Dual Digitizer Readout and Run Control Implementation

at the FASER *experiment*

Edward Galantay

Supervised by Prof. Anna Sfyrla

Co-supervised by Dr. Claire Antel
and Dr. Brian Petersen

Department of Nuclear and Particle Physics

Faculty of Sciences

University of Geneva

UNIVERSITÉ
DE GENÈVE
FACULTÉ DES SCIENCES

July, 2024

Acknowledgements

I would like to thank all those who made this thesis possible:
Prof. Anna Sfyrla, who suggested this project.
Dr. Claire Antel for helping me during the long and multiple tests of the Run Control and for reviewing and providing feedback during the writing of the thesis.
Dr. Brian Petersen, for his help in working with the digitizers and for providing the opportunity to work on the detector in the tunnel.
Dr. Enrico Gamberini, for guiding me through all the cryptic C++ errors during the DAQling integration.
Not forgetting my parents, who agreed to proofread the thesis, thoroughly looking for (and finding) mistakes.

Abstract

FASER (ForwArd Search ExpeRiment) is a small and inexpensive LHC experiment, located in a previously unused tunnel, designed to probe the far-forward region of the ATLAS interaction point for Long-Lived Particles (LLPs) such as dark photons or Axion-Like Particles (ALPs).
The FASER detector underwent a calorimeter upgrade that significantly extends the operational energy range of the calorimeter using a split-readout method but that required adding a second digitizer to the FASER TDAQ system.
A new dual-readout system of the digitizers has thus been implemented, with special attention on the synchronization between the digitizers. This system has been in operational use since February 2024, following successful high-rate tests on the full detector.
Additionally, the FASER Run Control software, which facilitates control of the data acquisition system through an intuitive web interface, required a major overhaul to improve modularity, maintainability, and to be capable of automatic recovery of the TDAQ system in case of issues. This overhaul involved significant architectural changes and updates to subsystems such as configuration, logging, and error handling. It facilitated the addition of features to FASER, such as automatic restarts and opens the way for other experiments to use the Run Control.

Contents

List of Abbreviations

1

Introduction

1.1 | FASER at CERN

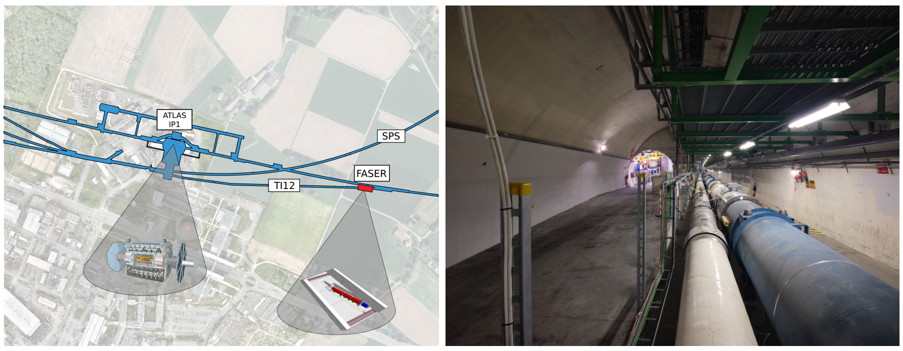
With its 27 km ring, the Large Hadron Collider (LHC) at CERN is the world's largest particle accelerator and a remarkable achievement in terms of its construction, control and maintenance. The LHC has enabled major advances to be made in our understanding of the Standard Model (SM) and has led to the discovery in 2012 of the last puzzle piece of the Standard Model, the Higgs boson. Since beginning operations in 2008, two operational periods (Run 1 and Run 2) have been successfully completed and Run 3 is currently ongoing and will continue until the end of 2025 [1]. After three years of shutdown (Long Shutdown 3), the next phase, Run 4, will begin, marking the start of the High-Luminosity LHC (HL-LHC) era. This major upgrade will enable significantly more collisions per second, thanks to improved magnets, collimators, superconductors, and other advancements [2].

Despite extensive studies, no physics beyond the Standard Model has been found, motivating the search for new physics in unexplored regions such as forward physics, which studies high-energy particle collisions at very small angles.

The major experiments of the LHC were designed to study high transverse momentum (p_T) phenomena, making these detectors highly inefficient for low pseudorapidity (forward region) studies. However, this forward region presents unique opportunities to investigate much lighter particles that are weakly coupled to the Standard Model.

The Forward Search Experiment (FASER) is a small (5 m x R=0.1 m) and inexpensive experiment at CERN that takes advantage of the existing LHC and ATLAS infrastructure to target light and weakly-interacting particles in the far forward region of the ATLAS interaction point IP1. Proposed in 2017 and approved by CERN in 2019, FASER was built over the next two years and began taking data in summer 2022, at the start of LHC Run 3. FASER will be operational during Run 3 and Run 4 during HL-LHC.

The aim of FASER is to study physical processes in these forward regions, potentially leading to the discovery of new light and weak interacting particles, ranging from MeV



(a) (b)

Figure 1.1: FASER location with respect to ATLAS, the SPS, and the LHC (a). Picture of FASER near the LHC tunnel (b). The picture was taken in the direction of IP1.*Picture: Maximilien Brice.*

to GeV, which are less coupled to the Standard Model, such as dark photons or axion-like particles. These weakly-interacting particles typically travel great distances before decaying into Standard Model particles, necessitating a detector placed at a sufficient distance from their production site to study their decay products.

FASER is located on the beam collision axis, 480 meters from the ATLAS Interaction Point 1 (IP1) in the TI12 tunnel, previously used to link the Super Proton Synchrotron (SPS) and Large Electron-Positron Collider (LEP), as shown in Figure 1.1. This location is separated by over 100 meters of rock from IP1.

The later addition of FASERν [3], an emulsion detector with a target mass of 1.1 tonnes, allows studies of high-energy neutrinos of all flavors produced from proton-proton collisions at IP1.
Since the beginning of LHC Run 3 in summer 2022, FASER has demonstrated its capabilities, including the first direct observation of neutrino interactions at a particle collider experiment [4, 5].

1.2 | Physics motivation

By placing FASER 480 meters from IP1 in the tunnel, FASER hopes to find light, very weakly-interacting particles LLPs, produced at IP1, travelling long distances through concrete and rock without interacting and then decaying into visible particles within the detector's decay volume. Such particles are produced with transverse momentum approximately equal to their mass $p_T \sim m$. Taking into account FASER's acceptance,

$\theta \leq 1$ mrad, their energy can be deduced to be around the TeV:

$$\theta \simeq \tan\theta = \frac{p_T}{p} \sim \frac{m}{E} \ll 1 \tag{1.1}$$

allowing hypothetical decays such as:

$$\text{LLP} \to e^+e^-, \mu^+\mu^-, \pi^+\pi^-, \gamma\gamma, \dots \tag{1.2}$$

The main Long-Lived Particle candidate is the dark photon. Dark photons A' are part of dark sector models. These models propose a new spin-1 vector boson that interacts weakly with SM particles through kinetic mixing with the SM photon. The dark photon acts as a mediator between the SM and hypothetical dark matter particles. FASER is sensitive to dark photon masses $m_{A'}$ in the $2m_e < m_{A'} < 2m_\mu$ range. The decay product would then be a positron and an electron $A' \to e^+e^-$, with a detector signature shown in Figure 1.2.

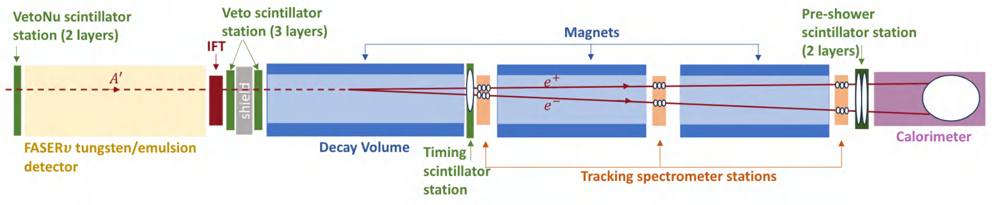

VetoNu scintillator station (2 layers)
Veto scintillator station (3 layers)
IFT
shield
Magnets
Pre-shower scintillator station (2 layers)
A'
e^+
e^-
FASERν tungsten/emulsion detector
Decay Volume
Timing scintillator station
Tracking spectrometer stations
Calorimeter

Figure 1.2: Signature example of a dark photon decay into a positron and an electron. The dark photon enters the detector from the left and decays in the decay volume. [6]

Dark photons potentially decaying in the FASER volume are primarily produced from the decay of π^0 mesons, with a branching ratio of the $\pi^0 \to \gamma\gamma$ process modified by the dark photon's mass :

$$B\left(\pi^0 \to A'\gamma\right) = 2\varepsilon^2 \left(1 - \frac{m_{A'}^2}{m_{\pi^0}^2}\right)^3 B\left(\pi^0 \to \gamma\gamma\right)$$

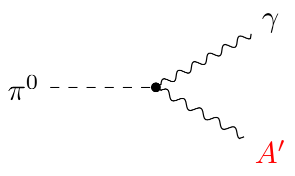


Figure 1.3: Production of a dark photon via pion decay.

Another possible Long-Lived Particle (LLP) candidate would be Axion-Like Particles, that could be produced by very high energy photons interacting with the atomic nuclei of the Neutral Particle Absorber (TAN) (shown on Figure 1.5) via the Primakoff effect and finally decaying into a pair of photons.

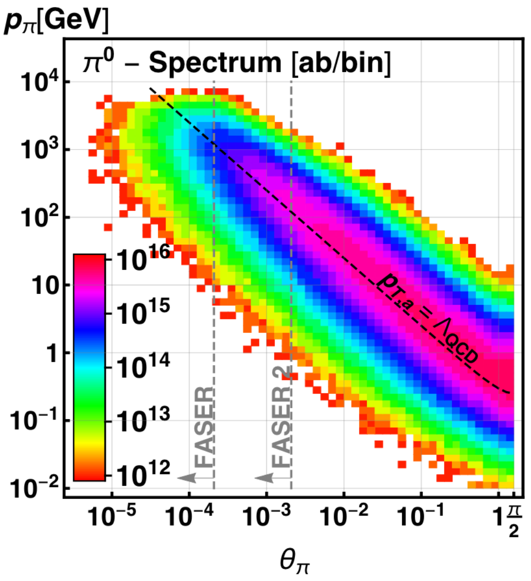


Figure 1.4: Differential pion production rate in the (θ, p) plane, with respect to the beam axis and momentum, from Monte-Carlo simulations. The angular acceptance for FASER and a potential upgraded new experiment, FASER2 are indicated by the vertical dashed lines [7]

.

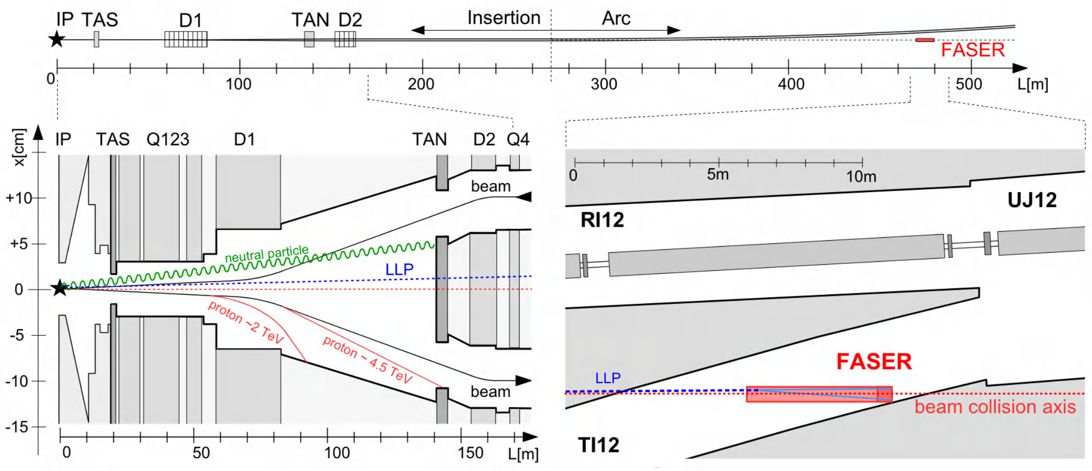
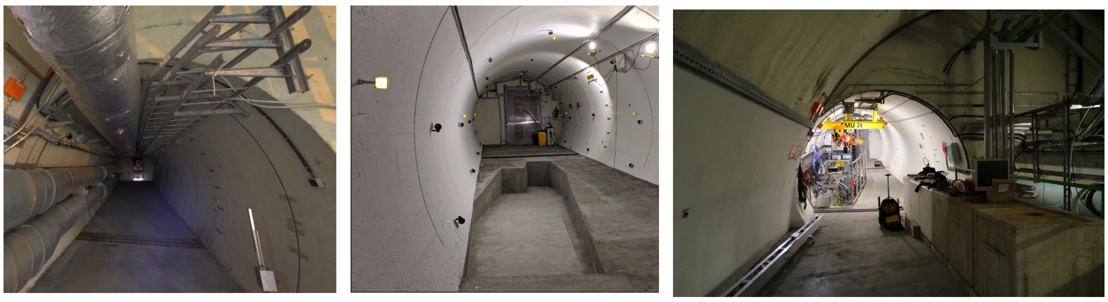


Figure 1.5: Production of LLP from IP1 to FASER. The TAN protects the LHC magnets from radiation, but can also be used as a fixed-target experiment for Axion-Like Particle (ALP) production [7].

2 The FASER Detector

2.1 | Overview

As mentioned in section 1.1 , the TI12 tunnel was chosen as the location for the FASER experiment because of its suitable orientation with respect to the IP1 line of sight (LOS). However, it was necessary to adjust the vertical inclination of the experiment, as the tunnel is inclined. To this end, a trench was dug in the tunnel floor, over a distance of approximately 6 meters to a maximum depth of 64 cm (see Figure 2.1 b). In this trench, an aluminum base serves as a support for the experiment.



(a) Before FASER work in the TI12 tunnel. [8]

(b) After TI12 cleared out and the trench finished. [8]

(c) Current layout of FASER, during YETS 2023-24.

Figure 2.1: TI12 evolution : from unused tunnel to the FASER cavern, in YETS 2023-24.

After a certain time of data taking, it became apparent that background noise was coming from stray particles coming from the beam in the direction of IP1, caused by secondary interactions of the incoming beam with a quadrupole magnet close to FASER. Concrete blocks were therefore installed between the TI12 entrance and the LHC tunnel (as seen in the picture in Figure 2.1 (c), bottom right).

FASER is composed of several subsystems (with the corresponding colors in Figure 2.2):

- The magnets (blue)
- The tracking system (brown)
- The scintillators (yellow) and the calorimeter (red)
- FASERν (dark red)

Each subsystem is briefly explained in the following sections.

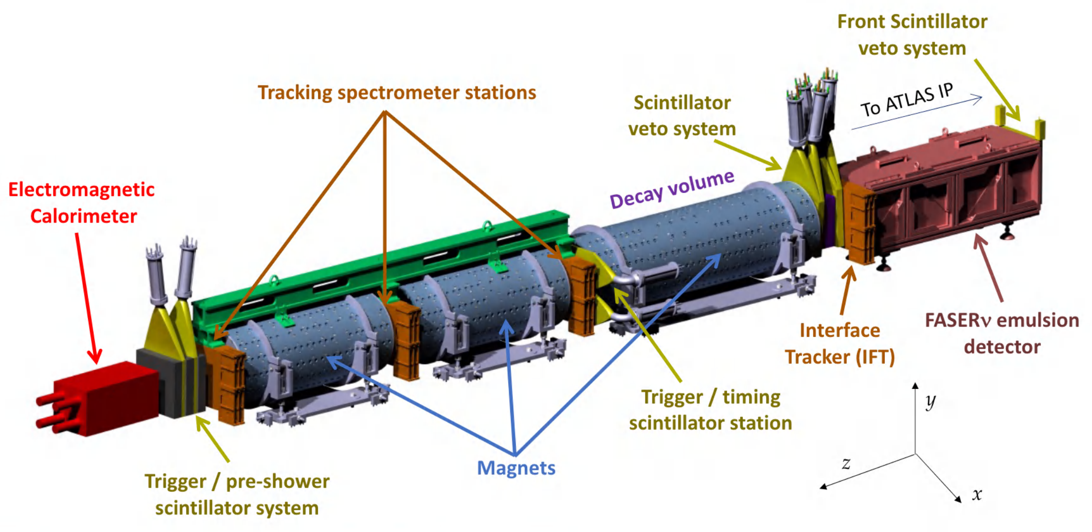

Front Scintillator veto system
Tracking spectrometer stations
Scintillator veto system
To ATLAS IP
Electromagnetic Calorimeter
Decay volume
Interface Tracker (IFT)
FASERν emulsion detector
Trigger / timing scintillator station
y
z
x
Trigger / pre-shower scintillator system
Magnets

Figure 2.2: The FASER detector with its sub-systems. The z-direction points in the opposite direction to the ATLAS IP1. [8]

2.2 | Magnets

To further separate the slightly separated particles arriving in FASER, three permanent dipole magnets ($r_{\text{inner}} = 0.1$ cm, $r_{\text{outer}} = 21.5$ cm) of 0.57 T are used. The permanent nature of the magnets requires no high voltage power supply or cooling, greatly reducing monitoring and maintenance.

The first magnet (1.5m long) surrounds the decay volume, and the other two (1m long) are used in the tracking spectrometer.

The three magnets, based on the Halbach design [9], enable a compact architecture while ensuring an strong, homogeneous magnetic field inside the magnets.

As shown in Figure 2.3, each 16x12 (18) block constituting the short (long) magnet is made of Samarium Cobalt Sm_2Co_{17}.

The magnets are one of the elements of FASER made specifically for the experiment, making them the most expensive parts of the experiment.

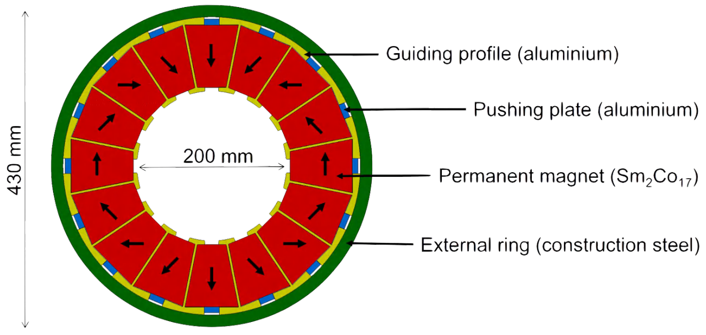

Guiding profile (aluminium)
Pushing plate (aluminium)
Permanent magnet (Sm_2Co_{17})
External ring (construction steel)
200 mm
430 mm

Figure 2.3: Cross-section of the dipole magnets used for the FASER experiment. Each block has a different magnetic field direction to produce a homogeneous field inside the assembled magnet. [8]

2.3 | Tracking system

The tracking system consists of the tracking spectrometer and the Interface Tracker (IFT). The tracking spectrometer, consisting of three stations located at either end of the two short magnets and covering the entire magnet apertures, detects two oppositely-charged particles from an LLP decay in the decay volume. A resolution of $O(100\ \mu\text{m})$ enables us to detect these very collimated particles and attempt to reconstruct their trajectory.

The IFT consists of one station, located just after FASERν, and acts as a link between the emulsion detector tracks and the other detector tracking stations. The spectrometer tracking and IFT stations (Figure 2.5 b) are each composed of three layers. One layer is itself made up of eight silicon strip modules (SCT), spares from the ATLAS SCT barrel detector (shown in Figure 2.4). These double-sided modules are staggered, four per side, as shown in Figure 2.5 a.

With the exception of the SCT modules, everything else (readout, cooling, etc.) has been specifically designed for FASER.

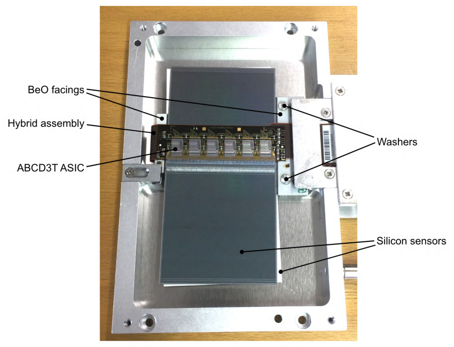
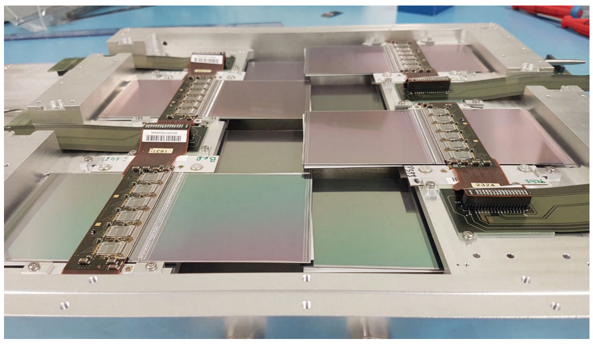


Figure 2.4: An SCT module with its ASIC readout board. [8]


(a) A tracker plane with the eight SCT modules (four per side). [8]

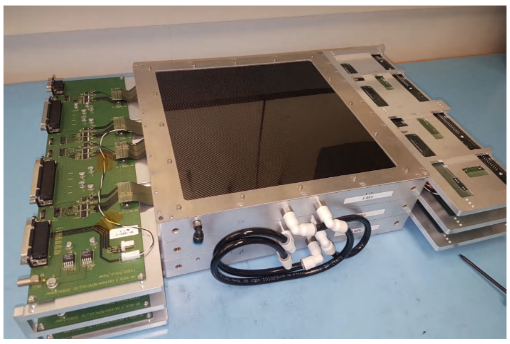

(b) A tracker station, made of three tracker planes. [8]

Figure 2.5: Design of a tracking station.

2.4 | Calorimeter & Scintillator system

The scintillator system consists of four stations. The first station, located in front of FASERν, is made up of two scintillator counters. The second station is located in front of the decay volume, and contains four scintillators. A 10 cm block of lead between the four scintillators prevent muon bremsstrahlung before the detector from interfering with the rest of the experiment. It will stop the photons, or generate a shower that can be detected by the second two scintillators. At the time of the later installation of FASERν, one of the two front veto scintillators replaced a layer of the veto station, leaving it unused. It was only with the addition of the second digitizer that the 4th layer of the veto station was reused. These two stations are used to veto charged particles entering the detector, since the events of interest are mainly those produced by particles coming from the decay volume. The timing scintillator station consists of two counter scintillators, and unlike the previous two stations, each scintillator is read by two PMTs. This station, located just after the decay volume, enables precise measurement of the timing of events and triggers on charged particles.

The last station, the preshower, is used to differentiate between photon and neutrino interaction events in the calorimeter, thanks to tungsten plates located between and around the two scintillators.

To avoid the risk of backsplash (signal collected by the preshower station and caused by particles interacting with the calorimeter), graphite blocks are also present between the scintillators of the preshower.

The calorimeter is made up of four ECAL modules, spares from LHCb experiment's outer electromagnetic calorimeter. Each module consists of 66 layers of lead and plastic scintillators, as well as wavelength-shifting fibers. Until January 2024, the module

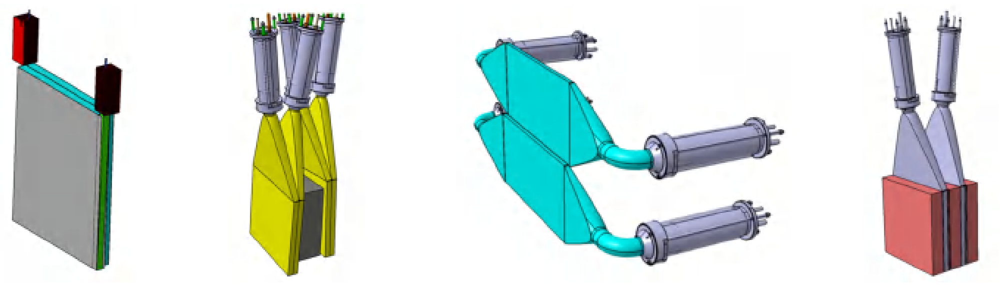
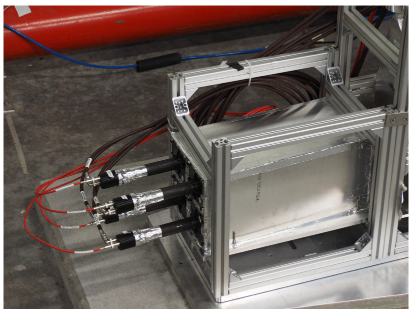
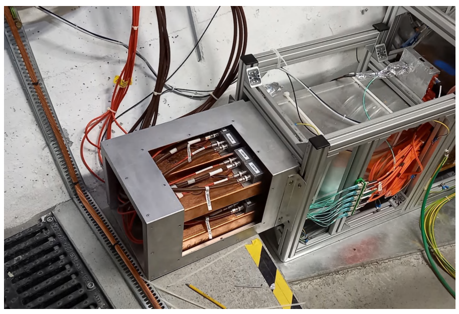


Figure 2.6: The different scintillator geometries. From left to right : the first station, the second station, the timing station and the preshower station. [8]

was read by a single PMT, limiting the operating range of energy, and making calibration more difficult (see section 4.2). Since the restart of data taking after the Year-End Technical Stop (YETS) 2023-24, each module is followed by a light splitter at the end of which are two PMTs, increasing the energy range of the calorimeter (see section 4.2 and Figure 2.7). The 4 (then 8) PMTs are enclosed by a metal cage (RF shield), used to reduce noise in the PMTs, due to a GSM antenna installed a few meters from the the calorimeter (the location is shown in Figure 2.8 and the impact of the GSM antenna is discussed in section 4.5.2).



(a) Old : single readout, four PMTs. [8]



(b) New : dual readout, eight PMTs. *Picture: Brian Petersen*

Figure 2.7: FASER Calorimeter

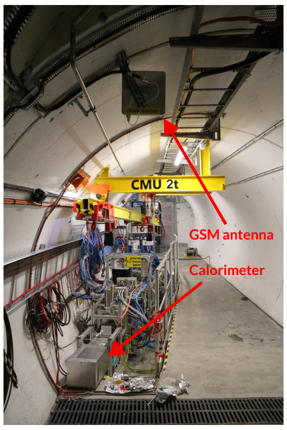


Figure 2.8: Photo of FASER, showing the calorimeter with the GSM antenna above it.

2.5 | FASERν

FASERν (FASERnu) [3] is an emulsion detector, measuring approximately 25x30 cm^2 (shown next to FASER in Figure 2.9) and weighing 1.1 tonnes. The detector's large mass is due to 770 tungsten plates, each 1mm thick, which in total is equivalent to 220 radiation lengths.

The high density of the tungsten allows neutrinos arriving in the detector to interact with the material, while emulsion films between each tungsten plate mark the trajectory of charged particles. The high resolution of this type of detector makes it possible to have more than 10^6 tracks/cm^2. On the other hand, as a passive detector, the emulsion films have to be extracted manually in order to be scanned before the number of trajectories is too large to distinguish them. Only once they have been scanned can the information provided be included in the offline analysis process. With current luminosity at the FASER location, emulsion films need to be replaced every three months on average, only during LHC technical stops.

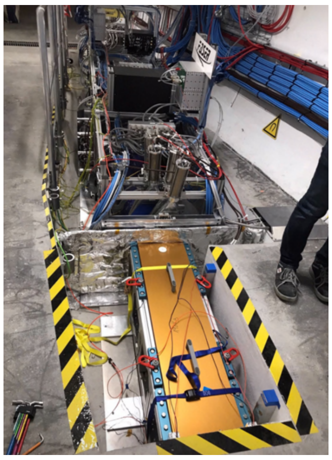


Figure 2.9: Picture of FASERν (without the cover), installed in the trench, in front of FASER. The picture points in the opposite direction to ATLAS. [8]

The high precision of FASERν makes it possible to retrieve many characteristics, such as the interaction vertex of neutrinos, or the energy of particles interacting with the emulsion and tungsten.

The active part of emulsion films is made up of silver bromide crystals of 200 nm (shown in Figure 2.10). These crystals interact with the charged particles, forming agglomerates of around 2 μm long, which allow the reconstruction of particle trajectories.

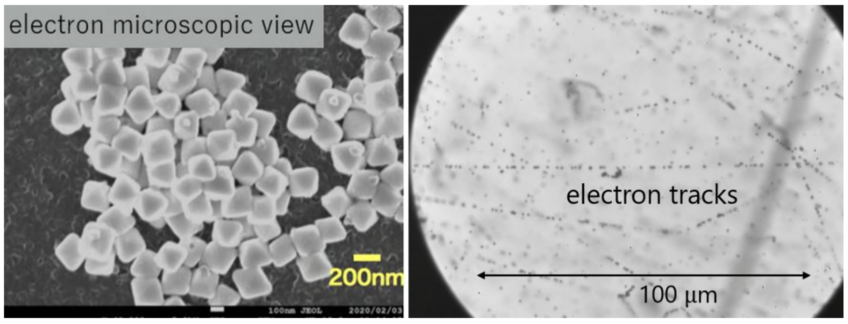


Figure 2.10: Left : microscopic view of silver bromide crystals with scale for comparison. Right : Tracks from β-rays into the emulsion film. [8]

3

The Trigger and Data Acquisition System

3.1 | Overview

The TDAQ system of FASER (represented as a simplified diagram in Figure 3.1) is a relatively simple yet robust system designed to detect extremely rare processes. To minimize maintenance as much as possible, given that the tunnel is inaccessible during data taking, FASER does not have a control room with 24/7 shifts. Therefore, the monitoring of the experiment and alerts play a major role, relying on existing tools like Grafana or custom-made interfaces such as the Online Histogram Monitoring and the Run Control. Some actions are also automated, such as the automatic restart of the daq system during a new LHC fill, which is when new protons bunches are injected in the LHC.

When a high-energy particle passes through the detector, signals from the scintillators and calorimeter modules are digitized by two commercial digitizers (CAEN vx1730s, more information in chapter 4), and an initial trigger logic is applied, based on preset thresholds. The triggers are then sent to the central part of the DAQ system: the Trigger Logic Board (TLB). The TLB is a custom FPGA-based board that, from all its inputs, can decide to issue a global trigger Level 1 Accept (L1A). This signal is then received by the digitizers and the Tracker Logic Boards (TRBs), which enable data readout from the sub-components. The read data is packaged into data fragments, which are then transferred via a fiber optic network to the DAQ PC at the surface, where the data fragments can then be assembled to a full event by the event builder and written on disk by the file writer.

The DAQ software is built on a lightweight and open-source C++ framework, DAQling [11], developed at CERN by the Detector Technologies and Infrastructure group. DAQling consider DAQ processes as modules, responsible for specific tasks, such as data readout, monitoring or compression. The DAQ software is described in section 3.3.

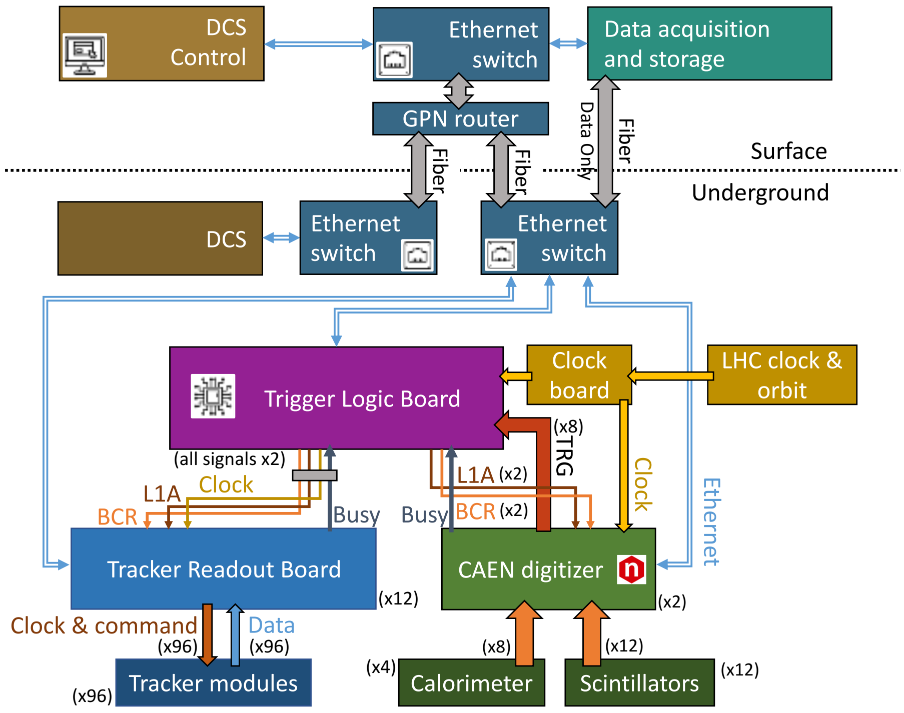


Figure 3.1: Diagram of the FASER TDAQ architecture, where the arrows represents the different types of connections between the TDAQ components. The sub-detector modules are read by the readout boards (TRB and Digitizers). The TLB decides if the trigger signals generated by the digitizers match its preloaded logic. If so, it activates the readout of the boards via Ethernet. At the surface, each DAQ readout processes reads the data from the sub-detector systems and sends the generated readout fragment to the event builder, which build the full event. The file writer records the event to file. [10] (*updated*)

3.2 | TDAQ Hardware

The clock system

The electronic boards in the TDAQ system use the LHC clock and the orbit signal, respectively 40.08 MHz and 11.245 kHz as reference clock. The clock system is described in Figure 3.2. Both signals are transmitted via the Timing, Trigger and Control system (TTC) and received by the BST Receiver Interface for Beam Observation System (BOBR), a VME module developed by the LHC Beam Instrumentation Group. It decodes and converts the acquired signals to electric signals which can then be used via LEMO connectors on the front panel. However, the BOBR clock has jitter due to the power modules, changes during the LHC energy ramp, and possible discontinuities when there is no beam in the LHC. This is problematic for the FASER electronics boards, as they require a clean and stable reference clock with a constant phase with respect to the LHC clock for precise timing.

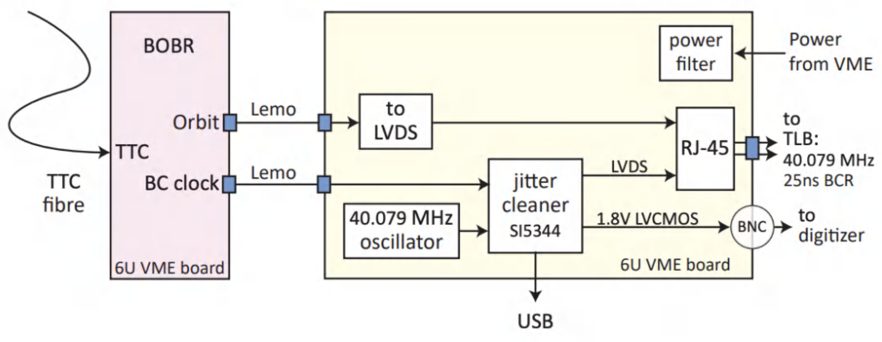


Figure 3.2: Diagram of the FASER Clock and BOBR boards. The Beam Synchronous Timing (BST) signal is transmitted via the TTC fibre to the BOBR. It decodes the input signals and convert them to electric signals, transmitted to the FASER Clock Board to clean the signal using an off-the-shelf clock cleaner board. The cleaned clock signal can then be used for the other electronics boards. [10]

A custom board has therefore been designed for FASER, the FASER Clock Board (FClock), to clean the jitter. The FClock consists of an off-the-shelf clock cleaner board [12] and a VME adapter board, which provides a cleaned signal with zero delay relative to the LHC clock, ensuring phase consistency across power cycles, even with frequency changes during the LHC cycle. The cleaned clock is then sent to the TLB as LVDS signals and to one of the two digitizers as single-ended LVCMOS.

The digitizer boards

Two commercial digitizer boards (16 channels, 14-bit CAEN vx1730s [13], shown in Figure 3.3) are used to continuously digitize the signals from the calorimeter and the scintillators at a rate of 500 MHz. A channel-specific trigger occurs if the signal is under/over a configurable threshold. Each channel trigger signals are then combined in pairs using a configurable logic. The trigger signals are finally sent to the TLB as LVDS signals.

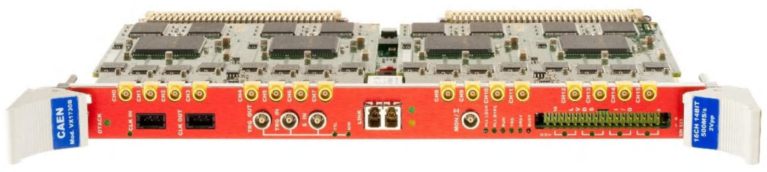


Figure 3.3: The commercial vx1730s digitizer from CAEN. The board is used in a VME crate. [13]

Upon receiving an L1A trigger from the TLB, the digitizer boards transfer samples into a buffer, including all enabled channels and a data header with additional event

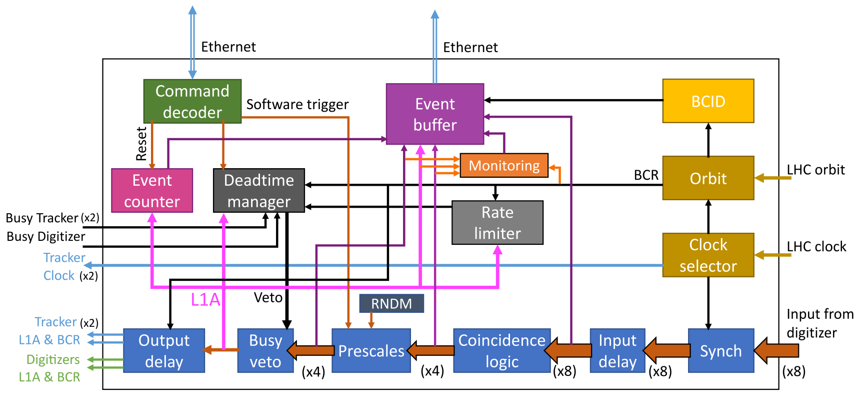


Figure 3.4: Diagram of the functionalities of the Trigger Logic Board. [10]

information, and then read out by the readout system. The digitizers are controlled via Ethernet using a Struck SIS3153 VME interface board [14]. To measure signal arrival times precisely, a copy of the LHC clock from the FClock card is sampled on one digitizer channel, allowing precise measurements of arrival time of the signals with respect to collisions occurring in IP1.

The digitizers are discussed in more detail in chapter 4.

The Trigger Logic Board

The TLB is the central trigger logic processor of the DAQ system. It is based on a General Purpose I/O (GPIO) board, developed by the University of Geneva. Its different functionalities are described in the diagram of Figure 3.4.

The TLB receives and combines up to 8 trigger input signals from the digitizer through LVDS signals along with the LHC clock and LHC orbit signals from the FClock (right-hand-side of Figure 3.4). An additional input trigger comes from the LED Calibration System through a TTL signal. It synchronizes these trigger signals with the LHC clock, applying input delays and finally send the final trigger decision via the L1A signal to the digitizers and the TRBs. The TLB also outputs a BCR (Bunch Counting Reset) signal, generated on every LHC orbit signal to the readout boards (digitizers and tracker readout boards).The Bunch Counter ID (BCID), corresponding to the number of clock cycles between the last BCR and the trigger signal is also computed by the TLB.

The 8 trigger inputs are combined into 4 physics trigger items via a Look-Up Table (LUT), with a configurable coincidence logic. The TLB can also generate an internal trigger in three different modes: fixed rate, pseudo-random rate, and software command. Each trigger source can be "prescaled" by a factor n (only every nth signal is kept). Upon an L1A, the TLB sends a data packet containing information about the event to the DAQ system. At a configurable rate, the TLB can publish monitoring data packets containing metrics before / after prescale and veto.

To manage trigger vetoes and minimize system deadtime, The TLB implements several sources of trigger vetoes:

- Tracker busy veto: Triggered when the tracker readout boards are busy reading out event data.
- Digitizer busy veto: Triggered when the digitizer's internal readout buffer reaches a certain threshold.
- Simple deadtime veto: Prevents triggers within a fixed number of bunch-crossings after an L1A.
- BCR veto: Prevents overlap between L1A and BCR processing by vetoing triggers around the BCR signal.
- Rate limiter: Limits the L1A rate to prevent uncontrolled high rates from noisy input channel.

The Tracker Readout Board

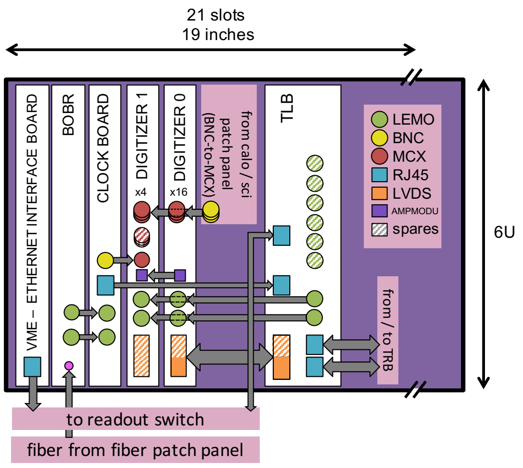
The Tracker Logic Board (TRB) operates and reads out data from the Silicon Strip (SCT) modules. Like the TLB, the TRB is based on the GPIO board from the University of Geneva. They are housed in custom mini-crates on the detector. One TRB can read out eight SCT modules, which necessitates a total of 9 TRBs, plus an additional 3 TRBs for the IFT.

When commands are sent from the host PC to the tracker modules, the TRB converts them into bit-stream encoded data, which is understandable by the tracker modules.
In standalone mode, the TRB can generate internal triggers for calibration scans, operating on its internal 40 MHz clock. When integrated into the full system, it synchronizes with the TLB clock.

Location of the boards

Nine TRBs are housed in a dedicated mini crate positioned above the middle tracking station while 3 TRB are housed in another mini crate, above the IFT station. Each TRB receives 24 V power and includes a temperature sensor. The TLB and the digitizer are

(a) Photo of the VME crate. From left to right : Crate controller, digitizer 0, TLB, digitizer 1, FClock, BOBR.



(b) Schematic view of the VME crate with the connections. The layout of the boards has been optimized to simplify the connections display. [10] (*updated*)

Figure 3.5: Disposition of the electronic boards in the VME crate, located next to the detector.

located in a VME crate located several meters away from the detector, but still in TI12. A photo and a schematic view of the VME crate are shown in Figures 3.5(a) and 3.5(b) .

3.3 | TDAQ Software

3.3.1 | DAQ software framework

The FASER DAQ software [15] (represented in a simplified diagram in Figure 3.6) is built on top of DAQling [11], an open-source C++ data acquisition framework suited for small and medium experiments. DAQling is built around modules, which can communicate with each other via the ZeroMQ messaging library [16]. DAQling also provides tools such as ERS [17] for error logging or monitoring via Redis [18] and InfluxDB [19]. The configuration of the modules is done via JSON, with the possibility of using tools such as JSON Schema [20] for modularity and validation of the configuration files. DAQling allows managing the different processes/modules registered in the configuration files through commands, some of which are available by default and others that can be added.

Each readout electronics (TRB, TLB, Digitizers) board in FASER is managed by a dedicated software process, packaged as a DAQling module (called a *receiver*). They are responsible for controlling, reading, and packing the readout data into a sub-detector fragment. Each sub-detector has a specific ID, allowing identification of the origin and the event to which the fragment belongs. The data fragment is then sent to the event builder module, which is responsible for reconstructing the entire event from the fragments of all the sub-components. The event is then sent to the file writer module to be recorded to file. To avoid a high rate of I/O file operations, the file writer stores the events in a buffer, which, if it exceeds a certain threshold, will write the buffer to the file. The event is also sent to all the monitoring modules, which can perform monitoring on the full event and populate histograms. At regular intervals, a module (histogram archiver) takes a snapshot of the histograms and saves them in a file. There are several streams to which the events can be sent. When reconstructing the event, the event builder also performs quality checks. It checks that the BCID number of each fragment is the same, and that the fragments are not flagged as corrupted. If fragments do not arrive at the event builder within a configurable time, the event is also flagged and directed to a dedicated stream. A data compression module, developed by FASER, allows compressing events from the physics stream before they reach the file writer, reducing the event size by half. Events from other streams are not compressed.

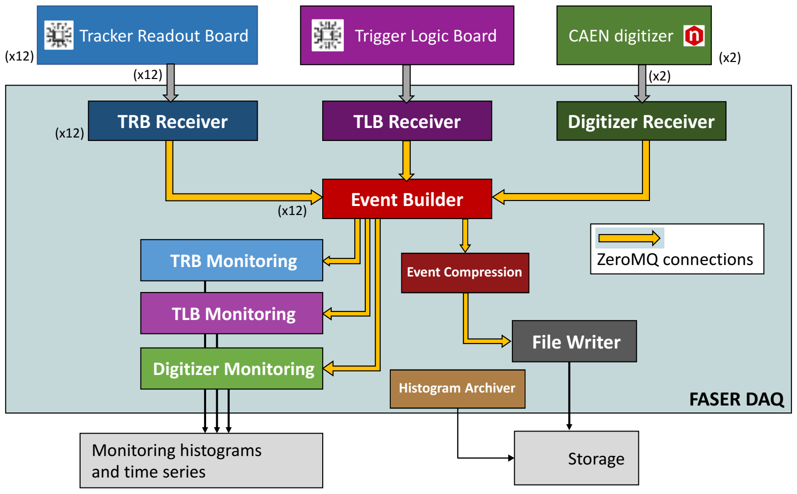


Figure 3.6: Simplified diagram of the DAQ processes used for FASER. The receiver modules handles readout and event fragment creation, while monitoring modules have access to the entire event to publish monitoring metrics. [10] (*updated*)

3.3.2 | Event format

The structure of the event format is composed of a header, a body, and an optional trailer. The header contains event identification information such as data size, event counter, event identifier number, and timestamp and is provided by the event builder. The body contains the sub-detector fragments, such as the TLB, digitizers, TRBs, and the event builder. These data fragments are structured the same way as the full event: a header and a body, respectively the Readout Receiver (ROR) Event Header and the Readout (RO) fragment. An optional trailer is also possible. The ROR Event Header contains the fragment origin and eventual flags caused by hardware issues. The RO fragment contains the raw readout data coming from the different sub-detectors and thus is specific to the sub-detector electronics. A diagram of the event structure is shown in Figure 3.7.

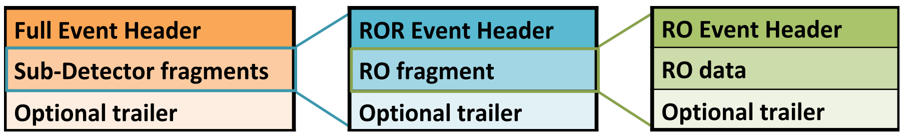


Figure 3.7: Structure of the event format. The sub-detector fragment has the same structure for all sub-detector components. [10]

3.3.3 | Control and Monitoring Tools

Since FASER does not have a Control Room, several monitoring and control tools have been put in place to ensure the stability and quality of data acquisition. Some come directly from DAQling, while others have been specifically implemented for FASER, described in the following sections.

Run Control

DAQling allows representing the DAQ system through a "control tree" where the different nodes correspond to the modules specified in the configuration file. The nodes can each be controlled independently according to an advanced Finite State Machine. Commands executed on a node are propagated to all child nodes. It is possible to *exclude* certain nodes so that they do not contribute to the status of the parent node and do not receive its commands. The different possible states of the nodes are represented in Figure 3.8.

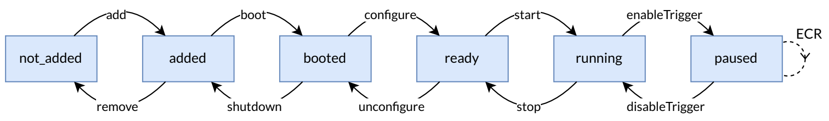


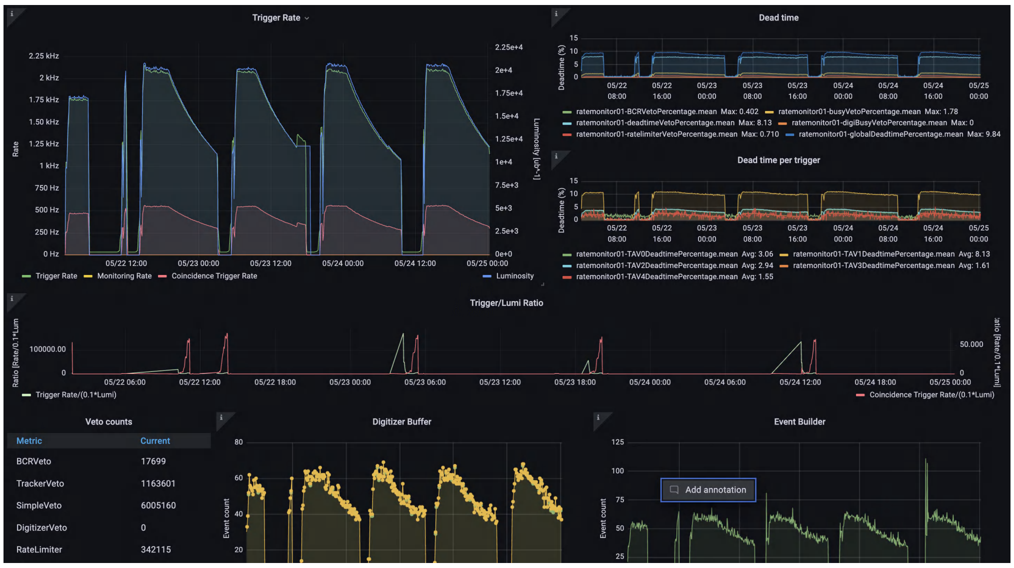
Figure 3.8: FSM states with their respective commands.

To improve the interaction between the DAQ system and a user, a web interface is available, allowing visualization of the DAQ system and execution of commands on the various nodes (processes). When starting data taking, the user specifies the type and a comment associated with the run, with the number provided by an application (Run Service). Upon stopping, the user can add an end comment. The type, comment, and other run-related information are sent to the Run Service, which stores them in an Oracle database. The interface also allows easy log consultation.
The Run Control is detailed in Chapter 5.

Monitoring

Monitoring of the DAQ system is done via time-series metrics, published by every module, and via data quality histograms published by monitoring modules.

Metrics are directly handled by DAQling and are published to InfluxDB and a Redis database, which can then be viewed in Grafana (see Figure 3.9) and a small part on the web interface of the Run Control.


Trigger Rate
Rate
Luminosity [ub^-1]
Trigger Rate
Monitoring Rate
Coincidence Trigger Rate
Luminosity
Dead time
Deadtime (%)
ratemonitor01-BCRVetoPercentage.mean Max: 0.402
ratemonitor01-busyVetoPercentage.mean Max: 1.78
ratemonitor01-deadtimeVetoPercentage.mean Max: 8.13
ratemonitor01-digiBusyVetoPercentage.mean Max: 0
ratemonitor01-ratelimiterVetoPercentage.mean Max: 0.710
ratemonitor01-globalDeadtimePercentage.mean Max: 9.84
Dead time per trigger
ratemonitor01-TAV0DeadtimePercentage.mean Avg: 3.06
ratemonitor01-TAV1DeadtimePercentage.mean Avg: 8.13
ratemonitor01-TAV2DeadtimePercentage.mean Avg: 2.94
ratemonitor01-TAV3DeadtimePercentage.mean Avg: 1.61
ratemonitor01-TAV4DeadtimePercentage.mean Avg: 1.55
Trigger/Lumi Ratio
Ratio [Rate/0.1*Lum
Trigger Rate/(0.1*Lumi)
Coincidence Trigger Rate/(0.1*Lumi)
Digitizer Buffer
Event count
Event Builder
Add annotation

Veto counts

Metric	Current
BCRVeto	17699
TrackerVeto	1163601
SimpleVeto	6005160
DigitizerVeto	0
RateLimiter	342115

Figure 3.9: Example of a Grafana dashboard. Global information such as the trigger rate are displayed (top-left). Information specific to sub-detector (digitizer) and software (event builder) components are also visible.

The histograms are populated by DAQ processes which can do monitoring on full events. The histograms are published to the Redis database and then viewed in a custom web interface. The interface allows to display live histograms, perform some small checks, and flag problematic histograms. Comparisons between several snapshots of the same histogram are also possible. An example of histograms displayed on the online histogram web app is shown on Figure 3.9.

Histograms saved in a file by the histogram archiver module can be viewed via a web application similar to the online histogram viewer,

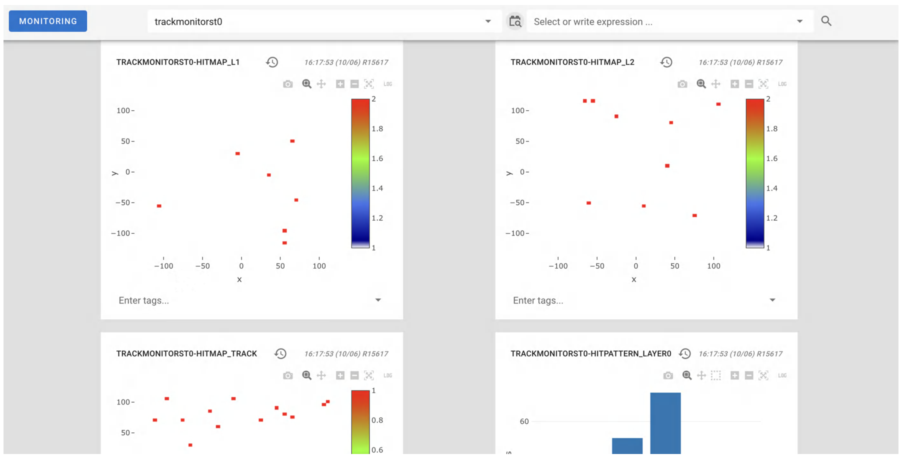

MONITORING
trackmonitorst0
Select or write expression ...
TRACKMONITORST0-HITMAP_L1
TRACKMONITORST0-HITMAP_L2
TRACKMONITORST0-HITMAP_TRACK
TRACKMONITORST0-HITPATTERN_LAYER0
16:17:53 (10/06) R15617
Enter tags...

Figure 3.10: Interface of the histogram monitoring web app, where 1D and 2D histograms can be displayed.

The Detector Control System

The Detector Control System (DCS) uses a Supervisory Control and Data Acquisition system [21] (SCADA) employed by LHC experiments at CERN. It controls, configures, and monitors the operational states of the FASER detector. This includes environmental parameters such as the temperature and humidity of the tracker, thanks to a custom Tracker Interlock and Monitoring (TIM) board, which also serves as a hardware safety interlock for the tracker. The DCS also controls the high and low voltage systems and the Power Distribution Units. Checks on the monitored values allow action in case of problems, through alerts, or automatic procedures. The DCS also publishes its monitored values on Grafana.

4

Upgrade to calorimeter split readout

4.1 | Motivation

Until the end of 2023, the calorimeter consisted of just four modules, in a 2 x 2 configuration and the major drawback of this configuration was the energy range coverage.

For dark photon-like events, we expect high energy signals up to 3 TeV, which force us to run the Photomultiplier Tubes (PMTs) at low gain. An optical filter reduces the signal amplitude by a factor of 10 to avoid having to run the PMTs at very low gain, where their response is non-linear.

On the other hand, the only source of particles that can practically be used for calibration of the calorimeter response is the high rate of muons: minimum ionizing particles depositing an energy of around 0.3 GeV in the calorimeters. At low gain, the signals from these muons are almost completely hidden in the noise, even with a good digitizer (RMS noise is about 3-4 ADC counts, with 1 ADC count = 0.12 mV with the current physics run settings).

Before 2024, this problem was overcome by dedicating a period of data taking to collecting collision muon events at high gain. The calibration measurements were then extrapolated to the low gain region. The accuracy of the extrapolation was controlled by a calibration LED system: an LED emits short signals while scanning different PMT input voltages.

Two modes were therefore required: low gain for physics data taking, and high gain for MIP calibration.

Although the method was effective, almost a third of the total data collected was spent on calibration, which reduced the dataset for the physics analysis.

4.2 | Proposed solution

The solution considered was, for each calorimeter module, to divide the light in two paths and have two PMTs instead of a single one. This is done by using a fibre bundle where the light split would follow a 1:30 ratio (high:low), as shown in Figure 4.1. The two PMTs would then operate at mid gain (at a gain three times higher than the low gain values), eliminating the need for the filter and gain calibration.

One light path would then target the high-energy range (low gain PMT), and the other the low-energy range (high gain PMT), specifically, covering the following energy range :

- Low energy range : 0.1 - 100 GeV
- High energy range : 3 - 3000 GeV

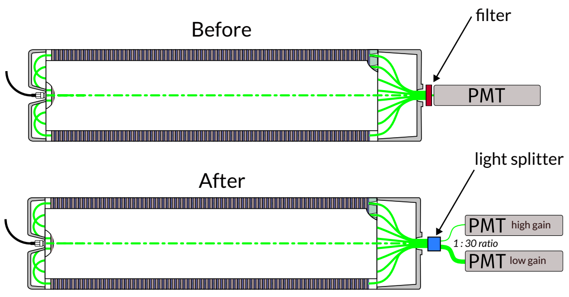


Figure 4.1: Setup before and after adding a light splitter.
Before : the light is first attenuated by a filter before being collected by the PMT.
After : the filter is removed in favor of a light splitter that divides the incoming light. It is then split at a ratio of 1:30. The small fraction is collected by the high-gain PMT for low-energy signals, and the large fraction is collected by the low-gain PMT for high-energy signals. [8] (*updated*)

The overlap between the two PMTs is intentional, as it allows cross-calibrations to be performed.

With four additional PMTs came also four additional sources to read from, but since all 16 digitizer channels were already used, a second digitizer was needed.

At the time of FASER installation, FASERν was not approved yet, so when FASERnu was installed in the tunnel, some adjustments were needed: a veto layer had to be disconnected to use the one of the front veto layer, in front of FASERnu, as both acts as a veto layer. With the additional digitizer, it would allow to connect the four additional PMT signals and the previously disconnected veto layer. The remaining channels could be used in the future for other sources.

The new architecture, which brings new possibilities to the DAQ, also comes with expected challenges:

1. The occupancy of both digitizer's buffer should be the same. If it is not the case, the buffer of one of the digitizers could be filled to the maximum, without being

the case in the second digitizer, which would induce a `BUSY` signal, blocking acquisition in one of the digitizers. This can lead to synchronisation problems when reconstructing events and a potential loss of important events.

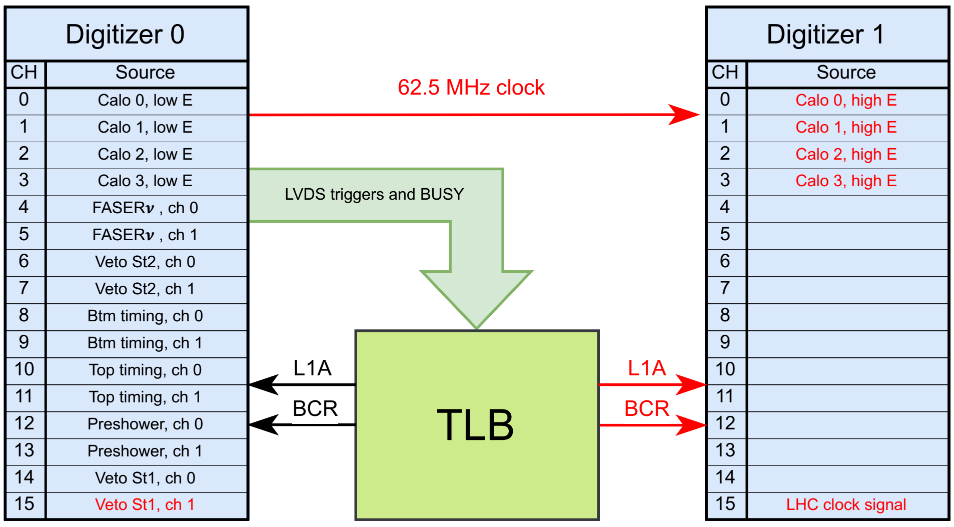
2. The digitizers need to be synchronized with each other.

3. The maximum trigger rate would be lower, as more data is read out and processed.

The digitizers must therefore be read out in "parallel", i.e. instead of first reading the entire buffer of the first digitizer before moving on to the second digitizer, the readout will alternate between each digitizer after only reading N events, where N is configurable (generally corresponding to the chosen value of the BLT). Unfortunately, true parallel readout is not possible, which is a limitation due to the crate controller used.

The second point concerns the digitizer's internal clocks. Even with digitizers of the same model, the internal clocks are not exactly the same as small variations can happen (e.g. temperature change, slightly different nominal clock rate). Bad digitizer synchronization leads to unpredictable offsets in data acquisition, making offline processing much more complicated. Wrong synchronization can be minimized by using the `CLK IN` connector, designed to provide an external clock as a reference. The last point is unavoidable, but can be minimized by adjusting readout parameters.

The proposed new mapping is described in Figure 4.2.



Digitizer 0

CH	Source
0	Calo 0, low E
1	Calo 1, low E
2	Calo 2, low E
3	Calo 3, low E
4	FASERν , ch 0
5	FASERν , ch 1
6	Veto St2, ch 0
7	Veto St2, ch 1
8	Btm timing, ch 0
9	Btm timing, ch 1
10	Top timing, ch 0
11	Top timing, ch 1
12	Preshower, ch 0
13	Preshower, ch 1
14	Veto St1, ch 0
15	Veto St1, ch 1

Digitizer 1

CH	Source
0	Calo 0, high E
1	Calo 1, high E
2	Calo 2, high E
3	Calo 3, high E
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	LHC clock signal

Figure 4.2: Mapping of the different scintillators and calorimeter modules to the 16 + new 16 digitizer channels. Red labels indicate new / changed elements.

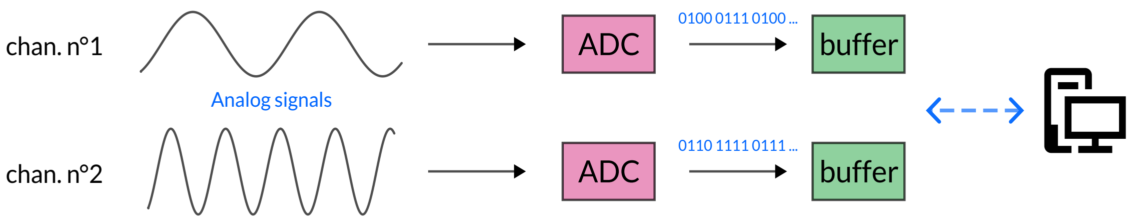


Figure 4.3: The analog signals are first digitized via the ADC, then stored temporarily, waiting to be read out by a computer. Additional channels allows multiple analog signals to be processed (in this case, two channels).

The 62.5 MHz external clock frequency has been decided, based on existing configuration files provided by the CAEN manufacturer.

The channel used by the clock, because of its high activity (continuously alternating signal, unlike the other channels, where the signals are pulses), is isolated in the last channel of the second digitizer so as not to create noise in the adjacent channels.

Reading the second digitizer also results in a new sub-detector-fragment in the global event. This sub-fragment is identified by the SourceID associated to the digitizers and the digitizer number (0 or 1). The new fragment and the five new channels increases the size of an event from 12.8 kB to 15 kB.

This increase also has an impact on readout speed. Thus, the BLT is increased from 1 to 4. This divides the number of calls to the digitizers by 4 for the same number of events, compensating for part of the slowdown. A higher number of BLTs is possible, but also more risky, since if a network packet is dropped, all the events linked to the same BLT will be lost.

4.3 | The digitizer

A digitizer is an electronic acquisition device which converts analog signals into digital data using Analog-To-Digitial Converters (ADCs). The digitized signals are then stored in a buffer, waiting to be read by an external computer. Multiple signals can be digitized in parallel, via different channels, as depicted in Figure 4.3. Communicating with the digitizer can be done via several interfaces, such as the Peripheral Component Interconnect (PCI) or the VersaModules Eurocard bus (VMEbus). For firmware updates or PLL reconfiguration, an optical link interface is used (`LINK` input in Figure 4.4)

A digitizer has several important characteristics. Firstly, the resolution of the ADC is determined by the number of bits used to describe the value of the analog signal. The dynamic range represents the maximum and minimum signal voltage that the digitizer can measure in a single acquisition. A resolution of 14 bits for a dynamic range of

2 Volts gives, for example, $2^{14} - 1 = 16\,384$ signal levels. A single level therefore is $2/16\,384 \sim 0.12$ mV, which correspond to the minimum value the digitizer can convert. A smaller dynamic range will convert a smaller signal amplitude, but will be limited to smaller input signals.

The sampling rate determines the frequency at which the ADC can convert signals and therefore influences the temporal resolution of the digitized data. For example, a sampling rate of 500 MHz corresponds to one digitization (one sample) every 2 ns.

4.3.1 | The vx1730s digitizer

The vx1730s is a commercial digitizer card, manufactured by CAEN [13], and is highly configurable via writes to specific memory registers. It has a 14 bits ADC resolution with a selectable dynamic range of 2 V_{pp} / 0.5 V_{pp} and a sampling rate of 500 MS/s (500 000 000 samples per seconds). The vx1730s differs from its predecessor, the 1730 (used as the first digitizer in the experiment) in not needing to be manually calibrated on start-up, as this is done automatically.

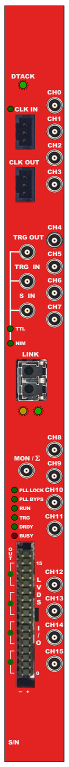


Figure 4.4: Front view of the digitizer.

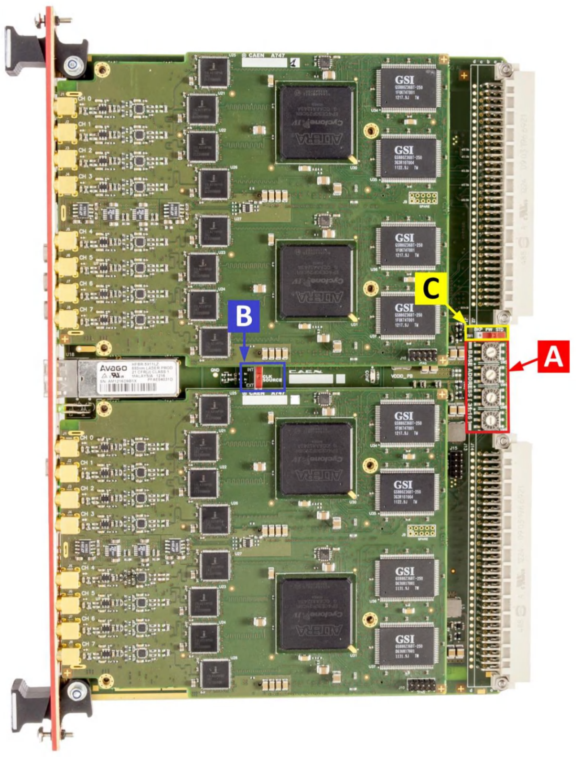


Figure 4.5: Side view of the digitizer. [22]

All I/O connectors are located on the front of the board (except for the connection

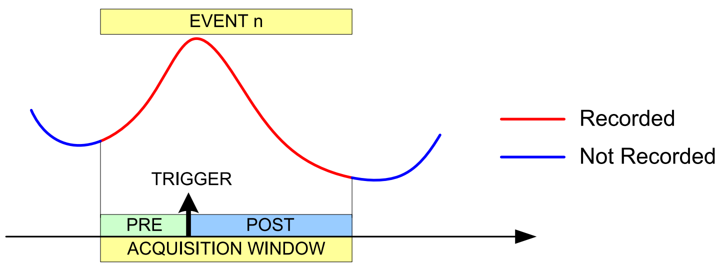


Figure 4.6: After the trigger signal, the samples included in the acquisition window are then saved in a channel-specific buffer, awaiting readout. [22]

to the VMEbus back plane), so that the digitizer can be placed in a VME crate alongside other modules.

The base address (A), clock source (B) and flash page (C) can be configured on the right-hand side (Figure 4.5).

Input analog signals can be connected to input MCX connectors via LEMO cables, labeled from `CH0` to `CH15`. During data acquisition, analog signals are digitized and temporarily stored in a circular memory buffer.

Trigger

With the vx1730s, a trigger signal can be delivered in several ways. A common external trigger (for all channels) can be given via the `TRG IN` TTL input or the LVDS connectors. A software command can also produce a trigger signal. Finally, a channel can self-trigger if the digitized signal exceeds a configurable threshold. The trigger signals are combined in groups of two channels (eight groups in total) with a programmable logic (AND, OR, XOR). More information can be found in the official documentation [22].

If an acquisition trigger signal is received, the digitizer saves the current samples inside a configurable acquisition window. The acquisition window can contain samples coming before the trigger signal (pre samples) and after the signal (post samples), as illustrated in Figure 4.6.

The width of the acquisition window depends directly on the configuration of the SRAM memory, which is independent for each channel. Each memory SRAM is capable of storing 640 000 (-10 S) samples. It is then possible to organize this memory into several buffers, with one buffer corresponding to one event. The memory can be divided into a maximum of 1024 buffers, i.e. 1024 events, each with $640\text{kS}/N_b = 640\ (-10)\ \text{S} = 630\ S$, with $N_b = 1024$, the number of buffers. Once the 1024 buffers have been filled, the memory will be marked as `FULL` and no further triggers will be accepted. Table 4.1 lists the different buffer configurations with the corresponding number of samples.

Buffer Number (N_b)	Number of samples (N_{sample})
1	640 kS *(-10 S)*
2	320 kS *(-10 S)*
4	160 kS *(-10 S)*
8	80 kS *(-10 S)*
16	40 kS *(-10 S)*
32	20 kS *(-10 S)*
64	10 kS *(-10 S)*
128	5 kS *(-10 S)*
256	2.5 kS *(-10 S)*
512	1.25 kS *(-10 S)*
1024	640 S *(-10 S)*

Table 4.1: Buffer organisation. A higher number of buffer allows more events to be stored, but less samples per event. 10 samples are removed due to internal logic in the digitizer.

A custom number of samples, smaller than the buffer size, can be configured by choosing an N_LOC value so that is formula is respected :

$$N_{\text{sample}} = 10 \cdot \text{N\_LOC}, \text{ with N\_LOC the configurable parameter}$$

However, this doesn't change the number of buffers.

Clock distribution

The clock distribution happens on two domains : `OSC-CLK` and `REF-CLK`. Optical link, USB and Local Bus communication are handled by the `OSC-CLK`, which is fixed at 50 MHz. `REF-CLK` can be either external (via the `CLK-IN` input) or internal, via the 50 MHz oscillator, and will enable the clock distribution device and the Phase Loop Lock (PLL) to generate a ADC sampling clock of 500 MHz and the trigger logic synchronization clock (TRG-CLK) of 250 MHz.

`REF-CLK` source can be toggled via the dip switch B (`EXT / INT`) on Figure 4.5. The AD9510 must be reconfigured if an external PLL reference clock is supplied with a frequency other than the default 50 MHz.

Event Structure

A stored event in the channel buffer is made of 32-bit words, where each bit / sequence of bits is associated with a specific information of the event. It has always the same following structure :

- **A Header** : 4 x 32-bits words where information such as the event size (the total number of words in the event), the channel mask (which channels are activated) and the Trigger Time Tag (TTT) are stored.
- **The Data** : a variable number of 32-bits containing the digitized signals of the activated channels.

The TTT provides a time reference to the trigger which increments at a frequency of 125 MHz (i.e. 4 ADC clock cycles) from the start of acquisition and ends at the stop, or at an `S IN` / LVDS signal and is read at half this frequency, i.e. 62.5 MHz (16 ns). When the counter reaches its maximum, $(2^{31} - 1)$, a rollover occurs and the 32nd bit is set to 1 (during the first rollover), and the TTT continues to increment up to 2^{32} before rolling over again at $2^{31} - 1$ (see Figure 4.7).

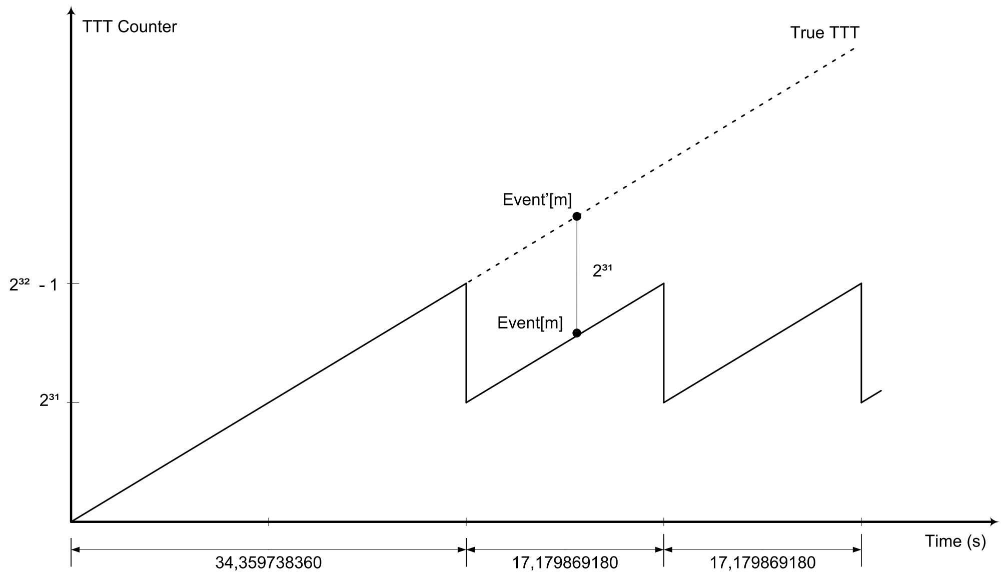


Figure 4.7: TTT Counter time evolution. *Credits: CAEN*

The structure of the event is described in greater detail in Figure 4.8 and in the user manual [22].

Data Transfer

The vx1730(s) digitizer is compatible with several data transfer modes. For FASER, the 2eSST mode is used because it allows data transfers of up to 200 MB/s. Another feature related to data transfer is the Block Transfer (BLT). Block Transfer readout allows multiple events to be read per digitizer call. For example, a BLT of 2 halves the number

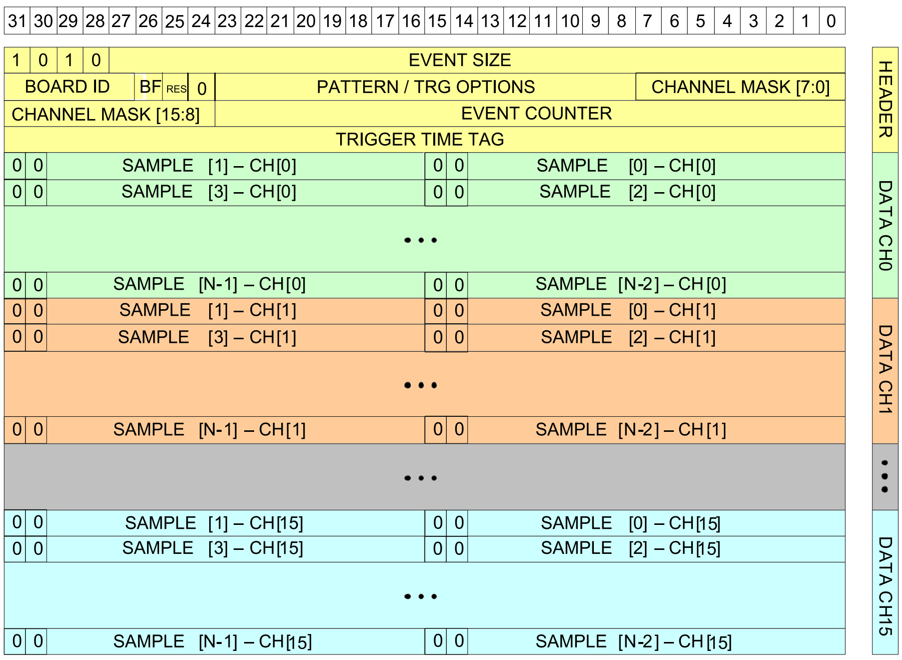


Figure 4.8: Structure of an event with all channels (16) activated. [22]

of digitizer accesses, which is an important factor in optimizing the readout system, as digitizer calls are an unavoidable bottleneck.

4.4 | Standalone tests

A test setup was constructed in a lab room at CERn in order to commission the new dual digitizer readout. It includes a *sis3153* crate controller, two digitizers and a spare TLB.

As one of the vx1730s digitizers is already used by the FASER experiment itself, the tests were performed with a spare digitizer (vx1730) and the new digitizer (vx1730s). This presents no issues, given that both digitizers have the same characteristics, except that the old one requires manual calibration, which is not the case for the **s* model.

4.4.1 | Digitizer code structure changes

Given the relatively short time available to integrate a second digitizer in the current DAQ setup (a first version was to be ready at the end of December, with installation in January during YETS), a complete restructuring of the digitizer code was not possible, given that all new implementations must be rigorously tested. However, several guidelines were taken into account during implementation:

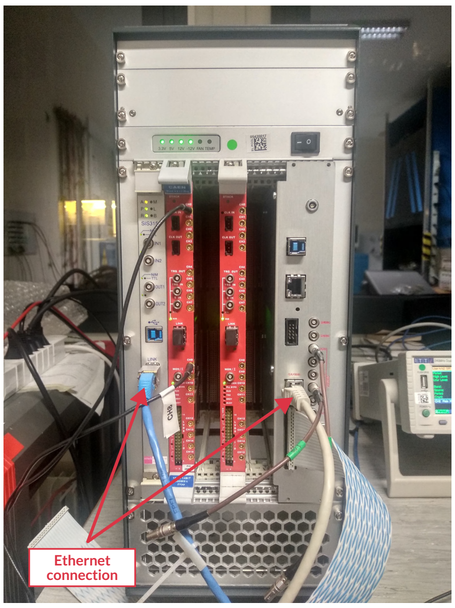


Figure 4.9: From left to right : the crate controller, the two digitizers and the TLB.

- Follow the C++ 17 standard [23] as closely as possible, without interfering with low-level code related to digitizer / crate interface communication.
- The new code implementation should not only be able to handle an additional digitizer, but should also be easy to add additional digitizers in the future (assuming they are of the same model)
- The already existing digitizers configuration files should also work on the new system.

To manage the configuration of several digitizers in the same json configuration file, a new `digitizers` field has been created. This field takes as its value an array of objects, each linked to the individual digitizer configuration. These objects contain the same fields as the previous version, except for those common to all digitizers, such as the readout parameters and the trigger mode. Those parameters have been removed from the individual objects in order to avoid redundant fields.

Since both digitizers would receive the same commands at the same time (start, stop, read data, etc.), a class was implemented to propagate actions performed on this class to all digitizers. The structure of the new digitizer codebase is illustrated in Figure 4.10 and its structure is summarized below.

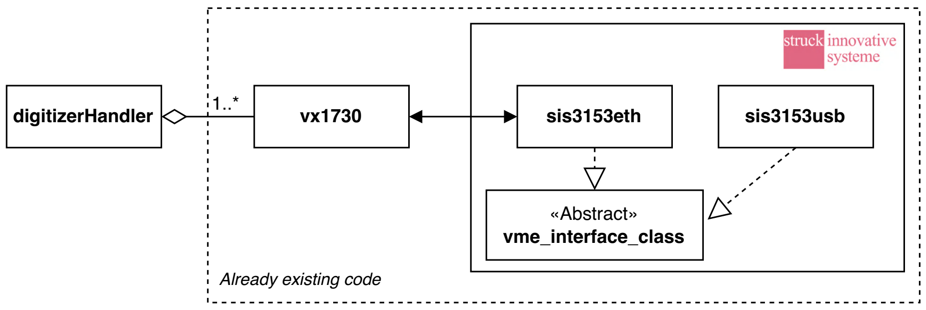


Figure 4.10: Simplified class diagram of the digitizer codebase [24]. The `vx1730` class is a wrapper around the low level `sis3153eth` class, provided with the crate controller. The `digitizerHandler` class allows to perform actions on all registered `vx1730` objects.

The code is mainly divided into two parts: the code supplied by the crate controller manufacturer, *struck innovative systeme* [25], and the code implemented specifically for FASER. The crate controller code had to be slightly modified to improve packet loss and error handling.

The abstract class `vme_interface_class` instantiates two other classes that simplify communication with the crate controller (and therefore with the digitizer). For FASER, communication is performed via Ethernet, so the `sis3153eth` class is used. The `vx1730` implements functions to link "high-level" actions, such as digitizer configuration and control, with read/write registers specific to each action via the provided `sis3153` class. The `digitizerHandler` class manages several `vx1730` objects, each corresponding to a digitizer.

To improve readability, a C++ structure has been added, `READOUT`, which holds the raw data and the monitoring values for a single readout operation. Its code implementation is provided in Code Fragment 1.

```
struct READOUT {
    unsigned int nerrors = 0; // # errors occuring in a single readout
    size_t eventSize; // expected size of the event
    size_t nwordsObtained = 0; // # received words during readout
    std::map<std::string, float> monitoring; // basic monitoring
    UINT *raw_payload; // the software buffer which contains the raw data
    ~READOUT() { // destructor
        delete[] raw_payload;
        raw_payload = nullptr;
    }
};
```

Code Fragment 1: Definition of the the `READOUT` struct.

4.4.2 | Synchronization tests

Without Synchronization

The first step is to check that digitizer synchronization is working properly, and to understand the impact of setting the external clock on the second digitizer. To achieve this, an initial test was performed with the two digitizers and a signal generator, shown on Figure 4.11. The signal generator sends short negative square-wave signals at the same time to both digitizers on one channel.

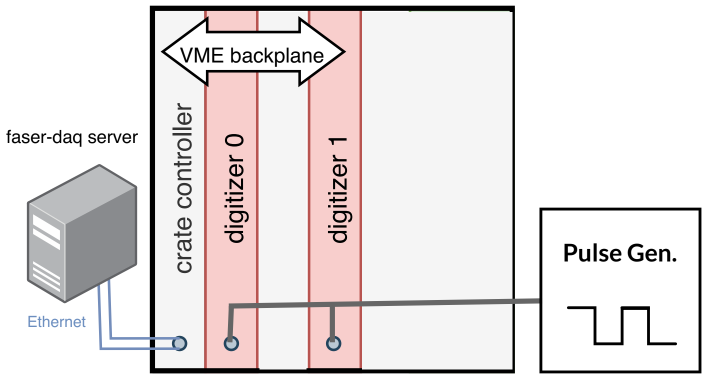


Figure 4.11: Setup with only the two digitizers and a pulse generator, sending a square signal in both digitizers. Without synchronization.

By selecting the trigger operating mode to `self-trigger`, the digitizers choose to trigger an event when the signal in a channel exceeds a defined voltage threshold.

The signals are sent simultaneously, but since both digitizers are not synchronized, the trigger should not happen at the same time, resulting with two offset signals, as shown in Figure 4.12.

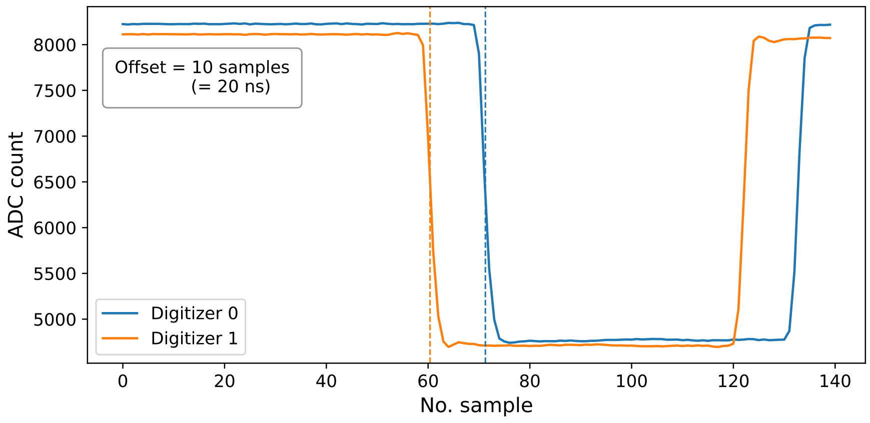


Figure 4.12: Plot of one pulse from the pulse generator, as seen by each digitizer. The offset between the two signals indicates a difference in clock rates.

To calculate the offset between the signals, we take the difference between the two half rise-time. The falling part of the signals is fitted with a logistic sigmoid function [26]. With $\sim$ 6200 events, we obtain the distribution shown in Figure 4.13. The conversion from the number of samples to time can be made by multiplying the number of sample by 2, given that a sample is taken every 2 ns (500 MHz). We can see that in the majority of cases, the offset between the two digitizers is neither zero nor constant.

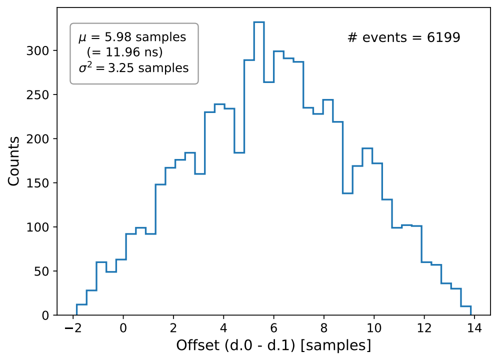


Figure 4.13: Distribution of the offsets for $\sim$ 6200 events. Two synchronized digitizers would give a much narrower distribution.

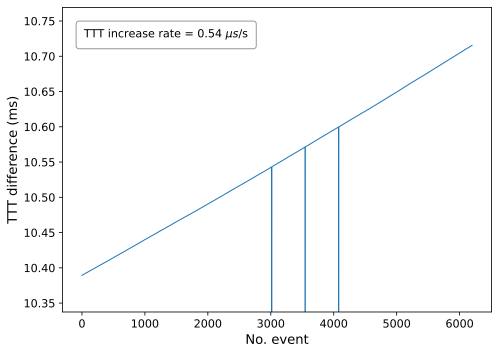


Figure 4.14: Evolution of the difference between the Trigger Time Tag for both digitizers. The rate can lead us to the clock rate mismatch of the two digitizers.

Another measure of de-synchronisation between the digitizers is to look at the Trigger Time Tag. The TTT is incremented as soon as data acquisition begins. As operations can only be sequential between the VME crate modules and the crate controller, the start of the digitizers is offset, which inevitably leads to a difference in the TTT. Since the incrementation of the TTT is managed by the digitizer's internal clock, the desynchronisation of the clocks induces a constantly increasing offset of the two TTTs, as can be seen in Figure 4.14. From the increase in the offset, we can compute the difference in frequency between the two clocks. With the rate found in Figure 4.14, the difference is 0.54 μHz, which is a small difference but enough to create misleading timing of events.

The vertical lines that can be seen on the graph in Figure 4.14 are due to the TTT rollover. When the offset between two TTTs is sufficient, one of the digitizers can perform its TTT rollover (as shown in Figure 4.7) while the other is not yet at its maximum TTT. This results in an immediate difference of slightly less than $(2^{32} - 1) - 2^{31} = 2\,147\,483\,647$ TTT values, equivalent to ~ 17 seconds.

With Synchronization

CAEN provides ready-to-use configuration files for reprogramming the PLL. Both digitizers have to be reprogrammed, since internal clocks are used by default :

- Digitizer 0 : *In* : internal (500 MHz)$\rightarrow$ *Out* : external (62.5 MHz)
- Digitizer 1 : *In* : external (62.5 MHz) $\rightarrow$ *Out* : deactivated

The clock is transmitted from one digitizer to another by a Single-ended-to-differential A318 cable.

With the same configuration, we were able to compare with and without synchronisation.

By synchronizing the two digitizers, we can see that the distribution of offsets is now limited to just two values. The synchronisation is also confirmed by no increase in the difference in TTT values, shown in Figure 4.17. On the other hand, small jumps of 2 TTT values occur regularly. These jumps, mentioned in the documentation, are unavoidable because of the limitations of the digitizers and the fact that the TTT is incremented every 8 ns and read every 16 ns. Thus, a jump of 2 TTTs corresponds to a 16 ns (8 samples) offset, which is

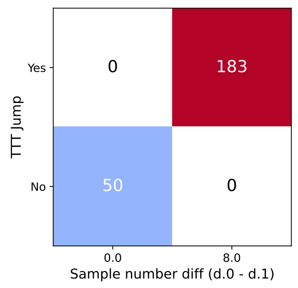


Figure 4.15: Grid of the different TTT values and the sample number offset between the two signals, as seen by the digitizers.

confirmed by the distribution described in Figure 4.16.

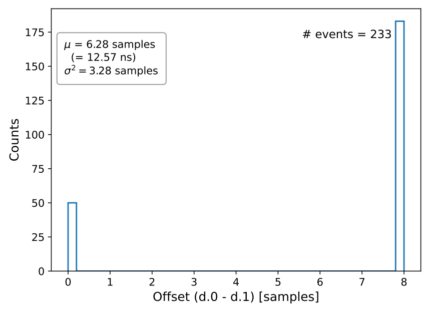


Figure 4.16: Distribution of the offsets for 233 events. The offsets are limited to only two values.

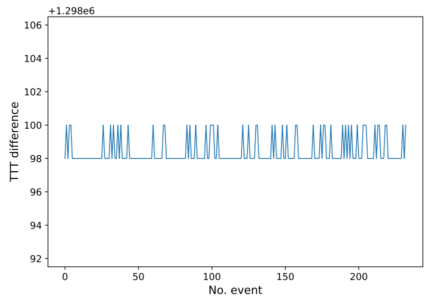


Figure 4.17: Evolution of the difference of TTTs. There is no more increase rate, but jumps of 2 ns are present.

To differentiate the case where two signals are simultaneous, but with an offset due to the inaccuracy of the digitizers, from the case where two signals are truly offset by 16 ns, we can refer directly to the difference in TTT, the different combinations of which are shown in Figure 4.15. Thus, two signals offset by the same TTT correspond to asynchronous signals. This technique could help to correct these inevitable jumps in the offline event reconstruction software.

4.4.3 | DAQling integration

The integration of an additional digitizer into DAQling has necessitated changes in two major areas of the codebase: The first concerns the `DigitizerReceiver` module, responsible for reading the digitizers and the `DigitizerMonitor` module, responsible for creating and filling in data quality metrics histograms.

Digitizer Receiver

To facilitate integration of the digitizer, a new C++ structure, `MonitoringValuesHandler`, has been added. This structure stores the metrics registered for publication in InfluxDB for time series visualisation, and its definition is described in Code Fragment 2.

```
struct MonitoringValuesHandler {
    MonitoringValuesHandler() : temps(16) {}
    //
    std::atomic<int> pedestal[16];
    std::vector<std::atomic<int>> temps;
    std::map<std::string, float> monitoring;

    std::atomic<int> hw_buffer_space;
    std::atomic<int> hw_buffer_occupancy;
    std::atomic<int> triggers;  // number of triggers
    std::atomic<float> time_read;
    std::atomic<float> time_parse;
    std::atomic<float> time_overhead;
    std::atomic<int> corrupted_events;
    std::atomic<int> empty_events;

    // intermediate monitoring variables, reset after each read sequence
    float intermediate_read_time = 0;
    size_t intermediate_receivedEvents = 0;
    float intermediate_parse_time = 0;
};
```

Code Fragment 2: Definition of the the `MonitoringValuesHandler` structure.

Each module has a set of functions to implement, which will then be executed during the various FSM states defined by DAQling. The function executed during data-taking is the part most affected by the addition of the new digitizer. As explained in section 4.2, the digitizers must be read out alternately, N events per readout. Therefore, most of the changes have been to transform the existing code to this alternate readout schema. A diagram summarising the logic of this function is shown in Figure 4.18. Note that this logic is repeated until the measurement is stopped (stop command sent to the module), or when an error or crash occurs.

At each iteration, depending on the occupancy of the digitizer buffers, the process will either pause for 1 millisecond, which allows the polling rate to be set slightly higher than the expected rate, but it limits unnecessary network traffic. If at least one of the buffers is not empty, readout will take place in groups of N events (configured by the BLT size) or smaller if the number of events in the buffer is less than the BLT size. After the N events have been read, the raw payloads are converted into a C++ data fragment, before being sent to the event builder. Once this operation has been completed, the readout moves on to the next digitizer, and so forth, until the number of events retrieved initially falls to 0 for both digitizers. Theoretically, this means that there is a difference

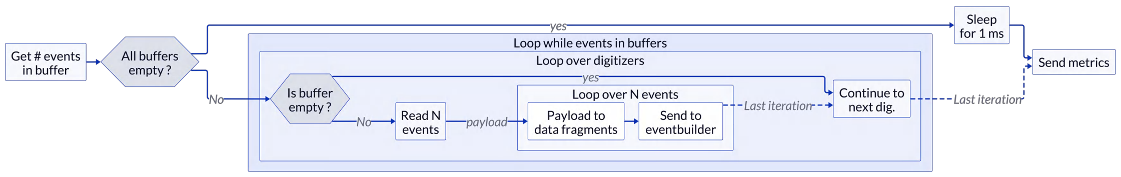


Figure 4.18: Diagram of the digitizer readout logic during data taking.

of no more than N events between the two digitizers. Finally, once the readout sequence is complete, the monitoring metrics are updated so that they can be viewed on Grafana (example of two Grafana time series plots on Figure 4.19) and Redis via InfluxDB.

Digitizer Monitor

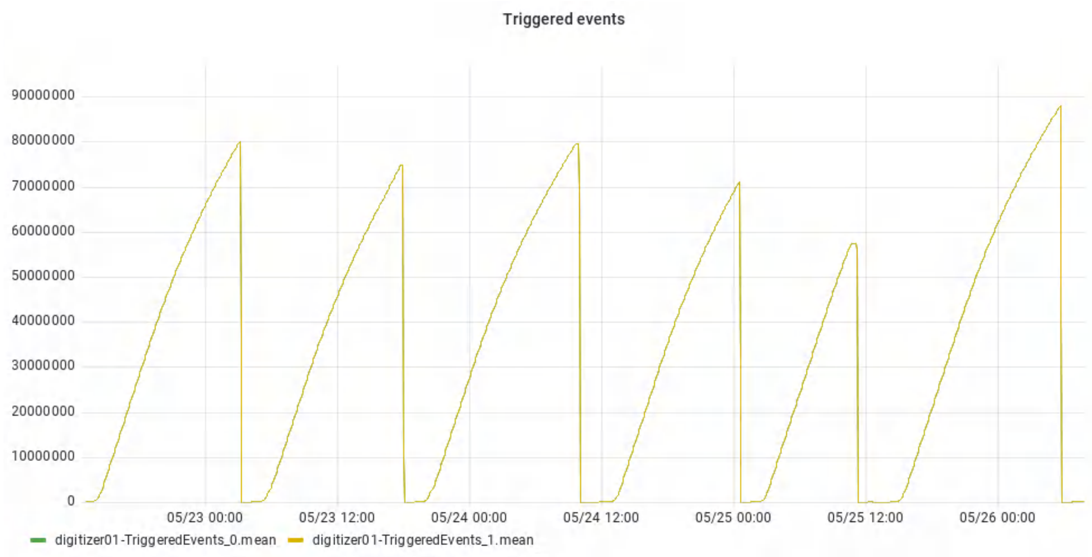
The changes were mainly linked to the addition of histogram metrics for the new digitizer.

To simplify channel remapping with the addition of the digitizer, a new field has been added to the module's JSON configuration file, allowing the source to be linked directly to the channel used. The various fields are described in the Code Fragment 3. The number of channels in the second digitizer, instead of starting again from 0 to 15, continues from 16 to 31. Thus, for example, the channel labelled 10 on the second digitizer will correspond to channel $(15 + 1) + 10 = 26$.

```json
{
    "...",
    "mapping_channels": {
        "input_clock": 31, // channel 15 from 2nd digitizer.
        "fasernu_ch0": 4,
        "fasernu_ch1": 5,
        "veto_st2_ch0": 6,
        "veto_st2_ch1": 7,
        "btm_timing_ch0": 8,
        "btm_timing_ch1": 9,
        "top_timing_ch0": 10,
        "top_timing_ch1": 11
        },
    "..."
}
```

Code Fragment 3: JSON configuration fields related to channel mappings.

With these changes, the histogram data linked to the two digitizers appears in Re-



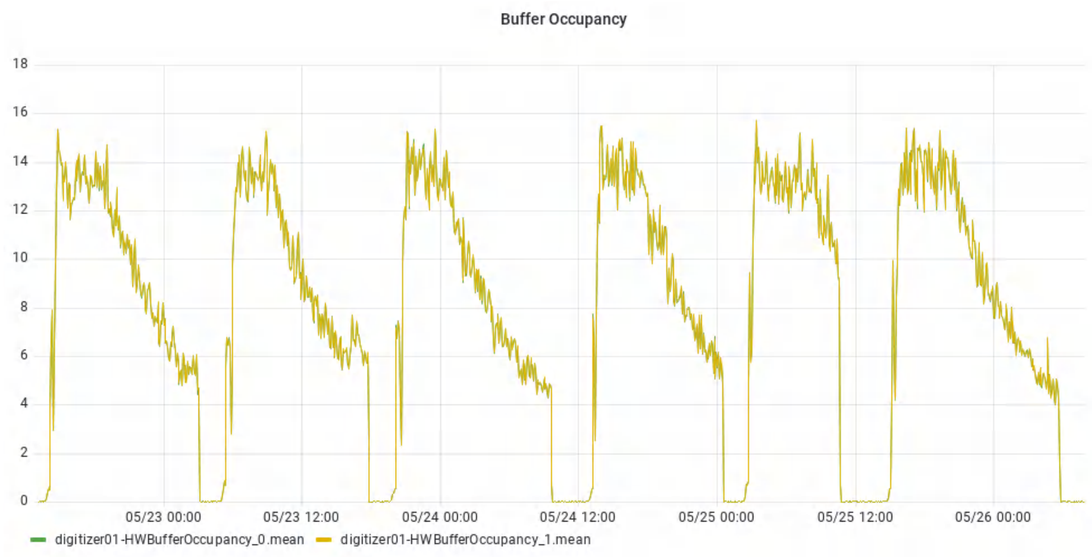
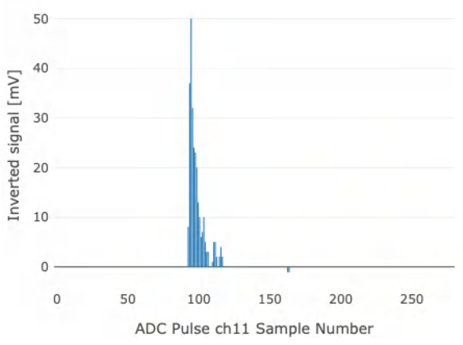
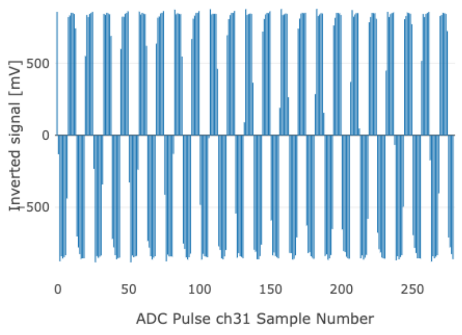


Figure 4.19: Examples of Grafana time series for the number of triggered events (top) and the buffer occupancy (bottom) for both digitizers.The fact that the two curves overlap is a promising, because it means that the two digitizers are well synchronized. In particular, we can see that the number of events in the buffers (buffer occupancy) is almost identical, thanks to the alternate readout. The metrics responsible for updating the buffer occupancy and the triggered events are stored in the `MonitoringValuesHandler` structure, respectively at L.9 and L.10 from Code Fragment 2. The data represents a succession of fills, the progressive fall in the trigger rate corresponding to a drop in the number of bunches in the LHC.

dis, and is displayed in the online monitoring histogram viewer (an example for each digitizer is shown in Figure 4.20)



(a) ADC values for channel 11, which, according to the configuration mapping, L. 11, corresponds to the first channel of the top scintillator in the timing station.



(b) ADC values for channel 31, which correspond (L. 4 in the configuration) to the clock channel, hence the difference with the other ADC values histogram.

Figure 4.20: ADC values from both digitizers displayed on the online histogram monitoring web page.

4.5 | TI12 Installation

4.5.1 | Digitizer and Calorimeter installation

The second digitizer and PMT modules were installed in the tunnel in two stages:

- Installation of the digitizer in the crate.
- Installation of the new PMT modules.

The cables had to be disconnected from the front panel of the existing digitizer in order to insert the second digitizer into the VME crate. As explained in section 4.4.2, the PLLs were also correctly reprogrammed in order to synchronise the two digitizers. The procedure was completed without any major problems, thanks to the previous tests performed on the surface.

After testing the calorimeter with a single PMT module exchanged (shown in Figure 4.21), the other modules were replaced. When testing the system without the protective metal cage, it was noticed that the upgraded calorimeter did not solve the problem of noise related to the GSM antenna present a few meters away. Thanks to the fact that the new PMT modules are approximately the same size as the old PMTs, the cage could be

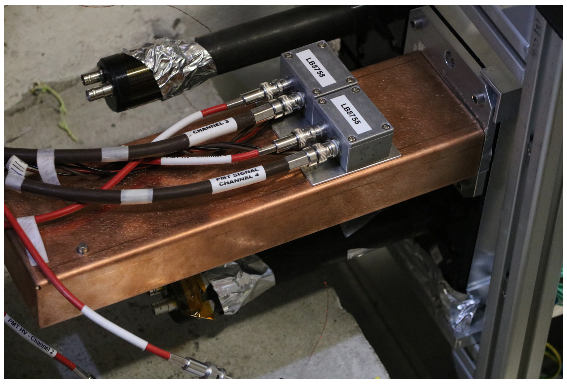


Figure 4.21: Picture of a new PMT module mounted on the calorimeter, alongside the old PMTs.

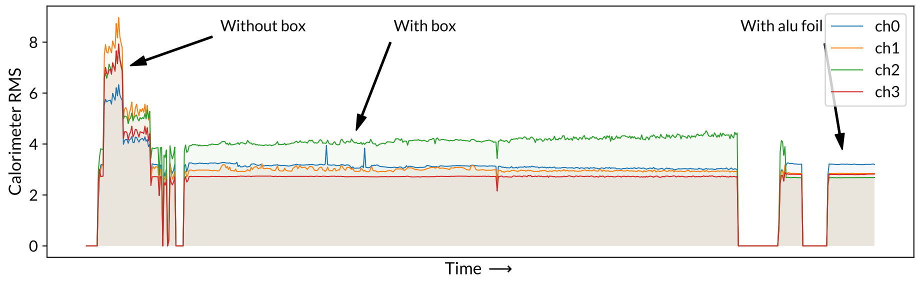


Figure 4.22: Plot of the RMS noise for the calorimeter channels as a function of time, showing the impact that the addition of the metal box and aluminium foil has on the noise caused by the GSM antenna.

put back in place, which greatly reduced the noise. Wrapping the cables in aluminium foil also helped to reduce the impact of the antenna to an acceptable noise level. The impact of the cage and aluminium foil is shown in Figure 4.22.

4.5.2 | High rate tests with the combined system

These tests were performed mainly to find out the limits of the digitizer trigger rate, i.e. the trigger rate at which the digitizer buffers fill up faster than they empty. If the readout system is not fast enough, the buffer will fill up little by little until it is completely full. At this point, the digitizer emits a BUSY signal which stops the acquisition of new events, allowing the buffer occupancy to decrease. In Figure 4.24, we see when the readout logic is fast enough (low buffer occupancy) and when it is not (buffer occupancy reaches 100%). For the purposes of this tests, the trigger rate is generated jointly by the LED calibration system and the TLB. The LED calibration system allows a gradual increase in the trigger rate, while the TLB provides a baseline. The different tests, as displayed

Test	Nb. Channels	Buffer Length (BL)	Block Transfert (BLT)	Max Trigger Rate
1	22	300	1	2 kHz
2	21	280	1	2.1 kHz
3	21	280	2	3 kHz
4	21	280	4	3 kHz (rate limited)
5	21	280	4	3.7 kHz
6	21	280	8	3.9 kHz

Table 4.2: Results of the high rate tests. Test number 4 reached the maximum rate allowed by the rate limiter, whose limit was increased to 5611 Hz in order to be able to continue the tests.

in Figure 4.24 are listed in Table 4.2.

There was a good improvement between tests 2 and 3, as well as 3 and 5, i.e. for 2 and 4 events per readout. The change from 4 to 8 events per readout is not sufficiently advantageous to take the risk of reading 8 events per readout, as if a communication/network problem occurs and packets are lost, all 8 events are lost. So, according to the performed tests, 4 events per readout seems to be the best compromise.

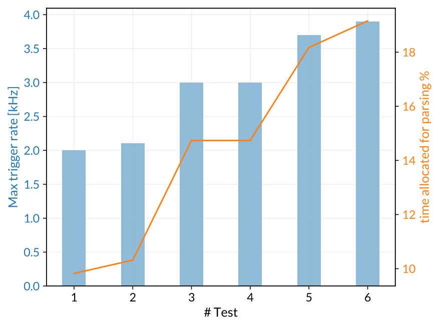


Figure 4.23: Max trigger for the tests performed, as well as the proportion of time needed to parse the data from the digitizer.

It is also interesting to look at the proportion of time allocated to event parsing for a given maximum rate. In Figure 4.23, we can see that for a BLT of 8 with a maximum rate of around 4 kHz, i.e. a time window of 0.00025 seconds, the percentage of time allocated to parsing is almost 20%, which suggests that optimisations could perhaps be made in the future.

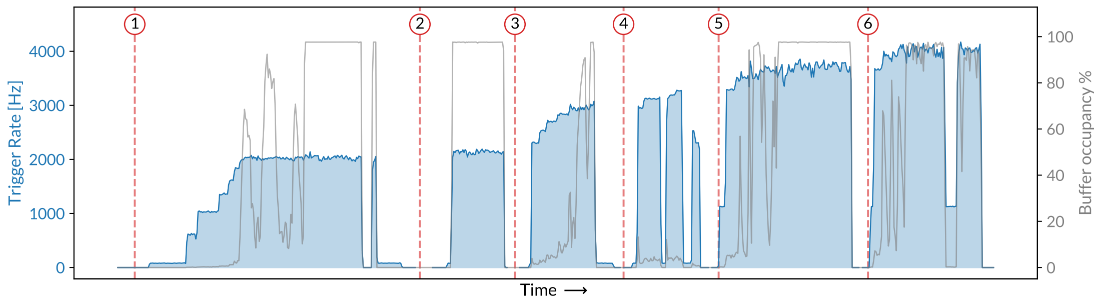


Figure 4.24: Rates and occupancy during high rate tests

5

The Run Control Implementation

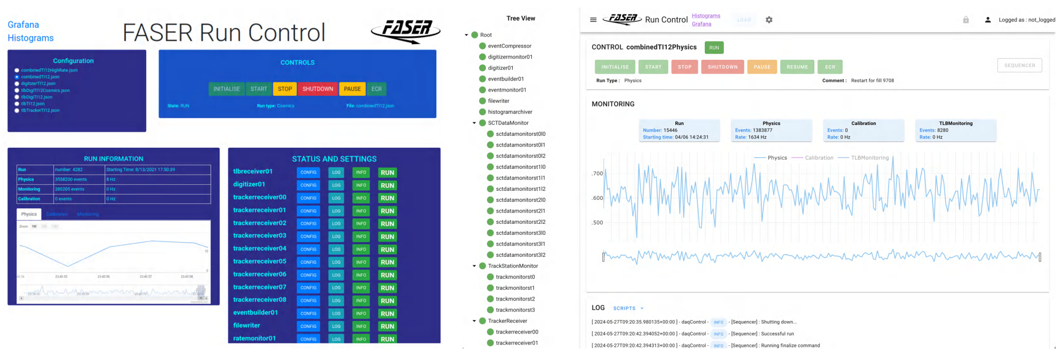
5.1 | Motivation

Since FASER began operating, several web interfaces have been used to control its data acquisition system. With DAQling, the data acquisition framework used by FASER, provided only a Command Line Interface (CLI), but it lacked several features and had the typical drawbacks of CLIs in general, such as the need to authenticate and connect to the server to control the DAQ and very little support for DAQ process monitoring. The use of the CLI could also allow multiple users using it at the same time, which could lead to race conditions (when multiple processes try to access a common resource).

An initial web interface (shown in Figure 5.1(a)) was implemented in 2019, providing configuration and control of the DAQ system as well as visual feedback. With the arrival of DAQling `v0.10.0` in 2021 and the introduction of the DAQ tree structure, enabling individual control of modules, a new implementation of the Run Control, shown in Figure 5.1(b), was then developed. At the time, it provided new features such as user authentication for a more restrictive access of the web interface, interlock preventing race conditions and more granular control on the DAQ modules. Over the last two years, the Run Control has been tested and used successfully. Other sub-interfaces and features have been added progressively. Monitoring helper tools have also been implemented to provide better insight of the DAQ system.

As new features were added, the application became increasingly difficult to maintain because the Run Control had not been designed to be scalable.

In 2023, the question of making FASER's Run Control available to other experiments was raised. This meant including a template Run Control software in the DAQling framework, which could then be extended by other experiments. The integration of the Run Control presented a number of challenges, especially as the version of the FASER Run Control at the time was very difficult to scale. In addition, the DAQling features and those added by FASER Run Control were closely linked, making the system complex to modify. Some DAQling features were not implemented or were abandoned



```
[ 2024-05-27T09:20:35.980135+00:00 ] - daqControl - INFO - [Sequencer] : Shutting down...
[ 2024-05-27T09:20:42.394052+00:00 ] - daqControl - INFO - [Sequencer] : Successful run
[ 2024-05-27T09:20:42.394313+00:00 ] - daqControl - INFO - [Sequencer] : Running finalize command
```

(a) Old version, before 2021. It didn't implement authentication nor interlock.

(b) New version, after 2021. Didn't change with the reimplementation.

Figure 5.1: Run Control GUI interfaces.

because FASER did not use them, such as the ability to manage several run configurations at the same time, which had to be implemented in order to create a template for DAQling. This feature in particular required a thorough reimplementation of the Run Control.

To create a Run Control template for integration in DAQling, it was necessary to re-implement the Run Control software, separating the purely DAQling parts from the FASER-specific elements to enable updates to be made to the template without generating conflicts with the experiment-specific parts. The architecture of the Run Control software in the form of a Python package suited this approach well.

In addition, server configuration, logging and error handling needed to be redesigned to provide a more stable structure for future extensions. As well as being necessary for integration with DAQling, these changes would benefit FASER directly, as some requested features were difficult or impossible to implement in the current state of the application, such as automatic restart when a module encountered an error/crash. Although the Run Control had to be reimplemented at both backend (server) and frontend (web interface) levels, the aesthetics of the web interface did not need to change, as the panel-based interface structure (Figure 5.3) was already well suited to the modular aspect of the new architecture.

The following chapters describe the general organisation and the different parts of the Run Control, including the additions related to FASER.

5.2 | General Tour

Overview

The main features of the Run Control GUI are listed below. The features will be discussed in section 5.3.

Authentication

Users must identify themselves via CERN Single Sign-On (SSO) [27] or automatically via a value based on user's IP address in order to interact with the web interface.

DAQ Control

It is possible to send individual or grouped commands to DAQ modules, while preventing simultaneous commands from multiple users via an interlock system.

Monitoring

Regular process status checks are performed with the ability to send custom alerts to Mattermost [28] in case of crashes or errors (configurable).

Automation

An optional automatic restart can be performed if a crash or error occurs. The processes that trigger an automatic restart on crash or error can be selected using regular expressions. The automatic restart process stops if a manual command is issued.
Also available as a standalone script, the sequencer can be used to create execute run sequences with different parameters and run configurations. A sequence step stops after a configurable time or number of events before starting another. The sequencer is fully controllable from the web interface via a dedicated panel.

Logs

DAQ process logs can be consulted in real time. The logs of scripts referenced in the run configuration can also be consulted.

Extensibility

A Python class is available to facilitate interaction between external scripts and the Application Programming Interface (API) of the Run Control. For example, the sequencer can be used to configure sequences via a configuration file. Each sequence corresponds to a complete DAQ cycle (initialise → start → stop → shutdown). The sequencer is available as a Python script, but also has a dedicated interface on the Run Control web interface (shown in section 5.2).

Configuration

The Run Control is configurable via a JSON file, allowing application settings to be modified, such as the authentication method, regular expressions for automatic restart, and toggling of single run mode (only one active run configuration is allowed). A Local mode is available, which ignores communication with external services, such as the run service or InfluxDB and is mainly for development/testing purposes.

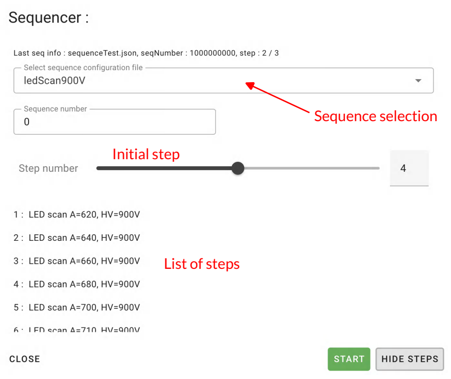
The Web Interface

The Run Control interface (Figure 5.3) consists of a header, a side bar and the main area, where panels associated with different features are present. The header contains links to the Grafana interface and monitoring histograms, a dropdown for selecting the run configuration file and options related to interlock and authentication.

If a run configuration is loaded, the side bar displays the modules present in the configuration with their status in a tree structure. The main area contains the control panel. This is where the state of the DAQ system can be controlled. The sequencer can be accessed via the dedicated button on the right of the panel.

The sequencer is displayed as a pop-up window, where the sequence configuration file and the initial step can be selected. The generated list of steps is also displayed in a dedicated part, as shown in Figure 5.2.

The second panel, the monitoring panel, displays basic monitoring data related to the current run, in the form of time series and text. The last panel is the logs panel. It displays logs related to all run control activities and attached processes (more information in section 5.3.3). It is also possible to consult the logs of scripts specified in the configuration file and launched automatically when a configuration is initialized (auxiliary scripts). When a module (not a category) is selected on the tree in the side panel, a new panel appears, listing all the metrics, in real time, associated with this module.



Last seq info : sequenceTest.json, seqNumber : 1000000000, step : 2 / 3

Select sequence configuration file: ledScan900V

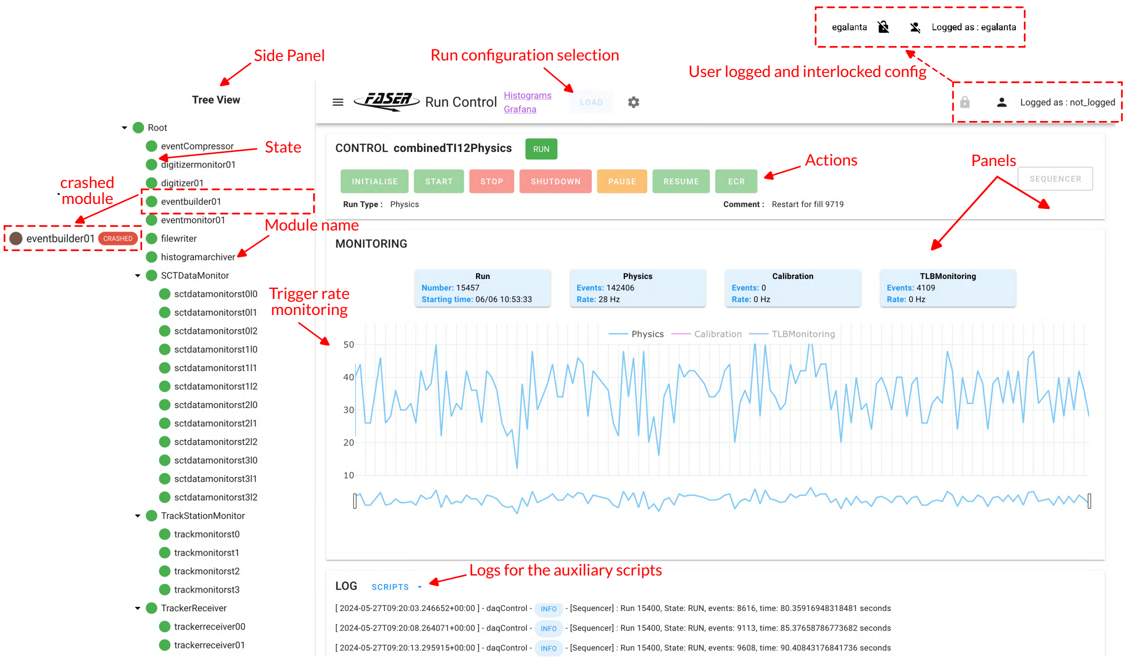
Sequence number: 0

Step number: 4

1 : LED scan A=620, HV=900V
2 : LED scan A=640, HV=900V
3 : LED scan A=660, HV=900V
4 : LED scan A=680, HV=900V
5 : LED scan A=700, HV=900V
6 : LED scan A=710, HV=900V

Figure 5.2: Pop-up window for the sequencer interface.

It is also possible to access the module's logs, ei-
ther in their entirety (not updated in real time) or the tail of the logs (in real time).



```
[ 2024-05-27T09:20:03.246652+00:00 ] - daqControl - INFO - [Sequencer] : Run 15400, State: RUN, events: 8616, time: 80.35916948318481 seconds
[ 2024-05-27T09:20:08.264071+00:00 ] - daqControl - INFO - [Sequencer] : Run 15400, State: RUN, events: 9113, time: 85.37658786773682 seconds
[ 2024-05-27T09:20:13.295915+00:00 ] - daqControl - INFO - [Sequencer] : Run 15400, State: RUN, events: 9608, time: 90.40843176841736 seconds
```

Figure 5.3: Web interface of the Run Control. It is structured around a series of panels, each dedicated to a specific function: control, monitoring and log.

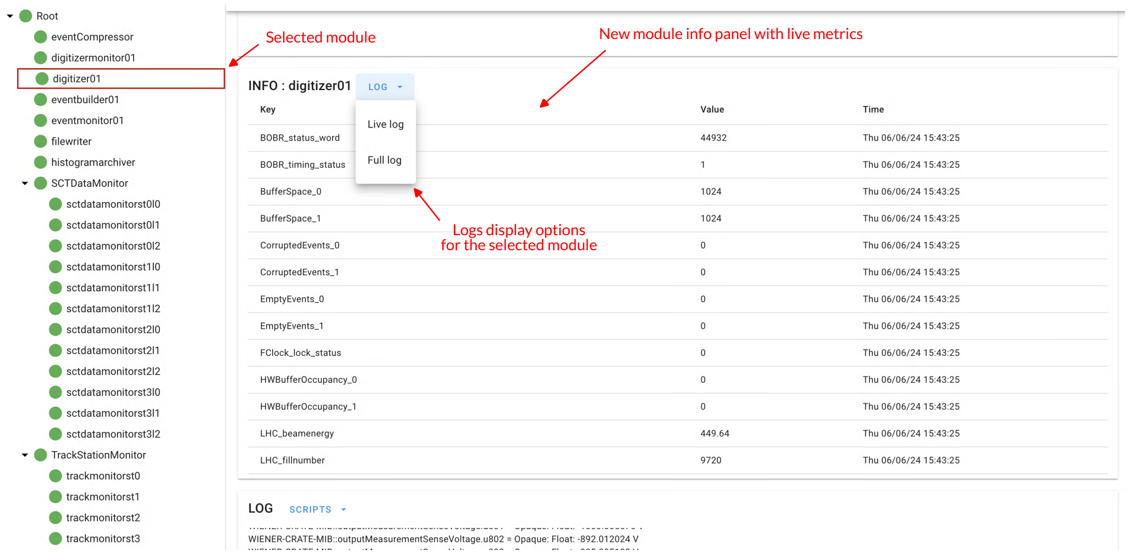
5.3 | Implementation

5.3.1 | Technologies used

VueJS and Vuetify

The browser interface (hereafter referred to as the frontend or client) uses the Vue.js [29] framework. Vue.js is a powerful, versatile and open-source JavaScript framework for creating user interfaces. Vue.js is called "progressive" because it can be gradually integrated into existing projects, adding only the necessary functionality, thereby keeping dependencies to a minimum.

Vue.js can be used with tools such as Vite [30] or Webpack [31] to produce optimised JavaScript code, but this has the disadvantage of considerably increasing the number of dependencies to be installed (notably Node.js and npm) when the application has to be deployed on a server. Vue can also be used via a Content Delivery Network (CDN)



Key	Value	Time
BOBR_status_word	44932	Thu 06/06/24 15:43:25
BOBR_timing_status	1	Thu 06/06/24 15:43:25
BufferSpace_0	1024	Thu 06/06/24 15:43:25
BufferSpace_1	1024	Thu 06/06/24 15:43:25
CorruptedEvents_0	0	Thu 06/06/24 15:43:25
CorruptedEvents_1	0	Thu 06/06/24 15:43:25
EmptyEvents_0	0	Thu 06/06/24 15:43:25
EmptyEvents_1	0	Thu 06/06/24 15:43:25
FClock_lock_status	0	Thu 06/06/24 15:43:25
HWBufferOccupancy_0	0	Thu 06/06/24 15:43:25
HWBufferOccupancy_1	0	Thu 06/06/24 15:43:25
LHC_beamenergy	449.64	Thu 06/06/24 15:43:25
LHC_fillnumber	9720	Thu 06/06/24 15:43:25

```
WIENER-CRATE-MIB::outputMeasurementSenseVoltage.u802 = Opaque: Float: -892.012024 V
```

Figure 5.4: When a module is selected in the tree view, a new panel appears containing live metrics and the log menu.

to include it more easily in existing HTML web pages (the method chosen for the Run Control project).

Vue works with a system of components. A Vue application is made up of a main instance and components, which are a simplified Vue instance with a given name. Each component has a lifecycle (an overview of which is shown in Figure 5.5), where it is possible, at each stage, to register and execute functions linked to the specific hooks, some of which are listed below.

- `created`: This function is called when all operations relating to the initialisation of the component's states have been completed.
- `mounted`: called when the Document Object Model (DOM) of the component has finished being initialized.
- `updated`: called when the DOM of the component has been modified.
- `unmounted`: called when the component is unmounted, before being destroyed.

The components encourage a modular approach, facilitating code reusability and separation of concern. Although using a javascript framework adds additional knowledge in addition to the language, when the page is interactive and elements change frequently, it improves the development experience as well as the responsiveness of the application.

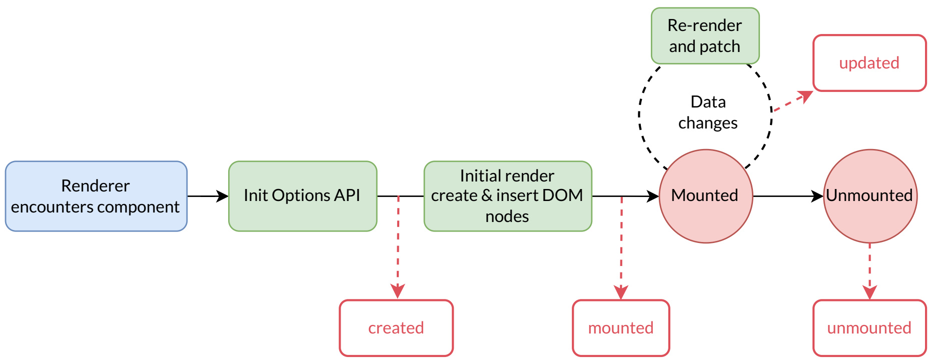


Figure 5.5: Simplified diagram of the lifecycle of a Vue component with the main hooks used. Inspired by the diagram in the official documentation [32].

Vuetify is an open-source library of UI components and is partially used for the styling of elements on the interface. It simplifies the implementation of various graphical elements and integrates well with the Vue.js framework.

Flask

The server side is powered by Flask [33], an open-source micro-framework written in Python. It makes it easy to create web applications by providing the basic functionality. Extensions can be added to extend its capabilities.

Its routing system makes it easy to define "routes" (URLs) to Python functions using decorators, as illustrated by Code Fragment 4.

```python
from flask import Flask, current_app # import Flask package and the proxy
↪ object to the app
app = Flask(__name__) # creates the Flask instance

# route decorator, linking "/" to index function
@main.route("/", methods=["GET", "POST"])
def index():
    CONFIG = current_app.config["daqControl"] # getting the server
    ↪ configuration object
    return render_template("index.html", localOnly=CONFIG["local_only"])

if __name__ == '__main__':
    app.run(debug=True) # starting the server in DEBUG mode.
```

Code Fragment 4: Minimal example for a Flask application, which displays the Run Control web interface at the base URL (route) "/".

Blueprints can be used to structure an application into several modules, centralising routes and making application development and maintenance more efficient.

Redis

The Run Control uses Redis to enable communication with other parallel processes and to save temporary data. Redis [18] (REmote DIctionary Server) is an open source database stored in memory, renowned for its speed and high performance. Although Redis is a database stored in memory and not on disk, it offers the possibility of backing up data permanently using point-in-time backups to disk (snapshots). As well as serving as a database, Redis provides a publication/subscription system (Pub/Sub), useful for events handling.

Redis is accessible via its redis-cli command line, as shown in Code Fragment 5, but it is also possible to interact with it via libraries specific to each language.

```
r = redis.Redis(host="localhost",port=6379, db=4) # creating the connection
... # some code
r.set("selectedConfig", "fullPhysicsConfig") # selecting run configuration
↪  "fullPhysicsConfig"
... # some code

value = r.get("selectedConfig") # retrieve value from key "selectedConfig
... # some code
```

Code Fragment 5: Example of setting and getting a value using the `redis-py` package.

Redis works with a key-value principle, where the key is a string and the value can take various forms such as a set, a list or a hash.

5.3.2 | Project architecture

The Run Control is used via the `daqControl` Python package. This package contains several sub-packages specific to each part of the application : Main, Authentication, Control and Custom, which will be explained in the next sections.

A recurring problem in Flask applications is the "circular import", which occurs when two modules depend on each other. To overcome this, a Flask extension has been implemented to make configuration data and any other object that needs to be globally accessible, available.

The most important element of the Run Control is the `daqHandler`. It is associated with a single run configuration (several instances can therefore coexist) and is responsible for loading the configuration, checking its validity and propagating commands issued by the client (browser or via the run control helper class, see section 5.3.7). The main attribute of the `daqHandler` is the `nodeTree`. Introduced by DAQling in version `v0.10.0`, the nodeTree provides a tree representation of the data acquisition system, where each leaf of the tree corresponds to a module. The daqHandler can be inherited to extend its capabilities. All daqHandlers are themselves managed by the `dhManager` class, which is responsible adding active daqHandlers and removing inactive ones.

A separate thread is dedicated to the stateChecker, a function that continuously checks the status of modules present in daqHandlers and that can be extended according to the user's needs.

In the `utils` folder, functions and classes are made available for standalone use, such as Mattermost notifications or configuration file validation.

5.3.3 | Logging system

As mentioned in section 5.1, it was important to have a logging system capable of aggregating different sources and controlling where the logs are sent.

A customisable and versatile system was implemented using the standard `logging` [34] module supplied with Python. This system is configurable, directly in the code or from a JSON file.

The Python `logging` module has three key concepts: loggers, handlers and formatters. Loggers are responsible for capturing logs generated by the application and passing them on to handlers. Handlers retrieve messages from the loggers and send them to different destinations. Custom handlers can be implemented by sub-classing the default handler. The format in which messages are sent is managed by the formatters. As for the handler, custom formatters can be implemented from the default formatter. A logger may therefore contain several handlers, each responsible for a destination, whose format is managed by a formatter (as shown in Figure 5.6).

For the Run Control, the `daqControl` is created and is accessible for all sub-packages. Three handlers are added to this logger :

- A `FileHandler`, responsible for saving logs to disk.
- A `StreamHandler`, for display on standard output.

- A `SocketIOHandler`, a custom handler implemented specially for FASER to communicate between the server and the Web Interface. It is this handler which updates the logs displayed on the web interface.

While the `StreamHandler` handler uses a default formatter, the `FileHandler` and `SocketIOHandler` uses a custom formatter that formats messages in a JSON format. This format not only facilitates communication between server and client, but also allows logs to be saved in a `.jsonl` (json line) format, which is easier to analyse and integrate into other analysis and monitoring tools.

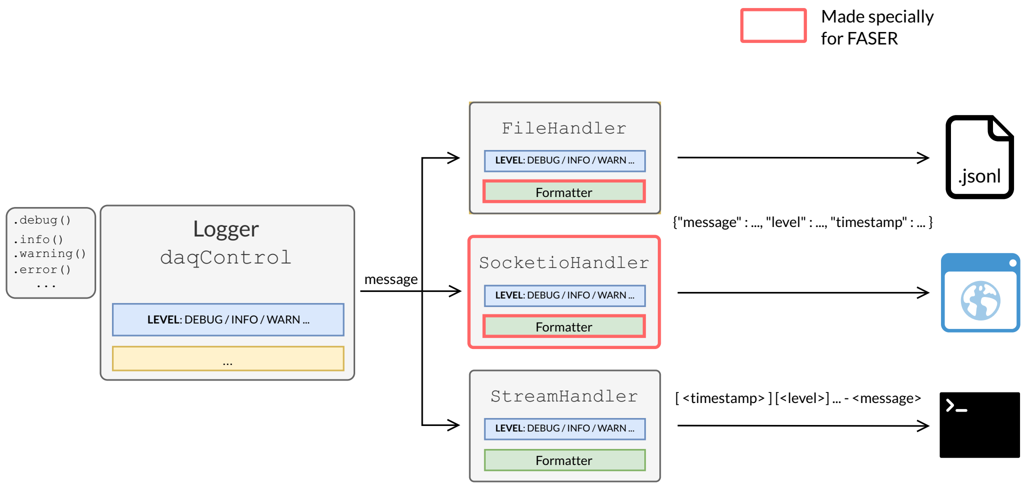

Made specially for FASER
.debug()
.info()
.warning()
.error()
...
Logger
daqControl
LEVEL: DEBUG / INFO / WARN ...
...
message
FileHandler
LEVEL: DEBUG / INFO / WARN ...
Formatter
.jsonl
{"message" : ..., "level" : ..., "timestamp" : ... }
SocketioHandler
LEVEL: DEBUG / INFO / WARN ...
Formatter
StreamHandler
LEVEL: DEBUG / INFO / WARN ...
Formatter
[<timestamp>] [<level>] ... - <message>

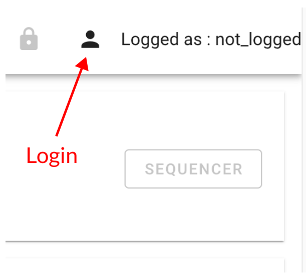
Figure 5.6: Diagram of the Run Control logging system

These specific handlers provide granular control over how logs are output and saved. As an example, logs from Flask's debugging tools do not need to be sent to an interface user, but it is still useful to keep them to ensure that the application is working correctly. On the other hand, an error from Flask should be sent to the interface so that it can be dealt with as quickly as possible.

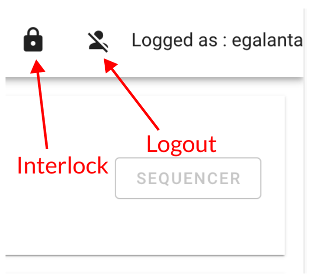
The introduction of the new `daqControl` logger provides a good basis for future implementations, since a new functionality could be implemented and benefit directly from the already existing logging infrastructure.

5.3.4 | Authentication

Since the Run Control handles the operation of the experiment, it is necessary to limit control to authorised members of the FASER collaboration. To achieve this, Single-Sign



(a) To take control of the interface, the user first have to login.



(b) Then, the user have to request interlock via the lock icon.

Figure 5.7: Part of the web interface dedicated to authentication and interlock.

On [27] authentication has been implemented (shown on Figure 5.7 (a)) , taking advantage of CERN's existing infrastructure, the CERN Single-Sign On, powered by Keycloak [35]. It has several advantages:

- Secure authentication via two-factor authentication.
- Possibility of using CERN e-groups to restrict access and define several levels of authentication.
- A user already logged in on one of the CERN pages does not need to re-authenticate on the application.

If SSO authentication is not required (for tests, for example), the Run Control falls back to a basic automatic login via a IP address, where the IP address is used to distinguish two different users. This method requires no configuration, but it is not very reliable, as several users can have the same IP address, under certain conditions.

SSO authentication requires several steps before it becomes operational. The application must be registered on the CERN Application Portal [36]. Registration provides identifiers that can then be configured in the Run Control. It is also on this platform that it is possible to specify the e-groups authorised to take control of the Run Control interface. A description of the workflow for the two authentication methods is shown in Figure 5.8.

5.3.5 | Interlock

Interlock is a feature that prevents several users from performing actions simultaneously on the same run configuration. This ensures that there is no conflict between two

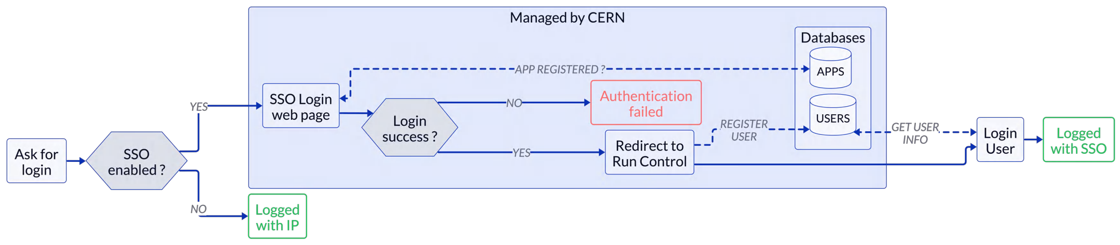


Figure 5.8: Diagram of the Run Control authentication logic.

different commands. A user can only request the interlock if they are logged in, as the interlock uses the *upn* (CERN login name) as an identifier if SSO is enabled (as shown in Figure 5.7 (b)), otherwise the IP address. If the configuration is not already locked, the interlock will be granted, otherwise it will be refused. Once the interlock has been granted, the user has a certain amount of time (which can be configured) to carry out his actions. If, at the end of this time, the user has not performed any action, the interlock will be automatically released.

The interlock is based on the ability of Redis to manage temporary keys - the interlock is released when the keys expire. If the interlock is active, user actions will extend the duration of the interlock. A diagram of the interlock process is shown in Figure 5.9.

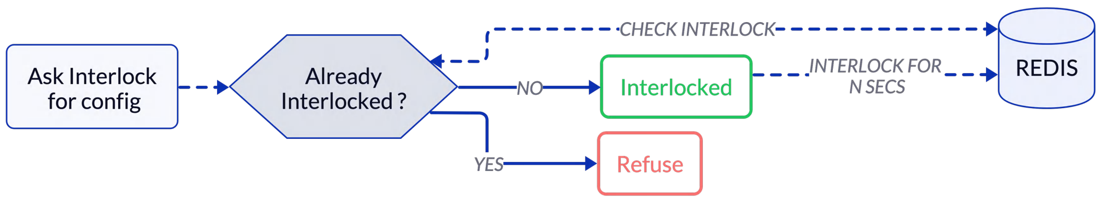


Figure 5.9: Diagram of the interlock logic, where N is the configurable interlock time duration.

5.3.6 | Actions

In the Run Control, "Commands" correspond to the commands registered by DAQling and enable the modules to move between their different states of the finite state machine.

On the other hand, "Actions" correspond to operations which influence the general state of the Run Control and are implemented by the Run Control itself. Therefore the DAQ modules are interacted via the actions rather than directly interacted with the commands. On the interface, the commands can still be accessed by activating the "expert" mode from the menu.

FASER implements seven actions: `INITIALISE`, `START`, `STOP`, `PAUSE`, `RESUME`, `ECR` and `SHUTDOWN`.

- `INITIALISE` prepares the DAQ system by configuring the DAQ modules according to the run configuration selected and launches any auxiliary scripts defined in the same configuration file.
- `START` (`STOP`) starts (stops) a new run.
- `PAUSE` (`RESUME`) deactivates (reactivates) the global trigger, while continuing the same run.
- `ECR` resets the event counter. It is automatically done before an event count overflow, but the action can be performed in case of de-synchronization between fragments.
- `SHUTDOWN`, stops all DAQ modules and all auxiliary scripts. In the event of an error/crash in the modules, this action resets the DAQ system.

When an action is performed, an *inTransition* flag is raised on the `daqHandler` responsible for the specific run configuration, which prevents any parallel action on the same `daqHandler` until the end of the action.

The action is divided into three parts: action pre-sequence, action sequence and action post-sequence.

Action pre-sequence

The action pre-sequence part allows operations to be performed before commands are sent and depends on the actions performed:

- `INITIALISE`: reloads the configuration to take account of possible changes to the configuration file and starts the auxiliary scripts.
- `START`: If local mode is active, the run number is set to 1 000 000 000 (arbitrary value). Otherwise, the next run number is retrieved from the run service.
- `STOP`: Run-related information is sent to the run service (if local mode is disabled).
- `SHUTDOWN`: If a crash/error occurs and this action is performed, the Run Control sends a request to the run service to report it. The auxiliary scripts launched during the initialisation phase are then stopped.

Action	Sequence	Additional instructions
INITIALISE	add, configure	save log locations
START	start	-
PAUSE	disableTrigger	-
ECR	ECR	-
RESUME	enableTrigger, resume	-
STOP	stop	-
SHUTDOWN	unconfigure, shutdown, remove	-

Table 5.1: Table of the sequence of commands associated with the actions.

The `PAUSE`, `RESUME` and `ECR` actions do not include instructions in the pre-sequence part.

Action sequence

The sequence part contains the sequence of commands to be performed. A summary of the command sequences for each action is listed in table 5.1. Except for the sequence linked to `INITIALISE`, no command sequence requires additional instructions.

Action post-sequence

Finally, the action post-sequence part contains the instructions for moving on to the next state, depending on the outcome of the command sequence.

The IDLE state of the Run Control is `DOWN` and only the `INITIALISE` action is permitted. If the `INITIALISE` action is successful, the Run Control will go into `READY` state, where it will be possible to start a new run. On the other hand, if a problem occurs during module configuration, the Run Control state will be `ERROR`, allowing only the `SHUTDOWN` action.

Once data acquisition has started, the Run Control state is `RUN`, and it is possible to `PAUSE` (and then `RESUME` or `ECR`) or `STOP` to stop data acquisition. At the end, the daqHandler removes its *inTransition* flag to allow new commands to be performed.

5.3.7 | Run Control Class

Although the main method of controlling the DAQ system is via the Run Control web interface, it is possible to create external scripts to control the state of the run via a simple Python class. Currently, the class provides shortcuts for the following actions: `INITIALISE`, `START`, `STOP` and `SHUTDOWN`. It is also possible to retrieve information about an active run or a particular module, and to send logs via the Run Control logger. The diagram in Figure 5.10 illustrates the use of the RunControl class.

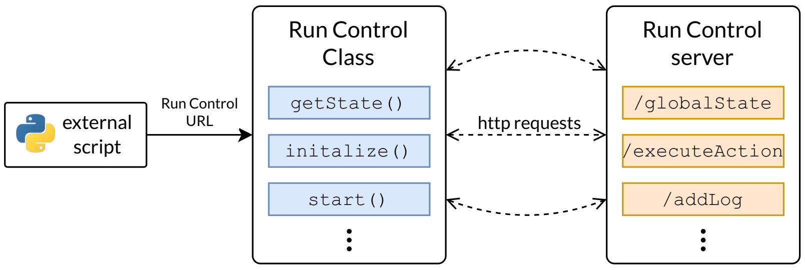


Figure 5.10: A diagram illustrating the use of the Python class to remote control the Run Control application.

If the already implemented methods are not sufficient, it is always possible to access other Run Control API endpoints via raw HTTP requests.

For FASER, several scripts (mentioned in section 5.2) take advantage of this interface, such as the `fillChangeHandler`, which automatically restarts a run when there is a fill change at the LHC, or the `sequencer`, which automatically performs run sequences.

5.4 | Outlook

The re-implementation of the Run Control has resulted in a much more modular and modifiable structure by separating the core functions from the other parts. It has also reintroduced the possibility of having several active run configurations, while offering the option of deactivating this feature via the 'single run mode'. The introduction of a new logging system has made it possible to unify the various sources (existing and future) through a single system, improving error management.

Although most of the core codebase is decoupled from experiments specific functionalities, a small coupling still exists due to the way Flask is integrated. Further thinking is needed to address this issue. Moreover, modifying certain parts of the application that may change for other experiments still requires a good understanding of the Run Control. This includes adding configuration elements, replacing the default `daqHandler`, or modifying frontend elements.

The remaining steps needed for the Run Control to become a template for future experiments will therefore be to finish the separation between core features and other features and ease the extension of the application, which will reduce the knowledge required to extend and customize the software for future uses.

a

6

Conclusion

The FASER experiment extends the LHC physics potential by probing the far-forward regions to find evidence of Long-Lived Particles such as the dark photon and Axion Like Particles. This could lead to advances in Physics beyond the Standard Model. During the last two years of data taking, FASER has demonstrated its capabilities by observing the first collider neutrinos [5] and first dark photon studies [6].

An upgrade of the FASER calorimeter system to a split readout system significantly extended the dynamic range of the calorimeter energy sampling but necessitated adding a second digitizer to the previous single digitizer readout. Standalone tests showed that synchronization of the digitizers was necessary, and a dual readout was implemented. The new implementation allows scalable integration of additional digitizers in the future. High-rate tests were successfully performed on the full detector, demonstrating that the addition of a digitizer can still handle trigger rates higher than expected, provided the readout parameters are optimized. Since its implementation in February, the experiment has resumed data taking successfully, and the new calorimeter split readout system's early performance results seem promising [37]. Future work could focus on event parsing in the digitizer event processing software, which was shown to occupy 20% of processing time.

The FASER Run Control provides a web interface for manual control of the FASER TDAQ system as well as impose control autonomously, for instances when set to operate a sequence of calibration runs. The lack of scalability of the FASER Run Control and the need to add features such as automatic restarts led to a major re-implementation and restructuring, which constituted the second part of this master's project. This was done through a new Python package structure where core functionalities were implemented, as decoupled as possible from potential extensions. With the upgrade of subsystems such as configuration, logging, and error handling, the new architecture benefited FASER by making it more user friendly and modular, but it also opened the way for other experiments to use the Run Control.
Further improvements in the decoupling of the software's core features could be done,

especially concerning the backend framework Flask, as it requires specific frontend-related file organisation that does not allow easy modifications without being familiar with the Run Control software.

Having been approved by CERN to continue operating during Run 4 at the High Luminosity LHC era, FASER promises interesting future results, particularly with future upgrades.

Bibliography

[1] *LHC Long Term Schedule*. URL: https://lhc-commissioning.web.cern.ch/schedule/LHC-long-term.htm.

[2] CERN Yellow Reports: Monographs. *CERN Yellow Reports: Monographs, Vol. 10 (2020): High-Luminosity Large Hadron Collider (HL-LHC): Technical Design Report*. Dec. 17, 2020. DOI: 10.23731/CYRM-2020-0010. URL: https://e-publishing.cern.ch/index.php/CYRM/issue/view/127. preprint.

[3] F. Collaboration et al. *Technical Proposal: FASERnu*. Jan. 9, 2020. URL: http://arxiv.org/abs/2001.03073. preprint.

[4] F. Collaboration et al. "First Direct Observation of Collider Neutrinos with FASER at the LHC". In: *Physical Review Letters* 131.3 (July 19, 2023), p. 031801. ISSN: 0031-9007, 1079-7114. DOI: 10.1103/PhysRevLett.131.031801.

[5] CERN. *First Collider Neutrinos Detected*. CERN Courier. Apr. 24, 2023. URL: https://cerncourier.com/a/first-collider-neutrinos-detected/.

[6] F. Collaboration. "Search for Dark Photons with the FASER Detector at the LHC". In: *Physics Letters B* 848 (Jan. 2024), p. 138378. ISSN: 03702693. DOI: 10.1016/j.physletb.2023.138378.

[7] F. Collaboration et al. "FASER's Physics Reach for Long-Lived Particles". In: *Physical Review D* 99.9 (May 15, 2019), p. 095011. ISSN: 2470-0010, 2470-0029. DOI: 10.1103/PhysRevD.99.095011.

[8] H. Abreu et al. "The FASER Detector". In: *Journal of Instrumentation* 19.05 (May 1, 2024), P05066. ISSN: 1748-0221. DOI: 10.1088/1748-0221/19/05/P05066.

[9] K. Halbach. "Design of Permanent Multipole Magnets with Oriented Rare Earth Cobalt Material". In: *Nuclear Instruments and Methods* 169.1 (Feb. 1980), pp. 1–10. ISSN: 0029554X. DOI: 10.1016/0029-554X(80)90094-4.

[10] F. Collaboration et al. “The Trigger and Data Acquisition System of the FASER Experiment”. In: *Journal of Instrumentation* 16.12 (Dec. 1, 2021), P12028. ISSN: 1748-0221. DOI: `10.1088/1748-0221/16/12/P12028`.

[11] *DAQling: A Software Framework for the Development of Modular and Distributed Data Acquisition Systems*. GitLab. URL: `https://gitlab.cern.ch/ep-dt-di/daq/daqling`.

[12] *Si5341-d-Evb, Skyworks*. URL: `https://www.skyworksinc.com/en/Products/Timing/Evaluation-Kits/clock/si5341-evaluation-kit`.

[13] *VX1730*. CAEN - Tools for Discovery. URL: `https://www.caen.it/products/vx1730/`.

[14] *SIS3153 USB3.0 and Ethernet to VME Interface*. URL: `https://www.struck.de/sis3153.html`.

[15] *FASER DAQ: Top Level Software for the FASER Trigger and Data Acquisition System*. URL: `https://gitlab.cern.ch/faser/online/faser-daq`.

[16] *ZeroMQ*. URL: `https://zeromq.org/`.

[17] *ERS - Error Reporting Service*. GitLab. Apr. 17, 2024. URL: `https://gitlab.cern.ch/atlas-tdaq-software/ers`.

[18] *Redis*. URL: `https://redis.io/`.

[19] *InfluxDB*. Sat, 15 Jan 2022 15:32:09 +0000. URL: `https://www.influxdata.com/home/`.

[20] *JSON Schema*. URL: `https://json-schema.org/`.

[21] *SCADA*. In: *Wikipedia*. June 1, 2024.

[22] L. Pagliai. “V1730/VX1730 & V1725/VX1725 User Manual”. In: ().

[23] *C++17 - Cppreference.Com*. URL: `https://en.cppreference.com/w/cpp/17`.

[24] *Faser / Online / Digitizer-Readout · GitLab*. GitLab. Apr. 30, 2024. URL: `https://gitlab.cern.ch/faser/online/digitizer-readout`.

[25] *Struck Innovative Systeme Homepage*. URL: `https://www.struck.de/`.

[26] *Scipy.Special.Expit — SciPy v1.13.0 Manual*. URL: `https://docs.scipy.org/doc/scipy/reference/generated/scipy.special.expit.html`.

[27] *Single Sign-On*. In: *Wikipedia*. June 5, 2024.

[28] *Mattermost*. URL: `https://mattermost.com/`.

[29] *Vue.Js*. URL: `https://vuejs.org/`.

[30] *Vite*. vitejs. URL: `https://vitejs.dev`.

[31] *Webpack*. webpack. URL: `https://webpack.js.org/`.

[32] *Vue.Js*. URL: `https://vuejs.org/guide/essentials/lifecycle.html`.

[33] *Flask Documentation (3.0.x)*. URL: `https://flask.palletsprojects.com/en/3.0.x/`.

[34] *Logging — Logging Facility for Python*. Python documentation. URL: `https://docs.python.org/3/library/logging.html`.

[35] *Keycloak*. URL: `https://www.keycloak.org/`.

[36] *CERN Applications Portal*. URL: `https://application-portal.web.cern.ch/`.

[37] *20th International Conference on Calorimetry in Particle Physics*. Indico. URL: `https://indico.cern.ch/event/1339557/contributions/5898500/`.