UNIVERSITÀ DEGLI STUDI DI PADOVA

Dipartimento di Fisica e Astronomia "Galileo Galilei"

Corso di Laurea Triennale in Fisica

Tesi di Laurea

Design and development of Data Quality Monitoring protocols for the integration tests of the JUNO large PMT electronics

Relatore

Prof. Alberto Garfagnini

Correlatore

Dr. Beatrice Jelmini

Laureando

Alberto Coppi

Anno Accademico 2020/2021

Contents

Introduction

The Jiangmen Underground Neutrino Observatory (JUNO) is a multi-purpose neutrino experiment under construction in the south of China at 700 m underground. Its main purpose is to determine the Neutrino Mass Ordering (NMO) through the detection of reactor antineutrinos, produced in the Taishan and Yangjiang nuclear power plants, 53 km away from the experimental hall. An excellent energy resolution of 3% at 1 MeV and a large fiducial volume are an important ingredient for reliable results ($3-4\sigma$ significance) within 6 years of data taking.

An important role is also played by the JUNO readout electronic chain, which digitises and processes the signal. This work aims to design and develop a protocol for the integration tests of some components of the JUNO electronics, to be tested after the mass production and before installation in the experiment.

The first chapter introduces the JUNO detector: we highlight its main characteristics, with a focus on the electronics chain which drives the acquisition and online processing of data.
The second chapter describes the protocol developed to test the Global Control Units (GCU) electronic cards and their integration with the full electronics chain. The protocol involves short as well as long tests for a total of 1-2 days for each GCU. The test will take place in Kunshan (China), with about 270 GCUs tested in parallel, aiming at testing about 7000 GCUs in total.

All the work described in the following has been organized in a GitLab repository [1].

Chapter 1

JUNO detector

The JUNO detector [2, 3], shown in Figure 1.1, consists of a large acrylic sphere with a diameter of 35.4 m, called **Central Detector** (CD), supported by a stainless-steel truss. The sphere is filled with 20 kton of **Liquid Scintillator** (LS), surrounded by 17612 20-inch **PhotoMultiplier Tubes** (PMT) and 25600 3-inch PMTs. This structure is immersed in a cylindrical water pool filled with 35 kton of high-purity water which works as a shield from environmental background. The pool is also instrumented with about 2400 20-inch PMTs, which constitutes a **Cherenkov detector** for cosmic muons. Finally, the **top tracker** is placed at the top of the pool. Both the top tracker and the Cherenkov detector are part of the **VETO detector** for cosmic muons.

Inside the CD, electron antineutrinos interact with the protons of the LS through inverse beta decay reactions:

$$\bar{\nu}_e + p \to e^+ + n$$

where the final state positron releases its kinetic energy through scintillation and then annihilates with an electron, thus producing optical photons in number proportional to the energy of the incoming antineutrino. The neutron moderates in water and finally interact with another proton through the reaction:

$$n + p \to d + \gamma$$

This reaction happens, on average, 200 µs after the first one, so that the antineutrino interactions have a very clear prompt-delayed signature. Then, these optical photons can be detected by PMTs and the energy of the antineutrino can thus be reconstructed through various methods [4, 5].

The detector has been designed to reach 3% energy resolution at 1 MeV, needed to obtain significant results within 6 years of data gathering.

1.1 Readout Electronics

Considering the underwater architecture of the large PMTs electronics sketched in Figure 1.2(a), we can notice that groups of three PMTs are connected to water-tight housings, also known as **Under Water Boxes** (UWB), which contain the essential front-end and readout electronics of the system:

- three modules that generate the High Voltage (HV) from a low-voltage input for operating the 3 PMTs independently;
- one **Global Control Unit** (GCU), integrating six Analogue to Digital Units (ADU) and two Field Programmable Gate Arrays (FPGA); the GCU performs the data digitization, buffering and processing, and monitors and controls all the relevant parameters.

The underwater scheme is implemented to minimize the deteriorating effects of analog signal transmission over long cables. The connection between the underwater and the "dry" electronics is achieved through two 100 m long CAT5e/CAT6 cables. The CAT5e one is reserved for slow control operations

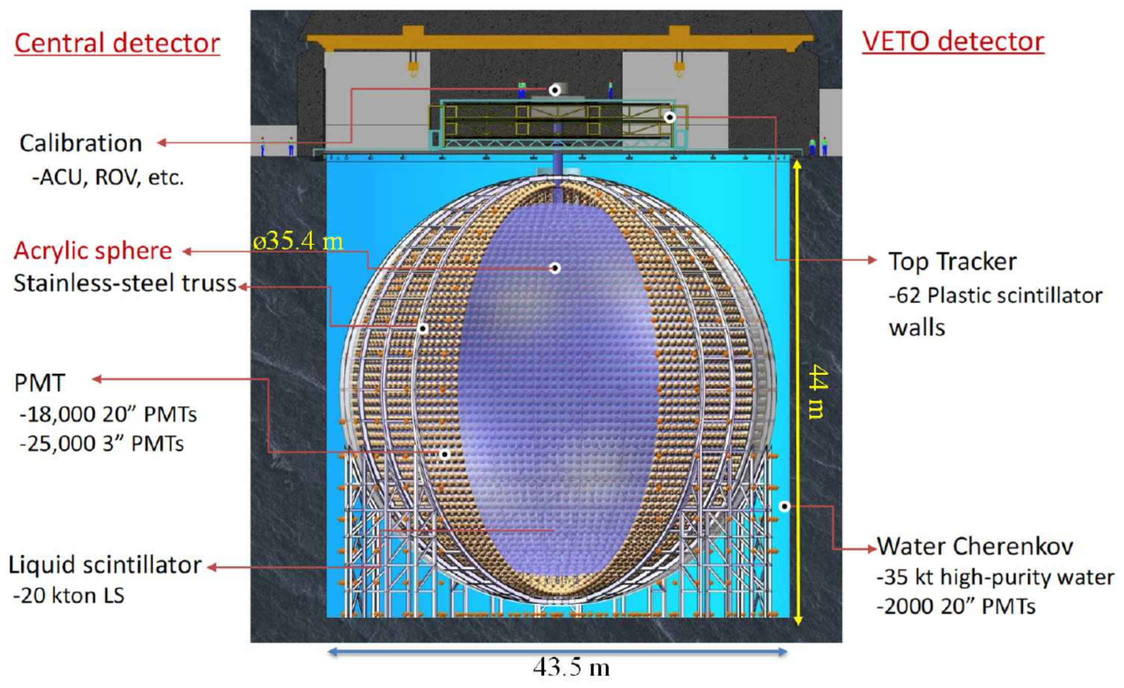
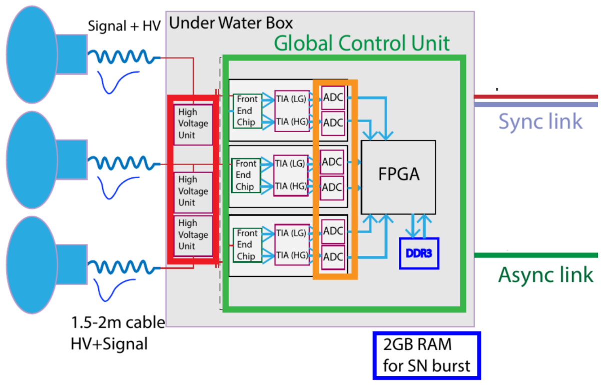
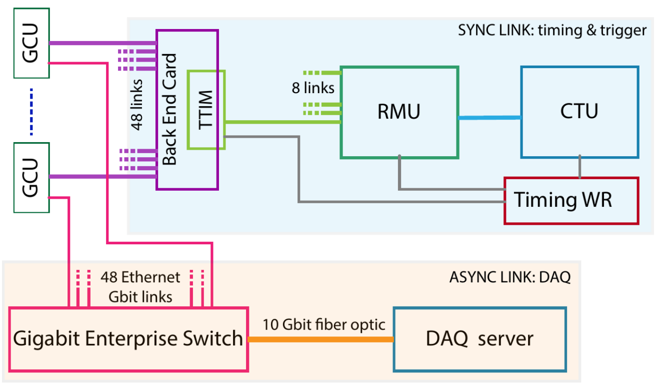


Figure 1.1: Schematic view of the JUNO detector.



(a) "Wet" electronics



(b) "Dry" or backend electronics

Figure 1.2: Schematic view of the Readout Eletronics chain

and data readout, the other one serves as synchronous link for low latency communication between frontend and backend electronics.

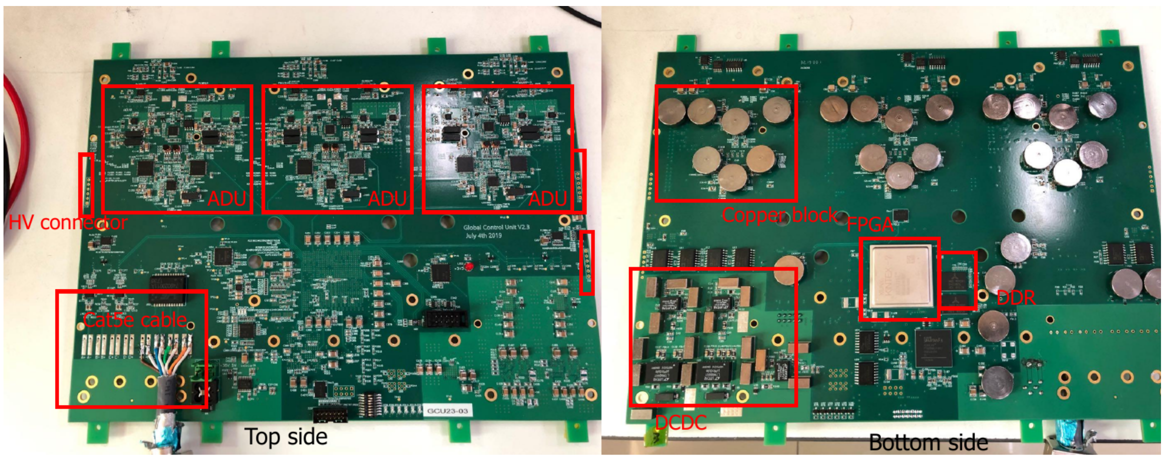
The "dry" electronics is sketched in Figure 1.2(b). It can be divided in two parts: synchronous and asynchronous link. The electronics on the synchronous link is composed by:

- **Central Trigger Unit** (CTU), which receives local triggers generated by the GCUs and may send back a trigger validation signal embedded with the trigger timestamp. It is also responsible for the synchronization of all GCUs internal clocks;
- **Reorganize & Multiplex Unit** (RMU), which makes 21 BEC communicate with the CTU. To do this, 3 FPGAs are mounted, each one dealing with $7x1$Gb/s input links and $1x6.25$Gb/s output link;
- **Back End Card** (BEC), which is responsible for the communication between 48 GCUs and the RMU, the power, and the timing and trigger distribution system. To do so the **Trigger and Time Interfacing Mezzanine** mounted on the BEC deals with the fan-out of the signal and guarantees long distance high speed data transfer;

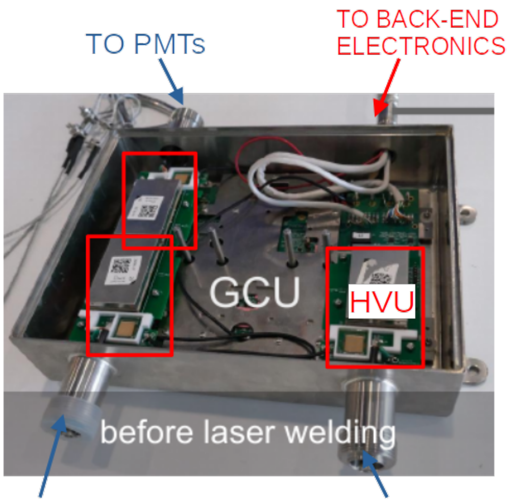
- **White Rabbit switch** (WR), which is responsible for providing a 62.5 MHz clock to the CTU, RMUs and BECs.

PMTs can detect optical photons produced by the antineutrinos interactions, "signal" in the following, but may also acquire background events, *e.g.* cosmic muons, or send to GCUs false signals, coming from the intrinsic PMT dark noise. One of the main tasks of the readout electronics is to recognize data representing possible signal and discard the PMT dark noise. In order to do this a trigger algorithm [6] has been implemented on the GCUs (detailed description of Trigger modes in Section 1.2.2). Note that data acquired by GCUs represents only a small part of the total information about one event, as photons related to a single antineutrino interaction may be revealed by different PMTs connected to different GCUs. Subsequently, the CTU makes a global correlation in order to generate a trigger validation which is then sent back to the GCUs, which finally transfer, through the asynchronous link, all the fragments of the signal to the Data Acquisition system (DAQ).

1.2 Global Control Unit



(a) Last version of GCU hardware. HV connections, ADUs, FPGA and DDR are highlighted



(b) GCU inside the UWBox. HVUs are also mounted

Figure 1.3

The GCU, shown in Figure 1.3, is the intelligent component of the read-out electronics. It is used to perform the readout of the analog signals coming from the PMTs board and to handle all the data packaging, processing and buffering. In case of a supernova explosion, the trigger rate for interesting events will increase, probably to a point where the managing of the data readout via Ethernet would be impossible to be sustained; this has led to the decision of implementing a buffer capable to store at least one second of raw data. To perform this task a DDR3 Random Access Memory (RAM) has been assembled in the GCU. A slow control and monitoring system has been implemented to handle the technical aspects of the experiment, such as high voltages and temperatures, reporting and acting on changes in the status of the electronics. **Both DAQ and slow control operations are carried out via IPbus [7], through the asynchronous link**. The IPbus is a communication protocol for controlling hardware devices. It consists of a virtual bus with 32-bit word addressing and 32-bit data transfer.

The GCU also provides support for the global synchronization process. All of JUNO's GCUs (about 7000) must be synchronized and aligned within a global time in order to tag the triggered events with the correct timestamp. The required time accuracy is 16 ns. BEC, RMU and CTU are synchronized through the White Rabbit Network [8], which exploits the **Precision Time Protocol**, also known as IEEE-1588 protocol [9]. The Timing, Trigger and Control system [10], developed at CERN, is used as a base for the distribution of synchronous broadcast and individually-addressed messages between the BEC and GCUs. The clock used for alignement and synchronization runs at 62.5 MHz (*i.e.* 16 ns of pulse width). The GCU board also presents a 62.5 MHz oscillator that provides a local clock signal used for the IPBus transmissions. The signal processing is carried out by a Xilinx Kintex 7 FPGA, while the reprogramming controller relies on a Xilinx Spartan 6 FPGA.

1.2.1 GCU related hardware

In this section the main hardware components that can be found inside the UWB connected to the GCU board are described. Note that every UWB is connected to three large PMTs. Every PMT is thus identified by a channel number ranging from 0 to 2. The logical pattern which describes how the GCU main hardware components work together is shown in Figure 1.4.

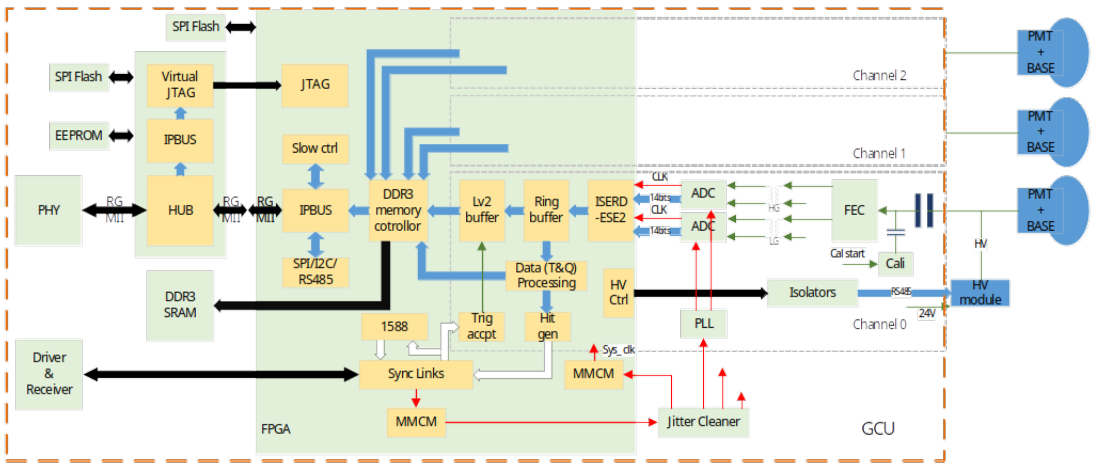


Figure 1.4: GCU hardware logical pattern: the main components are represented.

High Voltage Units

Each PMT is equipped with a voltage divider. The GCU board provides the connections for three High Voltage Units (HVU), also anchored in the UWB (Figure 1.3). For each PMT, a single HV potential will be generated and sent to the PMT through a coaxial cable, about 1.5 m long. Note that the possible signal generated by the PMT is transmitted through the same cable, thus the two signals are coupled. All voltages required for the photo cathode, the field shaping electrodes, and any PMT-specific are derived from this HV potential through the voltage divider. The output voltage range goes from 800 V to 3000 V. The maximum anode current is 300 µA. The PMTs will have their cathodes on ground potential, consequently the output of the signal will be on positive HV. Individual HV units are monitored locally by a micro-controller inside the unit interfaced to the PMT electronic channel, and the parameters are set by the user through the GCU.

Analog to Digital Units

The Analogue to Digital Unit (ADU) is composed by an **Application Specific Integrated Circuit** (ASIC) assembled in the GCU, that digitizes the input signal. The ADU features two TLG121G **Analogue to Digital Converters** (ADC), developed by Tsinghua University [11]. A single ADC digitizes at 14 bit with a rate of 1 Gsps (total bitrate sent to FPGA is 42 Gbps). The digital interface is based on a Double Data Rate (DDR) parallel bus. The data is synchronized with a 500 MHz clock. This sampling clock is granted by an external **Phase-locked loop** mounted on the ADU that receives the GCU system clock of 62.5 MHz. The input analogue signal from the PMT is processed by the ADU in the following steps:

1. the signal is provided to the **Front End Chip** (FEC) component [12], that has the function of protection for the following electronics and is also a current amplifier ASIC with different gains, designed to optimize the input for the following components.
2. The two FEC generated outputs go through a **TransImpedance Amplifier** (TIA) to convert the current signal to a voltage signal through both high (for low energy events) and low (for high energy events) gain. This ensures that ADCs never saturates so that we can collect every detail of the event.
3. The voltage outputs are later converted from a single-ended logic to a differential logic using amplifiers.
4. The signal is finally digitized by the two ADCs into 14-bits words.
5. The 14 bit words coming from each ADC are sent out to the FPGA in Low-Voltage Differential Signaling logic.

The raw digitized stream presents a baseline at a positive value that allows data to be represented by the unsigned data type despite the negative nature of the signal pulses.

It is interesting to notice that the communications between PMTs and GCUs (power and signal) goes through a single coaxial cable similar to the one shown in Figure 1.5. The power to the PMT is in Direct Current (DC) while the signal of an event is in the form of an inverted peak. To purify the signal from the DC power base a capacitor is introduced in series before the signal enters FEC. The circuit thus build is similar to an High Pass Filter and can be studied through Laplace transforms:

Figure 1.5: Coaxial connectors

$$
\begin{aligned}
T(s) &= \frac{Z_C}{Z_C + Z_R} = \frac{(sC)^{-1}}{R + (sC)^{-1}} \overset{\tau=RC}{=} \tau^{-1}\frac{1}{s+\tau^{-1}} \\
V_{DC}(s) &\simeq \frac{V_0}{s} \\
V_{out}(s) &= V_{DC}(s)T(s) = \frac{V_0}{s} + \frac{V_0}{s+\tau^{-1}} \\
V_{out}(t) &= \mathcal{L}^{-1}[V_{out}(s)] = V_0(1 - e^{-t/\tau})
\end{aligned}
$$

where $T(s)$ is the transfer function of the High Pass Filter, $V_{DC}(s)$ is the power signal approximated to be at a fixed value of V_0 and $V_{out}(t)$ is the signal we obtain. It is clear that after a few times τ the signal coming from DC power drops to zero.

Testpulser circuit

The testpulser circuit is sketched in Figure 1.6. The signal, which is a reversed peak is generated through the following steps:

1. an input voltage, V_{DAC}, is set by the user on `ADC0_TPL` point. This is done through IPBus and a Digital to Analog Converter (DAC);

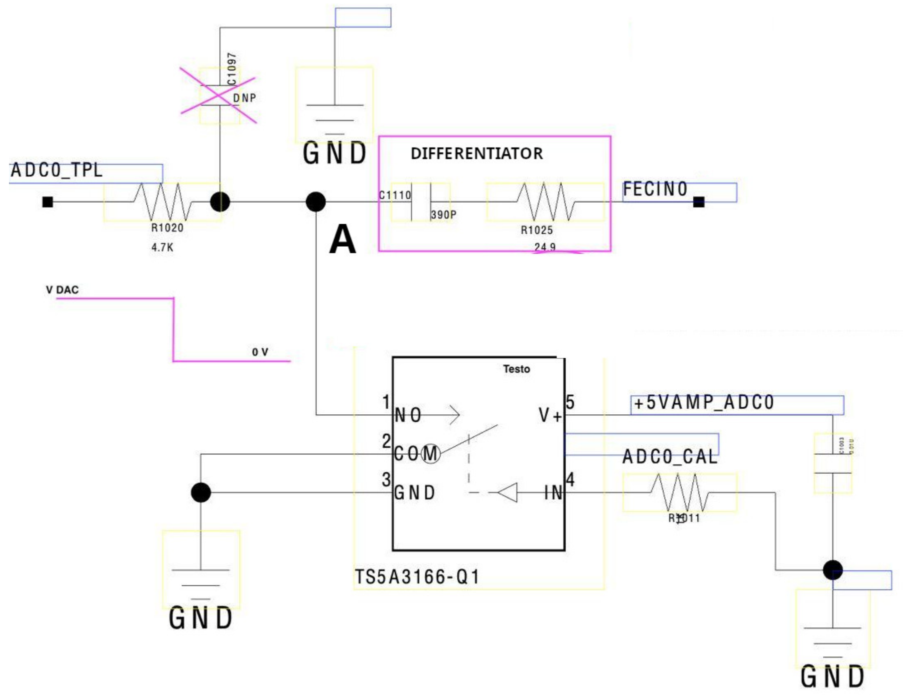


Figure 1.6: Testpulser circuit sketch

2. the user sets also a time interval, Δt. The "A" node is then connected to ground through the integrated circuit and subsequently disconnected. This steps are executed recursively after Δt interval;

3. a very narrow square peak is thus generated on the "A" node with a frequency of $f = 1/\Delta t$ of the form $V_A = V_{DAC}(\Theta(t - t_0) - \Theta(t))$, where $\Theta(t)$ is the Heaviside step function.

4. this waveform is the input of a differentiator circuit, which shapes the input to be similar to a PMT signal; the output is sent to FEC.

Field Programmable Gate Array

All of the tasks the GCU board has to execute, are performed on a FPGA which is an Integrated Circuit designed to be configured by a customer or a designer after manufacturing (hence field-programmable). In an FPGA all the operations are computed by its hardware circuitry and components. FPGAs are vastly used in physical facilities and front-end electronics. For the purposes of the JUNO experiment, a Xilinx Kintex-7 (XC7K160T-2LI FFG676) is used.

Sensors for slow control monitoring

The following is a list of sensors inside the GCU board which we can use for hardware monitoring:

- HV.ch(0,1,2): High Voltage value provided by HVU, one per channel;
- temp.ch(0,1,2): temperature of the HVU, one per channel;
- temp.box: hot spot of the GCU, close to FPGA. This is useful to check if the GCU is overheating;
- VCCINT [13]: FPGA internal supply voltage. It must be around 0.95 V;
- VCCBRAM [13]: the supply voltage for FPGA RAM memorires. It must be around 0.95 V;
- VCCAUX [13]: FPGA auxiliary suplly voltage. It must be around 1.80 V;
- VREFP [13]: external reference voltage for FPGA internal ADC. It must be around 1.25 V;

- VREFN [13]: external reference voltage for FPGA internal ADC. It must be connected to the ground pin of a source around 1.25 V.

Each sensor can be read through IPBus, so that all these values can be monitored during tests as well as during data taking sessions.

1.2.2 GCU Data Management

In this section, a description of the working principles of the GCU Data Management is given. We are describing the data flow from the PMTs to the local memory, and the subsequent flow towards DAQ.

Trigger modes

The JUNO experiment foresees running in two trigger modes: *trigger validation* mode and *auto-trigger* mode. In both cases, a trigger algorithm is used to recognize events: employing a specific threshold level, it determines if there are any possible signals, labels them with a timestamp and saves them in a Level 1 (L1) cache. Both options are discussed more closely in the following.

In case of **Trigger Validation Mode**, each GCU generates a trigger request whenever an event is detected above threshold and waits for a trigger validation signal. The local trigger requests coming from the GCUs are processed by the CTU. If the CTU validates an event, it forms a global triggering decision and sends a command to the GCUs to save the fragment of the validated event in a **First-In-First-Out** (FIFO) memory. The command includes a timestamp, used to save the correct event. Clearly, for this step, GCUs and CTU must be properly synchronized in time, thus the communication goes through the synchronous link. The simplest trigger decision is based on multiplicity, therefore information from all of the channels above threshold is required. Monte Carlo studies [14] have shown that collecting the total number of active channels within a 300 ns window is sufficient to suppress the background from dark noise random coincidences and to guarantee 100% efficiency for the detection of $\bar{\nu_e}$ events. An important aspect regarding energy reconstruction that potentially favors a global trigger is the fact that this is the only configuration that will allow the recording of waveforms for channels in which the PMT signals are below an individual triggering threshold, i.e. the detection of low-charge photoelectron.

In **Auto Trigger Mode** any event (including dark noise pulses and background events) that exceeds the acquisition threshold for a single photoelectron is buffered for a fixed time window (hundreds of nanoseconds) in the FIFO memory and then transmitted, on request, to DAQ.

Raw Data Format

The official JUNO DAQ software requests data from the GCUs FIFO units which then transmit their content through the asynchronous link to store it on the DAQ server in binary files, one per channel. The final structure of acquired data is still under development, but the general idea is that data are grouped in data packets and each packet is composed of a variable number N_w of sequences of 16 bits, which are referred to as **words** and read as hexadecimal values, and stored in binary files. Every data packet is wrapped by two sequences of 8 words, the **header** and the **trailer**, which allow to recognise and separate each packet from the others, providing also important information which uniquely characterises that specific data packet. In the current raw data structure [15], the header is composed of the following words:

1. the header starting word (0x805a, in hexadecimal numeral system), which is fixed for all the data packets;
2. the GCU **channel number**, which can be 0, 1 or 2 and occupies 2 out of 16 bits, and in the future will also include the data type, which identifies if the data packet was acquired through the DDR3 RAM (supernova events), default acquisition or other types (2-3 bits);
3. the data packet size, referred to as **trigger window** (in future this may be referred to as "waveform window"), including header and trailer, given in units of 8 words;

4. the cyclic **trigger count**, a number that increases every time the GCU triggers until it reaches its maximum value of 0xffff (equal to 65535 in decimal numeral system) and starts again from 0x0000;
5. the **timestamp**, a 64 bits sequence (4 words, may be reduced to 3 words, granting almost a month of continuous data taking anyway) which represents the time reference given in units of 8 ns;

The trailer is composed of:

1. the trailer starting sequence, a 6-words pattern fixed for all the data packets;
2. the **GCU ID** number;
3. the trailer ending word (0x0869), which is fixed for all the data packets.

In future the 5th and 6th words may be used for firmware version checksum, thus avoiding misunderstandings during raw data processing and analysis in debug and test sessions.

The addition of other information in place of unused bits is under discussion.

An example of the final digitized waveform, extracted from a data packet, is shown in Figure 1.7.

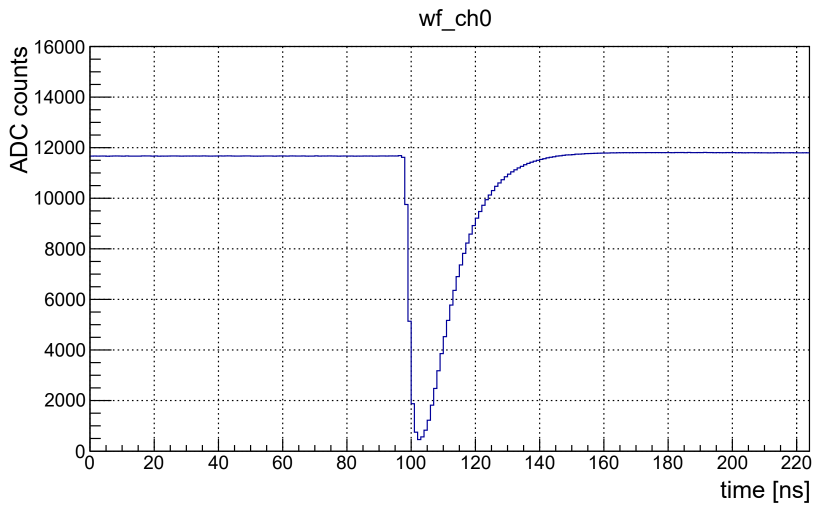


Figure 1.7: Example of the final digitized waveform obtained from a data packet

Chapter 2

Test Protocols

The first chapter highlighted that GCUs are the fundamental piece which enable the JUNO detector to be a potential innovation in the field of neutrino interactions. We remind that they will be installed in Under Water Boxes, connected to PMTs, and they must run for at least 6 years in order to achieve the significance required to determine the Neutrino Mass Ordering. The mass production of these boards is starting and the installation is planned for spring 2022. Thus the need of a test protocol which guarantees the correct behaviour of GCUs becomes urgent.

Three experimental setups are involved to develop and execute the test protocol. The first one is a mock-up installation at **Laboratori Nazionali di Legnaro** (LNL). It is composed of 13 working GCUs and a BEC. GCUs are mounted inside special boxes which provide fans as cooling system and disposed on two racks, as shown in Figure 2.1. The machine used for DAQ and raw data processing as well as debugging of scripts and programs have the following technical specifications:

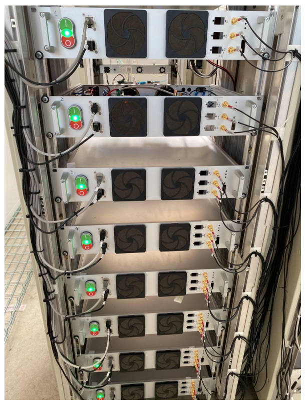


Figure 2.1: Rack with GCUs at LNL

- OS: Ubuntu 18.04 LTS
- CPU: Intel(R) Xeon(R) Silver 4114 CPU @ 2.20GHz, 20 cores 40 threads
- RAM: 32 GB
- Storage: Hard Disks
- Softwares: gcc 7.5.0, ROOT v6-18-04, Python 3.x

Another mock-up installation has been set up at the **Institute of High Energy Physics**, Beijing, China and is composed of 11 GCUs. This setup will be used as proof of portability of all the softwares we used at LNL and to collect more data. Finally, a facility in **Kunshan, China**, has been built for mass production and test purposes. Here, GCUs will be mounted inside UWBs, then tested for 1-2 days; if the UWB passes the tests, it will be laser welded and finally sent to the experimental site for installation. The amount of GCUs that have to be tested is $\simeq 7000$, about 270 at a time due to test room dimensions and space available to place these components.

The first step of the work was to define our goals:

1. provide a few, easy, and clear steps to set up GCUs through user-defined settings;
2. write easy-to-run scripts which enable semi-automatic data acquisition;
3. provide a semi-automatic and fast data processing and analysis over a relatively large amount of data;
4. provide quick, graphical way to check for problems together with detailed analysis for each GCU.

All the protocol scripts and programs, as well as other people's work we rely on, has been grouped and uploaded to a GitLab repository [1]. Here we created two branches:

1. `master`, where we focus on the implementation of tests data acquisition and analysis
2. `setup-dev`, where we focus on the simplification of setup process

This is the status of the repository on 6th September 2021. We plan to modify and update the `setup-dev` branch, adding the automatic configuration of BEC and CTU, in sight of the integration tests in Kunshan.

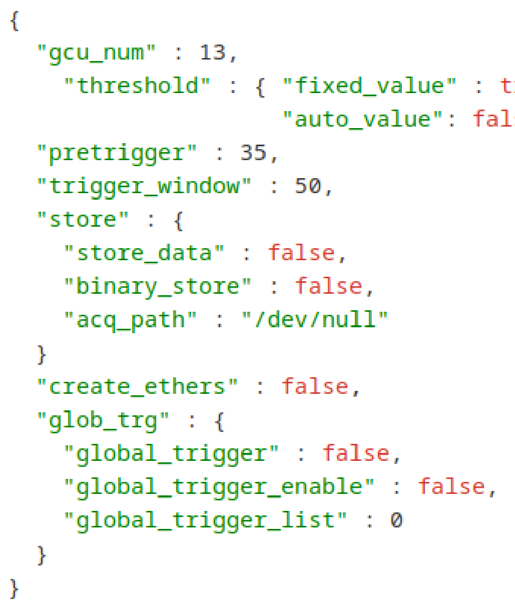
2.1 GCU setup

Every GCU will be connected to computing machines through CAT5e and Gigabit Enterprise Switches based network. In order to individually address each GCU, a QR code is printed on each board, providing a unique MAC Address. These addresses are then written into a file, usually `/etc/ethers`, where they are associated to an Internet Protocol (IP) address (*e.g.* 10.10.10.x at LNL setup). Note that networks usually exploit only the last number of the IP address, thus allowing only 256 (1 byte = 8 bit = 2^8 sequences) possible connections. The network in Kunshan has to expand this range using also the second to last number.

The first part of the work was to reorganize and make more simple the configuration process. We considered the following configurable parameters:

- `gcu_num`, the number of GCUs attached to the network;
- `threshold.fixed_value`, which tells the scripts if the user wants the threshold value to be fixed for all GCUs;
- `threshold.th_value`, the value of the user-defined fixed threshold;
- `threshold.auto_value`, which tells the scripts if the user wants the threshold value to be calculated individually by the GCU for each channel. The GCU then takes some baseline samples and calculates mean and standard deviation;
- `threshold.n_sigmas`, the number of sigmas the user wants the threshold value to be away from the baseline mean value;
- `pretrigger`, the number of words, which is the time in units of 8 ns, read before the waveform falling edge (Figure 1.7), with an offset of about 22 words. For example, if we set the time the GCU starts recording the waveform at 0 and we want the falling edge to be at 144 ns, then pretrigger $= 144/8 + 22 = 40$;
- `trigger_window`, the number of words acquired for each event, including header and trailer. Subtracting 2 we obtain the signal duration in units of 8 ns;
- `store`, together with other similar parameters, it is now useless as for DAQ we are using the official JUNO software;
- `create_ethers`, which following a fixed pattern, creates the ether file, thus skipping the manual addition of MAC and IP addresses;
- `global_trg.global_trigger[global_trigger_enable, global_trigger_list]` through which the user can choose which GCUs to run in local Auto Trigger Mode or in Trigger Validation Mode.

For this purpose, we upgraded the python script `quick_start.py` and renamed it in `quick_start2.py`. This script creates a `Makefile` based on the parameters the user set in the JSON file `setting.json`, shown in Figure 2.2. The `Makefile` then takes care of creating all the files needed for connection and runs all the scripts needed for configuration. In `setup-dev` branch, we got rid of `Makefile`, by including its instructions inside the script `quick_start_nomake.py`. We provide also the bash script



```
{
  "gcu_num" : 13,
    "threshold" : { "fixed_value" : true, "th_value" : 10000,
                    "auto_value": false, "n_sigmas": 5},
  "pretrigger" : 35,
  "trigger_window" : 50,
  "store" : {
    "store_data" : false,
    "binary_store" : false,
    "acq_path" : "/dev/null"
  }
  "create_ethers" : false,
  "glob_trg" : {
    "global_trigger" : false,
    "global_trigger_enable" : false,
    "global_trigger_list" : 0
  }
}
```

Figure 2.2: Content of the `settings.json` file used to set configuration parameters

`template_setenv.sh` which sets some useful environment variables and must be modified accordingly, renamed as `setenv.sh` and run before running tests. All these steps are grouped inside the bash script `setup.sh`, so that the user have simply to:

1. set his/her own settings in `settings.json`;
2. set his/her own settings in `setenv.sh`, open a terminal and run `source setenv.sh`;
3. run `. setup.sh n` if no BEC is installed, `. setup.sh y` otherwise.

2.2 Raw Data Processing

Binary data collected through the DAQ software have to be processed in order to make them more accessible, get rid of useless bits, save disk space, and make their analysis and plotting easier.

The tool `gcu-proc`, developed by *Katharina von Sturm* and *Riccardo Callegari* [16], takes binary data as input and processes them to obtain a ROOT [17] file. This can be done in two different ways:

1. analyzing each channel binary data independently;
2. analyzing all channels together in order to group events with the same timestamp, thus providing a reliable reconstruction of the charge deposited during the event.

The output is a ROOT file containing a `TTree` with several useful quantities, like the waveforms, the values of the baseline and noise, the waveform minimum. The complete structure of the `TTree` is shown in Figure A.1. The file contains also the settings specified for the processing of the data. In the first case, each `vector` has size 1 and the `TTree` has as much entries as the number of the acquired events; in the second case each `vector` has the size equal to the number of channels active during acquisition.

2.3 Tests

In this Section, we describe in detail the purpose of each test, what type of data we need, how we acquire them, how we do the analysis and the outputs we obtain.

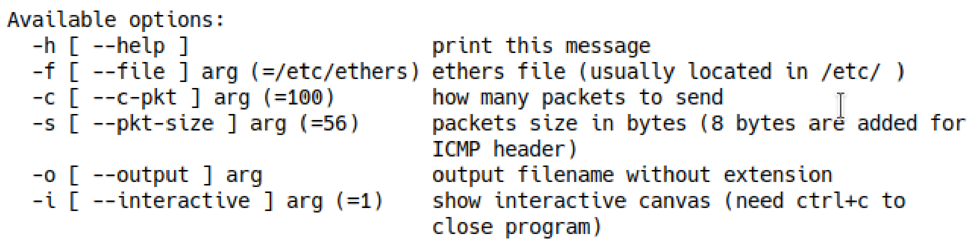
All the analysis programs are written in C++ and based on the ROOT framework [17]. Pursuing adaptability, we exploit the BOOST libraries [18], a C++ library providing excellent and powerful tools for easy programming. In particular, we use `boost_program_options` to read options from command line. This allows us to use positional arguments, that is arguments associated to a variable by their position in the command line instead of their associated option. This means that, *e.g.*, if we have to analyze the files in a folder all together we can simply run: `./<program> -x <some option> *` where the asterisk will be substituted by all filenames inside the current folder.

All the programs can be executed in two different modes: interactive and non-interactive. In interactive mode, the results of the analysis are saved into ROOT files and also plotted to interactive `TCanvas`es. In this way, the user can zoom the graph to directly spot possible problematic behaviours of a GCU and look for its specific results in the output files in a second moment. To end the program, a `Ctrl+c` must be sent to the terminal. In non-interactive mode, no `TCanvas`es are displayed. If running these scripts through a Secure Shell (SSH) tunnel, this can save some time. The program exits automatically after having analyzed all data and having written the output files. All the programs can be automatically compiled by mean of a `Makefile` located in `analysis/` folder.

We know that a large number of GCUs will be tested in Kunshan, with a different setup with respect to the one in Legnaro. Thus we initially decided that testing the GCUs performances by comparison with fixed values was unsuitable. For example, we could acquire a large amount of events from all GCUs and define a mean value for the baseline and its standard deviation (σ), and use these values for comparison. However, the baseline value can be changed through the firmware, potentially making our tests useless. In order to avoid this situation, we plot values we expect to be similar versus the channel ID, which is $ID = 10\ GCU_n + ch_n$, where GCU_n is the GCU number (starting from 0) ad ch_n is the channel number inside that GCU. The large amount of GCUs ensures that outliers are noticeable. These tests are run in Auto Trigger Mode, as we want all the signals read by GCUs to be saved and analyzed.

2.3.1 Test 1: Ping

This test simply checks that all GCUs are connected and running. The analysis program is located in `analysis/ping/test_ping`. This is the only program which takes care of both data acquisition and analysis. Available command line options are shown in Figure 2.3.



```
Available options:
  -h [ --help ]                    print this message
  -f [ --file ] arg (=/etc/ethers) ethers file (usually located in /etc/ )
  -c [ --c-pkt ] arg (=100)        how many packets to send
  -s [ --pkt-size ] arg (=56)      packets size in bytes (8 bytes are added for
                                   ICMP header)
  -o [ --output ] arg              output filename without extension
  -i [ --interactive ] arg (=1)    show interactive canvas (need ctrl+c to
                                   close program)
```

Figure 2.3: Available options for the ping test program

Data acquisition

As first step, a `ping` command is summoned. By default, 100 packets of size 64 KBytes are sent to each GCU. The IP addresses are recovered by default from the file `etc/ethers`. The options `-f -w 1` are used in order to send packets as fast as they come back and wait no more than 1 second for each GCU. The command output is written into a data file in a special way so that useful values can be easily recovered in the next step. Note that the ping tool directly calculates the mean response time and its standard deviation.

Data analysis

The mean response time and its standard deviation as well as the packet loss fraction are recovered from the data file and plotted together versus the GCU_n as shown in Figure 2.4. The mean response time depends on the length of CAT5e cables used to connect GCUs to the PC on the asynchronous

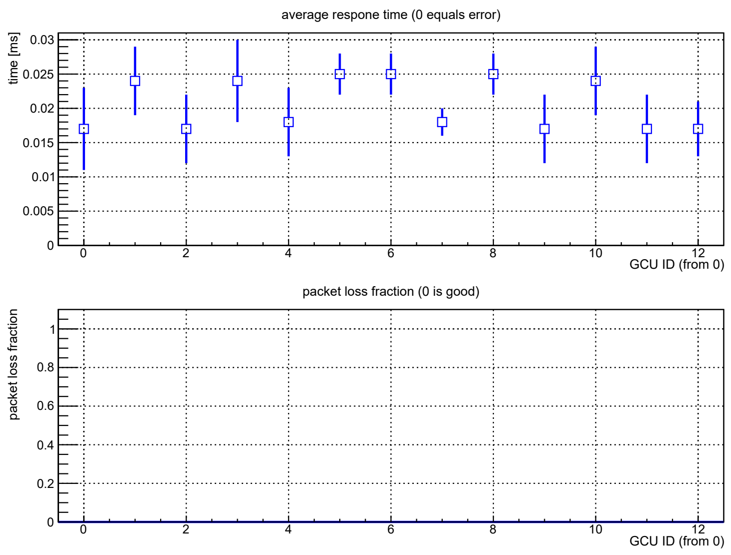


Figure 2.4: Ping test performed with 13 GCUs at LNL. There is no packet loss and the response time is similar for all GCUs

```
Available options:
  -h [ --help ]                print this message
  -f [ --file ] arg            filenames to analyze
  -n [ --num-gcu ] arg         number of GCUs used in acquisition
  -m [ --min-adc ] arg (=220)  value of ADC output below which it saturates
  -s [ --start ] arg (=0)      analyze events strating from <x>
  -o [ --output ] arg          output filename without extension
  -i [ --interactive ] arg (=1) show interactive canvas (need ctrl+c to close
                               program)
```

Figure 2.5: Available options for the ADC linearity test program

link. We expect it to be different from zero. On the other hand the packet loss fraction has to be zero. The `TCanvas` is saved in ROOT, PDF, and PNG files.

2.3.2 Test 2: ADC Linearity

The linearity test checks for strange behaviour of the ADC. To do this, we use the testpulser circuit described in Section 1.2.1. Currently, it is activated and controlled through IPBus protocol. We set different testpulser amplitudes and recover the mean baseline and the waveform minimum (Figure 1.7) of each event.

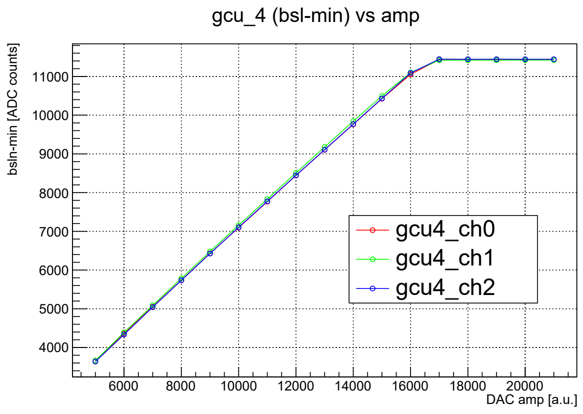
Data acquisition

Data can be automatically acquired by mean of `gcu_acq/run_test2.sh` script. User can modify this script to set its own range of pulse amplitudes. The script has to be run as "`. run_tes2.sh` <gcu> `<optional:channel>` `<optional:frequency (Hz)>`" where <gcu> is the number of GCUs connected, <channel> are the channels the user wants to test (by default '0 1 2') and <frequency> is the frequency of the pulses sent to the ADC. A 300 seconds time data taking is carried out for each chosen pulse amplitude.

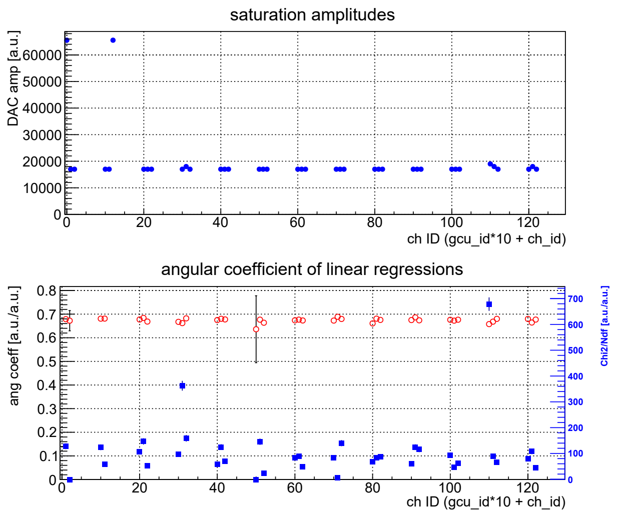
Data are automatically moved to `data/<current_date>/Test2_linearity/run<x>/tp_Amp<y>`, processed by `gcu-proc` without event building, and saved to `data/tier1/<same_as_inputpath>`.

Data analysis

The program `analysis/linearity/test_linearity` is used to analyze data. Available command line options are shown in Figure 2.5. Note that "file" is a positional argument. Data are analyzed recovering the pulse amplitude from data path. Thus it is important to stick to the default paths structure. Then, an histogram of the difference between baseline and waveform minimum is built for each channel of each GCU for each amplitude, filling it with all the corresponding events. The mean and the standard deviation are calculated from these histograms and inserted in linearity plots, like the one shown in Figure 2.6(a). All the linearity plots obtained during a test at LNL with 13 GCUs are shown in Figure A.2 in appendix. Before filling the histograms, the program checks if the waveform minimum is below `min-adc` option value and, if so, that amplitude is labelled as saturation amplitude. If for one channel no saturation is found, this value is set to `UINT32_MAX` $= 2^{16} - 1$. The phenomenon of saturation happens when the input to the ADC is too large to be represented by 14 bits. Then, a graph with the smallest saturation amplitudes versus ID is built. In Figure 2.6(b), data were taken by using the high gain ADU, thus leading to a saturation amplitude of about 20000 DAC counts. We expect that in the final version of the firmware no saturation happens as the GCU automatically switches to the best suitable ADU. The linearity graphs are finally fitted through linear regressions, ranging from 0 to the last amplitude before saturation. The angular coefficient and its error are plotted together with the normalized χ^2, as shown in Figure 2.6(b). All the graphs, except



(a) Example of a linearity plot for the three channels of one GCU



(b) Top: saturation amplitude vs channel ID
Bottom: results from the linear fit vs channel ID, in blue the normilized χ^2

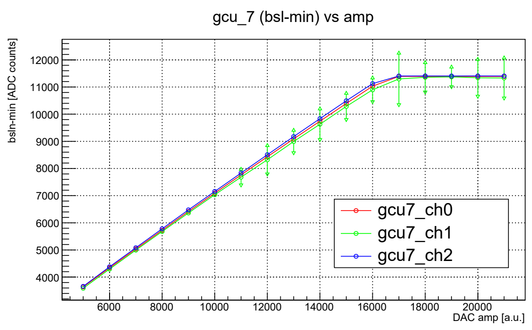
Figure 2.6: Results from a linearity test performed with 13 GCUs at LNL. DAC amplitude ranges from 5000 to 21000 by step of 1000 DAC counts.

histograms, are saved in ROOT, PDF, and PNG files.

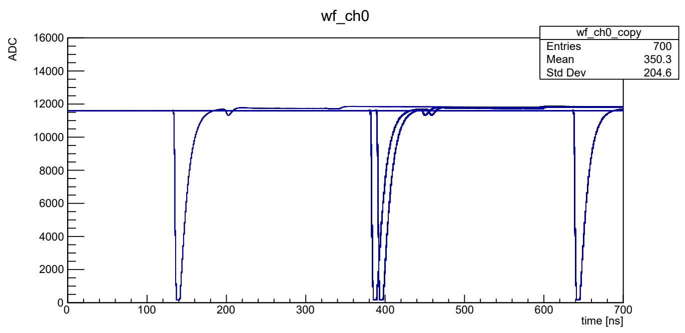
Comments on the obtained results

First of all, we notice that IDs 00 and 12 have the max saturation amplitudes. In fact, this is a known issue in the LNL setup as these channels are dead. Thus, their linearity graphs are not built.

Moreover, as we can see from Figure 2.6(a), we expect every ADC to be linear with respect to the pulse amplitude and the histograms to have a really tight distribution thus leading to very small errors in the linearity graphs. This leads to a very large $\chi^2 = O(10^2)$ in the fitting results. Despite χ^2 is not the most suitable variable to verify the accuracy of the fit because of the small number of data, we chose to depict this value in Figure 2.6(b) as sometimes happens that it approaches zero or is greater with respect to other channels results, thus indicating some strange behaviour. This happens for IDs 02, 50, and 71; for example, the behaviour for ID 71 is shown in Figure 2.7(a). This straightforwardly means that the correspondent histograms have a large distribution. Investigating this phenomenon, we discovered that, despite we set the GCUs in Auto Trigger Mode, some waveforms are misaligned, as shown in Figure 2.7(b).



(a) Linearity plot for GCU 7



(b) Waveforms misalignement for channel 1 of GCU 7

Figure 2.7: Highlight of some unexpected behaviours from one GCU

2.3.3 Test 3: Stability over time

During the acquisition of data, we expect some values to be stable. This means that, if we measure these quantities over time, their distribution has to be narrow. As we exploit a fixed amplitude pulse, the quantities we consider for this test are: waveform minimum position (min bin), baseline mean, baseline sigma and charge reconstructed by each channel. Please note that these values are correlated to `gcu-proc` settings, so the user have to correctly set them in order to have reliable results.

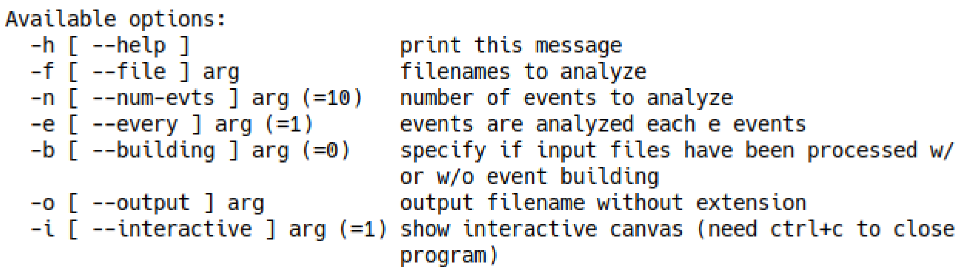
Data acquisition

Data can be automatically acquired by means of `gcu_acq/testpulse/run_test3.sh` script. The script has to be run as "`. run_test3.sh <gcu>`" where <gcu> is the number of GCUs connected. The user can modify the script in order to change the duration of data acquisition, the test pulse frequency, and its amplitude. By default data are taken for one day at a frequency of 0.5 Hz. We suggest 0.1-1 Hz as frequency range, in order to occupy less space on hard disks. The pulse amplitude is fixed at 15000 DAC counts, thus suitable for testing the high gain ADU, but another one day data taking can be performed with an amplitude of 50000 DAC counts to test also the low gain ADU.

Data are automatically moved to `data/<current_date>/Test3_stability/run<x>/tp_Amp<y>`, processed by `gcu-proc` without event building, and saved to `data/tier1/<same_as_inputpath>`.

Data analysis

The analysis program is located in `analysis/stability/test_stability`. Available command line options are shown in Figure 2.8. Note that "file" is a positional argument.



```
Available options:
  -h [ --help ]               print this message
  -f [ --file ] arg           filenames to analyze
  -n [ --num-evts ] arg (=10) number of events to analyze
  -e [ --every ] arg (=1)     events are analyzed each e events
  -b [ --building ] arg (=0)  specify if input files have been processed w/
                              or w/o event building
  -o [ --output ] arg         output filename without extension
  -i [ --interactive ] arg (=1) show interactive canvas (need ctrl+c to close
                              program)
```

Figure 2.8: Available options for the stability test program

Given the frequency, the user can adjust "num-evts" and "every" to plot an event every <x> seconds.

For example, to plot one event every minute from a dataset taken for one day at 1 Hz:

$$t_{\text{acq}} = 1\ day = 86\,400\,\text{s}$$
$$f = 1\,\text{Hz}$$
$$n_{\text{evts, tot}} = t_{\text{acq}} \cdot f = 86400$$
$$every = 1\ min \cdot f - 1 = 59$$
$$numevts = \frac{n_{\text{evts, tot}}}{every + 1} = 1440$$

Then, the program simply builds value versus ID graphs, accordingly with user options, which are the projection of 3D graphs to 2D graphs through suppression of the time coordinate. An example is shown in Figure 2.9. We chose this type of format because it is simpler to spot a strange behaviour and identify the channel ID. These graphs are saved to ROOT, PDF, and PNG files.

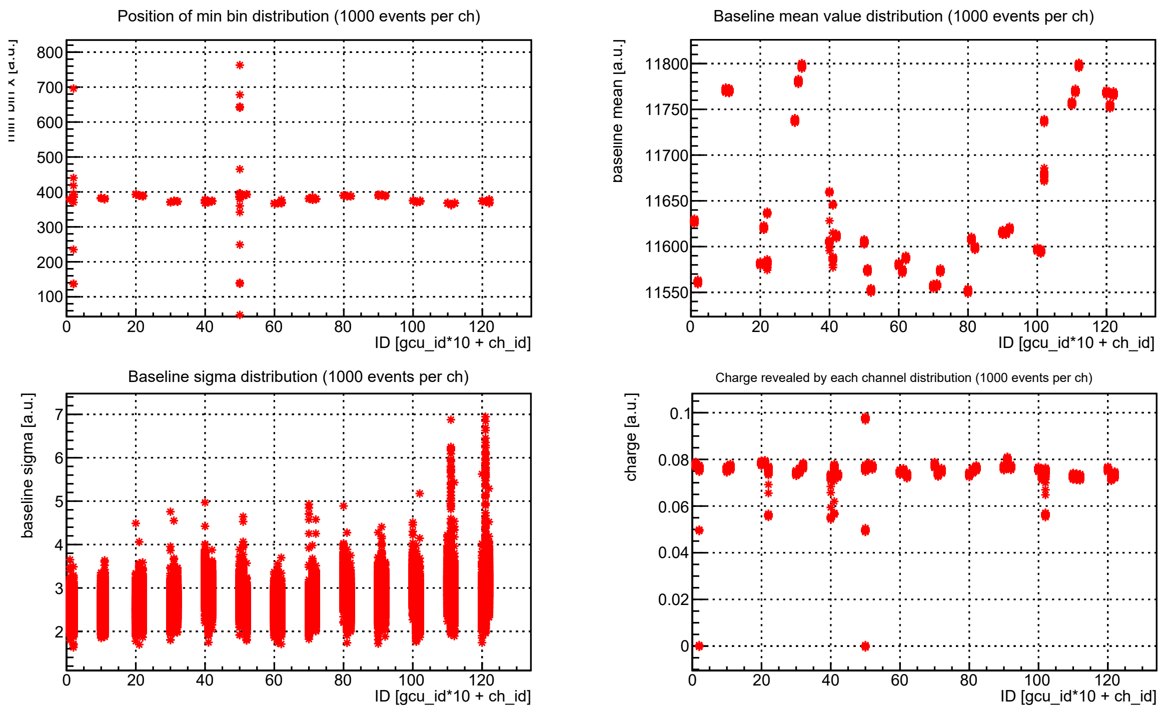


Figure 2.9: Results from stability test performed with 13 GCUs at LNL. Data were taken for 1 day at 0.5 Hz and one event per minute is plotted.

Comments on the obtained results

Besides dead IDs 00 and 12, which are not displayed as they didn't acquire any data, we notice that IDs 02 and 50 have a strange behaviour. In fact, the position of the waveform minimum and the charge reconstructed have a very large distribution, despite most of the values are in the same range as others IDs. These results confirm the waveforms misalignement spotted in the previous test.

2.3.4 Test 4: Slowcontrol

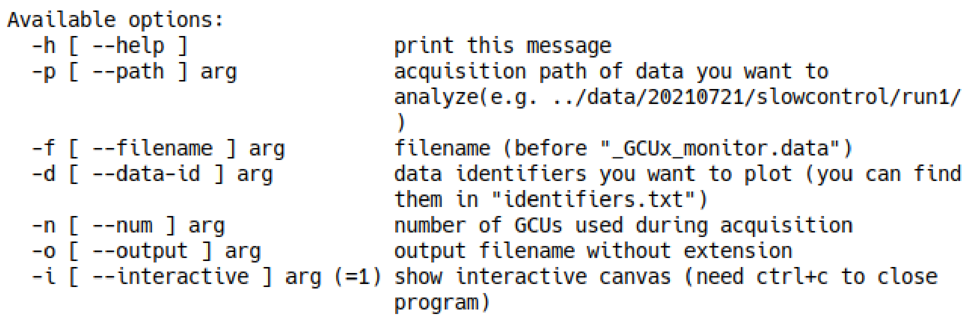
This test aims to keep track of all the values that the sensors installed on the GCU measure. We do this through IPBus protocol and neither the DAQ software, nor the testpulser are involved. Thus Slowcontrol can be performed in parallel with all other tests to save time and also to verify the correct functionality of the components while they are working and stressed. We focus on the `temp.box` sensor as it is located in the hottest spot of the GCU, on the FPGA, and it can reveal if the UWB is overheating. In Figure A.3 in appendix, other sensors' readouts are shown.

Data acquisition

To acquire data, we use the scripts `gcu_acq/slowcontrol/monitor.sh`, in which we define some functions, and `gcu_acq/slowcontrol/monitor_start.sh`, which calls these functions and starts the acquisition. The latter can be run as "`./monitor_start <gcus:13> <sleep:time-in-seconds>`", where the first argument is the number of GCUs attached to the system and the second one is the time interval between two consecutive measurements. We made this script executable so that, when the user is connected through SSH tunneling, it can be run in background by means of `nohup` [19]. All readable sensors are acquired, together with their acquisition time, and data are saved to the path `data/<current_date>/Test4_slowcontrol/run<x>/`, in a special csv-like format, ROOT interpretable. To read sensors, we use programs written by *Ivano Lippi* on behalf of INFN-LNL, which test one sensor at a time, thus every measure takes about 1 second per GCU. The acquisition is continuous, until a kill signal is sent to the process.

Data analysis

The analysis is performed by `analysis/slowcontrol/test_slowcontrol`. Available command line options are shown in Figure 2.10. Note that "data-id" is a positional argument and it represents the sensors whose data are used for the plots.



```
Available options:
  -h [ --help ]               print this message
  -p [ --path ] arg           acquisition path of data you want to
                              analyze(e.g. ../data/20210721/slowcontrol/run1/
                              )
  -f [ --filename ] arg       filename (before "_GCUx_monitor.data")
  -d [ --data-id ] arg        data identifiers you want to plot (you can find
                              them in "identifiers.txt")
  -n [ --num ] arg            number of GCUs used during acquisition
  -o [ --output ] arg         output filename without extension
  -i [ --interactive ] arg (=1) show interactive canvas (need ctrl+c to close
                              program)
```

Figure 2.10: Available options for the slowcontrol test program

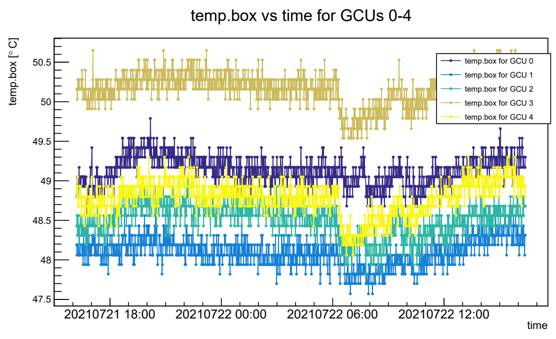
The program then reads data and paints them into different `TCanvas`es, with 5 GCUs each. An example of `temp.box` plot can be found in Figure 2.11. These plots are saved in ROOT, PDF, and PNG files.

Comments on the obtained results

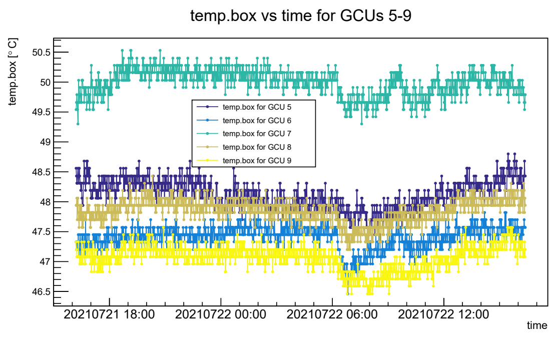
We note that GCUs 3, 7, 11 and 12 present a $\sim 2\,^{\circ}\mathrm{C}$ higher temperature with respect to other GCUs. This effect may be caused by their different position in the rack or small differences in the cooling system. Assuming a room temperature of $25\,^{\circ}\mathrm{C}$, we see that the temperature difference between the room and the FPGA is about 22 to $25\,^{\circ}\mathrm{C}$.

It is also interesting to compare these results to the ones obtained during the under water test [20] carried out at Y-40 The Deep Joy, a pool 42.15 m deep filled with thermal water. This test took place on May 23-25 and involved one GCU which was in a UWB. The sealed box has been slowly immersed to the deepest point of the pool, turned on and the testpulser was finally activated. In Figure 2.12 results are displayed.

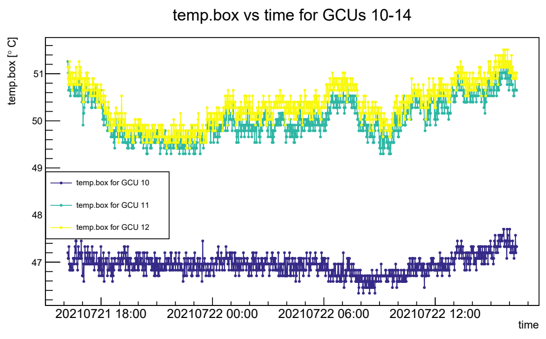
Note that inside UWBs the GCUs can dissipate heat only passively. Despite this, the FPGA temperature, revealed by the `temp.box` sensor, stabilizes around $56\,^{\circ}\mathrm{C}$ after about 1 hour from when the GCU was switched on. As the water temperature is about $33\,^{\circ}\mathrm{C}$, we obtain a temperature difference $\Delta T \simeq 23\,^{\circ}\mathrm{C}$, very similar to the result obtained at LNL. It is shown also that the baseline slightly moves down as temperature rises to its stable value.



(a)

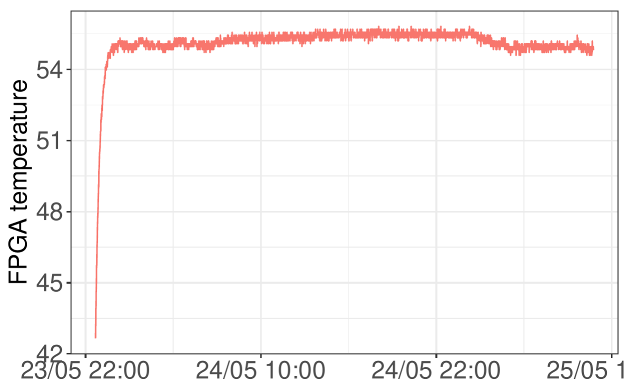


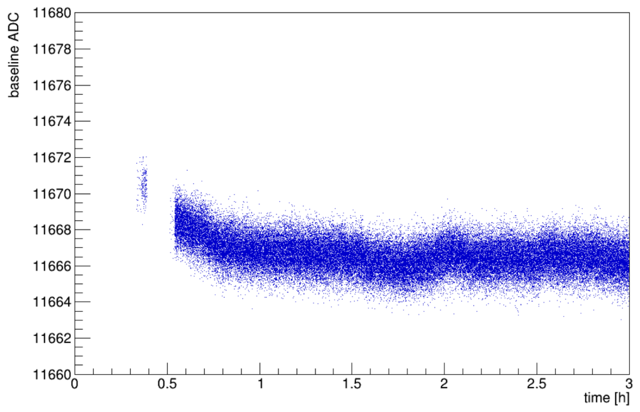
(b)



(c)

Figure 2.11: Results from a slowcontrol test performed with 13 GCUs at LNL. Only readings from the `temp.box` sensor are shown.





(a) Results from slowcontrol test. Read `temp.box` sensor.

(b) Results from stability test. Baseline value is monitored over time.

Figure 2.12: Results from tests at Y-40 with 1 GCU inside the UWB.

Chapter 3

Conclusions

The JUNO large PMTs electronics test protocol has been designed, developed and tested on the LNL setup. We achieved a fast and simple GCUs setting up, easy-to-use scripts for semi-automatic data acquisition, and developed analysis program. We defined a fixed structure for data saving and results printing and drafted a step by step manual, trying to reduce human fallacy.

The analysis programs are very efficient, especially in non-interactive mode, taking only a few seconds to obtain the results. This reduces downtime. They occupy a small amount of RAM and don't stress the CPU in running time, also on less powerful machines. They rely on very common tools as the BOOST libraries and the ROOT framework, thus granting portability on almost all machines used for scientific purposes. The results shown highlight also that they are effective in spotting strange behaviours of GCUs, providing a quick tool to identify broken boards.

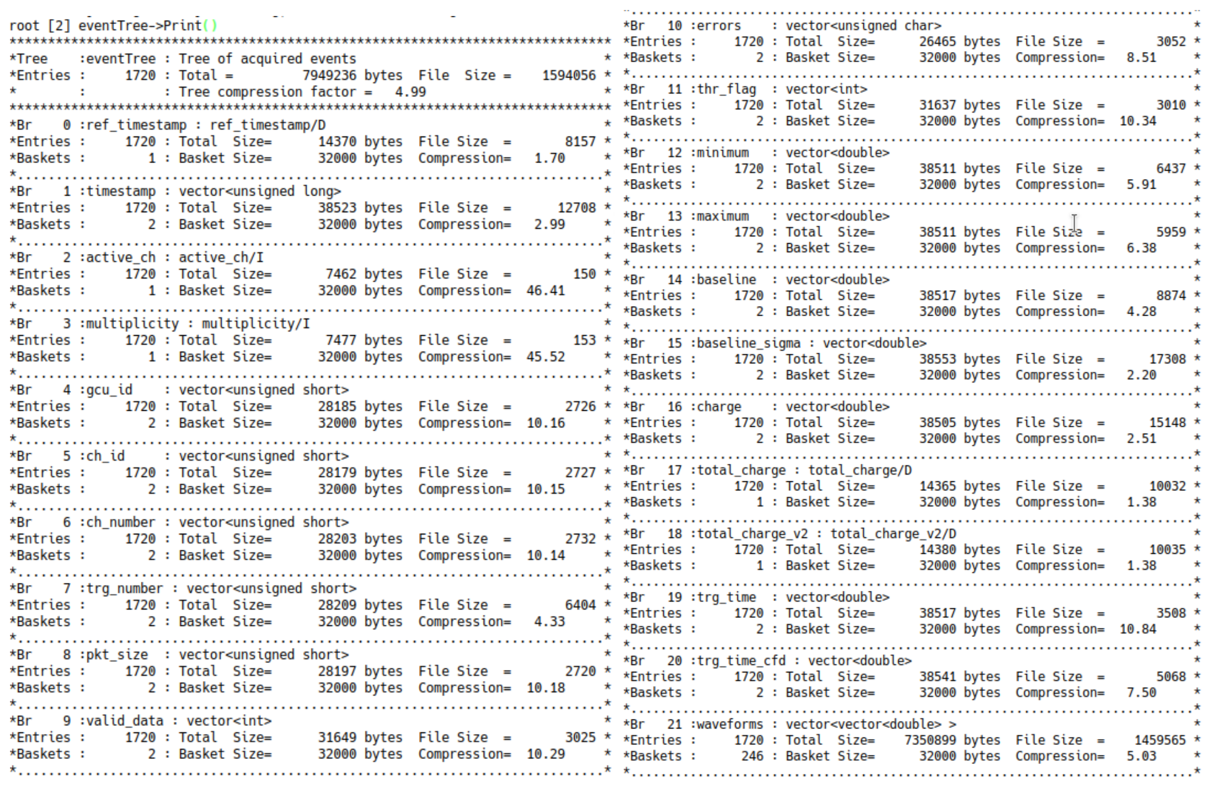
The protocol will be used for the integration tests of the electronics during the mass production phase and it will be a basis for the tests carried out during the installation in JUNO, in spring 2022. Thus, it will be tested also at IHEP, providing another useful test platform to gather data on the protocol behaviour, its usability, and its reliability.

A future possible development is to define some values to be compared between different setups. For example we want to define a reconstructed charge value in the stability test, associated with its standard deviation; if any GCU present a different value (out of a range defined through the standard deviation) for this quantity it is automatically marked as problematic. The objective is to produce a single output where GCUs are marked with "passed/not passed" labels, preserving the possibility to consult all the plots and tools already implemented.

Appendix A

Secondary figures and plots

The Figure A.1 shows the complete structure of the `eventTree` saved in the output ROOT file of `gcu-proc`.



```
root [2] eventTree->Print()
******************************************************************************
*Tree    :eventTree : Tree of acquired events                                *
*Entries :     1720 : Total =         7949236 bytes  File  Size =    1594056 *
*        :          : Tree compression factor =   4.99                       *
******************************************************************************
*Br    0 :ref_timestamp : ref_timestamp/D                                    *
*Entries :     1720 : Total  Size=      14370 bytes  File Size  =       8157 *
*Baskets :        1 : Basket Size=      32000 bytes  Compression=   1.70     *
*............................................................................*
*Br    1 :timestamp : vector<unsigned long>                                  *
*Entries :     1720 : Total  Size=      38523 bytes  File Size  =      12708 *
*Baskets :        2 : Basket Size=      32000 bytes  Compression=   2.99     *
*............................................................................*
*Br    2 :active_ch : active_ch/I                                            *
*Entries :     1720 : Total  Size=       7462 bytes  File Size  =        150 *
*Baskets :        1 : Basket Size=      32000 bytes  Compression=  46.41     *
*............................................................................*
*Br    3 :multiplicity : multiplicity/I                                      *
*Entries :     1720 : Total  Size=       7477 bytes  File Size  =        153 *
*Baskets :        1 : Basket Size=      32000 bytes  Compression=  45.52     *
*............................................................................*
*Br    4 :gcu_id    : vector<unsigned short>                                 *
*Entries :     1720 : Total  Size=      28185 bytes  File Size  =       2726 *
*Baskets :        2 : Basket Size=      32000 bytes  Compression=  10.16     *
*............................................................................*
*Br    5 :ch_id     : vector<unsigned short>                                 *
*Entries :     1720 : Total  Size=      28179 bytes  File Size  =       2727 *
*Baskets :        2 : Basket Size=      32000 bytes  Compression=  10.15     *
*............................................................................*
*Br    6 :ch_number : vector<unsigned short>                                 *
*Entries :     1720 : Total  Size=      28203 bytes  File Size  =       2732 *
*Baskets :        2 : Basket Size=      32000 bytes  Compression=  10.14     *
*............................................................................*
*Br    7 :trg_number : vector<unsigned short>                                *
*Entries :     1720 : Total  Size=      28209 bytes  File Size  =       6404 *
*Baskets :        2 : Basket Size=      32000 bytes  Compression=   4.33     *
*............................................................................*
*Br    8 :pkt_size  : vector<unsigned short>                                 *
*Entries :     1720 : Total  Size=      28197 bytes  File Size  =       2720 *
*Baskets :        2 : Basket Size=      32000 bytes  Compression=  10.18     *
*............................................................................*
*Br    9 :valid_data : vector<int>                                           *
*Entries :     1720 : Total  Size=      31649 bytes  File Size  =       3025 *
*Baskets :        2 : Basket Size=      32000 bytes  Compression=  10.29     *
*............................................................................*
```

(a)

```
*............................................................................
*Br   10 :errors    : vector<unsigned char>                                  *
*Entries :     1720 : Total  Size=      26465 bytes  File Size  =       3052 *
*Baskets :        2 : Basket Size=      32000 bytes  Compression=   8.51     *
*............................................................................*
*Br   11 :thr_flag  : vector<int>                                            *
*Entries :     1720 : Total  Size=      31637 bytes  File Size  =       3010 *
*Baskets :        2 : Basket Size=      32000 bytes  Compression=  10.34     *
*............................................................................*
*Br   12 :minimum   : vector<double>                                         *
*Entries :     1720 : Total  Size=      38511 bytes  File Size  =       6437 *
*Baskets :        2 : Basket Size=      32000 bytes  Compression=   5.91     *
*............................................................................*
*Br   13 :maximum   : vector<double>                                         *
*Entries :     1720 : Total  Size=      38511 bytes  File Size  =       5959 *
*Baskets :        2 : Basket Size=      32000 bytes  Compression=   6.38     *
*............................................................................*
*Br   14 :baseline  : vector<double>                                         *
*Entries :     1720 : Total  Size=      38517 bytes  File Size  =       8874 *
*Baskets :        2 : Basket Size=      32000 bytes  Compression=   4.28     *
*............................................................................*
*Br   15 :baseline_sigma : vector<double>                                    *
*Entries :     1720 : Total  Size=      38553 bytes  File Size  =      17308 *
*Baskets :        2 : Basket Size=      32000 bytes  Compression=   2.20     *
*............................................................................*
*Br   16 :charge    : vector<double>                                         *
*Entries :     1720 : Total  Size=      38505 bytes  File Size  =      15148 *
*Baskets :        2 : Basket Size=      32000 bytes  Compression=   2.51     *
*............................................................................*
*Br   17 :total_charge : total_charge/D                                      *
*Entries :     1720 : Total  Size=      14365 bytes  File Size  =      10032 *
*Baskets :        1 : Basket Size=      32000 bytes  Compression=   1.38     *
*............................................................................*
*Br   18 :total_charge_v2 : total_charge_v2/D                                *
*Entries :     1720 : Total  Size=      14380 bytes  File Size  =      10035 *
*Baskets :        1 : Basket Size=      32000 bytes  Compression=   1.38     *
*............................................................................*
*Br   19 :trg_time  : vector<double>                                         *
*Entries :     1720 : Total  Size=      38517 bytes  File Size  =       3508 *
*Baskets :        2 : Basket Size=      32000 bytes  Compression=  10.84     *
*............................................................................*
*Br   20 :trg_time_cfd : vector<double>                                      *
*Entries :     1720 : Total  Size=      38541 bytes  File Size  =       5068 *
*Baskets :        2 : Basket Size=      32000 bytes  Compression=   7.50     *
*............................................................................*
*Br   21 :waveforms : vector<vector<double> >                                *
*Entries :     1720 : Total  Size=    7350899 bytes  File Size  =    1459565 *
*Baskets :      246 : Basket Size=      32000 bytes  Compression=   5.03     *
*............................................................................*
```

(b)

Figure A.1: ROOT `TTree` structure of binary data independently processed with `gcu-proc`

We show also other results from the linearity test performed at LNL with 13 GCUs in Figure A.2. Note that GCU 0, 5 and 7 exhibit the same strange behaviour due to waveform misalignment.

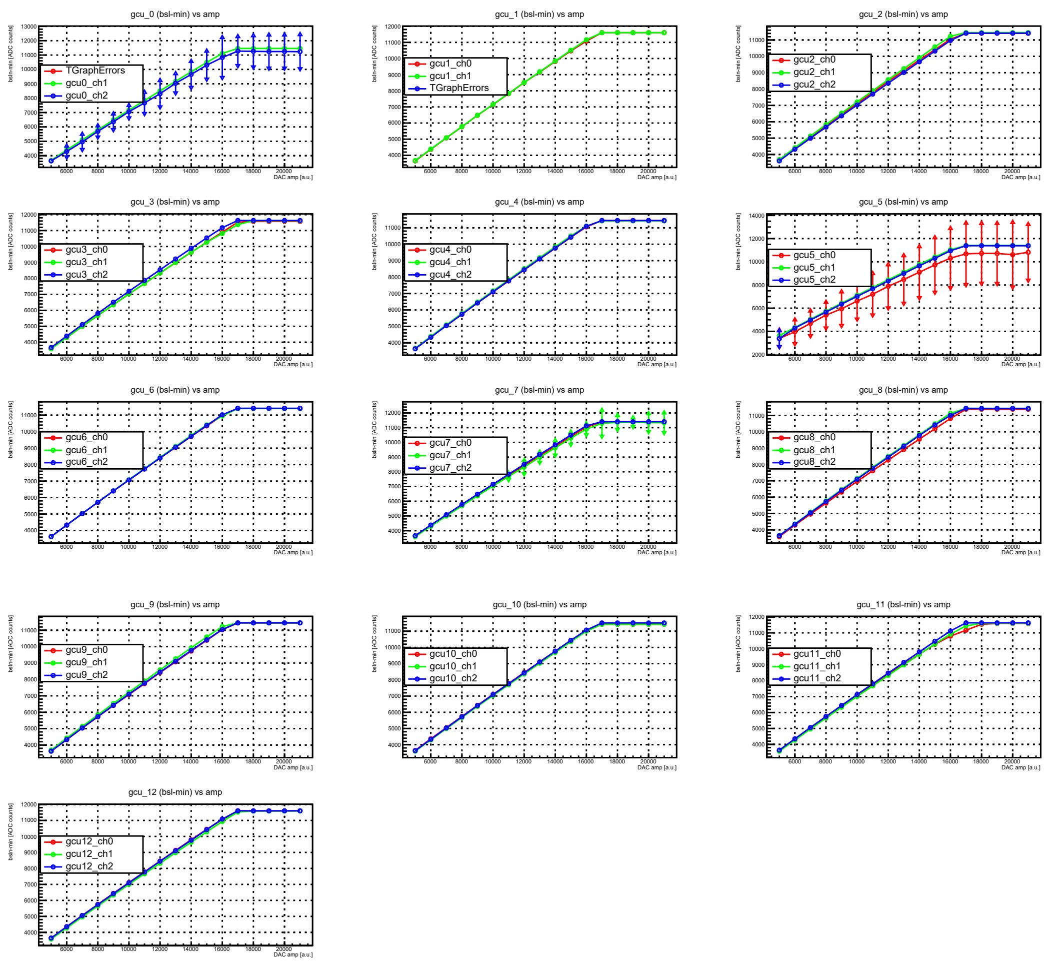


Figure A.2: Linearity plots of all 13 GCUs at LNL. DAC amplitude ranges from 5000 to 21000 DAC counts.

In Figure A.3 there are two plots obtained from the same data taking shown as slowcontrol test results example. We depict two different values, read from `HV.ch0` and `VCCBRAM` sensors. We see that these potentials are very stable.

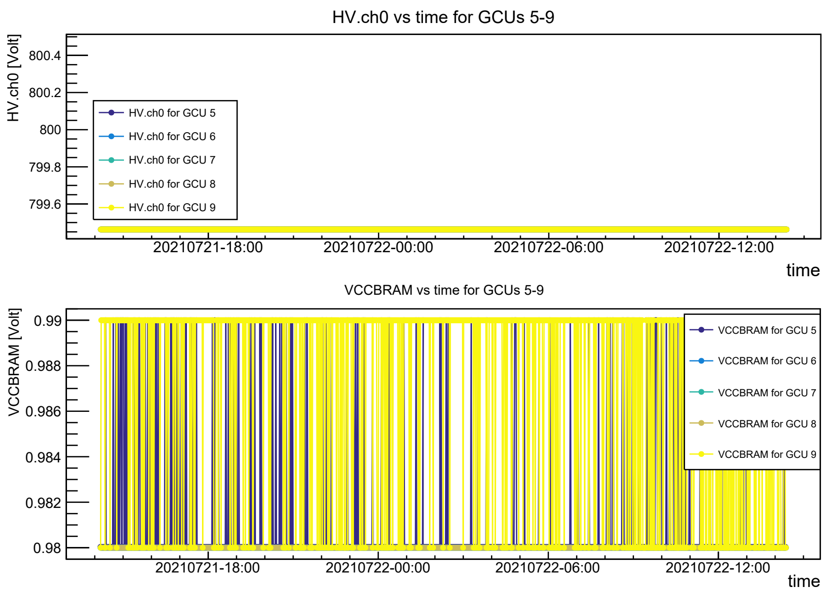


Figure A.3: Results from slowcontrol test performed with 13 GCUs at LNL. Readings from `HV.ch0` and `VCCBRAM` from GCUs 5 to 9 are shown

Bibliography

[1] Alberto Coppi and Beatrice Jelmini. Sept. 2021. URL: https://baltig.infn.it/coppi/gcu_integration_tests.

[2] Angel Abusleme et al. *JUNO Physics and Detector*. 2021. arXiv: 2104.02565 [hep-ex].

[3] Fengpeng An et al. "Neutrino physics with JUNO". In: *Journal of Physics G: Nuclear and Particle Physics* 43.3 030401 (Feb. 2016). ISSN: 1361-6471. DOI: 10.1088/0954-3899/43/3/030401. arXiv: 1507.05613. URL: http://dx.doi.org/10.1088/0954-3899/43/3/030401.

[4] W. Wu et al. "A new method of energy reconstruction for large spherical liquid scintillator detectors". In: *Journal of Instrumentation* 14.03 (Mar. 2019), P03009–P03009. DOI: 10.1088/1748-0221/14/03/p03009. URL: https://doi.org/10.1088/1748-0221/14/03/p03009.

[5] Zhen Qian et al. "Vertex and energy reconstruction in JUNO with machine learning methods". In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 1010 (Sept. 2021), p. 165527. ISSN: 0168-9002. DOI: 10.1016/j.nima.2021.165527. URL: http://dx.doi.org/10.1016/j.nima.2021.165527.

[6] Filippo Marini. "Design and tests of the FPGA embedded trigger algorithms for the large PMT JUNO Electronics". 2018. URL: http://tesi.cab.unipd.it/59743/.

[7] C. Ghabrous Larrea et al. "IPbus: a flexible Ethernet-based control system for xTCA hardware". In: *Journal of Instrumentation* 10.02 (Feb. 2015), pp. C02019–C02019. DOI: 10.1088/1748-0221/10/02/c02019. URL: https://doi.org/10.1088/1748-0221/10/02/c02019.

[8] CERN. *White Rabbit Official CERN website*. URL: https://white-rabbit.web.cern.ch/.

[9] D. Pedretti et al. "Nanoseconds Timing System Based on IEEE 1588 FPGA Implementation". In: *IEEE Transactions on Nuclear Science* 66.7 (July 2019), pp. 1151–1158. ISSN: 1558-1578. DOI: 10.1109/tns.2019.2906045. URL: http://dx.doi.org/10.1109/TNS.2019.2906045.

[10] CERN. *The TTC Website*. URL: https://ttc.web.cern.ch/.

[11] Ru Jin et al. "Reliability study of custom designed ADC for the Jiangmen underground neutrino observatory". In: *Radiation Detection Technology and Methods* 4 (Apr. 2020), pp. 203–207. DOI: 10.1007/s41605-020-00171-3.

[12] Yan X.B. et al. "FEC". In: *JUNO internal document database* JUNO-doc-2432-v1 (Apr. 2017).

[13] XILINX. *Kintex-7 FPGAs Data Sheet: DC and AC Switching Characteristics*. Mar. 2021. URL: https://www.xilinx.com/support/documentation/data_sheets/ds182_Kintex_7_Data_Sheet.pdf.

[14] T. Adam et al. *JUNO Conceptual Design Report*. 2015. arXiv: 1508.07166 [physics.ins-det].

[15] Filippo Marini. "New Header/Trailer structure". In: *JUNO internal document database* JUNO-doc-7449-v1 (Aug. 2021).

[16] Riccardo Callegari. "Characterization and tests of 39 channels of the JUNO large PMT electronics". 2020. URL: http://tesi.cab.unipd.it/65100/.

[17] CERN. *ROOT Data Analysis Framework*. URL: root.cern.

[18] *BOOST C++ Libraries*. URL: https://www.boost.org/.

[19] *nohup*. URL: https://en.wikipedia.org/wiki/Nohup.

[20] Katharina von Sturm et al. *UWbox test in deep water in collaboration with Y-40*. May 2021.