FERMI NATIONAL LABORATORY

Final Report

Development and testing of the light detection readout system for the ICARUS experiment

Supervisor: Wesley Ketchum

Intern: Federico Roccati

"La vita si svolge felicemente,
in molteplici attività"

Abstract

This document reports the work done during my internship at Fermilab in the summer of 2017 under the supervision of Wesley Ketchum.

The task was divided into two parts. First the development and design of the DAQ code for the light readout system of the ICARUS detector were addressed. Second, the performance of the DAQ system was investigated.

Contents

Chapter 1

The DAQ system for light in the ICARUS experiment

The ICARUS detector is the far detector of the Short Baseline Neutrino (SBN) program. The main scientific goals of the program are the following:

- search for ν_e appearance and ν_μ disappearance in the Booster Neutrino Beam (BNB),
- followup on the MiniBooNE low energy excess,
- explore the phase space of short-baseline neutrino oscillations,
- precisely measure neutrino-argon cross section,
- further develop the Liquid Argon Time Projection Chamber (LArTPC) technology for the long-baseline DUNE experiment.

These goals are achieved placing three detectors on the line of the BNB (see Fig. 1.1): SBND, the Short Baseline Neutrino Detector (under construction), MicroBooNE (operating) and ICARUS (recently arrived at Fermilab from CERN).

These detectors share the same technology, but differ in size.

SBND, is located 110 meters from the BNB target, and has 112 tons of Liquid Argon (LAr) within the active volume of its time projection chamber (TPC) and light detection systems (LDS).

MicroBooNE is located 470 meters from the BNB target, and consists of a 8250-wire TPC and 32 photomultiplier tubes (PMTs) which instrument 80 tons of LAr in the active volume. The cryostat was filled in 2015 and the detector is currently operating.

The ICARUS T600 detector, divided into two cryostats holding LAr-TPC modules and photodetectors, will serve as the Short-Baseline Program Far Detector. It is the farthest from the BNB target, at a distance of 600

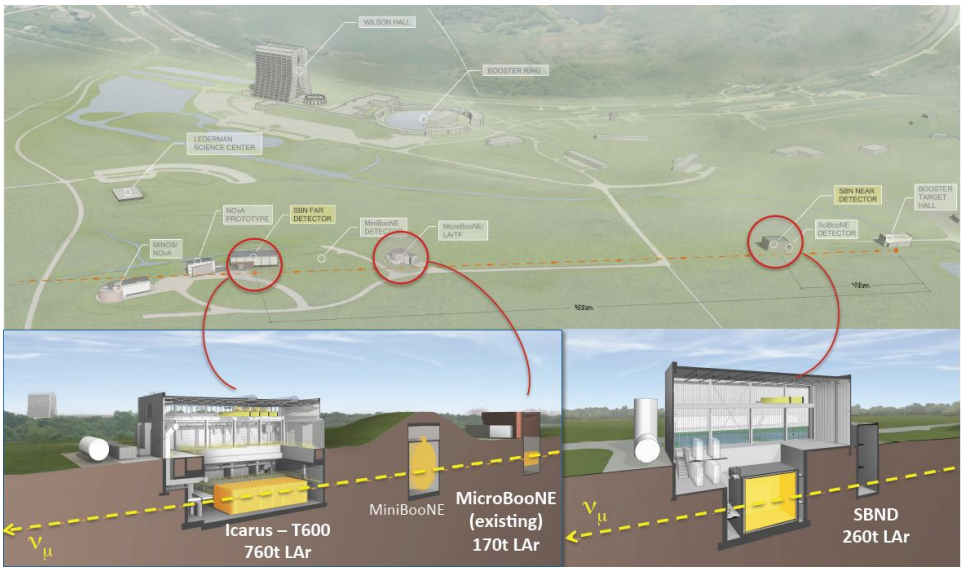

Figure 1.1: Pictorial view of the SBN setup.

meters, and it is the largest of the detectors with 500 tons of liquid argon in the active volumes.

All three detectors are LArTPC. A TPC is a kind of detector that allows to reconstruct a 3D image of the interactions happening in the active volume. It can be thought as a square box where a certain electric field is applied. The electric field goes from the anode plane to the cathode plane. At the anode plane three wireplanes with different orientation are placed. The measurement principle is the following: a high energy particle hits an atom/molecule in the chamber. If the incoming particle is energetic enough it will ionize it's target producing an electron and a positive ion. The electrons get collected at the wireplanes near the anode plane where they induce current and induce a signal. Two wireplanes allow for a 2D reconstruction of the event, while the time stamp of the event, combined with the drift velocity of the electrons, gives the the information to complete the 3D image.

1.1 Light production in the detector volume

Besides being ionized, some argon atoms can be just excited to an excited state and deexcite producing scintillation light. The main deexcitation mode are two: a fast one ($\sim ns$), and a slow one ($\sim \mu s$). One of the main reason to use LAr is its transparency to scintillation light which can be measured by photodetectors and used as a trigger (the fast component) to provide the precise event time stamp.

The detection of light from the PMTs becomes then an important issue

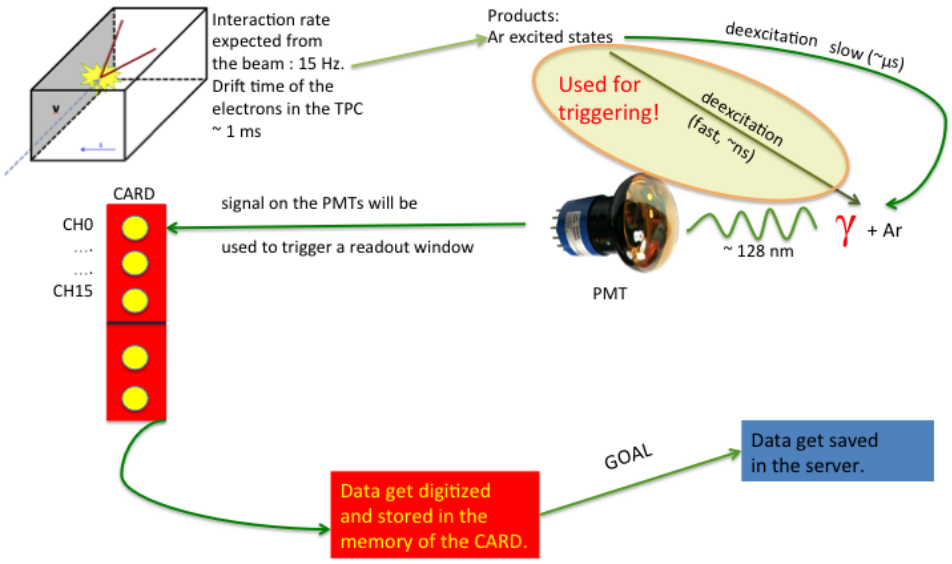

Figure 1.2: Pictorial view of the light production and readout

in event reconstruction.

In the following we will focus on the data acquisition system of light for the ICARUS detector.

The interaction rate expected from the BNB is of the order of 15 Hz. The scintillation light produced lies in the UV regime (∼128 nm), see Fig. 1.2. This light is detected and amplified by the PMTs which will be connected to the channels of the readout card. The analog signal going into the card get digitized into the digitizer and saved into the card memory. The goal of the DAQ system is to read the data fast enough from the card and store them in an external server to avoid memory overflow.

1.2 The test stand - operation mode

The test stand of the light readout is placed at the D0 building at Fermilab. It consists of two crates: one hosting the server and the other hosting the cards, see Fig. 1.3

All the tests were made using a CAEN V1730 16-channel waveform digitizer. The main mode of operation consisted in using a pulse generator to generate:

- an acquisition window (to mimic the BNB rate) at 15 Hz with a width

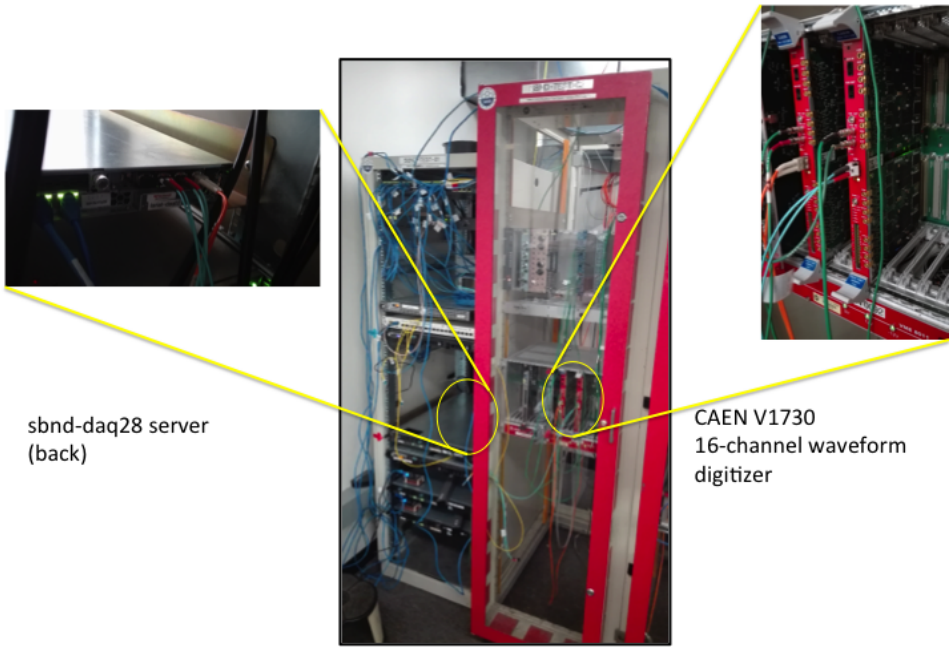

Figure 1.3: Picture of the test stand.

of 2 ms (drift time of the electrons from the interaction point + some buffer to round up)

- a readout window to mimic the PMT signal at 5 kHz

The first signal was plugged into the S-IN channel of the card, while the second in the TRG IN one.

1.3 DAQ code

artdaq is the product/framework used at Fermilab in the context of data acquisition.

1.3.1 Object structure

Among the many features of artdaq, the most important for the tests is that it allows the user to create **artdaq-fragments** which are C++ objects made of three parts:

- an **artdaq header**,

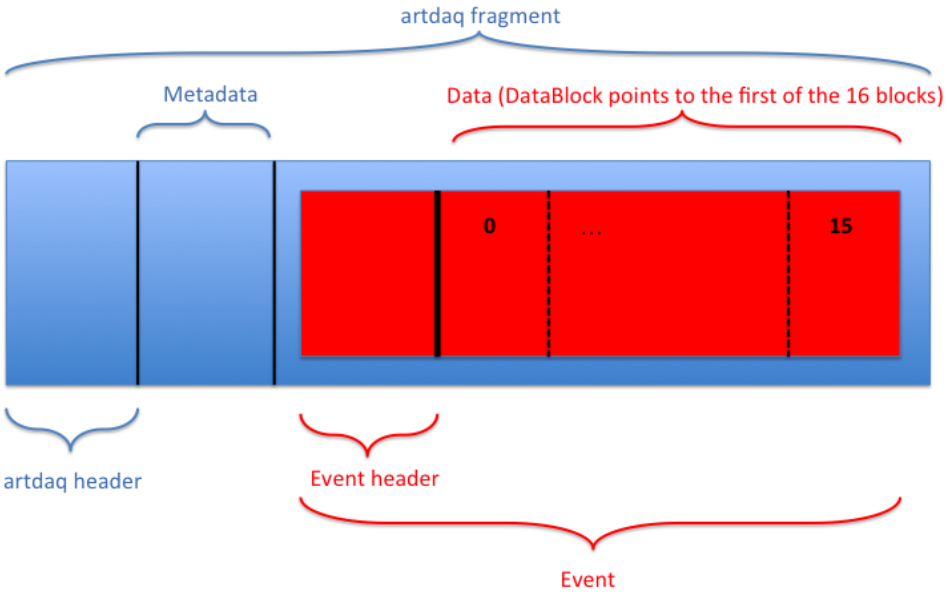

Figure 1.4: Structure of an event in artdaq.

- a Metadata object, which incorporate in the fragment any other useful information regarding the event (such as the number of channels in use, the number of samples, the time stamp of the event, etc..)
- the data part. Here is were the data of an event are stored. Each event is further divided into an **event header**, where information such as the event size, event counter, etc.. are stored, and a data block, which is a pointer to unsigned int that points to the address in memory where data reside.

1.3.2 test_driver code

Once the the pulse generator is setup as described in the previous section, the data acquisition can start. The tool used to acquire data is called test_driver. Artdaq allows also the user to configure the acquisition through configuration files (.fcl files). So when we launch the acquisition we can configure the readout typing

```
test_driver -c "config file"
```

The main part of the test_driver code is the getNext function. Here is where new data are read to create a fragment. Inside this function the

getNext_ function defined in CAENV1730Readout_generator.cc is called. This is where the real acquisition starts. One of the main goals of the internship was to write this code in such a way that all the PMT events (i.e. the readout windows, i.e. the pulse into the TRG IN channel) could be packed into one single artdaq fragment as shown in Fig. 1.6.

In the next section the documentation of the readout generator (pseudo)code, which packs PMT events into TPC events, is provided.

1.3.3 Readout generator

The pseudocode of the getNext_ looks like the following:

```
start_acquisition{
   set prev_eventcounter to 0
[...]
}
[...]
getNext_{
   Software/Hardware Trigger
   ReadData from the card
   if(there is no data) { return }
   Get number of PMT events (i.e. readout windows) in the amount of data read
   Create Metadata
   for(each PMT event){
      get its event counter from the header (it starts from 0)
      if(there are still readout windows to read){
         increment eventsize
         set prev_eventcounter to current eventcounter
         }
         else{
         create the fragment with the read data
         set prev_eventcounter to 0
   }
return
}
```

Figure 1.5: getNext_ function pseudocode

At each call of the getNext_ function a certain amount of data is read from the card. At this stage we don't know how many acquisition windows have been opened (it could be none, one, one and a half, etc..). We just know that n readout windows have been read. Each readout window has it's own event counter in its header which is set to zero at the beginning of every acquisition window. The idea of this code is then fairly simple: we have a global variable (prev_eventcounter) defined in the overlay class that keeps track of the already read readout windows **in an acquisition window**, and a local variable (eventcounter) that keeps track of the already read readout windows **in a getNext_ call**. If the event number is not 0 then we are still

in the same acquisition window, so we want to keep expending the data size of the fragment we will create, and we call again getNext_ . When the event number is again 0, that means a new acquisition window opened, therefore we create the fragment (i.e. we pack all the data of the PMTs in one TPC event) and we set the prev_eventcounter again to zero.

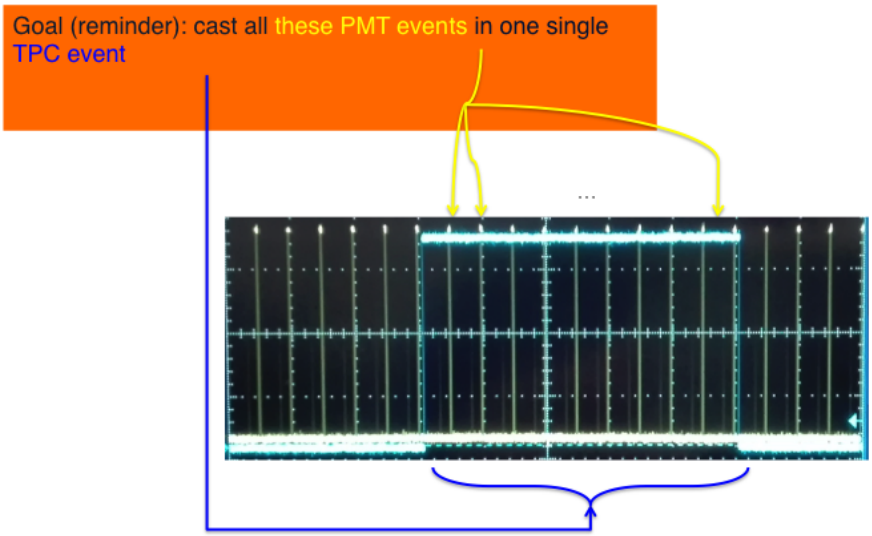

Figure 1.6: Picture of the scope in our operation mode. Blue: acquisition window (i.e. TPC event) at 15 Hz. Yellow: (i.e. readout windows) PMT trigger at 5 kHz

Chapter 2

Performance and timing of the DAQ system

The second major task of the internship was the study of the timing of the DAQ. The good news from the previous chapter is that the DAQ does what we want it to do. However, the acquisition and processing of data is not allowed to last long. In fact it has to keep up with the interaction rate from the BNB (15 Hz).

The speed of the DAQ system is essentially the speed of the getNext_ function, where the acquisition takes place.

The strategy used here was to place several timers in the getNext_ function to know the speed of each block.

As can be seen from the pseudocode in the previous chapter, the main blocks in the getNext_ function are

- the call of the ReadData function,
- the call of the function that gets the number of rw's in the amount of data read,
- initialization of metadata
- the for loop where we create the fragment (or keep asking for data)

For completeness we show here again the pseudocode we showed before, but instead highlighting the timers placed in it to measure its speed (see Fig. 2.1).

The main times of interests are:

1. the time spent in the ReadData function when there is **NO** data to read (end_ReadData - start_ReadData). We can refer to this time as the **NO_data time** or **latency time**.

```
getNext call{
Start_getNext
Trigger
    start_ReadData
    CAENReadData call
    end_ReadData
        If(there is no data){
            ReadData_NO_data = end_ReadData -  start_ReadData
            time_getNext =   ReadData_NO_data
            return}
    end_ReadData
    ReadData = end_ReadData - start_ReadData
    GetNumEvents
    Create Metadata
    Create fragment
    time_getNext = ReadData (+ time to write ReadData time in a file) +
                   time_GetNumEvents + time_CreateMetadata + time_CreateFrag
    return}
```

Figure 2.1: Pseudocode of the getNext_ function highlighting the timers used to measure its speed.

2. the time spent in the ReadData function when there is data to read (end_ReadData - start_ReadData). We can refer to this time as the **ReadData time**.

3. the **getNext_ time**, which is the time from the start of the getNext_ function and the next return statement (we can therefore have a getNext_NOdata and a getNext_YESdata). In the case there is data to read, we compute the getNext_ time as sum of the intermediate timers of the various blocks of the code. This is done so that, if required, we could monitor also the time spent in each single block of the code.

The study of these times is of great importance for the DAQ system. The acquisition time should be dominated by the ReadData function, which at the moment is a CAEN library function. All the are other parts can be controlled by us and then should be made as fast as possible.

Before showing the results, we should point out that all the measurements presented in this report were conducted using a recordLength of 2000 (i.e. 2000 samples), which means a readout time of 4 μs, considering that the sampling of the signal is at 2 $ns/sample$. This is done because 4 μs are enough to readout the slower component of the light produced after an

interaction in the detector.

2.1 Plots and results

In this section we will show and comments the results regarding the timing of the DAQ code as presented in Fig. 2.1.

First we show what happens when we don't find any data to read (NO data) and second when there is data to read (YES data).

2.1.1 NO data

When we issue the test_driver command, most of the times the ReadData function doesn't find any data to read. This allowed us to have immediately a high statistics for the NO_data time (if we set the number of events to generate at 2000 in the configuration file we have a statistics of $\sim 10^6$). Furthermore, the NO_data time and the getNext_NOdata time are basically the same, so we can focus on the latter.

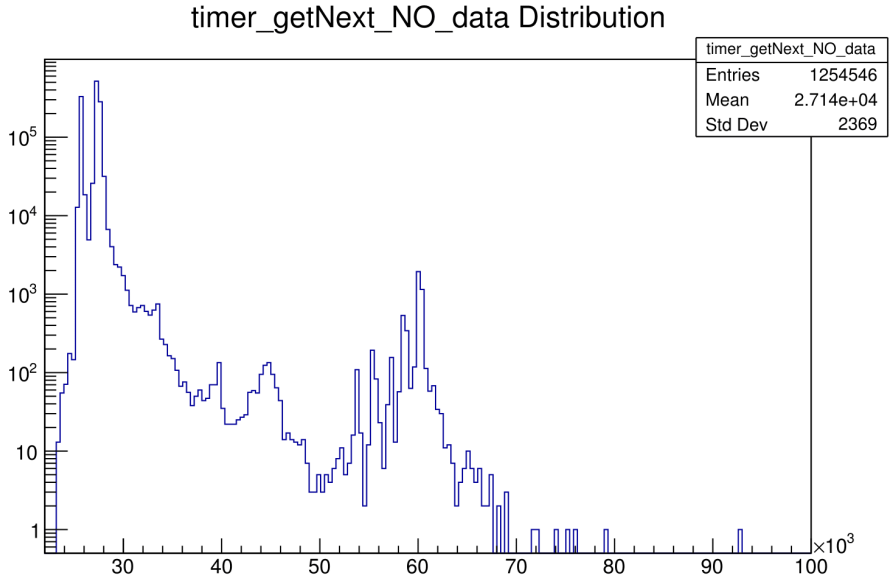

Figure 2.2: Distribution of getNext_NO data time in ns. 200 bins.

Conclusion

As one might hope, when there is no data to read the DAQ doesn't waste much time into the ReadData function. From the plot in Fig. 2.2 we can see that this time is always less than $\sim 100\mu s$ which is way below the 66 ms threshold.

2.1.2 YES data

The statistics for the ReadData and getNext_ times depends on the number of events we want to generate (that we set in the configuration file). Therefore, either one runs the test_driver code many times or runs it once generating many events. We went for the former option in this analysis. In particular, a statistics of $\sim 2\dot{1}0^5$ was enough to have meaningful results.

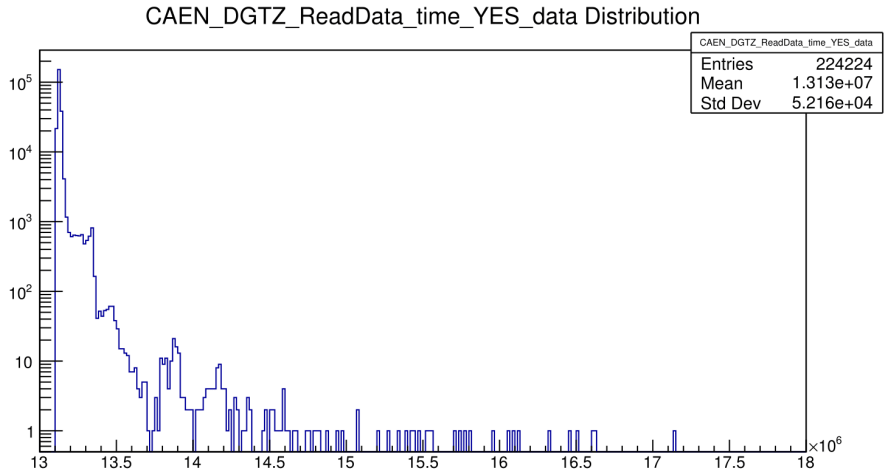

Figure 2.3: Distribution of ReadData_YES_data time in ns. 200 bins.

Conclusion

When the ReadData function finds data to read, the time spent in the getNext_ function increases by 3 orders of magnitude, more precisely they are most ~ 19 ms. This time is mainly dominated by the ReadData time as can be seen from Fig. 2.3. They are shifted by roughly 0.5 ms, which is the time take by the remaining operations (getting the number of events, creating metadata, creating the fragment).

2.1.3 Future investigations

The results we presented in this chapter are surely promising, being this the first version of the DAQ, but not still optimal. The main thing that should be looked at is how the 19 ms spent in the getNext_ function vary when we increase the light readout time for longer, i.e. when we change the recordLength to 4000 or more. If the getNext_ increases linearly with the readout time, than this is a serious issue because we would saturate the threshold imposed by the beam rate.

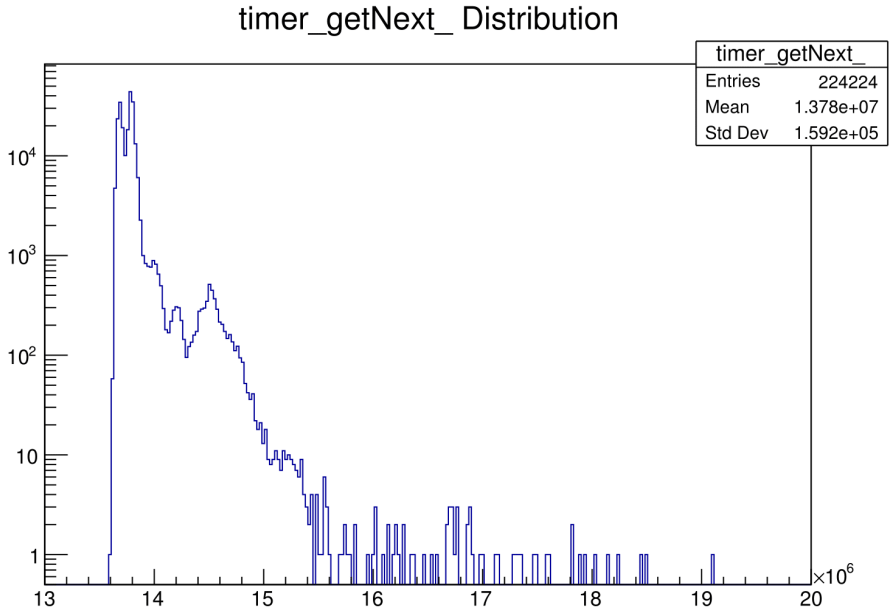

Figure 2.4: Distribution of getNext_YES_data time in *ns*. 300 bins.

To this goal all the code outside the ReadData function should be made as fast as possible (e.g. reducing at a minimum the useless copy of data when creating the fragment).

An additional feature that should be looked at is the total bandwidth, which is the speed at which the data are saved into the local server from the card.