::string get_serial() const
```

Description:	Should be used to get the serial number of the oscilloscope that produced a given binary file
Return:	Serial number string. It returns **"N/A"** in case when serial number is not available
Arguments:	None

```
std::string get_model() const
```

Description:	Should be used to get the model name of the oscilloscope of a given parser (in model namespace)
Return:	Model name string
Arguments:	None

```
bool get_status() const
```

Description:	Returns the current status of the last performed parsing
Return:	**true** if the last parsing succeeded. **false** otherwise
Arguments:	None

```
unsigned get_n_channels() const
```

Description:	Returns the number of available channel of a given oscilloscope
Return:	Number of available channels
Arguments:	None

```
CHANNEL get_active_channel() const
```

Description:	Should be used to monitor the current active channel
Return:	Current active channel
Arguments:	None

```
unsigned get_wave_size() const
```

Description:	Should be used to know how many points in a waveform (AC)
Return:	Number of data points
Arguments:	None

```
float get_voltage_per_div() const
```

Description:	Should be used to get voltage per division (AC)
Return:	Voltage per division (in mV)
Arguments:	None

```
float get_vertical_position() const
```

Description:	Should be used to get vertical position of a zero level (vertical shift). NOTE: it is not necessary the position of the true ground (AC)
Return:	Vertical position (in mV)
Arguments:	None

```
float get_time_per_div() const
```

Description:	Should be used to get time per division (AC)
Return:	Time per division (in μs)
Arguments:	None

```
float get_time_step() const
```

Description:	Should be used to get the time between two consecutive points (AC)
Return:	Time between two consecutive points (in μs)
Arguments:	None

```
float get_trigger_time() const
```

Description:	Returns the time of the trigger arrival counting from 0
Return:	Trigger time (in μs)
Arguments:	None

```
float get_length() const
```

Description:	Should be used to get the length of a waveform (AC)
Return:	Length of a signal (in μs)
Arguments:	None

```
float get_height() const
```

Description:	Should be used to get the height of a waveform. It simply the visible voltage range on the oscilloscope window (AC)
Return:	Height of a signal (in mV)
Arguments:	None

```
void print() const
```

Description:	Prints the primary information of the content of a binary file
Return:	None
Arguments:	None

5.3 class analyzer

Description

—

public enums

```
enum BASELINE { MODE, AVERAGE };
```

Description:	Should be used to choose method to calculate the baseline level (see below)
Values:	**MODE, AVERAGE**

public members

```
analyzer( const oscilloscope* pi )
```

Description:	Constructor
Arguments:	**const oscilloscope* pi** - pointer to an **oscilloscope** object

```
void analyze()
```

Description:	This function performs the analysis process. NOTE: before the invocation this function the **oscilloscope::parse** function must have been called and the analysis config set (see Sec. 3.1.2)
Return:	None
Arguments:	None

```
void set_baseline_time( float interval )
```

Description:	Sets the range to be used during the baseline calculation (see below). If **interval** < 0 or exeeds the length of a waveform the range $[0, 0.5t_{\text{trigger}}]$ will be used NOTE: The **parser::parse** must have been called before setting the baseline interval
Return:	None
Arguments:	**float interval** - right edge of the range used to calculate the baseline (left is zero) (in μs). Default is **-1**

```
void set_gate( float integral_start, float integral_stop )
```

Description: Should be used to specify the integration range. The integration processis simply the sum of the voltage displacements from the baseline level:

$$\text{integral} = \delta T \sum_{t_i \in [t_{\text{start}}, t_{\text{stop}}]} (v_0 - v_i), \tag{1}$$

where

δT - time step between two consecutive points

v_0 - the baseline level

v_i - voltage mark of the i-th point

t_i - time mark of the i-th point

$t_{\text{start}}, t_{\text{stop}}$ - the beginning and the end of integration, respectively

If **interval_start** < 0 or exceeds the length of a waveform it will be set to 0. If **interval_stop** < 0 or exceeds the length of a waveform it will be set to the length of a waveform

NOTE: The **parser::parse** must have been called before setting the gate

Return: None

Arguments: **float integral_start** - the time to itegrate from (in μs). Default is -1
float integral_stop - the time to integrate to (in μs). Default is **-1**

```
void set_baseline_method( BASELINE method )
```

Description: Should be used to choose a method to calculate the baseline level. Two methods of the baseline calculation are available in the library. The first one is the mode-method. To choose it you should use the **analyzer::BASELINE::MODE** as an argument. In this case the baseline is calculated simply as the mode of the voltages in the specified range:

$$\text{baseline} = \underset{t_i \in [0, t_{\text{baseline}}]}{\text{mode}} \{v_i\}, \tag{2}$$

where

v_i - voltage mark of the i-th point

t_i - time mark of the i-th point

t_{baseline} - right edge of the range used to calculate the baseline (**interval** in the **analyzer::set_baseline_interval** memberfunction)

The second method calculates the baseline as the average value of voltages:

$$\text{baseline} = \frac{1}{N} \sum_{t_i \in [0, t_{\text{baseline}}]} v_i, \tag{3}$$

where N is the number of points in the range from 0 to t_{baseline} and the rest means the same as in Eq. (2). To use it you should use the **analyzer::BASELINE::AVERAGE** as an argument

Return: None

Arguments: **BASELINE method** - Method identifier for the baseline calculation

```
float get_gate_time() const
```

Description:	Returns the time difference between edges of integration range (AC)
Return:	Length of integration range (in μs)
Arguments:	None

```
float get_integral_start() const
```

Description:	Returns left edge of integration range (AC)
Return:	left edge of integration range (in μs)
Arguments:	None

```
float get_integral_stop() const
```

Description:	Returns right edge of integration range (AC)
Return:	Right edge of integration range (in μs)
Arguments:	None

```
float get_baseline() const
```

Description:	Returns the baseline of a signal (AC)
Return:	Baseline level (in mV)
Arguments:	None

```
float get_amplitude() const
```

Description:	Returns the maximum displacement from the baseline (AC)
Return:	Maximum amplitude (in mV)
Arguments:	None

```
float get_amplitude_time() const
```

Description:	Returns the time of the maximum displacement from the baseline (AC)
Return:	Time of the maximum point (in μs)
Arguments:	None

```
float get_pkpk() const
```

Description:	Returns the voltage difference between the maximum and minimum of a waveform (AC)
Return:	Voltage difference between the extremums (in mV)
Arguments:	None

```
float get_integral() const
```

Description:	Returns the integral over time in the range specified with the **analyzer::set_gate** member-function (AC)
Return:	Integral (in mV$\times$ μs)
Arguments:	None

```
void print() const
```

Description:	Prints the result of analysis (AC)
Return:	None
Arguments:	None