C++ interface for working with binary data files produced by CAEN WaveDump software

LRDPRDX

April 14, 2020

Contents

1 Introduction

CAEN provides several softwares that a digitizer can run for data acquisition: DPP-CI (now deprecated), DPP-PSD, CAENScope, WaveDump (https://www.caen.it/products/caen-wavedump/) (hereinafter the WD), etc. The software described in this document is designed to work with the data acquired with WD software. WD allows user to save the acquired data in two formats: either binary or ASCII. Since saving data in binary format provides higher record rate and more information about the data, and takes less space on disk, one should choose it in order to get better performance in case when analysis of large number of events is needed (for example, to get integral spectrum or average waveform).

However, parsing of binary files requires a bit more complicated algorithms comparing with those needed in case of using ASCII files. So this software is intended to provide a simple C++ interface to work with WD's binary data files. Moreover, this software can be easily "extended" by user in the following sense. Each event consisting of data points (or a waveform) and some additional info is represented by and accessible through the **`Event`** class (described below). This allows user to write his/her own functions to perform any kind of analysis on an event using only **`Event`** object which stores given event.

WARNING: This software is NOT considered to work with ASCII files. So use it only with binaries produced by the WD software.

Operating CAEN digitizer with WD software for data acqusition requires a configuration file containing device settings. This file is just a text file which (mostly) consists of lines like this

```
<parameter> <value>
```

that define digitizer's behavior.

Of course, it is possible to create such file directly using a text editor just once and any time another configuration is needed just copy and change it. However, it would be a bit more convenient to have a tool that constructs configuration file automatically.

2 Installation

2.1 Stand-alone

Assuming you have unpacked the package into the directory **<package_dir>** the installation is the following:

```
> cd <package_dir>/make
> make compile
> make link
> make clean
```

The first **`make`** line compiles dynamic library. The second one places it into standard location (**`/usr/lib`** in this case); here you probably need root privileges. The third one deletes object files that are no longer needed.

That's all! At this point you should be able to use the library. However, the following step is recommended. It is useful to add package's include to standard C++ include path of your system. On Ubuntu adding the following lines in **`.profile`** (or **`.bash_profile`**) does the work:

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

with **path/to/<package_dir>** replaced with the correct path to **<package_dir>** directory.

Now logout and login back.

In other Linux distribution you probably would have to use similar procedure.

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 install Boost Filesystem Library on your system.

After all the prerequisites are installed the final step is to compile everything into '.so' library:

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

3 Usage

3.1 Stand-alone

3.1.1 Entire structure

In order to prevent name conflicts and keep global scope clean everything is placed into namespace **`caen`**.

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

NOTE: In the below code the **`using`** declaration is *not* assumed so each entity of the library appears with the **`caen::`** prefix.

So far there are three main classes on the scene. Those are: **`Event`**, **`Parser`**, **`Analyzer`**. The diagram on Fig.1 illustrates how they interact.

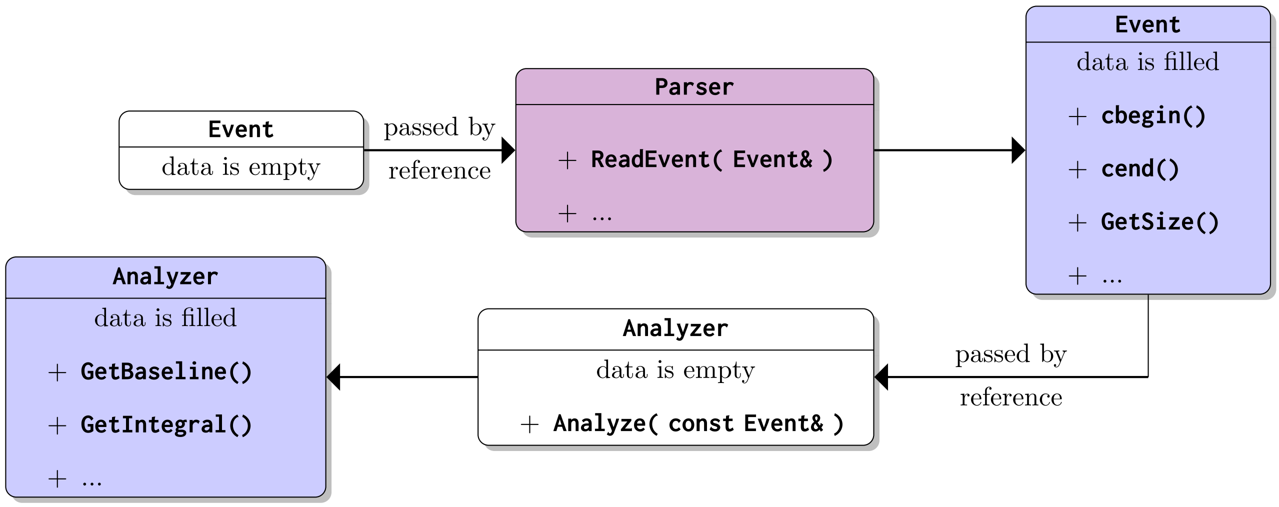

Figure 1: Interaction between classes during processing an event.

In this section one will find a brief description of the abovementioned classes and how to work with them. For more details see sec 4.

`Event` class. The **`Event`** class represents a single event. It serves to store primary information of an event: its size, trigger time tag, ..., and data points. Also it has some additional member functions that might be useful (see sec.4).

One usually needs only one instance of this class (see below).

`Parser` class. The **`Parser`** class is designed for event construction. More precisely it takes an empty event and *fill* it according to the data in a binary file produced by WD.

`Analyzer` class. The **`Analyzer`** class is designed for (no surprise) performing simple analysis on the event data. Unlike the **`Parser`** class this class deals with events which are already constructed and filled by the **`Parser`** object. **`Analyzer`** *doesn't change an event.* In other words, events are read-only for **`Analyzer`**.

`Board` class. There is one more class which must be mentioned. It is the **`Board`** class. It represents a digitizer's model and serves as a translator between abstract digital and physical scales. For example, the same waveform may be of different length (having the same number of points) depending on a digitizer which produced it because the length (or sample time) depends on the sample rate.

3.1.2 Read an event

In order to be able to read an event from WD's binary file user should follow these steps:

1. Choose correct digitizer's model

2. Create **Parser** object
3. Set path to file to parse using **Parser::SetPathToFile** member function
4. Create **Event** object
5. Call **Parser::ReadEvent** or **Parser::ReadEventAt** member function with previously created **Event** object as an argument
6. Use **Event**'s member functions to retrieve information about the read event

Algorithm 1: Reading an event from a binary

Let us describe each step in detail below.

Choosing digitizer's model. In order to select digitizer's model user has to pass object of **Board** class as an argument to the constructor of **Parser** class (see below). **Board** class can be instantiated using either of two constructors. The first one allows user to choose a model from a list of known digitizers (constructed from this list). The second one is used when the desired model is not present in the list. For more details see sec.4.

```
#include "CaenParser.h"//CaenBoard.h header included here

caen::Board boardOne( caen::Board::N6720 );//using built-in enum
caen::Board boardTwo( 12, 250000000, 2.0 );//providing board's characteristics by hand
```

NOTE: If you didn't set the include path as it was recommended in Sec.2 you would probably have to provide full path in the **include** directive. However, it is not the case if you specify the include path (with **-I** option) when compiling.

Creating Parser object. **Parser** class has the only constructor that takes **Board** object as an argument:

```
#include "CaenParser.h"

caen::Parser parser( caen::Board::N6720 );//note copy-initialization for Board class
```

Setting path to binary. Once you have chosen the board's model it's time to specify file to work with. Its type must be **std::string**. Most probably the name of such file would be **waveN.dat** (unless has been changed by user) where **N** is channel number (starts from 0):

```
parser.SetPathToFile( "path/to/waveN.dat" );//note extension
```

The path may be either absolute or relative.

Creating Event object. The **Event** class has the only constructor that takes no arguments:

```
#include "CaenEvent.h"

caen::Event event;
```

As it was mentioned in Sec.3.1.1 one usually needs only one instance of each **Event** and **Parser** class. But there are cases when several instances are useful (or even necessary). For example, in case when comparing several channels is required. As far as WD produces separate data file per channel one should have its own **Parser** instance for each channel. And as it will be clear from the below text in this case one should also use its own **Event** instance for each channel.

Reading an event

Next step is to read an event from a binary file produced by WD. To do this user should call **ReadEvent** member function with **Event** object as an argument (passed by reference):

```
parser.ReadEvent( event );
//OR
parser.ReadEventAt( 0, event );
```

ReadEvent member function returns **true** if the event has been successfully read and **false** otherwise. Here are some steps that **ReadEvent** member function performs:

1. Reset current event
2. - (**ReadEvent**) Try to read and fill the event from the position followed after the end of previous event — current position indicator (or from the beginning of the file in case when no event hasn't been read yet)
   - (**ReadEventAt**) Try to read and fill the event from the specified position in binary file
3. (**ReadEvent** only) If 2. succeeded then update position indicator

The above algorithm is not exactly what **ReadEvent** does but here are several things to note. First of all, it *resets the current event*. This means that at the moment of invocation of **ReadEvent** member function a variable that was passed to it *is no longer stores whatever it did before the invocation* even if reading failed. The second thing to note is that **ReadEvent** member function reads one event at a time. You could only read events one-by-one, one after another. Every time you want to read new event you must invoke **ReadEvent** member function.

Eventually the complete program may look like this:

```
#include "CaenParser.h"
#include "CaenEvent.h"


int main()
{
    caen::Parser parser( caen::Board::N6720 );
        parser.SetPathToFile( "path/to/waveN.dat" );
    caen::Event event;

    while( parser.ReadEvent( event ) )//read event by event
    {
        //Information about the current event is accessible
        //through Event's member functions
        std::cout << event.GetSize() << "\n";
    }

    return 0;
}
```

Code 1: Print size of each event stored in **waveN.dat** file

3.1.3 Analyse an event

Often having just primary data is not enough. One may need some property of a signal that is not in the data file. It might be integral over a given time interval or time point at the maximum displacement from the baseline. Those purposes are the **Analyzer** class was written for. It is defined in **CaenAnalyzer.h** file. This class is instantiated by calling the only constructor that takes no arguments:

```
#include "CaenAnalyzer.h"

caen::Analyzer analyzer;
```

In order to perform analysis of an event user calls correspondig mamber function of this class with the desired **Event** object as an argument (passed by reference):

```
analyzer.Analyze( event );
```

Analysis config. However the above line is not enough for complete analysis. User should provide the *analysis config*. Some of the calculations use the user-defined parameters. For example, if you want to get integral over time of a signal you should specify time range of integration. Moreover, in order to calculate integral it is required to know the baseline (or zero level). So far **Analyzer** has two simple methods to calculate zero level. For both of them user should also specify time range (see Sec.4). So the analysis config consists of the following settings (the order is not of importance):

- Method to calculate baseline. Now there are two methods (see sec.4). Member function to use: **SetBaselineMethod(Analyzer::BASELINE method)**
- Time interval for baseline calculation. It starts at the beginning of a signal and ends at the time defined by user. Member function to use: **SetBaselineInterval(double T)**
- Time interval for integral calculation. It starts and ends at time points defined by user. Member function to use: **SetGate(double start, double end)**

NOTE: The analysis config should be done *before* the invocation of **Analyze** member function.

So the complete program with analysis part usually contains the following lines:

```cpp
#include "CaenParser.h"
#include "CaenAnalyzer.h"
#include "CaenEvent.h"


int main()
{
    caen::Parser parser( caen::Board::N6720 );
        parser.SetPathToFile( "path/to/waveN.dat" );
    caen::Event event;
    caen::Analyzer analyzer;

    //Analysis config
    analyzer.SetBaselineInterval( 0.03 );//set range for baseline calcultation: the first 30 ns of the signal
    analyzer.SetGate( 0.04, 0.1 );//set integral range between 40 and 100 ns

    while( parser.ReadEvent( event ) )
    {
        //Perform analysis
        analyzer.Analyze( event );
        //Now the results are available through the getters
        std::cout << analyzer.GetIntegral() << "\n";
    }

    return 0;
}
```

Code 2: Print integral of each event in **waveN.dat** calculated within the range from 40 to 100 nanosecs

3.1.4 Compilation

Provided you put your code in the file named **example.cpp** the following line should compile it into **example** executable:

```
g++ example.cpp -std=c++11 -I<path/to/package_dir>/inc -lcaenparse -o example
```

It is more convenient to put this in a **Makefile** (one may find an example of such **Makefile** in **example/stand_alone** 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 **CaenTreeParser** class using the same **Board** as was used to produce the binary
2. Set the analysis config using member-functions of the **CaenTreeCreator** class
3. Call the **CreateTree** member-function
4. Compile and run

Algorithm 2: Creation of a **TTree**

Below is the explanation of the above algorithm in detail.

The CaenTreeCreator class. This class is responsible for creation of a **TTree** from a data binary file. To create an instance of this class you, firstly, include its header:

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

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

```
TreeCreator( caen::Board board,
             const std::string& pathToDataDir,
             const std::string& pathToTreeFile,
             const std::string& treeFileName )
```

where

board — board model which was used to record data

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
     |
     |__wave01.dat
     |__wave02.dat
     |__...
```

In this case, provided the second argument in the **CaenTreeCreator** 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
    |  |
    |  |__wave01.dat
    |  |__wave02.dat
    |  |__...
    |
    |_data2
      |
      |__wave01.dat
      |__wave01.dat
      |__...
```

then, provided the second argument in the **CaenTreeCreator** 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
    |  |
    |  |__wave01.dat
    |  |__wave02.dat
    |  |__...
    |
    |_data2 <--hierarchy matters
      |
      |__wave01.dat
      |__wave02.dat
      |__...
      |__subsubdata <-- hierarchy DOESN'T matter: expanded
         |__wave01.dat
         |__wave02.dat
         |__...
```

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

After the instantiating the **CaenTreeCreator** you should set the analysis config (see Sec. 3.1.3) which will be used when constructing the **TTree**:

```
tc.SetIntervals( 0.1, 0.05, 0.2 );//baseline - from the beginning to 100 ns; integral - from 50 to
    200 ns
```

NOTE: Setting the analysis config for the **TreeCreator** class differs slightly from that for the **analyzer** class (see Sec.4.6)

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

```
#include "ROOT/CaenTreeCreator.h"
#include "CaenParser.h"
```

```
void CreateTree()
{
    CaenTreeCreator tc( caen::Board::N6720, //N6720 ADC was used for recording
                      , "./Data", "./", "myFirstTree" );
    tc.SetIntervals( 0.03, 0.03, 0.1 ); //baseline - from 0 to 30 ns; integral - from 30 to 100 ns

    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/<package_dir>/src/ROOT/CaenTreeCreator/CaenTreeCreator_cpp.so" );
}
```

where **/path/to/<package_dir>** is the absolute path to the **<package_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 **<package_dir>/example/ROOT/create_tree** directory.

4 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 **caen::** 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.

4.1 Class Board

Defined in **CaenBoard.h**

Description

Represents a digitizer's model. Needed to transform digital quantities into real time and voltage values specific to concrete digitizer.

public enums

Board::MODEL

Description:	Represents a list of known CAEN digitizers
Values:	**DT5720, DT5724, DT5725, DT5730, DT5740, DT5740D, DT5742, DT5743, DT5751, DT5761, N6720, N6724, N6725, N6730, N6740, N6740D, N6742, N6743, N6751, N6761, V1720, V1724, V1725, V1730, V1740, V1740D, V1742, V1743, V1751, V1761, VX1720, VX1724, VX1725, VX1730, VX1740, VX1740D, VX1742, VX1743, VX1751, VX1761**

public members

Board::Board(unsigned resolution, uint64_t sampleRate, double FSR)

Description:	Constructor	
Arguments:	**unsigned resolution** -	resolution of a digitizer in bits
	uint64_t sampleRate -	sampling rate of a digitizer in S/s (samples per second)
	double FSR -	full scale range in V_{pp}

Board::Board(Board::MODEL model)

Description:	Constructor
Arguments:	**Board::MODEL model** - model of a digitizer from built-in list (see **public enums**)

unsigned Board::GetResolution() const

Description:	Should be used to get resolution of a digitizer
Return:	Resolution in bits (**unsigned**)
Arguments:	None

uint64_t Board::GetSampleRate() const

Description:	Should be used to get sample rate of a digitizer
Return:	Sample rate in S/s (samples per second) (**uint64_t**)
Arguments:	None

double Board::GetSampleTime() const

Description:	Should be used to get time difference between two neighboring points in a waveform
Return:	Sampling time (in μs) (**double**)
Arguments:	None

double Board::GetFSR() const

Description:	Should be used to get full scale range of a digitizer
Return:	Peak to peak voltage of a full scale range (in volts) (**double**)
Arguments:	None

double Board::GetLSB() const

Description:	Should be used to get least significant bit of a digitizer
Return:	Resolution in volts (**double**)
Arguments:	None

4.2 Struct `Point`

Defined in `CaenEvent.h`.

Description

Represents a single data point in a waveform. Has two public data members: `time` and `voltage`

`public` members

`Point::Point( double time, double voltage )`

Description:	Constructor. Not intended to be used by user
Arguments:	`double time` - real time (in μs) of the point `double voltage` - real voltage (in mV) of the point

`double Point::time`

Description:	Real time (in μs) of the point in a waveform

`double Point::voltage`

Description:	Real voltage (in mV) of the point in a waveform

4.3 Class **Event**

Defined in **CaenEvent.h**

Description

Represents a single event — consists of primary information of an event: header (see WDDOC) and data points. Also contains position indicator at which an event starts and ends in binary file

public typedefs

typedef typename std::vector<Point>::const_iterator const_point_iterator

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

typedef typename std::fstream::pos_type pos_t

Description: Alias for **std::fstream::pos_type**

public enums

Event::POLARITY

Description: Represents signal polarity

Values: **NEGATIVE, POSITIVE**

public members

Event::Event()

Description: Constructor

Arguments: None

Event::const_point_iterator Event::cbegin() const

Description: Should be used to get the first point of a waveform. **cbegin()** of **std::vector<Point>**

Return: Iterator to the beginning (**std::vector<Point>::const_iterator**)

Arguments: None

Event::const_point_iterator Event::cend() const

Description: Should be used to check if the end of a waveform has been reached. **cend()** of **std::vector<Point>**

Return: Iterator to the end (**std::vector<Point>::const_iterator**)

Arguments: None

void Event::Clear()

Description: Resets the event. Sets all its data members to their initial values

Return: None

Arguments: None

void Event::SetPolarity(Event::POLARITY pol)

Description: Sets signal polarity. The polarity matters for **Analyzer** when calculating integral and extremum points

Return: None

Arguments: **Event::POLARITY pol -** polarity of a signal

`pos_t Event::GetStartPosition() const`

Description:	Getting position of the starting byte of the event
Return:	Position in the file where the event starts from if the event has been read without errors and **`0`** otherwise (**`pos_t`**)
Arguments:	None

`pos_t Event::GetEndPosition() const`

Description:	Getting position in the file after the last byte of the event
Return:	Position in the file **`next to`** the end of the event if the event has been read without errors. **`0`** otherwise (**`pos_t`**)
Arguments:	None

`size_t Event::GetSize() const`

Description:	Should be used to determine number of points in a waveform
Return:	Size of **`std::vector<Point> points`** data member (**`size_t`**)
Arguments:	None

`double Event::GetLength() const`

Description:	Should be used to get length of recording window in microseconds
Return:	Time duration (in μs) of a signal (**`double`**)
Arguments:	None

`Event::POLARITY Event::GetPolarity() const`

Description:	Should be used to get polarity of a signal
Return:	Polarity of a signal (**`Event::POLARITY`**)
Arguments:	None

`double Event::GetTimeStep() const`

Description:	Should be used to get time difference between two neighboring points in a waveform
Return:	Sampling time (in μs) (**`double`**)
Arguments:	None

`void Event::Print() const`

Description:	Prints the event in a nice human readable form
Return:	None
Arguments:	None

`uint32_t Event::GetBoardID() const`

Description:	
Return:	board ID (**`uint32_t`**)
Arguments:	None

`uint32_t Event::GetPattern() const`

Description:	
Return:	(**`uint32_t`**)
Arguments:	None

`uint32_t Event::GetChannel() const`

Description:	Should be used to determine to what input channel the event belongs
Return:	Channel number (**`uint32_t`**)
Arguments:	None

`uint32_t Event::GetEventCounter() const`

Description:	Should be used to get counter of the event. NOTE: the first event in a file is not necessarily has count number equal to 0. Also it resets on overflow so several events could have the same value
Return:	Count number of the event (**`uint32_t`**)
Arguments:	None

`uint32_t Event::GetTriggerTimeTag() const`

Description:	
Return:	(**`uint32_t`**)
Arguments:	None

4.4 Class `Parser`

Defined in **`CaenParser.h`**

Description

This class is designed for processing binary files produced by WD. Main function of **`Parser`** class is to translate raw data in WD's binaries into understandable information — events — represented by **`Event`** class.

`public` members

`Parser::Parser( Board board )`

Description:	Constructor
Arguments:	**`Board board`** - model of a digitizer

`void Parser::SetPathToFile( const std::string& pathToFile )`

Description:	Used to specify path to binary file to be working with. NOTE: this member function calls **`Reset`** member so setting new path changes object's state
Return:	None
Arguments:	**`const std::string& pathToFile`** - path to WD's binary file. Could be either absolute or relative. NOTE: it must be a full name including the extension if present.

`bool Parser::ReadEventAt( pos_t pos, Event& event) const`

Description:	Tries to read the event started from a given position in the file. Note **`const`** specifier. In the context this means that this member doesn't change parser's state (in comparison with **`ReadEvent`** member). Usually used when reading individual event.
Return:	**`true`** if the event has been read and filled successfully. **`false`** otherwise. (**`bool`**)
Arguments:	**`pos_t pos`** - position (absolute) in the file where reading an event should start **`Event& event`** - event to be filled

`bool Parser::ReadEvent( Event& event)`

Description:	Tries to read the event started from the position *next to* the last byte of previously read event. If succeeded this member function *changes* parser's state (in comparison with **`ReadEventAt`** member): increments number of read events, updates position indicator for the next reading, etc. Usually used when reading group of events.
Return:	**`true`** if an event has been read and filled successfully. **`false`** otherwise. (**`bool`**)
Arguments:	**`Event& event`** - event to be filled

`size_t Parser::GetNEvents() const`

Description:	Should be used to determine how many events are successfully read so far.
Return:	Current value of **`nEvents`** data member (**`size_t`**)
Arguments:	None

`void Parser::Reset()`

Description:	Sets all data members to their initial values
Return:	None
Arguments:	None

`void Parser::PrintCurrentPosition() const`

Description:	Prints byte-position in the file is being parsed next to the last successfully read event in hexadecimal. This member function is used, for example, in **`ReadEventAt`** member function in order to notify user where an error occured while trying to parse the file
Return:	None
Arguments:	None

`void Parser::Print() const`

Description:	Prints object in a nice human-readable form
Return:	None
Arguments:	None

4.5 Class **Analyzer**

Defined in **CaenAnalyzer.h**

Description

This class is designed to analyze events. Under analysis it is assumed obtaining some *secondary* information about an event: baseline position, integral over a given time interval, etc. This class works with **Event** objects in the *read-only* mode i.e. it doesn't change their states.

public enums

Analyzer::BASELINE

Description: Enumerates the names of available methods for baseline calculation.

AVERAGE - Calculate baseline as the average value over a given time interval:

$$V_0 = \frac{1}{N_T} \sum_{t_i \in [0,T]} V(t_i),$$

where

$V(t_i)$ — voltage at time t_i

T — time interval in μs

N_T — number of points within the specified time interval T

MODE - Calculate baseline as the most frequent value (mode) in a given time interval

Values: **AVERAGE, MODE**

public members

Analyzer::Analyzer()

Description: Constructor

Arguments: None

void Analyzer::Analyze(const Event& event)

Description: Performs simple analysis on a waveform:

- baseline calculation
- integral calculation
- max and min points searching
- peak-to-peak calculation

After the invocation results of analysis are available through corresponding getterss (see below).

Note that this member function doesn't change the event's state.

Return: None

Arguments: **const Event& event -** event to be analysed

void Analyzer::SetGate(double start, double stop)

Description: Should be used to specify limits of integration. Integration is calculated as the sum of point displacements from the baseline:

$$\text{integral} = \pm \sum_{t_i \in [\text{start},\text{stop}]} (V(t_i) - V_0)\Delta t,$$

where
$V(t_i)$ — voltage at time t_i
V_0 — baseline level
Δt — sampling time
The sign — plus or minus — depends on a signal polarity and is chosen so the integral of unipolar signal is positive.

NOTE: There is no check or validation of provided values. For example, if you provided incorrect limits such as **start** > **stop** it wouldn't be a warning or anything and the integral would be equal to **0**.

Return: None

Arguments: **double start -** left limit of integration (in μs)
dobule stop - right limit of integration (in μs)

---

void Analyzer::SetBaselineInterval(double T)

Description: Should be used to specify time interaval which will be used to calculate the baseline level of a signal (see **Analyzer::BASELINE**). Default is **AVERAGE**

Return: None

Arguments: **double T -** length of the interval started from the beginning (in μs) used in the baseline calculation.

---

void Analyzer::SetBaselineMethod(Analyzer::BASELINE method)

Description: Should be used to choose how to calculate baseline of a signal (see **Analyzer::BASELINE**)

Return: None

Arguments: **Analyze::BASELINE method -** enumerated method's name

---

double Analyzer::GetGateInterval() const

Description: Should be used to get time difference between limits of integration

Return: Length of integration interval (in μs) (**double**)

Arguments: None

---

double Analyzer::GetIntegralStart() const

Description: Should be used to get left limit of integration

Return: Start of integration (in μs) (**double**)

Arguments: None

---

double Analyzer::GetIntegralStop() const

Description: Should be used to get right limit of integration

Return: End of integration (in μs) (**double**)

Arguments: None

`double Analyzer::GetBaseline() const`

Description:	Should be used to get baseline (zero level) of a signal
Return:	Baseline level (in mV) (**`double`**)
Arguments:	None

`Point Analyzer::GetMaxPoint() const`

Description:	Should be used to get the point with the maximum positive displacement from baseline
Return:	Maximum positive point (**`Point`**)
Arguments:	None

`Point Analyzer::GetMinPoint() const`

Description:	Should be used to get the point with the maximum negative displacement from baseline
Return:	Maximum negative point (**`Point`**)
Arguments:	None

`double Analyzer::GetPkPk() const`

Description:	Should be used to get voltage difference between positive and negative extremum in a waveform.
Return:	Peak-to-peak value (in mV) (**`double`**)
Arguments:	None

`double Analyzer::GetIntegral() const`

Description:	Should be used to get result of integration
Return:	Integral value (in mV$\cdot\mu$s) (**`double`**)
Arguments:	None

4.6 class `CaenTreeCreator` (ROOT)

Description

This class is used to create a **`TTree`** from binaries

`public enums`

`CaenTreeCreator::SAMPLE`

Description:	Should be used to choose what directories to include when constructing a tree. See the **`CaenTreeCreator::CreateTree`** function
Values:	**`ALL, INCLUDE, EXCLUDE`**

`public` members

`CaenTreeCreator::CaenTreeCreator( caen::Board board, const std::string& pathToDataDir, const std::string& pathToTreeFile, const std::string& treeFileName )`

Description:	Constructor	
Arguments:	**`caen::Board board`**	- ADC model Must be of the same model as the one used to record data
	`const std::string& pathToDataDir`	- A valid path to the data directory (see Sec.3.2)
	`const std::string& pathToTreeFile`	- A valid path where the **`.root`** file will be placed
	`const std::string& treeFileName`	- Name of the **`.root`** file with the resulting **`Tree`**s (without an extension)

`void CaenTreeCreator::SetPathToDataDir( const std::string& pathToDataDir )`

Description:	Should be used to set path to the data directory	
Return:	None	
Arguments:	**`const std::string& pathToDataDir`**	- A valid path to the directory which will be used to iterate through to create **`TTree`**s

`void CaenTreeCreator::SetPathToTreeFile( const std::string& pathToTreeFile )`

Description:	Should be used to set path to the file with the resulting **`TTree`**s	
Return:	None	
Arguments:	**`const std::string& pathToTreeFile`**	- A valid path where the **`.root`** file with the resulting **`TTree`**s will be placed

`void CaenTreeCreator::SetTreeFileName( const std::string& treeFileName )`

Description:	Should be used to set the name of the resulting **`.root`** file to which the resulting **`Tree`**s will be written	
Return:	None	
Arguments:	**`const std::string& treeFileName`**	- Name of the file (without an extension)

void CaenTreeCreator::SetIntervals(Double_t baselineTime, Double_t integralStart, Double_t integralStop)

Description:	Should be used to set the analysis config	
Return:	None	
Arguments:	**Double_t baselineTime**	- the right edge of the interval for the baseline calculation
	Double_t integralStart	- time to start integration
	Double_t integralStop	- time to stop integration

std::string CaenTreeCreator::GetPathToDataDir() const

Description:	Should be used to get current path to the data directory
Return:	Path to the data directory
Arguments:	None

std::string CaenTreeCreator::GetPathToTreeFile() const

Description:	Should be used to get current path to the file with the resulting **TTree**s
Return:	Path to the file
Arguments:	None

std::string CaenTreeCreator::GetTreeFileName() const

Description:	Should be used to get the current name of the file which the resulting **TTree**s to write to
Return:	Name of the file (without an extension)
Arguments:	None

void CaenTreeCreator::CreateTree(SAMPLE mode = ALL, const std::string& target = "")

Description:	Create a **TTree** from binaries	
Return:	None	
Arguments:	**mode**	- **CaenTreeCreator::SAMPLE::ALL**, **CaenTreeCreator::SAMPLE::INCLUDE** or **CaenTreeCreator::SAMPLE::EXCLUDE**
	target	- target directory name which is used to create a **TTree**: if the first argument **mode** is **CaenTreeCreator::SAMPLE::ALL** then this argument is ignored and all the subdirectories will be used. If **mode** is **CaenTreeCreator::SAMPLE::INCLUDE** then only the directory named **target** will be used. If **mode** is **CaenTreeCreator::SAMPLE::EXCLUDE** then all the directories will be used except the one named **target**