Monitoring and calibration of the ALICE time projection chamber

Dag Toppe Larsen

Dissertation for the degree philosophiae doctor (PhD)
at the University of Bergen

June 2010

Contact: Dag Toppe Larsen
Institute of Physics and Technology
University of Bergen
Allégaten 55
5007 Bergen
Norway

Email: dagtl@ift.uib.no
Private email: dag.toppe.larsen@gmail.com

Download: http://web.ift.uib.no/~dagtl/dag-phd.pdf

Vi skal ikkje sova burt sumarnatta;
ho er for ljos til det.

[1]

Acknowledgements

I have spent the past seven years studying in the experimental nuclear physics group in Bergen. Throughout these very busy years, I have had the chance to expand my knowledge in physics, to meet many interesting people, to work on challenging tasks, to visit places of the world I might otherwise never see.

First of all, I would like to thank Professor Dieter Röhrich, my supervisor, for giving me the opportunity to work in the group, and for great guidance in both physics and life. Your intuitive and inspiring way of explaining physics has made the topic so much more interesting and enjoyable. I have deeply appreciated being your student.

A special thank goes to my co-supervisor, Professor Joakim Nystrand. You are always patient and helpful, and a good teacher of physics. I thank Professor Kjetil Ullaland, for being cheerful and joking, teaching me electronics, and helping with many practical issues. "Tusen takk" to Professor Håvard Helstrup, for always thoroughly answering my questions more like a friend, and final reading and commenting of my thesis. I thank Luciano Musa, for his guidance to the detector electronics. I am impressed by both your overview and detailed knowledge of the electronics. Many thanks go to Marian Ivanov, for introducing me to the complex field of calibration of the TPC, when I came to GSI I was new to this topic. All your great help have been essential to my thesis.

A large part of my work has been on the DCS for the TPC and PHOS. Matthias Richter, Sebastian Bablok, Johan Alme, Dominik Fehlker, Christian Lippmann and Attiq Ur Rehman provided valuable input and help to the development of the FeeServer in general, and for the TPC in particular, while Per-Thomas Hille and Øystein Djuvsland helped me with the PHOS FeeServer. I learnt a lot from you, and had a great time working with you!

I also had the chance to meet and work with the following colleagues and friends: Boris Wagner has been my office mate, we have had countless interesting discussions. I enjoyed working with Kalliopi Kanaki on the HLT part of the drift velocity calibration. Like me, Ketil Røed, Kenneth Aamodt, Gaute Øvrebekk, Øystein Haaland, Kyrre Skjerdal, Therese Sjursen, Alex Kastanas, Lijiao Liu, Meidana Huang, Henrik Qvigstad and Are Stangeland all had the chance to spend considerable time in both Bergen and at CERN. Especially at CERN, I often had the chance to join some of you, as well as Jochen Thäder, Magnus Mager and many others for lunch or dinner, or some completely

different activity. You have made my stays at CERN much, much more enjoyable!

The other major part of my work was with the TPC drift velocity calibration, because of which I frequently had to travel to GSI. During these stays, I enjoyed sharing office with Alexander Kalweit, who helped me through a rather steep learning curve with the calibration framework. I appreciated the stimulating environment at GSI, including the lunch and the following coffee breaks with colleagues in the group.

I would like to thank my parents Britt and Eiulf, and my brother Ken, your constant support could not mean more to me. Over the past years, I have been ever more frequently away, and had less time to spend with you. I am always looking forward to come home to you!

Special thanks go to my dear Hongyan for your constant love and encouragement. Thank you for reading through my thesis and giving valuable comments. Thank you for being there for me all time the past years. I have been very lucky to have met you and have you by my side.

There are also many other people who have helped me, in one way or another, to make it possible for me to finish this work — too many to be listed here. Thank you all.

Dag Toppe Larsen

Bergen, June 2010

Preface

The aim of the *A Large Ion Collider Experiment* (ALICE) experiment at CERN is to study the properties of the *Quark–Gluon Plasma* (QGP). With energies up to 5.5 $A\,TeV$ for $Pb+Pb$ collisions, the *Large Hadron Collider* (LHC) sets a new benchmark for heavy-ion collisions, and opens the door to a so far unexplored energy domain. A closer look at some of the physics topics of ALICE is given in Chapter 1.

ALICE consists of several sub-detectors and other sub-systems. The various sub-detectors are designed for exploring different aspects of the particle production of an heavy-ion collision. Chapter 2 gives some insight into the design.

The main tracking detector is the *Time Projection Chamber* (TPC). It has more than half million read-out channels, divided into 216 *Read-out Partitions* (RPs). Each RP is a separate *Front-End Electronics* (FEE) entity, as described in Chapter 3. A complex *Detector Control System* (DCS) is needed for configuration, monitoring and control. The heart of it on the RP side is a small embedded computer running the FeeServer software, providing a means for remote configuration and continuous monitoring of the FEE. Chapter 4 gives details of the implementation of this software, and also shows the performance measurements. In Chapter 5, potential improvements to the FeeServer class factorisation is discussed.

Converting the electronics signals, as measured by the sub-detectors, into useful physics data is a complicated process. This is called the calibration. Every sub-detector has its unique set of calibration tasks and challenges. Chapter 6 looks into some of the aspects of calibrating the electron drift of the TPC. This discussion is continued in Chapter 7, where the concrete AliRoot framework for some of the TPC calibration tasks is described. Chapter 8 dwells on the specifics of the TPC drift velocity calibration.

Finally, the status of the effort is given in Chapter 9.

Contents

List of Figures

List of Tables

Chapter 1

Ultra-relativistic heavy ion collisions

1.1 Heavy ion collision

The goal of heavy-ion collisions at ultra-relativistic energies is to study the property of strongly interacting systems, which are described by the theory of *Quantum Chromo-Dynamics* (QCD). It was predicted by QCD that a new state of matter, the so-called QGP can be created in such collision systems, if the temperature and energy density exceed a certain threshold, obtained by increasing the kinetic energy of the colliding beams. In this new state of matter, the constituent partons, namely quarks and gluons, are freed from nucleons in which they are normally confined by the strong force. The phase diagram (Figure 1.1) shows the different phases of strongly interacting matter. A phase transition separates hadronic matter from the QGP over a wide range of the baryon chemical potential μ_b; at small μ_b a cross-over is predicted at a critical temperature $T_c \approx [160, 170] MeV$ [4, 5].

In the laboratory, there have been decades of efforts in heavy-ion collisions at the highest possible energy, *Alternating Gradient Synchrotron* (AGS) [6, 7, 8], *Super Proton Synchrotron* (SPS) [9, 10, 11] and *Relativistic Heavy Ion Collider* (RHIC) [12, 13, 14]. For the fixed target experiment at AGS and SPS, the energy density might have been high enough to create this hot and dense matter [15]. Later on, the RHIC at *Brookhaven National Laboratory* (BNL) in the United States reached a centre-of-mass energy of $\sqrt{s_{NN}} = 200\ GeV$ in $Au + Au$ collisions. There have been several indications of creation of a new state of matter in the most central $Au + Au$ collisions at RHIC, *e.g.* jet quenching and high p_T suppression [16, 17], observed by the four major experiments: *Broad Range HAdron Magnetic Spectrometer* (BRAHMS), *Pioneering High-Energy Nuclear Interactions eXperiment* (PHENIX), PHOBOS and *Solenoidal Tracker At Rhic* (STAR).

The LHC facility at CERN has a design goal of 5.5 $A\ TeV$ for $Pb + Pb$ collisions, compared to the $Au + Au$ collisions at RHIC of 200 $A\ GeV$, allowing for higher energy

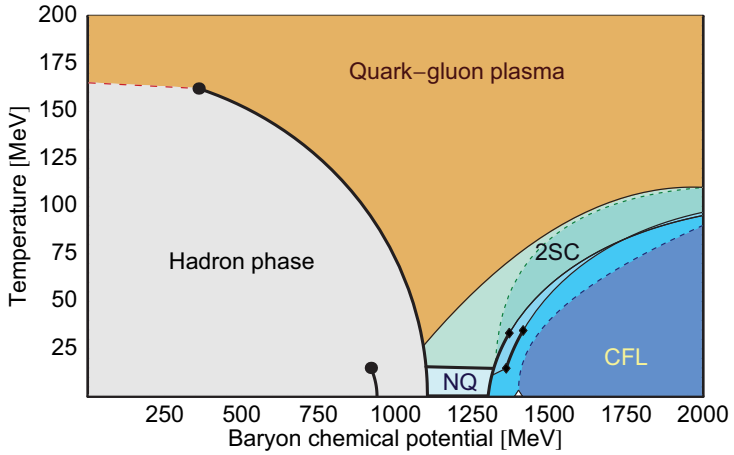

Figure 1.1: The phase-space diagram of strongly interacting matter. Different states are indicated. In general, most of the regions of the phase-diagram are to a large extent unknown, and further exploration of it will be an important topic for future experiments. [2]

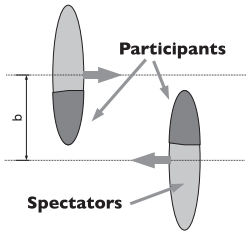

Figure 1.2: Schematic view of a heavy-ion collision. The *impact parameter* $\boldsymbol{b}$ is the vector between the centres of the nuclei at the collision. The *reaction plane* is defined by $\boldsymbol{b}$ and the projectile trajectory. [2]

densities and temperatures. This will give an opportunity to study the new state of matter — QGP — in detail, and furthermore to understand how the universe has been evolving to what it is now.

In the following, a short introduction to the heavy-ion physics will be given, followed by a brief discussion regarding the QCD theory and *Lattice QCD* (LQCD) calculations. Finally, two selected topics, flow and jet quenching, are discussed as evidences of creation of QGP, and current experimental observations will be shown.

According to the theory of relativity, nuclei at relativistic velocity are *Lorentz-contracted* in the direction of movement, making them appear as disk-like shapes. Consequently, a nuclear collision can be illustrated schematically as two approaching disks, as shown in Figure 1.2. The dark areas of the two approaching nuclei is the over-lapping region of the collision; the nucleons contained within are called *participants* of the collision. The remaining nucleons of the nuclei are called *spectators*, as they will continue

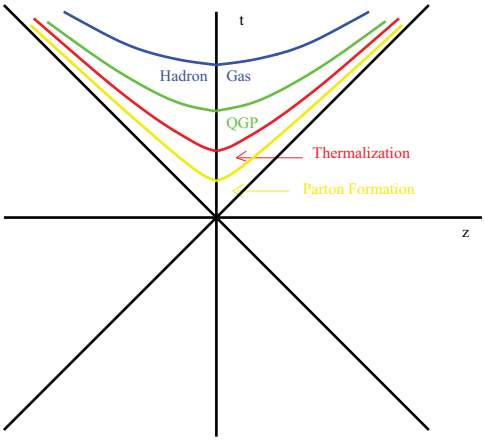

Figure 1.3: Light-cone collision space–time coordinate system. The four stages of the collision evolution are indicated in separate colours. [3]

moving with approximately original speed during and after the collision. An important parameter describing the centrality of a collision, is the vector between the centres of each nucleus, called the *impact parameter* $\boldsymbol{b}$. A smaller impact parameter indicates a more central collision. Also, the number of participants in each collision illustrates the centrality of a collision with the same projectile and target. When the impact parameter is small, the overlapping region is large and the total number of participants in the collision is large. In such collisions, the system is extremely heated and squeezed. Partons from participants of the highly compressed region interact with each other, creating a "soup" of free quarks and gluons. The number of particles produced in the collision can be used as a measure of the collision centrality.

The evolution of the collision can be divided into four main stages: initial parton scattering; fireball formation; hadronisation; and hadron freeze-out. It is convenient to describe these four stages in a light-cone coordinate system, Figure 1.3. The partons of the nucleons inside two colliding nuclei start scattering at partonic level when the nuclei hit each other. This is the initial condition of the collision when a fireball is created. A QGP may be produced inside the fireball if the energy density is sufficiently high. The expanding post-collision system cools down. When the temperature gets below a critical temperature threshold, hadrons freeze out.

There are two extreme scenarios for describing the interaction between two colliding nuclei. The *Landau* [18] picture is based on *full stopping* — all nucleons of the colliding nuclei come to a full stop, and the energy carried by the colliding nuclei is deposited in the vicinity of the collision centre-of-mass. This is a hydro-dynamical model, which implies a net-baryon distribution close to a Gaussian around mid-rapidity, and will result in a Gauss-distributed rapidity spectrum of produced particles. The *Bjorken* [19] picture, on

the other hand is based on *transparency* — most of the nucleons will only lose a small fraction of their energy and momentum. Hence, the nuclei will continue with almost their original speed after interaction. In this picture, the central region will be net-baryon free. Such a scenario is expected for LHC energies.

1.2 Perturbative quantum chromo-dynamics

The theoretical model describing heavy-ion collisions is QCD. It is based on non-abelian gauge theory, and is a part of the *standard model*. The strong force is described in terms of *colour* charge, carried by quarks and gluons. Quarks come in six *flavours*, grouped into three *generations*. In addition there are the corresponding anti-partners. The generations are *u*p, *c*harm and *t*op with charge $\frac{2}{3}e$ and their counterparts *d*own, *s*trange and *b*ottom with charge $-\frac{1}{3}e$. There are three colours: *r*ed, *g*reen and *b*lue; as well as three anti-colours for anti-particles. Each quark carries just *one* colour. The gluon is the force-exchange particle of the strong force, and will carry *both* colour *and* anti-colour. QCD requires all free particles to be colour neutral. Hence, free quarks can not exists, but are *confined* inside *hadrons*. Besides quark–anti-quark pairs of the corresponding colour–anti-colour — *mesons* — also hadrons with three quarks of all three colours — *baryons* — are colour-neutral [20].

Under the physics conditions found on earth, *nuclear matter* is the only stable phase. Only the two quarks with the lowest energy level, u and d, exist in nucleons. Probing other forms of matter requires creating physics conditions with temperatures and baryon densities significantly higher than those found in ordinary nuclear matter. This can be achieved by colliding heavy ions at ultra-relativistic speeds, which will break up the nucleons, and a short-lived state with the desired conditions is created. To the best of current understanding, a soup of quarks and gluons may be created in such collisions. This is called *quark matter*.

QCD predicts *asymptotic freedom*, *deconfinement* and QGP.

Asymptotic freedom stems from SU(2) gauge theory. The field of the strong force has two components, one Coulomb-like that decreases with the square of the distance, and one that increases linearly [20]. Hence, at short distances the quarks are only weakly bound by the strong force, and may be considered semi-free; or, *asymptotic* free.

When a threshold of temperature and density of baryons is approached, the baryons start to overlap, and distinct baryons gradually cease to exist as the temperature or density increases. Since the distances between the quarks are now very short, they may adhere to the principle of asymptotic freedom. Thus, the quarks are free to move; they are no longer confined.

When two ultra-relativistic heavy ions collide, the temperature and baryon density is expected to be high enough to create a comparatively large volume where a soup

of deconfined quarks and gluons may exist in equilibrium — the QGP. It is a direct implication of QCD and SU(3) group theory. Figure 1.1 shows the *phase-diagram*, where the region of QGP is indicated. The search for it and the study of its properties may be the most important topic of current ultra-relativistic collision experiments. As will be shown later, some experiments claim to have observed a form of QGP, or a more strongly coupled state — *Strongly coupled QGP* (sQGP) — at RHIC [21, 22].

QCD can not be solved exactly, however, effective methods exist [23]. The strong coupling α_s is reduced at decreasing space-time distances, *i.e.* high energies, and it is possible to describe the coupling as a perturbative expansion. Hence, it can be treated similar to the *Quantum Electro-Dynamics* (QED). This is called *perturbative QCD* (pQCD). On the other hand, at large distances, pQCD suffers severe problems, such systems can only be treated non-perturbativelly [23].

The rules of pQCD follow the Feynman rules, also allowing for gluon–gluon interactions [23]. However, the strong coupling is 30–100 times greater than that of QED, $\alpha/\pi = \mathcal{O}(10^{-3})$. Thus, it is often necessary to include higher-order Feynman diagrams to reach the required precision.

Equation 1.1 [2, 24] shows the pQCD calculation of the cross-section to produce a hadron h at given p_T. $f_{i/A}\left(x_1, Q^2\right)$ and $f_{j/B}\left(x_2, Q^2\right)$ are the *Parton Distribution Function* (PDF) (the momentum distribution for the partons of a hadron) for two hadrons A and B, respectively. $\frac{d\hat{\sigma}^{ij\to kl}}{d\hat{t}}$ is the differential cross section for the scattering $ij \to kl$, which can be calculated by pQCD at *Leading Order* (LO) or *Next-to-Leading Order* (NLO). $D_{k\to h}\left(z, \mu_F^2\right)$ is the fragmentation function describing the hadronisation of a parton k into a hadron h with a fraction z of the momentum. In vacuum, these quantities evolve with the fragmentation scale μ_F^2, which is obtained from global fits. x_1 and x_2 are are the fractions of the initial momentum carried by the partons. This is the standard factorisation for pQCD calculations of hard scattering [24]. The equation is also valid for nuclear collisions, assuming they can be considered pure super-positions of many $nucleon + nucleon$ collisions, without any medium effects.

$$\frac{d\sigma^{AB\to h}}{dp_T^2 dy} = \sum_{i,j,k=q,\bar{q},g} \int dy_2 \frac{dz}{z^2} x_1 f_{i/A}\left(x_1, Q^2\right) x_2 f_{j/B}\left(x_2, Q^2\right) \frac{d\hat{\sigma}^{ij\to kl}}{d\hat{t}} D_{k\to h}\left(z, \mu_F^2\right) \quad (1.1)$$

Medium effects will modify the cross-section

1.3 Lattice QCD

As was mentioned in the previous section, pQCD gives good results for short distances, but suffers severe limitations at long distances. Problems that involve calculations at such scale includes the freeze-out transition to hadronic matter [23].

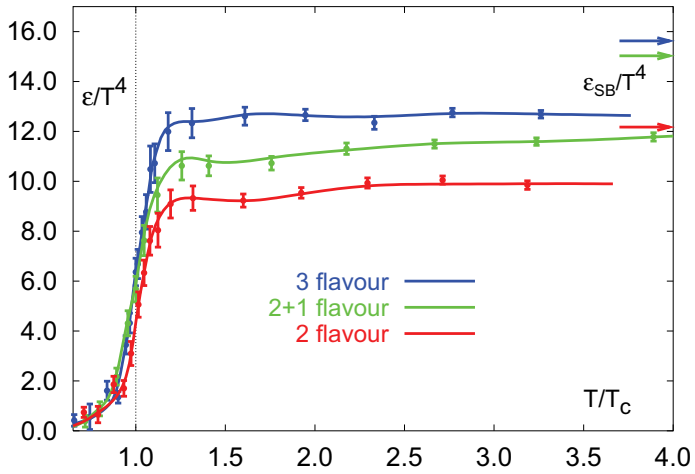

Figure 1.4: LQCD predictions for the energy density as function of temperature. Around the critical temperature T_c, *i.e.* $T/T_c = 1$, the active number of freedoms rises rapidly, and there is a phase transition from a hadronic to a partonic state of matter. The behaviour is typical of an ideal quark–gluon gas. LQCD predicts T_c to be in the range 160 to 170 MeV [4, 5]

To avoid such problems at large scales, an alternative approach is developed. LQCD is a numerical method, where the QCD interactions, rather than taking place in a continuum, are placed on a discrete *lattice*, with a limited lattice spacing a. Lattice calculations are highly suitable for parallel processing on computers. Better results can be obtained by decreasing the lattice spacing, thus also increasing the computing power needed. Although the method in general involves approximations to simplify the calculation, it does not involve those of pQCD. Hence, it can be applied to system of very large distances. However, LQCD can only predict thermodynamical variables of a system in equilibrium.

Usually, LQCD calculations are performed assuming *zero* baryon density. This may be an adequate description of systems with very low baryon density, such as the *Big Bang* and possibly the conditions at LHC. However, effort is being undertaken to improve the results of calculations with a *non-zero* baryon density.

Figure 1.4 shows the LQCD predictions for the energy density as function of temperature [4]. Around the critical temperature T_c, *i.e.* $T/T_c = 1$, the active number of freedoms rises rapidly, and there is a phase transition from a hadronic to a partonic state of matter. The behaviour is typical of an ideal quark–gluon gas. LQCD predicts T_c to be in the range 160 to 170 MeV.

1.4 QGP signatures

Ever since the start of RHIC there has been extensive discussions over the evidence for a new state of matter. A number of indications have been observed.

It is not possible to detect QGP directly in experiment, but it is possible to detect particles produced in relativistic heavy-ion collisions, where QGP might have been created. On one hand, it is possible to calculate the production of particles from heavy-ion collision using theoretical models, given either the presence or absence of a QGP in such processes. A clearly identifiable discrepancy between those two pictures is called a *signature* or *signal*. An important method is also the comparison of the result from heavy-ion collisions to those of proton collisions scaled by the number of binary collisions.

Several potential signatures have been proposed. *Electromagnetic probes*, such as direct photons can give information of the early stage of the process since they do not interact strongly with the medium. The *J/ψ quarkonia production* is expected to be sensitive to the matter due to its interaction with the medium. Existence of collective *flow* at partonic level can be understood as a signature of the strongly interacting nuclear matter. *High-p_T* suppression and jet quenching also give hints to the existence of the deconfined state of matter, because high-p_T particles are expected to be moderated by the medium, and the away-side jet is very probably absorbed by the QGP. In addition, *Hanbury-Brown Twiss* (HBT) source size measurements and *chiral symmetry restoration* can provide signatures for QGP.

The four major experiments at RHIC [25, 26] are dedicated to measure such QGP signals. Their measurements of three very interesting signals, namely the collective flow, high-p_T suppression and jet quenching, will be discussed in the following. In general, the measurements seem to be consistent with a sQGP state of matter. This is a strong hint that also a "real" QGP should exist at even higher energies.

1.4.1 Collective flow

During the early stage of the collision, pressure gradients inside the collision volume are created, and in turn give rise to *collective* flow among particles. The amplitude of flow is dependent on collision energy and centrality, and the property of the produced matter, *e.g.* its compressibility, and can reveal information of the early stage and development of the collision [27, 28, 29].

Modern flow analysis mainly distinguishes between three types of flow [30]. The two first, *in-plane* and *out-of-plane* transverse flow (relative to projectile trajectory), are relevant for non-central collisions where a *reaction plane* can be defined. The reaction plane is defined by the impact parameter and the direction of movement of the projectile nuclei, as shown in Figure 1.5. In-plane flow is associated with particles emitted in the reaction plane, whereas out-of-plane flow is particles emitted approximately perpendic-

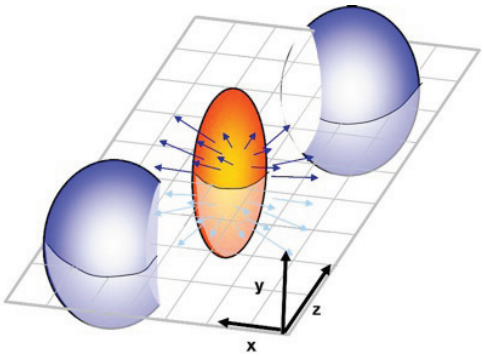

Figure 1.5: Collision geometry and definition of the reaction plane, namely the x–z plane defined by the impact parameter and direction of the colliding nuclei.

ular to the reaction plane. For collisions in the lower part of the energy domain, the presence of the spectators will block the emission of in-plane flow in the direction of the spectators, leading to enhanced out-of plane flow, where there are no spectators. At ultra-relativistic energies this effect is much smaller since spectators vanish much faster. Rather, pressure gradients will enhance in-plane flow. The third type of flow is *radial* flow, meaning the flow is isotropic in all directions. This is relevant for the most central collisions where it is not possible to define a reaction plane, and the pressure gradients are isotropic.

For high-energy, non-central collisions, it is useful to describe flow in terms of the Fourier transform of the azimuthal distribution of particles. In the most general case, the differential distribution of produced particles can be written in the form of a Fourier series with respect to the reaction plane [31, 32], as in Equation 1.2. The Fourier coefficients are given by Equation 1.3, which is averaging over all outgoing particles from the collision.

$$E\frac{d^3N}{d^3p} = \frac{1}{2\pi}\frac{d^2N}{p_t dp_t dy}\left(1 + \sum_{n=1}^{\infty} 2v_n \cos\left[n\left(\phi - \psi_r\right)\right]\right) \tag{1.2}$$

$$v_n = \langle cos\left[n\left(\phi - \psi_r\right)\right]\rangle \tag{1.3}$$

For these equations, ψ_r is the azimuthal angle of the reaction plane relative to the fixed experiment frame, and ϕ is the azimuthal angle of each produced particle, also in the experiment frame. $\psi_r - \phi$ is the azimuthal angle of the particle in the relative reaction plane frame. The first coefficient, v_1, is the *directed flow*, meaning the overall flow has an offset in either *direction* along the x-axis as result of a uneven momentum transfer from the two colliding nuclei to the fireball. The second coefficient, v_2 is the

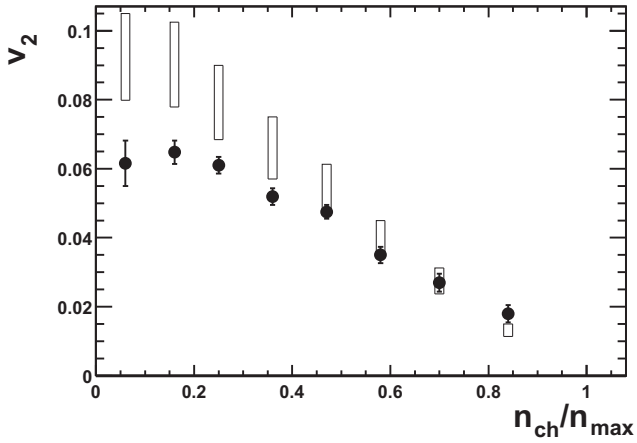

Figure 1.6: Centrality dependence of v_2 in $Au + Au$ collisions at $\sqrt{s_{NN}} = 130\ GeV$, measured by the STAR experiment at RHIC, compared to expectations from a hydro-dynamical model [33]. Solid points represents the measurements, open rectangles show the range of values expected in the hydro-dynamic limit.

elliptic flow, the ratio of in-plane to out-of-plane flow.

A QGP is expected to behave close to a perfect fluid, where the hydro-dynamical model will apply. Flow measurements in agreement with this can be a QGP signature.

Figure 1.6 shows the centrality dependence of v_2 in $Au + Au$ collisions at $\sqrt{s_{NN}} = 130\ GeV$, measured by the STAR experiment at RHIC, compared to expectations from a hydro-dynamical model [33]. Solid points represents the measurements, open rectangles show the range of values expected in the hydro-dynamic limit. The measurements are well explained by the hydro-dynamical model.

Figure 1.7 shows v_2 normalised by number of constituent quarks as function of p_T normalised by number of constituent quarks for various charged particles in minimum bias $Au + Au$ collisions, measured by the STAR experiment at RHIC [34]. The measurements show a nice scaling, which is an indication of collectivity developed at partonic level.

Figure 1.8 shows v_2 as function of p_T for various charged particles in minimum bias $Au + Au$ collisions at $\sqrt{s_{NN}} = 200\ GeV$, measured by the STAR experiment at RHIC [35]. Ω and ϕ contain the heavier s-quark, whereas π and p do not. Despite the difference in mass, there is no or little difference in p_T dependence, also implying that collective flow was developed already at a partonic level.

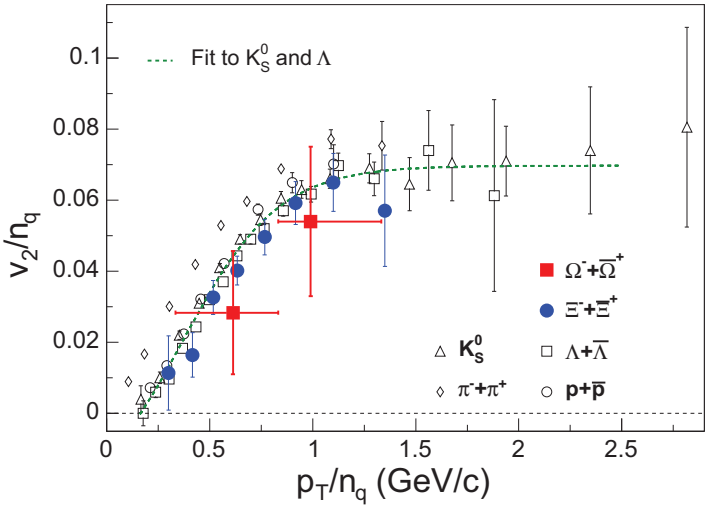

Figure 1.7: v_2 normalised by number of constituent quarks as function of p_T normalised by number of constituent quarks for various charged particles in minimum bias $Au + Au$ collisions, measured by the STAR experiment at RHIC [34].

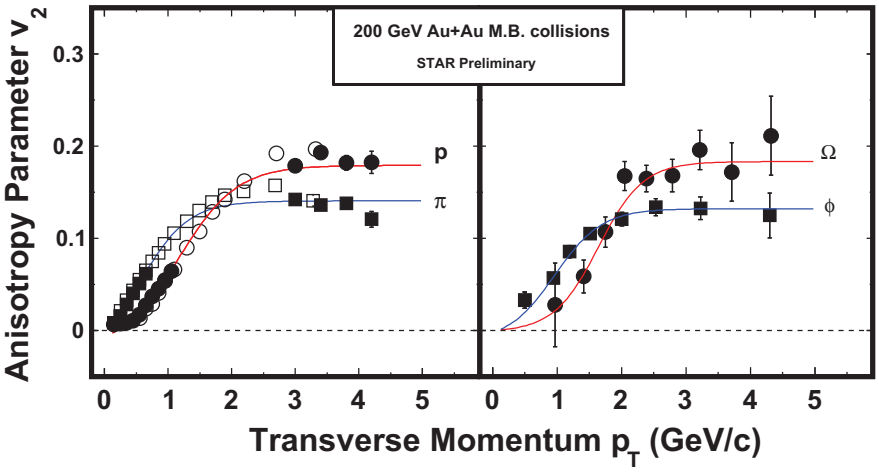

Figure 1.8: v_2 as function of p_T for various charged particles in minimum bias $Au + Au$ collisions at $\sqrt{s_{NN}} = 200\ GeV$, measured by the STAR experiment at RHIC [35]. Ω and ϕ contain the heavier s-quark, whereas π and p do not.

1.4.2 High-p_T suppression and jet quenching

Production of a QGP in the centre of the collision implies that a fast particle traversing through the matter will see a strongly interacting coloured state of matter — partonic matter — rather than a colour neutral state of matter — hadronic matter. This will have consequences for the propagation. A hard collision may be described using pQCD, where the partons contained in the initial nuclei are scattered off each other, and finally fragment into hadronic showers. A particle with high p_T traversing the medium of partonic matter, may lose energy through bremsstrahlung radiation of gluons, and its momentum is distributed over a larger number of partons. Hence, high-p_T particles are suppressed.

High-p_T particles produced in $p+p$ collisions provides information on pQCD and the PDF in protons, as well as the fragmentation functions of the partons. For relativistic heavy-ion collisions, it is a sensitive probe of the strongly interacting matter produced in the collisions, because a modification of their momentum distribution may be due to an energy loss mainly via gluon radiation induced by soft collisions of the leading partons or the radiated gluons in the medium.

The nuclear modification factor is a tool to quantify nuclear effects on particle production in $A+A$ collisions with respect to that in $p+p$ collisions, which is defined as the ratio of particles produced in $A+A$ collisions to that in $p+p$ collisions, scaled by the average number of binary collisions in $A+A$ collisions, as in Equation 1.4 [2]. One would expect $R_{AA}=1$ if nuclear collisions are simple superpositions of $p+p$ collisions, without any nuclear effect. RHIC has measured the R_{AA} for the range of energy available in these experiments [16, 17].

$$R_{AA} = \frac{d\sigma_{AA}/dp_T^2dy}{\langle N_{coll}\rangle d\sigma_{pp}/dp_T^2dy} \tag{1.4}$$

Figure 1.9 shows R_{AA} as function of centrality for high-p_T particles, measured by the STAR (upper) and PHENIX (lower) experiments at RHIC [16]. Circles show measurements, rectangles show the range of values expected in theoretical calculations with parton energy loss. There is stronger suppression for high-p_T particles in central collisions than in peripheral collisions. In central collisions, with a large number of participants, a strong suppression is observed. In peripheral collisions, on the other hand, only a few nucleons are involved as participants, and a R_{AA} close to one is seen. With just a few participants, the situation of a peripheral collisions is similar to that of $p+p$ collisions, and no, or very little, nuclear modification is expected. The suppression at central collisions might be attributed to the presence of a QGP. High-p_T particles traversing the colour-dense QGP will lose some of the momentum, *i.e.*, there will be fewer high-p_T particles observed compared to the non-QGP situation.

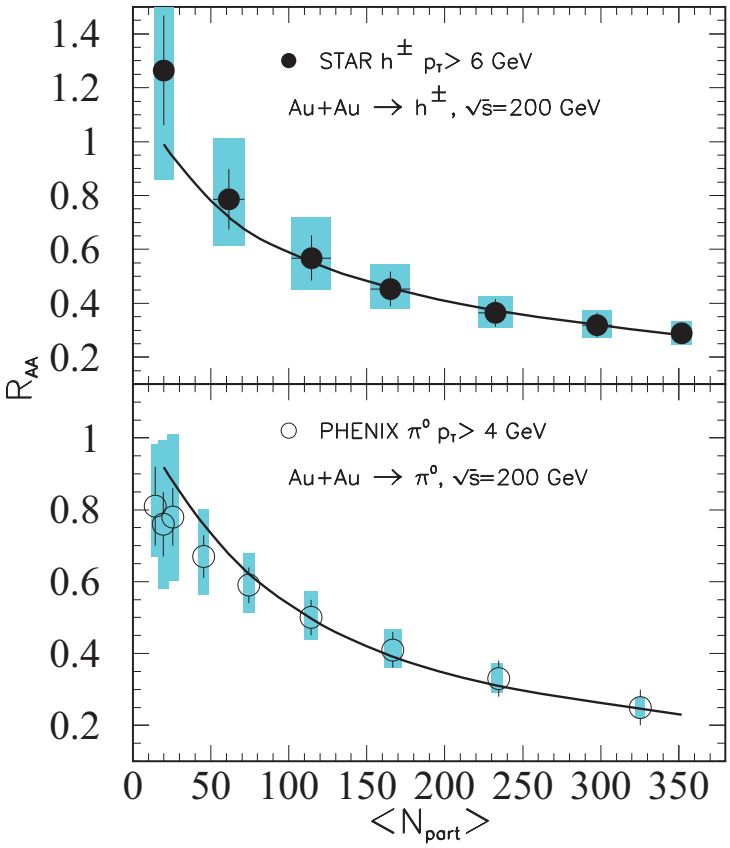

Figure 1.9: R_{AA} as function of centrality for high-p_T collisions, measured by the STAR (upper) and PHENIX (lower) experiments at RHIC [16]. Circles show measurements, rectangles show the range of values expected in theoretical calculations with parton energy loss.

Figure 1.10 shows the R_{AuAu} and R_{dAu} as function of p_T for central $Au + Au$ and minimum bias $d + Au$ at $\sqrt{s_{NN}} = 200\ GeV$, respectively, measured by the BRAHMS experiment at RHIC [17]. While R_{dAu} shows an enhancement at intermediate p_T, the central R_{AuAu} collisions show a significant suppression at high-p_T. The different results show that the high-p_T particles are very likely suppressed by the hot and dense medium created in central $Au + Au$ collisions.

Figure 1.11 shows a compilation [36] of R_{AA} of collisions at $\sqrt{s_{NN}} = 200\ GeV$, measured by the PHENIX and STAR experiments at RHIC [37, 38, 39, 40]. Panel *a* shows R_{dAu} and R_{AuAu} for hadrons in minimum bias collisions. While R_{dAu} is enhanced for $p_T > 2\ GeV/c$ because of the Cronin effect [41], R_{AuAu} is suppressed. Panel *b* shows R_{dAu} and R_{AuAu} for η and π^0 at central collisions. For R_{dAu} neither suppression nor enhancement is observed, but central $Au + Au$ collisions show a suppression for both

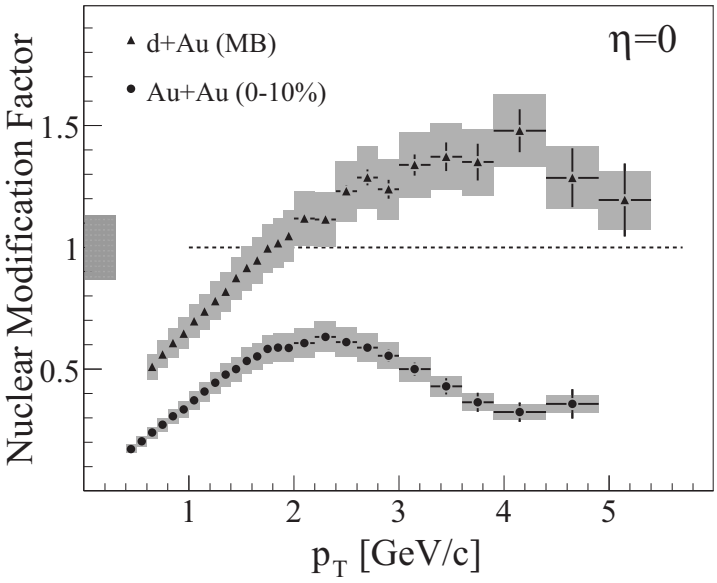

Figure 1.10: R_{dAu} and R_{AuAu} as function of p_T for central $Au + Au$ and minimum bias $d + Au$ at $\sqrt{s_{NN}} = 200\ GeV$, respectively, measured by the BRAHMS experiment at RHIC [17].

particles. Panel *c* shows R_{AuAu} for direct γ, η, π and $\pi^{\pm}$ in central collisions. The direct γ do not interact strongly, and are not suppressed. On the other hand, η, π^0 and $\pi^{\pm}$ are suppressed. The different behaviours indicate that the suppression is due to the coloured medium.

Jets are high-momentum hadron showers emitted as a results of the hard scattering of partons from nucleons of the colliding nuclei, and are produced from the hadronisation of a back-to-back quark–anti-quark pair, as shown in Figure 1.12. They are emitted back-to-back for momentum conservation. The presence of a QGP is assumed to make them suffer strong energy loss through induced gluon radiation while traversing the colour-dense medium. The energy loss scales with the distance which the jet has to traverse through the medium. For a back-to-back jet pair produced at the edge of a central collision fireball, the jet with the shorter exit distance will be detected almost unattenuated, whereas the jet which has to pass through most of the fireball may be completely suppressed [16]. This is called *jet quenching*.

Figure 1.13 shows dihadron azimuthal correlations of high-p_T particles measured by the STAR experiment at RHIC [13]. The left panel shows the near-side and away-side jets in minimum bias $p + p$ and central $d + Au$ and $Au + Au$ collisions. In the $p + p$ and $d + Au$ collisions, both jets are visible, whereas in $Au + Au$ collisions, the away-side jet is almost completely suppressed. This is believed to be caused by the presence of a QGP,

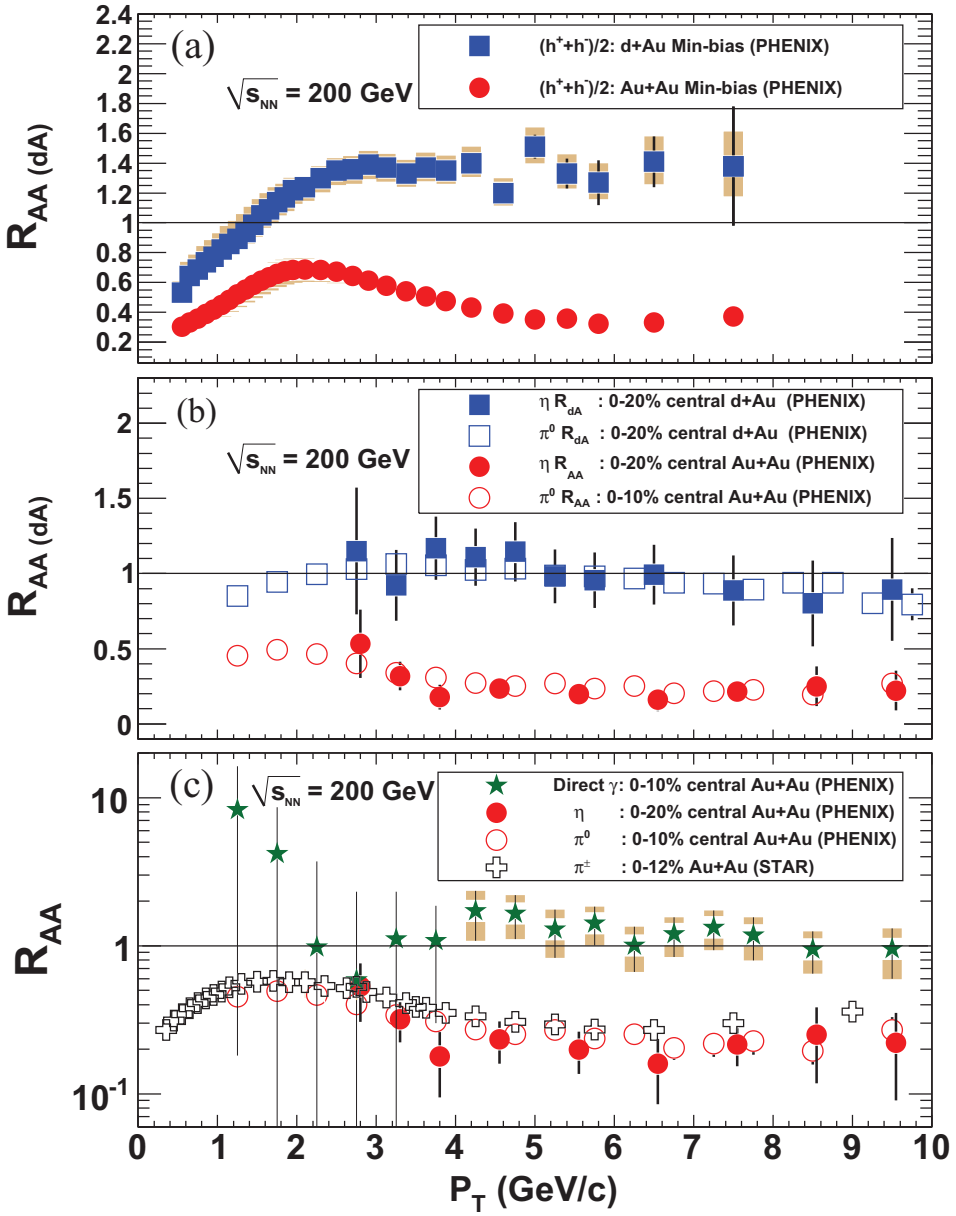

Figure 1.11: Compilation [36] of R_{AA} of collisions at $\sqrt{s_{NN}} = 200\ GeV$, measured by the PHENIX and STAR experiments at RHIC [37, 38, 39, 40]. Panel *a* shows R_{dAu} and R_{AuAu} for hadrons in minimum bias collisions. Panel *b* shows R_{dAu} and R_{AuAu} for η and π^0 at central collisions. Panel *c* shows R_{AuAu} for direct γ, η, π and $\pi^{\pm}$ in central collisions.

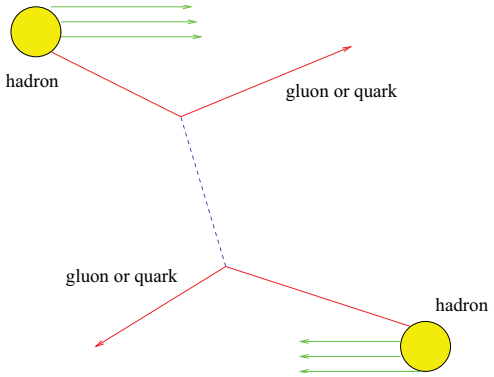

Figure 1.12: Jet production by the leading back-to-back quark–anti-quark pair. A hadronic shower is produced. [3]

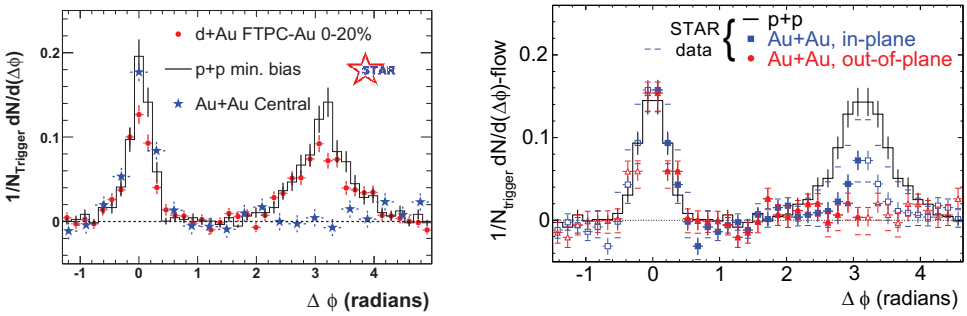

Figure 1.13: Dihadron azimuthal correlations of high-p_T particles measured by the STAR experiment at RHIC [13]. The left panel shows the near-side and away-side jets in minimum bias $p+p$ and central $d+Au$ and $Au+Au$ collisions. In the $p+p$ and $d+Au$ collisions, both jets are visible, whereas in $Au+Au$ collisions, the away-side jet is almost completely suppressed. The right panel shows the suppression of in-plane and out-of-plane jets in $Au+Au$ compared to $p+p$.

absorbing the jet traversing it. In central $Au+Au$, the near-side jets, close to the edge of the fireball, will show no or little suppression, since it only has to traverse the QGP for a short distance. The away-side jet, on the other hand, has to traverse most of the fireball, and will be almost completely absorbed. In $p+p$, $p+Au$ and peripheral $Au+Au$ collisions, the energy deposition in the collisions region does not allow for the creation of a QGP, and both jets escape. The right panel of Figure 1.13 shows the suppression of in-plane and out-of-plane jets in $Au+Au$ compared to $p+p$. Because of the geometry of the fireball, the jets have to traverse a longer distance inside it when they are emitted out-of-plane than in-plane. Accordingly, the suppression is larger for out-of-plane jets.

* * *

The results from RHIC indicate that the matter found in central $Au + Au$ collisions is deconfined, but behaves like a perfect liquid, *i.e.* still shows strong correlations. The collisions at higher energies (at LHC) might produce matter which is not only deconfined, but also weakly interacting, *i.e.* an "ideal" QGP.

Both, flow studies and jet reconstruction and azimuthal correlations, require a well calibrated tracking system. The TPC is the main tracking detector of ALICE. The DCS continuously monitors the TPC operating parameters, *e.g.* temperature, pressure and gas composition. These parameters are stored and used for the calibration of the TPC. The two main foci of this thesis will be selected aspects of the TPC DCS and calibration: (a) the software for maintaining and controlling the read-out electronics (the *FeeServer*) and (b) the drift velocity calibration and its calibration framework.

Chapter 2

A large ion collider experiment

ALICE [42] is an experiment at the LHC dedicated to heavy-ion physics, residing in the experimental cavern of the previous L3 experiment. It is aiming at re-creating the conditions of the early universe before the on-set of confinement and to study the properties of the QGP. Most of its sub-detectors are confined inside the L3-magnet. With a magnetic field strength up to about 0.5 Tesla, it is the largest conventional magnet of its size. The magnetic field is parallel to the z-axis. Figure 2.1 shows cut-through view of the ALICE detector.

2.1 Large hadron collider

The LHC, Figure 2.2, is a 28-km circumference accelerator and storage ring for protons and heavy ions, installed in the tunnel of the previous *Large Electron–Positron Collider* (LEP) accelerator. It is buried approximately 100 m below surface and has eight equally spaced caverns, of which four are utilised by the main experiments: *A Toroidal Lhc ApparatuS* (ATLAS) (point *1*), ALICE (point *2*), *Compact Muon Solenoid* (CMS) (point *5*) and *LHC-Beauty* (LHCb) (point *8*). The remaining points are used for *Radio Frequency* (RF) system (point *4*), beam dump (point *6*) and beam cleaning (points *3* and *7*). The RF system is responsible for accelerating the beam. Beam dumps are used when disposing the beam, while the beam cleaning facilities are collimators that remove particles which are either spatially or momentum-wise far away from the their particle bunch.

Two particle beams are accelerated in opposite directions. In contrast to LEP, separate beam pipes for each direction are required since LHC does not collide particles–anti-particles. At the four experimental sites, the beam pipes cross each other, and the beams of opposite directions are focused to collide with each other. The beams are not continuous streams of particles, but divided into 2808 bunches of approximately 1.15×10^{11} protons each. At full energy, the bunch length is 7.55 cm.

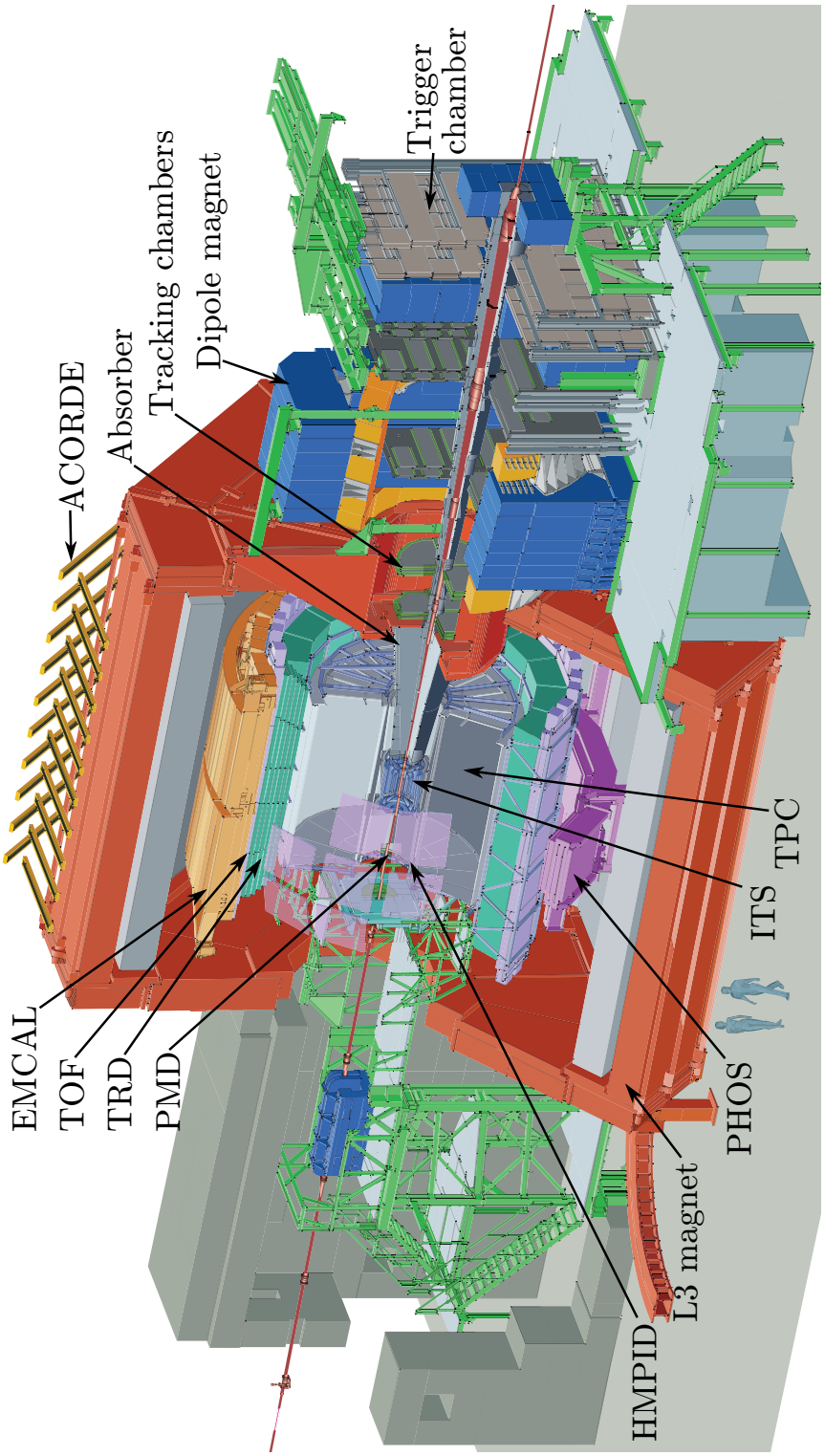

Figure 2.1: Schematic view of ALICE.

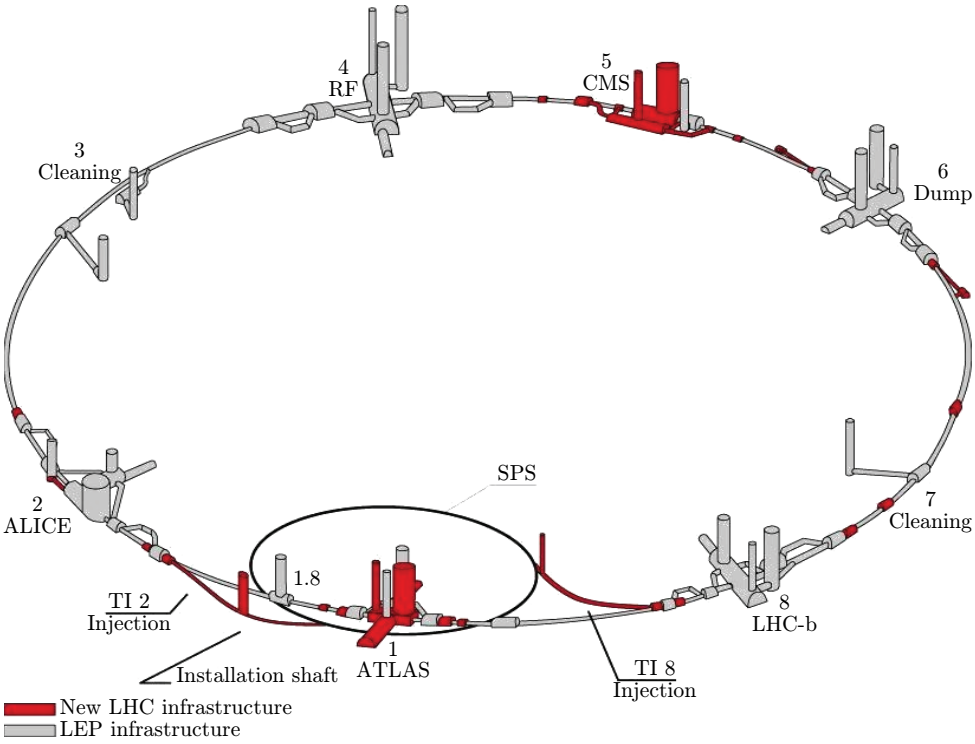

Figure 2.2: Schematic view of the LHC.

For protons and lead the respective energies for each beam are 7 TeV and 2.76 TeV per nucleon, giving total centre-of-mass collisions of $\sqrt{s} = 14\ TeV$ and $\sqrt{s_{NN}} = 5.5\ TeV$. A luminosity of $10^{27} cm^{-2} s^{-1}$ is expected for $Pb+Pb$, and $10^{34} cm^{-2} s^{-1}$ for $p+p$, although for ALICE the proton beams will be defocused to provide a luminosity of about $3 \times 10^{30} cm^{-2} s^{-1}$. The Protons are initially accelerated by *LINear ACcelerator* (LINAC) *2*, then through the *Proton Synchrotron* (PS) booster, PS, SPS, and finally injected into the LHC. The sequence is slightly different for lead: LINAC *3*, *Low Energy Ion Ring* (LEIR), PS, SPS, LHC.

2.2 ALICE sub-detectors

2.2.1 Time projection chamber

The TPC [46] is the main tracking detector in the ALICE experiment. Apart from tracking, measuring the charged particle momentum and having a good two-track separation, it also provides *Particle IDentification* (PID) via energy loss of particles going through the TPC. The TPC is expected to perform well at multiplicities as high as $dN_{ch}/d\eta = 8000$ in the particle momentum range $[0.1,\ 100]\ GeV/c$ within $|\eta| < 0.9$.

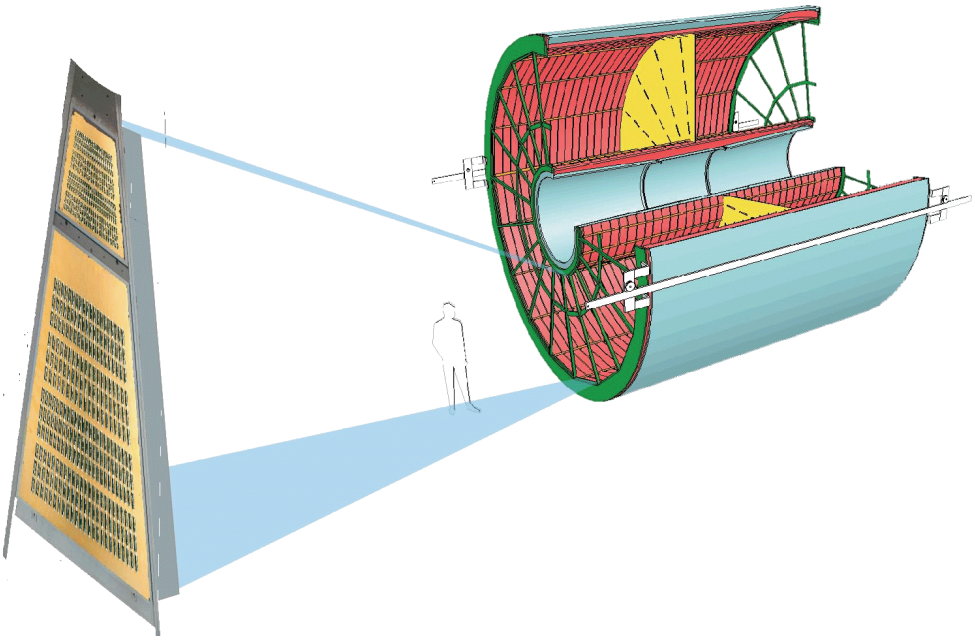

Figure 2.3: Schematic view of the TPC. The MWPCs of one sector enlarged for visibility. A person is shown to the scale of the TPC.

Tracking efficiency is required to be better than 90 %, and the dE/dx resolution better than 10 %. Further, the TPC alone will have a momentum resolution of about 1 % at 2 GeV/c and 10 % at 50 GeV/c. For $p+p$ collisions a read-out rate of about 1.2 kHz is expected, while for central $Pb+Pb$ collisions about 0.3 kHz [47].

The TPC is shaped as a horizontal cylinder, divided by a 100 kV *Centre-Electrode* (CE) in two 250 cm drift volumes along the length axis. The active radial region is 85 cm to 247 cm.

The 90 m^3 drift volume is filled with a counting gas composed of 85.7 % Ne, 9.5 % CO_2 and 4.8 % N_2. A cold, light gas is used to assure low diffusion and low multiple scattering. Field distortions are minimised because of the high ion mobility and few ionisation electrons per unit length. The electronics design noise figure is 1000 RMS e^- (700 actually achieved); not limiting the position resolution will require a signal/noise ratio of at least 20. Given the pad sizes and the small ionisation energy, a rather strong gas gain of up to 2×10^4 is needed. The electric drift field of 400 V/cm in combination with the gas mixture gives a drift time of 92 μs. The drift velocity is non-saturated, which in turn requires the temperature stability and homogeneity inside the TPC to be within 0.1 K [46].

Data read-out is performed at the two opposing detector end planes, which are divided in 18 azimuthal sectors. Each sector is again divided radially into two MWPC: the *Inner*

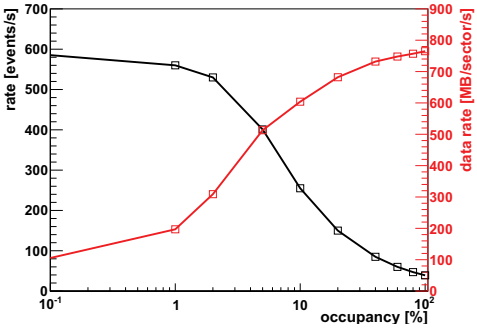

Figure 2.4: Event rate (black, left scale) and data rate (red, right scale) as function of occupancy, for full TPC read-out. At 100 % occupancy the theoretical maximal data rate of 770 MB/s is reached. At 0 % occupancy the data rate is 595 Hz, however applying sparse read-out increases this to 1386 Hz (not shown, as it only significantly departs at low occupancy). [43]

Read-Out Chamber (IROC) and *Outer Read-Out Chamber* (OROC). In total for the TPC, the MWPCs have 557 568 read-out pads. Combined, the two sector chambers are read out by six RPs: two for the IROC and four for the OROC. An RP is an electronic entity, and consists of a *Read-out Control Unit* (RCU) with up to 25 *Front-End Cards* (FECs). Eight *ALice Tpc Read-Out* (ALTRO) chips are mounted on a FEC, each is capable of reading out 16 read-out pads. Data is forwarded from the RCU via a 1.25 Gb/s optical fibre. A DCS board equipped with an embedded *Advanced Risc Machine* (ARM) processor running Linux is attached to the RCU for control and monitoring. Radiation tolerant electronics is needed to sustain the radiation from the collisions.

Data from the RPs are forwarded to the *Data AcQuisition* (DAQ) and the *High Level Trigger* (HLT) using optical fibres. The geometrical organisation of the RPs gives a total of 216 fibres, six per sector. Each fibre is capable of a data rate of 1.25 Gb/s, corresponding to 160 MB/s. Depending on the radial position in the sector, a RP may have 25 (innermost) to 18 (outermost) FECs. Reserving the same bandwidth for each FEC, only the most populated RPs can utilise the full bandwidth.

Consequently, for the full sector, the maximum data rate is 770 MB/s. Benchmark tests, Figure 2.4, show that this is indeed achievable for high-occupancy events where zero-suppression has been applied. The test was conducted using 1000 time bins and all channels filled with same data. Considering the case of low-occupancy events, read-out is possible at an event rate of 595 Hz (0 % occupancy) using full read-out. The electronics also supports sparse read-out, in which case empty channels are entirely stripped, including headers. Applying this technique, the read-out rate more than doubles to 1386 Hz. The data rate in these two cases are 70 MB/s and 927 kB/s, respectively. There is ongoing effort to increase this rate even further by optimising the read-out firmware of the RCU [48, 47].

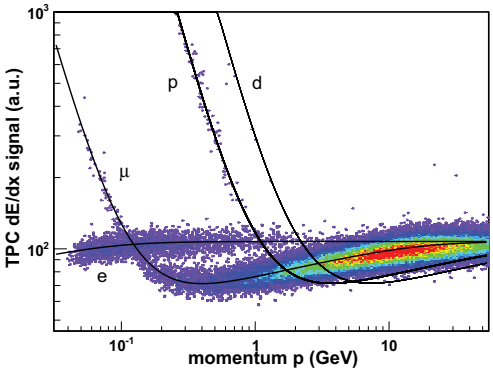

Figure 2.5: A demonstration of particle identification by TPC with cosmic data (2008). Particles can be identified by a few σ cut around the Bethe-Bloch function (black curves) using the TPC signal dE/dx and momentum. [44]

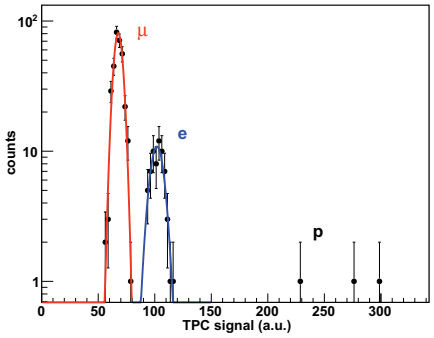

Figure 2.6: TPC dE/dx distribution within $500 < p < 600$ MeV/c momentum bin, shows good separation of low momentum electrons and muons. [44]

Maintaining the noise at a lowest possible level is important to achieve low data rates applying zero-suppression. The noise figure is required to be less than 1000 e^- *Root Mean Square* (RMS) of base-line, corresponding to one *Analogue–Digital Converter* (ADC) count. Noise levels are obtained from periodically taken pedestal runs, showing that the noise figure is about 0.7 ADC count (700 e^-), well within the requirement. They are close to the natural limit, and do not change with time. Using sparse read-out, a zero-suppressed empty event is less than 70 kB (noise); without zero-suppression 10000 times larger.

As mentioned above, the TPC requires the spatial temperature variations to be no more than 0.1 K. Also the variations with time have to be minimised as this will have impact on the drift velocity. An additional complication is the 27 kW of heat generated from the FEE on the end planes. The cooling is accomplished by wrapping the

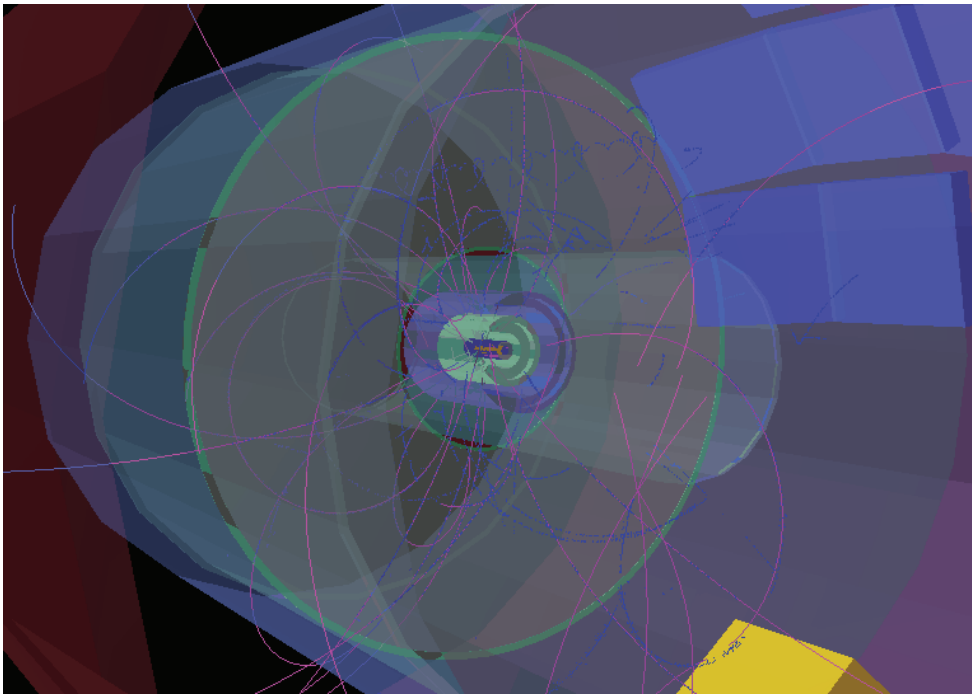

Figure 2.7: The first collision event recorded by the TPC. The event is shown with the ALICE event monitor. [45]

FEE in custom copper water-cooled envelopes, as well as water cooling of the chamber bodies themselves. There is also heat-screening towards the other detectors and the environment. The water flow is handled by about 60 independently adjustable circuits. To protect the FEE from leaks, the cooling circuits are under-pressured. About 500 temperature sensors have been installed to monitor the temperature fluctuations inside the TPC. Measurements during commissioning show temperature variations with $\sigma_T = 0.1\ K$ and $\Delta T_{max} = 0.3\ K$. At this time some cooling loops were not yet fully operational, which caused out-liers in the temperature measurements. It is expected that the required temperature homogeneity and stability of $\Delta T_{max} = 0.1\ K$ is achieved with the cooling system fully operational for all loops.

The TPC has currently a resolution of 5.7 % for dE/dx, determined from 7×10^6 cosmic events [46]. This is very close to the required resolution of 5.5 %, and will allow for particle identification up to 50 GeV/c. Figure 2.5 shows the measured dE/dx signal of cosmic tracks compared to the Bethe–Bloch equation for various particles, and in Figure 2.6 the dE/dx distribution for the momentum range $500 < p < 600\ MeV/c$.

For space points, a resolution for $r\varphi$ in the range $[300,\ 800]\ \mu m$ has been achieved for tracks with high momentum, *i.e.* small inclination angles. This is in agreement with results from simulations.

Momentum resolution is determined by tracking cosmic muons independently in the upper and lower halves of the TPC, then comparing p_T at the centre of the beam line. Hence, p_T resolution can be plotted as function of p_T. Currently, the achieved resolution, 6.5 % at 10 GeV/c, is not yet in agreement with the requirement of 4.5 % at 10 GeV/c. A problem has been identified in the software and is expected to be solved soon.

Construction of the TPC was completed in 2006. Pre-commissioning was carried out at the assembly site on the surface, close to the underground experimental area. Subsequently it was lowered into the experimental shaft to be integrated in the overall experiment. Commissioning was going on until summer 2008, when beam was expected. During this period the performance was gradually improved, and is now in accordance with the requirements. On November 20th 2009 the ALICE TPC saw the first collisions. Figure 2.7 shows the first collision recorded by the TPC. While the commissioning phase is completed, there is still work ongoing to further improve and extend the software side, both for calibration and analysis.

2.2.2 Photon spectrometer

PHOton Spectrometer (PHOS) [49] is a high resolution electromagnetic calorimeter. Using lead tungstate crystal $PbWO_4$ scintillators (20 radiation lengths, X_0) to absorb the incoming photons, it achieves a very high energy resolution. For photons of 1 GeV/c, $\sigma_E/E = 0.04$.

PHOS is designed to have five modules, of which three are installed so far. Most likely only four will be installed. The modules are mounted in a "cradle" with a radius of 4.6 m, designed to align the front of each module towards the centre of ALICE, where collisions take place. In the ϕ direction, it covers the lowermost 100° (20° per module). Along the z-axis, it covers about one metre; the total phase-space coverage is 3.7 %.

PHOS can measure π^0 and η via γ-decay with a p_T up to 100 GeV/c, which makes it well-suited for R_{AA} measurements for π^0 and direct γ, as well as hadron correlations with γ and π^0. Measurements of initial temperature via direct photon spectra and searching for signatures of chiral symmetry restoration are other areas where PHOS is designed to perform well.

From an electronics point of view, there are very many similarities to the TPC electronics. Both rely on the same scheme of a DCS board and an RCU with two branches of FECs. Also, the DCS software is similar. The main difference is the design of the FECs. Although the FECs of PHOS also rely on the ALTRO for processing, it only has half as many channels per FEC as TPC. Also, as PHOS uses *Avalanche Photo-Diode* (APD) for detecting the scintillation light, the signal level and shape is different than for the TPC, hence also a different amplifier and shaper is used. In addition, each APD needs a unique voltage level in the range $[300, 400]$ Volt. Dedicated hardware on the FEC

supplies this individual electrical potential to each APD. The correct level is set from the *Board Controller* (BC) of the FEC. PHOS has a special FEC, the *Trigger Read-out Unit* (TRU), for fast read-out of the deposited energy to generate level *0* and *1* triggers.

2.2.3 Electro-magnetic calorimeter

Electro-Magnetic CALorimeter (EMCAL) [50] shares many similarities with PHOS: they are both calorimeters, and rely heavily on the same electronics for data read-out and software for both DCS and analysis. However, the coverage and resolution are different. While PHOS is located underneath the TPC and *Transition Radiation Detector* (TRD), EMCAL is located above the TPC; the phase-space coverage is 23 %. Since it relies on plastic scintillators, the energy resolution is lower, and the measurable p_T is limited to 100 GeV/c. Also the spatial resolution is lower. Like PHOS, it has the ability to generate level *0* and *1* triggers.

2.2.4 Di-jet calorimeter

Di-jet CALorimeter (DCAL) is a proposed new sub-detector, similar to EMCAL, but underneath the TPC, where it will fill the space not already covered by PHOS.

2.2.5 Inner tracking system

Inner Tracking System (ITS) [51] is a silicon detector system for reconstructing the primary collision vertex, as well as secondary vertices of decaying hadrons containing heavy quarks. It has six layers, each of which is organised as a "shell" around the beam pipe, the innermost has a radius of only 3.9 cm, the outermost 43 cm. Neighbouring pairs of layers are organised as separate sub-detectors, with individual characteristics. Starting from the beam-pipe, the *Silicon Pixel Detector* (SPD), *Silicon Drift Detector* (SDD), and *Silicon Strip Detector* (SSD). The SPD has a resolution along the z-axis of 70 μm and 12 μm in the $r\phi$-plane.

2.2.6 Transition radiation detector

TRD [52] is "surrounding" the TPC, and consists of six layers of plastics with varying dielectric constants, altered with wire-chambers. Charged particles passing a boundary between two media of different dielectric constants will emit transition radiation photons, which are picked by the inter-leaved wire-chambers. The probability of generating transition radiation at a given medium boundary scales linearly with the Lorentz γ-factor of the particle. Since this probability is relatively low, a large number of layers are required. The inner radius of the TRD is 2.9 m, the outer 3.7 m.

The TRD is well-suited for distinguishing electrons from pions above 1 GeV/c. Like the TPC, it is sectioned into 18 sectors. It can trigger on high-momentum electrons and contribute to tracking. The resolution along the z-axis is 2.3 cm, and 400 μm in the $r\phi$-plane.

2.2.7 Time-of-flight

Time-Of-Flight (TOF) [53] is placed in the next "shell" outside the TRD, located at an radius of 3.7 m to 3.99 m, divided into 18 sectors matching those of the TPC and TRD. It is used to measure the time it takes a particle to traverse the ITS, TPC and TRD from the interaction point. The particle mass can be calculated from the time of flight, using track length from inner detectors. It also provides triggers for other detectors.

2.2.8 High momentum particle identification detector

The purpose of the *High Momentum Particle Identification Detector* (HMPID) [54] is to extend ALICE's separation capabilities for particles of very high momentum. For π/K, the separation is increased to 3 GeV/c, for K/p, 5 GeV/c. It is a *Ring-Imaging CHerenkov* (RICH) detector, based on detecting Cherenkov radiation, which is emitted when a particle traverses a medium faster than the speed of light (for the medium). The angle of the Cherenkov-shockwave relative to the particle track will depend on the speed of the particle. Combined with momentum measurements from other detectors, this can be used to determine the particle mass.

2.2.9 Muon spectrometer

The *MUON spectrometer* (MUON) [55, 56] is located only on the C-side of ALICE, and measures quarkonia decaying in the di-muon channel, such as J/ψ, Υ, and their excited states. Good mass resolution is required to separate these states. For Υ states, a resolution of 100 MeV/c^2 is needed. Five cathode strip tracking stations are interleaved with absorbers and bending magnets to successively measure momentum, deflection angles and time-of-flight to allow identification.

2.2.10 Zero degree calorimeter

The *Zero Degree Calorimeter* (ZDC) [57] measures the number of spectator nucleons for heavy-ion collisions, thus providing an estimate for the centrality. It is located on both sides of ALICE, some 116 m away from the interaction point. There are separate calorimeters for neutrons and protons. A level *1* trigger can be generated.

2.2.11 Forward multiplicity detector

The *Forward Multiplicity Detector* (FMD) [58] is made from five layers of silicon strips, and is used for measuring the multiplicity at small angles relative to the z-axis. A large fraction of the phase-space is covered: $-3.4 < \eta < -1.7, 1.7 < \eta < 5.0$, full azimuth.

2.2.12 Photon multiplicity detector

Photon Multiplicity Detector (PMD) [59, 60] is a forward detector for measuring the multiplicity distribution of photons, covering the phase-space $2.3 < \eta < 3.7$, full azimuth. The photons are from decayed π^0 and η. Two gas proportional chambers are used as detectors.

2.2.13 Time-zero

Time-Zero (T0) [58] measures the time of the collision, "time-zero", with high precision, and serves as a reference for TOF. The resolution is 25 *ps*, and it is used to discriminate potential primary vertices inside and outside the nominal collision region. A primary vertex outside the collision region is taken as a beam–gas interaction, and is discarded, whilst one on the inside is considered a beam–beam collision, initiating a level *0* trigger for the other detectors.

2.2.14 Veto

Veto (V0) [58] has a larger acceptances than T0, making it more suited as a trigger for p+p collisions. Also, charged particle densities in the range $-3.6 < \eta < 1.6, 2.8 < \eta < 5.1$ can be measured. Like T0, primary vertices, inside the nominal reaction region can be identified.

2.3 Trigger system

ALICE is a triggered [61] experiment. That means the crossing particle bunches will be focused to collide at a pre-determined time. Whenever this happens, there will be a trigger signalling the sub-detectors to collect data. The triggers are organised in an hierarchy: level *0*, *1*, *2*, and *3*. A level *0* trigger is issued for (almost) every collision. Level *3* trigger is generated by the HLT on events that contain interesting physics. How a given detector reacts to a trigger of a certain level, is defined by the detector itself. The triggers are distributed to all relevant parts of the detectors as messages via the dedicated fibre-optical links of the *Timing, Trigger and Control* (TTC) system.

The sub-detectors may choose to ignore a trigger if needed. For example, read-out of data from the previous collision for a certain sub-detector may not have finished when there is a trigger for a new collision; the sub-detector is *busy* when this happens.

2.3.1 TTCrx of DCS board

The DCS board is equipped with hardware for handling trigger messages. Specifically, it has a receiver for the fibre-optically transmitted trigger messages and a custom designed *Application-Specific Integrated Circuit* (ASIC) for processing them, the so-called *TTC Receiver* (TTCrx) chip. The RCU will obtain the trigger messages from the DCS board. Configuration of the TTCrx chip is done via an *Inter-Integrated Circuit* (I^2C) bus internally on the DCS board. A command-line tool available on the DCS board, can be used to manipulate the configuration registers. If verification of the ALTRO registers from the FeeServer during the gaps of the read-out orbit is to be implemented, support for the FeeServer to read the trigger messages from the TTCrx chip has to be implemented.

2.3.2 Busy-box

The only chance to capture data from a collision is at the very moment the produced particles cross the detector. While it is possible to read out data at the FEC side at a high event rate, one can not forward data to DAQ and HLT at the same pace; only "interesting" events will be stored. However, at the time the FECs are gathering the event data, it is not yet possible to tell if the event is interesting or not. This is only the case if one or more of the trigger detectors triggered on it. To solve this, collision data from up to four events are stored in an internal memory of the ALTROs, the *event buffers*. Only after some trigger detector has marked the event as interesting (level *2 accept*), the RCU will be told to retrieve the data from the FECs, and forward them over the *Detector Data Link* (DDL) optical fibre to DAQ and HLT. Remaining events are discarded as "uninteresting".

An additional complication arises as each sub-detector has several RCUs that all provide fragments of the complete event; it is necessary to know that all such fragments have been received by DAQ and HLT before read-out of the next event begins. TPC, PHOS, EMCAL and FMD rely on the Busy-Box for keeping track of the read-out status of all DDL links. Specifically, the BusyBox is maintaining an event counter for each individual RCU. Whenever an event is read into the buffer, the counter is incremented, and opposite, when an event is read out or discarded, the counter is decremented. When there are no more free event buffers, the detector is not ready for collecting data of a new event; the detector is "busy".

At the DAQ side, the DDL is terminated in a *Destination Interface Unit* (DIU) connected to a *Destination Read-Out Receiver Card* (DRORC). A DRORC is a *Peripheral*

Component Interconnect (PCI)-Express extension card that allows the data from the detector DDL to be transferred to servers. All DRORCs have a separate, dedicated link (over a standard *category 5* "Ethernet cable", but with entirely different signalling) to the Busy-Box, *i.e.*, for the TPC the Busy-Box has as many as 216 links, while for FMD only three, as the two extreme cases.

The Busy-Box uses these links to enquire the DRORCs of the status of the read-out. As long as not all DRORCs have acknowledged that "their" RCU has at least one free event buffer, the BusyBox will signal state *busy* to the *Central Trigger Processor* (CTP).

Once all have a free buffer, the Busy-Box will no longer be in state *busy*, and the CTP may choose to trigger for a new event. The new event will increment the event counter of all RCUs, once again causing the Busy-Box to go busy if no more free buffers are available for at least one RCU.

For control and monitoring of the Busy-Box, a separate version of the FeeServer has been developed. It is based on the same framework as the TPC FeeServer; the command set and monitoring values of course are adapted to the needs and requirements of the Busy-Box.

2.3.3 Central trigger processor

ALICE relies on a set-up called the CTP to generate triggers to the FEE. It can be seen as a hub that accepts triggers from trigger detectors, processes them, and generates triggers for the FEE. The basis of all triggering is the 40 MHz bunch-crossing clock. However, this is a mere theoretical limit. The actual collision rate will be far below this for ALICE. The different sub-detectors can be triggered at different rate. Fast detectors, like PHOS, might operate at a few kHz; slow ones, like TPC, will stay below one kHz.

2.3.4 PHOS trigger

PHOS has properties that make it very well suited as a triggering detector. Specifically, it can be used to trigger on photon energy. However, specialised hardware support is needed for this. Standard event analysis of PHOS data would require data to be read out and analysed in software on computers. To be used for triggering, this process is too slow. Rather, a simplified scheme has been developed. From every ordinary FEC there is a direct link to a special FEC, the TRU. There is one TRU per RCU branch. The direct link is used to transfer information of the energy captured by the APD from the $PbWO_4$ crystals, although not at full resolution. This information is collected by the TRUs, which in turn forward them to the *Trigger-OR* (TOR). While the TRU can only see the energy collected by one branch, the TOR can see the "full" picture from all five modules. The TOR can issue triggers to the CTP based on programmable criteria. The

information gathered by the TRUs are also read out by the RCU as part of the event data.

Also for the TOR a specialised FeeServer has been developed, based on the standard FeeServer framework shared with all other FeeServers. Control and monitoring of the TRUs are handled as an integral part of the PHOS FeeServer. EMCAL is also using the a similar TOR and TRUs for generating triggers on energetic events.

2.4 High-level trigger

The over-all purpose of the HLT [61] is to reduce the overwhelming amount of data produced by ALICE to a manageable level. A data rate of approximately 1.5 GB/s can be written to disk. The detectors are capable of producing data at a rate one to two orders of magnitude higher. The challenge is to determine *which* events to keep. The approach of HLT is to reconstruct *all* events in real time, using less accurate, however much faster, reconstruction algorithms. Although dedicated trigger detectors exist to generate triggers from for example very energetic events, HLT allows implementing much more sophisticated triggers based on physics, such as jets or D^0s, found in reconstructed data. The reconstruction is sufficiently accurate to analyse the events for interesting physics that can justify permanent storage. In that case, the reconstruction and a trigger decision output will be forwarded to DAQ.

Some analyses require very high statistics, and should be run on largest possible data sets. HLT allows such analyses to be included in the real-time processing of the events, so that they can utilise the full statistics of events not being stored as well.

The sub-detectors need to be *calibrated* before the data collected can be used for accurate reconstruction off-line. For example, the drift velocity of the TPC has to be calculated to obtain an accurate position measurement so that the tracks will match those of ITS and TRD. This typically requires a calibration-*object* to be calculated from data, either normal collisions or under special conditions, henceforth written to the *Condition DataBase* (CDB). Such objects can be gradually refined as data is being processed in several off-line *passes*. The first pass to generate the initial calibration objects is called pass *0*. However, these objects can be calculated by HLT, eliminating the need for pass *0*.

2.5 Data acquisition

DAQ [61] is responsible for collecting and storing data forwarded from the various detectors. The data is received from detectors via one or more DDL, terminated at a *Local Data Concentrator* (LDC), which again will forward the data to a *Global Data*

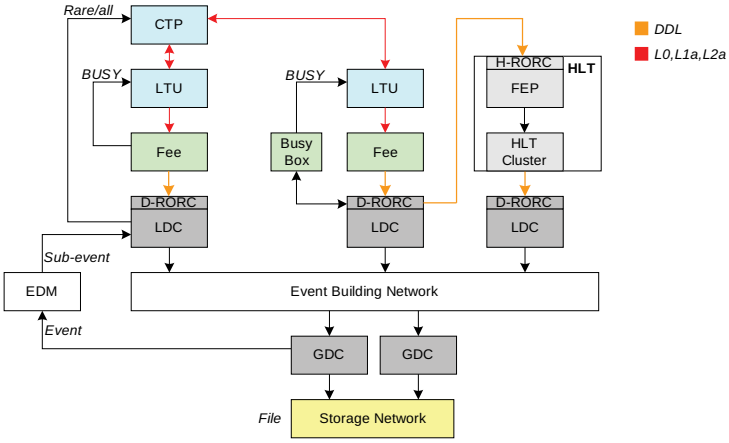

Figure 2.8: Overview of DAQ, including HLT and CTP Also shown is the integration of the BusyBox. [62]

Concentrator (GDC), where complete events are built from data from multiple detectors. Detectors are grouped into *partitions*. The partitions are handled independently. A detector may only participate in one partition at any given time. However, a detector outside the partition may be used as trigger detector.

HLT and DAQ are receiving the data in "parallel", *i.e.* they both receive identical copies. DAQ has different modes of operation concerning permanent storage of events: store events *selected* by HLT, *flag* events or store *all* events. For production runs, HLT is expected to select the events to store. However, at low collision rates, it may be desirable to keep all events, irrespective of HLT output.

Figure 2.8 shows the overall overview of DAQ, HLT, CTP and some other components.

Chapter 3

Front-end electronics components

3.1 DCS hierarchy overview

The FEE is the electronics which is integrated on the detector itself, *i.e.*, that is not located in the *Counting Room* (CR) or other areas surrounding the detectors. It can be considered as an integral part of the detector.

Being part of the detector, there are requirements that apply to the FEE more than to other electronics: sustain higher levels of radiation, fail-safe (physical access blocked for up to a year), compactness (limited space inside detector), dispatch little heat (cooling complicated inside detector, also temperature gradients undesirable), and little noise (weak analogue signals from the detector are not yet digitalised). And of course, this has to be done without compromising performance. These constraints have to be taken into account when designing the FEE.

The detector and the FEE is controlled and monitored by a DCS [61]. As shown in Figure 3.1, a distributed three-layer hierarchical DCS has been devised to control and monitor the TPC: *field layer* is defined as the FEE itself; *control layer* is the software, FeeServer, running on the FEE of each RP, as well as the lower part of the ICL; and *supervisory layer* is the upper part of the ICL and the *ProzessVisualisierungs- und Steuerungs-System* (PVSS)-based [63] *Graphical User Interface* (GUI) the shifter is operating. Configuration of the FEE is accomplished by sending binary configuration data blocks to the FeeServer, which will interpret and execute them accordingly. Values of registers of special importance, such as FEC temperatures, voltages and currents are being monitored and published. The state from the integrated state-machine, indicating the overall system state, is also published. Upon receiving a high-level configuration command from the GUI, ICL assembles configuration blocks for the FeeServer by retrieving all relevant configuration parameters from the database. ICL also collects all data points published by FeeServer, and forward them to the GUI. There is a full integration with the *Experiment Control System* (ECS), enabling operation of the TPC by

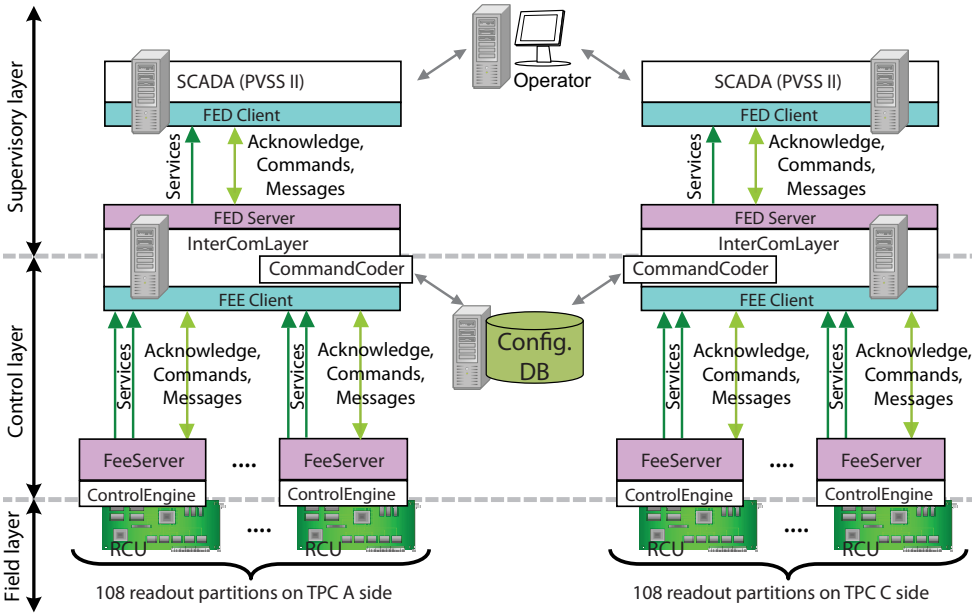

Figure 3.1: TPC DCS working principle. PHOS is similar, but with fewer partitions. [43]

the ALICE shifter.

For optimisation of performance, the two detector sides are handled by separate systems. Throughout the following chapters, the individual sub-systems shown in the figure will be described in detail. Although the TPC and other sub-detectors sharing the same electronics will have the main focus, the general concepts also apply to other sub-detectors.

3.2 Front-end electronics for TPC and PHOS

For the TPC, a scheme for the FEE based on a DCS board connected to a RCU with a bus connecting several FECs was designed [64, 65]. The same scheme, with only a few changes, was also chosen for PHOS and later EMCAL. The DCS board is shared with TRD, though TRD does not have the concept of separating the RCU and FECs.

The DCS board can be seen as a computer controlling and monitoring the overall system. It does not directly participate in the data read-out. The RCU can be compared to a motherboard. It has one bus which connects to the DCS board, and two buses for reading out the FECs, In addition, it has an interface to the SIU card, which has a 1.25 Gb/s fibre-optical transceiver for forwarding the data from the detector to DAQ and HLT.

A FEC is the entity that performs the actual data collection. Weak, analogue signals

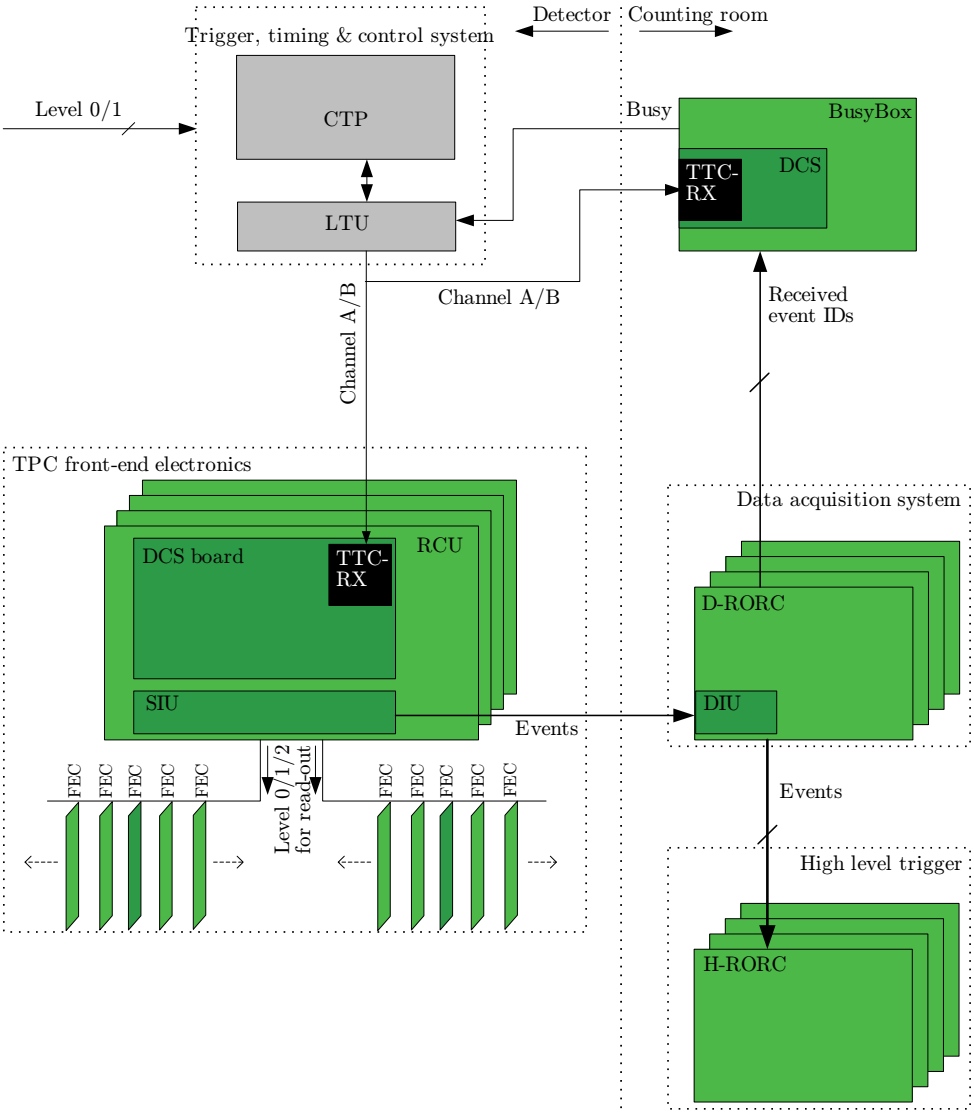

Figure 3.2: TPC trigger and data read-out working principle. Adapted from [62].

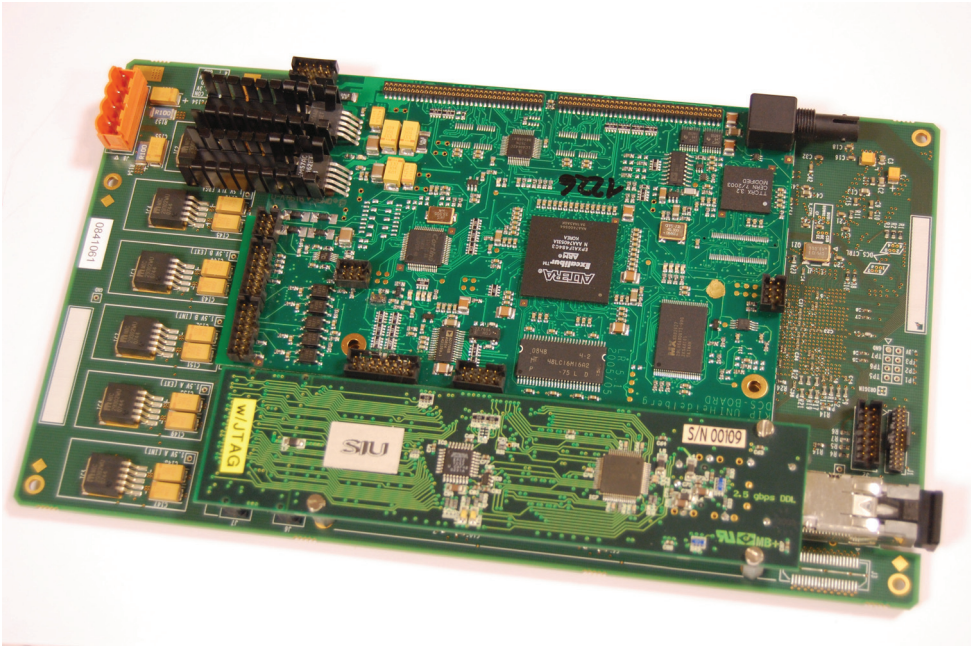

Figure 3.3: Picture of a DCS board (top) and a SIU (bottom) mounted on an RCU (largest, lowermost).

are taken as input. Henceforth, they are amplified, shaped, digitalised before they are finally presented to the bus to the RCU. Figure 3.2 shows a schematic view of the trigger and read-out system for the TPC, while a picture of the setup of an RCU, DCS board and a SIU connected together is shown in Figure 3.3.

3.3 DCS board

The DCS board can be considered the "heart" of the configuration and monitoring of the FEE. Essentially, it is a small computer, with *Central Processing Unit* (CPU), memory, "hard disk" (actually a flash disk) and network. The FEE may be thought of as "peripheral" electronics in this context. Via the bus to the RCU, it has full access to the hardware of the FEE, including registers for configuration and monitoring. The communication with the outside world is achieved through network connectivity. In Figure 3.4, a picture of a DCS board is shown, while Figure 3.5 shows a block diagram of the communication of the DCS board with the RCU and via the network.

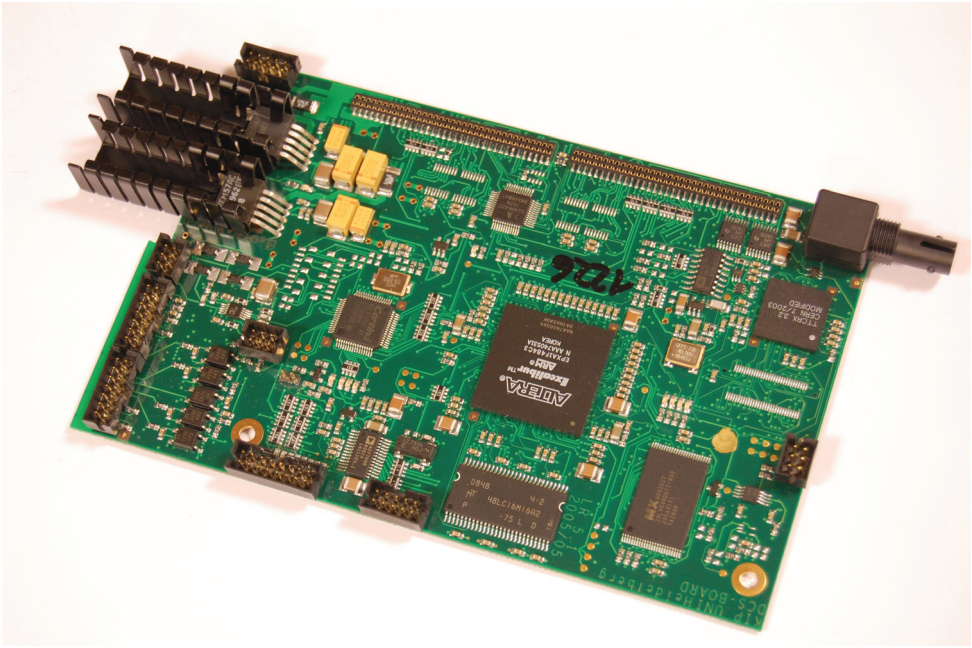

Figure 3.4: Picture of a DCS board. The optical trigger receiver upper right, just below, the TTCrx chip. The large chip in the middle is the main FPGA. On the top, the connectors for the DCS bus can be seen.

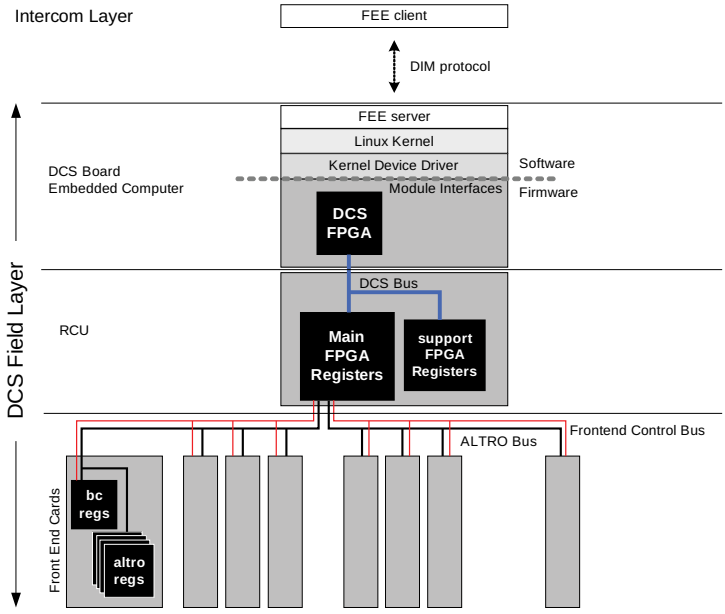

Figure 3.5: Block diagram of DCS board communication. The embedded computer running Linux is shown in the middle. Above, the communication with the ICL via DIM is shown. Below, the communication with the RCU, and further the registers of the FECs are shown. [62]

3.3.1 Hardware components

The hardware design of the DCS board is centred around a FPGA with an ARM CPU core. The FPGA is manufactured by Altera. The CPU core runs at 40 MHz, has a *Memory Management Unit* (MMU) and 32-*bit* wide registers. It is capable of running a minimalistic Linux system. Also, there is 32 MB of *Random Access Memory* (RAM) and an 8 MB flash disk.

The DCS board is also used to receive trigger messages from the CTP. For this purpose, it is equipped with a fibre-optical receiver, as these messages are transferred optically. A custom-made ASIC processes trigger messages. The fibre also provides the system with a 40 MHz clock signal, which is used by the FEE as the system clock, though it can also be switched to use an internal clock for stand-alone applications. This is particularly useful in stand-alone test setups where the trigger is not available. The trigger information and clock is available to the RCU and henceforth the FECs via the RCU bus between the RCU and the DCS board.

The DCS board is equipped with a simple *Recommended Standard* (RS)-232 serial interface. [1] It only provides *send* and *receive* data lines; all flow control is expected to take place in software. The main purpose is to enable direct console access from a regular computer during development and testing. In this case, no network is required to interact with the Linux system console. If the DCS board for some reason (for example update of the network configuration fails) is no longer accessible via network, it provides a last resort for access. During normal operation in the underground experimental area, the DCS boards are impossible, or at least very hard, to access for up to a year. Hence, if network access fails, the serial interface can not easily be connected to a computer for debugging. One option that was considered, but unfortunately not been implemented, is to connect the serial interface of two neighbouring DCS boards. If one of the DCS boards were to become inaccessible, this would allow log-in from the other linked DCS board via the serial console. Although such failure is not expected to happen, the severity of not being able to utilise the partition controlled by said DCS board for the remainder of the period still makes such interconnect intriguing.

3.3.2 Firmware

The firmware, including software, of the DCS board is stored on a flash disk. It is divided into four partitions. The first partition contains the boot environment. This is a simple boot loader that will set up the basic environment before it loads and passes control to the Linux kernel. Most importantly, it will set the *Media Access Control* (MAC) address of the Ethernet interface. The 32 *Most Significant Bit* (MSB) is a fixed value. However,

[1] The parameters of the serial interface are: 57600 baud rate, eight data bits, no parity, one stop bit, and no flow control.

the 16 *Least Significant Bit* (LSB) correspond to the board number (the serial number of the board). Since this number is encoded into the flash, it is important to take care not to program a DCS board with a firmware prepared for a different board. In that case it will be programmed with a wrong board number.

The second partition is the Linux kernel just mentioned above. It will keep control of the board for the remainder of the time while it is still powered on.

The third partition is the root file system mounted and used by the Linux system. It is based on *ROM File System* (ROMFS), a file system specially designed for use on flash disks, exploiting the advantages of flash storage whilst trying to minimise the disadvantages. One feature is *compactness*, *i.e.* there is little overhead. Although the file system is mounted *Read–Write* (RW), the system is configured to only open files *Read-Only* (RO) for normal operation. This is important as the DCS board is usually powered off simply by cutting the power, without any proper shutdown procedure. Hence, there is no chance to flush file caches to disk, and partially written files will be left in an inconsistent state. It should also be noted that there is another compelling reason for not writing to the flash disks as part of normal operation: flash disks generally have a limited number of write cycles before they are "worn out". When this limit has been exceeded, the frequency of read failures will gradually increase, and the system will no longer be reliable. However, it is often necessary, or at least desirable, to make temporary files. To facilitate this, but at the same time prevent writing to flash, a 1-MB RAM disk is used for casual temporary files that are not to be kept, whilst persistent temporary files will be stored on network-provided storage.

The fourth and final partition contains the firmware for the FPGA itself. The design it contains will allow the Linux system to utilise and communicate with the hardware.

The contents of the flash is fully accessible to the Linux system via a standard device file in the */dev* directory. Hence, the DCS board can be used to update itself, typically by accessing a new version of the firmware via network, and write it to the device file. The firmware may be updated remotely using the *remoteupdate4.sh* script, which relies on *Secure CoPy* (SCP) and *Secure SHell* (SSH) to transfer and install the firmware. After a reboot, the DCS board is running the new firmware. Alternatively, it can also be reprogrammed using the *Joint Test Action Group* (JTAG) interface.

3.3.3 Operating system — Linux

As already briefly mentioned, the DCS board is equipped with a CPU that is used to run Linux. The resources, such as memory and computing power available on an embedded computer is rather limited compared to what is found on larger computers. Hence, the configuration is optimised with respect to limited resource utilisation. The kernel is compiled to only include necessary modules. Likewise, only applications needed for

managing the FEE, directly or indirectly, are included. Often, software include a rich feature set that can be useful in very specific scenarios. For an ordinary computer, the size of the executable versus functionality balance is not a consideration. However, for embedded systems it may be desirable to reduce the functionality of the software to a minimum to save disk space, as well as RAM and CPU cycles. A scaled-down version, called *BusyBox*, of the most popular command-line tools for Linux is available. It is not to be confused with the *Busy-Box* used by some detectors to keep track of the state of the data read-out. Whereas each command on a ordinary Linux system corresponds to a distinct executable, all BusyBox commands are in reality one single executable. The distinct commands are defined as aliases to the main BusyBox executable, with special parameters to indicate which command to invoke. Also, the reduced executables are less resource demanding, both in terms of memory and computing power.

For the overall configuration, it is important to limit the number of processes loaded. Firstly, loading the processes takes time, and will delay the boot sequence. Secondly, if they remain loaded, they will consume valuable resources.

All these methods are utilised by the DCS board Linux system. The result is a "full" Linux system on less than 8 MB of flash disk space [66]. However, the quest to save space by reducing functionality can also have unfortunate side effects, particularly if any of the removed functionality at some point is needed. The Linux systems obtain *Internet Protocol* (IP) addresses via *Dynamic Host Configuration Protocol* (DHCP) on boot. For the TPC this means 216 DCS boards will send DHCP requests almost simultaneously. If the DHCP servers are not able to handle this flow fast enough, some of the DCS boards will fail to obtain an IP address, and become non-functional. To mend this problem, it was necessary to increase the number of times the DHCP client would try to obtain a lease before giving up. Unfortunately, this was one of the features that had been removed from the BusyBox DHCP client. Consequently, it was necessary to compile and distribute the full version of the client to the DCS boards.

Matching kernel modules and software have been written to communicate with the custom hardware and firmware of the DCS board. Through these, full access to the hardware is possible, thereby enabling the desired monitoring and control of the FEE. The FeeServer is designed to perform this task; it may be considered a *relay station* utilising the Ethernet as a bridge between the hardware and the outside world.

3.3.4 Tools

The FeeServer is the main software tool for performing the remote FEE configuration and monitoring during production runs. However, for debugging and development at the hardware level, encoding commands to the FeeServer for reading or writing a given register requires an infrastructure that may not be desirable in such cases. Rather, this

is done much more conveniently by using standalone command-line tools.

One of the main tools is the RCU shell, *rcu-sh*. It is a simple command-line tool giving access the FEE by writing to registers of the RCU (or other boards connected to the DCS board via the same bus). The main operations are to read and write to memory. This is done using the *r* and *w* parameters, respectively. The default action is to read or write one word. For example, *rcu-sh r 0x8000* will *r*ead the word located at address *0x8000* and write it to the screen. Like-wise, *rcu-sh w 0x8000 0x1* will *w*rite the value *0x1* to address *0x8000*. It is also possible to read a range of addresses from memory in one operation by indicating the range as the third parameter: *rcu-sh r 0x8000 64* will *r*ead *64* words, starting at address *0x8000*.

If it is desirable for RCU shell to perform a number of sequential operations, they may be listed in a text file, in which each line contains the parameters one would pass to *rcu-sh* on the command line (without *rcu-sh* itself). This file is passed to RCU shell in batch mode, which is toggled by the *b* parameter. Assuming we call the file *batch-commands.txt*, the full command is *rcu-sh b batch-commands.txt*.

There are some more parameters understood by RCU shell. The *wait* statement will make RCU shell halt for the specified time. Perhaps not very useful in interactive mode, it is mainly intended for batch operation, where it for example can be necessary to wait for the electronics to stabilise or return a result. The *driver reset* keyword will try to reset the driver used to access the RCU.

Many standard Unix and Linux tools are used by the DCS board. Most of them are available in the simplified BusyBox version. The two main exceptions are *Micro DHCP Client Daemon* (UDHCPCD) and *SSH Daemon* (SSHD). UDHCPCD is a client for the DHCP. A BusyBox version is available, but did not have functionality needed for setting time-out. SSHD is the server daemon for the SSH protocol. It is used for gaining remote access to the DCS board. Both interactive log-in and file transfer is possible. For SSHD, no BusyBox version is available at all.

3.3.5 File system layout and scripts

A partition of the DCS board firmware flash holds the root file system for the Linux system, enabling standalone operation without network if needed. A small RAM-disk holds a file system for temporary files in the */tmp* directory. The flash disk has a limited number of write cycles before it wears out; thus it is not suited for the frequent creation, deletion and change typically experienced by temporary files. Since the DCS boards are turned off by cutting power, writing to files on the flash is only expected to take place exceptionally.

Network File System (NFS) is used to provide server-side storage to the DCS boards. Normally, two shared directories are mounted: one RO, and one RW. They are mounted

on sub-directories of */mnt*, *dcbro* and *dcbrw* respectively. The first part, *dcb* implies the share is in fact for *DCs Board* (DCB). The second part, *ro* and *rw*, reflects the access nature of the shared resource.

The RO file system is intended for files that there is no need to — and indeed should not — be changed from the DCS board side. Most important among them are start-up scripts and the FeeServer binary. Also, various tools, libraries, scripts, updates, *etc.* which may need to be accessed from the DCS board either as part of normal operation or dedicated debugging, is preferably stored here as they share the requirement of not being modifiable from the DCS board.

The RW file system is used for files that the DCS boards have to be able to modify. Since all DCS boards have permission to write and in principle also over-write or delete any file, it follows that the presence or contents of these files may not be critical to the functioning of the DCS boards. Typically, log files and other temporary files of no "profound" importance are stored here. In addition, output files from updates and debugging of the DCS boards may be kept here. All DCS boards mount same shared directories.

The motivation for storing such temporary files on server-side storage is two-fold. Firstly, as mentioned the flash storage has a limited number of write cycles, hence temporary files have to be written to a RAM-disk. This will cause the files to be lost on every power cycle or reboot. While this clean-up might have certain operational advantages, it also means it will not be possible to inspect "old" logs in case of failures or other problems. Secondly, server-side storage makes it possible to examine the logs without logging into the board itself. Taken into account the large number of boards, checking the logs of all boards would be very time consuming. Using this approach, it is possible to employ shell scripts or other types of software on the server to analyse and extract useful information from all DCS boards in a single operation. In addition, the vast storage available on the server makes it possible to keep logs for longer period of time. In case of problems, it will be possible to go back and check when a certain error started occurring.

The main start-up script of the DCS boards is the */etc/init.d/rcS* file. It is executed just after the kernel has finished loading, and is responsible for setting up a working system. Tasks are such as loading kernel libraries and setting up networking. The last task is to load other start-up scripts located either in */etc/init.d/boot* and */mnt/dcbro/boot*, located on the local and the remote file systems, respectively. This allows easy changes to the DCS board configuration on the server side, with no need to modify the DCS board itself. Apart from easy modifications to the boot process with no need to distribute the updates to the DCS boards, the main advantage is the fail-safeness. Any change to the DCS board itself can in principle make the DCS board to not boot properly and become inaccessible, which leads to a situation where the error can not be corrected. In such

cases, the only method of recovery is to re-program them from scratch via JTAG. This would be extremely unfortunate as it requires physical access to the boards. On the other hand, in case of problems due to errors in the server-stored configuration files, the mistakes can still be corrected on the server side. At most, a reboot would be required on the DCS board side.

3.3.6 Start of FeeServer

The FeeServer is the main software tool of controlling the FEE remotely. The binary is located in, and run from, a directory shared from the server: */mnt/dcbro/bin/feeserver*. Hence, an upgrade can be performed simply by replacing this file with a newer version on the server. The previous sub-section discussed how start-up scripts are invoked on boot. The FeeServer is loaded using such script. The name of the script is *S49StartFeeserver.sh*, and is located on the server-side boot script directory */mnt/dcbro/boot*. In case the FeeServer for some reason unexpectedly fails and exits, the script has provisions for automatic restarts of the FeeServer. Output from the FeeServer are redirected to log files under */mnt/dcbrw/fee-logs/*. The log files are named according to the DIM name of the relevant FeeServer. For the FeeServer located on *side A*, *sector 00* and *partition 0*, the name is *TPC-FEE_0_00_0.txt*. Other FeeServer are named in a likewise manner.

3.3.7 Network

The DCS board is equipped with a modified Ethernet interface. According to the Ethernet standard, an electric transformer should be used between the Ethernet lines and the electronics to decouple the potential of the electronics from the potential of the lines. However, as the DCS board is exposed to the very strong magnetic field inside the L3 magnet, this is not possible. Instead the decoupling is done through a purely electronical circuit. Such modified Ethernet interface is sometimes referred to as *Easynet*. Full duplex is supported. The electronical decoupling used can not achieve the frequency bandwidth needed for 100 Mb/s data rate. As a consequence it is caped at 10 Mb/s even though the rest of the hardware fundamentally can support 100 Mb/s. This is not considered as a limitation, because the Ethernet is only used for monitoring and configuring the FEE, not data read-out. The only exception may be the case of transferring pedestal values for the ALTROs. For assembling binary configuration data blocks, the ICL is used. Currently it configures the RPs sequentially, implying only 10 Mb/s of the server's 1000 Mb/s bandwidth can be effectively be utilised. However, a new version of the ICL, *Java ICL* (JICL) is expected to be able to do such configuration in parallel for all RPs it serves. For the case of the TPC, the ICL has to serve 108 RPs (separate ICLs for each of the two barrel ends). The maximal combined data rate will be 1080 Mb/s, only slightly exceeding the data rate of the server. Hence, it should not represent a

limiting factor.

As network protocols, standard IP is employed on top of the Ethernet connection. Consequently, standard Linux and Unix network tools can be used. In particular, this applies to NFS for file sharing, SSH for interactive remote log-in, SCP for file transfer, DHCP for assignment of IP addresses, *etc.* based on hardware MAC addresses, *Domain Name System* (DNS) for host name look-up and *Network Time Protocol* (NTP) for synchronising local time to global time. In addition, the custom software, the FeeServer, is designed to rely on network for communication with the outside world.

Network is loaded by the */etc/init.d/rcS* start-up file. There are two main scenarios for obtaining network parameters such as IP address, net mask, broadcast address, gateway and time server. The primary strategy is to obtain them from DHCP. Since this is the method used in the experimental setup, the timeouts are rather generous. The reply-timeout is 3 s. If no reply has come within this time, a new request is sent. This procedure is repeated ten times before giving up; hence total timeout is 30 s.

If DHCP fails, it falls back to the configuration given in */etc/init.d/network.txt*. The IP address stored in this file is *10.0.x.y*, where the 16 MSB in $x.y$ is the board number, and, henceforth, the 16 LSB is the MAC address. The net-mask is *255.255.0.0*, and broadcast address is *10.0.255.255*. Neither gateway nor time server is configured. It is possible to access the board configured with these parameters by configuring a computer in the same physical network with compatible parameters. [2]

3.4 DCS bus

The *DCS bus* is the main extension bus of the DCS board. In most cases it is used to connect to an RCU, however other devices have been designed for use with the DCS board as well. From the hardware design, the bus is very versatile. The exact usage of the pins is determined by the firmware of the DCS board and the other device sharing the bus.

For communication with the RCU, the bus can at any given time be in one of three *modes* of operation: *message buffer*, *flash*, or *select map*. The mode refers to the target device of the communication, and the same physical bus lines are used in all modes. The mode of the bus is set on the DCS side. In practice, it is done via an *Input–Output ControL* (IOCTL) system call to the Linux device driver for the bus. Care must be

[2] However, in practice this is difficult to do in the ALICE experimental setup. The link to the switch of the DCS boards goes through routers where only addresses in the IP-ranges assigned to the networks are passed through. Hence, the computer must be connected directly to the switch of the DCS boards. For each of the two stacks of switches serving the two TPC sides, there is a spare Ethernet cable (the other of the pair of cables going to the outlet used by each stack) to the network starpoint. However, gaining physical access to this will involve the *Information Technology* (IT)-department. Most likely, this option is reserved for last resort attempts when the experimental area is inaccessible and one suspects DHCP might have failed for a DCS board.

taken to assure that only one application tries to set the mode of the bus at any given time. It is possible for a process to change the mode of the bus whilst another process is accessing it. The result is almost destined to be some sort of malfunctioning, including possibilities of corrupting the firmware.

Although this bus arrangement is valid for RCU, it may not be the case for some of the other systems that relies on the DCS board for control and monitoring, such as the trigger BusyBox. The BusyBox has neither flash nor Actel, and the programming of the FPGAs is expected to always be done from software. The BusyBox may have one or two FPGAs. For the two-FPGA version, it has two select-map devices. Since there is no Actel, the select map is accessible directly from the DCS board. As the overall number of lines of the DCS bus is limited, the message buffer bus width is limited to 16 bits, in contrast to the 32 available for the RCU. The select map bus is always eight bits wide.

3.4.1 Message buffer-operation

Communication between the DCS board and the main FPGA takes place via the *message buffer*. This is the only memory that can be used for bi-directional communication; the flash and select map memories are only for programming firmware of FPGAs. The memory is laid out by the firmware of the FPGA; it defines a communication interface. Instructions or configuration of the RCU by the DCS board is done by writing to specific, predefined registers. Likewise, the FPGA firmware will write results and messages to registers for the DCS board to read. Since the memory is located on the RCU, the DCS board is the "active" part of the communication in the sense that the responsibility for transferring data over the bus lies on its side. The RCU has no means of "pushing" data to the DCS board, or writing to it. There is an interrupt line to the DCS board that can be utilised to notify the DCS board of events requiring attention on the RCU.

Parts of the FPGA memory is used to expose values of particular interest. This is typically counters and other values that can be used to assert the operation of the firmware and FEE. These registers are set by the RCU, and may only be read by the DCS board.

Other parts of the memory is used for configuring the behaviour of the firmware or the FEE; they may be considered as "input-parameters" to the operation. For example, the FECs can be turned on and off by writing to a register.

3.4.2 Flash-operation

The RCU is equipped with a flash device to store the firmware of the main FPGA of the RCU. FPGAs are either based on flash or RAM. The firmware of flash-based are persistent, in contrast to RAM-based, which has to be reprogrammed after a power-cycle. RAM-based are also more prone to single-even upsets from radiation. However,

at the time on design, flash-based FPGAs of the size needed were not available, hence one based on RAM had to be used despite these issues. The external flash is used to program the main FPGA on power-on. A smaller flash-based FPGA, manufactured by Actel, is used to read the configuration from the flash and program the main FPGA. This procedure is optional, and may be disabled if desired.

To recover radiation induced errors in the design loaded into the RAM of the main FPGA, the Actel can be programmed to *scrub* the main FPGA. This is a technique supported by the FPGA, where blocks of the firmware can be read from and written to the FPGA (shown in Figure 3.6) whilst it is operating. Hence, the Actel can read the same frame of firmware design from both the FPGA and the flash and compare them. If they differ, it will update the FPGA with the frame from the flash, without disturbing data-taking. This process will loop continuously over all frames [67].

3.4.3 Select map-operation

The *select map* operation mode of the DCS bus allows for direct access to the FPGA firmware memory from the DCS board. In particular, this makes it possible to program the firmware to the FPGA directly from the DCS board in software. If automatic programming of the FPGA by the Actel is not desired, it can be done by the Linux boot scripts, or whenever needed at a later stage instead.

3.5 RCU

In the introduction, the RCU was described as the motherboard where everything is connected together. As the name implies, it is in control of the data read-out. The firmware of the embedded FPGA will read out data from the FECs, and forward it to the SIU module and henceforth to the DDL. Read-out can be performed autonomously without interaction with the DCS board, except for the trigger messages received through the DCS board and the initial configuration. The RCU is designed to allow the DCS board to carry out monitoring without disturbing data read-out. In firmware, this is facilitated by a separate module of the RCU firmware, the *Safety and Monitoring module* (SM). In hardware, there are two buses to the FECs; one for data read-out, and one for monitoring. Hence, also monitoring of the FECs is possible without interrupting data read-out.

An FPGA made by Xilinx is used as the core of the RCU. Its main task is to read out data from the FECs, decode it, then re-encode it by a data format used to transfer data to DAQ and HLT via the DDL optical fibres. The firmware can either be loaded from a flash disk with the aid of an auxiliary FPGA, made by Actel, or by software from the DCS board. Implementing functionality in the FeeServer to allow remote, central update

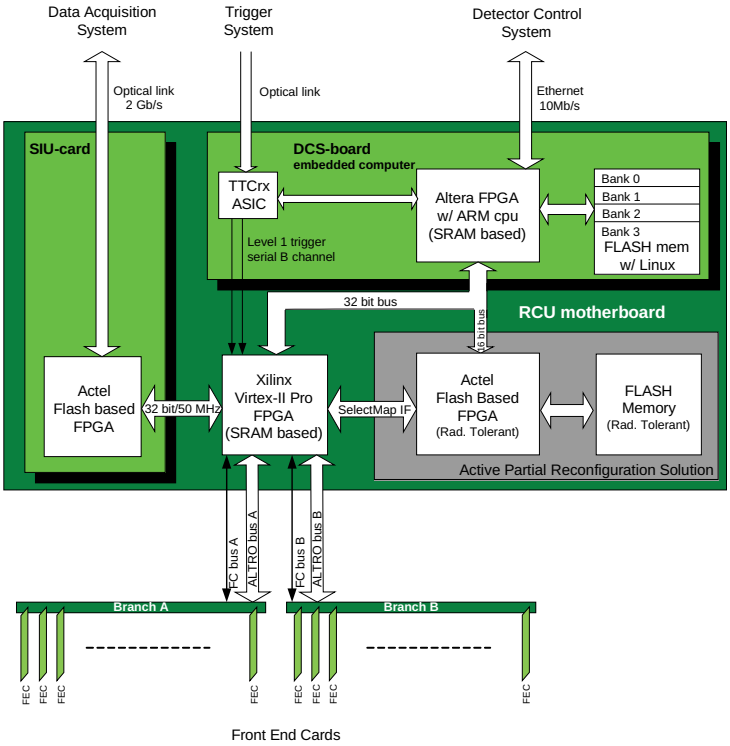

Figure 3.6: Dataflow of the RCU. Connections to the DAQ, CTP, DCS board and FECs are shown. [62]

of the firmware is being considered. If enabled, the auxiliary FPGA can automatically load the firmware to the Xilinx FPGA on power-on. Alternatively, it can be also be loaded automatically by the DCS software. For debugging and development, it might be preferable to load it manually through software when needed. The firmware for the Xilinx FPGA controls the behaviour of the RCU, and is based on a modular design [68, 69]. Figure 3.6 shows the dataflow for the RCU and to related sub-systems.

GTL and I^{2}C buses

The DCS board can access the FECs via the RCU. There are two "parallel" access buses: the *Gunning Transceiver Logic* (GTL) bus and the I^{2}C bus. The names reflect the bus standards they are based on, they are both very common in commercial applications. Both the FEC BC and the ALTRO registers are accessible via the GTL bus, through the I^{2}C bus only the BCs can be reached. For performance, the buses are split into two independent branches; that is, functionally two buses. Each branch has a theoretical limit of 16 FECs, *i.e.* 32 for the two branches together.

The GTL is a fast bus running at 40 MHz to transfer data collected by the ALTROs

to the RCU, where it can be forwarded via the optical fibre. It is commonly referred to as the ALTRO bus since it transfers data collected by the ALTROs.

However, during data-taking, the GTL bus is fully occupied by reading out data; any attempt by the DCS board to use it for accessing FEC registers, such as monitoring, will ruin this process. The slow control bus was designed specifically for this purpose — allowing access to the BCs for monitoring and control without interfering with the data read-out. For this purpose, a high transfer rate is not required. Its slow transfer rate and control purpose has earned it the common name *Slow-Control* (SC) bus.

Since the ALTROs are only reachable via the GTL bus, their initial configuration has to be done via this bus. This is, however, not in conflict with data taking, as this can only begin once the ALTROs and other parts of the FEE are completely configured. In some cases, there might be some performance gain in configuring the BCs via the GTL bus as well, since it operates at higher speed. One such example can be the PHOS APD voltages.

One use case, however, where access to the ALTRO via the I^2C bus would be very useful, is verification of the ALTRO registers during runs. Radiation may change the contents of the registers at a very low rate. Implementing verification by utilising the GTL for read-back and reconfiguration during gaps of the read-out orbit, *i.e.* when the bus is not used for read-out between two events, is being considered depending on the experience gained during first $Pb + Pb$ data. Reading and writing the ALTRO registers through the I^2C bus would be trivial compared to intercepting the read-out orbit without disturbing data taking.

Both the GTL bus and the I^2C bus are accessible to the DCS board through message buffer memories. For the GTL bus, sequence of specially encoded instructions are written to the *Instruction Memory* (IM). When the DCS board has given the RCU the *execute* signal by writing to another, specific register, the RCU starts executing the content of the IM. The instructions are commands to read or write given registers on the FECs, both ALTRO and BC. There are also instructions for flow-control, though they are in practice not used. Results from the instructions are written to the *Result Memory* (RM), where the DCS board can collect them. For read commands, the value read is encoded and stored here. If errors occurred, they can be accessed from dedicated registers.

A similar procedure exists for the I^2C bus, In contrast to the GTL bus, only one read or write command can be issued at a time. One message buffer register is used for addressing the BC register, for writing another holds the data to be written as well. After giving the *execute* signal (different register than for ALTRO IM), the FEC BC is accessed. In case of read operation, the result is available in a register. Possible errors are flagged in the error register.

For the TPC, a variable number of FECs are used, depending on the radial location of the RP in the sector. The innermost RCUs have 25 (13 and 12 for the two branches),

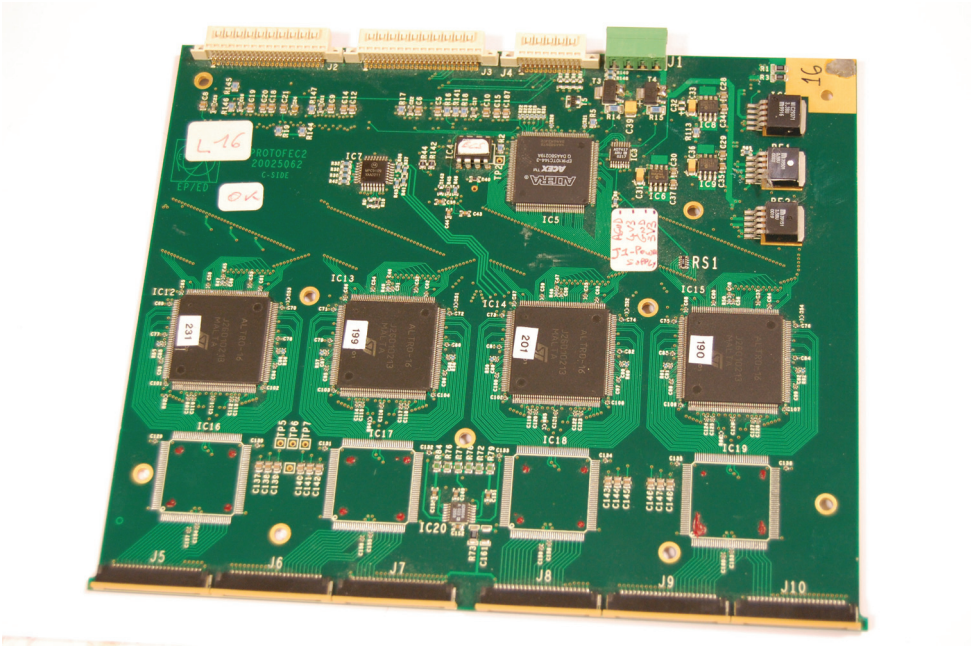

Figure 3.7: Picture of a TPC FEC. The widest connector on the top is for the GTL bus, the narrower for the I^2C bus. On the bottom are connectors for cables to the read-out pads. The Altera chip on the top is the FPGA containing the BC. The four large chips in the middle is the ALTROs, with the corresponding, slightly smaller PASA chips just below. Opposite side of the FEC looks similar.

the outermost have 18 (nine for each branch). Higher particle track densities close to the interaction point demands higher spacial resolution for the innermost partitions; hence the "counter-intuitive" arrangement of more read-out channels for less area covered. PHOS has 14 FECs for data read-out and one special FEC for trigger decision for each branch, giving a total of 30 FECs.

3.6 TPC and PHOS FECs

Both TPC and PHOS have a similar arrangement of FECs for collecting data. Input to the FECs is a weak analogue signal. For TPC, this signal comes from a read-out pad, onto which charge from the particle track is induced. For PHOS, it comes from an APD that measures photons from a scintillating crystal. Though there are some implementation differences, the signal from both detectors goes through a shaping amplifier before it is digitised, then digitally filtered, in the ALTRO chip. Here it is temporarily stored in the event buffers for read-out by the RCU through the GTL bus, if a trigger is given.

A TPC FEC is shown in 3.7, while Figure 3.8 shows a PHOS FEC.

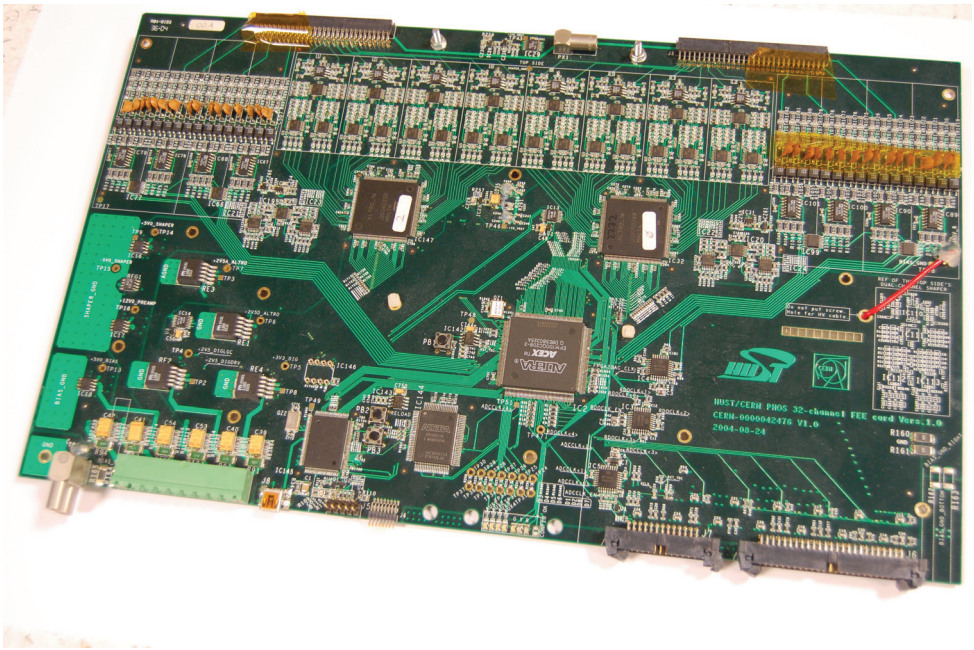

Figure 3.8: Picture of a PHOS FEC. The widest connector on the bottom is for the GTL bus, the narrower for the I^2C bus. On the top are connectors for cables to the read-out pads. The two uppermost chips are the ALTROs. Opposite side of the FEC looks similar.

3.6.1 Board controller

The BC is responsible for controlling the parts of the FECs which is not directly related to data read-out. In particular, this means *monitoring*. The FEC has sensors for environmental parameters like temperature, voltages and currents. If the values of these parameters are not within a certain range, the functioning and lifetime of the FEC may be severely hurt. The BC has registers where the values are accessible for read-out via the I^2C bus. Also, if the values are beyond a configurable threshold, the BC may signal an interrupt to the RCU, requesting action to be taken. The BC has configuration registers for setting the threshold, turning continuous monitoring on or off, *etc.*

The BC is very tied to the hardware design of FEC. Hence, the design is different for TPC, PHOS, FMD and TRU FECs.

3.6.2 PASA

The PASA chip is designed specifically for the needs of the ALICE TPC. As the name implies, it amplifies and shapes the analogue signal from the read-out pads. Afterwards, the signal is forwarded to the ALTRO chip for further processing. It features a gain

of almost 13 mV/fC. Each chip has 16 parallel, completely independent channels for processing read-out signals. The ability to process multiple channels per chip is crucial to achieve the high number of read-out channels of the TPC.

3.6.3 ALTRO

After passing through a shaping amplifier, the pad signal is processed by the ALTRO, which will digitise and digitally filter it. The ALTRO is using a 10-*bit* ADC capable of 10 million samples per second. The digital filtering is performed in four stages. The first stage removes systematic effects and low frequency perturbations as part of a base-line correction for tail-cancellation, which is the next stage. Tail cancellation removes the tail of the pulses within 1 μs of the peak. Fully programmable filter coefficients allow for removal of a wide range of tail shapes. Next, non-systematic perturbations of the base-line superimposed on the signal are removed by applying a base-line correction moving average filter. The full chain is performed completely for each channel independently.

After filtering, the signal base-line is constant within one ADC count. This will allow for very efficient zero-suppression, greatly reducing the data size, while preserving interesting signals.

Configuration registers accessible via the GTL bus determines the behaviour of the ALTRO.

3.6.4 Interrupt

Each branch of FECs has a shared interrupt line for all FECs. If an error happens on one of the FECs, the FEC BC will signal this by pulling the interrupt line low. The FEC BC will continue pulling it down until the RCU has cleared the error on the FEC concerned. There are two kinds of errors, soft and hard. Soft errors are defined as not potentially physically harmful to the FEC. Hard errors may physically damage the board. Too high temperatures, voltages or currents are all examples of hard errors. In case of hard error, the RCU will immediately shut down the concerned FEC as a pre-emptive precaution. This will not cause the run to stop, but data from the specific FEC will of course be lost. When the board has been switched off, the interrupt line will be cleared automatically, as it is not possible for the FEC to continue asserting the interrupt line when it is off. For soft errors, the error register has to be cleared manually. This will not be performed by the RCU autonomously, rather it has to be done in software from the DCS board. There is an interrupt line from the RCU to the DCS board. When the RCU senses an interrupt from the FECs, it will poll the error register of the FEC and write this to its own error registers. The DCS board will access this register to determine which FEC caused the interrupt, and handle it accordingly. If the interrupt register on the FEC is not cleared by software, the FEC will not release the interrupt line.

During commissioning, it was noticed that a design error in the FEC BC firmware occasionally led the BCs to read out erroneous values from the monitoring ADCs. For example, temperatures of $127°C$ were reported. While obviously wrong, it would cause the RCU to turn off FECs reporting such temperatures, hurting data taking. For this reason, all errors are currently set as soft errors. The values of the FEC BC registers that can cause the BCs to trigger an interrupt are monitored by the FeeServer, and forwarded to PVSS. The frequency of values that are over the threshold that will actually cause an error interrupt, is monitored. Hard errors might be enabled once these results, under real running environment, are determined to be at an acceptable level. Until then, error conditions are monitored in PVSS, where action will be taken manually by shifters.

Chapter 4

FeeServer software

Particles traversing the sub-detectors generate analogue signals in the dedicated sensors. The number of sensors is very large, for example, the TPC alone has more than half a million channels. Also, transferring the signals over long distances will lower the signal-to-noise ratio; hurting detector performance. Both these issues dictate that the electronics for digitalising and processing the signals has to be embedded as an integral part of the detectors. This is the FEE. One of the main challenges of such embedded system is the *inaccessibility* — access to the electronics may require partly dissembling the detector, rendering physical intervention impossible for up to a year during a run period. As a consequence, the electronics must be highly controllable and configurable remotely. The inaccessibility also implies broken electronics may not be replaced until the end of the run period. Hence, monitoring of parameters, such as voltages, currents and temperatures, that may indicate a pending fault, has to be in place. Combined with the complexity of the electronics, this puts demanding requirements to the system for configuring and monitoring the electronics.

A setup based on a small Linux computer system embedded into the FEE was chosen. On top of this, control and monitoring are handled by a custom software, the FeeServer. Although the same requirements may also have been fulfilled by a system implemented entirely in hardware, it would be difficult to make it as flexible as the software-supported solution. For example, the embedded computer allows DCS and FEE experts to log into a Linux system where direct access to all hardware is possible via device files. Command-line tools may be used for development and debugging. Also the firmware of the RCU card and the DCS board is accessible as device files, allowing remote update. Another advantage of the software approach is the ease of implementing complex control and monitoring logic. New versions of the FeeServer is deployable by distributing a single file.

The FeeServer is relying on the DIM protocol for communication over network. DIM provides essentially two functionalities: messages and services. The FeeServer is using an derived version of DIM, called the FEE protocol. Here, the message is used for relying

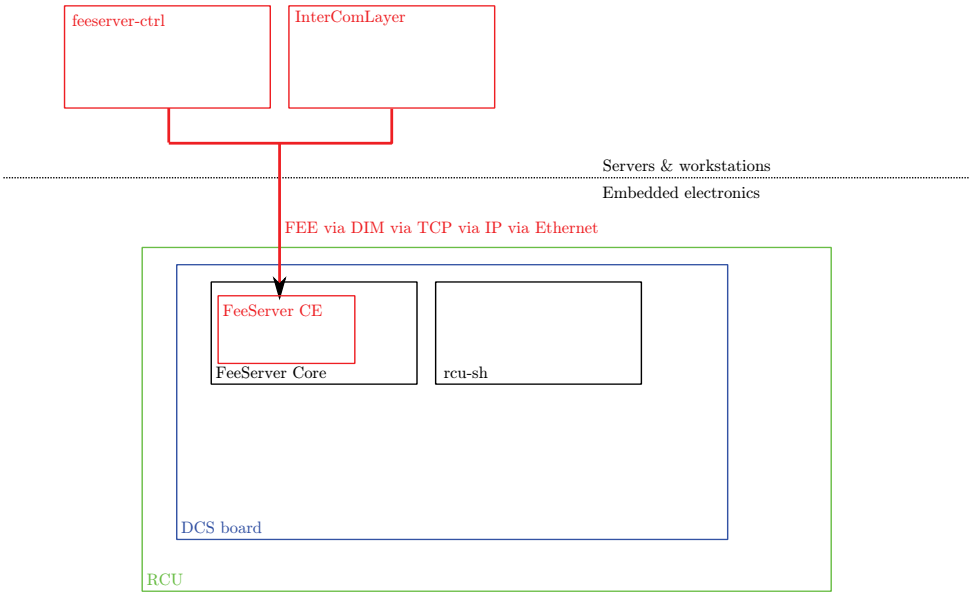

Figure 4.1: Block diagram of the communication between the RCU, DCS board, FeeServer CE, FeeServer Core and client software connecting to the FeeServer via network.

commands to the FeeServer, whereas the services are used to forward values published by the FeeServer.

The FeeServer is divided into two parts: a *general* part, and a *specific* part. The interface between them consists of functions declared by the general part and implemented by the specific part. Finally, the parts are linked as a single executable. Figure 4.1 shows a block diagram of the communication between the different components.

4.1 FeeServer Core

The general part is called the FeeServer *Core*. It provides the *core* framework for publishing and updating services, and receiving commands via the DIM and FEE protocol. It knows however nothing about *what* is published, or *how* to interpret the received commands. This is the responsibility of the specific part. There are several versions of the FeeServer, all adapted to a specific usage. However, the Core is shared by all. Further details on the FeeServer Core are given in [70].

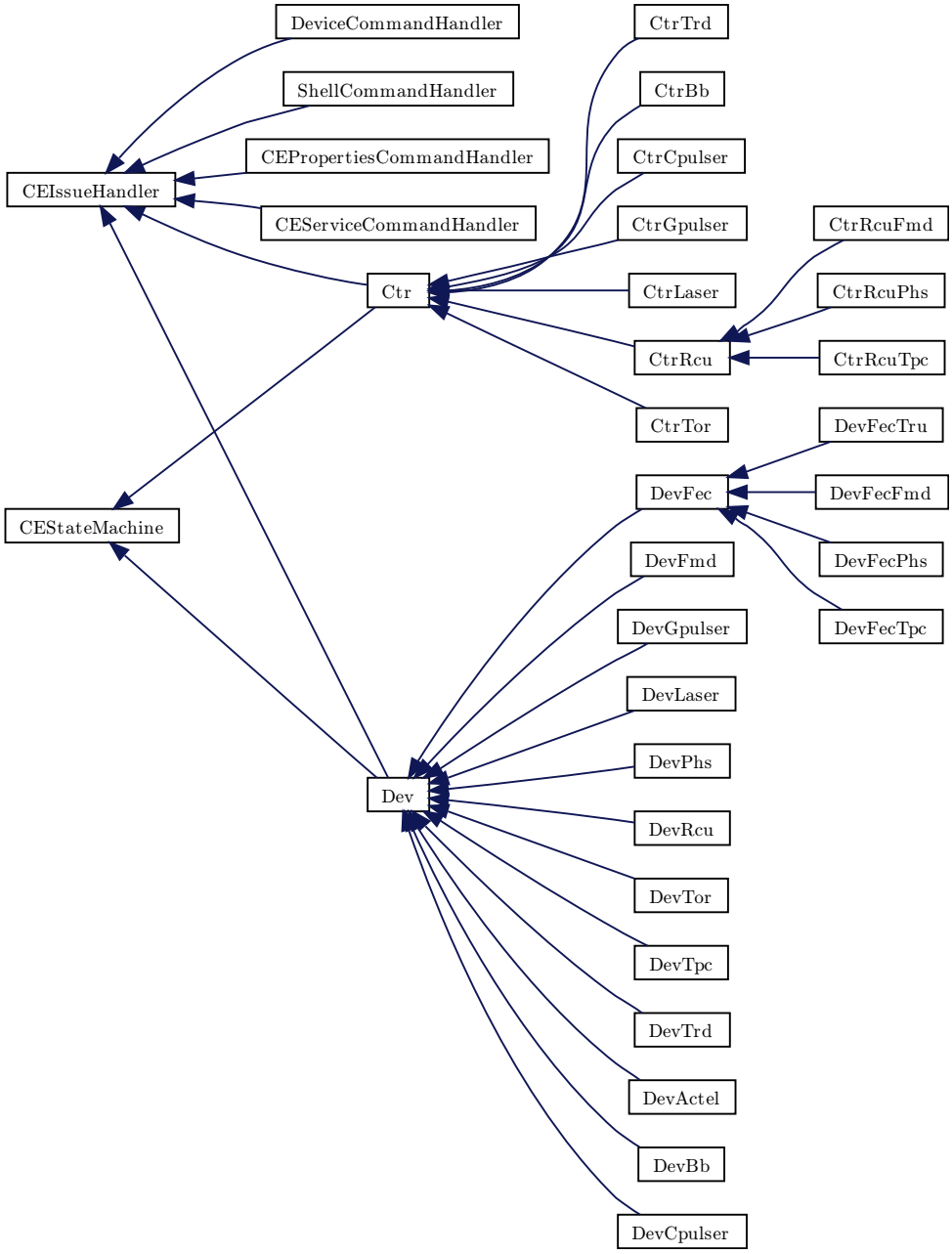

Figure 4.2: Inheritance diagram of the main FeeServer CE classes. Note that the framework prepared for TRD and FMD are not used, since they choose to develop separate CEs.

4.2 FeeServer ControlEngine

The specific part is called the FeeServer CE. It is responsible for providing the *specific* implementation for updating of published services, and interpreting commands received by the Core. Originally, the interface between the Core and the CE was devised as the starting point for developing FeeServers for different purposes, such as for TPC, PHOS, BusyBox, *etc.*; *i.e.* the CE would be completely different for all species of the FeeServer. However, the functionality and complexity of the CE has increased much. Most of the CEs share very similar requirements for state machine, defining devices and handling of commands and services. This has made it possible to develop a common framework, shared by all CEs, on top of the Core–CE interface. Using this, a FeeServer for a new appliance can be created simply by implementing a few virtual classes from the CE framework.

The relation of the most important classes are given in Figure 4.2, showing the class diagram.

The framework of the FeeServer CE is based is based on the following entities:

- *control engine* — the main part of the FeeServer CE, user of the other classes;
- *device* — represents a particular part of the FEE of particular interest, also hardware access;
- *service* — interface for publishing values via FeeServer Core;
- *issue* — command handling;
- *state machine* — customisable to specific requirements.

The FeeServer CE has gone through several development iterations. Most importantly, the current version is object oriented and based on the *C++* programming language. In the past, the CE was implemented in plain *C* [71, 72].

4.2.1 Control engine

As already mentioned, there are many versions of the FeeServer, adapted to different sub-detectors and auxiliary systems. The framework provides the *Ctr* class as the base for implementing new CEs, providing common functionality needed by all. It is in turn based in the *ControlEngine* class, which implements the basic interface to the FeeServer Core. By also inheriting from the *CEStateMachine* and *CEIssue*, a state machine and command handling are provided as well. The state-machine is used for the overall *MAIN* state of the FeeServer. All CE commands sent to the FeeServer are received by the CE; it is the responsibility of the CE to dispatch the commands to the right devices.

A new CE is made by inheriting from the *Ctr* class and over-riding the needed functions. If an (almost) completely different CE is needed, it is possible to derive a CE for the FeeServer of a given appliance directly from the *ControlEngine* base class, however this will disable most of the framework; essentially requiring a full re-implementation.

There can only be one instance of the ControlEngine class; the type of the class determines the identity of the FeeServer. This should be obvious; the very same instance of a FeeServer can not have both, say, TPC and PHOS identity!

4.2.2 Device

A *device* is the FeeServer CE's representation of hardware. It can be a whole card, like an RCU or a FEC, a bus, a specific chip, like the Actel, a full system, like the Busy-Box, or just a module of a firmware design, or any other entity that it makes sense to distinguish physically or logically. The framework provides a base class, *Dev*, to simplify the creation of new devices for the FeeServer. The *Dev*-class already inherits from *CEStateMachine* and *CEIssue*, which will provide any new sub-class of *Dev* with a framework for a state machine and command handling. In addition, the *Dev* class provides default implementations of a state machines that is always in state *ON*, and a command handler without any commands defined. Either or both can be used directly by trivial devices that does not need this functionality, but only wishes to take advantage of the *Dev* framework, or as a starting-point for more complex devices that will over-ride the needed functions.

Some of the most important virtual functions that can be over-ridden by *Dev* sub-classes:

- *ArmorDevice()* — initialisation (armour) of the device;
- *EvaluateHardware()* — determine the state of the device (usually the hardware) at start-up;
- *ReEvaluateHardware()* — determine the state of the device at run time;
- *EnterState*NAME*()* — executed when device entering state *NAME*;
- *LeaveState*NAME*()* — executed when device leaving state *NAME*;
- *SwitchOn(...)*, *Shutdown(...)*, *Configure(...)*, *Reset(...)*, *Start(...)*, *Stop(...)* — executed when corresponding *transition* occurs;
- *GetGroupId()* — return the group *IDentification* (ID) for command handling;
- *issue(...)* — implementation of binary command handling;
- *HighLevelHandler(...)* — implementation of high level command handling;
- *PreUpdate()* — executed *before* a (service) update cycle;
- *PostUpdate()* — executed *after* a update cycle.

In addition, a number of other function exists. A complete list can be found in the *Dev.hpp* file [73].

It is also possible to define devices outside of the *Dev* framework provided. Either a completely stand-alone class can be made, or just inheriting from either *CEStateMachine* of *CEIssue*. However, this is in general not recommended, as they can not be used along other *Dev*-objects in the framework, but exists as an option for very special cases.

4.2.3 Service

A *service* is a data-point published by the FeeServer via the DIM protocol. Henceforth, a FeeClient can subscribe to this service. Whenever the value of the data-point is changed, the FeeClient will receive an update from the FeeServer. The opposite is also possible; a FeeClient can use the service channel to *set* the value of the data-point. Obviously, this will only work if the FeeServer supports this for the given channel, *and* it is physically possible. For example, trying to set the value of a data-point representing the temperature measured by some sensor will *not* change the temperature. Since the FeeServer procedure for getting and setting a data-point is completely independent, it is also possible to define a service that can only set a value, but not get it. A typical example would be a hardware register that can only be written, *i.e.*, writing a value will give an instruction to the firmware, but the instruction can not be read back.

From the DIM protocol, a service is identified by its *name*. The combination of FeeServer name and service should be unique.

There are two ways of publishing a service from the CE: either using the low-level *RegisterService(...)* function; or the high-level framework from the *Ser* class. In general, the later option is preferred for general numeric register values, and the former for complex services that can not easily be generalised.

Publishing a service is a two-step process. First, at FeeServer start-up the services to be published has to be registered. This is done by calling the *RegisterService(...)* function from the *ArmorDevice()* function of every class wishing to publish services. The second part is the FeeServer update cycle, where it will check for all services whether an updated value is available. There is a one second break after the update cycle has finished before the next begins. To check for updated values, a user-specified function is called. There is also a user-specified function to call if a FeeClient is trying to set the value of a service channel. Either (or both) can be *NULL*, disabling the set or get functionality.

A service can have one of three data types: *float*, *integer* or *string*. The two first are considered *numeric* and are treated slightly differently than the *strings*. Every numeric service can have *deadband*. If the change of the value of the data-point is within this, it will considered not to have changed, hence not updated to the FeeClients. *0* is a valid value for the deadband.

In some cases, like hardware registers, the code for obtaining the values from many data-points is identical; the only change is the address of the register. The *RegisterService(...)* allows registration of three free parameters that will be passed the call-back functions. Using these to identify the specific service, for example as indices to arrays or object pointer, the same call-back functions can be used for several similar services. This scheme is also used for the call-back functions setting values.

Parameter listing of *RegisterService(...)*:

- *enum ceServiceDataType type* — the type of the service, *float*, *integer* or *string*;
- *const char* name* — service name;
- *float defDeadband* — deadband width;
- *ceUpdateService pFctUpdate* — *get* call-back function;
- *ceSetFeeValue pFctSet* — *set* call-back function;
- *int major*, *int minor*, *void* parameter* — free parameters that will be passed to the call-back function, typically used when same function handles several similar services and needs to identify the specific service, for example they may constitute indices to arrays or pointer to object.

Using the *RegisterService(...)* function directly requires every service to have its own static call-back functions. However, more often than not, this task can be off-loaded to the framework. Most of the services being published are "verbatim" hardware registers of either the DCS or the I^2C buses. The CE provides a framework for publishing such "general" services, the *Ser* base class. Every service is represented by an instance of a sub-class of *Ser*. The constructor of such object will automatically register the service using the *RegisterService(...)* function.

The DCS bus has several modes of operation, depending on hardware and firmware configuration, each giving access to a pre-defined memory on the auxiliary board to which the DCS bus is connected. For the RCU, the *DevMessagebuffer* class mode *1* gives access to the message buffer, mode *2* to the select map, and *3* to the flash.

The constructors of the general service types defined so far:

- *SerMbAddrS(std::string name, unsigned int address, signed int convfactor, signed int deadband, DevMsgbuffer* msgbuffer, unsigned int mode=1)* — signed integer from DCS bus;
- *SerMbAddrF(std::string name, unsigned int address, float convfactor, float deadband, DevMsgbuffer* msgbuffer, unsigned int mode=1)* — float from DCS bus;
- *SerFecRegS(std::string name, unsigned int fec, unsigned int reg, signed int convfactor, signed int deadband, DevFecaccess* fecaccess)* — signed integer from I^2C bus;
- *SerFecRegF(std::string name, unsigned int fec, unsigned int reg, float convfactor, float deadband, DevFecaccess* fecaccess)* — float from I^2C bus.

Parameters for the constructors, in order of appearance:

- *std::string name* — service name;
- *unsigned int address* — register address of DCS bus;
- *signed int convfactor* — conversion factor, can be *1*;
- *signed int deadband* — deadband, can be *0*;
- *DevMsgbuffer* msgbuffer* — pointer to *DevMessagebuffer* object to use;
- *unsigned int mode=1* — DCS bus mode, as used by *DevMessagebuffer*;

- *unsigned int fec* — hardware number of the FEC;
- *unsigned int reg* — register address on the FEC;
- *DevFecaccess* fecaccess* — pointer to *DevFecaccess* object to use.

For both float and integer services, the values are read as integers from the hardware registers, then multiplied by the conversion factor specified in the constructor; enabling publishing of for example "real" temperatures. The services of float data type is converted to floats as part of the multiplication.

The *Ser* framework provides standard call-back functions for both getting and setting data-points for all the currently defined service types. Whether an attempt to read or write a given register *actually* succeeds, will depend entirely on the hardware read–write nature of the register; trying to set a temperature sensor register will not change the temperature!

It is of course possible to extend the *Dev* class with sub-classes for more data types, if so should be needed. In general, the extra overhead of extending the framework will only be worthwhile for general types of data-points that will occur multiple times, preferably in multiple classes. For exceptional types only used once, it might be more efficient to use the *RegisterService(...)* function directly without going through the ControlEngine framework.

4.2.4 Issue

Commands are received by the FeeServer Core via the command-channel. After determining the command is destined for the CE and not itself, the command is forwarded to the CE. Any class, like *Dev*, wishing to be able to receive commands, have to inherit from the *CEIssueHandler* class, and implement the functions *issue(...)* and *GetGroupId(...)*. The first function will contain the actual implementation of the commands the class can handle. Every class has to be assigned a *command group* ID, which should be return by the last function.

When a command is received by the framework, the header is inspected, and the command group ID is extracted. The ID is compared to the ID of all registered classes. Henceforth, the command is forwarded to *issue(...)*-function of the class with matching ID. If none is found, an error message is returned. Each command group has a set of defined *commands*. The unique combination of a command group ID and command gives the *command ID*. Both are defined in the file *rcu_issue.h*, with corresponding command names. Any application wishing to send commands will typically include this file for easy mapping from named to command codes.

Each class is in principle free to implement the *issue(...)*-function as it sees fit. However, the typical implementation will be based on a *switch* for the command ID. The *issue(...)*-function will return the number of bytes processed. The data sent via the

command channel may contain several commands concatenated. When an *issue(...)*-call returns, the number of bytes processed is subtracted. If the framework finds another header, the new command ID is extracted, and the corresponding *issue(...)* function is called. This process is repeated until there are no further commands.

The *Ctr* class use a "special" command, *FEE_CONFIGURE_32*, to initiate a configuration sequence. This command will contain "ordinary" commands. During the configuration, the FeeServer *MAIN* state will be *DOWNLOADING*; when it has finished, *CONFIGURED*.

In addition, there are high-level commands, which are "human-readable" commands, in contrast to the ordinary commands, which are purely binary. For example, by sending *<fee>CE_SET_LOGGING_LEVEL 0</fee>* via the command channel, the logging level of the FeeServer is set to *0* (debug). Support for such commands is enabled by implementing the *HighLevelHandler(...)* function. A typical implementation will involve searching for a text string matching the name of the command. Such search has to be explicitly implemented for a command; the framework will not attempt to automatically convert high-level commands to commands found in the *rcu_issue.h* file. Since their are no IDs to identify, the framework will call the *HighLevelHandler(...)* function of all classes until a match is found.

In addition, there is a special group of high-level commands for triggering certain state transitions for the FeeServer, which are identified by *action* rather than *fee*. As an example, *<action>GO_STANDBY</action>* will make the FeeServer enter the state *STANDBY*.

4.2.5 State machine

The framework provides a *state machine* for classes that need this functionality. Any class deriving from the *CEStateMachine* base class will have its private state machine. The *Dev*-class, base of all *devices*, already inherits from this class.

The behaviour of the state machine can be customised for each class individually. This is done by over-riding functions from the base class, and defining transitions between states. All states are identified by a unique numerical ID, with a corresponding name. The number can not be changed, however, the name can. By default, a set of states are provided, most of them corresponding to states used by PVSS, in addition to a number of *user* states. Both types of states can be re-defined, however, this should only be done exceptionally for the PVSS states. Since both IDs and names are matching those used by PVSS, a re-definition will give a state with the same ID as a PVSS state, but with a different name. In addition the PVSS states have certain pre-defined transitions; this should be considered if re-defining such states. For consistency, PVSS states should be used whenever possible, however, in some cases additional states are needed.

The framework allows for defining new transitions between any states. A transition will have a name, and will take the state machine to a specific state. However, the transition may be allowed to start from any of a specified list of states. Additional checks may be specified before allowing the transition to take place.

By overloading functions of the base class, it is possible to specify logic to be executed whenever the state machine enters or leaves a state. Code can also be executed in conjunction with the transition itself, *i.e.*, not tied to the specific state that is entered or left, but by the actual triggering of the transition. This is useful as several transitions may start or end at a certain state, but depending on the purpose of the transition, not always the same code should be executed. For example, if the state machine is modelling the some underlying hardware, the code related to entering and leaving a state may set the hardware in a "physical" state corresponding to the state machine state, while the instructions for the transition will represent a higher level configuration. If the states are *on* and *off*, the hardware will be turned on when the state *on* is entered, and turned off when state *off* is entered. A transition *configure* might end in state *on*, but also configure the electronics as part of the transition.

The *meaning* of each state of a given state machine, is to a very large extent determined by the class itself. In some cases, it may represent real hardware. This is in particular the case of the *Dev*-class and derivatives. In other cases, it may represent the abstract state of software. The state of certain classes may influence the state of other classes. The *MAIN* state machine represents the "overall" state of the FeeServer. However, this does not necessary imply that all other state machines are in the same state as the *MAIN*. Rather, the logic of how the *MAIN* state machine reacts to the state of some other state machine has to be programmed explicitly, where a high-level understanding of the overall system is paramount. Typically, the *MAIN* state machine will enter state *error* if any of the sub-ordinate state machines is in this state. However, this does not have to be the case. It is possible to imagine cases where the state of a given state machine is determined not to be of any, or at least not sufficient, importance to the overall system. Optionally, the *MAIN* state might be *mixed*, to signal not all state machines are in a state that trivially can be translated to state *running*, but still the overall state is not *error*. Any state machine might take the state on other state machines into account.

The inter-dependence of state machines, and that the individual state machines can have different sets of states, transitions, and criteria for allowing a given transition, arguably makes the state machines the overall most complex part of the FeeServer, even though the basic "building blocks" are fundamentally simple.

4.2.6 Base classes and inheritance

Creating a CE for a new appliance is done by implementing a few derived classes from the framework base classes, *Ctr*, *Dev* and *Ser*. The *Ctr*-derived class can be considered the CE itself, and is the only "required" class for a new FeeServer CE (however, not very useful since it would not be able to interact with hardware without *Dev*-classes). Classes based on *Dev* are representing the various pieces of hardware the CE needs to control and monitor. Most importantly, the command handling is implemented (the *issue(...)*-function) and services registration. In principle, *all* hardware of a certain CE can be supported in a single class derived from *Dev*, however, it is considered "good practice" to split physically or logically distinct functionalities into separate classes. For "simple" appliances, like the BusyBox, all functionality will fit into the BusyBox "container", whereas more complex systems, like the *TPC* and *PHOS* several *Dev*-classes are needed. The instances of the classes derived from *Dev* is created and destroyed by the *Ctr* class. Typically, all instances live for the entire life-time of the FeeServer.

It is rarely necessary to derive new classes from the *Ser* class, as the default provided types are sufficient. For "one-of-a-kind" services, it is most likely easier to use the FeeServer Core framework directly rather than deriving a new *Ser*-class. However, for types that will be used several times, the overhead of implementing a new class can be justified.

The framework has been designed to make creating a new CE easy. Both the *Ctr* and *Dev* base classes are designed to provide meaningful default implementations of most functions, for example state machine and command handling, so that appliances that do not need some of this functionality do not have to make an effort implementing it. It is possible to create a new, basic CE prototype literally within minutes.

4.2.7 Interrupt handling

The DCS bus has *interrupt* lines that may be used by the RCU and other users of the bus to gain the attention of the FeeServer. Specifically, the FeeServer listens to the *SIGUSR1* interrupt, which is mapped to one of the DCS bus interrupt lines. Other lines may also be used. The device file of the interrupt driver is */dev/irq/irq*.

The main use case for the interrupt is the *RCU*. If one of the *FECs* experiences an error, it will send an interrupt to the RCU, which in turn will signal an interrupt to the DCS board and the FeeServer. The FeeServer is then expected to read an error register on the RCU to determine the cause of the interrupt, both which FEC and the type of error. After decoded the contents of the error register, the FeeServer will clear the register. Depending on the error, the FeeServer will determine what action to take. For less serious incidents, it may choose to inform upper layers, or in some case, ignore the error completely. For more serious errors, direct intervention on the FEE may be

needed. If the error is also defined as an *hard* error, the RCU will automatically turn of the FEC in question. In that case the FeeServer may try to turn the FEC on again, if it is considered safe. This will also require re-configuration of both ALTRO registers, including pedestal memories, and BC registers, since they are cleared when turned off. Such re-configuration will require both the IM instructions for setting the ALTRO and BC registers, as well as the pedestal values to be stores in the FeeServer (same as for verification).

So far, interrupts have not been enabled in the RCU firmware, as it is desirable to collect real-world statistics for the frequency hard errors that will turn off FECs, and the ratio of *actual* errors over *erroneous* errors. Turning off FECs unnecessarily will make the event data incomplete, though it will not stop data taking. Consequently, only the basic support for interrupts has been implemented in the FeeServer so far, not the actual *handling* of interrupts outlined above.

The gate pulser FeeServer is planed to be a "test bench" for interrupt handling. The gate pulser will use interrupts for a different purpose than the RCUs.

4.3 Versions

A number of variations over the FeeServer exist for the different sub-detectors using it, as well as other auxiliary equipment. In addition to the FeeServer described below, TRD and FMD relay on different implementations of the FeeServer, that to some extent are compatible with this FeeServer.

4.3.1 TPC and PHOS

The PHOS and TPC FeeServers are closely related, as they both relay on the RCU and ALTRO, and uses identical bus structure with FECs for data read-out. Hence, the command set is almost identical. The only exception is functionality that only exists on either type of FECs. Mainly, this is related to the different design of the BCs, as well as presence of APDs and TRUs for PHOS. To support this, the PHOS FeeServer has some additional commands. So far, there has been no need for TPC-specific commands.

Among the FeeServers, the TPC and PHOS FeeServers have the most extensive command set.

A major part of the TPC and PHOS FeeServers is related to the FECs. In particular, the FeeServer requires access to them. This is handled by the *DevFecAccess* class. Originally, it was based on the *Dev* base class, but has later become an independent class since none of the functionalities provided by the *Dev* class was needed. As the class is currently only used for monitoring and control of the BCs, it can only access the FECs via the I^2C bus, but can easily be extended to also support access of both BCs and

ALTROs via the GTL bus. Access via the GTL bus will interrupt data taking, hence limiting the usage to periods of time when data is not taken, *i.e.* before data taking. The exception is if ALTRO verification during the read-out orbit gaps is implemented. For this case, information from the TTCrx chip will be used to determine when the GTL bus is not busy. The class is also used for monitoring and publishing BC registers, such as temperatures.

Configuration of ALTRO and BC registers before data taking is done via binary instructions written to the IM of the RCU. A *Binary Large OBject* (BLOB) of instructions is assembled by the *CommandCoder* (CoCo) and sent via the *RCU_EXEC_INSTRUCTION* FeeServer command, which will write the BLOB to the IM and execute it. In addition, the FeeServer supports storing IM instructions in memory or on local disk for faster configuration. The *RCU_ALTRO_INSTRUCTION_STORE* instruction will store the IM instructions BLOB to local FeeServer memory rather than immediately executing it. Issuing *RCU_ALTRO_INSTRUCTION_EXECUTE* will cause IM instructions stored in memory to be written to the IM and executed. *RCU_ALTRO_INSTRUCTION_READ_FILE* and *RCU_ALTRO_INSTRUCTION_WRITE_FILE* will read and write the BLOB to a file, respectively. To clear the memory, *RCU_ALTRO_INSTRUCTION_CLEAR* can be used. Utilising this scheme, IM instructions can be sent to the FeeServers only once, then re-executed for every start of a run. This is particularly important with the current ICL, which only permits configuring the FeeServers sequentially. The IM instructions stored in memory will also be needed for ALTRO verification, if implemented. Storing the BLOB to a file makes the configuration persistent even in case of FeeServer restart.

The pedestal memories of the ALTROs are used for baseline noise subtraction. Each channel has a ten-bit memory matching the width of the ADC. In the past, the RCU firmware was aware of this, and had functionality in the form on a dedicated memory area to aid setting these values. However, in the current implementation of the firmware, this memory no longer exists. Rather, this is now handled by the FeeServer. The *RCU_WRITE_ALTRO_PEDMEM* command will write 1024 pedestal values for a given channel. Since writing the pedestal values to the ALTROs has to go through IM instructions, the FeeServer will embed the pedestal values into proper IM instructions internally. To save bandwidth when transferring the values, as well as memory when processing them, three ten-bit words are truncated into the first 30 bits of a 32-bit word, giving a total of 342 words as payload to the command. Storing the pedestal values to memory or file, like the IM instructions above, has been considered. However, since the data size for pedestals is vastly larger than those typically expected for configuring the ordinary ALTRO registers, some additional care is needed. If verification of the ALTRO pedestals is desired, having a copy of the pedestal values in memory is needed.

TPC

The TPC is historically the "original" user for which the FeeServer CE was developed. Later the CE was generalised and extended to other appliances. With instances running on 216 DCS boards, it may also be considered the "main" user. Besides the TPC FeeServer itself, many of the other FeeServer species, like the *laser*, *etc.*, is part of the TPC ecosystem.

PHOS

The PHOS APDs need individual bias voltages for optimal performance. These voltage values are written to dedicated registers in the BCs, which again control embedded voltage regulators. A command, *PHOS_APD_INIT*, implemented by the *DevFecPhos*, is responsible for setting the APD voltage settings for a given FEC. The payload is organised as 32 words, one for each APD, of 32 bits. Each word consists partly of the actual APD voltage value, and partly of the corresponding Hamming-code for the value. The Hamming code has to be calculated in advance by the CoCo, and will be used by the BC to verify the integrity of the APD voltage settings. If it is not set, *i.e. zero*, it will not be used. Simple errors may be corrected by the BC. Non-correctable errors will be signalled in a BC register, monitored by the FeeServer. The FeeServer is keeping a copy of the APD values in memory, thus enabling re-programming of the register. Currently, the APD values are not being written to file recover from restarts. However, this can easily be implement if needed.

Each PHOS FEC branch has a special FEC, a TRU, for quickly detecting particularly energetic event to trigger other detectors. Since the register lay-out of the TRU is different from the ordinary FECs, it is also implemented as a separate class, *DevFecTru*.

So far, EMCAL is using the same FeeServer as PHOS, as the requirements are very similar. If the requirements at some point should diverge, a separate version can easily be made.

4.3.2 Trigger-or

The TOR is used in conjunction with the PHOS TRUs to provide level *0* and *1* triggers. This FeeServer is not very complex, and mainly supports basic commands for setting configuration registers, as well as publishing relevant services.

4.3.3 Busy-box

The Busy-Box is used to determine whether a sub-detector is ready to be triggered for a new event. The main command, implemented by the *DevBb* class, is *BB_INIT*, which will configure the Busy-Box with a typical, pre-defined, configuration set. If further

customisation of the configuration is needed, it also provides a set of instructions to configure each configuration parameter individually. For each FPGA, a string of 128 characters, each representing the state of a given DDL link, as read from the status memory is published. The characters are either *0*, *1*, *2*, *3* if these values are indeed read from the status memory; if some other value is read, the character *U* (*u*nknown) is published; if reading fails entirely, *F* (*f*ailure) is published.

4.3.4 Laser synchronisation

The laser system generates 336 laser tracks inside the TPC, providing a very accurate and valuable tool for calibration. The FeeServer of the laser system is currently the most complex of the FeeServers for "auxiliary", non-detector appliances. Apart from providing a set of both high- and low-level commands for explicit configuration, it mainly utilises implicit configuration via *states*. By externally triggering the transition to a certain state, a corresponding set on configurations is applied. The functionality is implemented in the *DevLaser* class.

4.3.5 Gate pulser

The gating grid can "open" and "close" the TPC read-out chambers for drifting electrons. Similarly to the FeeServer of the laser system, the FeeServer of the gate pulser will change the configuration when transacting to certain states, in addition to provide high- and low-level configuration commands. As for most FeeServer, a set of relevant services is published. The gate pulser FeeServer is likely to be the first to take advantage of interrupts. The class of the gate pulser is *DevGpulser*.

4.3.6 Calibration pulser

The calibration pulser is for calibrating the read-out electronics of the TPC. So far, the FeeServer for this device has not been implemented.

4.4 General DCS infrastructure

The FeeServer is part of a larger over-all DCS framework, the main parts of which will briefly be introduced in the following.

4.4.1 InterComLayer

It has been mentioned that there is a software, the FeeServer, running on the DCS board whose purpose is to relay communication between the FEE and the network.

The main client for this server is the ICL. It can best be described as a hub in the DCS. Downwards contact is maintained with all DCS boards of the sub-detectors (TPC, PHOS and EMCAL). Also, it is connected upwards to PVSS. Horizontally, it accesses the configuration database.

The purpose of the configuration database is twofold. Firstly, it contains the layout of the FeeServers it is connecting to: base name, X, Y and Z dimensions, and list of all services provided by the FeeServers. For TPC, X is side, Y the sector and Z the partition. For PHOS, X is always 0, Y is the module, and Z the RCU of the module. Secondly, it contains sets of configurations for all FEE. All registers of all RCUs and FECs can be configured individually. Several configurations are allowed; each identified by a configuration number. When the FEE is to be configured, PVSS sends the configuration number and a list of FeeServers to receive the configuration. An integrated part of the ICL, the CoCo, is responsible for retrieving the relevant parameters from the data base and assemble binary data blocks the FeeServers can interpret. The CoCo is unique for each sub-detector relying on ICL.

ICL is subscribing to the services published by the FeeServer. The complete name of the services is constructed from the base name and the coordinates, onto which the individual service names are appended for each FeeServer. The services will again be exported from the ICL as a single channel containing name–value pairs. PVSS will subscribe to this channel for display on the operator GUI and for logging. It is in principle possible for PVSS to subscribe to the service channels directly from the FeeServer. However, PVSS can at most receive name–value pairs at a rate of approximately one kHz. The FeeServer will update those channels whose values have changed every second. Especially at start-up, when the temperatures have yet to stabilise, it is realistic to expect most of the channels to indeed be updated every second. Depending on the number of FECs, and the number of enabled services, each FeeServer may typically publish 100–150 services. For 216 FeeServers, this will total to about 30000 updates per second; *i.e.* about 30 times the limit of PVSS. Also, consider that the actually published services is just a sub-set of the exhausted list of available registers to monitor. Hence, filtering is needed. ICL will buffer the service values received from the FeeServer internally. If ICL receives an update for a service before the already buffered value has been forwarded to PVSS, the previous value will be overwritten by the new one. This can be thought of as a low-pass filter; the update rate is reduced to a level that is feasible for PVSS to monitor. However, values will only be discarded if the actual update rate from the FeeServers combined exceeds one kHz over longer periods of time. At stable running conditions the monitored registers will not change very frequently; it is realistic to expect the update rate to fit comfortably within one kHz.

Also from the FeeServer point-of-view, this relay mechanism is positive. Any additional client wishing to subscribe to the FeeServer channels can do so via the ICL; without

inducing further load on the comparatively computing power limited DCS board.

To further increase the performance configuring and service updates to PVSS, each TPC side is handled by separate servers for ICL and PVSS.

4.4.2 ICL interaction

The main user of the FeeServer is the ICL. Other users, like *Dim Information Display* (DID) and *feeserver-crtl* are mainly casually used as debug tools. The ICL has a list over all FeeServers and services it wishes to subscribe to. It also assumes there are standard communication channels, most importantly for *commands* and *messages*, which the FeeServer is expected to provide. The communication relies on the DIM protocol, which will set up the low-level communication channels.

As the name hints, the command channel is used by the ICL to send commands to the FeeServer. All commands are wrapped in global headers and trailers to identify them as valid commands to the FeeServer. Further, all commands are organised with a command ID. The ID tells the FeeServer which function will handle handle said command. A command, for example for reading a register, might return data via the communication channel.

The message channel is relying casual messages from the FeeServer to the ICL. These messages are divided into several *classes* of severity, for example *debug*, *info*, *warning* and *error*. The ICL in turn will forward the messages to PVSS where they may be displayed or logged. Typically, the messages will inform on various incidents happening during the course of FeeServer command handling or service update.

The ICL is configured to subscribe to a set of FeeServer *services*. Whenever the FeeServer detects there is a change in the underlying data the services points to, it will update the value of the service via functionality of the DIM framework. Henceforth, the ICL will be notified of the updated values, and can retrieve and forward it to PVSS.

4.4.3 Configuration database

The configuration database contains all parameters for configuring the FEE, as well as information needed by the ICL to determine the logical layout of the sub-detector. All registers of all RCUs and FECs can have individual values. There is also a need to maintain several complete sets of configuration parameters, tailored for the intended use. For example, many parameters will be different for beam events and cosmics events. Another example is the case of calibration runs, such as pedestals, where the level of background noise in the detector is determined. Each such configuration is identified with a “configuration number”.

Upon configuration, PVSS will send a command containing the configuration number and a list of FeeServers to receive the configuration. The CoCo will retrieve the

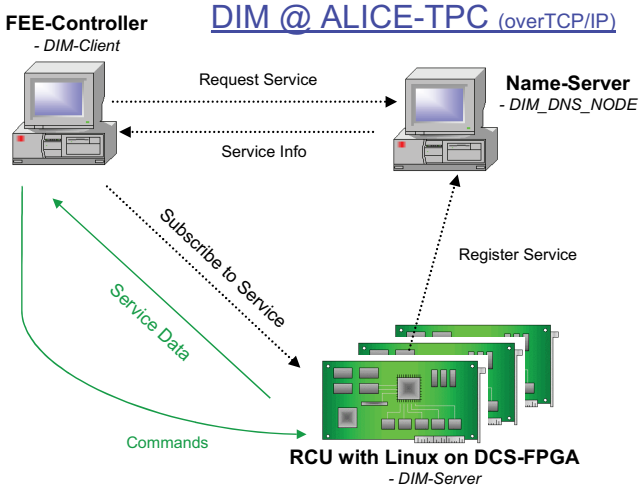

Figure 4.3: The interaction between FeeServer, DIMNS and FeeClient (*e.g.* ICL) in the DIM framework.

corresponding information from the database and assemble binary blocks, containing the configuration parameters, the FeeServers can interpret.

4.4.4 PVSS

PVSS is the high-level part of the DCS. It has two main parts. Firstly, it provides the infrastructure for forwarding monitoring and control of all sub-detectors to "central" DCS, the ECS. As part of the framework, DCS monitoring values are also being logged. Secondly, it provides the GUI used by the shifters to operate the detectors. PVSS is a CERN sanctioned standard framework, common for all detectors and experiments part of the LHC. It provides a relatively user-friendly point-and-click GUI usable for relatively inexperienced shifters, who by no means are experts on all parts of the system they control.

4.4.5 DIM

DIM [74, 75] is a high-level network protocol running on top of IP/*Transmission Control Protocol* (TCP) for exchanging commands and short messages between nodes, developed by CERN as the standard protocol for this purpose. It is based on the client–server model. Practically speaking, the FeeServer is the server, and the ICL the client. However, the ICL is the server in the communication with PVSS, where PVSS is the client. These two communication interfaces are independent, though. It is possible to build more

Description	*18 FEC*	*20 FEC*	*25 FEC*	*Sector*	*TPC*
Configuration size [B]	39 284	43 664	54 512	—	—
Create file from database and CoCo [s]	0.47	0.48	0.51	—	—
Transfer file to FeeServer and execute [s]	2.67	2.67	2.67	—	—
Total time configuration via file [s]	3.14	3.15	3.18	—	—
Time till *running* in PVSS, file [s]	≈ 7	≈ 7	≈ 7	—	—
Time till *running* in PVSS, ICL [s]	≈ 8	≈ 8	≈ 8	≈ 10	≈ 67
Reconfigure from FeeServer memory [s]	≈ 5	≈ 5	≈ 5	—	≈ 15
Pedestal configuration data size [MB]	—	—	≈ 4.7	—	—
Pedestal configuration time [s]	—	—	≈ 120	—	—

Table 4.1: DCS configuration performance

specific protocols on top of DIM. The communication FeeServer–ICL is based on the FEE protocol, while the ICL–PVSS communication is based on the *Front-End Device* (FED) protocol; two derivative DIM protocols. For the FEE protocol, each service has its separate channel. In contrast, for the FED protocol, all channels are "multiplexed" into one channel as a stream of name–value pairs. This allows a large number of services to be transferred without having to create individual channels for them.

A central part of the DIM framework is the DIMNS. Whenever a DIM server starts, it will connect to the DIMNS to register the services it provides, as well as the command channels. Likewise, when a client starts, it will connect to DIMNS to obtain lists of services and command channels available from the registered servers. This will allow automatic reestablishment of the client–server interaction if one of them should temporarily be unavailable. Figure 4.3 shows the interactions between the FeeServer, DIM and ICL.

4.5 DCS operation and performance

The FeeServer has been used for configuring and monitoring the TPC and some other sub-systems under real conditions since the start of the LHC last year. The commissioning period has seen numerous changes and improvements since the initial tests of the DCS.

Table 4.1 shows the performance measurements for configuring the TPC via the FeeServer. The same data are displayed graphically in Figure 4.4, except for the measurements for a full sector and the full TPC. The standard method of configuration is by PVSS sending a command to the ICL to configure a certain set of partitions. This process has been described in detail in previous chapters. In short, the ICL will ask the CoCo to obtain the relevant parameters from the configuration database and assemble binary configuration BLOBs to be executed by the FeeServer. The FeeServer will notify the ICL, which in turn will notify PVSS when this is done. From Table 4.1, this takes

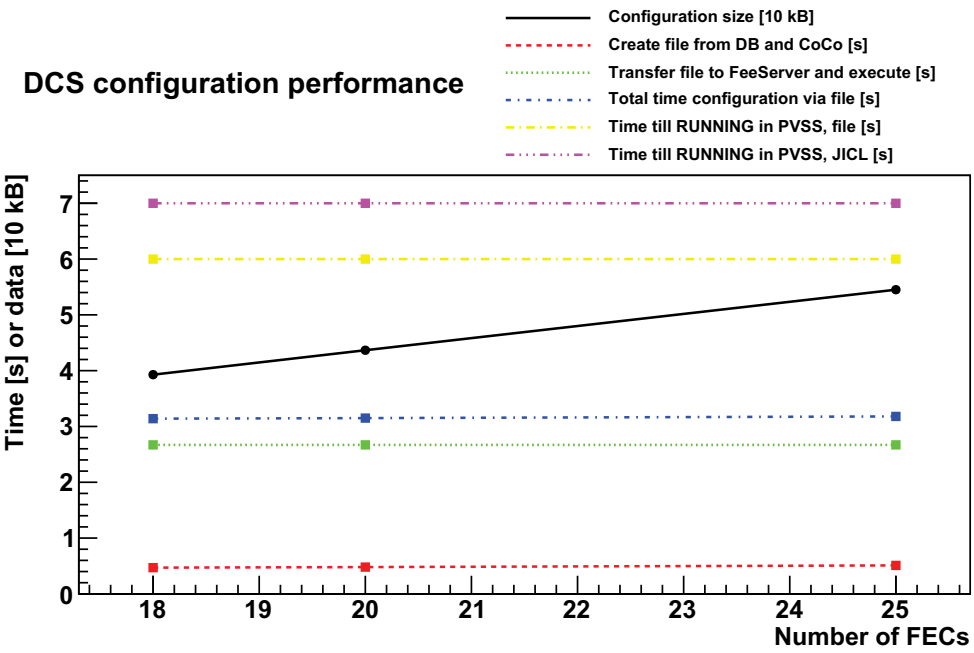

Figure 4.4: DCS configuration performance. Measurements are done for RCUs with 18, 20 and 25 FECs, with straight lines between. The *black* line shows the size of the configuration sent to the respective FeeServers. The *red* line shows the time needed by the CoCo to retrieve the data from the configuration database, and encode it in a format understood by the FeeServer, and store it in a file. The *green* line shows the time spent by *feeserver-ctrl* to transfer the data to the FeeServer, and the FeeServer to execute it. The *blue* line shows the combined time for the two previous steps, *i.e.*, total time for configuration via a file. The *yellow* line shows the time before the state of the RCU changes to *running* in PVSS when configuring from a file. Finally, the *purple* line shows the same, but with configuration done via JICL.

about seven seconds for a single partition, and 67 seconds for the whole TPC, where each side is served by separate ICLs. Currently, the CoCo can only configure the partitions *sequentially*, although the newly introduced JICL, in contrast to the old ICL, also allows for *parallel* configuration. The overall configuration time is expected to improve when this scheme has been implemented.

For measuring the performance of individual components, it can be useful to step outside the ordinary configuration method. Rather than being initiated by PVSS, the CoCo is manually stimulated to retrieve the configuration data from the database, and write it to a file. Through the use of the *feeserver-ctrl* tool, the command blocks are sent to the FeeServer, and the combined transfer and execution time is measured. Table 4.1 gives the time it takes for the CoCo to retrieve and assemble the configuration blocks to approximately half a minute, with a slightly higher value for the more FEC-

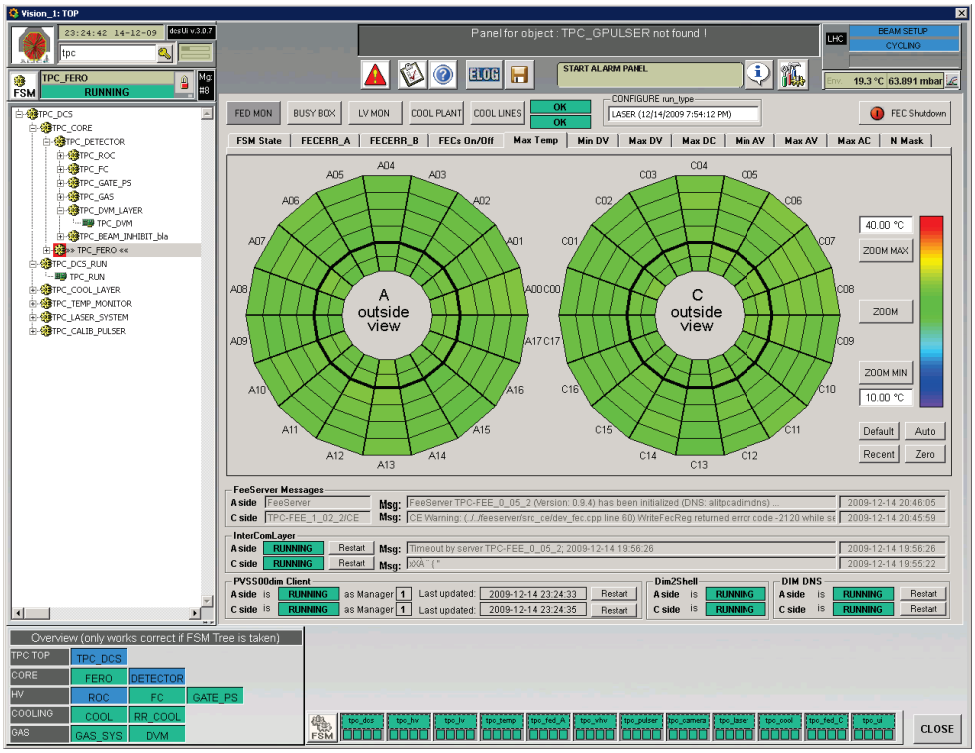

Figure 4.5: A PVSS panel showing graphically the FEC temperatures.

dense partition. Thereafter, 2.67 seconds are needed for the transfer to the FeeServer and execution. Despite the data size is increasing with the number of FECs, no time difference is noted for the transfer and execution time. In total, slightly more than three seconds are needed for retrieving the data from the database and for the FeeServer to execute it, for a single partition.

However, it takes about seven seconds from the file containing the configuration BLOB is sent till PVSS notices that the partition has been configured. At the end of the configuration, the FeeServer will change state to *running*, which ICL and in turn PVSS will notice from the published state channel. This overhead must be shared between the propagation of the state channel through ICL and PVSS and possibly the PVSS state machine. By comparison, running the ordinary configuration chain from PVSS via the ICL, is not much slower, giving an increased configuration time by one second to a total of eight. The FeeServer can store the configuration data locally in the memory of the DCS board. This allows for very fast reconfiguration of 15 seconds, where the ICL only has to tell the FeeServer to re-apply the previous configuration.

The measurements presented above is for configuration without tail parameters and pedestal memory, although the FeeServer can configure these as well. A configuration of

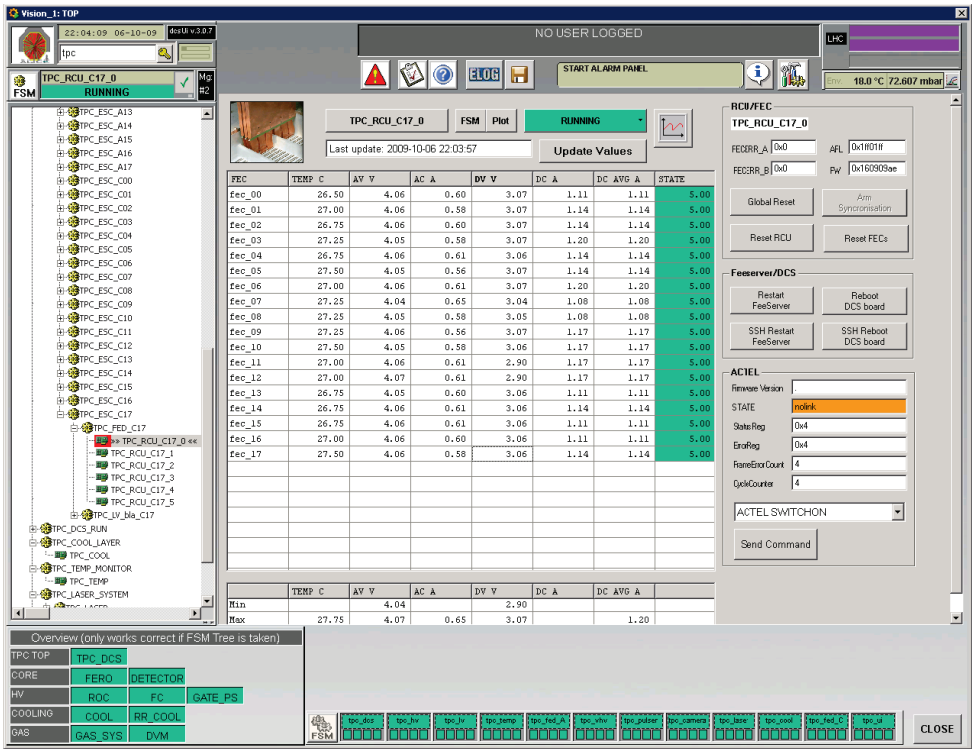

Figure 4.6: A PVSS panel showing numerically the FEC data-points.

a partition with 25 FECs takes about two minutes.

Configuration and monitoring of the TPC and the other sub-systems is done by the shifter via the GUI of PVSS. Figure 4.5 shows the PVSS panel for the TPC. The two barrels are graphical representations of the respective two end-planes of the TPC. In this case the colour represents the temperature of the FEE, as measured by the FeeServer. Although the temperature is measured for each FEC, such level of detail is not needed by the TPC shifter. Rather, the colour shows the highest temperature of the FECs of the partition. To the right, a scale shows the meaning of the colour-code. The scale is chosen so that the temperature of normal operation is green, while a lower temperature is indicated as blue. If the temperature is higher than expected, the partition will turn red. This may be an indication of problems with either the electronics or the cooling. A number of tabs are seen to the top, allowing the shifter to choose which parameters to display, for example the currents and voltages for the analogue and digital electronics of the FEE.

Figure 4.6 shows the values of the data-points the FeeServer is publishing for a specific FEE. This panel can be seen by clicking on the partition corresponding to the FeeServer on Figure 4.5. When the FeeServer detects a change of the value of hardware registers

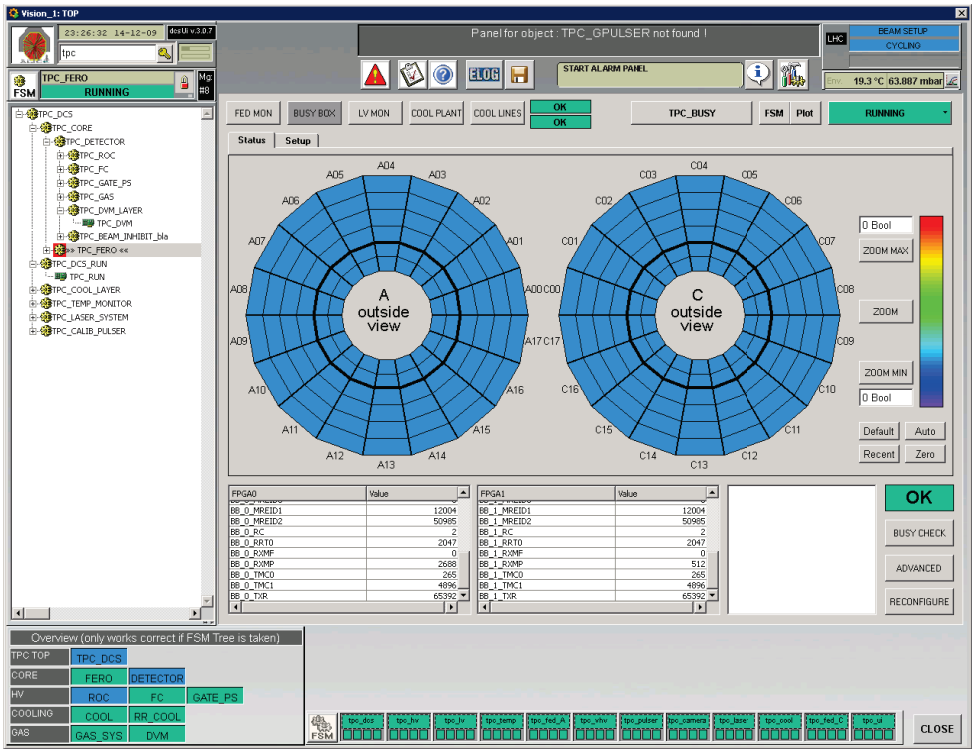

Figure 4.7: A PVSS panel showing graphically the BusyBox data-points.

underlying the data-point, it will update the ICL with the new value. ICL in turn, will update PVSS, and the new value will be displayed in the panels. The name of the columns in Figure 4.5 correspond to the names of the tabs in Figure 4.6.

Figure 4.7 shows the busy status of the *BusyBox* (BB) of the TPC. Since the BB has to keep track of the busy status of all the TPC partitions, the same barrel-display is used. Also here, the colours indicate the status.

Chapter 5

FeeServer refactoring outlook

Throughout the commissioning of the detector, a large number of modifications have been carried out on the FeeServer CEs. Some features that at the design phase were considered useful were in the end not quite as useful as envisioned. Likewise, new features initially not foreseen have been added. Changes to the firmware has required re-thinking of the inner workings of the FeeServer. Also, experience in interacting with it has provided a much more fine-grained picture of how to best control and monitor the FEE.

Overall, the CEs have become much more feature-rich and complex than initially foreseen. In particular, functionality for automatic fault handling has been added, as not to disturb the upper layers of DCS and shifters with recovering from faults that are not critical, and may be experienced from time to time as part of normal operation. The CE contains the parts of the FeeServer specific to the appliance, whilst the Core contains common functionality. The interface between the CE and the Core is at a relatively low level to allow for great flexibility when designing the CE. Initially, the relatively few features of the CE also made this interface the natural branching points for the various CEs. However, with increased levels of functionality and complexity, this may no longer be true. Large fractions of the CE code base is now shared between the various CEs. In particular, framework for service and command handling, state machine and hardware access. Creating a new species of the CE is now done by implementing new derived classes from the CE framework base classes.

The CE framework is currently working well, and is in a stable state. However, the class hierarchy and structure still carries some heritage from more simplistic past. Although it supports the current use cases well, there are features foreseen for the future that may be difficult to implement using the current framework. A refactorisation of the class structure will give a more flexible and configurable framework that will support future extensions better. For example, a fully configurable state machine will make it easier to deploy custom states and behaviour to the individual CEs. In particular, this will be very useful for implementing automatic reconfiguration of FECs, as this requires

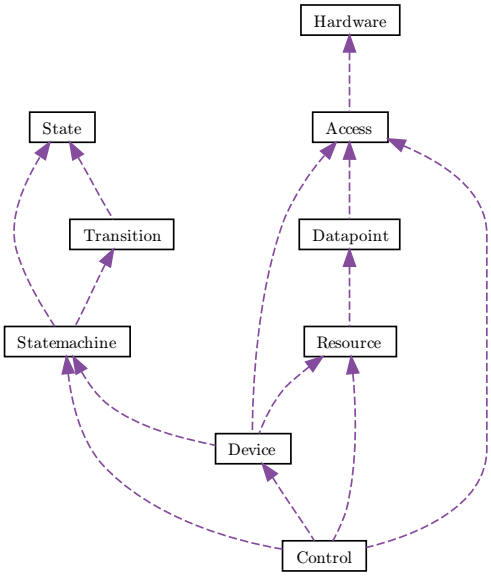

Figure 5.1: Simplified collaboration diagram for a possible refactorised FeeServer CE.

more advanced state handling.

The main task of refactorising the framework is not the coding involved. Rather, it is the testing and verification associated with asserting the CEs converted to the refactorised framework will behave identical to the old. Most importantly this applies to the complex behaviour of states and configuring.

Five main classes (*Access*, *Resource*, *Statemachine*, *Device*, *Control*) and a few "helper" classes (*Hardware*, *Datapoint*, *State*, *Transition*) are foreseen, as show in Figure 5.1.

5.1 Access class

A wide range of FEE hardware may be accessed by the FeeServer. This may be buses, interfaces, registers, memory, devices, files, *etc.* The fundamental purpose of the *Access* class is to provide a uniform interface for writing and reading, *accessing*, these different types of hardware. Each type of hardware will correspond to a *mode* of operation to the *Access* class. Typically, a *mode* will utilise a bus or interface directly accessible from the DCS board, although more complex scenarios exist. It is also possible to make a *mode* for accessing specific files, for example containing firmware to be loaded. In some cases the access to a bus or interface has first to go through another bus. This class will provide access to physical hardware by creating instances of classes representing them. Since there is only one unique instance of the physical hardware or file, it follows there should also only be one instance of the classes representing them; *i.e. singletons*. The i*Access* class will enforce this by being a *singleton* itself, with the instances of hardware

as static members.

Functions available from the *Access* class:

- *Access::Access()* — private constructor;
- *static signed int Access::GetInstance(Access* access)* — return only instance;
- *signed int Access::GetModeInstance(Hardware* hardware, const unsigned int mode=1)* — get pointer to singleton for specific
- *signed int Access::Read(const unsigned int address, unsigned int* data, const unsigned int words=1, const unsigned int mode=1)*;
- *signed int Access::Write(const unsigned int address, const unsigned int* data, const unsigned int words=1, const unsigned int mode=1).*

Parameter listing:

- *address* — to access;
- *data* — pointer to data buffer;
- *words=1* — number of words to read or write;
- *mode=1* — the access mode;

Since the class is a singleton, the constructor is private. The only instance can be obtained through the *GetInstance()*-function. The *words* and *mode* parameters are both assumed to be *1*; *i.e.* accessing *1* word from the message buffer, which both should by far be the most frequently used parameters. In general, a positive return value indicates success; a negative failure. Additional information may or may not be encoded in the return value.

A pointer to the singleton objects representing a specific *mode* can be accessed via the *GetModeInstance(...)*. This useful when the are functions available for that type of class, for example for setting the device file or sending IOCTL commands, for which there are no corresponding function available in the *Access* class.

The *Read(...)* and *Write(...)* functions operates with a native word size of 32 bits. However, the exact behaviour may depend slightly on the *mode*. For physical hardware, the *address* may refer to a physical register, *per* the addressing scheme employed by the particular hardware in question. The actual number of bits read or written may be less than 32 bits, determined by the width supported at hardware level. For *Read(...)*, the words will be zero-padded up to 32 bits; for *Write(...)*, superfluous bits will be discarded. The typical use for this is where a particular *address* points to a hardware register with a specific meaning, not just somewhere in a continuous data storage.

For devices that may be considered "true continuous storage", the data should be organised as 32 bit words, regardless of the underlying width. In particular, this applies to files, memory and firmware storage devices. In practical use, this distinction should be rather "natural", and not be a source of confusion.

The *Access* class should also provide external *read* and *write* commands through the *Resource* class, based on the above functions, that can be used by ICL to access

the hardware directly. The two first four-byte words of the payload should be used to specify the *address* and the number of *words*, the *mode* should be passed as the command parameter.

Hardware class

As mentioned above, the *Access* class uses classes to represent the *hardware* which is being accessed. This will be the *Hardware* class. It is intended as a base class, with multiple sub-classes, each representing a particular piece of hardware. So far the following classes, with corresponding *mode*s, are defined:

0. *HardwareDcs* — DCS bus;
1. *HardwareMessagebuffer* — message buffer via DCS bus;
2. *HardwareSelectmap* — select map via DCS bus;
3. *HardwareFlash* — flash via DCS bus;
4. *HardwareTtcrx* — TTCrx chip;
5. *HardwareFeci2c* — FECs via I^2C bus;
6. *HardwareFecgtl* — FECs via GTL bus;
- *HardwareFile* — "dummy" file;
- *HardwareMemory* — "dummy" memory.

The *HardwareDcs* gives access to the RCU or similar appliances. The exact utilisation of the bus depends on the firmware of the DCS board and the appliance connected to it. For the RCU, it can be in one of three different modes at any time, giving access to either the message buffer, select map or flash of the RCU. The mode is set using IOCTL commands to the device driver. This class is used as a basic access class; the access to the specific modes should be done in separate classes utilising this class. As far as this class is concerned, the current mode is unknown, hence it is not meaningful to use it directly, except in very low-level application. Rather, the specific classes will take the appropriate steps of asserting the bus is in the correct mode before use.

The *HardwareMessagebuffer* is used to communicate with the RCU firmware. Configuration of the FEE is done by writing to message buffer registers. The *HardwareSelectmap* gives access to the firmware of the main FPGA of the RCU. The main use case is programming the RCU, but also read-back for verification can be done. The flash of the RCU is used for storing the firmware externally from the FPGA. The *HardwareFlash* allows manipulation of the firmware contents of the flash similar to how the *HardwareSelectmap* can manipulate the firmware contents of the select map. The TTCrx is the trigger chip of the DCS board. Configuration of it will be possible via *HardwareTtcrx*. The RCU FECs can be accessed either via the *HardwareFeci2c* or the *HardwareFecgtl* buses. In addition, defining *mode*s for accessing the DCS board firmware is possible, but can be unwise, as frivolous write is very likely to render the DCS board unusable,

requiring reprogramming via JTAG. This should be considered very carefully if it is to be implemented.

The *HardwareFile* and *HardwareMemory* are slightly different than the other classes. Their main purpose is to be *general* classes that can *substituted* for the *real* classes for the purpose of testing and debugging. This allows the data intended to be read and written by, say the I^2C bus class to be written to a file or memory instead. The former is useful in case of debugging where it is useful to be able to inspect the contents of the file contain the full transaction, the later for testing when real hardware is not available, for example a DCS board without FECs or a PC without any RCU at all. When using these classes, care should be taken not to exhaust the file system or memory space available on the system, as the potential memory span can be huge. For the file, the addressing should correspond directly to that of the file, for the memory, a *std::map* should be used to save memory. The *HardwareFile* class may also be used for developing further *mode*s, for example for loading firmware from files, or editing configuration files via commands.

Hardware classes representing physical hardware should only have one instance, *i.e.* being singletons. Specifically, this is the *HardwareDcs*, *HardwareMessagebuffer*, *HardwareSelectmap*, *HardwareFlash*, *HardwareFeci2c* and the *HardwareFecgtl*. On the other hand, *HardwareFile* and *HardwareMemory* may exist in several instances pointing to different files and memory areas, one for each class they substitute.

The common public functions for the *Hardware* class and its sub-classes are analogous to those of the *Access* class, of course without the *mode* parameter, as it indicates to which instance of a *Hardware* class the call should be mapped:

- *virtual signed int Access::Hardware::Read(const unsigned int address, unsigned int* data, const unsigned int words=1)=0*;
- *virtual signed int Access::Hardware::Write(const unsigned int address, const unsigned int* data, const unsigned int words=1)=0*;

As seen in the listing above, the *Hardware* class is defined inside the *Access* namespace, as they are considered a property of, and only to be used by, the *Access* class. It is possible to define further functions. These will of course not be directly available through the *Access* class, but users with very specialised needs may access such functions through the *Access::GetModeInstance(...)* function. Before using an extended function, tests should be performed to assure the object in fact is of the expected type, and not substituted for a file or memory class.

5.2 Resource class

One of the main functionalities of the CE is to manage the FEE *resources*, hence the *Resource* class. The class organise the resources as a set of *data-points*; a generalised, high-level view of the resources available. Typically it will be a hardware register, or

the result of more complex logic. A data-point may be *published* as a *service* via a separate DIM channel, *issued* as a *command* via the command DIM channel, or both. Both method will allow values of a data-point to be read and written.

The class should be a singleton with private constructor, but two functions for registering data-points:

- *private Resource::Resource()* — private constructor;
- *static signed int Resource::RegisterDatapoint(const std::string name, const unsigned int type, const unsigned int id, const unsigned int address, const unsigned int mode=1, const double deadband=0, const double conversion=1, const signed int publish=true, const signed int service=true)* — "simple" data-point;
- *static signed int Resource::RegisterDatapoint(const std::string name, const unsigned int type, const unsigned int id, ResourceCallback* callback, const unsigned int service=true, const unsigned int command=true)* — "complex" data-point.

Parameter listing:

- *name* — of data-point;
- *type* — specifies the data type of the data-point, for *publish*, *0*, *1* or *2* for *signed int*, *float* or *char**, respectively;
- *address* — of register;
- *id* — for command handler;
- *mode=1* — *mode* for accessing the register, as used by *Access* class;
- *deadband=0* — how much new value may deviate from previously published value before an update is deemed necessary;
- *factor=1* — conversion factor between register value and data-point value;
- *service=true* — make service channel;
- *command=true* — make command handler;
- *callback* — pointer to external function to call when service value is to read or written, or command is received.

There are two types of services: for publishing the numeric value of single registers; and for publishing complex values such as *char*, or numeric values that require computation beyond a simple multiplication factor. The first type of service can be handled by the *Resource* class simply by specifying the details of the register to publish. For the later, a call-back function has to be specified.

For the call-back function pointer, this type should be defined:

- *typedef signed int (*ResourceCallback)(unsigned int id, unsigned int parameter, void* data, unsigned int size, std::vector<unsigned int>* result, unsigned int* processed).*

Callback function parameters are listed below:

- *parameter* — arbitrary information as specified by command;
- *data* — payload, to be processed;

- *size* — of *data*, 32-bit words;
- *result* — *std::vector* where eventual result may be appended to the end;
- *processed* — number of 32-bit words of *data* processed.

The call-back function will return a positive value on success, and a negative value for failure; optionally, extra information may be encoded. In case of failure, the *processed* parameter should be filled with the number of words the command is expected to process, or if this can not be determined, a lower limit for the number of words processed. This number may be used as input for attempting to recover processing the remaining data. It is entirely up to the implementation of the call-back function to determine how to treat the incoming parameters.

Since the simple data-points can only consist of a single 32-bit value, the framework can automatically make command service channels and command handlers. For the services, both high- and binary commands should be provided. Both types of commands should take the name of the data-point. The value of the data-point may be both gotten and set using the same command. For a low-level command received with a *parameter* of *zero*, the data-point is set to the value of the first 32-bit word of the payload, whose length is also assumed to be *1*. If the *parameter* is *non-zero*, the current value is returned. No changes are made to the register. For a high-level command with an empty payload, the current value is returned. If a value is found in the payload, the register is set to this value. Nothing is returned. This slightly inverse approach for the two types of commands is taken to assure backwards-compatibility with the *set* commands of existing CEs, where there in general would be separate commands for getting and setting registers.

Datapoint class

The list of *data-points* maintained by the *Resource* class contains objects of the *Datapoint* class. There are two sub-classes: one for the case of simple data-points, and another one for the complex case involving call-back functions. All involved parameters are stored in the respective classes.

Public functions:

- *Resource::DatapointAuto::DatapointAuto(const std::string name, const unsigned int type, const unsigned int address, const unsigned int mode=1, const double deadband=0, const double conversion=1, const signed int publish=true, const signed service=true)* — "simple" *Datapoint*;
- *Resource::DatapointManual::DatapointManual(const std::string name, const unsigned int type, ResourceCallback* callback, const signed int service=true, const signed int command=true)* — "complex" *Datapoint*;
- *virtual signed int Resource::Datapoint::Call(unsigned int id, unsigned int parameter, void* data, unsigned int size, std::vector<unsigned int>* result, unsigned int**

processed)=0 — *call* a *Datapoint*.

The class is defined inside the name-space of the *Resource* class, as it is not supposed to be used outside the scope of this class. Also, the parameters are the same as for the *RegisterDatapoint(...)*, without the *id* parameter, as this is handled by the *Resource* class as key to a *std::map* containing all *Datapoint* objects. The two first functions are the constructors for the two sub-classes. After that follows the function for passing a *call* to a data-point: *Call(...)*; of course over-ridden by the two sub-classes. When a *binary command* is received, the framework should pass this to the *Call(...)* of the *Datapoint* object with an *id* matching that of the command. The parameters as given by the command. When a *high-level command* is received, the framework should pass this too the *Call(...)* of the *Datapoint* object with a *name* matching that of the command. The framework should find the corresponding *id* of the command, *parameter* should be empty, *data* should be the parameter passed to the command, with a *size* of *one*. When a *service get* is received, the framework should pass this to the *Call(...)* of the *Datapoint* object with an *id* matching that of the service. The framework should find the corresponding *id* of the service, *parameter* should be empty, *data* should be empty, with a *size* of *null*. The returned value is expected to be the first word of *result*. When a *service set* is received, the framework should pass this to the *Call(...)* of the *Datapoint* object with an *id* matching that of the service. The framework should find the corresponding *id* of the service, *parameter* should be empty, *data* should be the parameter passed to the service, with a *size* of *one*.

This should be the case for both simple and complex *Datapoint*s. There will be complex *Datapoint*s which do not make sense to publish as services, for example writing the RCU IM. Such *Datapoint*s should be created with *service=false*. If a data-point will not be monitored by ICL, it should also be created with *service=false* as not to waste the resources of the *DCS* board on updating unnecessary services. If for some reason it is not desirable to have a command handler for a given *Datapoint*, it should be constructed with *command=false*.

For the simple *Datapoint*s, a complete framework for publishing services and handling commands should be provided. Typically, a simple *Datapoint* will be limited to hardware registers, however, it is expected that the majority of *Datapoint*s will fall into this category.

For complex *Datapoint*, the user has to provide a call-back function. Although every complex *Datapoint* has to be registered separately, the same call-back function may be specified to handle more than one *Datapoint* by implementing an internal *id* switch in the call-back function.

5.3 State machine class

A *state machine* is based on the idea that an *entity* must at any time be in *one* of a finite set of states. The *Statemachine* class should provide a framework for this.

Statemachine functions:

- *Statemachine::Statemachine(const std::string name)* — constructor;
- *signed int Statemachine::AddState(const std::string stateName, const unsigned int stateId, StatemachineCallback* enterCallback=NULL, StatemachineCallback* leaveCallback=NULL)* — add state;
- *signed int Statemachine:.AddTransition(const std::string transitionName, const unsigned int transitionId, const unsigned int transitionStateId, const unsigned int endStateId)* — add transition;
- *signed int Statemachine::AddTransitionStart(const unsigned int transitionId, const unsigned int startTransitionId)* — add start state of a transition;
- *signed int Statemachine::GetStateId(unsigned int* stateId)* — get the ID of current state;
- *signed int Statemachine::GetStateName(std::string* stateName)* — get the name of current state.

Parameter listing:

- *name* — of state machine, to be used as name of the state service channels;
- *stateName* — name of state, will be displayed state name service channel;
- *stateId* — ID of state, used internally and displayed in the state ID service channel;
- *enterCallback=NULL*, *leaveCallback=NULL*, — call-back functions called when a transition *leaves* a state and *enters* another, if *NULL* not called;
- *transitionName* — name of transition, for external transition triggering;
- *transitionId* — ID of transition, used internally;
- *transitionStateId* — ID of state used during transition;
- *endStateId* — ID of end state;
- *startStateId* — ID of start state.

For an entity wishing to create a state machine, the first step should be to create an object of type *Statemachine*. Henceforth, the desired states should be created individually using the *AddState(...)* function, then the transitions through the *AddTransition(...)* function. Since a transition may have several starting stated, they may be specified with the *AddTransitionStart(...)* function.

Once the transition starts, the state should be changed to the *transitionStateId*, and the corresponding *enterCallback* called. Then the *leaveCallback* of *startStateId*, then *enterCallback* of *endState*, then *leaveCallback* of *transitionStateId*. Finally, the state should be changed to *endStateId*.

For the call-back function pointer, this type should be defined:

- *typedef signed int (*StatemachineCallback)(const unsigned int startStateId, const unsigned int endStateId, const unsigned int transitionId).*

The framework should call the call-back functions with parameters indicating which state is left, entered, and the transaction. This will make it possible to make tailor the logic to the specific case.

The *Statemachine* class should use the *Resource* class to automatically register two service channels for the state machine: one channel bearing the *name* of the *Statemachine*, publishing the numeric *stateId*; and a second channel with the same name appended *_NAME*, publishing the *stateName* of the *StateMachine*.

It will be the responsibility of the entity creating the *StateMachine* to decide how to utilise it, and what can trigger a transition. Some instances of *Statemachine* may only mirror the state of actual hardware, and only have a few states, like *ON*, *OFF*, *RAMPING_UP* and *RAMPING_DOWN*. Other instances, like for the *Control* class, the state may be determined from a wide range of inputs, like the state of other *Statemachine*s, various error conditions during configuration, the state of verification, and external state transitions. The variations are too wide and complex to make a "one-size-fits-all"-interface.

State class

The *State* class is used by the *Statemachine class* to store the states. Each state should correspond to one instance of the *State* class.

Functions:

- *Statemachine::State::State(const std::string stateName, StatemachineCallback* enterCallback=NULL, StatemachineCallback* leaveCallback=NULL)* — constructor;
- *signed int Statemachine::State::GetStateName(std::string* stateName)* — return name of state.

The parameters mostly match those of *Statemachine::AddState(. . .)*, except for *stateId*, which should be used by *Statemachine* as a key to a *std::map* containing the *State*s.

Transition class

An instance of the *Transition* class is used by the *Statemachine* to define a single transition. The *Transition* object will maintain a list of states from which the transition may start.

Functions:

- *Statemachine::Transition::Transition(const std::string transitionName, const unsigned int transitionStateId, const unsigned int endStateId)* — constructor;
- *signed int Statemachine::Transition::AddStart(const unsigned int startTransitionId)* — add ID of start state;

- *signed int Statemachine::Transition::GetTransitionName(std::string* transitionName)* — return name of transition.

Also in this case, the *transitionId* is used as a key by the *Statemachine* class to a *std::map* containing all defined states. Otherwise, the parameters are as in *Statemachine::AddTransition(...)*.

5.4 Device class

A *device* can be any *physical* or *logical* entity it is considered meaningful to distinguish. For this purpose, the *Device* class should be provided. Typically, this may be a firmware module, a chip, a bus, a circuit board, or even a complete system. It should be possible to have a layered hierarchy of devices: one *Device* may represent the complete system, which again "owns" some circuits boards, which again has chips with firmware modules.

A rather wide flexibility is foreseen in the framework; a *Device* may pick only the parts needed. If needed, a *Statemachine* can be included. The *Resource* class can provide easy creation of services and command handlers, while the *Access* class will give physical access to the hardware.

Since a *Device* can represent a wide variety of entities, the interface should be flexible. The *Device* class is intended as a pure virtual base class, from which the actual *Devices* are derived.

The following pure virtual functions so far for seen for the base class:

- *virtual signed int Device::IsOk()=0* — check whether *Device* is functioning properly;
- *virtual signed int Device::Recover()=0* — try to recover if not functioning properly;
- *virtual signed int Device::DispatchTransitionId(const unsigned int transitionId)=0* — trigger transition;
- *virtual signed int Device::GetStateId(unsigned int* stateId)=0* — get ID of current state.

The *IsOk()* function is intended as high-level "good" or "not good" "state" for the framework, without having to know the details of the *Device*. In case the *Device* is not "not good", the *Recover()* function should implement a procedure to try ro recover to "good". A transition can be triggered externally using the *DispatchTransitionId(...)*. The *Device* may ignore the transition, or map it to some other transition if it is not meaningful for the *Device*. Preferably, the owner of the *Device* should have sufficient knowledge of the low-level workings of it to not try to trigger a transition it does not support. The *GetStateId(...)* can be used to obtain the current state of the *device*. If needed, it may be mapped to another state.

5.5 Control class

The *Control* class should be initiated from the FeeServer Core, and be the *main* class, the owner of all other objects. It carries certain similarities with the *Device* class in the way it owns other *Devices* and utilises the *Statemachine*, *Resource* and *Access* classes. However, the *Control* class is for *controlling* the overall system.

The FeeServer of different appliances will have different CEs. Different CEs can be made by deriving different sub-classes of the *Control* class. Since the FeeServer can have the identity of a given appliance, it follows there can only be one instance of the *Control* class, making it a singleton.

The base class should implementing the interface to the Core. For commands, it should forward them from the Core framework to the *Resource* class. For services, it should facilitate publishing from the *Resource* class to the Core. It will create a *Statemachine* for the overall *MAIN* state.

The handling of *FEE_CONFIGURE* commands should be done by the *Control* class (but through the *Resource* class).

The derived classes should create *Devices* and *Resources* according to their needs.

5.6 Outlook

The framework described above shows how the existing framework can be further generalised as a platform for supporting FeeServers for multiple appliances. Although only minor parts of it have been implemented so far, it can serve as a long-term guide whenever changes have to be made to the CE.

In addition, there are plans for implementing verification of the ALTRO registers during the empty slots of the data read-out orbit on the GTL bus. Single-event upsets are caused by radiation changing the contents of ALTRO and the RCU FPGA *Static RAM* (SRAM) registers. This can introduce logic errors in the firmware, or changing the configuration parameters. For the FECs such verification can be achieved by monitoring the trigger messages received by the DCS board to determine when the read-out orbit gaps will be. Whenever the bus is not busy with data read-out, the FeeServer can read back parts of the configuration data from the ALTRO registers, and compare it to the configuration stored in its memory. If an error is detected, the register will be automatically reconfigured. This scheme can easily be extended to automatic full reconfiguration of FECs which have been turned off either by the RCU after a hard error, or after a manual shutdown.

Chapter 6

Calibration overview

Before any new detector can be expected to fulfil its design goals, it has to be *calibrated*. Calibration is the process of converting the "raw" signals from the detector into meaningful physical measurements with units and well-defined error-bars.

When the detector is designed using *Computer-Assisted Drawing* (CAD), everything is "perfect". All parts have exact dimensions, they fit perfectly together, there is neither twisting nor bending, everything is perfectly aligned to each other. All fields are known, all subsystems are working as designed, there is no noise, the material budgets are perfect. The length of the detector does not change with magnetic fields. One might even assume external conditions, like temperature and pressure, are not changing with time. And so on. This is the *ideal* detector. When we implement a *physical* detector based on this idea, it will not be as perfect. This is well known, and is indeed taken into account during the design process. The *Technical Design Report* (TDR) of a detector gives estimates and limits for how far from the ideal detector the physical detector may be.

Since it is not possible to build an ideal detector, the imperfections of the physical one has to be *corrected*. Figure 6.1 shows the process from raw data, via corrections and calibration, to reconstructed data. When correcting a detector, various tools and

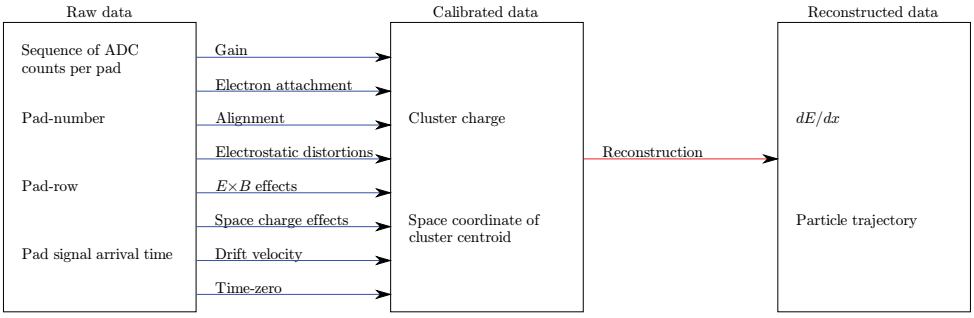

Figure 6.1: Simplified process from raw to reconstructed data for the TPC.

methods are used to determine how the imperfections of the detector manifest themselves in the data it produces, and what kinds of *inverse* transforms have to be applied to the data to make them appear like they were collected by a perfect detector.

Some of the deviations of the physical detector from the ideal one may be so small it is not worthwhile to correct; they may "drown" in other effects, or they are not very critical to the resolution of parameters of interest. Some may be impossible, or at least too difficult to correct.

In general, the corrections are done *after* the calibration to account for the distortions. However, the picture can be more blurred. For example, the drift velocity variations with time of the TPC may be considered both a calibration and a correction effect. In the first case, it can be argued that the time-dependence of the drift velocity variation is an intrinsic "feature" of a TPC, thus is part of the calibration. On the other hand, as in the later case, it can be argued that for the ideal detector, there should be no variations of the parameters influencing the drift velocity. Hence, the drift velocity variations should be corrected. Often, practical consideration can heavily influence whether a given effect will be considered as a part of the calibration or the correction for computing purposes.

There are two main categories of calibration:

- *static* — constant with time;
- *dynamic* — changing with time.

. It is possible to argue that all effects are changing with time over sufficiently long time intervals. For example, alignment may be static for years, but once the sub-detectors have been moved, it is changed. However, the distinction is important since it discriminates between effects where a fixed transformation can be used, and effects where the transformation has to be recalculated in given time intervals.

A general challenge with calibration is to determine the order to apply the corrections of different effects, *i.e.*, to *factorise* the calibration tasks. Very often, more than one effect will influence the resolution of a given parameter. Likewise, one effect might influence more than one parameter. Sometimes a specific order might be better for a given parameter, but another order is preferable for some other parameter. Generally, it is desirable to apply the corrections in the order that gives the best overall result.

Sometimes cyclic dependencies are encountered. An illustrating, simplified scenario can be the case where the resolution of parameter A is influenced by correction for effect I, which depends on parameter B, which is influenced by correction for effect II, which again depends on parameter A. The way out is to insert uncorrected values for some parameters in this circle. Care must be taken to choose the parameter so as to minimise the overall effect. Most of the time this can be determined from physics arguments, though considerable insight in the matter is needed. Since the order of the corrections can change the net values of some parameters, hence the resolution of the physics results, care must be taken if changing the order.

The TPC has a laser system for calibration of *Read-Out Chamber* (ROC) alignment, electric field distortions, $\boldsymbol{E} \times \boldsymbol{B}$ effect, gain and drift velocity. A $Nd : YAG$ 266 nm laser is used to generate four planes parallel to the beam axis, using 168 tracks on each side of the CE. The CE will emit photo electrons from scattered laser light. After a characteristic drift time, the read-out pads will receive this signal. Hence drift velocity and gain can be calculated.

The drift velocity depends on a number of parameters, most importantly variations in gas pressure and temperature, as well as slow changes in the gas composition. A number of sources are available for determination of drift velocity: matching tracks passing through the CE, both from cosmic events and beam collisions; laser events; matching TPC–ITS tracks; and a dedicated drift velocity monitor. These approaches may be combined to improve accuracy. For high-statistics methods, the obtained drift velocity value may be used directly for correction. In other cases it will be necessary to correct for changes in gas pressure and temperature at an event level. However, in all cases correction for top-bottom arrival time offsets, caused by pressure and temperature gradients in the TPC, is needed.

6.1 The electron drift vector

It is possible to derive a Langevin equation for the drift vector of the electron in a magnetic and electric field, as in Equation 6.1 [76], in which $\boldsymbol{v}_{Drift}$ is the *drift velocity vector*, μ is the *electron mobility*, related to the drift velocity as $\mu = e\tau/m_e = \boldsymbol{v}_{Drift}/|\boldsymbol{E}|$. The *electrical field* and *magnetic field* are $\boldsymbol{E}$ and $\boldsymbol{B}$, respectively. The cyclotron frequency is defined as $\omega = e|\boldsymbol{B}|/m_e$ and τ is the average time between collisions.

$$\boldsymbol{v}_{Drift} = \frac{\mu|\boldsymbol{E}|}{1+\omega^2\tau^2}(\hat{\boldsymbol{E}} + \omega\tau(\hat{\boldsymbol{E}} \times \hat{\boldsymbol{B}}) + \omega^2\tau^2(\hat{\boldsymbol{E}} \cdot \hat{\boldsymbol{B}})\hat{\boldsymbol{B}}) \tag{6.1}$$

Electrons in the TPC will experience both an electric field $\boldsymbol{E}$ and a magnetic field $\boldsymbol{B}$. $\boldsymbol{E}$ originates from the TPC itself, which is set up between the CE and the end planes. Like any electric field, it will exercise a force on the electron in the direction of the field. $\boldsymbol{E}$ is needed as an integral part of the TPC design; electrons from the ionised counting gas along the tracks of the produced particles have to drift towards the MWPCs of the end planes for detection.

$\boldsymbol{B}$ originates from the L3 magnet, in which the TPC is installed. Except for minor deviations, it is parallel to $\boldsymbol{E}$. A $\boldsymbol{B}$ field will exercise a force normal to both the direction of movement of the electron — resulting in a circular movement.

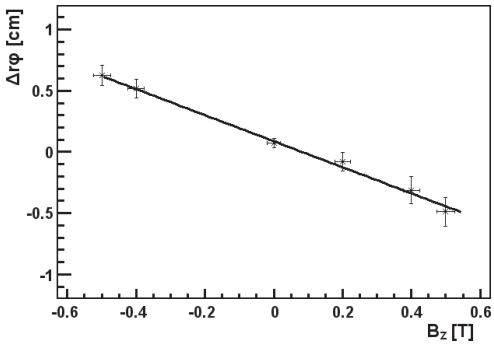

Figure 6.2: $\boldsymbol{E} \times \boldsymbol{B}$ correction as function of field strength [78]. An almost linear dependence is observed.

6.2 Effects influencing the electron drift

Correction of the electron drift effects is essential to obtain accurate position measurements with the TPC.

6.2.1 Mechanical distortions

The mechanical distortions can stem from both minor manufacturing imperfections and the mounting of the TPC in the experimental area. Small offsets in the location of the mounting points may introduce twisting or other forms for distortions to the read-out chambers. Such distortions are constant until the TPC is physically moved.

6.2.2 Electrostatic distortions

The electrostatic electron drift field between the CE and the end planes is homogenised by resistor rods along the z-axis. Despite the resistor rods, it is not possible to create a completely distortion-free drift chamber, *e.g.* from the finite widths of the equipotential strips, errors in resistor chain values, ordering of the resistors, shorted strips, mismatch of the field cage cylinder ground end voltage with the pad plane, deformations of the pad plane, *etc.* [77]. The minor inhomogeneities of the field will influence the drift of the electrons. The mechanical distortions of the TPC, discussed above, will deform the drift chambers, which in turn can also contribute to a inhomogeneous drift chamber.

6.2.3 $\boldsymbol{E} \times \boldsymbol{B}$

An electron in presence of $\boldsymbol{E}$ and $\boldsymbol{B}$ fields, will be exposed to a force in the direction of $\boldsymbol{E} \times \boldsymbol{B}$ causing it to drift in a helix movement along $\boldsymbol{E}$. However, for the special case where $\boldsymbol{E}$ and $\boldsymbol{B}$ are perfectly parallel, the radius of the helix movement will be *zero*, and the electrons drift is reduced to a movement only along $\boldsymbol{E}$. Unfortunately, $\boldsymbol{E}$ and $\boldsymbol{B}$ are

not perfectly parallel in the TPC due to inhomogeneities in both fields. For the volume occupied by the TPC, the deviations are approximately 1%. The displacement of the track caused by the angle between $\boldsymbol{E}$ and $\boldsymbol{B}$ is proportional to $z\omega\tau$, where z is the drift distance of the electrons, ω is proportional to $\boldsymbol{B}$, and τ is a characteristic time constant of the counting gas. For the full TPC drift length of 250 cm, the track deviation is the order of one cm. Considering that the spatial resolution of the TPC is up to 300 μm, the $\boldsymbol{E} \times \boldsymbol{B}$ effect has to be corrected.

$\boldsymbol{E} \times \boldsymbol{B}$ correction is performed using laser tracks. The TPC is equipped with a laser system generating 336 laser beams inside the TPC. The start and end points of the lasers are carefully surveyed. Hence, correction maps can be produced from the distortion measured by comparing reconstructed laser tracks to surveyed laser tracks. By repeating the measurements of the laser tracks for varying strengths of $\boldsymbol{B}$, a correction map as function of $\boldsymbol{B}$ is obtained. The correction map can be used to fit $\omega\tau$. Typical values are $\boldsymbol{B} = \{-0.5, -0.4, -0.2, 0, 0.2, 0.4, 0.5\}T$. The values of displacement at $\boldsymbol{B} = 0$ is subtracted to reduce the effect of misalignment.

Currently, the resolution of the correction map is approximately 350 μm. The correction data are generated by multiple laser events at varying field strengths, then measuring $\Delta r\varphi$ for each track. As shown in Figure 6.2 [78], $\Delta r\varphi < 0.7$ cm for longest drift in nominal field.

6.2.4 Gain

For calibration of the gain, radioactive krypton isotopes are mixed into the counting gas. The gain is very constant with respect to time; it is only necessary to re-calibrate after work on the electronics or the end-plates. The pad–pad gain variations are highly related to geometrical imperfections. The calibration is performed by injecting radioactive ^{83}Kr into the drift gas, then measure the decays at three different gain levels. The decay will be recorded by the TPC. Figure 6.3 is a typically obtained gain map. The achieved resolution is 4.2 % for OROC and 4.0 % for IROC.

6.2.5 Electron attachment

Oxygen present in the counting gas will capture drifting electrons. Practically speaking, it may be considered a *negative gain*, since it removes drifting electrons from the detector. Although oxygen is not a part of the counting gas mixture, there will be contaminations from the air, which is abundant with O_2. Oxygen removal is a constant process in the cleaning facility for the counting gas.

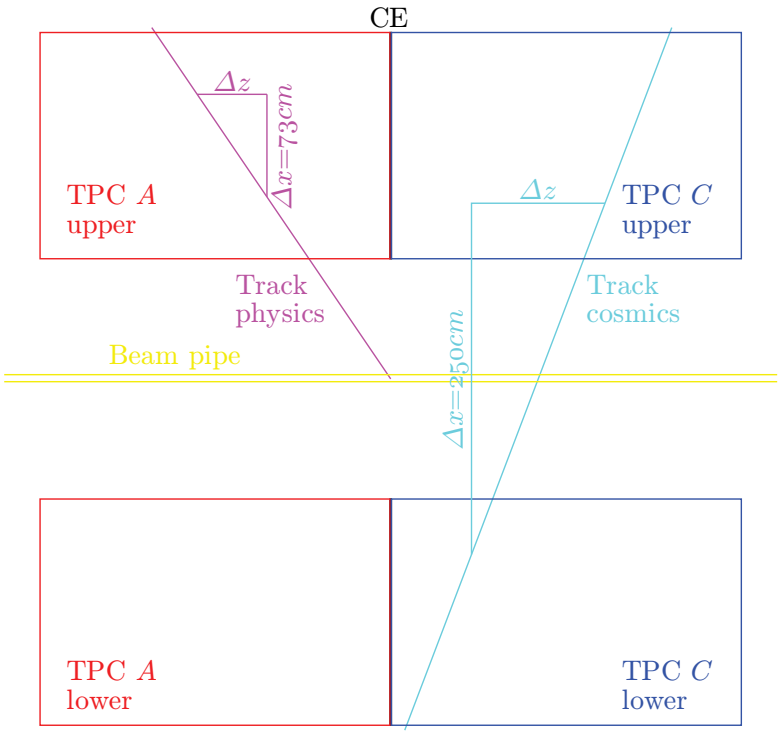

Figure 6.3: TPC electron attachment calculation principle. The signal is measured at two positions with a constant distance Δx (but different in the two cases of cosmics and physics tracks). The change in signal relative to the varying distance Δz is taken as a a result of electron attachment.

6.2.6 Space charge

Ions drifting towards the CE represent charge present in the drift regions, and will distort the drift path of the electrons [79]. The impact of the effect increases with the density of the charges. Such distortions may be hard to correct for. The gating grid is designed to prevent the ions from drifting back into the drift chambers, but at high collision rates, it will be open most of the time to accept the incoming electrons. Fortunately, the effect does not appear to be the limiting factor of the TPC; simulations show more than 10 000 interleaved events may be present in the TPC simultaneously, at which stage event disentangling is already becoming an issue.

6.2.7 Drift velocity

The drift velocity specifies how fast the electrons drift through the TPC. It can be affected by a number of influencing parameters, in particular temperature and pressure. The details will be treated in details in Chapter 8.

Chapter 7

TPC AliRoot calibration framework

7.1 Off-line classes

A dedicated framework for the calibration of the TPC exists within *AliRoot* [80]. Each calibration task is implemented in a separate class, derived from a common base class *AliTPCcalibTimeBase*. The base class defines the framework, and provides a common interface to the various calibration tasks. Currently, these functions are defined, and can be over-ridden by the derived calibration classes as needed [80]:

- *virtual void Process(AliESDEvent* event)* — whole *event*;
- *virtual void Process(AliTPCseed* track)* — only *tracks*;
- *virtual void Process(AliESDtrack* track, Int_t runNo=-1)*;
- *virtual Long64_t Merge(TCollection* li)* — merge sub-results from other instances;
- *virtual void Analyze()* — analyse raw data and extract fits;
- *virtual void Terminate()* — called before saving data;
- *virtual void UpdateEventInfo(AliESDEvent* event)* — update global variables;
- *virtual Bool_t AcceptTrigger()* — exclude events with inappropriate trigger mask;
- *virtual void SetTriggerMask(Int_t accept, Int_t reject, Bool_t rejectLaser)* — set masks of triggers to be accepted and rejected.

The *Process(...)*-functions are the entry-point for the calibration; the framework will *push* data to the calibration tasks using these functions. Since different tasks might need to be processed either at *event level* or *track level*, alternative interfaces are provided. *Process(...)* may be called multiple times for each instance of the calibration class; the results are accumulated in internal data structures. It is often desirable to process data in parallel, in multiple instances of the same class. In that case *Merge(...)* can *merge* the results stored in another instance into this instance. *Analyze()* can be used to request the data to be *analysed*. When all data has been processed, and the results are to be stored, *Terminate()* is called to allow the class to perform the necessary post-processing before terminating. The global variables of the calibration classes can be set via the *Up-*

dateEventInfo(...) function, which takes an *event* object as one of the parameters. The corresponding variables are copied from the object. Events are associated with *trigger*, indicating *what* triggered the event to be read from the sub-detectors. *AcceptTrigger()* determines whether to process events that do not fit a pre-set trigger *mask*. Finally, *SetTriggerMask(...)* can be used to set the mask of triggers to *accept*, *reject*, and whether to reject laser events regardless of mask.

A number of derived classes have been made so far, Figure 7.1 shows the inheritance hierarchy graphically:

- *AliTPCcalibAlign* — internal TPC chamber alignment;
- *AliTPCcalibCalib* — re-application of calibration at cluster level;
- *AliTPCcalibLaser* — drift velocity, unlinearities, $\boldsymbol{E} \times \boldsymbol{B}$ from laser tracks;
- *AliTPCcalibCosmic* — performance studies;
- *AliTPCcalibMaterial* — material calibration;
- *AliTPCcalibPID* — PID from dE/dx;
- *AliTPCcalibTime* — time-dependent drift velocity;
- *AliTPCcalibTimeGain* — time-dependent gain;
- *AliTPCcalibTracks* — cluster shape and error parametrisation;
- *AliTPCcalibTracksGain* — gain from tracks;
- *AliTPCcalibTrigger* — trigger calibration;
- *AliTPCcalibUnlinearity* — unlinear effects;
- *AliTPCcalibV0* — V0 calibration.

The two classes of importance for the drift velocity calibration [45] are *AliTPCcalibTime* and *AliTPCcalibLaser*. The framework will push events to *AliTPCcalibTime::Process(...)*. Here, a switch is implemented to distinguish events from *cosmics*, *beam* and *laser*, which are treated accordingly. The two former types will be handled by the drift velocity calibration class internally, while events of the later type will be pushed to *AliTPCcalibLaser::Process(...)*, and the drift velocity results read back.

AliTPCcalibCalib will re-apply the calibration at cluster level, and refit the tracks.

The calibration framework is invoked from the AliRoot analysis framework via the *AliTPCAnalysisTaskcalib* class, inheriting from the *AliAnalysisTask* analysis base class.

During development it might be useful to process multiple runs together to get a feeling of the fluctuations over a longer time span. In the production system, however, every run will typically be processed individually, since the calibration objects will be stored per run in the CDB.

7.2 Order of calibration

As mentioned in previous chapter, the order of the calibration is in general *not* arbitrary. For the ALICE TPC, the following order is implemented in *AliRoot* [80]:

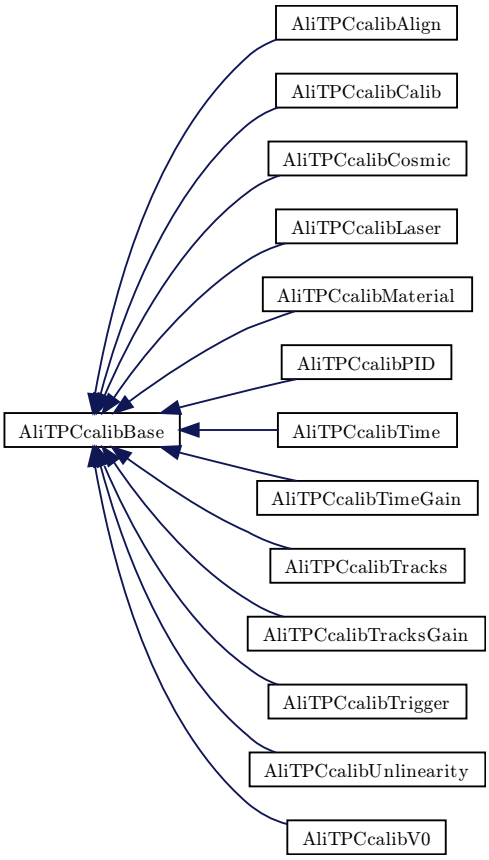

Figure 7.1: Inheritance diagram for TPC calibration classes. The AliTPCcalibBase is the base class for all calibration classes.

- time *zero* correction;
- transformation to local coordinate frame;
- drift velocity correction;
- transformation to global coordinate frame;
- $\boldsymbol{E} \times \boldsymbol{B}$ correction;
- time-of-flight correction;
- transformation to local tracking system;
- orthogonal alignment correction.

7.3 Condition database

The CDB is a database containing objects representing the *conditions* present during the data taking. This can be any time-dependent piece of information deemed relevant to the later calibration, reconstruction and analysis, collected from any of the sub-systems.

Typical data points are the detector configuration, *e.g.* magnetic field, gas temperature. From the CDB framework point-of-view, the objects are stored with a granularity of at most *one* run, or a range of *several* runs. However, the objects may contain more fine-grained samples. Often a default object with run range *zero* to *infinity* is provided for fall-back if more specific objects do not exist for certain runs. The objects are also *versioned*. This is in particular useful for calibration objects, since they are typically *refined* over several consecutive *passes*. If several versions are available, the latest version available for a specific run is chosen by default.

Drift velocity calibration object

For the TPC drift velocity calibration, the primary CDB calibration object is a *TObjectArray*. Currently, three *types* of objects are foreseen to be stored in the array: *THnSparseset<TARRAYF>*, *TGraph* and *AliSplineFit*. The *first* one is a four-dimensional sparse histogram of *time*, *run number*, *drift velocity correction* and environment *pressure–temperature* ratio. The *middle* one is a two-dimensional graph of *time* and *drift velocity correction*, from the histogram. This is the object to be used for the drift velocity correction during reconstruction. The *last* one is a two-dimensional spline fit from the graph. Initially, this was intended to be the object which is to be used by the reconstruction, however, since there is currently no *named* version of this object, it is for practical purposes not possible to include it in the array.

In addition, several statistical quantities may be stored: *mean*, *sigma* and *gain*.

Although the drift velocity corrections may be obtained from several *sources*, only the sources processed by the *AliTPCcalibTime* are stored in this array of calibration object; for example the values produced by *Goofie* and the CE detector algorithms are processed elsewhere, and stored in separate CDB objects.

There are a number of different *trigger* classes. Separate histograms and graphs are produced for the separate triggers, given sufficient statistics. Besides the conventional triggers as retrieved by the *GetFiredTriggerClasses()*, a "special" trigger, *all*, contains the data points from *all* triggers. This can be used in cases where there is too low statistics for a specific trigger class.

In most cases, the difference of drift velocity corrections are obtained from the two *sides* of the TPC with errors cancelled, therefore there is no need to distinguish between them. However, in the case of laser tracks, the precision is sufficiently high that it is possible to measure the difference.

The objects are retrieved from the array through their *names*; *ROOT* objects inheriting from *TNamed* base class have a *name* property. Unfortunately, *AliSplineFit* does not. A *naming convention* has been established for this. It consists of six fields, as shown in Table 7.1, separated by '_'. For simplicity, the names are all capitalised.

Type	*Quantity*	*Variable*	*Source*	*Trigger*	*Side*
THNSPARSESET<TARRAYF>	MEAN	VDRIFT	BEAM	ALL	
TGRAPH	SIGMA	GAIN	LASER	C0ASL-ABCE-NOPF-CENT	A
ALISPLINEFIT	ERROR	...	COSMICS	C0OB3-ABCE-NOPF-CENT	C
...	...	...	...	...	...

Table 7.1: Drift and gain calibration object naming convention.

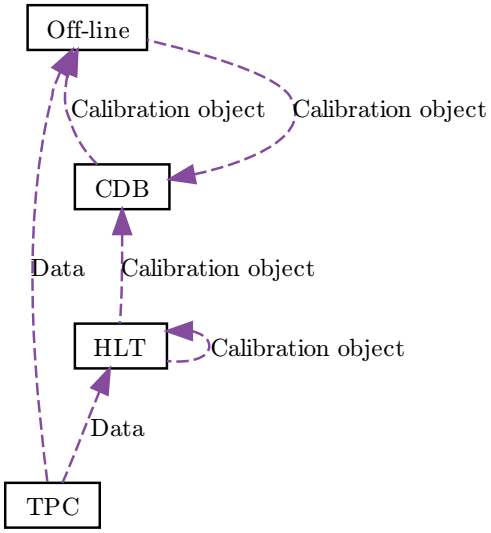

Figure 7.2: Collaboration diagram for TPC calibration objects. Both HLT and off-line can produce calibration objects to be stored in the CDB. HLT will use the object of the previous run for calibration of the current run.

Most of the fields have been outlined above. The *variable* field is to make the distinction between *gain* and *drift velocity* calibration objects clear, since they otherwise share the same naming conventions, though stored in separate CDB objects. The *trigger* field is the unmodified text string returned by the *GetFiredTriggerClasses()* function. It may contain several trigger classes. The last field, *side*, is optional.

7.4 HLT production of calibration objects

The calibration objects can be produced by *off-line* and HLT (*on-line*). Figure 7.2 shows the data flow. Both off-line and HLT have access to the data produced by the TPC, and can use this as input for the calculation of calibration objects. The objects are thereafter stored in the CDB, where they can be retrieved by off-line for reconstruction.

There are however, as will be shown, some differences in the calculations and what information is available to the on- and off-line systems. HLT, being an on-line system, can only make use of information — physics data and corresponding CDB objects —

which are available at the time the collisions take place. Off-line, on the other hand, can also take advantage of information collected later. For example, none of the methods for obtaining the drift velocity correction value can, for various reasons, provide an updated value more frequently than every halve an hour hour. Thus, HLT can only use the past drift velocity value to *extrapolate* the current drift velocity. Off-line, in contrast, can *interpolate* the past and future values to obtain the current value. Also, an on-line system typically trades accuracy for performance when reconstructing, further decreasing the precision to some extent.

However, the calibration objects provided by HLT are sufficiently accurate to be used as initial values for off-line. Off-line is expected to perform two reconstruction passes on the data, also producing refined calibration objects. Without the HLT calibration, an extra pass *zero* would have to be performed to create such initial values.

HLT is utilising the same calibration classes as off-line, making it easier to extend the HLT calibration to include further types of calibration.

Though the HLT and off-line calibration object share the same data format and are completely compatible, the procedure for writing them to the CDB is different, as they for practical reasons have different ways of accessing the CDB. While *off-line* may write them "directly", HLT has to go through the *shuttle*, which is responsible for collecting CDB objects from HLT and other sub-systems and "physically" store them in the CDB.

Chapter 8

Drift velocity calibration

Particles produced in the collisions traverse the TPC, leaving tracks of ionised gas and electrons along their paths. A strong electric field is set up along the z-axis, from the end planes to the CE, which causes the ions in the two volumes to drift to the CE, while the electrons drift towards the respective end planes for detection. For the full drift length of 250 cm, the nominal drift time is approximately 90 μs. However, there are variations, depending on counting gas composition, density, pressure, temperature, and electric-magnetic fields.

From a practical point of view, drift velocity variations correspond to a scaling of the TPC length along the z-axis. If the real drift velocity is higher than the velocity used for reconstruction, the TPC will appear "shorter" than it is, *i.e.* the track spacing will be truncated. And oppositely, a too low drift velocity will make the TPC appear longer, which leads to track spacing increasing. When plotted, Figure 8.1, a overly high drift velocity will cause the track segments on each side of the CE to overlap; the tracks of side A will extend into side C, and *vice versa*. Figure 8.2 shows the effect of a negative Δz on the track reconstruction, as seen in the on-line event monitor; the track segments overlap around the CE. On the other hand, a too low drift velocity will leave some volume on both sides of the CE on unpopulated with tracks. Figure 8.3 shows Δz variations as measured over a period of five weeks during the commissioning.

8.1 Influencing parameters

The drift velocity is a time-dependent function of a number of variable parameters:

- $\boldsymbol{B}$ — electric field;
- $\boldsymbol{E}$ — magnetic field;
- T — temperature;
- P — pressure;
- C_{CO_2} — concentration CO_2;
- C_{N_2} — concentration N_2.

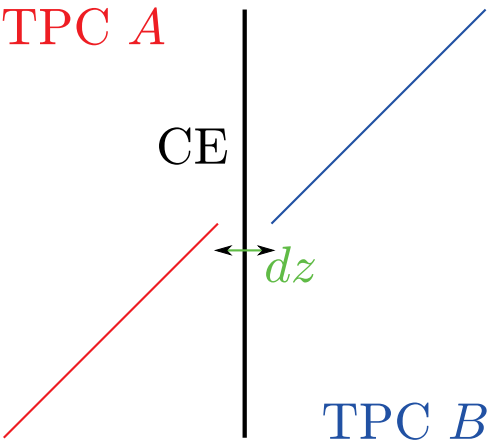

Figure 8.1: A positive Δz around the TPC CE, schematic view. The gap Δz between track segments is caused by incorrect drift velocity.

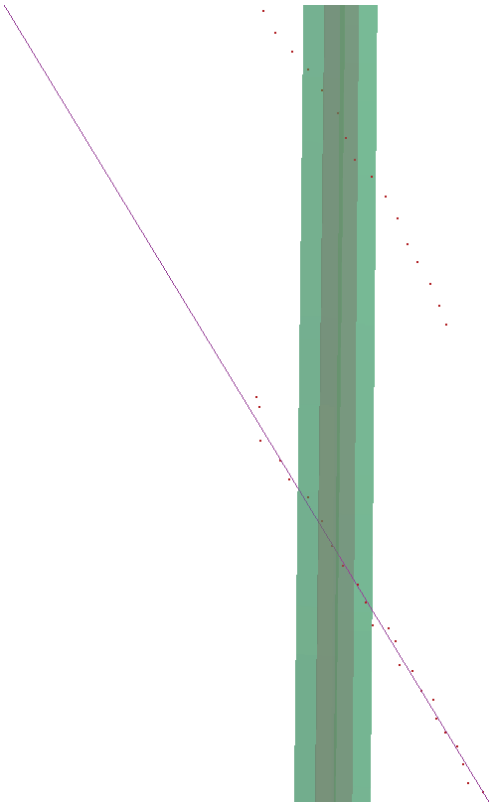

Figure 8.2: A negative Δz of approximately 5 cm (horizontal distance between two doted lines) around the TPC CE, as seen in the on-line event display. The negative value causes a overlap. The tracker fails to take the clusters on the left side into account, producing an extrapolated track into the left side based on only the clusters on the right side.

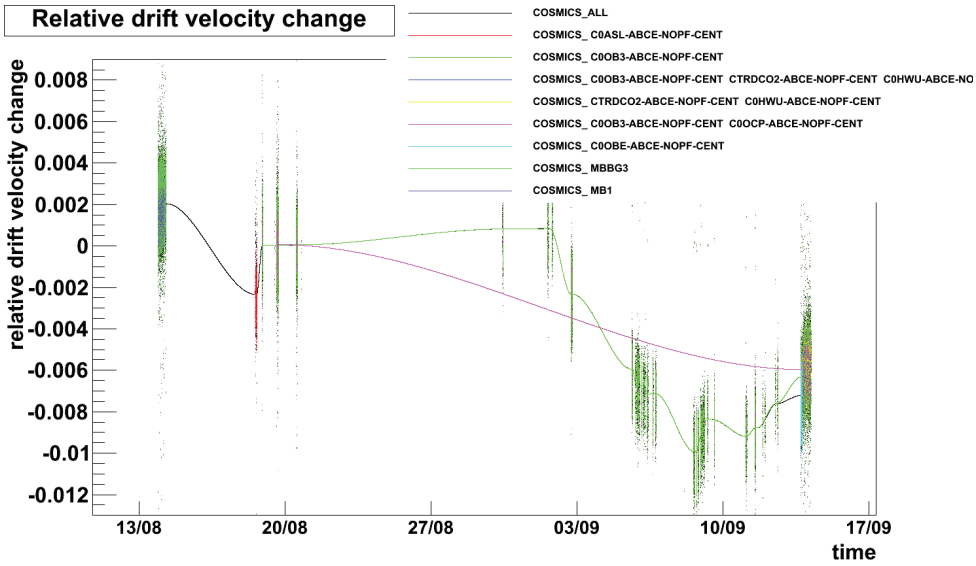

Figure 8.3: Drift velocity correction as function of time for the most frequent trigger classes. The *COSMICS_ALL* entry is produced by all classes combined. Both raw data and a spline-fit is shown. For sections without data-points, a fit is produced from the end-points.

Equation 8.1 [47] shows this:

$$v_d = v_d(\boldsymbol{E}, \boldsymbol{B}, N(P,T), C_{CO_2}, C_{N_2}) \tag{8.1}$$

The influence of electric and magnetic fields were discussed in Chapter 7, and can be obtained with high precision. For the drift velocity calibration, they are considered static. Rather, the primary concern are the comparatively fast change of counting gas temperature and pressure, and the slow change of counting gas composition. Gas pressure and temperature may change at a minute level. These parameters are known with high resolution from monitoring by dedicated sensors, and their theoretical influence on the drift velocity is known.

Since the TPC is not "pressurised", the pressure of the counting gas composition will follow the atmospheric pressure, hence the weather. The temperature in the experimental cavern where ALICE, including the TPC, is situated, is very stable once it has been closed for physics runs. However, heat produced by the detectors, especially the FEE, as well as the corresponding cooling systems, give rise to minor temperature gradients — both temporally and spatially. Although every effort has been made to minimise temperature variations, both by water cooling directly on the electronics enclosures and heat shielding towards the environment and other detectors, the 27 kW of heat generated by the FEE

is destined to make an impact on the environment. Also, the five-metre diameter of the TPC gives room for top–down temperature gradient. The TDR specifies the total temperature variation to be within 0.1 K [46].

The counting gas composition variations, in contrast, will require days to make a notable impact on the drift velocity.

Since the time constant of these two factors differ by orders of magnitude, it is practical to treat them separately, as in Equation 8.2.

$$v_d(t) = v_{d_{P/T}}(t) + v_{d_C}(t) \tag{8.2}$$

$v_{d_{P/T}}(t)$ is the relatively fast changing pressure and temperature component of the drift velocity, while $v_{d_C}(t)$ is the slowly changing gas composition variations. Most of the methods of obtaining the drift velocity will provide updated values about every 30 minutes. This applies to laser tracks, Goofie and cosmics track matching. However, it is possible to utilise the continuous measurements of temperature and pressure to produce "corrected" drift velocity corrections.

Further, for off-line correction, which can see future calibration values, it will be possible to combine the pressure and temperature corrections with interpolation of the consecutive values of base drift velocity corrections to form a smooth function. This obviously not possible for on-line calibration, though it might be possible to make predictions from the drift velocity trend. This will require careful investigation to limit the impact of false predictions.

In addition, there are factors that do not influence the drift velocity *per se*, but may influence the measurements of it:

- *trigger time offset*;
- *alignment*;
- *internal gradients*;
- *statistical fluctuations*.

The trigger time offset is the time offset from the collision took place to the detector receives the trigger. An inaccurate value causes an effect that is similar to incorrect drift velocity. For some of the methods, incorrect relative alignment of the sub-detectors can influence the measurements for drift velocity. Internal temperature and pressure gradients can be hard to map accurately in all three dimensions, and can give rise to local distortions that are not accurately taken into account.

8.2 Correction sources

The drift velocity correction can be obtained from several sources:

- track matching (cosmics and beam tracks):

  - TPC side A–TPC side C;
  - TPC–ITS;
  - TPC–TRD;
  - TPC–TOF; and
- laser tracks;
- Goofie — dedicated drift velocity monitor;
- distribution of last time bin.

Each method has both advantages and drawbacks. It is possible to combine the results from several sources in the over-all drift velocity calibration strategy.

8.2.1 Track matching

A particle traversing the detector might leave track segments in several sub-detectors. For a well-calibrated detector, the segments recorded by the different sub-detectors will match, making it possible to reconstruct the particle's path through the combined detector. The basis of obtaining the drift velocity correction from track matching is to exploit the mismatches to calculate a correction factor that will eliminate them.

If the drift velocity used for reconstruction is correct, the track of the particle passing through the CE will be reconstructed as a continuous track through both drift volumes, otherwise the mismatch at the CE will cause them to be considered two independent tracks. The track will have no other discontinuities beyond those usually experienced in track reconstruction. However, if the drift velocity is not correct, the tracks of the two sides will either overlap or leave a gap in the track along the z-axis. In the $r\phi$-direction, there should be no distortions caused by incorrect drift velocity.

Initially, the drift velocity of the TPC is assumed to be a nominal value close to average, corrected for currently measured gas pressure and temperature. The TPC tracks are reconstructed using this value. Then, the TPC tracks are attempted to be matched to the tracks recorded of the same particle, either by the other TPC side, or by some neighbouring sub-detector: ITS, TRD or TOF. The measured offset between track segments left in two neighbouring sub-detectors by the same particle is taken as the drift velocity correction value, Δz. If the track density is high or the error of the initial estimate for the drift velocity is large, it can be challenging to select the correct segments to match.

The method implicitly assumes that all other potential causes for the mismatch have already been corrected for, and that the incorrect drift velocity of the TPC is the *only* contributor to track mismatches. The TPC is the only detector needing drift velocity calibration; all other detectors are "fixed", and will in general not be influenced by the changing environment (temperature, pressure, *etc.*). I.e., their relative track alignments will not change much with time, except for physical intervention, and they

can be matched using static transformations. Hence, the condition for utilising the track-matching method should be fulfilled.

There are two sources of tracks: *cosmic* events and *beam* events. Most of the tracks will be produced in particle collisions. Since the statistics is very high, calculating the drift velocity correction as a running average might be an option for HLT. The cosmics tracks are from high-energy cosmic particles from outer space colliding with atmospheric particles, producing particles traversing the atmosphere before leaving a track in the TPC. Such events are not as frequent as beam collisions, hence it is not possible to use running average. However, since the tracks traverse the entire TPC, they can produce individual measurements of higher resolution. Collecting sufficient statistics for cosmics will take about 30 minutes. The track matching algorithms can utilise both types of tracks.

For matching the track segments in different sub-detectors, an optimised track matching algorithm is used. Conservative cuts assure that incorrect tracks are not matched; mismatched tracks will give the wrong drift velocity, and the ordinary track reconstruction will not see the track segments in different sub-detectors as parts of the same track.

TPC A–TPC C

This method [80] is based on matching tracks crossing the CE of the TPC. The track may come from cosmics or beam collisions, though the algorithms will treat the two cases slightly differently. Initially, a *nominal* average value for the drift velocity is assumed.

Initially, the track passing through the CE will be reconstructed as two separate tracks for each of the two halves of the TPC. The TPC track matching algorithms will try to search for potential matching tracks on each sides of the CE, which is done by searching for tracks with the same "direction" in some vicinity of the track on one side.

The algorithm applies slightly different assumptions in the case of cosmics tracks and beam tracks. In particular, it is assumed that cosmics particles are coming from outside the TPC, traverse it, and exit again. Particles from beam collisions, on the other hand, are assumed to originate approximately from the interaction point.

Figure 8.4 shows the drift velocity variations as function of time, measured by matching cosmics tracks from the two TPC sides, without any corrections. In Figure 8.5 the same measurements are plotted as function of $\Delta T/P$. Corrections for T/P has been applied in Figure 8.6. To the right, a time-dependent offset, ascribed to either a change of gas composition or trigger timing is observed. In Figure 8.7, this is corrected for as well.

The typical relative resolution of the TPC counting gas temperature and pressure is shown in Figures 8.8 and 8.9, respectively. For the pressure it is 6×10^{-5}, and for the temperature 1×10^{-5}, giving a total contribution to the uncertainty of 6.1×10^{-5}, *i.e.*, 150 μm for a drift chamber of 250 cm. Maintaining this resolution will require

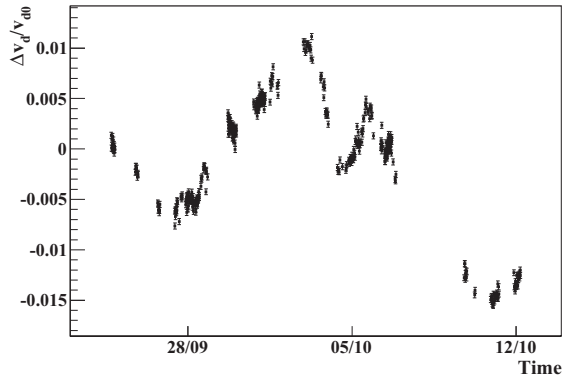

Figure 8.4: TPC uncorrected drift velocity as function of time. [80]

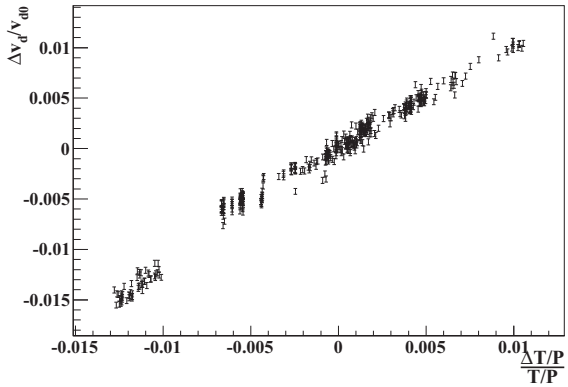

Figure 8.5: TPC uncorrected drift velocity as function of $\Delta T/P$. A close to linear dependence is observed. [80]

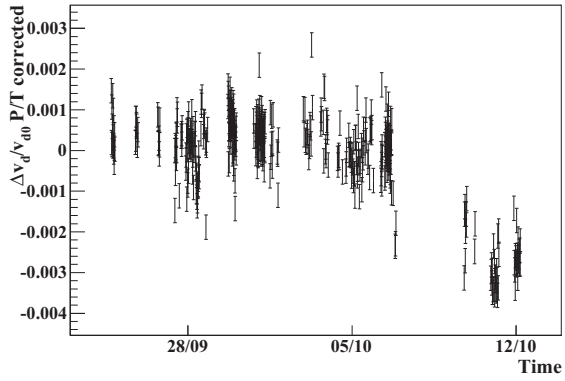

Figure 8.6: TPC drift velocity corrected for P/T. A time-dependent offset is seen to the right. [80]

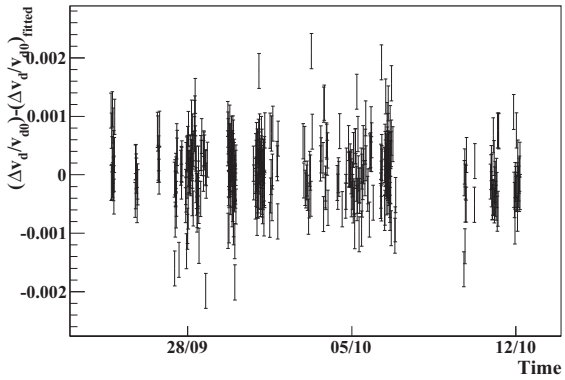

Figure 8.7: TPC drift velocity corrected for P/T and time-dependent offset. [80]

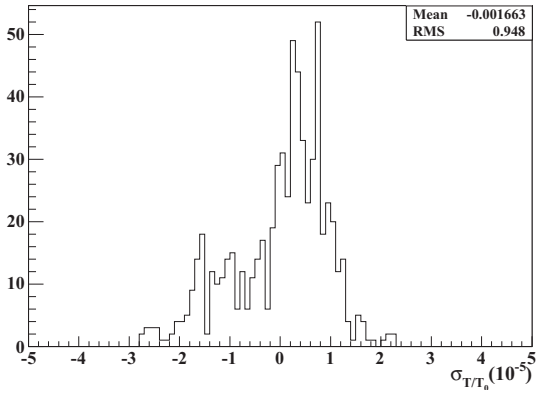

Figure 8.8: Relative resolution of TPC counting gas temperature. [80]

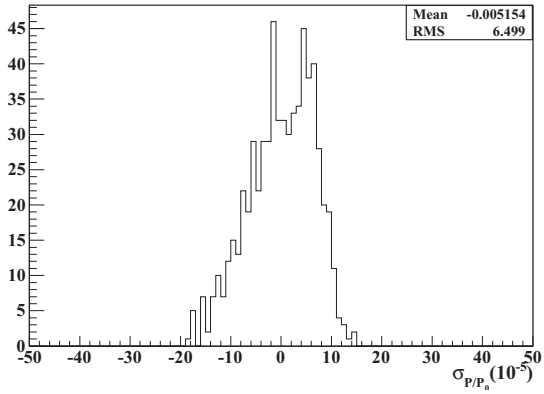

Figure 8.9: Relative resolution of TPC counting gas pressure. [80]

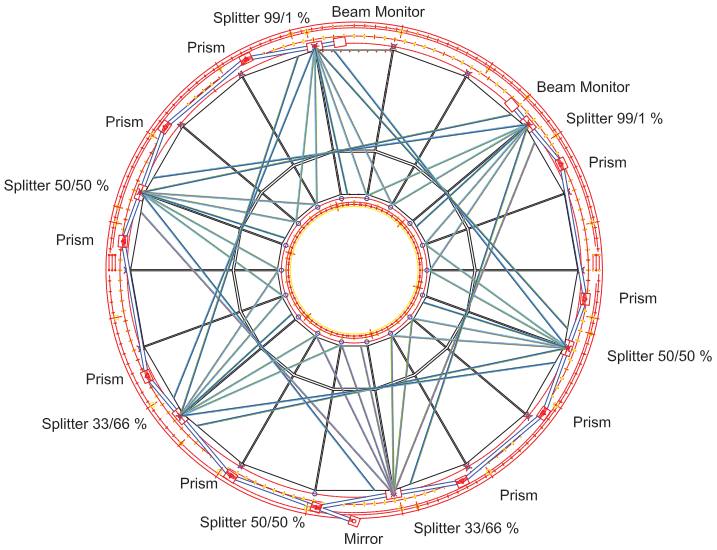

Figure 8.10: The laser tracks of the TPC. [47]

about 60 minutes to acquire sufficient statistics [80], but improvements in the calibration algorithms should bring this down to 30 minutes.

TPC–TRD, –ITS and –TOF track matching

This method is fundamentally not very different from that of TPC side A–C matching, which relied on matching tracks crossing the CE, i.e, a plane perpendicular to the z-axis. The drift velocity correction was obtained by calculating the overlap or gap of the tracks in the z-direction. For the matching of tracks from the TPC to those of ITS, TRD and TOF, tracks crossing some surface parallel to the z-axis are used. Once again, the correction is obtained from the mismatch in the z-direction. Here, however, there is no overlap or gap since the mismatch is along the axis crossed by the tracks. Rather, there is an offset along the z-axis between the corresponding tracks in the TPC and the other detectors. For this method, it is important that all involved detectors are well aligned.

8.2.2 Laser tracks

The tracks, see Figure 8.10, created in the TPC by laser beams can be used to correct the drift velocity [80, 81, 82]. A series of laser runs is expected to be taken approximately every 30 minutes. From a drift velocity calibration point of view, laser runs every five minutes would be desirable. At this time scaling the parameters influencing drift velocity should be sufficiently constant to allow direct use of the drift velocity correction factors

obtained. The frequency of the laser runs is limited by the life time of the crystals generating the laser beams; they should only be replaced every LHC shutdown, which is expected to be once a year.

Laser runs is the source of drift velocity correction values with highest resolution. Hence it can be used for calibrating the other correction methods. This can be done by comparing the correction values from the laser runs to the correction values obtained from other methods at the time of the laser runs. Hence, a new base line for the correction is obtained.

8.2.3 Goofie

Goofie is a dedicated drift velocity monitor for the TPC [83]. It consists of a gas chamber where the TPC counting gas is circulated at a slow rate. A known electric field is applied to the gas volume, and the drift time of electrons is measured over a fixed distances. It is approximately one meter long; the measured drift velocity has to be scaled to the TPC drift length, causing some loss of resolution. There can also be a temperature offset between the gas in the Goofie and the TPC, which has to be taken into account. Goofie requires about 30 minutes to collect sufficient statistics for an updated drift velocity value. The drift velocity value is exported via the DCS, and made available in the CDB.

8.2.4 Time-bin distribution

The signals from the TPC read-out pads are collected by the ALTROs, and stored in *time-bins*. It is possible to use the distribution of the last time bin (*i.e.* the bin of the longest drift distance) to estimate the TPC drift velocity. It is implemented in the CE *Detector Algorithm* (DA), and is processed on-line on DAQ, from where it is stored into the CDB.

8.3 Systematics of effects

Figure 8.11 shows the perfectly calibrated TPC tracks. This can be compared to the situation of incorrect drift velocity, which will manifest itself as a gap, Figure 8.12, or overlap, Figure 8.13, with length Δz between the two TPC drift volumes around the CE. There are, however, other effects that can have a similar manifestation. It is important to disentangle these effects, and correct them in the right order. Otherwise, new inaccuracies will be introduced.

The first effect is *misalignment*. Although every possible care is taken to produce and align the sub-detectors physically, there will always be minor discrepancies that will have to be corrected for in software. Figure 8.14 shows the result of incorrect alignment of the TPC with respect to the ITS. From the perspective of the ITS, the whole TPC

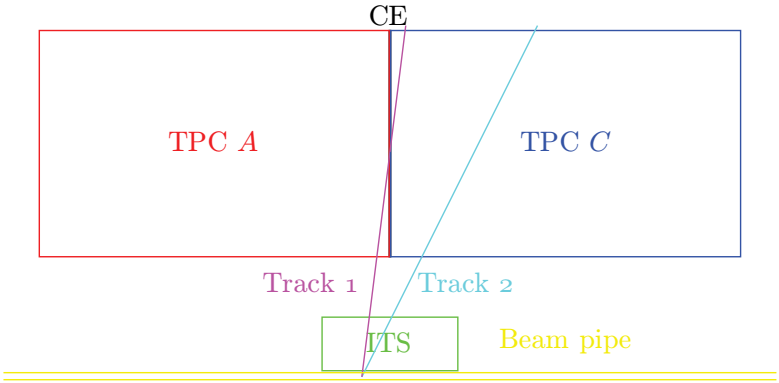

Figure 8.11: Perfectly calibrated TPC tracks.

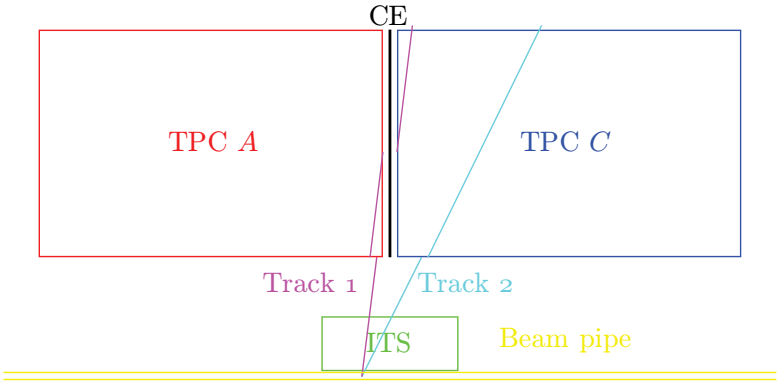

Figure 8.12: Impact of uncorrected positive Δz scaling on TPC tracks. Note the TPC end-planes are unmoved. The situation looks very similar to the case of t_0 shift, except the chambers appear shortened (scaled), not shifted. Negative Δz shown in Figure 8.13.

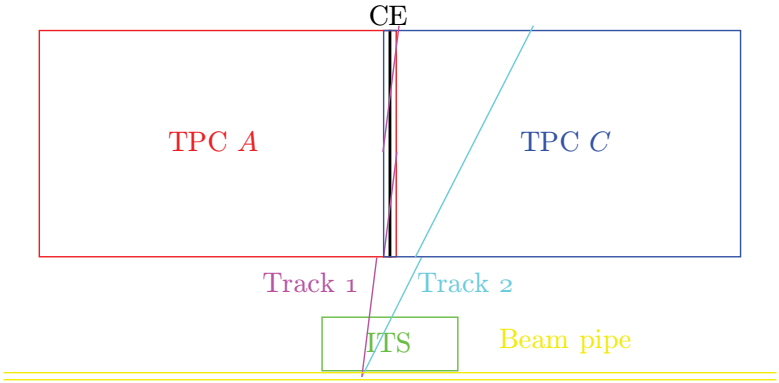

Figure 8.13: Impact of uncorrected negative Δz scaling on TPC tracks. The TPC end-planes are unmoved. Again, the situation is similar to that of a negative t_0 shift, except the chambers appear stretched (scaled).

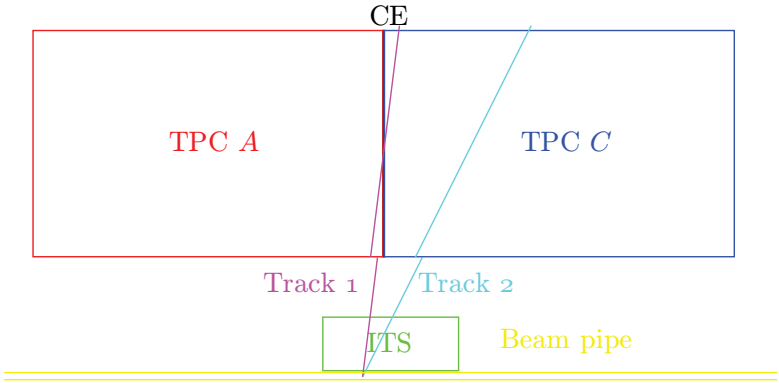

Figure 8.14: Impact of uncorrected TPC–ITS shift on TPC tracks. The whole TPC, including CE, has been shifted left-wards with respect to the ITS. A negative shift is right-wards.

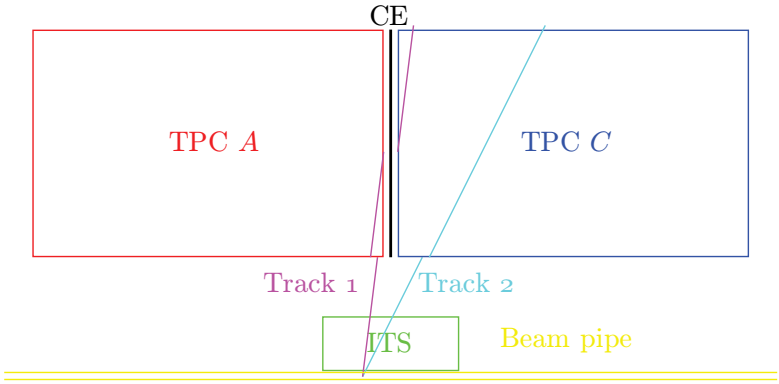

Figure 8.15: Impact of uncorrected t_0 shift on TPC tracks. Each chamber appears to have been shifted away from the CE, leaving a gap. If the t_0 shift is negative, the shift is towards the CE, resulting in overlapping chambers.

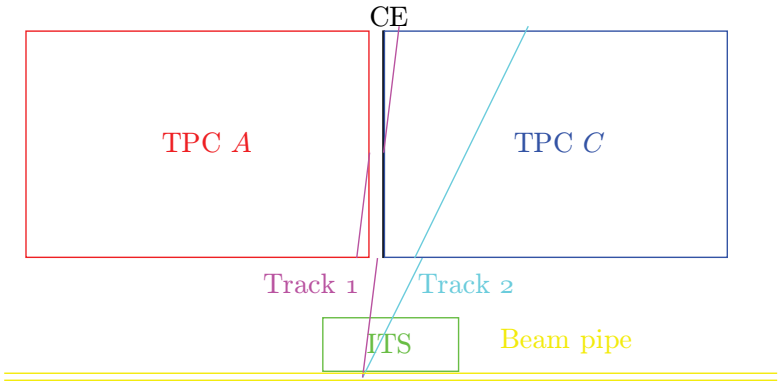

Figure 8.16: Impact of uncorrected t_0, TPC–ITS shift and Δz scaling on TPC tracks. All positive.

has been shifted to either side. Since the TPC is affected "globally", misalignment will not contribute to Δz, but since the positions of the TPC end planes are shifted, it will affect the algorithms for correcting Δz if not corrected beforehand. The alignment does not change with time, except when there has been physical intervention that may change the alignment.

The time of the collision, *time-zero*, has to be known to match tracks crossing the CE and read out from the two end planes. Otherwise, the two drift volumes will be shifted in opposite directions relative to the CE, in either direction, creating a gap or overlap similar to that produced by an incorrect drift velocity, contributing to the initially overall measured Δz. The time-zero may depend on the trigger type, as the different triggers will detect the collision at different time offsets. Ideally, the triggers should be calibrated to the same offset. The time-zero offset is not time-dependent, although the trigger offsets can change.

Incorrect drift velocity will *scale* the TPC drift chambers, *i.e.*, the effect will have its maximum at the CE, and linearly decrease towards the end planes, where it is *zero*. Incorrect time-zero offset, on the other hand, will *shift* the chambers, *i.e.*, the effect will be constant over the full drift length, Figure 8.15. Since both effects contribute to the measured Δz, they can be hard to separate, fortunately, only the drift velocity is changing with time. Although the apparent effect is the same, the different nature of scaling versus shift means they have to be corrected differently; time-zero is corrected by shifting the chambers according to the contribution of effect to Δz as measured at CE, drift velocity is corrected by obtaining a scaling factor that can be distributed over the full drift length. The time-zero offset has to be corrected for before drift velocity correction, otherwise the shift of the time offset will be interpreted as a drift velocity variation, hence wrongly "corrected" for. For the part of the TPC volume close to the CE, the effect of this misinterpretation will be small, for the volume close the the end planes it will however be large; without correction for time offset shift, large errors will be introduced to the track matching with other detectors.

In principle, it may be possible to distinguish between all these effects by attempting to match tracks from other sub-detectors to the TPC tracks, then compare the distance of the mismatch close to the CE and the end planes, hence obtaining a global and a relative shift component, and one scale component.

Figure 8.16 shows all effects present simultaneously. A challenge with such approach is the much lower density of tracks crossing both the TPC and the other detector close to the end planes. In reality the situation is simplified, since the alignment and time-zero calibration is constant, and drift velocity is the only time-dependent variable.

8.4 Strategy

Currently, the three main methods for obtaining the drift velocity correction are:

- *TPC side* A–B *track matching*;
- *laser tracks*;
- *TPC–ITS track matching.*

.

All methods provide updated values approximately every 30 minutes. It is possible to combine the results, for example, the high-precision measurements by the laser system can be used to calibrate the track matching methods. The results from all methods are calculated and stored to the CDB, and may be compared to determine which will give the best result. The main strategy is to use the HLT to generate drift velocity calibration objects to be used by off-line reconstruction, however, there is still work ongoing.

If other methods should prove to generate better results, the strategy may be revised. The flexibility of the framework allows for easy change of calibration method.

8.5 Impact of uncalibrated TPC drift velocity on physics

The TPC drift velocity calibration is the mapping of the measured particle trajectories to the "true" trajectories, taking into account, and correcting for, distortions that may disturb the measurements. If the calibration is not well done, the particle tracks will not appear at the correct spatial location. This will mainly have an impact at the borders between different detectors, both TPC side A–TPC side B as well as TPC–ITS, –TRD and –TOF, where it will be difficult, if not impossible, to perform efficient *track matching*. Track matching is to merge the track segments the same particle leaves in multiple sub-detectors when traversing them. Since the TPC drift velocity depends on the environment variables gas pressure and temperature, the track matching efficiency would literally depend on the weather if not done properly.

If the track matching fails, the different track segments will be interpreted as being created by different particles. This can have several consequences. There will appear to be *more* particles with *shorter* tracks. Most importantly, though, it will be impossible to trace the same particle through several sub-detectors. This also means it will not be possible to compile information of the properties of the same particle from several sub-detectors. The overall impact on the performance will depend on the physics to be studied. For example high-p_T-measurements, depend on good track-matching, and will suffer significantly, as well as the detection of tracks emerging from a secondary vertex, *i.e.* tracks with a large impact parameter. Flow measurements, on the other hand, might suffer less.

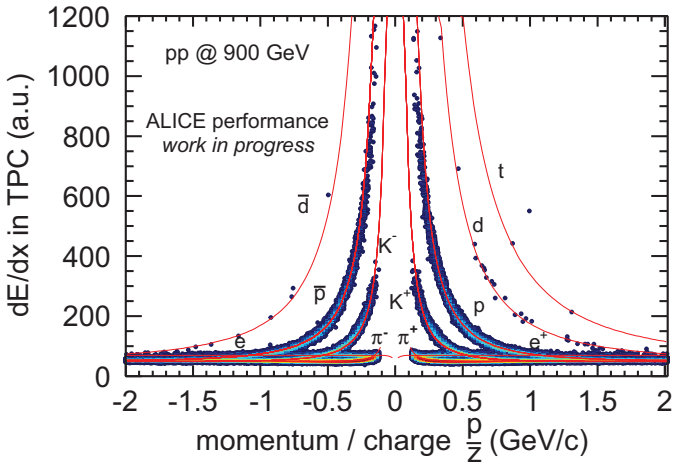

Figure 8.17: TPC dE/dx $p+p$ resolution. [84]

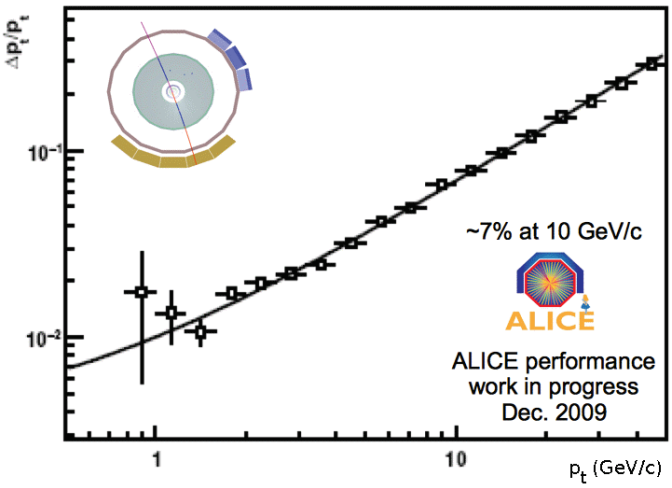

Figure 8.18: TPC p_T cosmics resolution. [84]

8.6 TPC performance

Before the LHC start-up last winter, the TPC was collecting cosmics data. After the start-up, $p+p$ data have been collected. Based on these sets of data, thorough performance studies have been carried out, and a number of plots showing the performance have been released.

The dE/dx resolution of the TPC for $p+p$ collisions is shown in Figure 8.17. The coloured dots indicate data points, while the red lines are Bethe-Bloch fits for various

charged particles. Figure 8.18 shows the TPC p_T resolution for cosmics data. Work is in progress for $p + p$ data.

Chapter 9

Conclusion and outlook

The ALICE experiment is now commissioned, and is currently collecting data from $p + p$ collisions, while waiting for $Pb + Pb$ collisions later this year. Although ALICE is optimised for $Pb + Pb$ collisions, the $p + p$ data are useful for the detector calibration, and also as a reference for the $Pb + Pb$ collisions.

The detector is performing close to the design goals, and work is in progress to improve the performance further. Although the LHC programme has begun, and ALICE is taking data, there are still upgrades foreseen. For some of the sub-detectors, like PHOS, EMCAL and TRD, further modules are to be installed. Other upgrades, like the new DCAL sub-detector, are yet to be installed. In addition, further sub-detectors, like a forward calorimeter, are in the planning stage.

The ALICE TPC is performing well. From data collected so far from cosmics and $p + p$ collisions, it shows good PID and tracking capabilities.

A DCS has been developed around networked computers embedded on the FEE of the detector itself. The computers are running a software called FeeServer, which allows remote clients to connect to it via the network. The client software will send binary blocks configuration data and instructions to the FeeServer, which will interpret the commands, and configure the electronics accordingly. Also, monitoring values from the FEE are published by the FeeServer and subscribed to by the client software. The data-points are give information of the operation status of the FEE or the FeeServer software itself, and will be displayed visually in a GUI where the shifter can easily inspect the status of the detector. The FeeServer is also used by auxiliary systems like the BusyBox, the gate pulser, the laser pulser, the trigger-or, and by other sub-detectors like the PHOS and EMCAL.

The FeeServer is performing well, and it has been shown that the processing time for the FeeServer of the configuration data is of the same order as the time for assembling the data blocks on the client side, and can not be considered a bottleneck. However, there is still room for improvement. Extensions for the FeeServer command set that will allow for more efficient and robust configuration is being considered. In particular,

this applies to the implementation of monitoring and correction of single-event upsets in the ALTROs and the SRAM of the RCU FPGA. Also, searching for and identifying situations where the FeeServer may not function properly has to continue.

Much effort has gone into the different aspects of the drift velocity calibration. Several methods for obtaining the various calibration parameters are utilised to provide independent measurements that can be compared for consistency. A time dependent variation of the drift velocity is intrinsic to a TPC detector. The drift velocity depends on a number of factors, with very large variations of the time constant. The most important parameters are the fast changes of gas temperature and pressure, as well as the slow gas composition variations. There are several methods for obtaining the drift velocity; the methods based on various types of track matching and laser tracks are the most important. It is possible to combine or augment the different methods to obtain a better overall result. Initial calibration values for the off-line reconstruction can be provided by the HLT, eliminating the need for a pass *0* reconstruction.

Overall, the ALICE experiment is living up to the expectation, and with the arrival of the $Pb + Pb$ collisions later this year, ALICE will have the chance to prove its real strengths.

Publications

Primary author

1. ALICE TPC commissioning results
   D. T. Larsen for the ALICE TPC Collaboration
   Nuclear Instruments and Methods in Physics Research **A617**, 35–39 (2010),
   doi:10.1016/j.nima.2009.07.002

2. ALICE TPC control and read-out system
   D. T. Larsen for the ALICE TPC Collaboration
   CERN Reports **006** (2009),
   http://cdsweb.cern.ch/record/1185010

Co-author

3. DCS Communication Software for the ALICE TPC Front-End Electronics
   M. Richter *et al.* [ALICE TPC Collaboration]
   CERN Reports **011** (2005),
   http://cdsweb.cern.ch/record/921200

4. A Distributed, Hetrogeneous Control System for the ALICE TPC Electronics
   [ALICE TPC Collaboration]
   Parallel Processing (2005),
   http://cdsweb.cern.ch/record/914454

5. The control system for the front-end electronics of the ALICE time projection chamber
   M. Richter *et al.* [ALICE TPC Collaboration]
   IEEE Trans. Nucl. Sci. **53**, 980 (2006),
   doi:10.1109/TNS.2006.874726

6. The ALICE Experiment at the CERN LHC
   K. Aamodt *et al.* [ALICE Collaboration]

JINST **0803**, S08002 (2008)
doi:10.1088/1748-0221/3/08/S08002

7. First proton–proton collisions at the LHC as observed with the ALICE detector: measurement of the charged particle pseudorapidity density at $\sqrt{s} = 900\ GeV$
   K. Aamodt *et al.* [ALICE Collaboration]
   European Physical Journal **C65** (2010),
   arXiv:0911.5430

8. The ALICE TPC, a large 3-dimensional tracking device with fast readout for ultra-high multiplicity events
   J. Alme *et al.* [ALICE TPC Collaboration]
   Submitted to Nuclear Instruments and Methods in Physics Research **A** (2010),
   arXiv:1001.1950

Including all publications as ALICE collaboration member co-author, a total of 59 publications.

Bibliography

[1] A. Lygre, *Eld av steinar* (Gyldendal, Oslo, 1948). b

[2] C. Salgado, (2009), arXiv:0907.1219. 2, 5, 11

[3] L. McLerran, Lect. Notes Phys. **583**, 291 (2002), arXiv:hep-ph/0104285. 3, 15

[4] F. Karsch, Nucl. Phys. **A698**, 199 (2002), arXiv:hep-ph/0103314. 1, 6

[5] F. Karsch *et al.*, Quark–gluon plasma **III** (2003), arXiv:hep-lat/0305025. 1, 6

[6] B. Holzman *et al.*, Nucl. Phys. **A698**, 643 (2002), arXiv:nucl-ex/0103015. 1

[7] C. Ogilvie, Nucl. Phys. **A698**, 3 (2002), arXiv:nucl-ex/0104010. 1

[8] BNL, *AGS web page*, 2010, http://bnl.gov/rhic/AGS.asp. 1

[9] M. Gazdzicki, Eur. Phys. J. ST. **155**, 37 (2008), arXiv:0801.4919. 1

[10] NA49-Collaboration, Phys. Rev. **C79**, 044904 (2009), arXiv:0810.5580. 1

[11] CERN, *SPS web page*, 2010, http://cern.ch/ab-dep-op-sps/. 1

[12] I. Arsene *et al.*, Nucl. Phys. **A757**, 1 (2005), arXiv:nucl-ex/0410020. 1

[13] J. Adams *et al.*, Nucl. Phys. **A757**, 102 (2005), arXiv:nucl-ex/0501009. 1, 13, 15

[14] BNL, *RHIC web page*, 2010, http://bnl.gov/rhic/. 1

[15] J. Rafelski *et al.*, J. Phys. **G35**, 044011 (2008), arXiv:0801.0588. 1

[16] X.-N. Wang, Nucl. Phys. **A750**, 98 (2005), arXiv:nucl-th/0405017. 1, 11, 12, 13

[17] I. Arsene *et al.*, Phys. Rev. Lett. **91**, 072305 (2003), arXiv:nucl-ex/0307003. 1, 11, 12, 13

[18] L. Landau, Izv. Akad. Nauk SSSR Ser. Fiz. **17**, 51 (1953). 3

[19] J. Bjorken, Phys. Rev. **D27**, 140 (1983), doi:10.1103/PhysRevD.27.140. 3

[20] B. Povh *et al.*, *Particles and nuclei* (Springer-Verlag, 1995), isbn:3540594396. 4

[21] R. Debbe *et al.*, J. Phys. **G35**, 104004 (2008), arXiv:0805.0780. 5

[22] H. Caines, Proceedings for the Rencontres de Moriond 2009 QCD session (2008), arXiv:0906.0305. 5

[23] J. Letessier *et al.*, *Hadrons and QGP* (Cambridge University Press, 2002), isbn:0521385369. 5

[24] K. Eskola *et al.*, Nucl. Phys. **A713**, 167 (2003), arXiv:hep-ph/0205048. 5

[25] Z. Xu, J. Phys. **G30**, 927 (2004), arXiv:nucl-ex/0404034. 7

[26] R. Snellings *et al.*, Eur. Phys. J. **C49**, 8790 (2007), arXiv:nucl-ex/0610010. 7

[27] J.-Y. Ollitrault, Phys. Rev. **D46**, 229245 (1992), doi:10.1103/PhysRevD.46.229. 7

[28] R. Lacey *et al.*, PoSCFRNC **021** (2006), arXiv:nucl-ex/0610029. 7

[29] A. Chaudhuri, (2010), arXiv:0910.0979. 7

[30] J.-Y. Ollitrault, Nucl. Phys. **A638**, 195 (1998), arXiv:nucl-ex/9802005. 7

[31] S. Voloshin *et al.*, Z. Phys. **C70**, 665 (1996), arXiv:hep-ph/9407282. 8

[32] A. Poskanzer *et al.*, Phys. Rev. **C58**, 1671 (1998), arXiv:nucl-ex/9805001. 8

[33] K. Ackermann *et al.*, Phys. Rev. Lett. **86**, 402 (2001), arXiv:nucl-ex/0009011. 9

[34] J. Adams *et al.*, Phys. Rev. Lett. **95**, 122301 (2005), arXiv:nucl-ex/0504022. 9, 10

[35] S. Shi *et al.*, Nucl. Phys. **A830**, 187c (2009), arXiv:0907.2265. 9, 10

[36] R. Nouicer, Submitted to Eur. J. Phys. (2009), arXiv:0901.0910. 12, 14

[37] B. Abelev *et al.*, Phys. Let. **B655**, 104 (2007), arXiv:nucl-ex/0703040. 12, 14

[38] S. Adler *et al.*, Phys. Rev. Lett. **91**, 072303 (2003), arXiv:nucl-ex/0306021. 12, 14

[39] S. Adler *et al.*, Phys. Rev. Lett. **96**, 202301 (2006), arXiv:nucl-ex/0601037. 12, 14

[40] S. Adler *et al.*, Phys. Rev. Lett. **98**, 172302 (2007), arXiv:nucl-ex/0610036. 12, 14

[41] A. Accardi, CERN reports **009**, 81 (2004), arXiv:hep-ph/0212148. 12

[42] ALICE-Collaboration, *Technical Proposal for A Large Ion Collider Experiment at the CERN LHC*, CERN, 1995, http://cdsweb.cern.ch/record/293391. 17

[43] C. Lippmann, *private communication*. 21, 34

[44] A. Kalweit *et al.*, WSPC proceedings (2009). 22

[45] A. Kalweit, *private communication*. 23, 96

[46] ALICE-Collaboration, *ALICE Technical Design Report of the Time Projection Chamber*, CERN, 2000, http://edms.cern.ch/document/398930. 19, 20, 23, 104

[47] J. Alme *et al.*, In press, NIM **A** (2010), arXiv:1001.1950. 20, 21, 103, 109

[48] A. Rehman, *private communication*. 21

[49] ALICE-Collaboration, *ALICE Technical Design Report of the Photon Spectrometer*, CERN, 1999, http://edms.cern.ch/document/398934. 24

[50] ALICE-Collaboration, *ALICE Addendum to the Technical proposal Electromagnetic Calorimeter*, CERN, 2006, http://aliceinfo.cern.ch/Collaboration/Documents/EMCal.html. 25

[51] ALICE-Collaboration, *ALICE Technical Design Report of the Inner Tracking System*, CERN, 1999, http://edms.cern.ch/document/398932. 25

[52] ALICE-Collaboration, *ALICE Technical Design Report of the Transition Radiation Detector*, CERN, 2001, http://edms.cern.ch/document/398057. 25

[53] ALICE-Collaboration, *ALICE Technical Design Report of the Time of Flight System*, CERN, 2002, http://edms.cern.ch/document/460192. 26

[54] ALICE-Collaboration, *ALICE Technical Design Report of the High Momentum Particle Identification Detector*, CERN, 1998, http://edms.cern.ch/document/316545. 26

[55] ALICE-Collaboration, *The Forward Muon Spectrometer of ALICE Addendum to the Technical Proposal for ALICE at the CERN LHC*, CERN, 1996, http://edms.cern.ch/document/316523. 26

[56] ALICE-Collaboration, *ALICE Technical Design Report of the Dimuon Forward Spectrometer*, CERN, 1999, http://edms.cern.ch/document/470838. 26

[57] ALICE-Collaboration, *ALICE Technical Design Report of the Zero Degree Calorimeter*, CERN, 1999, http://edms.cern.ch/document/398933. 26

[58] ALICE-Collaboration, *ALICE Technical Design Report on Forward Detectors: FMD, T0 and V0*, CERN, 2004, http://edms.cern.ch/document/498253. 27

[59] ALICE-Collaboration, *ALICE Technical Design Report of the Photon Multiplicity Detector*, CERN, 1999, http://edms.cern.ch/document/398931. 27

[60] ALICE-Collaboration, *ALICE Addendum to the Technical Design Report of the Photon Multiplicity Detector*, CERN, 2003, http://edms.cern.ch/document/575585. 27

[61] ALICE-Collaboration, *ALICE Technical Design Report of the Trigger, Data Acquisition, High-Level Trigger and Control System*, CERN, 2004, http://edms.cern.ch/document/398931. 27, 30, 33

[62] J. Alme, A trigger based readout and control system operating in a radiation environment, PhD thesis, University of Bergen, Bergen, 2008, http://cdsweb.cern.ch/record/1141616. 31, 35, 37, 47

[63] ETM, *PVSS web page*, 2010, http://www.etm.at/index_e.asp?id=2&amp%3Bm0id=6. 33

[64] L. Musa *et al.*, IEEE NSS Conference Record **5**, 3647 (2003), doi:10.1109/NSSMIC.2003.1352697. 34

[65] D. Larsen, CERN reports **006**, 586 (2009), http://cdsweb.cern.ch/record/1185010. 34

[66] T. Krawutschke, *DCS board web page*, 2010, http://frodo.nt.fh-koeln.de/~tkrawuts/dcs.html. 40

[67] K. Røed, Single event upsets in sram fpga based readout electronics for the time projection chamber in the alice experiment, PhD thesis, University of Bergen, Bergen, 2009, http://cdsweb.cern.ch/record/1244467. 46

[68] L. Musa *et al.*, *TPC FEE web page*, 2010, http://cern.ch/ep-ed-alice-tpc/. 47

[69] C. González-Gutiérrez *et al.*, IEEE NSS Conference Record **1**, 575 (2005), doi:10.1109/NSSMIC.2005.1596317. 47

[70] S. Bablok, Development and implementation of a safe and efficient communication software in a heterogeneous system environment of a major research project, Master's thesis, Fachhochschule Worms, Worms, 2004. 54

[71] D. Larsen, Utvikling av feilsøkingsverktøy for alice tpc forende-elektronikk via detektorkontrollsystemet, Master's thesis, University of Bergen, Bergen, 2006, http://web.ift.uib.no/~dagtl/dag-master.pdf. 56

[72] M. Richter *et al.*, IEEE Trans. Nucl. Sci. **53**, 980 (2006), doi:10.1109/TNS.2006.874726. 56

[73] D. Larsen *et al.*, *FeeServer source code*, 2010, http://web.ift.uib.no/kjekscgi-bin/viewcvs.cgi/. 57

[74] C. Gaspar *et al.*, Proceedings for CHEP (2000). 70

[75] C. Gaspar *et al.*, *DIM web page*, 2010, http://cern.ch/dim/. 70

[76] T. Lohse *et al.*, Advanced series on directions in high energy physics **9**, 81 (1993), isbn:9810214731. 91

[77] D. Vranic, CERN-ALICE-INT **22** (1997), http://cdsweb.cern.ch/record/689328. 92

[78] D. Larsen *et al.*, NIM **A617**, 35 (2010), doi:10.1016/j.nima.2009.07.002. 92, 93

[79] S. Rossegger, Simulation & calibration of the alice tpc including innovative space charge calculations, PhD thesis, Graz University of Technology, Graz, 2009, http://cdsweb.cern.ch/record/1217595. 94

[80] M. Ivanov *et al.*, *AliRoot TPC calibration documentation*, 2010, http://alisoft.cern.ch/AliRoot/trunk/TPC/doc/calib/. 95, 96, 106, 107, 108, 109

[81] ALICE-Collaboration, *AliRoot documentation*, 2010, http://aliceinfo.cern.ch/static/aliroot-new/html/roothtml/. 109

[82] ALICE-Collaboration, *AliRoot TPC source code*, 2010, http://alisoft.cern.ch/AliRoot/trunk/TPC/. 109

[83] D. Antończyk *et al.*, GSI Scientific Report , 348 (2004), http://www.gsi.de/informationen/wti/library/scientificreport2004/PAPERS/INSTMETH-20.pdf. 110

[84] ALICE-Collaboration, *Validated figures for ALICE presentations*, 2010, http://twiki.cern.ch/twiki/bin/viewauth/ALICE/ConferenceCommitteeFigures. 115

Glossary

ADC Analogue–Digital Converter. 22, 51, 52, 65

AGS Alternating Gradient Synchrotron. 1

ALICE A Large Ion Collider Experiment. iii, II, 16, 17, 19, 23, 24, 26, 27, 29, 30, 34, 44, 50, 96, 103, 117, 118

ALTRO ALice Tpc Read-Out. 21, 24, 28, 43, 47–51, 64, 65, 88, 110, 118

APD Avalanche Photo-Diode. 24, 29, 48, 49, 64, 66

ARM Advanced Risc Machine. 21, 38

ASIC Application-Specific Integrated Circuit. 28, 38

ATLAS A Toroidal Lhc ApparatuS. 17

BB BusyBox. 75

BC Board Controller. 25, 47–52, 64–66

BLOB Binary Large OBject. 65, 71, 73

BNL Brookhaven National Laboratory. 1

BRAHMS Broad Range HAdron Magnetic Spectrometer. 1, 12, 13

CAD Computer-Assisted Drawing. 89

CDB Condition DataBase. 30, 96–100, 110, 114

CE Centre-Electrode. 20, 91, 92, 94, 98, 101, 105, 106, 109, 110, 112, 113

CE ControlEngine. 54–60, 63, 66, 77, 78, 81, 83, 88, 102

CMS Compact Muon Solenoid. 17

CoCo CommandCoder. 65, 66, 68, 69, 71, 72

CPU Central Processing Unit. 36, 38–40

CR Counting Room. 33

CTP Central Trigger Processor. 29, 31, 38, 47

DA Detector Algorithm. 110

DAQ Data AcQuisition. 21, 28, 30, 31, 34, 46, 47, 110

DCAL Di-jet CALorimeter. 25, 117

DCB DCs Board. 42

DCS Detector Control System. iii, 16, 21, 24, 25, 28, 33, 34, 36–48, 51, 53, 54, 59, 63, 67–73, 77, 78, 80, 81, 88, 110, 117

DDL Detector Data Link. 28–30, 46, 67

DHCP Dynamic Host Configuration Protocol. 40, 41, 44

DID Dim Information Display. 69

DIM Distributed Information Management. 37, 43, 53, 54, 58, 69–71, 82

DIMNS DIM Name Server. 70, 71

DIU Destination Interface Unit. 28

DNS Domain Name System. 44

DRORC Destination Read-Out Receiver Card. 28, 29

ECS Experiment Control System. 33, 70

EMCAL Electro-Magnetic CALorimeter. 25, 28, 30, 34, 66, 68, 117

FEC Front-End Card. 21, 24, 25, 28, 29, 33, 34, 37, 38, 45–52, 57, 60, 63, 64, 66, 68, 69, 71–74, 80, 81, 88

FED Front-End Device. 71

FEE Front-End Electronics. iii, 22, 23, 29, 33, 34, 36, 38, 40, 41, 43, 45, 48, 53, 54, 56, 63, 67–69, 71, 74, 77, 78, 80, 81, 103, 117

FMD Forward Multiplicity Detector. 27–29, 50, 55, 64

FPGA Field Programmable Gate Array. 37–39, 45–47, 49, 67, 80, 88, 118

GDC Global Data Concentrator. 30

GTL Gunning Transceiver Logic. 47–51, 65, 80, 88

GUI Graphical User Interface. 33, 68, 70, 74, 117

HBT Hanbury-Brown Twiss. 7

HLT High Level Trigger. 21, 27, 28, 30, 31, 34, 46, 99, 100, 106, 114, 118

HMPID High Momentum Particle Identification Detector. 26

I^2C Inter-Integrated Circuit. 28, 47–50, 59, 64, 80, 81

ICL InterComLayer. 33, 37, 43, 65, 68–73, 75, 79, 84

ID IDentification. 57, 60, 61, 69, 85–87

IM Instruction Memory. 48, 64, 65, 84

IOCTL Input–Output ControL. 44, 79, 80

IP Internet Protocol. 40, 44, 70

IROC Inner Read-Out Chamber. 20, 21, 93

IT Information Technology. 44

ITS Inner Tracking System. 25, 26, 30, 91, 105, 109, 110, 112, 114

JICL Java ICL. 43, 72

JTAG Joint Test Action Group. 39, 43, 81

LDC Local Data Concentrator. 30

LEIR Low Energy Ion Ring. 19

LEP Large Electron–Positron Collider. 17

LHC Large Hadron Collider. iii, 1, 4, 6, 16, 17, 19, 70, 71, 110, 115, 117

LHCb LHC-Beauty. 17

LINAC LINear ACcelerator. 19

LO Leading Order. 5

LQCD Lattice QCD. 2, 6

LSB Least Significant Bit. 39, 44

MAC Media Access Control. 38, 44

MMU Memory Management Unit. 38

MSB Most Significant Bit. 38, 44

MUON MUON spectrometer. 26

MWPC Multi-Wire Proportional Chamber. 20, 21, 91

NFS Network File System. 41, 44

NLO Next-to-Leading Order. 5

NTP Network Time Protocol. 44

OROC Outer Read-Out Chamber. 21, 93

PASA PreAmplifier ShAper. 49, 50

PCI Peripheral Component Interconnect. 28

PDF Parton Distribution Function. 5, 11

PHENIX Pioneering High-Energy Nuclear Interactions eXperiment. 1, 11, 12, 14

PHOS PHOton Spectrometer. 24, 25, 28–30, 34, 48–50, 56, 57, 64, 66, 68, 117

PID Particle IDentification. 19, 96, 117

PMD Photon Multiplicity Detector. 27

pQCD perturbative QCD. 5, 6, 11

PS Proton Synchrotron. 19

PVSS ProzessVisualisierungs- und Steuerungs-System. 33, 52, 61, 68–75

QCD Quantum Chromo-Dynamics. 1, 2, 4–6

QED Quantum Electro-Dynamics. 5

QGP Quark–Gluon Plasma. iii, 1–5, 7, 9, 11, 13, 15–17

RAM Random Access Memory. 38–42, 45, 46

RCU Read-out Control Unit. 21, 24, 28–30, 34, 36–38, 41, 44–54, 57, 59, 63–65, 68, 69, 72, 80, 81, 84, 88, 118

RF Radio Frequency. 17

RHIC Relativistic Heavy Ion Collider. 1, 5, 7, 9–16

RICH Ring-Imaging CHerenkov. 26

RM Result Memory. 48

RMS Root Mean Square. 22

RO Read-Only. 39, 41, 42

ROC Read-Out Chamber. 91

ROMFS ROM File System. 39

RP Read-out Partition. iii, 21, 33, 43, 48

RS Recommended Standard. 38

RW Read–Write. 39, 41, 42

SC Slow-Control. 48

SCP Secure CoPy. 39, 44

SDD Silicon Drift Detector. 25

SIU Source Interface Unit. 34, 36, 46

SM Safety and Monitoring module. 46

SPD Silicon Pixel Detector. 25

SPS Super Proton Synchrotron. 1, 19

sQGP Strongly coupled QGP. 5, 7

SRAM Static RAM. 88, 118

SSD Silicon Strip Detector. 25

SSH Secure SHell. 39, 41, 44

SSHD SSH Daemon. 41

STAR Solenoidal Tracker At Rhic. 1, 9–15

T0 Time-Zero. 27

TCP Transmission Control Protocol. 70

TDR Technical Design Report. 89, 104

TOF Time-Of-Flight. 26, 27, 105, 109, 114

TOR Trigger-OR. 29, 30, 66

TPC Time Projection Chamber. iii, 16, 19–26, 28–30, 33–36, 40, 43, 44, 48–51, 53, 56, 57, 64, 66–69, 71, 72, 74, 75, 89–99, 101–115, 117, 118

TRD Transition Radiation Detector. 25, 26, 30, 34, 55, 64, 105, 109, 114, 117

TRU Trigger Read-out Unit. 25, 29, 30, 50, 64, 66

TTC Timing, Trigger and Control. 27

TTCrx TTC Receiver. 28, 37, 65, 80

UDHCPCD Micro DHCP Client Daemon. 41

V0 Veto. 27

ZDC Zero Degree Calorimeter. 26