FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

# Assignment of master's thesis

| | |
|---|---|
| **Title:** | Systematic Comparison of TPX and TPX3 devices regarding luminosity measurements in the ATLAS cavern |
| **Student:** | Bc. Petr Fiedler |
| **Supervisor:** | doc. Dr. André Sopczak |
| **Study program:** | Informatics |
| **Branch / specialization:** | Web and Software Engineering, specialization Software Engineering |
| **Department:** | Department of Software Engineering |
| **Validity:** | until the end of summer semester 2020/2021 |

## Instructions

Analyse the needs of the ATLAS (CERN) luminosity measurements using TPX and TPX3 devices regarding an automatic procedure to determine coherently the luminosity from 2015 to 2018 data. Design and implement a program that reads the TPX and TPX3 data and performs a noisy pixel removal automatically. Incorporate the timing information of the so-called ATLAS luminosity blocks. Test the program and produce performance plots regarding the LHC luminosity curve, short-term precision, linearity, and long-term stability. Compare the TPX, TPX3 luminosities to other ATLAS luminometers. This includes statistical processing of the data stored at the IEAP server to detect anomalies of the functionality of Timepix devices. The specifications of anomalies will be clarified by the supervisor.  The resulting distributions will be made available on a web interface.

*Electronically approved by Ing. Michal Valenta, Ph.D. on 19 November 2019 in Prague.*

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

# Systematic comparison of TPX and TPX3 devices regarding luminosity measurements in the ATLAS cavern

***Bc. Petr Fiedler***

Department of Software Engineering
Supervisor: doc. Dr. André Sopczak

May 6, 2021

# Acknowledgements

I would like to thank my supervisor, doc. Dr. André Sopczak, for all advises and active involvement during the work, and Ing. Michal Valenta, Ph.D. for his advises and consultations he provided. I also would like to thank my family and my girlfriend for the encouragement during my studies.

# Declaration

In Prague on May 6, 2021 . . . . . . . . . . . . . . . . . . . .

## Citation of this thesis

Fiedler, Petr. *Systematic comparison of TPX and TPX3 devices regarding luminosity measurements in the ATLAS cavern.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021. Also available from: ⟨`https://timepix.fiedl.eu/`⟩.

# Abstrakt

Tato práce se zabývá analýzou potřeb pro automatizovanou proceduru k určení měření luminosity ATLAS (CERN) pomocí zařízení TPX a TPX3. Práce se také zabývá návrhem a implementací softwaru, který čte TPX a TPX3 data, odstraňuje závadné pixely a vytváří různé výkonnostní grafy týkající se měření luminosity. Součástí práce je také návrh a implementace webového rozhraní, které zpřístupní výsledky.

**Klíčová slova** zpracování dat, analýza dat, Timepix, Timepix3, luminosita, ATLAS

# Abstract

This thesis deals with an analysis of the needs for an automatic procedure to determine the ATLAS (CERN) luminosity measurements using TPX and TPX3 devices. The thesis also deals with design and implementation of software that reads the TPX and TPX3 data, performs noisy pixel removal, and produces various performance plots regarding luminosity measurements. Furthermore, design and implementation of a web interface, which makes the resulting distributions available, is part of this thesis.

**Keywords** data processing, data analysis, Timepix, Timepix3, luminosity, ATLAS

# Contents

# List of Figures

# List of Tables

# Introduction

As particle physics is moving forward, particle accelerators have to operate on higher energy levels in order to observe rare high-energy interactions. To detect more of the rare events, data-taking has to be improved. New detectors, sensors, and chips are being developed, and the old ones are being improved. Such detectors are Timepix (TPX) [1, 2] and Timepix3 (TPX3) [3] developed as part of the Medipix Collaborations.

These devices were installed in A Toroidal LHC Apparatus (ATLAS) [4] cavern at the European Organization for Nuclear Research (CERN). They take data to measure and study the composition of the radiation field, induced radioactivity of surrounding material, particle and dose rates at different places in the ATLAS cavern, as well as the luminosity delivered by the Large Hadron Collider (LHC).

I have been working with these devices for about three years, and my work focuses on luminosity measurements. This work started with my bachelor's thesis [5], and continued with a CERN Summer Studentship [6]. During that time, various software was developed. Most of the software were single-use applications developed to produce distributions of physical parameters. These applications were usually created by copying one of the already existing applications and modifying it to suit the needs. However, there were some applications which were needed to be used repeatedly. As the existing software was developed by physicists with no special programming skills, its maintainability and performance were rather poor.

This thesis aims to replace all the previous software by a single application, which is easy to maintain and is high in performance. The goal is also to make the application highly scalable and configurable, so it can be easily extended to provide new functionality, and various aspects of its functionality can be changed without a need to edit its code.

Chapter 1

# Context overview

## 1.1 Devices

The purpose of this chapter is to provide an overview of the physical structure of the devices, their functionality, and the physics behind. The chapter is based on and combines information from [1], [7], [3], [8], and [9].

### 1.1.1 Timepix

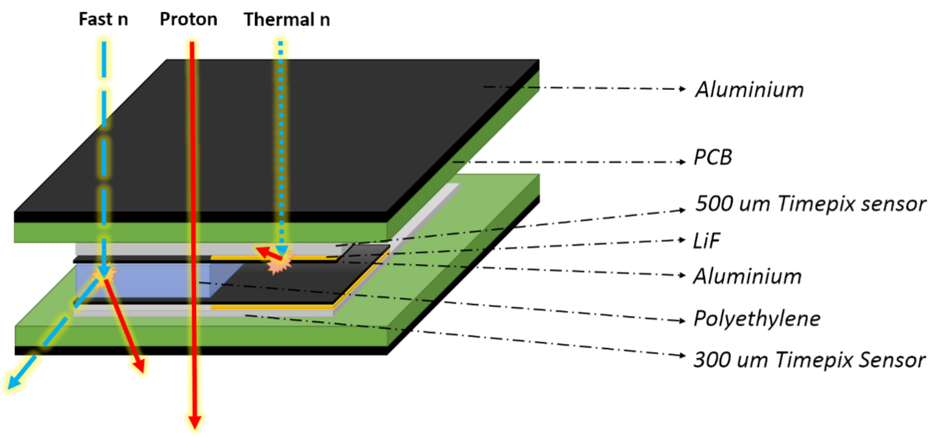


Figure 1.1: Schematics of the TPX device layout. [7, Figure 1(a)]

Sixteen two-layer TPX devices were installed in ATLAS cavern during the second run of LHC (Run-2) which took place from the 3rd of June 2015 [10, 11] until the 3rd of December 2018 [10, 12]. Each of the sixteen TPX devices consists of two hybrid silicon pixel sensors stacked facing each other. In between the sensors, a set of thermal neutron and fast neutron converters is placed. Each of the silicon sensors has a matrix of 256 × 256 pixels of size

55 µm × 55 µm. One sensor has a thickness of 300 µm and the other of 500 µm. The schematics of the device is shown in Figure 1.1. The thinner sensor is referred to as layer-1 and the thicker one as layer-2.

The area of the sensors is divided into four regions defined by the position of the neutron converters. One region is without any neutron converter. Two regions are covered by a 1.2 mm thick layer of polyethylene (PE), and one of them (referred to as PE+Al) has an 80 µm thick aluminum filter inserted below the PE layer. The last region is covered by a $1.6 \pm 0.3\,\mathrm{mg/cm^2}$ (i.e. $6.1 \pm 1.1$ µm) thick lithium-6 fluoride ($^6$LiF) foil with the 80 µm thick aluminum filter over it. The layout of the converters is shown in Figure 1.2.

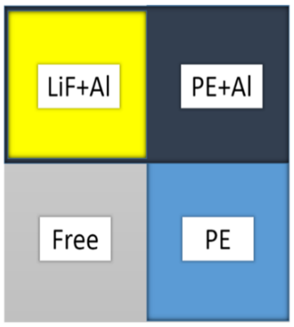


Figure 1.2: Schematics of converter layout in a TPX device. [7, Figure 1(b)]

When ionizing radiation interacts with the active sensor layer, electrons are displaced from atoms. The displaced electrons and the newly created holes are carriers of free charge. These carriers drift in the applied electric field towards the pixel electrodes, where the carriers are collected. During the drift and the collection of the free charge carriers, an electric current is induced at the electrodes of the corresponding pixel. The induced current is observed as an analog signal. The signal is amplified and compared to a global threshold, which is equalized for each pixel independently. If the voltage output signal crosses the threshold, a pulse with a width corresponding to duration when the output voltage remains over the threshold is sent into the logic circuits. The pulse is evaluated in different ways depending on the mode of operation and the result is stored in a 14-bit counter. Each pixel of each sensor can operate independently of each other in one of the following three modes:

- Counting mode — Every pulse increments the counter by one. The width of the pulse is ignored.
- Time-over-threshold (ToT) — The counter starts continuously incrementing with the start of the pulse and it stops increasing with the end of the pulse.
- Time-of-arrival (ToA) — From the moment the pulse starts, the counter is incremented until a global shutter signal is received.

When the global shutter signal is activated, the data acquisition in the whole chip stops and the data from the counters are read out. This effectively groups the data into frames. The period when the chip acquires data is called acquisition time and the period between two frames when the chip does not collect data is called dead-time. The average dead-time of TPX devices is about 90–100 ms [6].

The sixteen TPX devices are spread across the ATLAS cavern. As their locations differ, they are exposed to different conditions and therefore they use different configurations. The devices are labeled as TPX01 to TPX16 and they are being collectively referred to as TPX Network. Their locations are shown in Figure 1.3 and they are precisely described in Table 1.1. Twelve of the sixteen devices are installed in such a way that the sensor surfaces are perpendicular to the beam axis. The exceptions are devices TPX08, TPX09, TPX15, and TPX16. The devices TPX08 and TPX09 are installed horizontally, and the devices TPX15 and TPX16 are installed vertically parallel to the beam axis. [13]

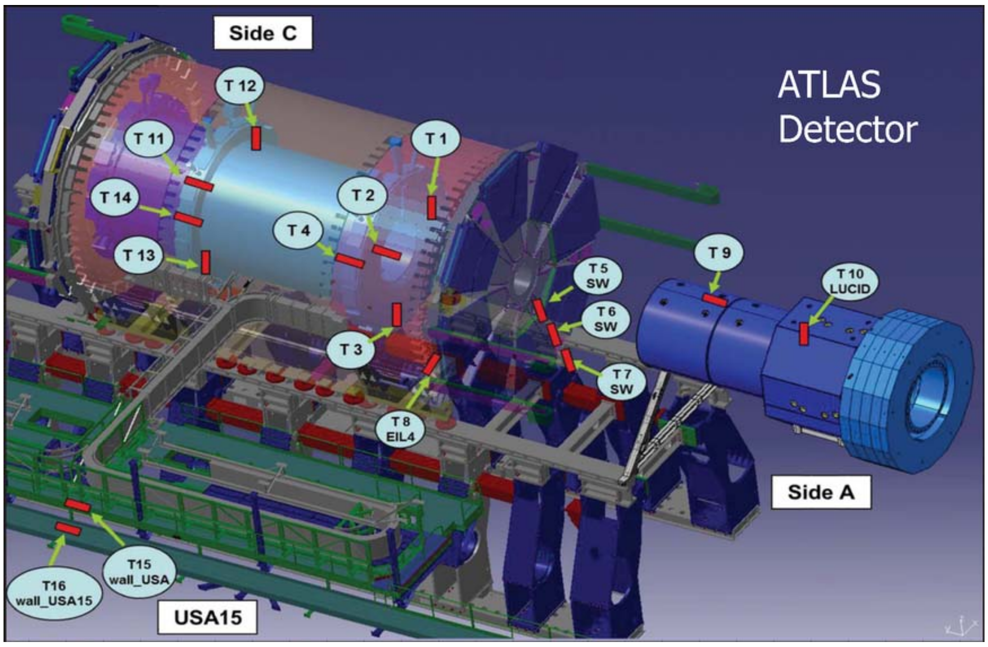


Figure 1.3: Schematics of locations of the TPX devices in the ATLAS cavern. The devices TPX01 to TPX16 are indicated as T1 to T16. [14, Figure 1]

Twelve of the sixteen devices are installed directly in the ATLAS detector. Devices TPX09 and TPX10 are installed on the shielding of the beam pipe. The device TPX15 is installed on the south wall of the ATLAS cavern, dividing the hall UX15 hosting the ATLAS detector from a neighboring service hall USA15. Lastly, the device TPX16 is installed in the service hall USA15.

To make the data consistent, there is a constraint on the setting of the

TPX devices in the ATLAS cavern. All pixels in a single TPX device are set in the same mode. To make things simpler, it is said that a device is set in some mode instead of that pixels of a device are set in some mode. Pixels of different devices are set to different modes to enable us collect more types of data. Twelve of the sixteen devices are set into the ToT mode, i.e. they have all the pixels set into the ToT mode. Devices in ToT mode usually use acquisition times of tenths of a second. Devices TPX02 and TPX12 are set into the hit counting mode. They use acquisition times of one second. Because of this, their frames are purposefully overexposed. This enables to count the hits with much higher precision as the ratio of dead time to active time is minimized. Finally, the devices TPX03 and TPX13 are set into the ToA mode. They use acquisition times of fractions of a millisecond. The same acquisition times are used also by TPX01 and TPX11 which are in the ToT mode. [13]

| device | X [mm] | Y [mm] | Z [mm] | $\rho$ [mm] | R [mm] |
|---|---|---|---|---|---|
| TPX01 | 670 | 880 | 3540 | 1106 | 3709 |
| TPX02 | −1100 | 180 | 3540 | 1115 | 3711 |
| TPX03 | 150 | −1130 | 3540 | 1140 | 3719 |
| TPX04 | −3580 | 970 | 2830 | 3709 | 4665 |
| TPX05 | 1320 | −494 | 7830 | 1409 | 7956 |
| TPX06 | 2370 | −1030 | 7830 | 2584 | 8245 |
| TPX07 | 3300 | −1590 | 7830 | 3663 | 8644 |
| TPX08 | −6140 | 0 | 7220 | 6140 | 9478 |
| TPX09 | 0 | 1560 | 15390 | 1560 | 15469 |
| TPX10 | 230 | 440 | 18859 | 496 | 18857 |
| TPX11 | 660 | 900 | −3540 | 1116 | 3712 |
| TPX12 | −930 | 670 | −3540 | 1146 | 3721 |
| TPX13 | 90 | −1100 | −3540 | 1104 | 3708 |
| TPX14 | −3580 | 970 | −2830 | 3709 | 4665 |
| TPX15 | −16690 | 50 | 5020 | 16690 | 17429 |
| TPX16 | −18900 | 50 | 5020 | 18900 | 19555 |

Table 1.1: Overview of locations of the TPX devices in the ATLAS cavern. All coordinates are relative to the interaction point with an uncertainty of 10 mm. The X-axis is perpendicular to the beam axis in the horizontal plane. The Y-axis is perpendicular to the beam axis in the vertical plane. The Z-axis is equivalent to the beam axis. The $\rho$ is the distance to the beam axis. The R is the overall distance to the interaction point. [15]

As the devices are exposed to a harsh environment, they sometimes stop working and need to be replaced or fixed. There are only six devices that have worked for the whole time and these are TPX02, TPX05, TPX06, TPX07, TPX12, and TPX14. The device TPX10 has never worked as its cables got damaged during the assembly of nearby detectors and the device's position did not allow for its repair. The other detectors have experienced outages in their functionality or have been replaced. The device TPX04 was replaced by a pair of TPX3 devices in 2018.

### 1.1.2 Timepix3

A TPX3 device was developed as the successor of the TPX device. A summary of the differences between TPX and TPX3 is in Table A.1. Four TPX3 devices were installed in ATLAS cavern in January 2018 during the 2017/2018 extended year-end technical stop in addition to the TPX devices which were already there. The four devices were installed in two pairs. The devices installed in a pair are stacked facing each other. Each of the devices has a 500 µm thick sensor with matrix of $256 \times 256$ pixels of size $55\,\mu\text{m} \times 55\,\mu\text{m}$.

An analog signal from pixel is amplified, compensated for current leakage, and compared to a global threshold, which is equalized for each pixel independently. If the signal crosses the threshold, a pulse with a width corresponding to duration when the output voltage remains over the threshold is sent into the logic circuits. The pulse is evaluated using two in different ways depending on the mode of operation and the results are stored in two different counters, one 14-bit counter and one 10-bit counter. Each pixel can operate independently of each other in one of the following three modes:

- ToA/ToT — The rising edge of the pulse starts a 640 MHz clock. The clocks is stopped by the rising edge of a 40 MHz clock. The ToT is measured from this moment on. At the same time, the ToA is measured. The ToT is stored in the 10-bit counter and the ToA is stored in the 14-bit counter.
- Only ToA — Same as the ToA/ToT mode, but only the ToA is measured.
- Event counting — Every pulse increments the 10-bit counter by one. The ToT is measured the same way as in the ToA/ToT mode, but it added to the 14-bit counter where the integrated ToT is stored.

The chip has two different readout modes. The first mode is called a data-driven mode. In this mode, the data are sent off the chip as fast as possible without any external command. When a pixel contains data, it sends a request signal to be read out. The data are shifted into a buffer which is shared by eight pixels. The buffer has storage capacity for data from two pixels. From there, the data are read out by communication done using an asynchronous 2-phase handshake protocol. The dead-time for each pixel is about 475 ns and maximum data rate the device is able to read out is $40\,\text{Mhits/s}\,\text{cm}^2$. The second mode is called a sequential readout mode. In this mode, data stays in the pixel counters until an external readout command is received. Any number of columns can be read out in parallel according to the command.

The TPX3 pairs are labeled TPX3_4 and TPX3_9. In each pair, the device closer to the interaction point has suffix A and the other one has suffix B. The devices are also often referred to by the first part of their chip ID, they are shown in Table 1.2.

The pair TPX3_9 is installed on the eastern wall (A-side) of the ATLAS cavern next to the shielding of the beam pipe and the pair TPX3_4 is located in the Central Barrel of ATLAS where it has replaced the TPX04 device. The locations are shown in Figure 1.4 and they are precisely described in Table 1.2.

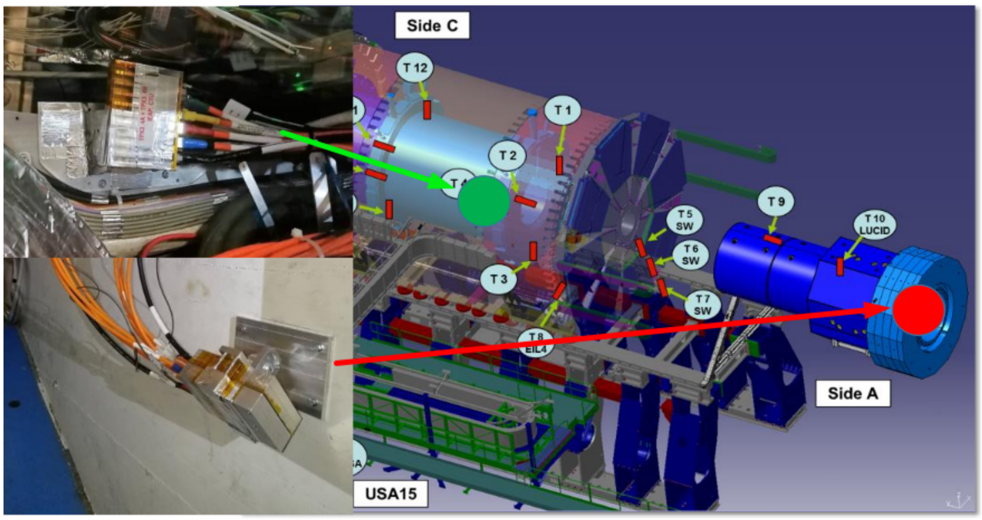


Figure 1.4: Schematics of locations of the TPX3 devices in the ATLAS cavern. The green spot is the pair TPX3_4A and TPX3_4B, and the red spot is the pair TPX3_9A and TPX3_9B. [8, Figure 4]

| device | chip ID | X [mm] | Y [mm] | Z [mm] | $\rho$ [mm] | R [mm] |
|---|---|---|---|---|---|---|
| TPX3_4A | J04-W0036 | −3580 | 970 | 2830 | 3709 | 4665 |
| TPX3_4B | I04-W0036 | −3580 | 970 | 2830 | 3709 | 4665 |
| TPX3_9A | I03-W0036 | 4000 | 3400 | 22900 | 5250 | 23494 |
| TPX3_9B | H03-W0036 | 4000 | 3400 | 22900 | 5250 | 23494 |

Table 1.2: Overview of locations of the TPX3 devices in the ATLAS cavern [9, Table 1]

In each TPX3 pair, the only device with neutron converters is the one further away from the interaction point, that are TPX3_4B and TPX3_9B. The same layout as for TPX devices was used. For comparison, the TPX3 layout is shown in Figure 1.5 and the TPX layout is shown in Figure 1.2.

The TPX3 devices installed in the ATLAS cavern are set in the ToA/ToT operational mode and data-driven readout mode. They are also synchronized with the LHC orbit clock and consequently with each other. The combination of the synchronization and the resolution of 1.5625 ns allows measurements of LHC fill bunch structures (section 1.2). The devices acquire data in 3-hour long periods. They are interrupted by 5–15 s long dead time periods when the devices are reconfigured.

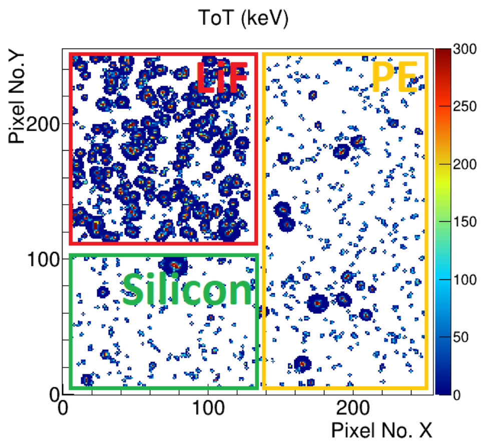


Figure 1.5: Schematics of converter layout in TPX3 devices. [9, Figure 1]

## 1.2 Physics

When the LHC is running, bunch trains are injected into the LHC. The injected bunches usually consists of either protons (hydrogen ions), or lead ions. A consecutive sequence of bunch trains forms a beam. An uninterrupted period when beams are circulated in the LHC machine is called a fill. Each fill has its own unique identification number. The first fill producing stable beams during the LHC Run-2 was fill 3819 [10, 11] and the last one was fill 7492 [10, 12]. The fill number have been four-digits long positive integer. However, it will have to be longer in the future, as Run-3 is being prepared and Run-4 is planned.

An LHC fill can be from a few minutes up to tens of hours long. Because the ATLAS experiment produces a lot of data and the data-taking and processing is costly and consumes a lot of energy, ATLAS is collecting data only when it is triggered. That happens when both beams are stable and ready to be collided. The collisions cause high-voltage ramp up in Pixel, Semiconductor Tracker, and muon system. Once the pixel system is turned on, ATLAS is declared “ready for physics”. Each data-set taken while ATLAS is continuously recording is referred to as an ATLAS run. Because the ATLAS data acquisition system could sometimes interrupt during data-taking, a new ATLAS run is started, and therefore there could be more than one run in a single LHC fill. Similar to LHC fills, each ATLAS run has its own unique identification number, too. The first ATLAS run with stable beams during the LHC Run-2 was run 266904 [11] and the last one was run 367384 [12]. The run number is six-digits long positive integer. [16]

When particles in the opposite beams collide, a shower of other particles is created. Although the particles collide very often, the collisions are actually quite rare relatively to the number of particles in the beams. The efficiency of the collisions is measured as luminosity. That is the collision rate of particles per the size of the cross-section of the beams [17]. For physics purposes, the luminosity is integrated for some well-defined data samples. The sample is called luminosity block (LB) and it is defined by its time. During one LB, instantaneous luminosity, detector and trigger configuration, and data quality conditions are assumed to be constant. In general, duration of one LB is approximately 60 s, however, the duration of an LB is flexible and actions that might alter one of the constant properties mentioned above trigger the start of a new LB before a minute has passed. The boundaries of each LB are defined in real time by the ATLAS Central Trigger Processor [18] during data-taking. [16, 19]

From the number of particles that have hit the sensor area, luminosity can be calculated. There are two different approaches how to do it, one approach calculates so-called absolute luminosity and the other one calculates so-called relative luminosity. The absolute luminosity is calculated [20] by computing a normalization factor from special ATLAS runs used for so-called van-der-Meer scan [21]. The calculation of the relative luminosity is much simpler, the number of particles per LB are normalized to luminosity as measured by another detector. This is usually done by taking the ratio between the number of particles and the reference luminosity as the normalization factor.

When a particle hits the sensor area of a pixel detector, it releases its energy into one or more pixels. The pixels that are hit by the particle constitute a cluster. The clusters are always coherent, that means there cannot be a gap between the pixels. The clusters are categorized into six classes based on their shape. The classes are dot, small blob, curly track, heavy blob, heavy track, and straight track. Examples of these clusters are shown in Table 1.3. A dot consists from only a single pixel, and it is created by low energy photons ($< 20\,\text{keV}$) and electrons. A small blob is a small round cluster, and it is left by x-ray photons ($\sim 50\,\text{keV}$) and electrons. A curly track is a curly non-linear line, and it is left by gamma rays ($> 50\,\text{keV}$) and electrons with energy in orders of MeV. A heavy blob (HB) is round and relatively big cluster, and it is made by heavy ionising particles with short range, such as $\alpha$-particles ($^4$He — helium atom with two neutrons), protons, slow neutrons, etc. A heavy track is a linear and relatively thick line, and it is left by heavy ionising particles, such as protons, ions, etc. A straight track is a linear thin line, and it is created by energetic light charged particles, such as minimum ionizing particles (mips), muons, etc. [7, 22]

Because the number of cluster should be the same as the number of particles which hit the sensor area, luminosity can be calculated from the number of cluster. This approach is called cluster counting. It might happen that the cluster data are not available, then an approach called hit counting can be used. If the device is in hit counting mode, then each particle increments counters of all the pixels it hits. The idea is that the average number of pixels in a cluster is constant. Then the sum of the numbers of hits over all pixels can be used to calculate luminosity, too, as it is larger than the number of clusters by the factor of the average number of pixels in a cluster. Another approach is to filter the clusters by the cluster type, by type of the particle, or by properties of the particle. One of these which is used is the thermal neutron counting, that means that only thermal neutrons are counted, that are neutrons with kinetic energy around 25 meV.

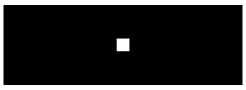
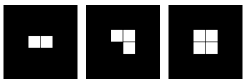
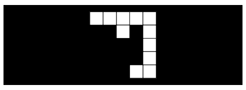
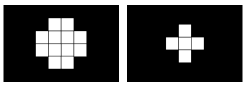
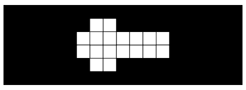
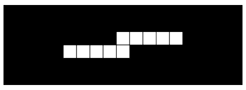
| | | |
|---|---|---|
| 1) Dot |  | Low energy photons and electrons |
| 2) Small blob |  | X-ray photons and electrons |
| 3) Curly track |  | Gamma rays and electrons (MeV) |
| 4) Heavy blob |  | Heavy ionising particles with short range ($\alpha$-particles, protons, ...) |
| 5) Heavy track |  | Heavy ionising particles (protons, ions, ...) |
| 6) Straight track |  | Energetic light charged particles (mips, muons, ...) |

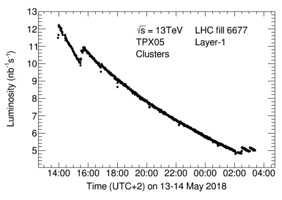
Table 1.3: Overview of cluster types. [7, Figure 2(a)]

When thermal neutrons hit the $^6$LiF converter, they sometimes react with it as $n + {}^6Li \to \alpha + {}^3H$. That means that when the neutron hits the lithium atom, it delivers enough energy so the lithium atom splits into an $\alpha$-particle and triton (hydrogen atom with two neutrons). The neutron is consumed in the process and it is contained in one of the two newly created particles. The particles are then registered by the silicon sensor, and they are observed as HB clusters. The probability that a thermal neutron interacts with the converter and it is detected with the correct signature is $0.475 \pm 0.006\,\%$. [7]

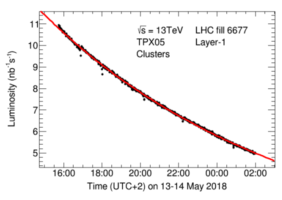
Nonetheless, thermal neutrons are not the only particles leaving behind the HB cluster. There are plenty of other particles that constitute background signal. It might be neutrons with different energy or even different particle altogether. However, the other neutrons and particles interact with silicon in the sensor. This background can be measured by counting the HB clusters in the part of the sensor that is not covered by any neutron converter. The rate of the HB clusters produced by reactions in the $^6$LiF converter is obtained by subtracting the rate of the background from the rate of the HB clusters in the $^6$LiF region. The effect of the neutron converters on a number of HB clusters are shown in Figure 1.5.

### 1.2.1 Luminosity curve

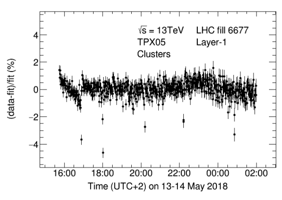
When the beams are colliding over and over again, they gradually loose their intensity as they contain less and less particles. Beam particles can also be lost in collisions with remaining gas in the LHC tube. Due to the decline of intensity, luminosity decreases, too. The luminosity curve describes the development of luminosity in time during a single run. The curve is shown in Figure 1.6(a).



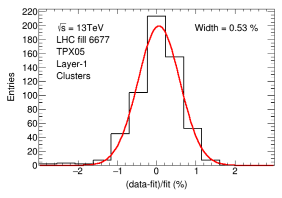
(a) Luminosity curve



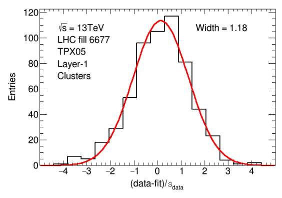
(b) Fitted part of the luminosity curve



(c) Deviations from the fitted curve



(d) Residuals



(e) Pull distribution

Figure 1.6: Distributions in the luminosity curve analysis. The data were acquired by TPX05 layer-1 during LHC fill 6677 and luminosity was estimated using the cluster-counting method. [6]

The analysis is based on data from TPX and TPX3 devices. It is performed to cross-check results of other ATLAS luminometers. A function describing the development of luminosity is fitted onto one or more parts of the luminosity curve. Usually, the function shown in equation 1.1 is used. The value of $\mu(t)$ describes the average number of interactions per bunch-crossing at time $t$, and the value of $\mu_0$ is the average number of interactions per bunch-crossing on the beginning of the run i.e. in time 0. The values of $\lambda_{\mathrm{bb}}$ and $\lambda_{\mathrm{g}}$ describe the rate at which the beams collide with each other or with residual gas in the

LHC tube [23].

$$\mu(t) = \mu_0 \frac{e^{-2\lambda_g t}}{[1 + \frac{\lambda_{bb}}{\lambda_g}(1 - e^{-\lambda_g t})]^2} \tag{1.1}$$

The luminosity curve is divided into parts by small dips in luminosity. These LBs are shorter than the usual 60 seconds. They are caused by LHC tuning of the beams. That is why luminosity raises after the dips. In the ideal case, each part would be fitted separately. However, for technical reasons only a part of the luminosity curve, where the jumps are not significant, were used and it was fitted with a single curve as seen in Figure 1.6(b).

When one has one or more fitted curves, one can use them to determine precision of our measurements. If one subtract the fitted curves from the data and then divide it by the values of the curves, one obtains the relative deviations from the fit. They are shown in Figure 1.6(c). The deviations can be projected to the y-axis and binned to make a histogram. This histogram shows residuals. It has a Gaussian distribution, so when it is fitted with Gaussian function. The mean of the function should be 0. The width (standard deviation) of the function tells the precision of the measurements. A fitted histogram of residuals is shown in Figure 1.6(d).

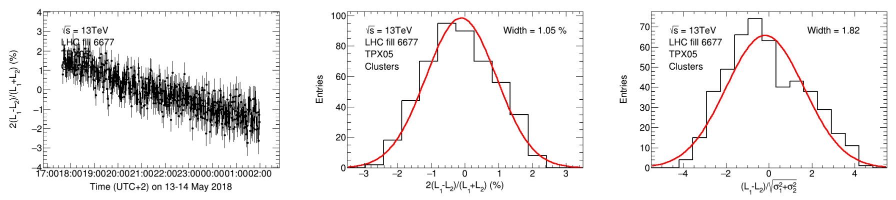
It is good practice to close almost every type of analysis by calculating a pull distribution [24]. It describes whether errors in the measurements are statistical or systematic. For this type of analysis, it is calculated by subtracting the fitted curves from the data and then each data point is divided by its uncertainty. The result is projected to the y-axis. The histogram is then fitted with a Gaussian function. The mean of the function should be 0. The width of the function describes significance of systematic uncertainties. When the width is exactly or very close to 1, the measurement is dominated by statistical uncertainties and that means that there is no space for improvement. However, if the width is significantly greater than 1, the measurement is dominated by systematic uncertainties and that means that there is room for improvement. There could be several reasons which cause the systematic errors. It could be either analysis, software, or the LHC itself. Examples are insufficient description of data by the luminosity curve, saturation effects, or fluctuations in the proton collision rates. An example pull distribution is shown in Figure 1.6(e).

### 1.2.2 Short-term precision

In order to determine the internal consistency of the TPX and TPX3 measurements, a short-term precision analysis is performed. It is measured by calculating the spread of relative differences in luminosity. The formula is in equation 1.2, where $L_1(t)$ and $L_2(t)$ are luminosities from different TPX layers or TPX3 devices in time $t$.

$$\Delta L_{\text{rel}}(t) = 2\frac{L_1(t) - L_2(t)}{L_1(t) + L_2(t)} \tag{1.2}$$

First, the differences are plotted in time. In the ideal case the data are spread around 0, however, it can happen that there is a slope. A slope in this stability plot indicates that the relation between data from the two sources is not linear. This case is shown in Figure 1.7(a).



(a) Internal stability between layers (b) Residuals (c) Pull distribution

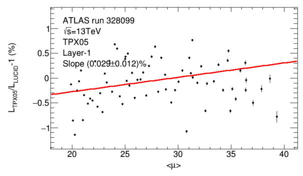
Figure 1.7: Example plots of short-term precision analysis. The compared devices are layers of TPX05. The data were acquired during LHC fill 6677 and luminosity was estimated using the cluster-counting method. [6]

The next step is to project the data to the y-axis. The resulting histogram should have a Gaussian distribution, so, it is fitted with a Gaussian function. The width of the fit describes the spread of the differences. An example of the fitted residual histogram is shown in Figure 1.7(b).

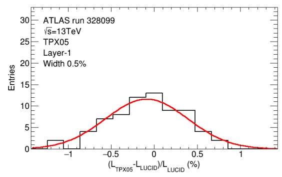
Finally, the pull distribution is calculated. As it is a ratio of data and their uncertainties, it does not need to be calculated for the relative difference but just for the absolute differences. The pull distribution would be the same for both, as there is no new source of uncertainty when the relative differences are calculated from the absolute ones. Because the formula for the absolute differences is $\Delta L_{\text{abs}} = L_1 - L_2$, and $L_1$ and $L_2$ are not correlated, one can calculate the uncertainty as $\sigma_{\text{abs}} = \sqrt{\sigma_1 + \sigma_2}$ using the uncertainty propagation formula. An example pull distribution for short-term precision is shown in Figure 1.7(c).

## 1.2.3 Linearity

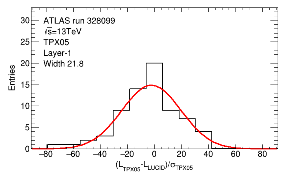
A normalization factor can be used to calculate luminosity only if the measurement is linearly dependent on it. Because of this, linearity analysis is performed. It is similar as the short-term precision analysis. It is done by plotting the difference between luminosity measured by the examined detector and luminosity measured by a reference detector relative to luminosity measured by the reference detector on the y-axis, and $\mu$ on the x-axis where $\mu$ is the average number of collisions per bunch crossing described by equation 1.1. The formula for the values on the y-axis is $\Delta L = \frac{L}{L_{\text{ref}}} - 1$. The plot is then fitted by a linear function. If the slope of the function is small, the measurement is linearly dependent on luminosity, otherwise, the dependence is not linear or the measurement depends also on some other factors. An example of this plot is shown in Figure 1.8(a). The figure shows that there is linearity within $0.029 \pm 0.012\,\%$.



(a) Linearity



(b) Residual



(c) Pull distribution

Figure 1.8: Example plots of linearity analysis. The data were acquired by TPX05 layer-1 during ATLAS run 328099 and luminosity was estimated using the hit-counting method. The reference detector is LUCID.

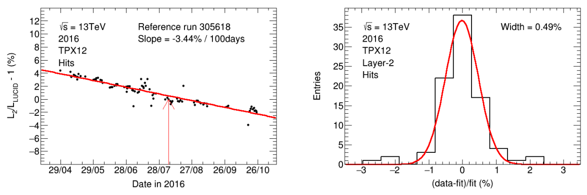
In order to quantify the agreement, the width of residuals is measured. In this analysis, residuals with respect to zero are used, i.e. the data in the linearity plot are projected to the y-axis and binned. The resulting histogram is fitted with a Gaussian function. The width of the function also determines whether the measurement is linearly dependent on luminosity. An example of the residual plot is shown in Figure 1.8(b).

As always, the analysis is closed by production of the pull distribution. For this kind of analysis, it is very simple. Because it is assumed that the reference detector has no uncertainty, therefore, all uncertainties originate in the examined measurement. As the pull distribution is the same for absolute differences as for the relative ones, the absolute differences are used to calculate the distribution. Therefore, the formula is $\frac{L - L_{\text{ref}}}{\sigma}$. An example of the pull distribution is shown in Figure 1.8(c).

The most common detector used as the reference detector is Luminosity Cherenkov Integrating Detector (LUCID) [25] which was developed as the main ATLAS luminometer. There are detectors which have non-linear dependence on luminosity, for example ATLAS Forward Proton (AFP) detectors [26].

### 1.2.4 Long-term stability

The devices in the ATLAS cavern are exposed to extremely harsh conditions due to a large amount of radiation produced by the collisions and the LHC itself. Because of the radiation, the devices take damage over time and their measurement capabilities deteriorate. The closer the devices are to the interaction point, the higher the radiation doses, and the higher the damage. The long-term effects of the damage can be observed with a long-term stability analysis.



(a) Long-term stability (b) Residuals

Figure 1.9: Example plots of long-term stability analysis. The data were acquired by TPX12 layer-2 during 2016 and luminosity was estimated using the hit-counting method. The reference run is indicated by the arrow. [6]

Luminosity is determined from a single device, or in case of TPX a single layer, and it is summed over the whole runs. The same is done for some other detector, device, or layer. The relative differences in the integrated luminosity are plotted. The one relative to which is the difference serves as reference detector. The formula is $\Delta L = \frac{L}{L_{\text{ref}}} - 1$, where $L$ is luminosity measured by the analysed device and $L_{\text{ref}}$ is luminosity measured by the reference detector. The difference is fitted vs time with a linear function and its slope determines the device deterioration relative to the reference detector. The example stability plot is shown in Figure 1.9(a). There is a red arrow, it shows the ATLAS run that was used to calculate the normalization factor.

The spread of the runs around the fitted curve is also measured. For this reason, the relative differences of the data to the fit are calculated and the results are binned. The created residual histogram has a Gaussian distribution and so it is fitted with a corresponding function. The width of the function determines the spread of the data around the fitted line and therefore, how stable is the effect of the radiation damage.

For some detectors, it can happen that the stability plot cannot be fitted with a linear function because the long-term effect of the radiation damage and the resulting degradation is too large and it is no longer linear. This effect was observed for TPX02 and TPX12 in 2017 and 2018. When this is the case, there should be no fit in the stability plot and therefore no residual plot.

## 1.3 Data and processing

All TPX and TPX3 devices installed in ATLAS store the data in text files. The procedure of processing the data and preparing it for use to perform analysis differs for the TPX and TPX3 devices. The TPX3 devices creates a new file every three hours as they are reconfigured. Each file is processed by a noisy pixels detection software and a map of noisy pixels is created. The map is then used when each file is processed by clustering software. The software creates a so-called cluster files and thanks to the map, they do not contain data caused by the noisy pixels. Then, each cluster file is processed by cluster type recognition software and information about the cluster types are inserted into the cluster files.

However, the TPX data are not processed by any noisy pixel detection nor removal software, they are processed only by a clustering software. The software is different than the one processing the TPX3 data. The TPX clustering software runs automatically when the TPX network is acquiring data, and it immediately determines the cluster types. Because of the software's automatic execution, there is no way how to include reliable detection of noisy pixels as the periods between reconfigurations are very long.

TPX and TPX3 cluster files are ROOT files. A ROOT file is a file format defined by the ROOT framework [27, 28], an open-source data analysis framework developed by CERN. The framework is a powerful tool that is used in a data processing and analysis. It is mainly used for its capabilities in I/O, and its easy interface when plotting and fitting graphs and histograms. The ROOT files can store any object whose class inherits from `TObject`.

One of these classes is `TTree`. It is similar to database tables. It consists of branches which are similar to columns in a database table. Each branch has its own type, which could be a primitive type, a class inheriting from `TObject`, STL collection, and more. Entries of a tree are logical equivalents of rows in a database table.

### 1.3.1 Timepix cluster file

The TPX cluster files use three naming conventions, MPX-like, per-day, and per-hour. The oldest convention is the MPX-like, it is inspired by the convention used for the predecessors of TPX, Medipix (MPX). The names of cluster files in this convention are `$Y$M$D_ATLAS$T_$H.root` where `$Y` is a four-digit year, `$M` is a two-digit month, `$D` is a two-digit day of the month, `$T` is a name of a TPX device, and `$H` is a two-digit hour in a 24-hour day. These files store data per single hour. This convention was used until 19th of June 2015. Then the per-day convention was used. The cluster file names in this convention are `$Y_$M_$D_$T.root` where `$Y`, `$M`, and `$D` are year, month and day, and `$T` is a TPX name. These files contain data per day. This convention was used until 29th of September 2016. Since then, the TPX cluster files use the per-hour naming convention. It is very similar to the previous one, the names are `$Y_$M_$D_$T_$H.root` where `$Y`, `$M`, `$D`, and `$T` are year, month, day, and TPX name, and `$H` is an hour.

A TPX cluster file contains three `TTree` objects, `calibData`, `dscData`, and `clusterFile`. These three objects holds all the TPX data necessary for all kinds of analysis. The structure of a TPX cluster file is shown in Figure 1.10.

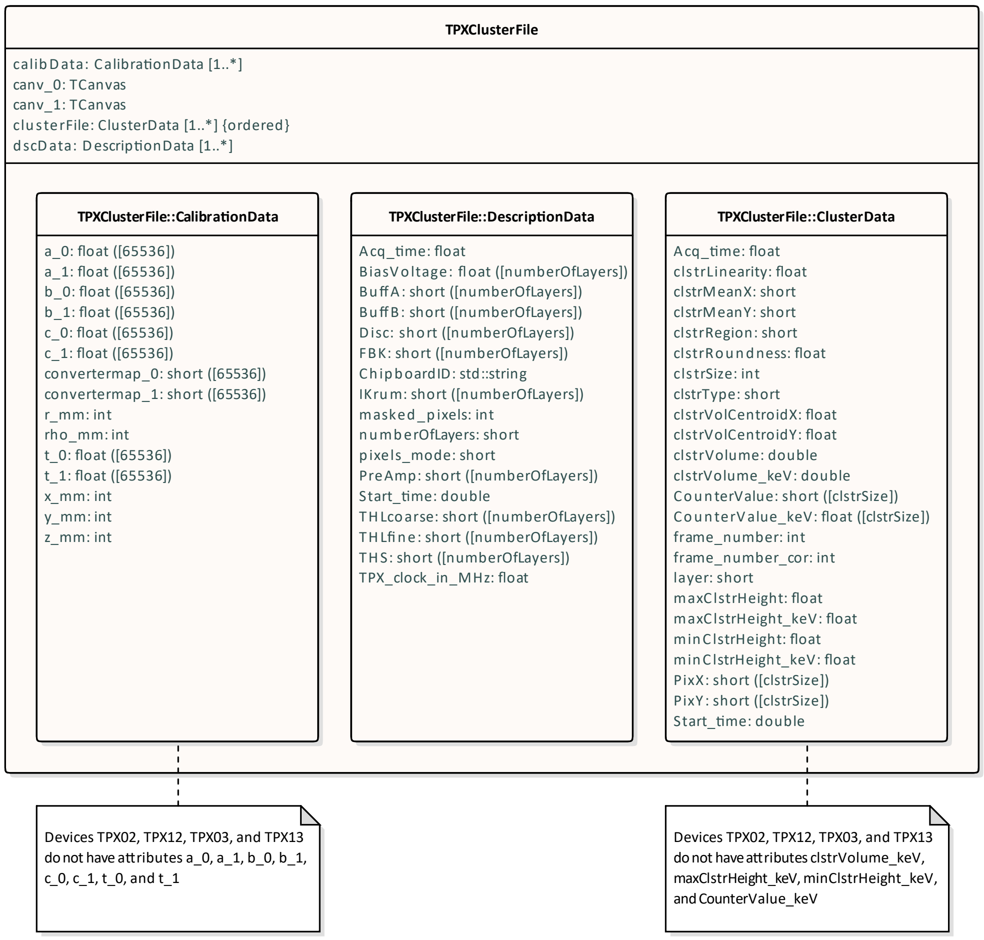

TPXClusterFile
calibData: CalibrationData [1..*]
canv_0: TCanvas
canv_1: TCanvas
clusterFile: ClusterData [1..*] {ordered}
dscData: DescriptionData [1..*]
TPXClusterFile::CalibrationData
a_0: float ([65536])
a_1: float ([65536])
b_0: float ([65536])
b_1: float ([65536])
c_0: float ([65536])
c_1: float ([65536])
convertermap_0: short ([65536])
convertermap_1: short ([65536])
r_mm: int
rho_mm: int
t_0: float ([65536])
t_1: float ([65536])
x_mm: int
y_mm: int
z_mm: int
TPXClusterFile::DescriptionData
Acq_time: float
BiasVoltage: float ([numberOfLayers])
BuffA: short ([numberOfLayers])
BuffB: short ([numberOfLayers])
Disc: short ([numberOfLayers])
FBK: short ([numberOfLayers])
ChipboardID: std::string
IKrum: short ([numberOfLayers])
masked_pixels: int
numberOfLayers: short
pixels_mode: short
PreAmp: short ([numberOfLayers])
Start_time: double
THLcoarse: short ([numberOfLayers])
THLfine: short ([numberOfLayers])
THS: short ([numberOfLayers])
TPX_clock_in_MHz: float
TPXClusterFile::ClusterData
Acq_time: float
clstrLinearity: float
clstrMeanX: short
clstrMeanY: short
clstrRegion: short
clstrRoundness: float
clstrSize: int
clstrType: short
clstrVolCentroidX: float
clstrVolCentroidY: float
clstrVolume: double
clstrVolume_keV: double
CounterValue: short ([clstrSize])
CounterValue_keV: float ([clstrSize])
frame_number: int
frame_number_cor: int
layer: short
maxClstrHeight: float
maxClstrHeight_keV: float
minClstrHeight: float
minClstrHeight_keV: float
PixX: short ([clstrSize])
PixY: short ([clstrSize])
Start_time: double
Devices TPX02, TPX12, TPX03, and TPX13 do not have attributes a_0, a_1, b_0, b_1, c_0, c_1, t_0, and t_1
Devices TPX02, TPX12, TPX03, and TPX13 do not have attributes clstrVolume_keV, maxClstrHeight_keV, minClstrHeight_keV, and CounterValue_keV

Figure 1.10: Structure diagram of a TPX cluster file. Note that files for devices TPX02, TPX12, TPX03, and TPX13 do not contains some of the data.

The `calibData` object contains data which cannot change without physical manipulation with the device as they describe the physical properties of the device. If the tree has more than one entry, all of them should be the same. The branches `x_mm`, `y_mm`, `z_mm`, `rho_mm`, and `r_mm` contain 32-bit integers holding the location of the device in mm as described in Table 1.1. All of the other branches end with either `_0` or `_1` depending on the layer the data are related to. The branches ending `_0` are related to layer-1 and the ones ending `_1` are related to layer-2. The branches starting `a`, `b`, `c`, and `d` contain arrays of 65,536 32-bit floating-point numbers. There are four number for each pixel. These arrays contain the calibration of the device. The numbers are used to calculate energy released by particle from the ToT data. Files for devices that are not in the ToT mode do not contain these four branches.

The branches starting `convertermap` contain arrays of 65,536 16-bit integers. Each element of the array describes in which neutron converter region is the pixel located. The regions are shown in Figure 1.2. Each element can take on values between 0 and 4, where the value of 1 means it is in the uncovered region, the value of 2 represents the PE region, the value of 3 means the PE+Al region, and 4 means the $^6$LiF region. There is also a part of the chip which is referred to as region 0. That is the combination of all of the pixels where the converter region might be ambiguous because there is the physical transition between the converters. The part of the chip is the combination of four one-pixel wide bands on the edges of the chip, and two five-pixels wide bands in between the regions.

Visual representation of the branches starting `a`, `b`, `c`, `d`, and `convertermap` are depicted in `TCanvas` objects `canv_0` and `canv_1` for layer-1 and layer-2, respectively. A `TCanvas` object is simply a canvas containing graphics. Each of these two canvases contains five two-dimensional histograms. Four histograms visualise values of each of the calibration number as a function of the pixel location. If the device is not in the ToT mode, the behavior of the calibration numbers visualisation is undefined. The fifth histogram visualises the neutron converter regions on the chip as defined by the `convertermap` branches.

The `TTree` named `dscData` contains data describing the configuration and settings of the device. The tree has one entry per each frame. The frame can be identified by its timestamp. It is stored in a 64-bit floating-point number in the branch `Start_time`. The timestamp is in the Unix timestamp format. The number uses decimal places to enable time resolution on sub-second level. The length of the frame is stored in the branch `Acq_time`. It contains a 32-bit floating-point number storing the acquisition time in seconds. Another important branch is `pixels_mode`, it contains a 16-bit integer describing the mode the device is in (section 1.1.1). The value of 0 means it is in the hit counting mode, the value of 1 represents the ToT mode, and the value of 3 means the ToA mode. Some of the pixels can get damaged and if they are identified already during the data acquisition, the data they collect can be ignored. Therefore, there are not data for all of the pixels in the file. The number of these pixels is stored in the branch `masked_pixels`. It is stored in a 32-bit integer. The value is usually between 0 and 100. The maximum possible value is 131,072 as it is the number of pixels on both layers together. The branch `ChipboardID` contains a standard C++ string holding unique identification name of the TPX device. The value can change when the device is replaced because of a malfunction or damage. And the branch `numberOfLayers` contains a 16-bit integer holding the number of layers of the device. Its value should be always equal to 2. The other branches describe the settings of the circuits like voltages, thresholds, etc.

The last `TTree`, named `clusterFile` contains the cluster data. Each entry of the tree represents one cluster. Some of the branches describe the shape of the cluster, some its location, some the frame information, and some describe the individual pixels. Also, some of the branches have suffix `_keV`, these branches are present only in files for devices which are in the ToT mode, as they contain information about deposited energy. The individual pixels are described by branches `CounterValue` and `CounterValue_keV`, `PixX` and `PixY`, and `layer`. The `CounterValue` branch contains an array of 16-bit integers storing the value of the counter of each of the pixels in the cluster. The length of the array is stored in the branch `clstrSize`. The information, represented by the counter value, depends on the operation mode of the chip (section 1.1.1). If the chip is in the ToT mode, there is also the `CounterValue_keV` branch which contains a 32-bit floating-point number storing the amount of energy deposited in the pixel in keV. The branches `PixX` and `PixY` contain arrays of 16-bit integers storing the coordinates of the pixels on the chip. The lengths of the arrays are the same as the length of `CounterValue`. The entries in the arrays take on values between 0 and 255 including. The location of the pixels is also described by the `layer` branch. It contains also a 16-bit integer, which holds a value describing in which layer was the cluster detected, and therefore, in which layer are the pixels located. The value is 1 or 2 as it corresponds to layer-1 and layer-2.

The branches describing the shape of the cluster are `minClstrHeight` and `maxClstrHeight`, `clstrSize` and `clstrVolume`, `clstrLinearity` and `clstrRoundness`, and `clstrType`. When the device is in the ToT mode, the branches which describe the shape of the cluster are `minClstrHeight_keV` and `maxClstrHeight_keV`, and `clstrVolume_keV`. The branches `minClstrHeight` and `maxClstrHeight` contain 32-bit floating-point numbers which hold the minimum and maximum values of `CounterValue` in the cluster. If the chip is in the ToT mode, their energy counterparts `minClstrHeight_keV` and `maxClstrHeight_keV` store the values also in 32-bit floating-point numbers. The branch `clstrSize` contains a 32-bit integer storing the number of pixels the cluster consists of. It holds the length of the arrays storing values for individual pixels. Because a cluster must consist from at least one pixel, and the chip has a grid of $256 \times 256$ pixels, the value of the integer is between 1 and 65,536 including. The `clstrVolume` branch is similar to `clstrSize` but instead of counting the number of pixels, it sums values of their `CounterValue`. The value is stored in a 64-bit floating-point number. As expected, it has also the energy measuring counterpart which also stores the value in a 64-bit floating-point number. The `clstrLinearity` branch describes how straight or curly the cluster is and the `clstrRoundness` branch describes how packed or spread the pixels are. They store the values in 32-bit floating-point numbers. The values of linearity are in interval of $(0, 1\rangle$, where the lowest values are usually assigned to curly tracks, and the highest values to straight tracks, small blobs, and dots. The roundness values are in a similar interval, they are in $(0, \sqrt{2\pi}\rangle$, where $\sqrt{2\pi} \approx 2.5$. The highest possible value is assigned to small blobs which consist of two pixels, and the lowest values are usually assigned to straight tracks. A single pixel cluster has both, roundness and linearity equal to one. Finally, the `clstrType` branch describes the category of the cluster as shown in Table 1.3. The value is stored in a 16-bit integer which takes on values between 1 and 6 including. The value of 1 represents a dot cluster, the value of 2 means small blob, 3 means HB cluster, 4 is heavy track, 5 is straight track, and 6 is curly track.

The location of the cluster is described using two different methods. The first method is to use the average of coordinates of each pixel in the cluster. The average coordinates are stored in the `clstrMeanX` and `clstrMeanY` branches. Each of them stores the values in 16-bit integers. The other, more advanced method is to use a weighted average of the coordinates. The values of `CounterValue` are used as weights, so the average equals to the volumetric center which is also the center of energy in the ToT mode. The weighted average coordinates are stored in the `clstrVolCentroidX` and `clstrVolCentroidY` branches. They store the values in 32-bit floating-point numbers. There is also the `clstrRegion` branch which describes the location of the cluster. It stores the neutron converter region in which every pixel in the cluster is located. The regions are shown in Figure 1.2. They are described by a 16-bit integer value. The meanings of the values are the same as for the `convertermap` branches. The values between 1 and 4 are assigned to the cluster when all of its pixels lay in that region. When at least one pixel lays in the region 0, the value of 0 is assigned to the cluster. Note that a single cluster cannot cross two converter regions without crossing the region 0.

Furthermore, the cluster is described by the frame in which the cluster is contained. All the information describing the frame are about time. The branches containing the information are `Start_time`, `frame_number` and `frame_number_cor`, and `Acq_time`. The `Start_time` and `Acq_time` branches are the same as the ones with the same name in `dscData`. The `frame_number` branch contains the index of the frame in the `dscData` tree. The index is stored in a 32-bit integer. Because some frames might be empty, they have no entry in the `clusterFile` tree, and so the index might miss some values. The `frame_number_cor` branch is there to compensate this. It counts only the non-empty frames. It stores the value also in a 32-bit integer. Using these two branches, additional information can be calculated, like the number of empty frames.

### 1.3.2 Timepix3 cluster file

The TPX3 cluster files use naming convention where each file has a name of `$D_$M_$Y_$H_$m_$T_$I.root` where `$D` is a day in a month with no leading zero, `$M` is a name of the month starting with capital letter, `$Y` is a four-digit year, `$H` is a two-digit hour, `$m` is a two-digit minute, `$T` is an identification of the chip, and `$I` is an index of the file in the month. The date and time mark the beginning of the dataset in the file. The identification of the chip, `$T` is the first part of the chip ID (Table 1.2) without the zero.

A TPX3 cluster file contains four `TTree` objects, `InfoTree_layer0` which describes the data acquisition period of the file and the configuration and status of the device, `analysisDescription` and `clusteranalysis_desc` containing parameters of the clustering process, and `clusteredData` holding the cluster data. The file also contains five one-dimensional histograms stored as `TH1F` objects and one canvas stored as `TCanvas` object. The structure of a TPX3 cluster file is shown in Figure 1.11.

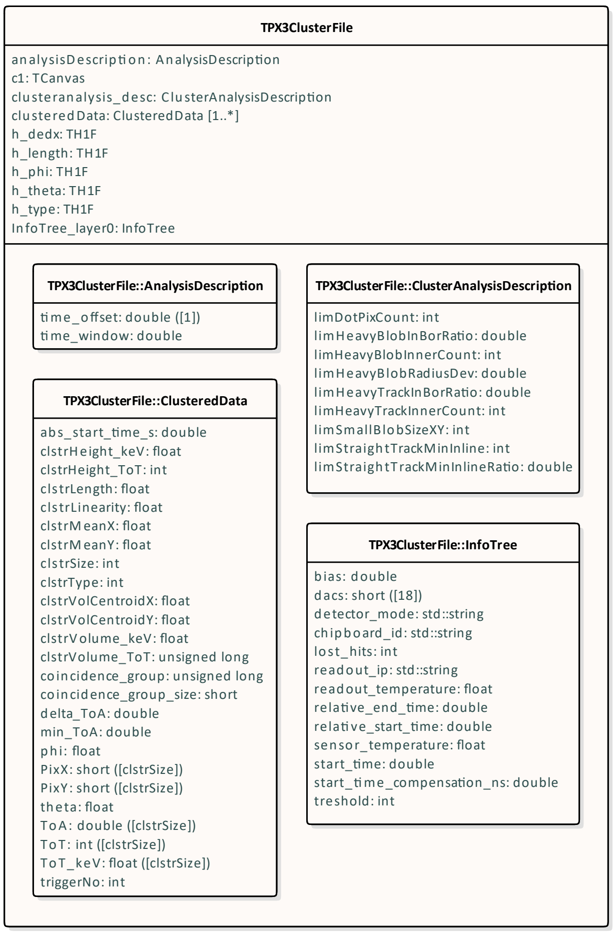

TPX3ClusterFile
analysisDescription: AnalysisDescription
c1: TCanvas
clusteranalysis_desc: ClusterAnalysisDescription
clusteredData: ClusteredData [1..*]
h_dedx: TH1F
h_length: TH1F
h_phi: TH1F
h_theta: TH1F
h_type: TH1F
InfoTree_layer0: InfoTree
TPX3ClusterFile::AnalysisDescription
time_offset: double ([1])
time_window: double
TPX3ClusterFile::ClusterAnalysisDescription
limDotPixCount: int
limHeavyBlobInBorRatio: double
limHeavyBlobInnerCount: int
limHeavyBlobRadiusDev: double
limHeavyTrackInBorRatio: double
limHeavyTrackInnerCount: int
limSmallBlobSizeXY: int
limStraightTrackMinInline: int
limStraightTrackMinInlineRatio: double
TPX3ClusterFile::ClusteredData
abs_start_time_s: double
clstrHeight_keV: float
clstrHeight_ToT: int
clstrLength: float
clstrLinearity: float
clstrMeanX: float
clstrMeanY: float
clstrSize: int
clstrType: int
clstrVolCentroidX: float
clstrVolCentroidY: float
clstrVolume_keV: float
clstrVolume_ToT: unsigned long
coincidence_group: unsigned long
coincidence_group_size: short
delta_ToA: double
min_ToA: double
phi: float
PixX: short ([clstrSize])
PixY: short ([clstrSize])
theta: float
ToA: double ([clstrSize])
ToT: int ([clstrSize])
ToT_keV: float ([clstrSize])
triggerNo: int
TPX3ClusterFile::InfoTree
bias: double
dacs: short ([18])
detector_mode: std::string
chipboard_id: std::string
lost_hits: int
readout_ip: std::string
readout_temperature: float
relative_end_time: double
relative_start_time: double
sensor_temperature: float
start_time: double
start_time_compensation_ns: double
treshold: int

Figure 1.11: Structure diagram of a TPX3 cluster file.

The file contains three hours of data acquisition. The configuration of the device is unchanged in this three-hours period, however, it can change in between the periods. The data acquisition period and the device configuration and status are described by the `InfoTree_layer0` object. The tree has only one entry per file. The unique identification name of the device is stored as a standard C++ string in the branch `chipboard_id`. Because the readouts of the TPX3 devices are connected to the data acquisition server by Ethernet cables, and they use TCP/IP for communication, they have also assigned IP addresses. The address of the readout of the device is stored also as a standard C++ string in the branch `readout_ip`. The mode of the device is also stored as a standard C++ string in the branch `detector_mode`. Because all of the TPX3 devices in the ATLAS cavern were operated in the ToA/ToT mode, the string always contains the value `" ToA & ToT"` followed by a carriage return character. Note that the value starts with a space. The space on the beginning and the carriage return on the end of the string are there because of a bug in parsing values during the cluster file production.

The timestamp of the beginning of the three-hour long data acquisition is stored in the `start_time` branch in a 64-bit floating-point number. The timestamp is in the Unix timestamp format. Because the number of seconds since 1st of January 1970 is 30-bit long between 2004 and 2038, there are 22 bits to store the time in sub-second level. However, these 22 bits make precision of just 238.4 ns whereas the precision of TPX3 is 1.5625 ns. Because of this, there is a need for additional precision. The value of `start_time` uses a resolution of millisecond level and the rest of the required precision is delegated to the `relative_start_time` branch. It also contains a 64-bit floating-point number which stores the start time of the data acquisition relative to `start_time` in seconds. It is usually equal to 50 ns and sometimes to 25 ns. There is a tightly related branch called `start_time_compensation_ns`. The value of its 64-bit floating-point number is usually equal to 0 but when the relative start time is equal to 25 ns, its value is also equal to 25. It is there to correct for a bug in the TPX3 chip which lead to time shifts of 25 ns of columns between 170 and 220 [29]. There is one other related branch, `relative_end_time` which contains the end time of the data acquisition relative to `start_time` in second. The time is also stored in a 64-bit floating-point number. The value is equal to 10,800 s (3 hours) and 50 ns.

As particles hit the chip, part of the energy is released in form of heat. The temperature of the sensor is stored in a 32-bit floating-point number in the branch `sensor_temperature` as degree Celsius. The value is usually around 65 °C. The readout of the device does not get warm from radiation as it is not in the ATLAS experimental cavern UX15 but in the neighbouring service cavern USA15. However, the readout gets warm by itself as every chip under load. The temperature of the readout is stored in a 32-bit floating-point number in the `readout_temperature` as degree Celsius. The value is usually 35–40 °C. The `lost_hits` branch contains a 32-bit integer which is always 0. The other branches describe the settings of the circuits like voltages, thresholds, etc.

The parameters, used during the production of clusters from raw data, are stored in the `analysisDescription` tree. It contains just two branches, `time_window` and `time_offset`, and it has only one entry per file. The `time_window` branch contains a size of a time interval in which all hits must be recorded so they can constitute a cluster. The size of the interval is stored in a 64-bit floating-point number in nanoseconds and its value is always 100 ns. The `time_offset` branch contains one-element long array of 64-bit floating-point numbers with value always equal to 0. It is the delay between the devices in the pair. However, the current clusters were produced for one device at one time, so the value is irrelevant.

The tree `clusteranalysis_desc` contains parameters used by the cluster type recognition software. There are nine parameters in the tree, and the tree has only one entry. The parameters describe the limit values of some of the cluster properties. These are for example: the maximum number of pixels in a dot cluster, the maximal size of small blob cluster, or the minimum number of pixels which have to be in line to make a straight track. Four parameters are stored in 64-bit floating-point numbers, three of them are ratios and one is a deviation. The rest, five parameters are stored in 32-bit integers, and they are counts. The parameters are not relevant anymore once the clusters are categorized.

The most important tree in the file is the `clusteredData` tree. It contains all the data about clusters. Each entry of the tree represents one cluster. The branches in the tree are similar to the ones in a TPX cluster file. The individual pixels in the cluster are described by the branches `Pix` and `PixY`, `ToT` and `ToT_keV`, and `ToA`. All of them contain arrays with length stored in the `clstrSize` branch. The branches `PixX` and `PixY` store the coordinates of the pixels on the chip in 16-bit integers. The timestamp of the hit is stored in `ToA`. It stores the number of nanoseconds from the beginning of the three-hour long data acquisition period. The energy released to the pixel is stored in the `ToT_keV` branch in a 32-bit floating-point number, and raw ToT is stored in a 32-bit integer in the `ToT` branch.

The timing of the cluster is described by the branches `abs_start_time_s`, `min_ToA`, and `delta_ToA`. The `min_ToA` branch contains simply the minimum ToA in the cluster, so it is also stored in a 64-bit floating-point number. The `delta_ToA` branch contains the difference between the maximum and minimum ToA, so it describes the length of the period when the particle was passing through the chip. It is also stored in a 64-bit floating-point number. Finally, the branch `abs_start_time_s` contains the minimum ToA converted to Unix timestamp. Because of this, the value has the issue of precision of only 238.4 ns.

The branches that describe the location of the cluster are `clstrMeanX` and `clstrMeanY`, `clstrVolCentroidX` and `clstrVolCentroidY`. The average coordinates are stored in the `clstrMeanX` and `clstrMeanY` branches. In contrast to the TPX cluster file, they store the value in a 32-bit floating-point number. The weighted average coordinates are stored also in 32-bit floating-point numbers in the branches `clstrVolCentroidX` and `clstrVolCentroidY`. The coordinates are weighted according to the values of ToT.

The branches describing the shape of the clusters are `clstrHeight_ToT` and `clstrHeight_keV`, `clstrVolume_ToT` and `clstrVolume_keV`, `clstrSize`, `clstrLinearity`, `clstrType`, and `phi`. The branches starting `clstrHeight` contain the maximum value of ToT or energy in the cluster. The value of the amount of energy is stored in a 32-bit floating-point number and ToT is stored in a 32-bit integer. The branches starting `clstrVolume` contain the sum of the measured energy or ToT over all pixels in the cluster. The energy is stored in a 32-bit floating-point number and ToT is stored in a 64-bit unsigned integer. The number of the pixels constituting the cluster is stored in the branch `clstrSize`. It stores the number in a 32-bit integer. The branch `clstrLinearity` describes how straight or curly the cluster is. The value is in interval of $\langle 0, 1\rangle$ and it is stored in a 32-bit floating-point number. The highest values are assigned to straight tracks, small blobs, and dots, and the lowest values are assigned to curly tracks. The type of the cluster is described by the `clstrType` branch, as shown in Table 1.3. The value is stored in a 32-bit integer and it is equal to a number between 1 and 6 including. The value of 1 represents a dot cluster, the value of 2 means a small blob, 3 means a curly track, 4 is a HB cluster, 5 is a heavy track, and 6 is a straight track. The branch `phi` contains an angle in degrees stored in a 32-bit floating-point number. It describes the angle of the longest line in the convex hull of the cluster with respect to the x-axis. If the line is parallel to the x-axis, the angle is 0°, and if the line is parallel to the y-axis, the angle is 90°. If all the pixels of the cluster are in only one row or in only one column, the calculation of the angle fails and it is set to 0. Therefore, all dot clusters and all two-pixel small blob clusters have `phi` equal to 0.

The branches named `coincidence_group` and `coincidence_group_size` describe how are clusters in different devices related. If there are clusters captured by different devices in the pair which seem that they could be created by the same particle, it is said that they are coincident. The branch `coincidence_group_size` stores the number of coincident clusters. The clusters in the same coincidence group have the same coincidence group identification number which is stored in the branch `coincidence_group`. The size is stored in a 16-bit integer and the identification number in a 64-bit unsigned integer. If the cluster is not coincident to any other, the group size is equal to 1.

There are also branches that do not fit in neither of the categories. These branches are `theta`, `clstrLength`, and `triggerNo`. The `theta` branch contains the angle under which the particle hit the chip. If the trajectory of the particle is perpendicular to the plane of the chip, the angle is equal to 0°, and if the trajectory of the particle is parallel to the plane of the chip, the angle is equal to 90°. The angle is stored in degrees in a 32-bit floating-point number. Note that the angle never takes on the extreme values as the probability of that happening is negligibly small. The branch `clstrLength` contains the distance which the particle traveled through the chip. It can be calculated as $\frac{300\,\mu\text{m}}{\cos\theta}$, where 300 µm is the thickness of the chip and the value of $\theta$ is stored in the `theta` branch. The length of the cluster is stored in micrometers in a 32-bit floating-point number. The last branch is `triggerNo`. It contains a 32-bit integer which is always equal to 0. The branch is there for legacy reasons and it is no longer used.

The file also contains five `TH1F` objects, they are one-dimensional histograms storing the values as 32-bit floating-point numbers. The histograms are `h_length`, `h_phi`, `h_dedx`, `h_theta`, and `h_type`. The prefix `h_` in the names marks the fact that they are histograms. The histograms `h_phi`, `h_theta`, and `h_type` are overviews of the values of the branches `phi`, `theta`, and `clstrType`. The histogram `h_length` contains information similar to the `clstrLength` but it holds the number of pixels the particle crossed. It can be calculated as $\sin\theta * \frac{\texttt{clstrLength}}{55\,\mu\text{m}}$ or as $\tan\theta * \frac{300\,\mu\text{m}}{55\,\mu\text{m}}$ where the 300 µm is the thickness of the chip and the 55 µm is the size of a pixel. And the histogram `h_dedx` describes how much energy a particle loses per distance it goes through the chip ($\frac{\mathrm{d}E}{\mathrm{d}X}$). The values are in units of keV per µm.

The last object in the file is the `TCanvas` object called `c1`. It basically contains `h_phi` and `h_theta` overlaying each other. They are distinguished by different colors and there is also a legend in the canvas.

### 1.3.3 Timepix3 noisy pixel removal file

The information about the removed noisy pixels in TPX3 devices are stored in noisy pixel removal files or NPR files for short. Its structure is shown in Figure 1.12. The files use the same naming as the cluster files which they are related to.

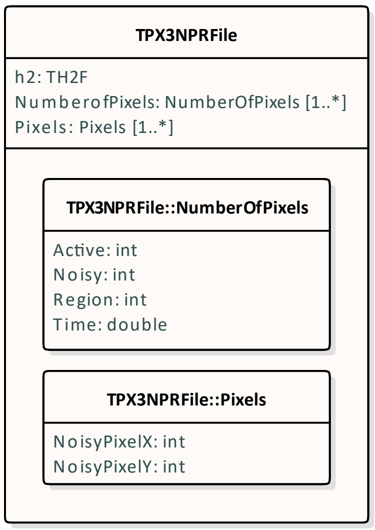


Figure 1.12: Structure diagram of TPX3 NPR file.

The file contains one `TH2F` object, and three `TTree` object. There are two instances of tree named `NumberofPixels`. Each of them has only a single entry. They store how many pixels are in which status. The branch `Active` contains the number of pixels which were used during the clustering, the branch

`Noisy` contains the number of noisy pixels, and the branch `Region` contains the number of pixels in a region 0. The region 0 is different for TPX3 devices than the one for TPX. It consists of four five-pixel bands on the edges of the chip. The devices with the $^{6}$LiF converter also include two bands separating the region from the rest of the chip. The horizontal band is nine-pixel wide and the vertical band is ten-pixel wide. These three branches store the values in 32-bit integers. The last branch, `Time` is present only in the second instance of the tree. It contains the timestamp of the start of the three-hour acquisition period stored in a 64-bit floating-point number.

The other `TTree` object is called `Pixels` and it stores the coordinates of the removed pixels. These are the pixels which are noisy or in the region 0. The coordinated are stored in branches called `NoisyPixelsX` and `NoisyPixelsY` in 32-bit integers.

The last object in the file is the `TH2F` object. It is called `h2` and it is two-dimensional histogram which stores its values in 32-bit floating-point numbers. Each bin of the histogram represent one pixel and they store if they are removed or not. A non-zero value means that the pixel is removed and zero value means that it is not removed.

### 1.3.4 ATLAS reference file

The ATLAS collaboration provides text files containing luminosity as measured by different detectors. We refer to these files as ATLAS reference files. The files are also known as Benedetto's files after the man who was producing them. There is one file per ATLAS run. They contain space separated values divided into lines. The first line gives the names of data stored in the second line. These two lines contain an overview summary of the run. The third line contains the names of data stored in the following lines. Each of them represents one LB. At the beginning of each line, there are a few entries describing the LB and they are followed by series of entries containing luminosity measured by different devices. Files for different years or corrections, used during calculation of luminosity, can contain different number of detectors.

Chapter 2

# As-is

The purpose of this chapter is to analyse and describe the status in which the data processing and analysis software was before the re-engineering.

## 2.1 Processes

In order to analyse the data, the cluster files have to processed and LBs have to be created. The production of LBs is very different depending on the kind of the device which is being processed. It is so because TPX3 cluster files have noisy pixels already removed whereas TPX cluster files not. Because of this, the noisy pixels have to be removed during the production of LBs. When the LBs are produced and the noisy pixels are removed the data can be analysed. The analysis software can work with both, TPX data and TPX3 data, as the analysis is very similar for both devices. The overview of the process of the data processing and analysis, and the data flow is shown in the activity diagram in Figure 2.1.

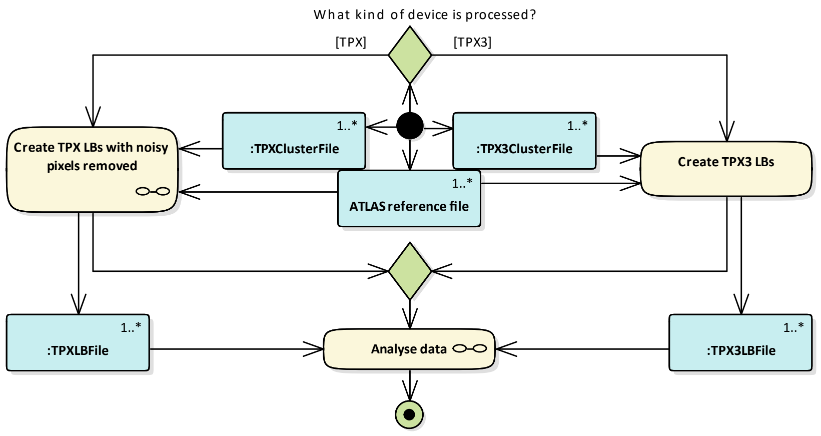


Figure 2.1: Activity diagram of data processing and analysis.

Both, the TPX LB production and the TPX3 LB production uses the same kind of data. The definition of the ATLAS runs and the LBs are read from the ATLAS reference files. The data used to produce the LBs are read from the corresponding cluster file. The result of both productions is a collection of LB files. There is one file per each ATLAS run in the collection.

### 2.1.1 Production of luminosity blocks

The production of LBs for TPX3 is very simple. First, the ATLAS reference files are read in to extract information about runs and individual LBs, and objects, enabling easy reading of all TPX3 cluster files, are instantiated and configured. Then the code starts to iterate over the ATLAS runs. For each run, it creates the output file and the output tree, and starts iterating over the clusters. It does nothing until it finds the first cluster which belongs in that run. Once it was found, the clusters are compared to the LBs. If the cluster belongs in the LB it starts to count the number of the clusters and sum their sizes until it finds the first cluster which does not belong in that LB. Then the information about the LB and the number of hits and clusters is written into the output tree. The next iterations are testing the clusters against the next LB. The loop over clusters ends once there is a cluster that was recorded after the ATLAS run ended. The next run starts to iterate over the clusters from exactly the position where the previous iteration ended. This can be done because both, the ATLAS runs and the clusters are sorted by time. Once the end of the ATLAS run is found, the output tree is written in the output file. The structure of the file is shown in Figure 2.13.

The production of LBs for TPX is more complex because noisy pixels have to be removed. The process consists of four main steps. They are shown in the activity diagram in Figure 2.2. Each activity in the diagram represents one execution of a program. The first step is to create the LBs without any definition of the noisy pixels. The noisy pixels are detected during the creation of the LBs and they are stored in a map (section 2.2.1). There is one map for each ATLAS run. Then, the maps are merged together. The merge produces single map for the whole year. The merge is done by performing a logical OR on the maps, therefore, if a pixel is marked as noisy at least in one ATLAS run, it is marked as noisy also in the final map.

Once the noisy pixel map is produced, the LBs are recreated. This time, the definition of the noisy pixels is in the map. Along the maps, LB files are also produced. However, they are very large and complicated, therefore, a post-processing was introduced to simplify the files and make them similar to the TPX3 LB files. It extracts only the data related to LBs and restores the raw values from the ones modified during the LB creation. These values are the numbers of hits before and after the noisy pixel removal. They are changed to the average number of hits per second during the LB creation, and the absolute number of hits per LB is restored during the post-processing. It also splits every file into two files, one for each layer.

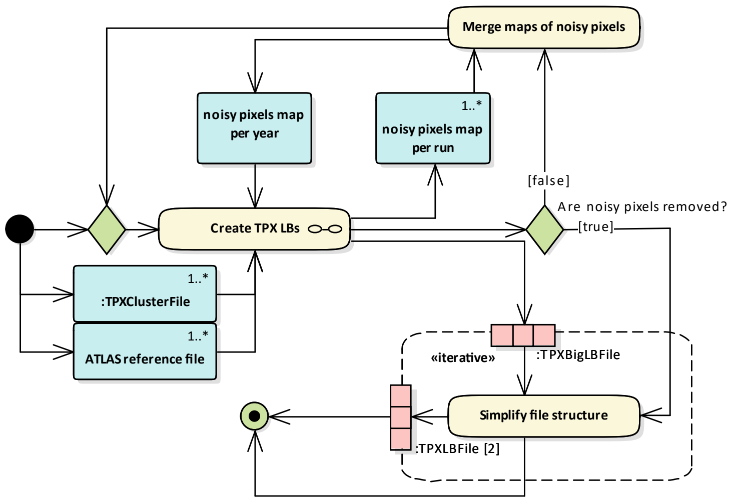


Figure 2.2: Activity diagram of TPX data processing.

The most time-consuming operation is the creation of the LBs, and it is performed twice during the production of the LBs. The activity diagram describes the process of the creation of LBs, shown in Figure 2.3. It starts with loading-in the ATLAS reference files. Then, the program starts to iterate over the ATLAS runs. In each run, a ROOT file is created. It contains the ATLAS reference data. Then, the program starts to iterate over the TPX devices. Each iteration starts with creating the output file and loading-in data from cluster files. Only the files which possibly contain the relevant data are opened. At the beginning of the first file, there are clusters which do not belong to the processed ATLAS run. These clusters are not loaded, however, they are iterated over. Once the beginning of the run is found, the data are loaded into series of arrays of maps and multi-dimensional arrays. Because the maps are indexed by the timestamp of the cluster/frame, there is a lot of unique entries in the maps, and therefore, the complexity of loading the data is $\mathcal{O}(n \log n)$ where $n$ is the number of frames. As there are hundreds of thousands up to a few millions of entries for a single ATLAS run, the loading takes some time.

Once the data are loaded, the noisy pixels need to be detected. It is done by creating forty histograms which plot the number of hits per pixel during the run, and then by fitting the histograms by a Gaussian function. There are four histograms per layer and region including region 0. However, for the actual noisy pixel detection only one of the four histograms is used. Because a noisy pixel accumulates huge numbers of hits, the mean of the Gaussian function is not even close the mean of the values in the histogram. Because of this, the fitting algorithm cannot find the mean of the function, and therefore,

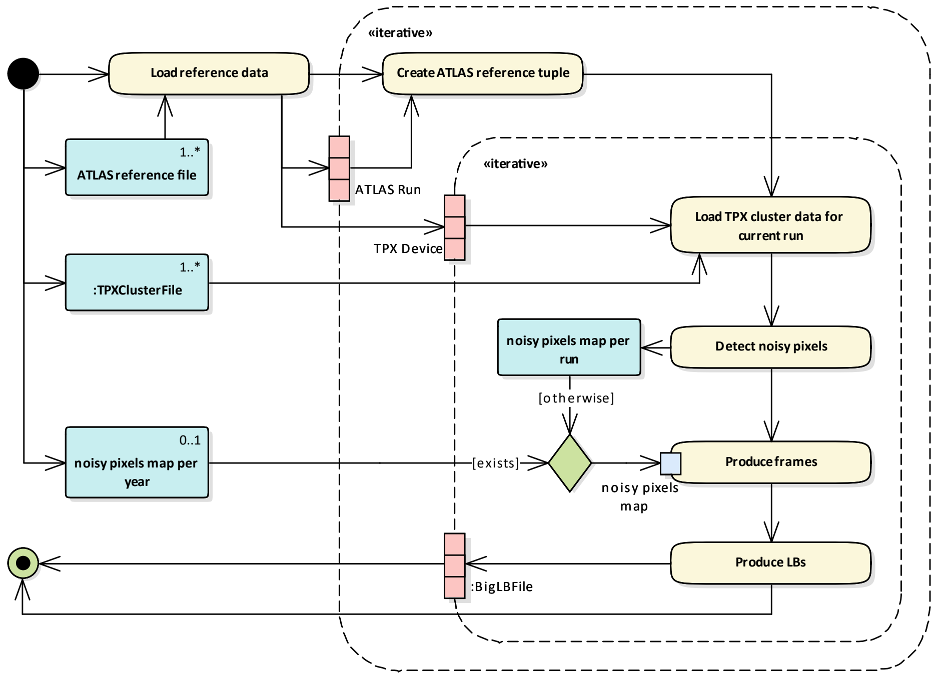


Figure 2.3: Activity diagram of TPX LBs creation.

a median has to be found and used as the initial value of the mean of the function. The algorithm cannot guess even the initial width of the function, so it has to be calculated. The sufficient initial width among the four years is $\frac{\mu}{11.8143}$ where $\mu$ is the mean of the function. This is important to note because it is a linear dependence. Then, the fitting algorithm makes very precise values from these gross estimates. The pixels are marked as noisy if the number of hits is greater than $\mu + n\sigma$ and they are marked as dead if the number is lower than $\mu - n\sigma$. The $\sigma$ is the width of the function, and $n$ is an arbitrary number specifying the strictness of the detection. The number is also referred to as sigma level. Note that higher sigma/strictness level means looser conditions for pixels to be kept. There are also other pixels marked for exclusion, the pixels belonging to the region 0 or to its extension. The extension of the region 0 are four four-pixel wide stripes on the edges of the chip. The stripes extend the region 0 on the edges from one pixel to five pixels. The noisy pixel detection is performed for three different strictness levels, $2\sigma$, $3\sigma$, and $5\sigma$. Everything regarding the noisy pixel detection is stored in the output file. The maps, which mark which pixels should be excluded and which not, are created. There is one map for each combination of ATLAS run, device, layer, and sigma level.

Then, the noisy pixel map for the whole year is loaded. The map which is loaded was created with $5\sigma$ level. If the map is not present, the $3\sigma$ map for the current run is used instead. Then, the program starts to iterate over the layers, frames from the layer, and the individual hits in the frame. It sums all the hits for each frame and stores the numbers in a tree. Every calculation is done in two variants, one removes the noisy pixels, and the other does not. Although, the complexity of iterating through all the frames and hits is linear, it takes quite a long time as there are tens of millions of hits to process in a single ATLAS run.

Only then, finally, the program starts calculating the LBs. It starts to iterate over the layers, LBs, and frames. With each LB it goes through all the frames, twice. In the first run, it calculates the value of the LB without the noisy pixel removal, and in the second run, it calculates the LB without the noisy pixels. Then, the LBs are stored in a tree. Again, the complexity of the cycles is linear but as there are hundreds of LBs and hundreds of thousands up to millions of frames in an ATLAS run, there are tens or even hundreds of millions of iterations to go through, twice, and that also takes a long time.

Processing a single ATLAS run for a single device can take between minutes up to hours, depending on the run and the device. Processing the standard dataset of TPX02, TPX12, TPX05, TPX06, TPX07, and TPX14 for a single year takes few weeks, and few days for a single device. However, the main problem is not the long time it takes but the memory consumption of the program. Because the data are first read in, the allocated memory is more or less equal to the size of the files. The average size of cluster data from a single run, and therefore the required amount of memory is 20–80 GB. Also, the program has to be launched with the maximum stack size of at least 65,536 kB. Because of this large memory requirements, the program had to be executed on special high-memory infrastructure, which required special authorization.

### 2.1.2 Analysis of data

All of the main data analysis code is in a single function. Each analysis has its own branch of conditional compilation created using preprocessor instructions. The program enables to work with different data sources by introducing the conditional compilation branches all across the program. Also, all of the program's configuration is located in a single header file, which conditionally includes other header files based on the type of the data with which the program is supposed to work.

When any configuration changes, the program has to be recompiled. Because of this, the program is recompiled with every launch. It is launched by its makefile. Because of this, the arguments are passed in variables to the makefile, and it uses the preprocessor to hard-code them into the program.

When a special functionality of configuration is required, the program has to be edited a the requirements programmed in. Because the program makes a lot of assumptions, a new code has to be added or an old code has to be commented out in order to make the special adjustments. As these are one-time needs, the program is edited back and forth.

All analyses start by loading the data, and filtering it only to the required subset. Then graphs for different properties are created, and calculation are done on the graphs. The graph are then used to project the data on the y-axis to plot histograms or pull distributions. The histograms are created with fixed number of bins either on the full range of data, or on specified subset. The subsets of ranges are specified also in branches of conditional compilation. Therefore, when a new range is required, it has to be programmed in. The graphs and histograms are often fitted, and these fits are then often used in other calculations.

All calculations and filtering are done immediately. When detector data from `TTree` objects is filtered, the structure of the objects is copied and then the filtered data is added entry by entry. When calculations are done on graphs, the data stored in vectors are copied and modified. The graph has to be copied and iterated over with every operation performed on it.

The graph and histograms are also drawn into canvases. The canvases also draw various labels over the graphs and histograms. When everything is drawn, the canvases are printed into PNG files.

## 2.2 Data structures

The whole program for production of TPX LBs consists of only five classes. Their structures and relationships are shown in Figure 2.4. The five classes are `MakeRootTuple`, `AtlasLumi`, `ActivationCalculator`, `CreateORFile`, and `withPrecision`. The most important object in the program is the instance of `MakeRootTuple`. It is instantiated and its method `MainLoop` is called from the `main` function. It just starts time measurement and calls the `FillHist` method. This method contains almost all the behavior and code executed during the LB creation. The method is over a thousand lines long. The class `MakeRootTuple` inherits from `AtlasLumi`. That is the class which responsibility is to read the ATLAS reference files and parse them. All of its behavior is contained in the `lumiInfo` method. The method is around four hundred lines long with around hundred lines of variable initializations, hundred and fifty lines of input stream reading chains, and hundred lines of variable assignments. The reading chains constitute a hard-coded mapping of the files. An `AtlasLumi` object contains a huge amount of member attributes which are filled during the reading of the files. However, only a small fraction of theses attributes is used. All of them are public and they are accessed directly from the `FillHist` method.

The class `CreateORFiles` is used when the maps of noisy pixels need to be merged. It is instantiated and its only method is called from the `main` function. The method is `processYear` and it contains all the behavior and code executed when merging the maps. It is totally independent of any other class.

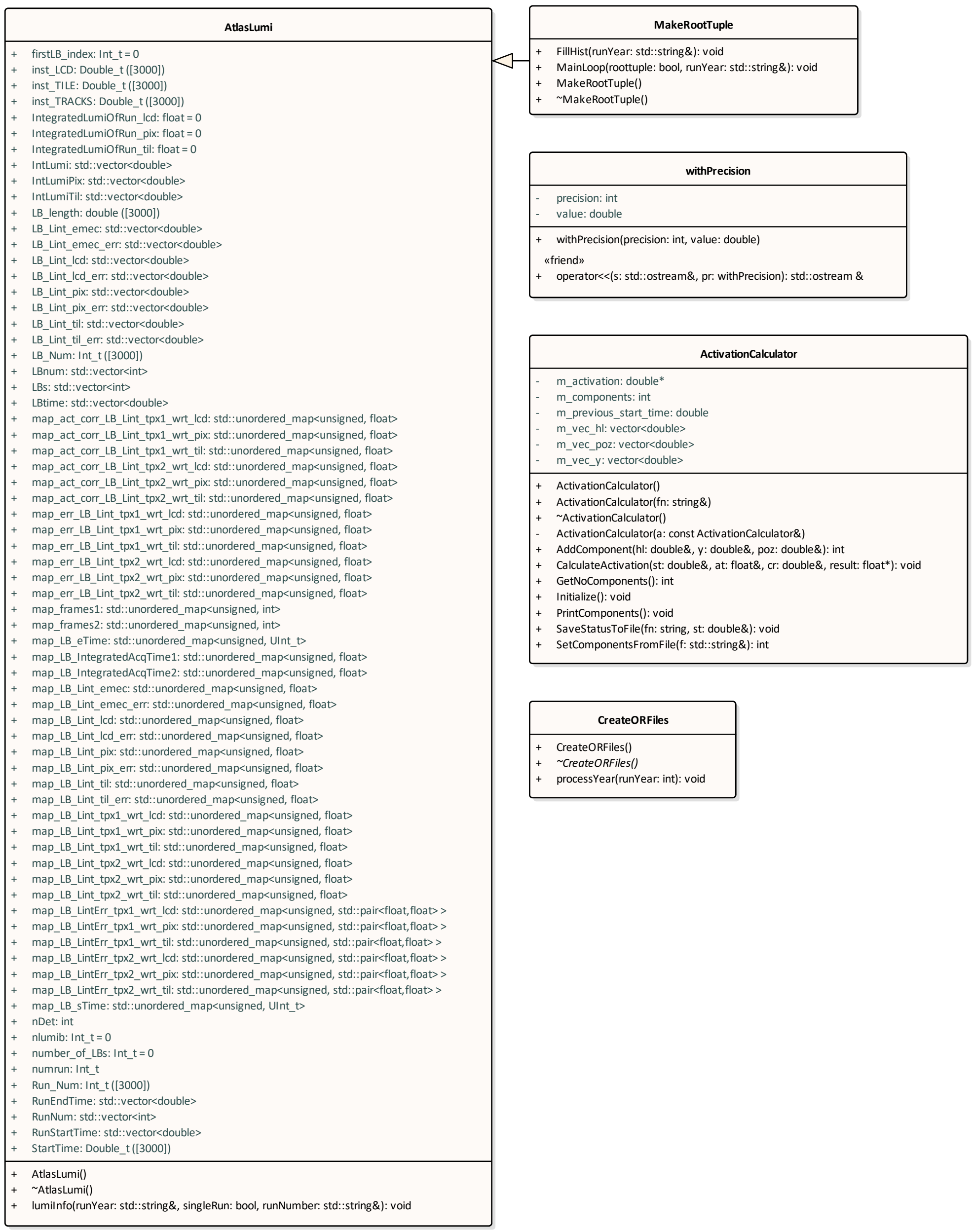

AtlasLumi
+ firstLB_index: Int_t = 0
+ inst_LCD: Double_t ([3000])
+ inst_TILE: Double_t ([3000])
+ inst_TRACKS: Double_t ([3000])
+ IntegratedLumiOfRun_lcd: float = 0
+ IntegratedLumiOfRun_pix: float = 0
+ IntegratedLumiOfRun_til: float = 0
+ IntLumi: std::vector<double>
+ IntLumiPix: std::vector<double>
+ IntLumiTil: std::vector<double>
+ LB_length: double ([3000])
+ LB_Lint_emec: std::vector<double>
+ LB_Lint_emec_err: std::vector<double>
+ LB_Lint_lcd: std::vector<double>
+ LB_Lint_lcd_err: std::vector<double>
+ LB_Lint_pix: std::vector<double>
+ LB_Lint_pix_err: std::vector<double>
+ LB_Lint_til: std::vector<double>
+ LB_Lint_til_err: std::vector<double>
+ LB_Num: Int_t ([3000])
+ LBnum: std::vector<int>
+ LBs: std::vector<int>
+ LBtime: std::vector<double>
+ map_act_corr_LB_Lint_tpx1_wrt_lcd: std::unordered_map<unsigned, float>
+ map_act_corr_LB_Lint_tpx1_wrt_pix: std::unordered_map<unsigned, float>
+ map_act_corr_LB_Lint_tpx1_wrt_til: std::unordered_map<unsigned, float>
+ map_act_corr_LB_Lint_tpx2_wrt_lcd: std::unordered_map<unsigned, float>
+ map_act_corr_LB_Lint_tpx2_wrt_pix: std::unordered_map<unsigned, float>
+ map_act_corr_LB_Lint_tpx2_wrt_til: std::unordered_map<unsigned, float>
+ map_err_LB_Lint_tpx1_wrt_lcd: std::unordered_map<unsigned, float>
+ map_err_LB_Lint_tpx1_wrt_pix: std::unordered_map<unsigned, float>
+ map_err_LB_Lint_tpx1_wrt_til: std::unordered_map<unsigned, float>
+ map_err_LB_Lint_tpx2_wrt_lcd: std::unordered_map<unsigned, float>
+ map_err_LB_Lint_tpx2_wrt_pix: std::unordered_map<unsigned, float>
+ map_err_LB_Lint_tpx2_wrt_til: std::unordered_map<unsigned, float>
+ map_frames1: std::unordered_map<unsigned, int>
+ map_frames2: std::unordered_map<unsigned, int>
+ map_LB_eTime: std::unordered_map<unsigned, UInt_t>
+ map_LB_IntegratedAcqTime1: std::unordered_map<unsigned, float>
+ map_LB_IntegratedAcqTime2: std::unordered_map<unsigned, float>
+ map_LB_Lint_emec: std::unordered_map<unsigned, float>
+ map_LB_Lint_emec_err: std::unordered_map<unsigned, float>
+ map_LB_Lint_lcd: std::unordered_map<unsigned, float>
+ map_LB_Lint_lcd_err: std::unordered_map<unsigned, float>
+ map_LB_Lint_pix: std::unordered_map<unsigned, float>
+ map_LB_Lint_pix_err: std::unordered_map<unsigned, float>
+ map_LB_Lint_til: std::unordered_map<unsigned, float>
+ map_LB_Lint_til_err: std::unordered_map<unsigned, float>
+ map_LB_Lint_tpx1_wrt_lcd: std::unordered_map<unsigned, float>
+ map_LB_Lint_tpx1_wrt_pix: std::unordered_map<unsigned, float>
+ map_LB_Lint_tpx1_wrt_til: std::unordered_map<unsigned, float>
+ map_LB_Lint_tpx2_wrt_lcd: std::unordered_map<unsigned, float>
+ map_LB_Lint_tpx2_wrt_pix: std::unordered_map<unsigned, float>
+ map_LB_Lint_tpx2_wrt_til: std::unordered_map<unsigned, float>
+ map_LB_LintErr_tpx1_wrt_lcd: std::unordered_map<unsigned, std::pair<float,float> >
+ map_LB_LintErr_tpx1_wrt_pix: std::unordered_map<unsigned, std::pair<float,float> >
+ map_LB_LintErr_tpx1_wrt_til: std::unordered_map<unsigned, std::pair<float,float> >
+ map_LB_LintErr_tpx2_wrt_lcd: std::unordered_map<unsigned, std::pair<float,float> >
+ map_LB_LintErr_tpx2_wrt_pix: std::unordered_map<unsigned, std::pair<float,float> >
+ map_LB_LintErr_tpx2_wrt_til: std::unordered_map<unsigned, std::pair<float,float> >
+ map_LB_sTime: std::unordered_map<unsigned, UInt_t>
+ nDet: int
+ nlumib: Int_t = 0
+ number_of_LBs: Int_t = 0
+ numrun: Int_t
+ Run_Num: Int_t ([3000])
+ RunEndTime: std::vector<double>
+ RunNum: std::vector<int>
+ RunStartTime: std::vector<double>
+ StartTime: Double_t ([3000])
+ AtlasLumi()
+ ~AtlasLumi()
+ lumiInfo(runYear: std::string&, singleRun: bool, runNumber: std::string&): void
MakeRootTuple
+ FillHist(runYear: std::string&): void
+ MainLoop(roottuple: bool, runYear: std::string&): void
+ MakeRootTuple()
+ ~MakeRootTuple()
withPrecision
- precision: int
- value: double
+ withPrecision(precision: int, value: double)
«friend»
+ operator<<(s: std::ostream&, pr: withPrecision): std::ostream &
ActivationCalculator
- m_activation: double*
- m_components: int
- m_previous_start_time: double
- m_vec_hl: vector<double>
- m_vec_poz: vector<double>
- m_vec_y: vector<double>
+ ActivationCalculator()
+ ActivationCalculator(fn: string&)
+ ~ActivationCalculator()
- ActivationCalculator(a: const ActivationCalculator&)
+ AddComponent(hl: double&, y: double&, poz: double&): int
+ CalculateActivation(st: double&, at: float&, cr: double&, result: float*): void
+ GetNoComponents(): int
+ Initialize(): void
+ PrintComponents(): void
+ SaveStatusToFile(fn: string, st: double&): void
+ SetComponentsFromFile(f: std::string&): int
CreateORFiles
+ CreateORFiles()
+ ~CreateORFiles()
+ processYear(runYear: int): void

Figure 2.4: Class diagram of the program creating TPX LB data.

The class `withPrecision` is a helper class used when a floating point number is printed into an output stream with a specific fixed precision. It stores the value and the precision during the construction of the object. Then, the object uses the them to print the value with desired precision in the fixed mode. The state of the stream is restored after the value is printed.

The last class `ActivationCalculator` was created and prepared for calculations of radiation induced by surrounding materials. It is the only class which implements some division of responsibilities. However, the class is not used because the code, which invokes it, was never finished and so it was commented out.

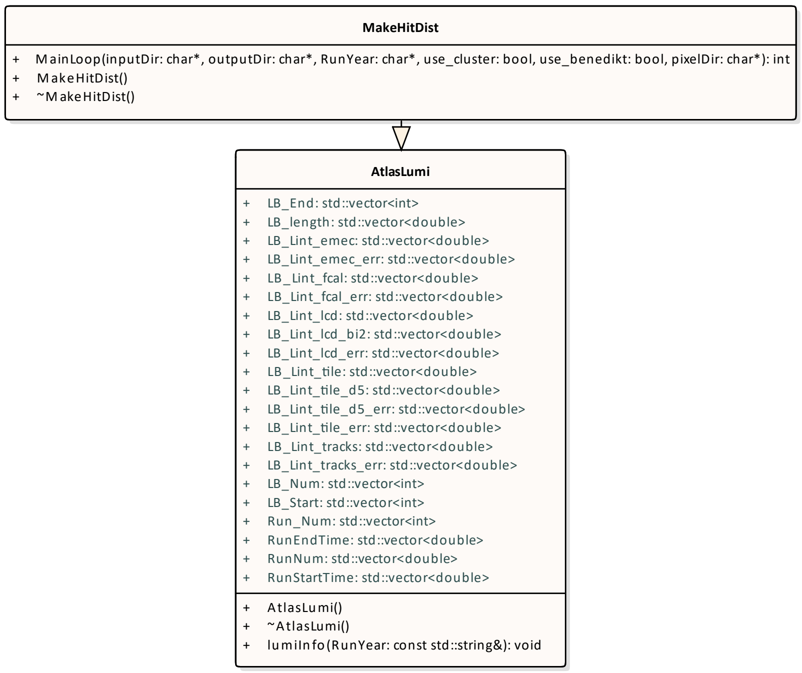
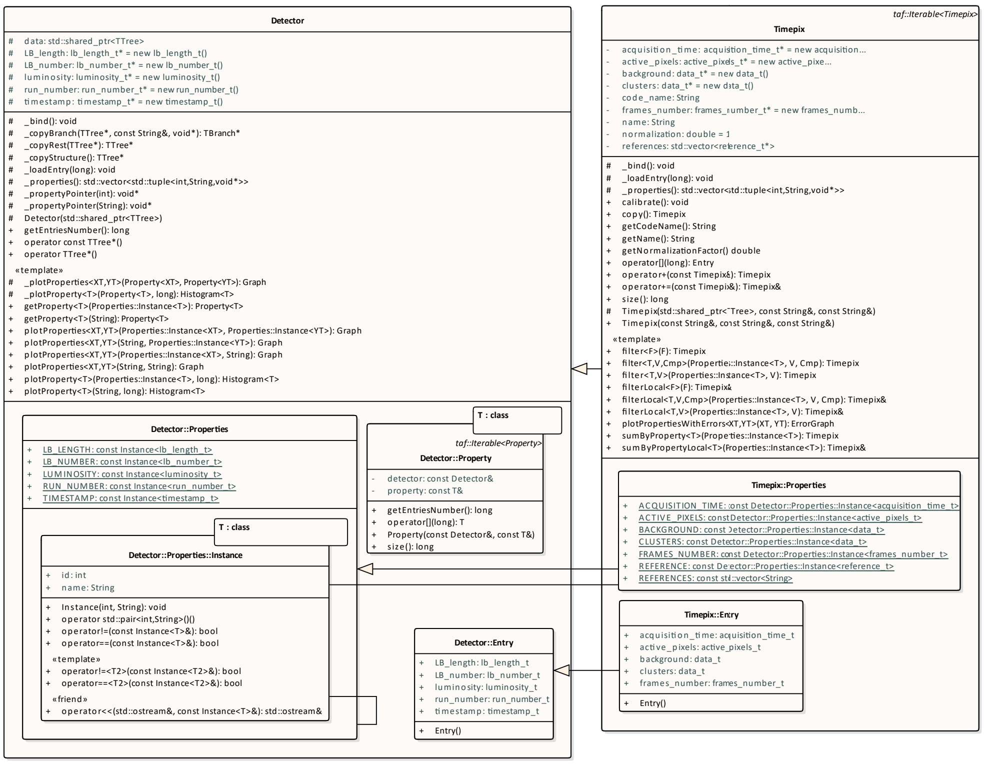

MakeHitDist
+ MainLoop(inputDir: char*, outputDir: char*, RunYear: char*, use_cluster: bool, use_benedikt: bool, pixelDir: char*): int
+ MakeHitDist()
+ ~MakeHitDist()
AtlasLumi
+ LB_End: std::vector<int>
+ LB_length: std::vector<double>
+ LB_Lint_emec: std::vector<double>
+ LB_Lint_emec_err: std::vector<double>
+ LB_Lint_fcal: std::vector<double>
+ LB_Lint_fcal_err: std::vector<double>
+ LB_Lint_lcd: std::vector<double>
+ LB_Lint_lcd_bi2: std::vector<double>
+ LB_Lint_lcd_err: std::vector<double>
+ LB_Lint_tile: std::vector<double>
+ LB_Lint_tile_d5: std::vector<double>
+ LB_Lint_tile_d5_err: std::vector<double>
+ LB_Lint_tile_err: std::vector<double>
+ LB_Lint_tracks: std::vector<double>
+ LB_Lint_tracks_err: std::vector<double>
+ LB_Num: std::vector<int>
+ LB_Start: std::vector<int>
+ Run_Num: std::vector<int>
+ RunEndTime: std::vector<double>
+ RunNum: std::vector<double>
+ RunStartTime: std::vector<double>
+ AtlasLumi()
+ ~AtlasLumi()
+ lumiInfo(RunYear: const std::string&): void

Figure 2.5: Class diagram of the TPX3 LB data production program.

The program for production of TPX3 LBs consists of even lower number of classes. There are only two classes in the whole program. One of them is the `AtlasLumi` class and the other one is `MakeHitDist` which is a parallel to the `MakeRootTuple` class. The structure of the classes is shown in Figure 2.5. The instance of `MakeHitDist` is created and its only method is called from the `main` function. The method is called `MainLoop`. It accepts arguments marking the input and output directories, year of the dataset, type of the cluster files, and directory where information about the noisy pixel removal are stored. The method also contains almost all the behavior and code executed during the LB production. The method is just around two hundred and fifty lines long. It basically contains just two nested loops and bunch of conditional branches. The class inherits from the other one, from `AtlasLumi`. The responsibility of this class is to read and parse the ATLAS reference files. All the behavior is contained in only one method, `lumiInfo`. The method is just around two hundred lines long as it is used just for 2018 data. Also, there are not any member attributes which are not used. This saves not just lines of the `lumiInfo` method but also the memory consumed by a `AtlasLumi` instance. However, the method still contains a chain of input stream reading which constitutes a hard-coded mapping of the reference file. All attributes of `AtlasLumi` are public and they are accessed directly from the `MainLoop` method.

The program for data analysis and plotting the results consists of several classes and templates. The main two classes are the `Detector` class and the `Timepix` class. The structure of these two classes is shown in Figure 2.6.



Detector
# data: std::shared_ptr<TTree>
# LB_length: lb_length_t* = new lb_length_t()
# LB_number: lb_number_t* = new lb_number_t()
# luminosity: luminosity_t* = new luminosity_t()
# run_number: run_number_t* = new run_number_t()
# timestamp: timestamp_t* = new timestamp_t()
# _bind(): void
# _copyBranch(TTree*, const String&, void*): TBranch*
# _copyRest(TTree*): TTree*
# _copyStructure(): TTree*
# _loadEntry(long): void
# _properties(): std::vector<std::tuple<int,String,void*>>
# _propertyPointer(int): void*
# _propertyPointer(String): void*
# Detector(std::shared_ptr<TTree>)
+ getEntriesNumber(): long
+ operator const TTree*()
+ operator TTree*()
«template»
# _plotProperties<XT,YT>(Property<XT>, Property<YT>): Graph
# _plotProperty<T>(Property<T>, long): Histogram<T>
+ getProperty<T>(Properties::Instance<T>): Property<T>
+ getProperty<T>(String): Property<T>
+ plotProperties<XT,YT>(Properties::Instance<XT>, Properties::Instance<YT>): Graph
+ plotProperties<XT,YT>(String, Properties::Instance<YT>): Graph
+ plotProperties<XT,YT>(Properties::Instance<XT>, String): Graph
+ plotProperties<XT,YT>(String, String): Graph
+ plotProperty<T>(Properties::Instance<T>, long): Histogram<T>
+ plotProperty<T>(String, long): Histogram<T>
Detector::Properties
+ LB_LENGTH: const Instance<lb_length_t>
+ LB_NUMBER: const Instance<lb_number_t>
+ LUMINOSITY: const Instance<luminosity_t>
+ RUN_NUMBER: const Instance<run_number_t>
+ TIMESTAMP: const Instance<timestamp_t>
T : class
Detector::Properties::Instance
+ id: int
+ name: String
+ Instance(int, String): void
+ operator std::pair<int,String>()()
+ operator!=(const Instance<T>&): bool
+ operator==(const Instance<T>&): bool
«template»
+ operator!=<T2>(const Instance<T2>&): bool
+ operator==<T2>(const Instance<T2>&): bool
«friend»
+ operator<<(std::ostream&, const Instance<T>&): std::ostream&
T : class
taf::Iterable<Property>
Detector::Property
- detector: const Detector&
- property: const T&
+ getEntriesNumber(): long
+ operator[](long): T
+ Property(const Detector&, const T&)
+ size(): long
Detector::Entry
+ LB_length: lb_length_t
+ LB_number: lb_number_t
+ luminosity: luminosity_t
+ run_number: run_number_t
+ timestamp: timestamp_t
+ Entry()
taf::Iterable<Timepix>
Timepix
- acquisition_time: acquisition_time_t* = new acquisition...
- active_pixels: active_pixels_t* = new active_pixe...
- background: data_t* = new data_t()
- clusters: data_t* = new data_t()
- code_name: String
- frames_number: frames_number_t* = new frames_numb...
- name: String
- normalization: double = 1
- references: std::vector<reference_t*>
# _bind(): void
# _loadEntry(long): void
# _properties(): std::vector<std::tuple<int,String,void*>>
+ calibrate(): void
+ copy(): Timepix
+ getCodeName(): String
+ getName(): String
+ getNormalizationFactor() double
+ operator[](long): Entry
+ operator+(const Timepix&): Timepix
+ operator+=(const Timepix&): Timepix&
+ size(): long
# Timepix(std::shared_ptr<TTree>, const String&, const String&)
+ Timepix(const String&, const String&, const String&)
«template»
+ filter<F>(F): Timepix
+ filter<T,V,Cmp>(Properties::Instance<T>, V, Cmp): Timepix
+ filter<T,V>(Properties::Instance<T>, V): Timepix
+ filterLocal<F>(F): Timepix&
+ filterLocal<T,V,Cmp>(Properties::Instance<T>, V, Cmp): Timepix&
+ filterLocal<T,V>(Properties::Instance<T>, V): Timepix&
+ plotPropertiesWithErrors<XT,YT>(XT, YT): ErrorGraph
+ sumByProperty<T>(Properties::Instance<T>): Timepix
+ sumByPropertyLocal<T>(Properties::Instance<T>): Timepix&
Timepix::Properties
+ ACQUISITION_TIME: const Detector::Properties::Instance<acquisition_time_t>
+ ACTIVE_PIXELS: const Detector::Properties::Instance<active_pixels_t>
+ BACKGROUND: const Detector::Properties::Instance<data_t>
+ CLUSTERS: const Detector::Properties::Instance<data_t>
+ FRAMES_NUMBER: const Detector::Properties::Instance<frames_number_t>
+ REFERENCE: const Detector::Properties::Instance<reference_t>
+ REFERENCES: const std::vector<String>
Timepix::Entry
+ acquisition_time: acquisition_time_t
+ active_pixels: active_pixels_t
+ background: data_t
+ clusters: data_t
+ frames_number: frames_number_t
+ Entry()

Figure 2.6: Structure diagram of the `Detector` and `Timepix` classes in the data analysis program.

The `Detector` class provides an API which enables detectors to read data from an instance of `TTree` and plot properties as histograms or graphs. The properties are indexed by the constants stored in the nested class `Properties`. This class is used only as a namespace. The constants are instances of more nested template `Instance`. It enables the constants to hold a name and an id. The property can be also retrieved from the detector. The obtained object is instance of the nested template `Property` with the template parameter bound according to the property `Instance`. The detector defines properties which are common for all ATLAS luminometers.

The `Timepix` class represents both, TPX and TPX3 devices. It reads LBs from all files which are located in the directory passed to the `Timepix` constructor. It also enables to filter the LBs according to values of their properties. The `Timepix` class further extends the collection of properties. Some of the properties are conditioned to the conditional compilation. The class also enables to iterate over its entries, and to retrieve the LB entries using a call of `operator[]`.

When the properties are plotted, an instance of either `Graph`, `ErrorGraph`, or `Histogram` is created, depending on the invoked method. The structure of these classes and other classes used for plotting is shown in Figure 2.7. The `Histogram` template is used only to fit and to plot a histogram. Its template parameter determines which instance of the ROOT class `TH1` is used.

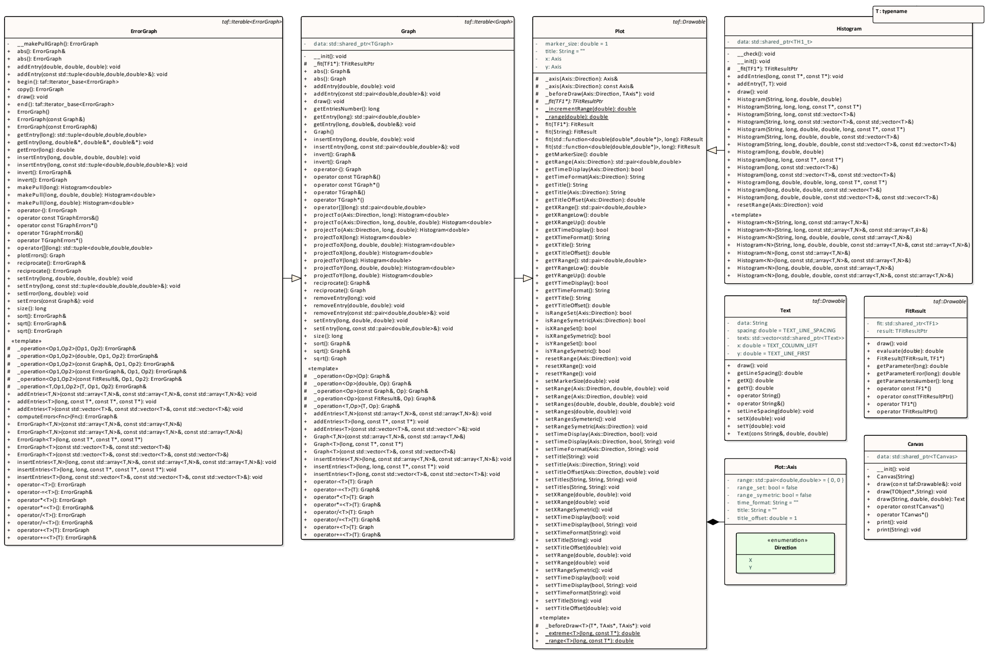


Figure 2.7: Structure diagram of plotting classes in the data analysis program.

The `Graph` class is used to fit and to plot a graph without any error bars. It also enables to be used in various calculations, to manipulate with its entries, and to project the entries on one of the axes to produce a histogram. A graph can be combined with other graphs and with fit results. When a graph is used in a calculation, the calculation is performed for every entry and they are stored in a new graph instance. This is performed for every operation in the calculation.

The `ErrorGraph` class is used to fit and to plot a graph with error bars for the Y values. It is a child class of the `Graph` class, therefore, it can be also used for the same purposes. The `ErrorGraph` class also enables to create a histogram storing the pull distribution.

All of these classes are descendants of the abstract class `Plot`. This class enables to set the ranges of the axes, as well as to set various traits which change the appearance of the plot when it is drawn into a canvas. A canvas is instance of the `Canvas` class. It used to draw plots and text labels, and to print itself into a PNG, PDF, or any other image file. When a plot is printed, it resets the canvas, therefore, the text labels have to be printed later over the plots. The labels are instances of the `Text` class. It is used to bind a text with its coordinates, and its line spacing.

All of the classes use custom class representing string. The class is the `String` class, and its structure is show in Figure 2.8. The class is a wrapper around the ROOT class `TString`. It purpose is to enable use any of `TString`, `std::string`, and C string interchangeably, and to simplify text manipulation and creation. The `String` class can turn any value of any printable type to its instance. It can be created using a list initialization, where all elements of the list are joined into a single string. The class can also create a string by joining a series of values and separating them by a delimiter. And it can also create a string using formatting which is used by the `printf` family of functions.

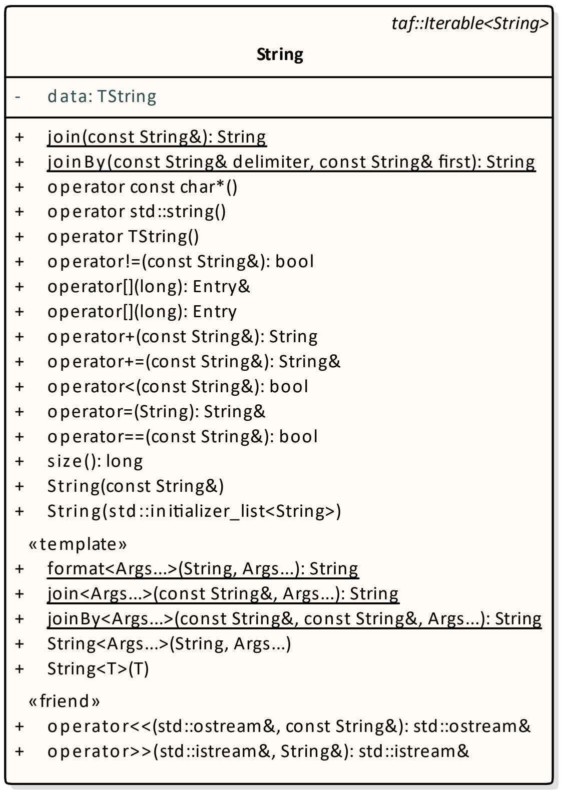

taf::Iterable<String>
String
- data: TString
+ join(const String&): String
+ joinBy(const String& delimiter, const String& first): String
+ operator const char*()
+ operator std::string()
+ operator TString()
+ operator!=(const String&): bool
+ operator[](long): Entry&
+ operator[](long): Entry
+ operator+(const String&): String
+ operator+=(const String&): String&
+ operator<(const String&): bool
+ operator=(String): String&
+ operator==(const String&): bool
+ size(): long
+ String(const String&)
+ String(std::initializer_list<String>)
«template»
+ format<Args...>(String, Args...): String
+ join<Args...>(const String&, Args...): String
+ joinBy<Args...>(const String&, const String&, Args...): String
+ String<Args...>(String, Args...)
+ String<T>(T)
«friend»
+ operator<<(std::ostream&, const String&): std::ostream&
+ operator>>(std::istream&, String&): std::istream&

Figure 2.8: Structure diagram of the `String` class.

Some of the classes like `Plot`, `Text`, or `FitResult` implement the interface `taf::Drawable`. This interface marks the fact that instances of the implementing classes can be drawn using the `draw` method. Some other classes like `Timepix`, `Property Graph`, `ErrorGraph`, or `String` inherit from the `taf::Iterable` mixin. The structure of this mixin and its related template `taf::Iterator_base` are shown in Figure 2.9.

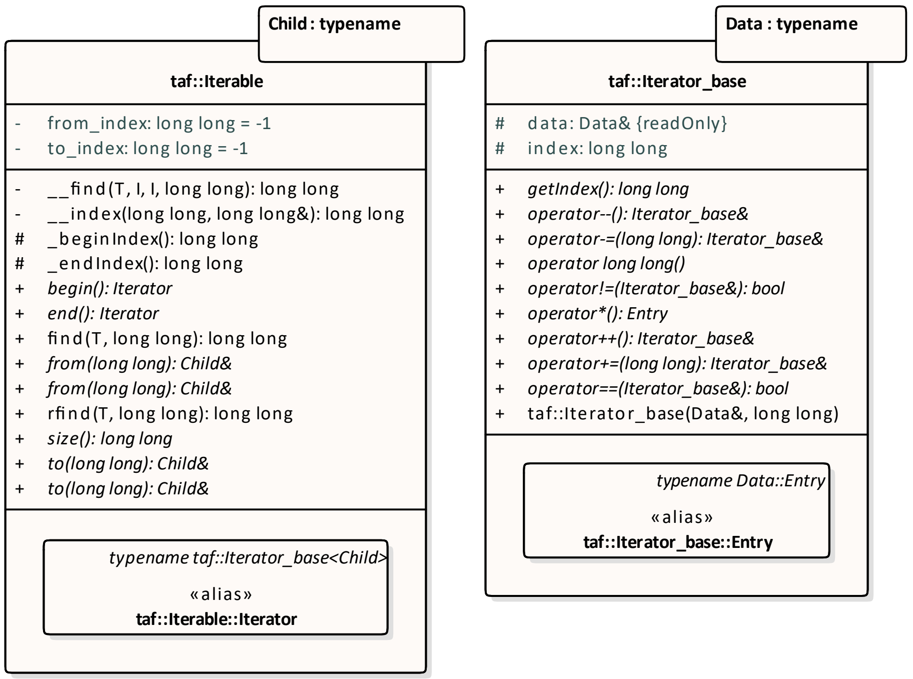

Child : typename
taf::Iterable
- from_index: long long = -1
- to_index: long long = -1
- __find(T, I, I, long long): long long
- __index(long long, long long&): long long
# _beginIndex(): long long
# _endIndex(): long long
+ begin(): Iterator
+ end(): Iterator
+ find(T, long long): long long
+ from(long long): Child&
+ from(long long): Child&
+ rfind(T, long long): long long
+ size(): long long
+ to(long long): Child&
+ to(long long): Child&
typename taf::Iterator_base<Child>
«alias»
taf::Iterable::Iterator
Data : typename
taf::Iterator_base
# data: Data& {readOnly}
# index: long long
+ getIndex(): long long
+ operator--(): Iterator_base&
+ operator-=(long long): Iterator_base&
+ operator long long()
+ operator!=(Iterator_base&): bool
+ operator*(): Entry
+ operator++(): Iterator_base&
+ operator+=(long long): Iterator_base&
+ operator==(Iterator_base&): bool
+ taf::Iterator_base(Data&, long long)
typename Data::Entry
«alias»
taf::Iterator_base::Entry

Figure 2.9: Structure diagram of the `taf::Iterable` mixin and the template `taf::Iterator_base`.

The `taf::Iterable` mixin adds the ability to iterate over to its child classes. The only precondition for this mixin is that the instances of the child have the `operator[]` and the method `size`. The mixin also enables to iterate over only a subset of entries in the child object. It uses the instances of the `taf::Iterator_base` template as the iterators. The instances have the template parameter bound to the child class. The iterator can move in both directions by any amount of entries, and it retrieves entries from the child object using the `operator[]`. However, because of its erroneous implementation, it does not meet the requirements for an iterator according to the C++ standard.

### 2.2.1 File structures

The first produced file is a map of noisy pixels. It is a text file which contains space separated zeros and ones. There is 65,536 entries in the file. Each entry represents one pixel. The value of one means that the pixel is marked as noisy, and the zero value means it is not noisy. The maps for individual runs are stored in a directory `excludedPixels` and the name of each map is `excluded_pixels_$D_L$L_run$R_${S}sigma.dat` where `$D` is the name of a TPX device, `$L` is a number of a TPX layer, `$R` is an ATLAS run number, and `${S}` is a sigma level. The merged maps for the whole years are stored in a directories which names are the years. The directories are in a directory `noisyPixels`. The name of each merged map is `noisy_pixels_$D_L$L_${S}sigma.dat`.

The first LB file is created during the production of TPX LBs. However, it contains no TPX data, instead it contains some of the ATLAS reference data. The structure of the file is shown in Figure 2.10. The files are stored in a directory `root` and the name of each file is `TPX_hits_$R_ATLAS.root` where `$R` is an ATLAS run number.

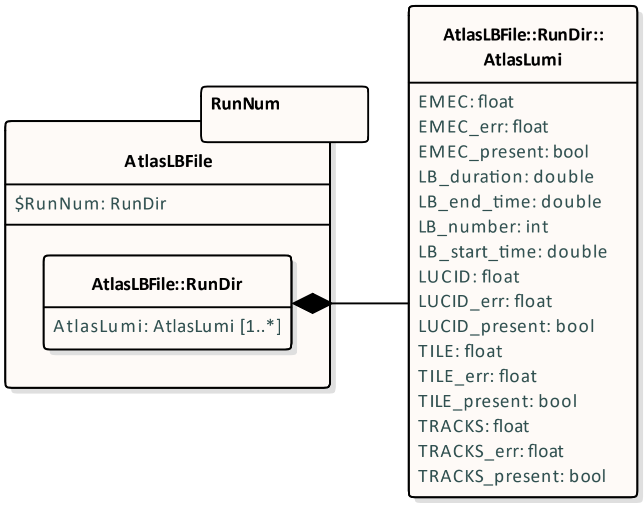


Figure 2.10: Structure diagram of an ATLAS LB File. Variable `RunNum` is used as a name of a directory. The dollar sign sigil syntax is used to mark a string substitution.

The file contains a single `TTree` object contained in a directory. The name of the directory is equal to the number of the ATLAS run, LBs of which the data describe. The `TTree` object is called `AtlasLumi`, and it contains sixteen branches. Four branches describe the LBs and the twelve branches are related to other ATLAS detectors. The timestamp of the start of the LB is stored in the branch `LB_start_time`, and the timestamp of the end of the LB is stored in the `LB_end_time` branch. The difference of these two timestamp is the duration of the LB, which is stored in the branch `LB_duration`. All of these three branches store their values in 64-bit floating-point numbers. The branch `LB_number` stores the identification number of the LB in a 32-bit integer.

The other twelve branches can be split into four group by three branches. Each group contains one branch containing estimated luminosity. The name of the branch is the name of the reference data. Each group also contains one branch storing the uncertainty of the luminosity. The name of this branch is the name of the reference data with suffix of `_err`. Both, the luminosity and the uncertainty branches store their values in 32-bit floating-point numbers. The last branch in the group stores whether the detector contains any data for the current LB. The name of the branch is the name of the reference data with suffix of `_present`. The branch stores the information in a Boolean value. The four groups are for reference data `EMEC`, `LUCID`, `TILE`, and `TRACKS`. The `EMEC` data come from the Electromagnetic Endcap (EMEC), the `LUCID` data come from LUCID, the `TILE` data come from the tile calorimeter, and the `TRACKS` data come from track counting in the ATLAS Inner Detector [4, 19].

The first TPX LB file is created during the production of LBs. The structure of the file is show in Figure 2.11. The files are also stored in the directory `root` and the name of each file is `TPX_hits_$R_$D.root` where `$R` is an ATLAS run number, and `$D` is a name of a TPX device.

The file contains objects stored in a structure of directories. The only thing the root directory contains is a directory, name of which is a number of an ATLAS run. This directory also contain only one directory, name of which is a name of a TPX device. This directory contains two `TTree` objects, two directories, and eighty `TH1D` objects. There are eight `TH1D` objects per each converter region and each TPX layer. Name of each `TH1D` object in the directory contains both, the number of the ATLAS run, and the name of the TPX device from which the data originate. The objects with name starting `hitsDist` are the histograms created during the noisy pixel detection. They store the distribution of number of hits. As the class name suggests, the values of the histograms are stored in 64-bit floating-point numbers. Note that the forty histograms store also the fitted Gaussian functions. The other forty histograms were also created during the noisy pixel detection. They store the information how many time was each pixel hit. The data are also stored in 64-bit floating-point number. The names of these histograms start with `hitsPix`.

The two `TTree` objects are `summedHits` and `fitResults`. The first tree, `summedHits` stores the values used to create the histograms. All of its data are stored in a single entry. The summed numbers of hits per each pixel are stored in the branch `summed_hits_reg`. They are stored in a three-dimensional array of 32-bit floating-point numbers. The dimensions separate the data by the TPX layer, the converter region, and the pixel. The `individual_hits_reg` branch stores the number of pixels which were hit during a single frame. It is similar to the summed hits but it ignores values of pixel counters. The hits are also stored in a three-dimensional array of 32-bit floating-point numbers. The last branch in the tree is the `IntLumi` branch. It stores integrated luminosity of the run which was read from the ATLAS reference files.

The second tree is `fitResults`. It stores the parameter values of the Gaussian functions used for fitting the histograms. The values are results of the fitting. All of them are stored in a single tree entry. The tree contains eight

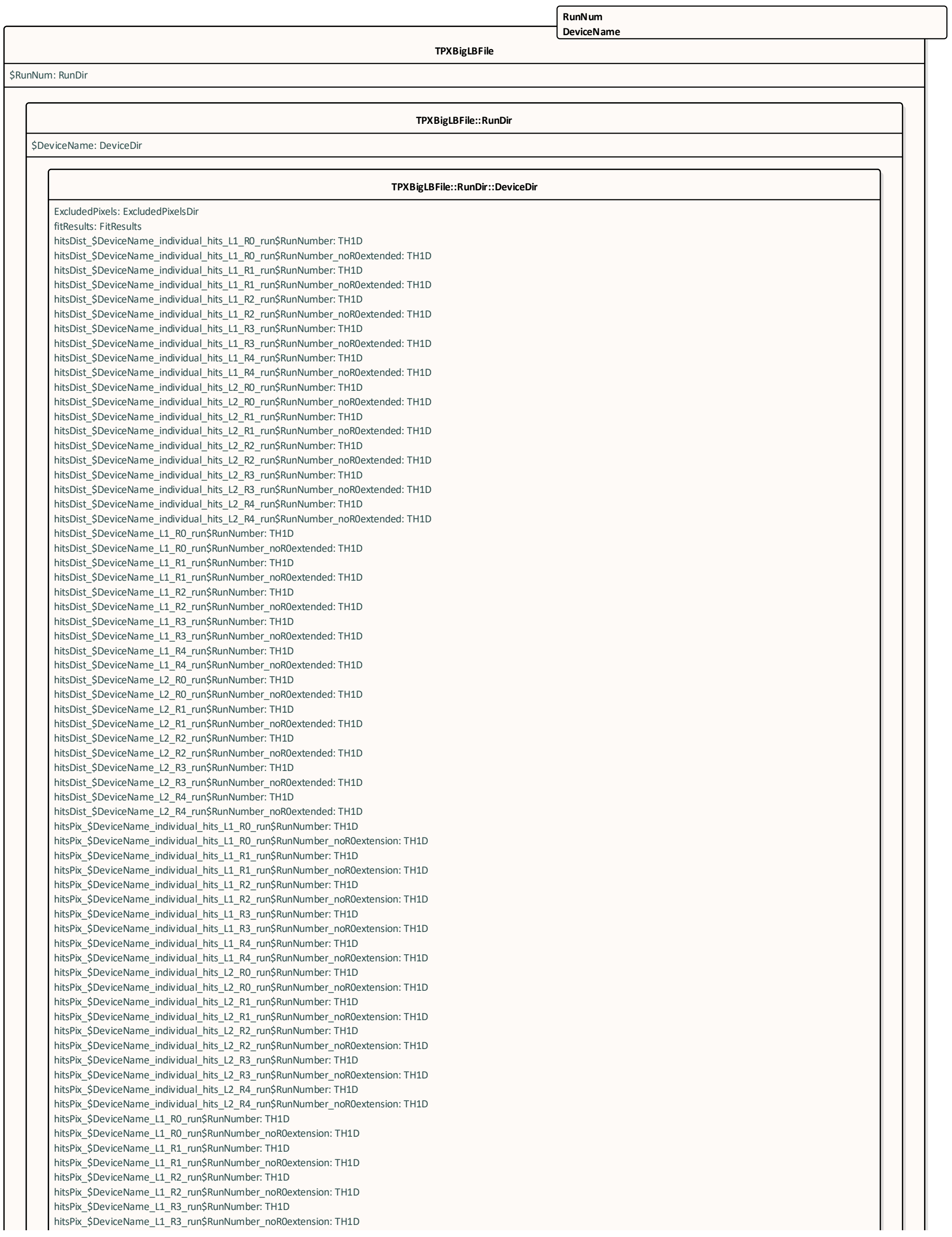

RunNum
DeviceName
TPXBigLBFile
$RunNum: RunDir
TPXBigLBFile::RunDir
$DeviceName: DeviceDir
TPXBigLBFile::RunDir::DeviceDir
ExcludedPixels: ExcludedPixelsDir
fitResults: FitResults
hitsDist_$DeviceName_individual_hits_L1_R0_run$RunNumber: TH1D
hitsDist_$DeviceName_individual_hits_L1_R0_run$RunNumber_noR0extended: TH1D
hitsDist_$DeviceName_individual_hits_L1_R1_run$RunNumber: TH1D
hitsDist_$DeviceName_individual_hits_L1_R1_run$RunNumber_noR0extended: TH1D
hitsDist_$DeviceName_individual_hits_L1_R2_run$RunNumber: TH1D
hitsDist_$DeviceName_individual_hits_L1_R2_run$RunNumber_noR0extended: TH1D
hitsDist_$DeviceName_individual_hits_L1_R3_run$RunNumber: TH1D
hitsDist_$DeviceName_individual_hits_L1_R3_run$RunNumber_noR0extended: TH1D
hitsDist_$DeviceName_individual_hits_L1_R4_run$RunNumber: TH1D
hitsDist_$DeviceName_individual_hits_L1_R4_run$RunNumber_noR0extended: TH1D
hitsDist_$DeviceName_individual_hits_L2_R0_run$RunNumber: TH1D
hitsDist_$DeviceName_individual_hits_L2_R0_run$RunNumber_noR0extended: TH1D
hitsDist_$DeviceName_individual_hits_L2_R1_run$RunNumber: TH1D
hitsDist_$DeviceName_individual_hits_L2_R1_run$RunNumber_noR0extended: TH1D
hitsDist_$DeviceName_individual_hits_L2_R2_run$RunNumber: TH1D
hitsDist_$DeviceName_individual_hits_L2_R2_run$RunNumber_noR0extended: TH1D
hitsDist_$DeviceName_individual_hits_L2_R3_run$RunNumber: TH1D
hitsDist_$DeviceName_individual_hits_L2_R3_run$RunNumber_noR0extended: TH1D
hitsDist_$DeviceName_individual_hits_L2_R4_run$RunNumber: TH1D
hitsDist_$DeviceName_individual_hits_L2_R4_run$RunNumber_noR0extended: TH1D
hitsDist_$DeviceName_L1_R0_run$RunNumber: TH1D
hitsDist_$DeviceName_L1_R0_run$RunNumber_noR0extended: TH1D
hitsDist_$DeviceName_L1_R1_run$RunNumber: TH1D
hitsDist_$DeviceName_L1_R1_run$RunNumber_noR0extended: TH1D
hitsDist_$DeviceName_L1_R2_run$RunNumber: TH1D
hitsDist_$DeviceName_L1_R2_run$RunNumber_noR0extended: TH1D
hitsDist_$DeviceName_L1_R3_run$RunNumber: TH1D
hitsDist_$DeviceName_L1_R3_run$RunNumber_noR0extended: TH1D
hitsDist_$DeviceName_L1_R4_run$RunNumber: TH1D
hitsDist_$DeviceName_L1_R4_run$RunNumber_noR0extended: TH1D
hitsDist_$DeviceName_L2_R0_run$RunNumber: TH1D
hitsDist_$DeviceName_L2_R0_run$RunNumber_noR0extended: TH1D
hitsDist_$DeviceName_L2_R1_run$RunNumber: TH1D
hitsDist_$DeviceName_L2_R1_run$RunNumber_noR0extended: TH1D
hitsDist_$DeviceName_L2_R2_run$RunNumber: TH1D
hitsDist_$DeviceName_L2_R2_run$RunNumber_noR0extended: TH1D
hitsDist_$DeviceName_L2_R3_run$RunNumber: TH1D
hitsDist_$DeviceName_L2_R3_run$RunNumber_noR0extended: TH1D
hitsDist_$DeviceName_L2_R4_run$RunNumber: TH1D
hitsDist_$DeviceName_L2_R4_run$RunNumber_noR0extended: TH1D
hitsPix_$DeviceName_individual_hits_L1_R0_run$RunNumber: TH1D
hitsPix_$DeviceName_individual_hits_L1_R0_run$RunNumber_noR0extension: TH1D
hitsPix_$DeviceName_individual_hits_L1_R1_run$RunNumber: TH1D
hitsPix_$DeviceName_individual_hits_L1_R1_run$RunNumber_noR0extension: TH1D
hitsPix_$DeviceName_individual_hits_L1_R2_run$RunNumber: TH1D
hitsPix_$DeviceName_individual_hits_L1_R2_run$RunNumber_noR0extension: TH1D
hitsPix_$DeviceName_individual_hits_L1_R3_run$RunNumber: TH1D
hitsPix_$DeviceName_individual_hits_L1_R3_run$RunNumber_noR0extension: TH1D
hitsPix_$DeviceName_individual_hits_L1_R4_run$RunNumber: TH1D
hitsPix_$DeviceName_individual_hits_L1_R4_run$RunNumber_noR0extension: TH1D
hitsPix_$DeviceName_individual_hits_L2_R0_run$RunNumber: TH1D
hitsPix_$DeviceName_individual_hits_L2_R0_run$RunNumber_noR0extension: TH1D
hitsPix_$DeviceName_individual_hits_L2_R1_run$RunNumber: TH1D
hitsPix_$DeviceName_individual_hits_L2_R1_run$RunNumber_noR0extension: TH1D
hitsPix_$DeviceName_individual_hits_L2_R2_run$RunNumber: TH1D
hitsPix_$DeviceName_individual_hits_L2_R2_run$RunNumber_noR0extension: TH1D
hitsPix_$DeviceName_individual_hits_L2_R3_run$RunNumber: TH1D
hitsPix_$DeviceName_individual_hits_L2_R3_run$RunNumber_noR0extension: TH1D
hitsPix_$DeviceName_individual_hits_L2_R4_run$RunNumber: TH1D
hitsPix_$DeviceName_individual_hits_L2_R4_run$RunNumber_noR0extension: TH1D
hitsPix_$DeviceName_L1_R0_run$RunNumber: TH1D
hitsPix_$DeviceName_L1_R0_run$RunNumber_noR0extension: TH1D
hitsPix_$DeviceName_L1_R1_run$RunNumber: TH1D
hitsPix_$DeviceName_L1_R1_run$RunNumber_noR0extension: TH1D
hitsPix_$DeviceName_L1_R2_run$RunNumber: TH1D
hitsPix_$DeviceName_L1_R2_run$RunNumber_noR0extension: TH1D
hitsPix_$DeviceName_L1_R3_run$RunNumber: TH1D
hitsPix_$DeviceName_L1_R3_run$RunNumber_noR0extension: TH1D

Figure 2.11: Structure diagram of a big TPX LB file. (Part 1/3)

branches in total, four store the mean of the Gaussian functions and the other four store the widths. The names of the branches storing the means start with `mean` and the names of the branches storing the widths start with `sigma`. All of the branches store the data in two-dimensional arrays of 64-bit floating-point numbers. The first dimension represents a TPX layer and the second dimen-

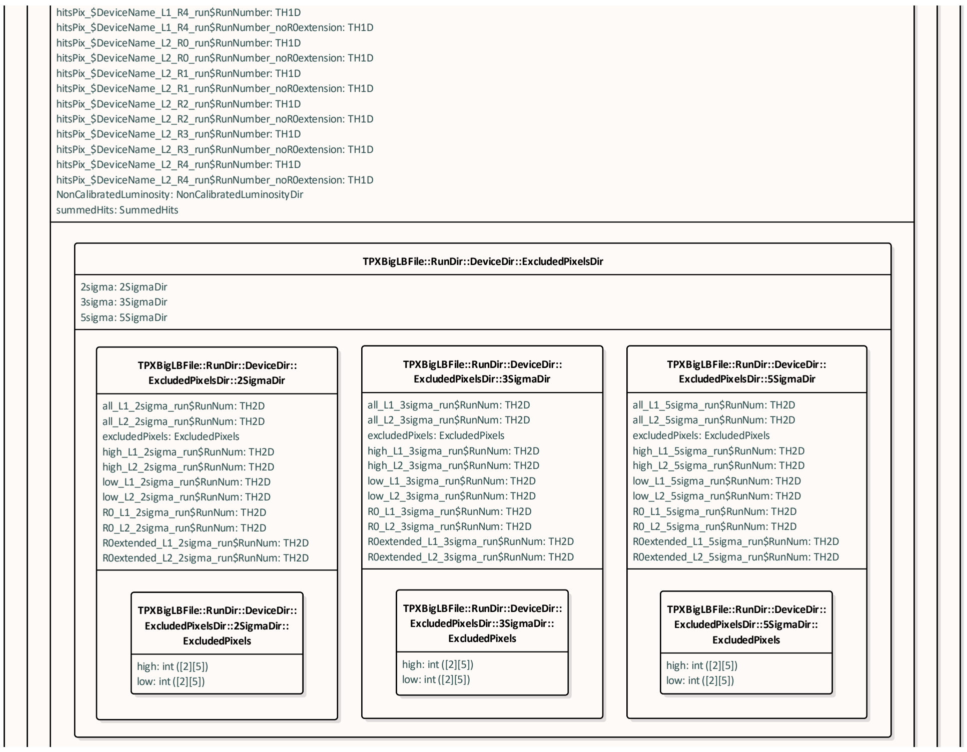

hitsPix_$DeviceName_L1_R4_run$RunNumber: TH1D
hitsPix_$DeviceName_L1_R4_run$RunNumber_noR0extension: TH1D
hitsPix_$DeviceName_L2_R0_run$RunNumber: TH1D
hitsPix_$DeviceName_L2_R0_run$RunNumber_noR0extension: TH1D
hitsPix_$DeviceName_L2_R1_run$RunNumber: TH1D
hitsPix_$DeviceName_L2_R1_run$RunNumber_noR0extension: TH1D
hitsPix_$DeviceName_L2_R2_run$RunNumber: TH1D
hitsPix_$DeviceName_L2_R2_run$RunNumber_noR0extension: TH1D
hitsPix_$DeviceName_L2_R3_run$RunNumber: TH1D
hitsPix_$DeviceName_L2_R3_run$RunNumber_noR0extension: TH1D
hitsPix_$DeviceName_L2_R4_run$RunNumber: TH1D
hitsPix_$DeviceName_L2_R4_run$RunNumber_noR0extension: TH1D
NonCalibratedLuminosity: NonCalibratedLuminosityDir
summedHits: SummedHits
TPXBigLBFile::RunDir::DeviceDir::ExcludedPixelsDir
2sigma: 2SigmaDir
3sigma: 3SigmaDir
5sigma: 5SigmaDir
TPXBigLBFile::RunDir::DeviceDir::ExcludedPixelsDir::2SigmaDir
all_L1_2sigma_run$RunNum: TH2D
all_L2_2sigma_run$RunNum: TH2D
excludedPixels: ExcludedPixels
high_L1_2sigma_run$RunNum: TH2D
high_L2_2sigma_run$RunNum: TH2D
low_L1_2sigma_run$RunNum: TH2D
low_L2_2sigma_run$RunNum: TH2D
R0_L1_2sigma_run$RunNum: TH2D
R0_L2_2sigma_run$RunNum: TH2D
R0extended_L1_2sigma_run$RunNum: TH2D
R0extended_L2_2sigma_run$RunNum: TH2D
TPXBigLBFile::RunDir::DeviceDir::ExcludedPixelsDir::2SigmaDir::ExcludedPixels
high: int ([2][5])
low: int ([2][5])
TPXBigLBFile::RunDir::DeviceDir::ExcludedPixelsDir::3SigmaDir
all_L1_3sigma_run$RunNum: TH2D
all_L2_3sigma_run$RunNum: TH2D
excludedPixels: ExcludedPixels
high_L1_3sigma_run$RunNum: TH2D
high_L2_3sigma_run$RunNum: TH2D
low_L1_3sigma_run$RunNum: TH2D
low_L2_3sigma_run$RunNum: TH2D
R0_L1_3sigma_run$RunNum: TH2D
R0_L2_3sigma_run$RunNum: TH2D
R0extended_L1_3sigma_run$RunNum: TH2D
R0extended_L2_3sigma_run$RunNum: TH2D
TPXBigLBFile::RunDir::DeviceDir::ExcludedPixelsDir::3SigmaDir::ExcludedPixels
high: int ([2][5])
low: int ([2][5])
TPXBigLBFile::RunDir::DeviceDir::ExcludedPixelsDir::5SigmaDir
all_L1_5sigma_run$RunNum: TH2D
all_L2_5sigma_run$RunNum: TH2D
excludedPixels: ExcludedPixels
high_L1_5sigma_run$RunNum: TH2D
high_L2_5sigma_run$RunNum: TH2D
low_L1_5sigma_run$RunNum: TH2D
low_L2_5sigma_run$RunNum: TH2D
R0_L1_5sigma_run$RunNum: TH2D
R0_L2_5sigma_run$RunNum: TH2D
R0extended_L1_5sigma_run$RunNum: TH2D
R0extended_L2_5sigma_run$RunNum: TH2D
TPXBigLBFile::RunDir::DeviceDir::ExcludedPixelsDir::5SigmaDir::ExcludedPixels
high: int ([2][5])
low: int ([2][5])

Figure 2.11: Structure diagram of a big TPX LB file. (Part 2/3)

sion represents a converter region. The names of the two branches, which contain the values which were actually used during the noisy pixels detection, are stored in branches `mean_noR0extended` and `sigma_noR0extended`.

The two directories are `ExcludedPixels` and `NonCalibratedLuminosity`. The `ExcludedPixels` directory contains just three other directories, one for each sigma level. Names of the directories are `2sigma`, `3sigma`, and `5sigma`. All three directories contain the same structure, only names of the objects differ. Each of them contains one `TTree` object and ten `TH2D` objects. A `TH2D` object is a two-dimensional histogram which stores the bin values in 64-bit floating-point numbers. There are five types of histograms for each layer. All of them use the histograms to store maps of pixels. The histograms with names starting `R0` mark the pixels which belongs to the region 0 and the histograms starting `R0extended` mark the pixels which belong to the extension of the region 0. Noisy and dead pixels are marked in the histograms starting `high` and `low`, respectively. The histograms starting `all` mark all the pixels which should be excluded. The sets of pixels in the histograms are disjunctive with the exception of the ones starting `all`, as they are conjunction of the other histograms. The structure of all the histogram names is `$H_L$L_${S}sigma_run$R` where `$H` is the start of the name, `$L` is a layer number, `${S}` is the sigma level, and `$R` is the ATLAS run number.

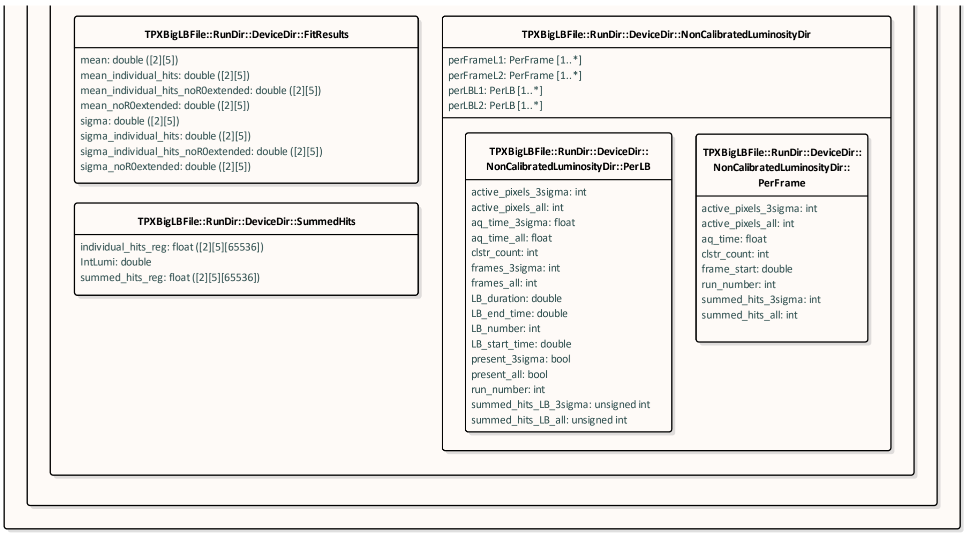

TPXBigLBFile::RunDir::DeviceDir::FitResults
mean: double ([2][5])
mean_individual_hits: double ([2][5])
mean_individual_hits_noR0extended: double ([2][5])
mean_noR0extended: double ([2][5])
sigma: double ([2][5])
sigma_individual_hits: double ([2][5])
sigma_individual_hits_noR0extended: double ([2][5])
sigma_noR0extended: double ([2][5])
TPXBigLBFile::RunDir::DeviceDir::SummedHits
individual_hits_reg: float ([2][5][65536])
IntLumi: double
summed_hits_reg: float ([2][5][65536])
TPXBigLBFile::RunDir::DeviceDir::NonCalibratedLuminosityDir
perFrameL1: PerFrame [1..*]
perFrameL2: PerFrame [1..*]
perLBL1: PerLB [1..*]
perLBL2: PerLB [1..*]
TPXBigLBFile::RunDir::DeviceDir::NonCalibratedLuminosityDir::PerLB
active_pixels_3sigma: int
active_pixels_all: int
aq_time_3sigma: float
aq_time_all: float
clstr_count: int
frames_3sigma: int
frames_all: int
LB_duration: double
LB_end_time: double
LB_number: int
LB_start_time: double
present_3sigma: bool
present_all: bool
run_number: int
summed_hits_LB_3sigma: unsigned int
summed_hits_LB_all: unsigned int
TPXBigLBFile::RunDir::DeviceDir::NonCalibratedLuminosityDir::PerFrame
active_pixels_3sigma: int
active_pixels_all: int
aq_time: float
clstr_count: int
frame_start: double
run_number: int
summed_hits_3sigma: int
summed_hits_all: int

Figure 2.11: Structure diagram of a big TPX LB file. Variables `RunNum` and `DeviceName` are used in names of objects and directories. As UML does not have any formal way how to mark this fact, the dollar sign sigil syntax is used to mark a string substitution. (Part 3/3)

The `TTree` object is `excludedPixels`. It contains only two branches and only one entry. The branches are `high` and `low`. They store the number of noisy and dead pixels for each layer and converter region. The values are stored in two-dimensional arrays of 32-bit integers.

The other directory in the device directory, `NonCalibratedLuminosity` stores luminosity information. It is called non-calibrated because the count rates are not normalized to luminosity. There are four `TTree` objects in the directory, two for each layer. One tree stores the count rates per frame and the other one stores the count rates per LB. The trees which store the count rates per frame are `perFrameL1` and `perFrameL2`. They store data in eight branches and each entry represents one frame. The branch `run_number` stores the number of the ATLAS run in a 32-bit integer. The start time of the frame acquisition is stored in the branch `frame_start`, and the acquisition time is stored in the `aq_time` branch. The start time is stored in a 64-bit floating-point number, and the acquisition time in a 32-bit floating-point number. The numbers of hits per the frame are stored in the branches `summed_hits_all` and `summed_hits_3sigma`. The first branch stores the number of hits without any noisy pixel exclusion, and the other stores the number of hits with a noisy pixel exclusion. Both branches store the values in 32-bit integers. The number of pixels, used to count the number of hits, are stored in the branches `active_pixels_all` and `active_pixels_3sigma`. The value of the first branch is 65,536 as its hit counting counterpart uses all the pixels on the chip. The second branch stores the number of pixels left after the noisy pixel exclusion. Both branches store the values in 32-bit integers. The excluded pixels were detected with the 5$\sigma$ strictness level, even though, the names of the branches say otherwise. The names contain `3sigma` from legacy reasons. The last branch, `clstr_count` contains the number of clusters recorded during the frame. It stores the number in a 32-bit integer.

The trees which store the count rates per LB are `perLBL1` and `perLBL2`. Each entry of the trees represents one LB. The branch `run_number` stores the number of the ATLAS run in a 32-bit integer. The start time of the LB is stored in the branch `LB_start_time`, the end time is in the branch `LB_end_time` and the duration i.e. the difference between end and start times is stored in the branch `LB_duration`. These three branches store the values in 64-bit floating-point numbers. The integrated acquisition time over the duration of the LB is stored in the branch `aq_time_all`. Only acquisition time of frames which are not empty is counted. The branch `aq_time_3sigma` stores the integrated acquisition time of frames which are not empty even after the noisy pixel removal. Both branches store the values in 32-bit floating-point numbers. The number of frames recorded during the LB is stored in the branch `frames_all`. Again, only frames which are not empty are counted. The branch `frames_3sigma` stores the number of frames which are not empty even after the noisy pixel removal. Both branches store the values in 32-bit integers. The numbers of pixels, used to count hits, are stored in the branches `active_pixels_all` and `active_pixels_3sigma`. They store the values in 32-bit integers. The summed numbers of hits are stored in 32-bit unsigned integers in branches `summed_hits_LB_all` and `summed_hits_LB_3sigma`. They are the average numbers of hits per second during the LB. The number of clusters recorded during the LB is stored `clstr_count` in a 32-bit integer. The last two branches, `present_all` and `present_3sigma` store whether there are any hits in the LB. They store the data as Boolean values. The noisy pixels were detected with the 5$\sigma$ strictess level, even though, the names of the branches suggest otherwise. The names contain `3sigma` from historical reasons.

This complicated LB file is simplified during the post-processing. Unlike the complicated LB file, the simplified file contains data for only one layer. This enables to work with each layer as if they are independent devices. The structure of the simplified file is show in Figure 2.12. The files are stored in a directory structure where path to each file can be written as `$D/L$L/LB_Run_$R.root` where `$D` is a name of a TPX device, `$L` is a layer number, and `$R` is a run number.

The file contains only one `TTree` object called `Data` which stores all the data. It basically contains the same data as the `perLBL1` and `perLBL2` trees in the complicated LB file. However, there are some changes to the data and some new data. The changes are in the branches which names are `summed_hits_all` and `summed_hits_3sigma`. In this file, they store the absolute numbers of hits recorded during the LB, and they store the values in 64-bit unsigned integers.

The new data are stored in the branches `LUCID`, `EMEC`, `TILE`, and `TRACKS`.

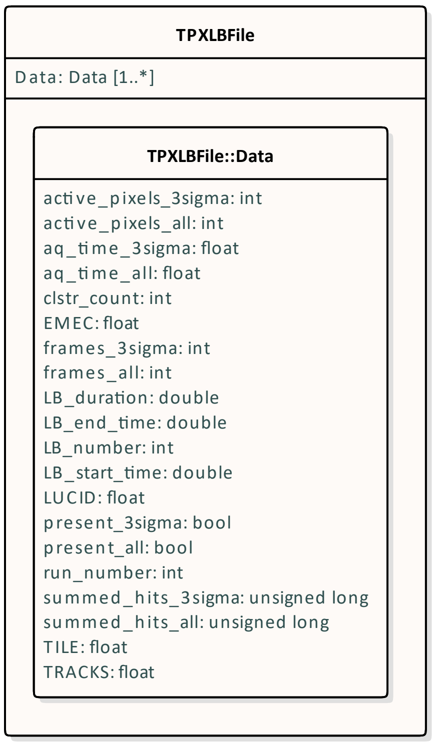


Figure 2.12: Structure diagram of a simplified TPX LB file.

They store luminosity as detected by other luminosity monitors. These values originates from the ATLAS reference files. All of the branches store the values in 32-bit floating-point numbers.

This simplified LB files were inspired by TPX3 LB files. Their structure is shown in Figure 2.13. The names of the files are `LB_Run_$R` where `$R` is a run number. Files for each device have to be stored in separate directory because there is no way how to distinguish them later.

The file contains only one `TTree` object called `Data` which stores all the data. The tree stores TPX3 data, ATLAS reference data, and a description of LBs. The number of the ATLAS run is stored in the branch `Run_number` in a 32-bit integer, and the number of the LB is stored in the `LB_number` also in a 32-bit integer. The duration of the LB is stored in 64-bit floating-point number in the branch `LB_length`, and the start time also in 64-bit floating-point number in the branch `Time`. The numbers of hits and clusters are stored in 32-bit integers in branches `Hits` and `Clusters`. The number of pixels, used to count the hits and clusters, is stored in the branch `Active_pixels` in a 32-bit integer. The rest of the branches store luminosity from other ATLAS luminosity monitors. These branches are `LUCID`, `LUCID_BI2_HitOR`, `EMEC`, `FCAL`, `TILE_D5` and `TILE_D6`, and `TRACKS_TIGHTMOD`. All of these branches store the values in 64-bit floating-point numbers.

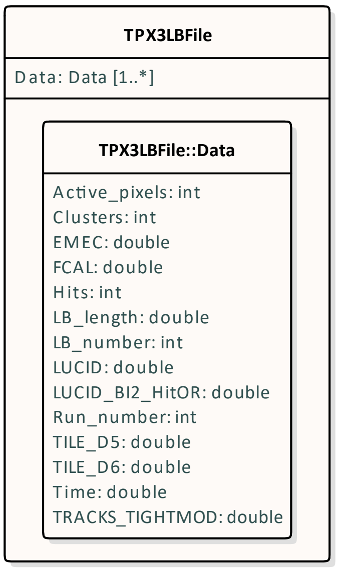


Figure 2.13: Structure diagram of TPX3 LB file.

CHAPTER 3

# Analysis and design

The idea behind the re-engineering is to reduce the number of time-consuming operations and to create software which is highly scalable and configurable without a need to recompile.

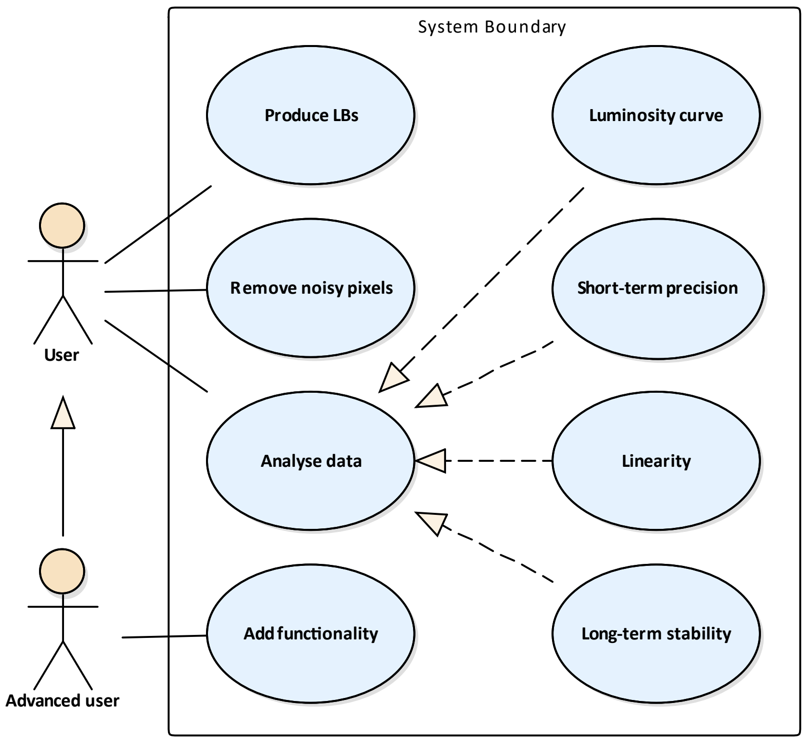


Figure 3.1: Use case diagram for the software.

First, the use cases for the new software were collected. As Figure 3.1 shows, eight use cases of three kinds were found. The first kind are use cases designated for regular users. All of these are related to data processing and analysis. They are the production of LBs, the noisy pixels removal, and analysis of data. Note that the data processing use cases are there to enable effective data analysis.

The second kind of use cases are the individual types of data analysis. These are the known types of data analysis, however more data analysis types could be added in the future. That is the reason for the third kind of use cases and that is the addition of functionality. This use case is designated for advanced users as it requires a knowledge about the software's API and ability to program in C++.

Because all of the use cases shown in Figure 3.1 work with the same data structures, the application is split into several shared libraries. All these libraries are loaded by a launcher which also sets up the environment, data structures and object instances used by all of the libraries. The process of launching the application is shown in Figure 3.2.

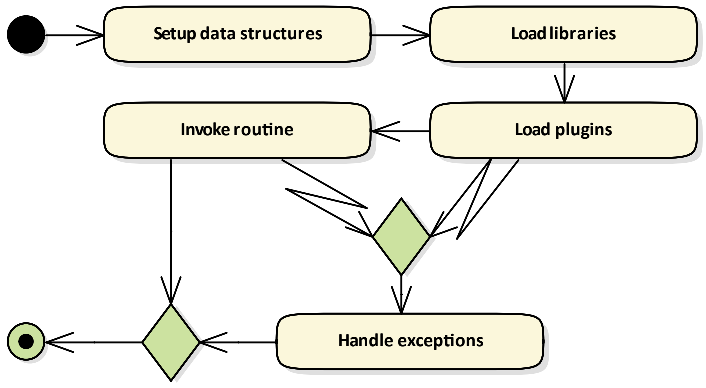


Figure 3.2: Activity diagram of launching of the application.

There are two kinds of shared libraries in the application, plugins and libraries. The shared libraries which serve simply as libraries are used to introduce new kinds of functionality which can be subsequently used by other libraries or plugins. The plugin libraries are used to extend already existing kinds of functionality. For example, there is a library which handles different detectors, and plugins can register new detectors and data formats. Another example is the core library which, among others, parses arguments and options, and launches routines. Plugins register new routines which can be executed, and new options which modify their behaviour.

Both libraries and plugins, are loaded and linked at run-time by the launcher, so they are independent of each other. The libraries are loaded first to provide all functionality, and then the plugins are loaded. Every plugin should contain a handler which is invoked when the plugin is loaded. The handler is used to register the extensions it contains. Plugins may also contain another handler which is invoked just before the plugin is unloaded. This handler is used to clean up what is needed to.

The only exceptions to this loading rule are core library and its dependencies. They are linked at launch-time because the core library contains the code responsible for loading the other libraries and plugins. It is separated from the launcher so the other libraries and plugins can use and share its code and its data like singleton instances independently of the launcher.

Plugins can also register routines. Each routine is registered by a name, and it can be invoked from the command line when the name of the routine is passed to the launcher. Because of this, each routine creates a new use case. Therefore, each use case shown in Figure 3.1 is implemented by one routine, and the routines are separated to several plugins.

The configurable aspect of the new software is done by putting all settings into a series of configuration files. These files can inherit from each other and one or more of them can be read by the final application. The configuration files are written in YAML for readability purposes and its easy-to-understand syntax. Reading and parsing of the files is handled by the launcher, so the libraries and plugins handle just with a representation of the data in the files.

The use cases of the first two kinds shown in Figure 3.1 are basically the same as the one for the previous software. Therefore, the focus is on improving the processes engaged in performing these use cases, mainly the data processing.

The main process of data processing shown in Figure 2.1 cannot be avoided nor simplified, however, the sub-process of the production of TPX LBs and the noisy pixel removal can be simplified. This process can be simplified into just two main steps which are shown in Figure 3.3.

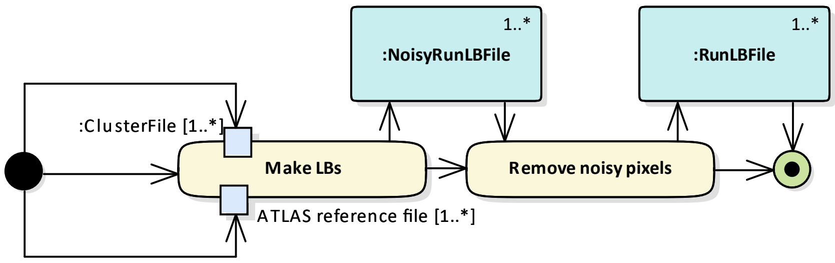


Figure 3.3: Activity diagram of processing of TPX data.

The first step is to create LB files without noisy pixels removed but with enough information so it can be done later. Structure of this file is shown in Figure 3.8. This step has to process the cluster files and therefore it is the most time-consuming operation. The difference from the previous software is that the cluster files are processed only once and all the subsequent processing and potential analysis can use these significantly smaller noisy LB files.

The second step is to detect and remove the noisy pixels from the LB files. This step is way faster then previously as it works with LB data instead of individual clusters. The newly produced LB files are again simplified and therefore smaller than the noisy LB files, because they do not need to contain information used to detect and remove the noisy pixels. The structure of this file is shown in Figure 3.9.

To further minimize the amount of calculations, the API for work with the detector data is designed in such a way that no calculations are done up to the point when the data are needed, usually to create a graph or a histogram. This is achieved by composing iterable objects. These objects can modify, combine, or filter the data. The calculations are done once the objects are iterated over.

There are two families of these iterable objects, the first family works with objects representing LB entries, and the other family works with numbers. There are also a few special iterable objects which extract values and uncertainties of individual properties of LB entries. By this, they create a bridge between these two families.

Plugins register types of detectors in an instance of `DetectorManager`. The instance is provided by the `getInstance` class method. This makes `DetectorManager` singleton. The related class diagram is shown in Figure 3.4. The types are registered by their name. Either the plugins provide a function which constructs instances of the detector, or they provide class representing the detector type. If a plugin provides the function, it either must be convertible to `DetectorConstructor`, or the parameters of `DetectorConstructor` must be convertible to the parameters of that function, and its return type must be convertible to a `Detector` pointer. If a plugin provides the class of the detector, the instance must be constructible using the same parameters which are accepted by `DetectorConstructor`.

Once the detector type is registered in `DetectorManager`, the instances of that detector can be created. This is done by calling the `getDetector` method and providing the name of the detector. This enables to add and use new detectors in already existing routines. The detector is looked up in the configuration files, and the corresponding constructor function is invoked, according to the type of the device specified in the configuration files. The function is invoked with two arguments, the first one is the name of the detector, and the second one is the configuration node which specifies the detector. The method has also an optional Boolean parameter which specifies whether the instance of the detector should be stored by the manager. The instances are stored by default and if the detector is required again, the already existing instance is returned.

A detector instance is a combination of `PropertyComposite` and `LBStream`. The `PropertyComposite` is an abstract template which specifies that its child classes or child templates are logical composites of properties. It defines an interface which enables to retrieve individual properties of the composite and their errors/uncertainties, to retrieve a list of properties of the composite, and to check if the composite contains the required property. A detector is composed of property streams, which can extract the required properties from the

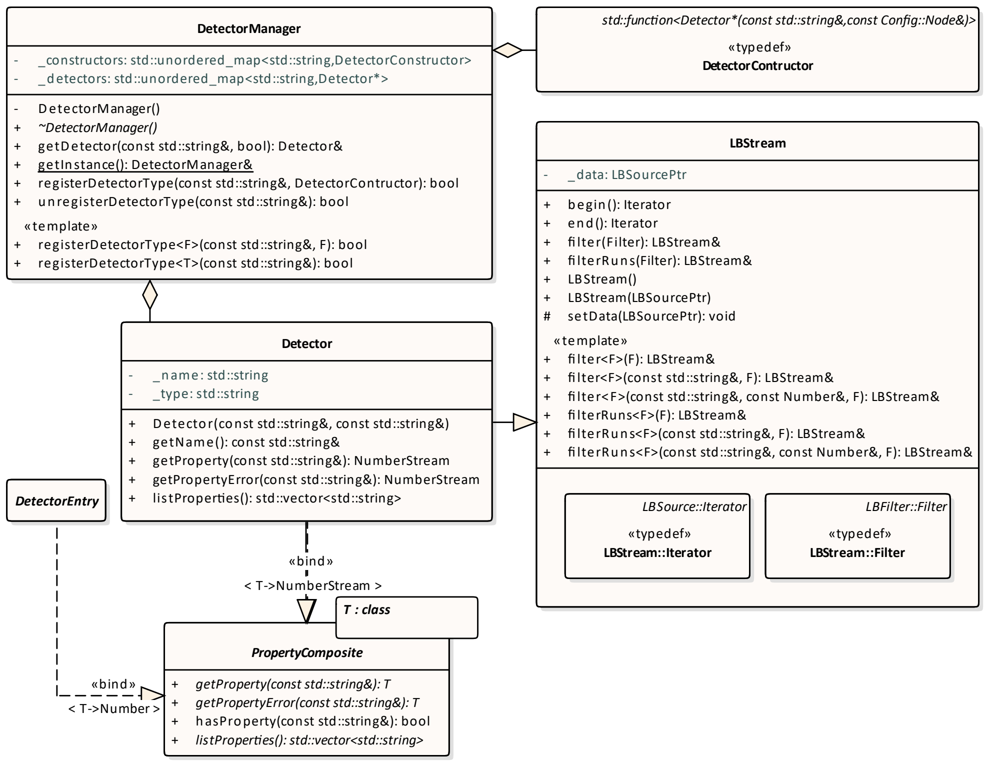

DetectorManager
- _constructors: std::unordered_map<std::string,DetectorConstructor>
- _detectors: std::unordered_map<std::string,Detector*>
- DetectorManager()
+ ~DetectorManager()
+ getDetector(const std::string&, bool): Detector&
+ getInstance(): DetectorManager&
+ registerDetectorType(const std::string&, DetectorContructor): bool
+ unregisterDetectorType(const std::string&): bool
«template»
+ registerDetectorType<F>(const std::string&, F): bool
+ registerDetectorType<T>(const std::string&): bool
std::function<Detector*(const std::string&,const Config::Node&)>
«typedef»
DetectorContructor
LBStream
- _data: LBSourcePtr
+ begin(): Iterator
+ end(): Iterator
+ filter(Filter): LBStream&
+ filterRuns(Filter): LBStream&
+ LBStream()
+ LBStream(LBSourcePtr)
# setData(LBSourcePtr): void
«template»
+ filter<F>(F): LBStream&
+ filter<F>(const std::string&, F): LBStream&
+ filter<F>(const std::string&, const Number&, F): LBStream&
+ filterRuns<F>(F): LBStream&
+ filterRuns<F>(const std::string&, F): LBStream&
+ filterRuns<F>(const std::string&, const Number&, F): LBStream&
LBSource::Iterator
«typedef»
LBStream::Iterator
LBFilter::Filter
«typedef»
LBStream::Filter
Detector
- _name: std::string
- _type: std::string
+ Detector(const std::string&, const std::string&)
+ getName(): const std::string&
+ getProperty(const std::string&): NumberStream
+ getPropertyError(const std::string&): NumberStream
+ listProperties(): std::vector<std::string>
DetectorEntry
«bind»
< T->NumberStream >
T : class
PropertyComposite
+ getProperty(const std::string&): T
+ getPropertyError(const std::string&): T
+ hasProperty(const std::string&): bool
+ listProperties(): std::vector<std::string>
«bind»
< T->Number >

Figure 3.4: Class diagram of the detector API.

stream of detector entries, which are provided by its `LBStream` interface.

The `LBStream` is a stream of detector entries. It composes and wraps around `LBSource` instances which are the iterable objects from the first family. The stream can be filtered by individual LB entries or by the whole runs. If the stream is filtered by runs using properties that can change during a single run, the behavior is undefined. When the stream is filtered the current `LBSource` instance is pass to `LBFilter`, which is a child of `LBSource`, and the filter source is stored instead. The entries are filtered only at the point when, the stream is iterated over.

Once the LBs are filtered, calculations and plotting using their properties can be performed. The properties are retrieved from the detector using the `getProperty` method and their uncertainties can be retrieved using the `getPropertyError` method. They return instances of `NumberStream`, which are streams of `Number` instances.

The `Number` object is a dynamic container which can store either a 64-bit floating-point number, a 64-bit integer, or a 64-bit unsigned integer, and dynamically switch between them. This enables to work with arbitrary data type without a loss in precision. The structure of the object is shown in class diagram in Figure 3.5. The number object can be constructed from any number type, it can be converted back to any number type, and also, it can be worked with as with any regular number type.

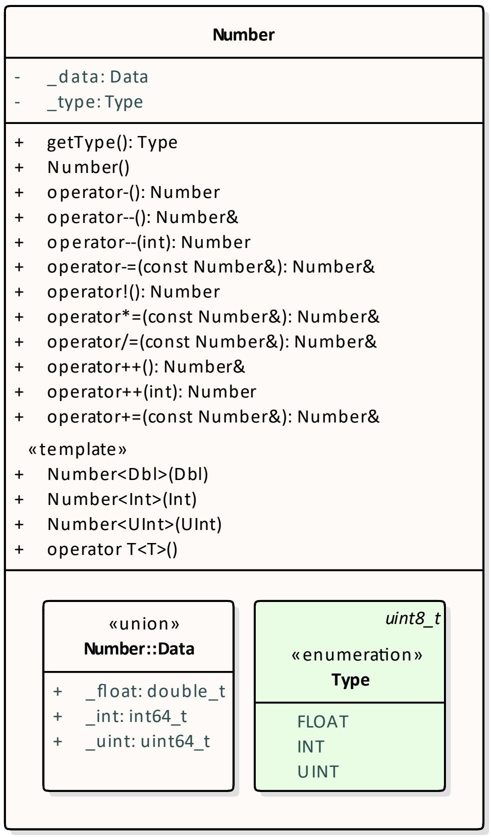
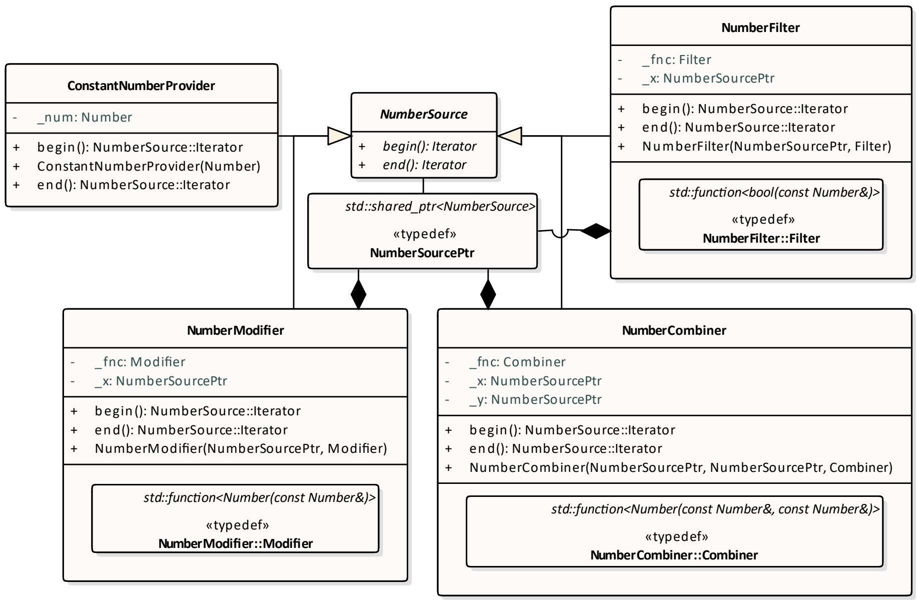


Figure 3.5: Structure diagram of the `Number` class.

The `NumberStream` is similar to the `LBStream`, but its main purpose instead of filtering are calculations. The stream composes and wraps around instances of `NumberSource` which are the iterable objects of the second family. The compositions are shown in the class diagram in Figure 3.6. The number streams can combine number sources using arithmetic binary operators and standard mathematical binary functions. They can be modified using various unary operators and functions. There is also a number source which provides constant numbers. This enables to use expressions like `2 * sqrt(a) / b`, where `a` and `b` are instances of `NumberStream`. The example expression creates an instance of `NumberStream` which contains an instance of `NumberCombiner`. This number source is composed of two other number sources, `NumberCombiner` and `b`. The other combiner is composed of `ConstantNumberProvider` and `NumberModifier`. Finally the modifier contains the source that belongs to the stream `a`. However, none of the calculations are done up until the point the stream is iterated over. The composed structure of number sources creates equivalent structure of iterators. The iterators do all the calculations as the numbers are passed through the structure. Because the `NumberSource` and `LBSource` families are connected by instances of `PropertyExtractor`, the numbers are calculated at the same time as they are provided by the detector.



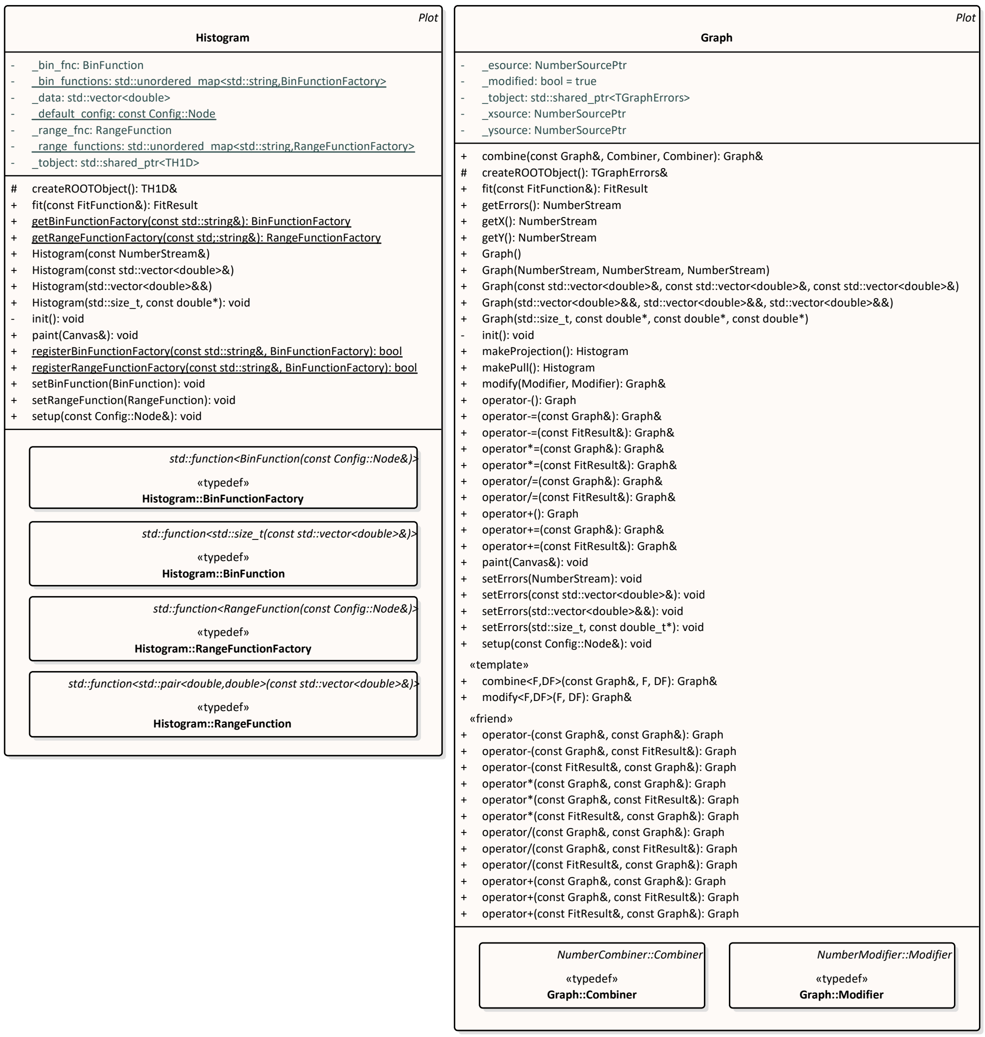
NumberFilter
- _fnc: Filter
- _x: NumberSourcePtr
+ begin(): NumberSource::Iterator
+ end(): NumberSource::Iterator
+ NumberFilter(NumberSourcePtr, Filter)
std::function<bool(const Number&)>
«typedef»
NumberFilter::Filter
ConstantNumberProvider
- _num: Number
+ begin(): NumberSource::Iterator
+ ConstantNumberProvider(Number)
+ end(): NumberSource::Iterator
NumberSource
+ begin(): Iterator
+ end(): Iterator
std::shared_ptr<NumberSource>
«typedef»
NumberSourcePtr
NumberModifier
- _fnc: Modifier
- _x: NumberSourcePtr
+ begin(): NumberSource::Iterator
+ end(): NumberSource::Iterator
+ NumberModifier(NumberSourcePtr, Modifier)
std::function<Number(const Number&)>
«typedef»
NumberModifier::Modifier
NumberCombiner
- _fnc: Combiner
- _x: NumberSourcePtr
- _y: NumberSourcePtr
+ begin(): NumberSource::Iterator
+ end(): NumberSource::Iterator
+ NumberCombiner(NumberSourcePtr, NumberSourcePtr, Combiner)
std::function<Number(const Number&, const Number&)>
«typedef»
NumberCombiner::Combiner

Figure 3.6: Class diagram of `NumberSource` and its child classes.

The data are usually used to create plots like graphs and histograms. The class structure of these plots is shown in Figure 3.7. The plots can be created from the number streams or directly from collections of data. The graph works also as a stream, so it can be used in various calculations. The graph can be combined with other graphs or with instances of `FitResult`. When two graphs are combined their uncertainties are combined using the formula shown in equation 3.1, and the values are assumed to be not correlated. If they are correlated the calculation of the uncertainty has to be done separately, and passed to the `setErrors` method. The instances of the `FitResult` class are retrieved from the `fit` method. They can be also created from constants to form constant functions. The instances can be also modified by unary operations, and combined with other `FitResult` instances.

$$\sigma_f = \sqrt{\left(\frac{\partial f}{\partial a}\right)^2 \sigma_a^2 + \left(\frac{\partial f}{\partial b}\right)^2 \sigma_b^2} \tag{3.1}$$

When the graph needs to use the data, usually to be plotted or fitted, it iterates over its number sources and the results are used to create the `TGraphErrors` object from the ROOT framework. The number sources are then replaced by other sources, which read the data from the `TGraphErrors` instance, where the results are already buffered.



```
Plot
Histogram
-    _bin_fnc: BinFunction
-    _bin_functions: std::unordered_map<std::string,BinFunctionFactory>
-    _data: std::vector<double>
-    _default_config: const Config::Node
-    _range_fnc: RangeFunction
-    _range_functions: std::unordered_map<std::string,RangeFunctionFactory>
-    _tobject: std::shared_ptr<TH1D>

#    createROOTObject(): TH1D&
+    fit(const FitFunction&): FitResult
+    getBinFunctionFactory(const std::string&): BinFunctionFactory
+    getRangeFunctionFactory(const std;:string&): RangeFunctionFactory
+    Histogram(const NumberStream&)
+    Histogram(const std::vector<double>&)
+    Histogram(std::vector<double>&&)
+    Histogram(std::size_t, const double*): void
-    init(): void
+    paint(Canvas&): void
+    registerBinFunctionFactory(const std::string&, BinFunctionFactory): bool
+    registerRangeFunctionFactory(const std::string&, BinFunctionFactory): bool
+    setBinFunction(BinFunction): void
+    setRangeFunction(RangeFunction): void
+    setup(const Config::Node&): void

    std::function<BinFunction(const Config::Node&)>
    «typedef»
    Histogram::BinFunctionFactory

    std::function<std::size_t(const std::vector<double>&)>
    «typedef»
    Histogram::BinFunction

    std::function<RangeFunction(const Config::Node&)>
    «typedef»
    Histogram::RangeFunctionFactory

    std::function<std::pair<double,double>(const std::vector<double>&)>
    «typedef»
    Histogram::RangeFunction
```

```
Plot
Graph
-    _esource: NumberSourcePtr
-    _modified: bool = true
-    _tobject: std::shared_ptr<TGraphErrors>
-    _xsource: NumberSourcePtr
-    _ysource: NumberSourcePtr

+    combine(const Graph&, Combiner, Combiner): Graph&
#    createROOTObject(): TGraphErrors&
+    fit(const FitFunction&): FitResult
+    getErrors(): NumberStream
+    getX(): NumberStream
+    getY(): NumberStream
+    Graph()
+    Graph(NumberStream, NumberStream, NumberStream)
+    Graph(const std::vector<double>&, const std::vector<double>&, const std::vector<double>&)
+    Graph(std::vector<double>&&, std::vector<double>&&, std::vector<double>&&)
+    Graph(std::size_t, const double*, const double*, const double*)
-    init(): void
+    makeProjection(): Histogram
+    makePull(): Histogram
+    modify(Modifier, Modifier): Graph&
+    operator-(): Graph
+    operator-=(const Graph&): Graph&
+    operator-=(const FitResult&): Graph&
+    operator*=(const Graph&): Graph&
+    operator*=(const FitResult&): Graph&
+    operator/=(const Graph&): Graph&
+    operator/=(const FitResult&): Graph&
+    operator+(): Graph
+    operator+=(const Graph&): Graph&
+    operator+=(const FitResult&): Graph&
+    paint(Canvas&): void
+    setErrors(NumberStream): void
+    setErrors(const std::vector<double>&): void
+    setErrors(std::vector<double>&&): void
+    setErrors(std::size_t, const double_t*): void
+    setup(const Config::Node&): void
 «template»
+    combine<F,DF>(const Graph&, F, DF): Graph&
+    modify<F,DF>(F, DF): Graph&
 «friend»
+    operator-(const Graph&, const Graph&): Graph
+    operator-(const Graph&, const FitResult&): Graph
+    operator-(const FitResult&, const Graph&): Graph
+    operator*(const Graph&, const Graph&): Graph
+    operator*(const Graph&, const FitResult&): Graph
+    operator*(const FitResult&, const Graph&): Graph
+    operator/(const Graph&, const Graph&): Graph
+    operator/(const Graph&, const FitResult&): Graph
+    operator/(const FitResult&, const Graph&): Graph
+    operator+(const Graph&, const Graph&): Graph
+    operator+(const Graph&, const FitResult&): Graph
+    operator+(const FitResult&, const Graph&): Graph

    NumberCombiner::Combiner
    «typedef»
    Graph::Combiner

    NumberModifier::Modifier
    «typedef»
    Graph::Modifier
```

Figure 3.7: Class diagram of `Graph` and `Histogram`.

The graph can be also projected to the y-axis or it can produce a pull distribution. The pull distribution is created by dividing all values by their uncertainties and then projecting them to the y-axis. When the data are projected, a histogram is created. The histogram can dynamically change its range and the number of bins. For this purpose it uses `BinFunction` and `RangeFunction`, which are created by corresponding factories. The factories can be registered in the `Histogram` class. Then, the histogram can use these factories to create the functions by providing a configuration node to the factories with all settings which the factories need. There are few predefined binning and range functions. The predefined binning functions are following:

- `fixed` — always returns a fixed number of bins.

- `sqrt` — returns the number of bins equal to square root of the number of entries.
- `sturge` — returns the number of bins according to the Sturge's rule i.e. $\log_2 n + 1$ where $n$ is the number of entries.
- `rice` — returns the number of bins according to the Rice's rule i.e. $2\sqrt[3]{n}$ where $n$ is the number of entries.
- `doane` — returns the number of bins according to the Doane's rule i.e. $1 + \log_2 n + \log_2\left(1 + \frac{|g_1|}{\sigma_{g_1}}\right)$ where $n$ is the number of entries, $g_1$ is the estimated 3rd-moment-skewness i.e. $g_1 = \frac{\sum(x-\overline{x})^3}{\left[\sum(x-\overline{x})^2\right]^{\frac{3}{2}}}$, and its standard deviation $\sigma_{g_1} = \sqrt{\frac{6(n-2)}{(n+1)(n+3)}}$
- `scott` — returns the number of bins according to the Scott's rule i.e. $\frac{\sqrt[3]{n}}{3.49\hat{\sigma}}(\max x - \min x)$ where $n$ is the number of entries, and $\hat{\sigma}$ is the sample standard deviation i.e. $\hat{\sigma} = \sqrt{\frac{\sum(x-\overline{x})^2}{n-1}}$
- `freedman-diaconis` — returns the number of bins according to the Freedman-Diaconis' rule i.e. $\frac{\sqrt[3]{n}}{2\mathrm{IQR}(x)}(\max x - \min x)$ where $n$ is the number of entries, and $\mathrm{IQR}(x)$ is the interquartile range.

And the predefined range functions are following:

- `fixed` — always returns a fixed range.
- `minmax` — returns the range between the minimum and the maximum entries.
- `gaus` — returns a range in which a logarithm of fitted Gaussian function is positive.

The `minmax` and `gaus` functions have two options. The first option is to offset the range by a percentage of its width. That means that if the offset is set to 10 %, the range will be 1.2 times larger as it will be widen by 10 % on each side. The second option is to round the range. The rounding always expands the range. The boundary of the range is rounded to the nearest multiple of five of order lower by one, except when the value of the highest order is equal to one, then the boundary is rounded to the nearest order lower by one. For example, when the upper boundary of a range is equal to 2345, it is rounded to 2500, because the highest order is equal to 2 and the nearest multiple is 2500. However, when the boundary is equal to 0.1234, it is rounded to 0.13, because the highest order is equal to 1 and the closest number rounded to order lower by one is 0.13.

The `gaus` function finds the range by repeatedly fitting and recreating the histogram. The histogram is fitted by a Gaussian function. The boundaries of the range are equal to $\mu \pm \sqrt{2\sigma^2 \ln a}$, where $\mu$ is the center of the function,

$\sigma$ is the width of the function, and $a$ is the height of the function. All the values are provided from the result of the fit. The boundaries of the range are then equal to the points where the logarithm of the fitted Gaussian function is equal to 0, or where the Gaussian function itself is equal to 1. The offset and rounding are then applied to the range if it is required. This process of fitting and recreating the histogram is repeated either until the difference between the previous range and the current one is sufficiently small, or until the algorithm reaches a limit of iterations. The limit can be set in the configuration. The `gaus` function also provides an option to fix the mean $\mu$ at specific value. This is useful for datasets which are known to be centered around specific value, like pull distributions and residuals which are centered around 0.

## 3.1 File structures

The files storing LB data were also re-engineered. The focus is to store as much relevant data as possible, separate this data by meaning, avoid duplication of data in the file where it is possible, and not store data related to other detectors or in case of TPX, even to other layers. The idea is also to pre-calculate as much data as it makes sense, but keeping the granularity on a level, where the data can be filtered and processed more, so things like neutron counting are possible. The file structures are also designed in such a way that they can be easily used by other software which is not related to this one. Also, the sizes of the variables are chosen so the file structure could also be used in the future during the LHC Run-3 or the High Luminosity LHC (HL-LHC) [30].

In case of TPX LB production, the first files to be produced are LB files without noisy pixels removed. The structure of the files is shown in Figure 3.8. These files must contain enough information so noisy pixel removal can be done later. When looking at the previous software, it was found out that calculating the number of hits in the LB per pixel instead of summing them up is sufficient.

The LBs are also already summed to create statistics per ATLAS run. Because the production has to go through all the data anyway, it can be done basically at no cost. The statistics can be used to determine the noisy pixels, or, after the noisy pixel removal, for various types of data analysis such as the long-term stability analysis.

The data are stored in three `TTree` objects. The `lb_data` object stores the data for individual LBs, `run_data` stores the summed data for the whole ATLAS run, and `description` stores all the remaining data, such as the physical information about the TPX layer, the whole device, or information about the run, which do not depend on the LB data.

The `lb_data` tree stores its data in nine branches. Ones of the most important branches are `hits` and `clusters`. The `hits` branch stores the number of hits per pixel in arrays of 65,536 64-bit unsigned integers. The index of

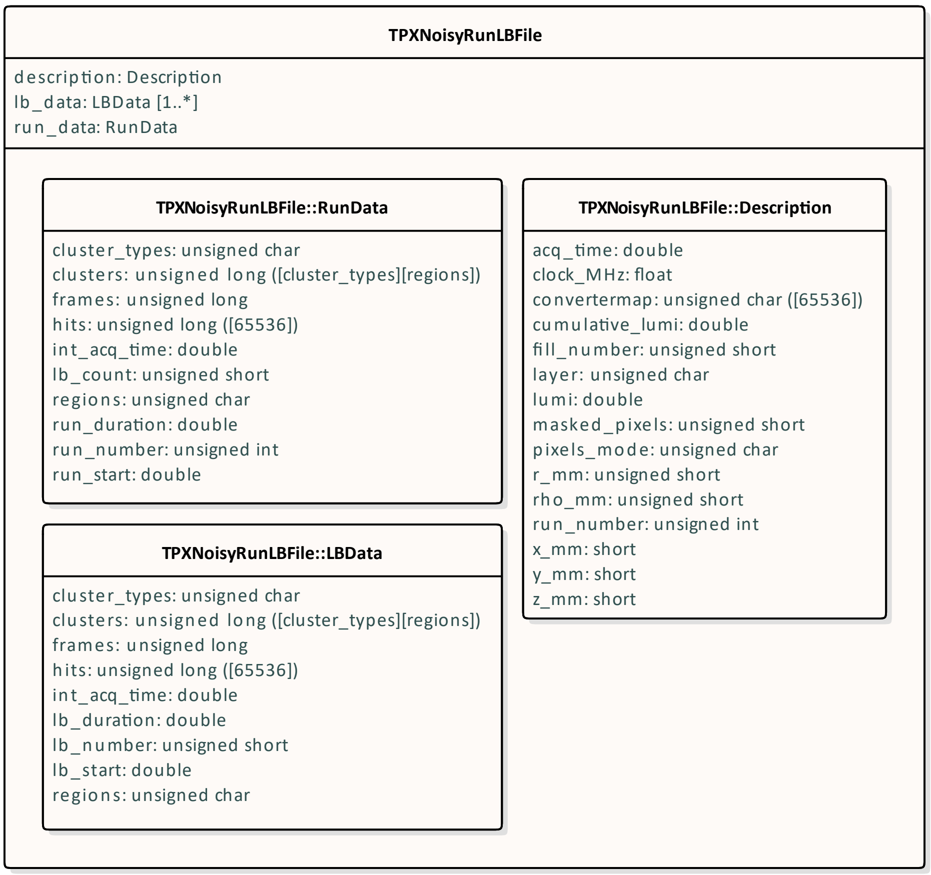


Figure 3.8: Structure diagram of a TPX LB file before noisy pixel removal.

the pixel is calculated as $x + 256y$, where $x$ and $y$ are the coordinates of the pixel. Storing the numbers of hits per pixel enables to exclude the hits when the pixels is determined to be noisy. The branch `clusters` stores the number of clusters in two-dimensional array of 64-bit unsigned integers. The first dimension serves to separate the clusters by their type, and the second dimension separates the data by their converter region. This separation enables to use advanced counting methods such as the thermal neutron counting, which counts the number of HB clusters in the $^6$LiF region and subtracts by it the number of HB clusters in the uncovered region, both normalized to the same area of the chip. The number of cluster types and the number of regions are stored in the branches `cluster_types` and `regions`, respectively. Both values are stored in an 8-bit unsigned integer. The number of cluster types is seven. Six of the cluster types are described by Table 1.3, and they are numbered the same way as in the cluster file. The seventh type is not really a cluster type, it is the sum of all of the clusters independently of their type. The index of the sum of all the clusters is 0. The number of regions is five and the numbering is the same as in the cluster file.

In order to be able to calculate average instantaneous luminosity per LB, the number frames and the sum of their acquisition times are also counted. The number of frames is stored in the branch `frames` in a 64-bit unsigned integer, and the integrated acquisition time is stored in `int_acq_time` in a 64-bit floating-point number.

The last three branches describe properties of the LB. The properties are: the identification number of the LB, the timestamp of the start of the LB, and the duration of the LB. The start timestamp and the duration of the LB are stored in the branches `lb_start` and `lb_duration`, respectively, both in a 64-bit floating-point number. The ID number is stored in the branch `lb_number` in a 16-bit unsigned integer. The sixteen bits are enough to store the IDs because the numbers are assigned sequentially from zero and they need to be unique only in the current ATLAS run, so the counting starts over with each new run. Note that each LB is usually around one minute and the maximum value of a 16-bit unsigned integer is 65,535. This means that in order to overflow the capacity, the ATLAS run has to be longer than circa 45.5 days. This is never the case as the intensity of the LHC fill would drop so low by that time that it would be impossible to use it and to do any meaningful measurements with it.

The other tree in the file, `run_data` stores the summed statistics of the LBs in ten branches. The branches `hits`, `clusters`, `frames`, and `int_acq_time` are the same as the one in the `lb_data` tree. These branches contain values which are equal to the sum of the values of their respective branches in the `lb_data` tree over all of its entries. The branches `cluster_types` and `regions` are identical to the ones in the `lb_data` as they describe the dimensions of the `clusters` array. The new branch `lb_count` stores the number of LBs which were summed. Because of this, the value is equal to the number of entries in the `lb_data` tree. The value is stored in a 16-bit unsigned integer as its maximum value is equal to or lower than the maximum value of the `lb_number` branch in the `lb_data` tree. The last three branches are also similar to their `lb_data` counterparts. They describe the same properties but for the ATLAS run. The start timestamp and the duration of the run are stored in the `run_start` and `run_duration` branches, respectively, both also in a 64-bit floating-point number. The start time of the run is set to be equal to the start time of its first LB. The duration is calculated by calculating the end time of the last LB in the run and subtracting the start time of the run from it. The identification number of the run is stored in `run_number` in a 32-bit unsigned integer. As the number is currently 6-digits long, it cannot fit in 16 bits, and because the number has increased by around 150,000 during the four years of LHC Run-1 and by around 100,000 during the four years of LHC Run-2, it is expected not to get over 1,000,000 neither during the three years of LHC Run-3 nor during the following years of HL-LHC.

Note that all branches storing counts, in both trees, with exception of lb_count, use 64-bit unsigned integers, as the numbers cannot be negative and overflows of 32-bit unsigned integer have been observed in the previous software. Also, all branches related to time use 64-bit floating-point numbers as most branches need the precision it provides, and few others use it for consistency.

The last object in the file is the description tree. It contains fifteen branches related to the TPX layer, the whole device, and the ATLAS run. The branches describing the ATLAS run in this tree do not depend on the LB data. These branches are run_number, fill_number, lumi, and cumulative_lumi. The run_number branch is the same as the one in the run_data tree. The branch is duplicated so the description tree can be used independently of the others. The fill_number stores the identification number of the LHC fill. The value is stored in a 16-bit unsigned integer. As the number is currently 4-digits long, it has to be at least 16-bit, and because the number has increased by around 2500 during the LHC Run-1 and by almost 3750 during the LHC Run-2, it is expected that the number of fills will not cross the 16-bit maximum of 65,536 anytime soon in the future. The branch lumi holds the reference luminosity of the run provided by the ATLAS collaboration, and the branch cumulative_lumi is the reference integrated luminosity from the beginning of the dataset up to the current ATLAS run. A dataset is a collection of consecutive ATLAS runs or LHC fills when the same particles were colliding with the same nominal energy in the same year. There are often several datasets per single year, usually these are: proton-proton collisions at 13 TeV, proton-proton collisions at 5 TeV, and collisions of lead ions. Both of the luminosity branches store the values in 64-bit floating-point numbers.

Because both TPX layers share a common readout, there are only two branches which describe purely the layer. One of them is the layer branch which determines whether the current layer is layer-1 or layer-2. The information is stored in a 8-bit unsigned integer and the possible values are 1 or 2. Even though the information is binary, the Boolean data type is not chosen for two reasons. First, the values are supposed to be numbers and not to represent truthness. And second, the Boolean type is usually stored as one byte value anyway. The second branch describing the layer is the convertermap branch which stores in which converter region each pixel lays. The values are copied from the cluster files from either the branch convertermap_0 or the branch convertermap_1 in the calibData tree, depending on the layer. In contrast to the cluster files, the values are stored in an array of 65,536 8-bit unsigned integers instead of 16-bit signed integers, because they lay in the range between 0 and 4.

The rest of the branches describe the TPX device, and all of the values are copied from the cluster files, however, some of the branches have different, more optimal data type than their cluster file counterparts, and some branches were renamed to make the naming consistent. One renamed branch is acq_time and it stores the acquisition time of a single frame. The value is stored in a 32-bit floating-point number and it is copied from the branch `Acq_time` in either the `dscData` tree or the `clusterFile` tree. Another renamed branch is `clock_MHz` which stores the frequency of the TPX clock in MHz. The frequency is stored in a 32-bit floating-point number and it is copied from the branch `TPX_clock_in_MHz` in the `dscData` tree.

One of the branches which stores the value in different data type is the `masked_pixels` branch. It stores the number of pixels which were excluded already during the data acquisition. The value is copied from the branch of the same name in the `dscData` tree, however, this branch stores the value in a 16-bit unsigned integer instead of 32-bit signed integer, because the value must lay in the range between 0 and 65,536 which is the number of pixels on the chip. The last value of 65,536 cannot be stored in a 16-bit number but that does not matter as that would mean that no pixels were actually used to collect data, and therefore, there would be no data to store. During the whole LHC Run-2, the value was always below 100, with the exception of TPX06 which had the value slightly over 1000.

Another branch with different data type is the `pixels_mode` branch. It stores the mode of the device. The value is copied also from the branch of the same name in the `dscData` tree. Because it can take up only one of the values of 0, 1, or 3, the value is stored in a 8-bit unsigned integer instead of 16-bit signed integer as it is in the cluster file.

The last five branches store coordinates of the location of the device. The branches `x_mm`, `y_mm`, and `z_mm` store the values in 16-bit integers, and the branches `r_mm` and `rho_mm` store the values in 16-bit unsigned integers. They use just 16 bits because no TPX device is located further away from the interaction point than 32 meters as the device located furthest is TPX16 which is not even 20 meters away. The x, y, and z coordinates can be both positive and negative as the interaction point lays in the middle of the ATLAS cavern, however, because the r and ρ coordinates are distances, they are always positive, and therefore, they are stored in unsigned integers as negative values would be invalid.

Once the noisy pixels are detected and removed, other files, which store TPX LBs, are produced. The structure of these files is shown in Figure 3.9. These files are very similar to the previous LB files. There are only four changes. The first two are that the `lb_data` and `run_data` trees do not need to store the hit counts per pixel anymore. Therefore, the `hits` branches in these two trees store the numbers of hits in a single 64-bit unsigned integer. This change saves around 3 MB per LB, that means around 1 GB per 5.5 hours of data-taking.

The other two changes are two new branches in the `description` tree. The first branch is the branch `sigma` which stores the strictness level of the noisy pixel removal. Its value is stored in a 32-bit floating-point number, because for better precision and configurability, the level does not need to be an integer. The second branch is the `active_pixels` branch which stores the number of

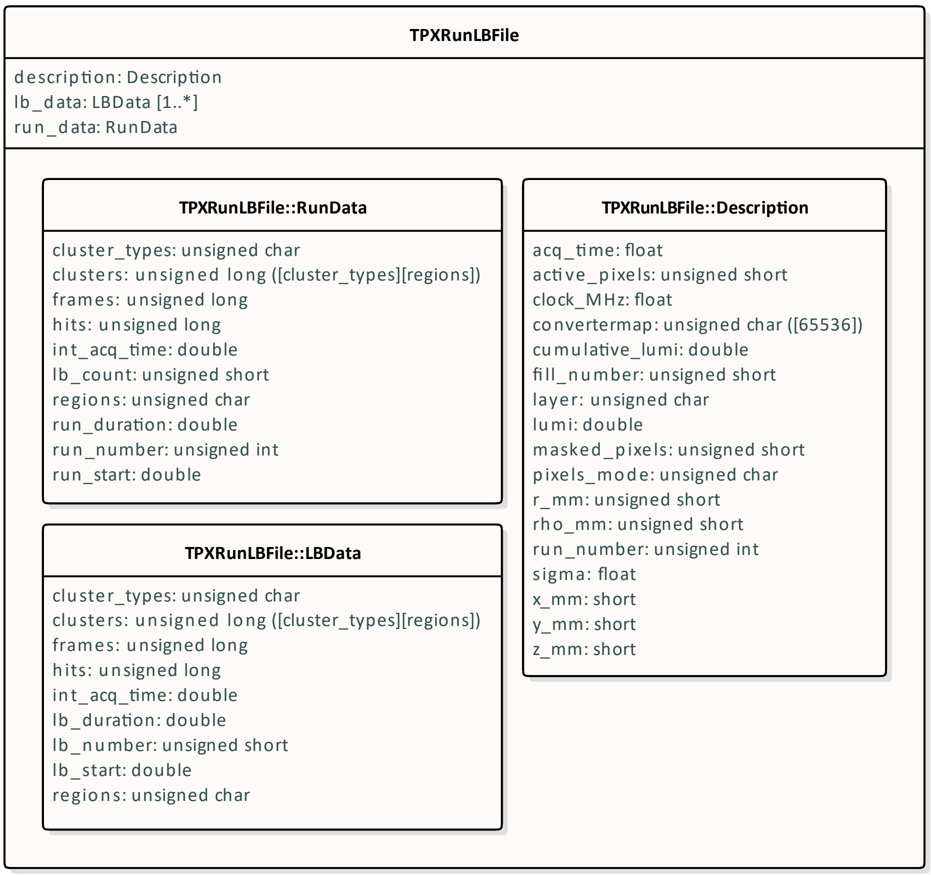


Figure 3.9: Structure diagram of a TPX LB file after noisy pixel removal.

pixels which are used to count the number of hits. This branch is essential in order to enable the so-called active pixels correction i.e. to normalize the number of hits as if the whole chip is used to count the hits. The number is stored in a 16-bit unsigned integer, because the value lays in the range between 0 and 65,536. The last value cannot fit in the 16 bits, but it does not matter as all pixels are never used to count the hits, since there is the non-empty region 0 which is always excluded.

The TPX3 LB file is very similar to the TPX LB file without noisy pixels. The structure of this file is shown in Figure 3.10. However, as the TPX3 devices are not frame based, the branches related to frames are removed. The trees `lb_data` and `run_data` do not contain the `int_acq_time` and `frames` branches, otherwise, the trees are identical.

For the same reason, the `description` tree does not contain the `acq_time` branch. However, this tree has more changes than just this one branch. It also removes the `layer` branch, because even though the TPX3 devices are installed in pairs, each device contains only one layer. The tree also removes the

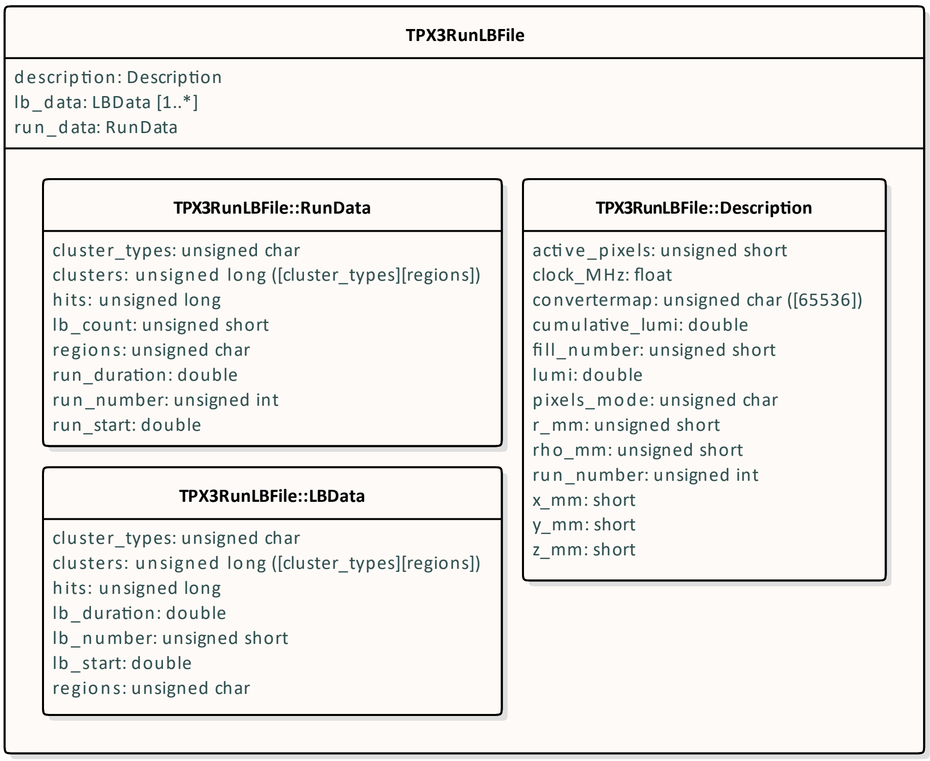


Figure 3.10: Structure diagram of a TPX3 LB file.

`masked_pixels` branch, because there is no data source, which contains this information. The same applies to the `sigma` branch. The branch `clock_MHz` always contains the value of 640, and the values of `convertermap` and of the coordinate branches have to be read from different data sources.

Chapter 4

# Realisation

Because the application has to work with ROOT files, it is best to build it around the ROOT framework. Because of this, the application needs ROOT libraries to be accessible. The framework does not has to be installed locally instead it is better to use CERN Virtual Machine File System (CVMFS) which serves for the purpose of distributing software. This software is available on CERN servers accessible using Linux Public Login User Service (LXPLUS), and it is also installed on `atlastpx2.utef.cvut.cz` [5].

The whole application uses the newest C++20, because there is no need for backward compatibility, as the application is built almost from scratch. However, it requires GCC of version at least 10.2 and compatible ROOT build. Both of these are available at CVMFS.

The C++20 is used for its new features like constraints and concepts [31], or the three-way comparison operator [32], as well as for its improvements in the C++ standard. Such improvements are also concerning the `constexpr` specifier. The specifier now enables wider range of use, and it was added to a lot of STL functions, methods, and constructors. However, the full potential of C++20 `constexpr` cannot be used, because GCC 10.2 does not yet implement these improvements into its libraries.

Because the application also has to read YAML files, the library `yaml-cpp` is used. The source code is available from GitHub, and it is compiled during deployment of the application.

The modular design splits the application into one executable, four libraries, and seven plugins. The executable is the launcher which handles all the loading and setup. One of the libraries is the core library which is the only library which is not linked at run-time. From the other three libraries, one provides streams, other handles detectors, and other provides graphs and histograms. About the plugins, one provides ATLAS reference detectors, one provides TPX detectors, creates TPX LB files, and removes noisy pixels, other provides TPX3 detectors, and four others provide the different kinds of data analysis.

Because of the amount of work, only prototype of the whole application was created. The prototype serves as a proof of concept. It is simplified, so not everything is configurable as it could be, and the TPX3 library and most of the analysis libraries are not implemented. Nonetheless, the prototype itself has over 8500 lines of code, over 270,000 characters, and over 100 classes, all in 48 header files and 39 source files.

## 4.1 Launcher and core library

When the application is launched it loads and links all the ROOT libraries, the `yaml-cpp` library, and the core library at launch-time. The core library is then used to load all the user libraries and plugins, it does so using the `dlopen` mechanism. When a plugin is loaded, the application looks for function called `onload` with one parameter, which is a reference to the `Application` object which is defined in the core library. The lookup is done using the `dlsym` function. If it is found, it is called. Once the `onload` function is invoked, it can register all the routines, detector types, and others.

However, there are few cases when the libraries and the plugins are not loaded. It is when user calls routine to install or reinstall new plugin or library. These routines are registered by the launcher itself. When a routine registered by the launcher are supposed to be invoked the libraries and the plugins are not loaded because it expected that they could be overwritten. That is because they, such as the detector library, might create objects with static life-time duration. When such an object is created and the shared library is rewritten or unloaded, the program crashes at the very end, when the destruction of static objects takes place.

When a plugin or a library are installed using the launcher's routine, all it needs are the header and source files and optionally a dependency file. The directory is copied into the same directory where are all plugins or libraries. Plugins and libraries have separate directories. If the the directory is already present only the files which has changed are copied. In case the source directory is also the target directory of the copying, no files are copied at all. Then the directory is checked for the dependency file. If it is present all libraries and plugins are installed first, before the installment of the current plugin or library. Then an universal makefile is written in the directory. The file is read from the launcher executable as it is linked into it. The linked file contains placeholders which are replaced when it is written out. Such a placeholder is for example in place of the name of the plugin or library. Once the makefile is written, it is used to install the plugin or library.

Because the application needs to be compiled with GCC version at least 10.2 and corresponding build of ROOT, it is recommended to first run the `source.sh` script, which is located in the project directory, using the `source` command. This scripts sets up the environment using CVMFS, if it is not done so, yet. If the script is not executed before, it is executed just before invoking makefile of each dependency and also of the plugin or library. The script is based on other ATLAS environment setup scripts which are rather heavy, and therefore, the processing of the script can take up to few seconds.

## 4.2 Detector library

The detector library implements various classes, some regarding the `Detector` API, some regrading the `LBStream` API, and some are used for handling ROOT files and `TTree` objects.

When a new detector type is registered to the `DetectorManager`, the passed function has to be either of type `DetectorContructor`, or the parameters of `DetectorConstructor` have to be convertible to the parameters of the function, and the return type of the function has be convertible to `Detector` pointer. Another option is to provide a class of the detector. Instances of this class has to be constructible from the parameters of `DetectorConstructor`. The convertibilities and the constructibility are ensured using the C++20 concepts. The concept are used in all of the libraries and plugins for every template parameter where it makes sense. The concepts are used to ensure that all required functionality is provided. If the functionality is missing, the concepts provide clear and short information of the problem, instead of long and cryptic messages which are printed without the concepts. The reason is that concepts make the compilation fail already during a declaration of a class with bound template parameter, however, without the concepts, the compilation fails when it encounters a use of functionality which is not provided.

The other examples of concept usage are `TreeReader` and `TreeWriter`. Because `TTree` objects are conceptually equivalent to database tables, the reader and the writer basically use ORM to read and write `TTree` objects. Because instances of both of these templates need to manipulate with the mapped objects, the objects are required to have certain functionality. The set of the required functionality is called *semiregular* by the C++ standard. This requires the object to be default constructible and *copyable*. The *copyable* concept requires it to be copy constructible, copy assignable, and *movable*, and the *movable* concept requires it to be move constructible, move assignable, and swappable. This set of requirements is ensured by the use of the `std::semiregular` concept which is provided from the `<concepts>` header.

## 4.3 Stream library

The main purpose of the stream library is to implement the `NumberStream` class and all the other related classes. This class enables to perform delayed calculations on streams of number. The calculation is stored as hierarchical structure of `NumberSource` instances, where each operation is represented by one `NumberSource` instance composed of other `NumberSource` instances.

The instances of `NumberSource` are object of various implementations of the `NumberSource` class because the class in incomplete. Therefore, their iterators have to behave differently, as they provide most of the functionality. However, because iterators are used to be returned by value, and values of instances of different classes are not covariant, they all have to return iterator of the same type. For this purpose, `NumberSource` implements its own iterator, which wraps around iterators of the common base `NumberIterator`. This wrapper just bridges all behaviour to the contained iterator which is stored in a pointer because pointers to types with the same common base are covariant.

When the structure of the `NumberSource` instances is iterated over, it creates equivalent structure of iterators. However, building the structure using the iterators retrieved from the other `NumberSource` instances would be inefficient as the size of the structure would be doubled because every iterator would be wrapped. For this reason, the wrapping iterator has the method `getUnderlaying`, which returns a pointer to the wrapped iterator. This enables to build the iterator structure without any wrapping iterators, and therefore every iterator in the structure brings in new functionality.

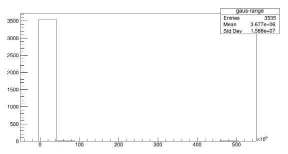
Because the streams has to work with any number data type, it uses `Number` objects. These objects can store either a 64-bit floating-point number, a 64-bit integer, or a 64-bit unsigned integer, and dynamically switch in between if necessary. When two of these object are combined, the type stored in the resulting object is determined according to the types in the original two objects and according to the operation. However, all operations follow a priority of types. This priority is different than the one built in C++. The type with the highest priority is the floating-point number, it is followed by the signed integer, and the unsigned integer has the lowest priority. The priorities of the signed and unsigned integers are reversed in C++.

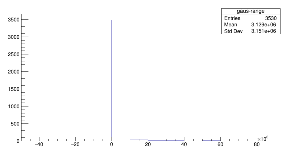
When two `Number` objects are added, the resulting type is equal to the type with the higher priority. When they are subtracted, the same rule applies, with exception of the unsigned integers. When both numbers are unsigned integers, the resulting type depends on their values. If the minuend is smaller than the subtrahend, the resulting type is signed integer, because the resulting difference is negative, otherwise, the resulting type is also the unsigned integer.

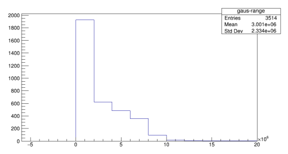
When two `Number` objects are multiplied, the same rule as for addition is applied. However, if they are divided, the rule is more complicated. If one of the divisor or the dividend is a floating-point number, the resulting type is also the floating-point number. Otherwise, the resulting type is floating-point number if a non-zero remainder would be left after the division. If there is no remainder, the resulting type is determined according the priorities.

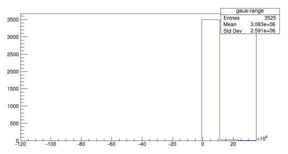
## 4.4 Graph library

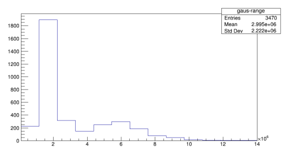
The most interesting part of the graph library is the ability of histograms to dynamically change their binning and range according to their data. The most interesting example is the `gaus` range function which finds the range by repeatedly fitting and recreating the histogram. It first creates a histogram with range computed by the `minmax` function, and with the number of bins computed using the Sturge's rule. The rule has a tendency to over-smooth the histogram which is a useful effect in this algorithm as it needs to work even on datasets with large fluctuations, and almost no precision is required. The histogram is then fitted with a Gaussian function, and the boundaries of the range are calculated as $\mu \pm \sqrt{2\sigma^2 \ln a}$. Then, the offset is applied, and then, the boundaries are rounded if it required. This range is than used to create a new histogram, and the process is repeated. This is done either until the difference between the previous range and the current one is smaller than 1 %, or until the algorithm reaches a limit of iterations. By default, the limit is set to 100 iterations, but it can be changed by the configuration.

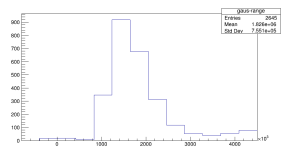

(a) The initial histogram

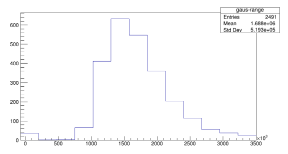

(b) The histogram after the first iteration

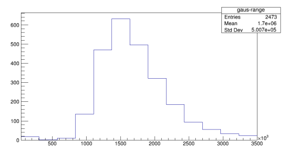

(c) The histogram after the second iteration


(d) The histogram after the third iteration


(e) The histogram after the fourth iteration


(f) The histogram after the fifth iteration


(g) The histogram after the sixth iteration


(h) The histogram after the seventh iteration

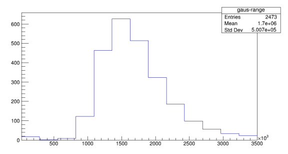

(i) The histogram in the final iteration

Figure 4.1: Iterations of the `gaus` range function for a distribution of hits. The hits originates in TPX02 layer-1 region 0 during the ATLAS run 350160.

An example evolution of the range is shown in Figure 4.1. This example uses distribution of hits in the region 0, which is known to have fluctuations in form of noisy pixels. The noisy pixels have much larger number of hits than the other pixels. This is visible in the initial histogram in Figure 4.1(a). The range is from $-50 \times 10^6$ to $550 \times 10^6$. The initial range is produced by the `mimax` range function with an offset of 10 % and with enabled rounding. This means that there is at least one noisy pixel with a hit count as large as $500 \times 10^6$. On the opposite side of the range, there are pixels with zero hit count because the region 0 contains masked pixels.

After the first iteration, the range is between $-50 \times 10^6$ and $80 \times 10^6$ as shown in Figure 4.1(b). And after the second iteration, the range is between $-6 \times 10^6$ and $20 \times 10^6$ as shown in Figure 4.1(c). In this step, the data, which were contained in a single bin in the previous two histogram, starts to spread into multiple bins. However, because of the shape of the histogram, which looks a bit like a right half of a Gaussian distribution, the range in the next step is wider than before. The range extends between $-120 \times 10^6$ and $35 \times 10^6$, and all the data are again contained in a single bin. This is shown in Figure 4.1(d). The algorithm tries again, and in the next step, the range is between 0 and $14 \times 10^6$ as shown in Figure 4.1(e).

The data are spread into multiple bins again, this time even more than before. The structure on right side from the peak is now more separated from the peak, and it starts to look like a second Gaussian distribution. This is most likely because the region 0 spans across the whole chip and borders all the converter regions. If there is one converter region which has higher hit rate than the others, it can have effect on the region 0 in the form of a second Gaussian distribution which is slightly shifted against the other data. This is why the noisy pixels have to be detected in each converter region separately. The region with the highest hit rate is most likely the uncovered region as there is no neutron converter which filters out low energetic particles.

The fit focuses on the peak and so the range in the next iteration is between $-1 \times 10^6$ and $4.5 \times 10^6$ as shown in Figure 4.1(f). This time, the peak starts to spread into multiple bins, and a distribution start to be visible. In the next step, the range is between $-100 \times 10^3$ and $3.5 \times 10^6$, and the distribution starts to be obvious. This is shown in Figure 4.1(g). The distribution is not Gaussian, in theory, the distribution should be Poisson, however the approximation by a Gaussian function is sufficient. In the next step, the range is tuned between 0 and $3.5 \times 10^6$ as shown in Figure 4.1(h). Finally, the last iteration produces the same range. This is shown in Figure 4.1(i). Because the ranges are identical, there is no difference between them and the algorithm ends.

The histogram, which invoked the `gaus` range function, is independent on the intermediate steps of the function. This is shown in Figure 4.2. The steps shown in Figure 4.1 belong to this histogram. One can see, that the range of the histogram corresponds to the one shown in Figure 4.1(i). However, this histogram sets its number of bins using the `sqrt` binning function instead of the `sturge` function. The histogram is then fitted by a Gaussian function again and the fit result is then drawn over the histogram. This fit result is the base for noisy pixel detection.

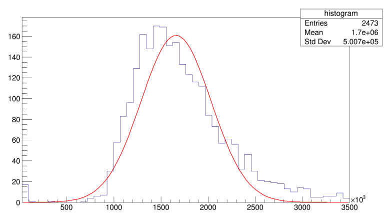


Figure 4.2: The distribution of hits. The histogram was created using the `gaus` range function and the `sqrt` binning functions. The histogram shows the distribution of hits in TPX02 layer-1 region 0 during the ATLAS run 350160.

## 4.5 ATLAS plugin

The data for reference detectors are read directly from the ATLAS reference files described in section 1.3.4. The difficulty is that the files for each year are stored in different locations, are named differently, and have different entries with different names. Furthermore, for a single reference detector, entries with different names are needed to be read from files from different years. This requires not just to iterate over different data sources based on the year of the data, but also to lookup different entries.

This is solved by a hierarchy of iterators. The top iterator goes over different years. The iterator below goes through a list of files for that year. And the last iterator goes through individual LB entries in a file. When the iterator below hits the end, the iterator above moves to the next element, and the iterator below is set to the beginning again. It resembles three nested `for` loops. All three iterators are encapsulated by a different, single iterator which behaves as if all LB entries were located in a single file. The top level is iterator of `std::map` which stores the names of the entries for each year. The map of the lists of files for each year is accessible by all instances of the encapsulating iterator, and it is used to retrieve the list of files, when an iterator moves to the next year.

## 4.6 Timepix plugin

The Timepix plugin registers two routines, `tpx-lb` and `tpx-npr`. The `tpx-lb` routine produces the LB files shown in Figure 3.8 from the TPX cluster files. The `tpx-npr` then removes the noisy pixels from this LB files and produces files shown in Figure 3.9. These routines implement use cases identified in Figure 3.1. The testing and performance of these routines is described in chapter 6.

The `tpx-lb` routine produces LBs for just a single TPX device at a time. However, it can produce LBs either for a single run, a single year, or for all data possible. The routine first fetches a reference detector from `DetectorManager` and filters it if necessary. The reference detector is then used to iterate over the LBs one by one. The cluster files are opened also one by one. Which cluster file have to be opened is determined from the timestamp of the current LB, so no cluster file is opened unnecessarily. Then, the algorithm start to iterate over the cluster entries in the file. If the timestamp of a cluster is lower than the timestamp of the start of the LB, the cluster if skipped. This is because ATLAS runs are not aligned with the cluster files and so they start somewhere in the middle of the files. The algorithm moves to the next LB once a cluster, which was recorded after the end of the current LB, is iterated over. When the end of the cluster file is reached, the next file is opened. If there is no more cluster entries for the LB, the algorithm moves to the next one.

The `tpx-npr` routine removes the noisy pixels for just a single TPX layer at a time. It removes the noisy pixels using the data from the whole year. Note that it can use only the data which were produced using the `tpx-lb` routine up to the date. The routine starts by iterating the noisy LB files one by one. If the ATLAS run stored in the file does not belong to the required year, it is skipped. The pixels are separated by the converter regions. For each region, a histogram of distribution of summed hits for the whole ATLAS run is created and fitted by a Gaussian function. If the distance of the number of hits of a pixel from the center of the function is greater than the required multiple of the function width, or if the pixels lays in the region 0, the pixel is marked for exclusion.

Once all noisy pixels are collected, the required extension of region 0 is also marked for exclusion. Then, the algorithm iterates over the files again, this time only over the ones, which were used in the first round. The numbers of hits are summed together for the run statistics as well as for all LBs in the file. Only numbers of hits originating in pixels, which were not marked for exclusion, are used in the summation.

Chapter 5

# Deployment

The application contains series of makefiles which enables for automated build and deployment. The process starts by invoking the general makefile, which is located in the project directory, using `make`. The structure of the directory is shown in Figure 5.1. Note that the application come only with the `src` directory and the `makefile`, `config.sh`, `default.yaml` files.

The makefile first checks whether all third-party dependencies are built and deployed. The only such dependency is currently the `yaml-cpp`. If the dependency is not present in the `dependencies` directory, its source code is automatically downloaded from GitHub using the `git clone` command. Then, the building scripts for the dependency are prepared using the CMake tool, which creates makefiles which are then invoked using `make`. Once the dependency is built, the resulting libraries are copied into the `lib` directory, in which all the libraries built in the future will be located. Then, the header files of the dependency are copied into a sub-directory of the `include` directory. To the sub-directory is given the same name as the dependency.

Once the dependencies are deployed, the core library is built. The library contains its own makefile among the source files, which is invoked by the general makefile. The invoked makefile builds the core library, and copies the resulting shared library into the `lib` directory. It also copies header files of the core library into the `include` directory, and then, the control is returned to the general makefile.

The next step is to build the launcher. It also contains its own makefile. When it is invoked, it compiles the source files to object files, but it also produces object files from two other makefiles located in the directory of the launcher. One makefile is called `makelib` and the other is called `makeplugin`. These two files are later used to install the libraries and plugins. Then all object files are linked together to produce the launcher executable, which is copied into the project directory.

Once the launcher is ready, it can be used to install the libraries and plugins. First, all libraries, which are present in the `src/lib` directory, are built

```
│ dependencies ........................... the directory of dependencies
│  └─ yaml-cpp .. the directory of the dependency downloaded from GitHub
├─ include ......................... the directory of includes/header files
│  └─ yaml-cpp ................... the directory of the dependency includes
├─ lib ........................................ the directory of libraries
├─ plugin ..................................... the directory of plugins
├─ src .................................... the directory of source codes
│  ├─ Core .............................. the directory of the core library
│  ├─ Launcher ..................... the directory of launcher source codes
│  ├─ lib ........................... the directory of library source codes
│  │  ├─ detector ................... the directory of the detector library
│  │  ├─ graph ......................... the directory of the graph library
│  │  └─ stream ....................... the directory of the stream library
│  └─ plugin ......................... the directory of plugin source codes
│     ├─ atlas ............... the directory of ATLAS plugin source codes
│     ├─ linearity................... the directory of the linearity plugin
│     ├─ lumicurve........... the directory of the luminosity-curve plugin
│     ├─ precision ........ the directory of the short-term-precision plugin
│     ├─ stability.......... the directory of the long-term-stability plugin
│     ├─ timepix..................... the directory of the Timepix plugin
│     └─ timepix3.................. the directory of the Timepix3 plugin
├─ config.sh....................... the environment configuration script
├─ default.yaml........................... the default configuration file
├─ makefile .............................. the general deployment script
└─ launcher.exe ............................... the launcher executable
```

Figure 5.1: Structure of the project directory.

one by one using the `./launcher.exe install lib` command. Every library is built using the `makelib` makefile. It builds the library, copies the binary into the `lib` directory and it also copies the header files into the `include` directory. Then, all plugins present in the `lib/plugin` directory are installed one by one using the `./launcher.exe install plugin` command. Every plugin is built using the `makeplugin` makefile. It builds the plugin and copies the resulting binary into the `plugin` directory. The installation routine is further described in section 4.1. After this step, the application is ready to be used.

It is highly recommended to invoke the `config.sh` script before the execution of the `make`. The script has to be invoked using the `source` command. If the script is not invoked, it is very likely that the deployment will fail. If it does not fail, but the script was not invoked, it is not guaranteed that the application will work.

Chapter **6**

# Measurements and testing

The efficiency and memory requirement of the `tpx-lb` and `tpx-npr` routines were measured on the `lxplus716.cern.ch` server. The speed and memory consumption was first tested on data for TPX02 run 350160 and then on data for TPX05 run 350160. The LBs for TPX02 were produced in 8.5 minutes. The measurement was repeated several times with almost identical results. The maximum virtual memory consumed during the process was below 2 GB, and the maximum resident memory was slightly over 1.5 GB.

The LBs for TPX05 were not produced, however the production was used to determine the speed of the routine in number of processed cluster entries per minute. The measurement was performed repeatedly, sometimes for several hours. It was found that the performance of the routine is stable and it is able to process around 95,000 cluster entries per minute, that is around 630 µs per entry. Because the used cluster files contain around 96,000,000 entries, it is estimated that the production of LBs for TPX05 run 350160 would take around 17 hours. However, there are known ways how to increase the speed performance, which can be implemented in the future.

In order to compare to the previous software, the LB production of TPX02 run 350160 was executed using the previous software on the same LXPLUS server. It was found that the previous code is much faster on skipping the cluster entries with timestamp smaller than the beginning of the ATLAS run. However, no LBs were produced, because after between 2.5 and 3 minutes, the program is killed by the OS as its virtual memory exceeds 32 GB which are available on the server. Nonetheless, from the output log, which prints the progress of the production, it was found that the program has processed less than 44,000 entries involved in the LB production before it was killed. That mean that the previous software was able to process on average less than 15,000 entries per minute. Note that because it stores the entries in a map structure, the complexity of inserting new entry is $\mathcal{O}(n \log n)$. Therefore, the performance of the previous software declines as it loads more and more data. Also note, that the program has exceeded the 32 GB limit but the cluster files are just 5.5 GB large.

This all means that in contrast to the previous software, the re-engineered one not just fits into the memory and has no special memory requirements, it is also already at least 6 times faster than its predecessor, and further performance improvements could be applied in the future.

About the `tpx-npr` routine, the execution is almost instantaneous and the memory consumption is lower than half a GB. This is major improvement, as in the previous software, the LBs had to be reproduced to exclude the noisy pixels.

The produced LBs were compared to the LBs produced by the previous software. The comparison is shown in Figure 6.1. The new LBs have a higher number of hits by about 35 %. This is due to the fact, that the previous LBs were produced with noisy pixels detected and removed using the whole year, whereas the new LBs have noisy pixels removed only using the single run. The same reason is expected to be the reason for the outliers. Nonetheless, the ratio between the old LBs and the new LBs is constant, and therefore it is considered to be correct.

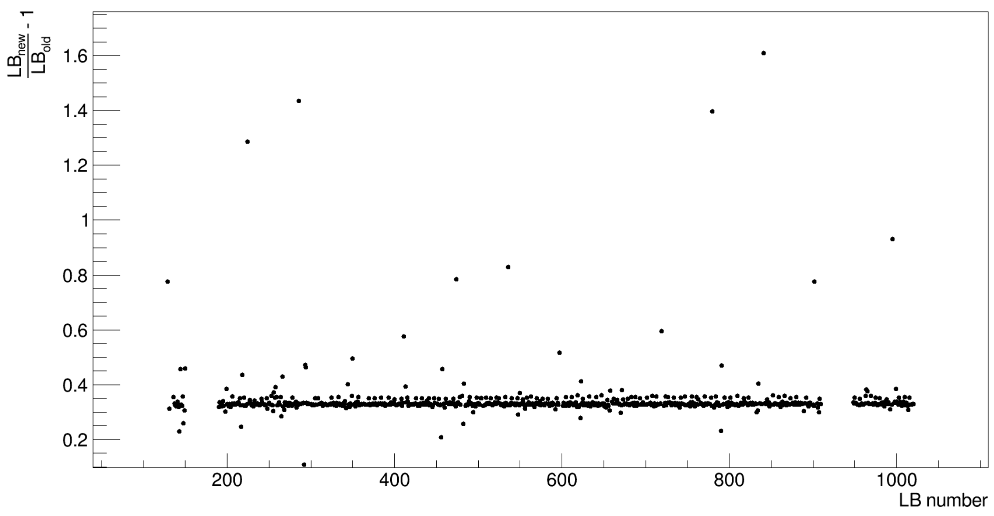


Figure 6.1: Comparison of LBs produced by different software. The LBs are created from TPX02 layer-1 run 350160.

The significant speed increase comes at the cost of an additional disk usage. In order to compare the file sizes, the data for TPX02 run 350160 were used. The cluster files, which have to be processed, are around 5.5 GB, the large LB file shown in Figure 2.11 is around 10 MB and the post-processed file shown in Figure 2.12 is around 30 kB, that means around 60 kB for both layers. Concerning the new files, the LB file before the noisy pixel removal is around 120 MB, so around 240 MB for both layers, and the LB file after noisy pixel removal is around 35 kB, so around 70 kB for both layers.

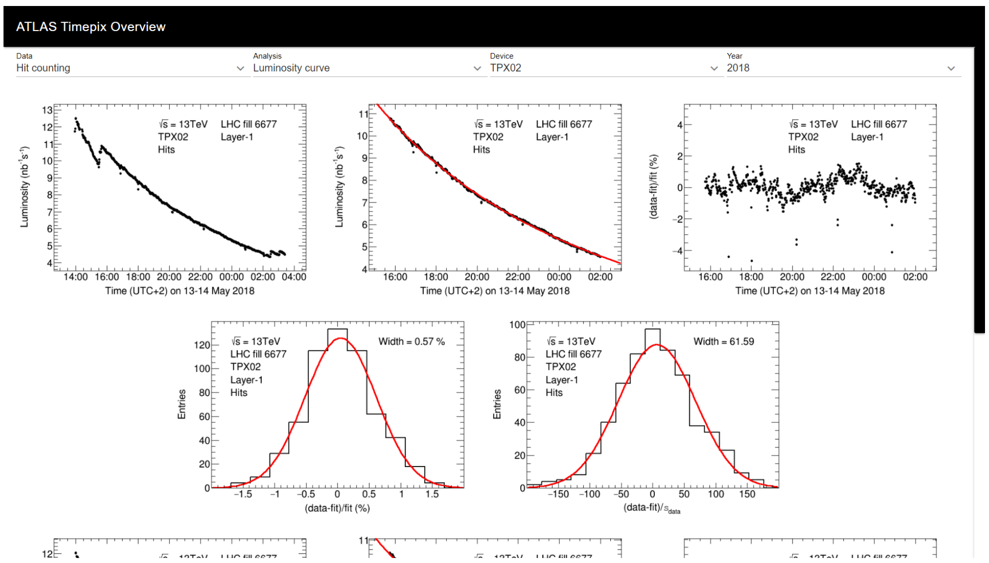
The cluster files for TPX02, TPX05, TPX06, TPX07, TPX12, and TPX14 for all four years are around 11 TB in total and the previous large LB files shown in Figure 2.11 are around 42 GB in total. Therefore, the total size of the noisy LB files for the same dataset is estimated to be between 480 GB and 1 TB. As the estimate was done using TPX02 which has fewer entries per cluster file, it is expected to be closer to the 1 TB.

Because the LB files without the noisy pixels are of similar size as the post-processed LB files, the total size of all of the files is expected to be similar, and therefore, to be around 200 MB.

Chapter 7

# Web page

In order to make the resulting plots available on a web interface, a web page is developed. The web page is developed from scratch using HTML, CSS, and JavaScript. The web page is made to comply with the Material Design guidelines [33], and it is made to be responsive.



ATLAS Timepix Overview

Data: Hit counting
Analysis: Luminosity curve
Device: TPX02
Year: 2018

Figure 7.1: Screenshot of the web interface.

The web page enables the user to select a type of analysis and various parameters about a dataset. Then, the web page displays the plots created from the selected dataset for the selected analysis. Currently, the parameters regarding the dataset are: the counting method, the TPX device, and year of the data acquisition.

The web page consists of three main areas, the header, the navigation, and the contents. The main structure and the header are designed using Material

Design Lite (MDL) [34], which was modified to match the desired style of the page. The navigation consist of four drop-down lists. The drop-down lists are custom made because they are not defined in MDL. The component was created by combining a text field and drop-down menu. The component does not use these other two components, it just combines their functionality and design. The contents part of the page is used to show a list of plots. The list is dynamically changed when a different dataset or analysis type is selected.

When a user clicks on a plot, it is opened over the whole page, and its surrounding turns black so the plot is better visible and the surroundings do not distract the user. Then, the plot can be zoomed in and out, enabling for closer look. The user can close the plot by clicking on it in the maximum zoom-out, or by clicking on the black background. All of this behaviour was also custom made, as there was no component which exactly suits this needs.

The navigation and the contents are placed into a scroll-able element. The element has custom designed scroll bar, which is made to fit the design of the page.

The web uses CSS3 and JavaScript standard ECMAScript 6. This might make the web page incompatible with older browsers. It also uses the modern APIs such as the fetch API to send requests and load the images, or the service worker API to cache the web page and the subsequently required plots. The web page and all of its resources are cached statically on load, however, the plots are cached only when they are demanded, because there are currently around 600 of them.

# Conclusion

This thesis fulfills all five goals of the task description. The first goal was to analyse the needs of the ATLAS luminosity measurements using TPX and TPX3 devices regarding an automatic procedure to coherently determine the luminosity from 2015 to 2018 data. The second goal was to design a software, which reads the TPX and TPX3 data, performs a noisy pixel removal automatically, and produces performance plots regarding the LHC luminosity curve, short-term precision, linearity, and long-term stability. The third goal was to implement this software, the fourth goal was to test the software, and the fifth goal was to make the resulting distributions (plots) available on a web interface.

The fulfillment of the first goal, which was to analyse the needs, is detailed in chapter 1 and chapter 2, and by the fact that the other goals were accomplished as well. The fulfillment of the second goal regarding the design of the software is given in chapter 3, and also by the fact that the next goal is accomplished. The third goal, the implementation of the software was accomplished by creating a prototype of the software (chapters 4 and 5), which serves as a proof of concept. The fulfillment of the fourth goal, testing the software is explained in chapter 6. The fifth goal, to make the results available on a web interface, is detailed in chapter 7, and demonstrated by the online web page.

The prototype has over 8500 lines of C++ code, over 270,000 characters, and over 100 classes. Because of its already high laboriousness, the prototype implements only a subset of functionalities, and it is designed for future extensions.

# Bibliography

[1] Llopart, X.; Ballabriga, R.; Campbell, M.; Tlustos, L.; Wong, W. Timepix, a 65k programmable pixel readout chip for arrival time, energy and/or photon counting measurements. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, volume 581, no. 1, Oct. 2007: pp. 485 – 494, ISSN 0168-9002, doi:10.1016/j.nima.2007.08.079. Available from: `http://www.sciencedirect.com/science/article/pii/S0168900207017020`

[2] Llopart, X.; Ballabriga, R.; Campbell, M.; Tlustos, L.; Wong, W. Erratum to "Timepix, a 65k programmable pixel readout chip for arrival time, energy and/or photon counting measurements" [Nucl. Instr. and Meth. A. 581 (2007) 485–494]. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, volume 585, no. 1, Jan. 2008: pp. 106 – 108, ISSN 0168-9002, doi:10.1016/j.nima.2007.11.003. Available from: `http://www.sciencedirect.com/science/article/pii/S0168900207022930`

[3] Poikela, T.; et al. Timepix3: a 65K channel hybrid pixel readout chip with simultaneous ToA/ToT and sparse readout. *Journal of Instrumentation*, volume 9, no. 05, May 2014: pp. C05013–C05013, doi:10.1088/1748-0221/9/05/c05013. Available from: `https://iopscience.iop.org/article/10.1088/1748-0221/9/05/C05013`

[4] The ATLAS Collaboration; et al. The ATLAS Experiment at the CERN Large Hadron Collider. *Journal of Instrumentation*, volume 3, no. 08, Aug. 2008: pp. S08003–S08003, doi:10.1088/1748-0221/3/08/s08003. Available from: `https://iopscience.iop.org/article/10.1088/1748-0221/3/08/S08003`

[5] Fiedler, P. *Analysis of Data from a Network of Pixel Detectors.* Bachelor's thesis, Czech Technical University in Prague, Prague, June 2018.

[6] Fiedler, P.; Sopczak, A. Systematic Comparison of TPX network and TPX3 devices regarding luminosity measurements in the ATLAS cavern - Summer Student project. Technical report ATL-COM-DAPR-2020-001, CERN, Geneva, Apr. 2020. Available from: `https://cds.cern.ch/record/2716042`

[7] Bergmann, B.; Caicedo, I.; Leroy, C.; Pospíšil, S.; Vykydal, Z. ATLAS-TPX: a two-layer pixel detector setup for neutron detection and radiation field characterization. *Journal of Instrumentation*, volume 11, no. 10, Oct. 2016: pp. P10002–P10002, doi:10.1088/1748-0221/11/10/p10002. Available from: `https://iopscience.iop.org/article/10.1088/1748-0221/11/10/P10002`

[8] Burian, P.; et al. Timepix3 detector network at ATLAS experiment. *Journal of Instrumentation*, volume 13, no. 11, Nov. 2018: pp. C11024–C11024, doi:10.1088/1748-0221/13/11/c11024. Available from: `https://iopscience.iop.org/article/10.1088/1748-0221/13/11/C11024`

[9] Bergmann, B.; et al. Relative luminosity measurement with Timepix3 in ATLAS. *Journal of Instrumentation*, volume 15, no. 01, Jan. 2020: pp. C01039–C01039, doi:10.1088/1748-0221/15/01/c01039. Available from: `https://iopscience.iop.org/article/10.1088/1748-0221/15/01/C01039`

[10] CERN BE-CO. Accelerator Performance and Statistics. [online], [cited 2020/12/10]. Available from: `http://acc-stats.web.cern.ch/acc-stats/#lhc/overview-panel`

[11] ATLAS Data Summary — 2015 - pp at 13 TeV. [online], [cited 2020/08/23]. Available from: `https://atlas.web.cern.ch/Atlas/GROUPS/DATAPREPARATION/DataSummary/2015/`

[12] ATLAS Data Summary — 2018 - PbPb. [online], [cited 2020/08/23]. Available from: `https://atlas.web.cern.ch/Atlas/GROUPS/DATAPREPARATION/DataSummary/2018hi/`

[13] Bergmann, B.; Billoud, T.; Leroy, C.; Pospíšil, S. Characterization of the Radiation Field in the ATLAS Experiment With Timepix Detectors. *IEEE Transactions on Nuclear Science*, volume 66, no. 7, July 2019: pp. 1861–1869, ISSN 1558-1578, doi:10.1109/TNS.2019.2918365. Available from: `https://ieeexplore.ieee.org/document/8720202`

[14] Sopczak, A.; et al. Determination of Luminosity With Thermal Neutron Counting Using TPX Detectors in the ATLAS Cavern in LHC Proton-Proton Collisions at 13 TeV. *IEEE Transactions on Nuclear Science*, volume 65, no. 7, July 2018: pp. 1378–1383, doi:10.1109/TNS.2018.2839683. Available from: `https://ieeexplore.ieee.org/document/8362718`

[15] Bergmann, B.; et al. Timepix detector network for luminosity monitoring and characterization of the radiation fields within the ATLAS cavern. In *ACES 2016 - Fifth Common ATLAS CMS Electronics Workshop for LHC Upgrades*, CERN, Mar. 2016. Available from: `https://indico.cern.ch/event/468486/contributions/1144326/`

[16] Aad, G.; et al. ATLAS data quality operations and performance for 2015–2018 data-taking. *Journal of Instrumentation*, volume 15, no. 04, Apr. 2020: pp. P04003–P04003, doi:10.1088/1748-0221/15/04/p04003. Available from: `https://iopscience.iop.org/article/10.1088/1748-0221/15/04/P04003`

[17] Gillies, J. Luminosity? Why don't we just say collision rate? [online], Mar. 2011, [cited 2020/12/10]. Available from: `https://home.cern/news/opinion/cern/luminosity-why-dont-we-just-say-collision-rate`

[18] Bertelsen, H.; et al. Operation of the upgraded ATLAS Central Trigger Processor during the LHC Run 2. *Journal of Instrumentation*, volume 11, no. 02, Feb. 2016: pp. C02020–C02020, doi:10.1088/1748-0221/11/02/c02020. Available from: `https://iopscience.iop.org/article/10.1088/1748-0221/11/02/C02020`

[19] Aaboud, M.; et al. Luminosity determination in pp collisions at $\sqrt{s}$ = 8 TeV using the ATLAS detector at the LHC. *The European Physical Journal C*, volume 76, no. 12, Nov. 2016: p. 653, ISSN 1434-6052, doi:10.1140/epjc/s10052-016-4466-1. Available from: `https://link.springer.com/article/10.1140/epjc/s10052-016-4466-1`

[20] Burkhardt, H.; Grafström, P. Absolute Luminosity from Machine Parameters. Technical report LHC-PROJECT-Report-1019. CERN-LHC-PROJECT-Report-1019, CERN, Sept. 2007. Available from: `http://cds.cern.ch/record/1056691`

[21] van der Meer, S. Calibration of the effective beam height in the ISR. Technical report CERN-ISR-PO-68-31. ISR-PO-68-31, CERN, Geneva, 1968. Available from: `https://cds.cern.ch/record/296752`

[22] Holy, T.; et al. Pattern recognition of tracks induced by individual quanta of ionizing radiation in Medipix2 silicon detector. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, volume 591, no. 1, June 2008: pp. 287 – 290, ISSN 0168-9002, doi:10.1016/j.nima.2008.03.074. Available from: `http://www.sciencedirect.com/science/article/pii/S0168900208004592`

[23] Sopczak, A.; et al. Precision Luminosity of LHC Proton–Proton Collisions at 13 TeV Using Hit Counting With TPX Pixel Devices. *IEEE Transactions on Nuclear Science*, volume 64, no. 3, Mar. 2017: pp. 915–924, ISSN 1558-1578, doi:10.1109/TNS.2017.2664664. Available from: `https://ieeexplore.ieee.org/document/7842554`

[24] Demortier, L.; Lyons, L. Everything you always wanted to know about pulls. Technical report CDF/ANAL/PUBLIC/5776, Fermilab, Apr. 2008, version 3.00. Available from: `https://lucdemortier.github.io/assets/papers/cdf5776_pulls.pdf`

[25] Jenni, P.; Nessi, M. ATLAS Forward Detectors for Luminosity Measurement and Monitoring. Technical report CERN-LHCC-2004-010. LHCC-I-014, CERN, Geneva, Mar. 2004. Available from: `https://cds.cern.ch/record/721908`

[26] Adamczyk, L.; et al. Technical Design Report for the ATLAS Forward Proton Detector. Technical report CERN-LHCC-2015-009. ATLAS-TDR-024, CERN, May 2015. Available from: `http://cds.cern.ch/record/2017378`

[27] ROOT: analyzing petabytes of data, scientifically. [online], [cited 2020/12/09]. Available from: `https://root.cern/`

[28] Brun, R.; Rademakers, F. ROOT — An object oriented data analysis framework. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, volume 389, no. 1, Apr. 1997: pp. 81 – 86, ISSN 0168-9002, doi:10.1016/S0168-9002(97)00048-X, new Computing Techniques in Physics Research V. Available from: `http://www.sciencedirect.com/science/article/pii/S016890029700048X`

[29] Bergmann, B. Usage of the TPX3-clustering SW. Technical report, IEAP CTU, 2018.

[30] Béjar Alonso, I.; et al. *High-Luminosity Large Hadron Collider (HL-LHC): Technical design report*. CERN Yellow Reports: Monographs, Geneva: CERN, 2020, doi:10.23731/CYRM-2020-0010. Available from: `https://cds.cern.ch/record/2749422`

[31] Constraints and concepts (since C++20). [online], [cited 2021/05/06]. Available from: `https://en.cppreference.com/w/cpp/language/constraints`

[32] Comparison operators. [online], [cited 2021/05/06]. Available from: `https://en.cppreference.com/w/cpp/language/operator_comparison`

[33] Google. Material Design. [online], [cited 2021/05/05]. Available from: `https://material.io/`

[34] Google. Material Design Lite. [online], [cited 2021/05/05]. Available from: `https://getmdl.io/`

Appendix A

# Summaries

| | Timepix | Timepix3 |
|---|---|---|
| Bibliography | [1, 2] | [3] |
| Coupling | One device with two layers | Two single-layer devices |
| Operation modes | ToT/ToA/Counting mode | Simultaneous ToT & ToA |
| Readout scheme | Frame-based | Data-driven |
| Dead time | Whole chip for 90–100 ms | Individual pixels for 475 ns |
| Time resolution | 100 ns | 1.5625 ns |
| Maximum hit-rate | 100 kHz | 79 MHz |
| Clock frequency | dynamic; up to 100 MHz; usually 10 MHz | two clocks; 40 MHz and 640 MHz |
| Maximum bandwidth | 102 Mbit/s for 100 MHz | 5.12 Gbit/s |
| Neutron converters | Both layers; 4 regions: free, PE, PE+Al, $^6$LiF (Figure 1.2) | TPX3_4B and TPX3_9B; 3 regions: free, PE, $^6$LiF (Figure 1.5) |

Table A.1: Summary of differences between installed TPX and TPX3 devices

Appendix B

# Acronyms

| | |
|---|---|
| **$^6$LiF** | lithium-6 fluoride. |
| **AFP** | ATLAS Forward Proton. |
| **API** | application programming interface. |
| **ATLAS** | A Toroidal LHC Apparatus. |
| **CERN** | European Organization for Nuclear Research. |
| **CSS** | Cascading Style Sheets. |
| **CVMFS** | CERN Virtual Machine File System. |
| **EMEC** | Electromagnetic Endcap. |
| **GCC** | GNU Compiler Collection. |
| **HB** | heavy blob. |
| **HL-LHC** | High Luminosity LHC. |
| **HTML** | HyperText Markup Language. |
| **I/O** | Input/Output. |
| **ID** | identifier. |
| **IP** | Internet Protocol. |
| **LB** | luminosity block. |
| **LHC** | Large Hadron Collider. |

| | |
|---|---|
| **LUCID** | Luminosity Cherenkov Integrating Detector. |
| **LXPLUS** | Linux Public Login User Service. |
| **MDL** | Material Design Lite. |
| **MPX** | Medipix. |
| **ORM** | object–relational mapping. |
| **OS** | operating system. |
| **PDF** | Portable Document Format. |
| **PE** | polyethylene. |
| **PNG** | Portable Network Graphics. |
| **STL** | Standard Template Library. |
| **TCP** | Transmission Control Protocol. |
| **ToA** | time-of-arrival. |
| **ToT** | time-over-threshold. |
| **TPX** | Timepix. |
| **TPX3** | Timepix3. |
| **UML** | Unified Modeling Language. |
| **YAML** | YAML Ain't Markup Language. |

Appendix C

# Contents of enclosed CD

```
 readme.txt ...................... the file with CD contents description
 project.................................the directory of the project
   src .................................the directory of source codes
   makefile...................................the deployment script
   config.sh....................the environment configuration script
   default.yaml........................the default configuration file
 thesis ...............the directory of LaTeX source codes of the thesis
 text .......................................the thesis text directory
   thesis.pdf.........................the thesis text in PDF format
   thesis.ps............................the thesis text in PS format
 web ...................................the directory of the web page
```