Master thesis

Implementation of the Cluster Pixel Data Format into the Belle II Analysis Framework

Implementierung des Cluster-Pixel-Datenformats in die Belle II Analyse Umgebung

September 24, 2015

Author:
Klemens Lautenbach
Justus-Liebig-Universität Gießen
II. Physikalisches Institut
Heinrich-Buff-Ring 16
35390 Gießen

Supervision:
Prof. Dr. Wolfgang Kühn
PD. Dr. J. Sören Lange AR

Contents

1 Motivation 1

2 The Standard model of Particle Physics 3
- 2.1 Fermions 4
- 2.2 Bosons 6
- 2.3 Conservation laws 6
- 2.4 Cabibo-Kobayashi-Maskawa Matrix 7
- 2.5 The unitary triangle 8
- 2.6 Υ-resonances produced in e^+e^--collisions 10
- 2.7 Time dependent CP-violation in B-meson decays 12

3 The Belle II Experiment 15
- 3.1 Motivation for the upgrade to Belle II 15
- 3.2 The SuperKEKB accelerator 16
  - 3.2.1 Luminosity 17
  - 3.2.2 The nano-beam-scheme 18
- 3.3 Experimental setup 19
  - 3.3.1 Pixel detector 20
  - 3.3.2 Silicon Vertex Detector 25
  - 3.3.3 Central Drift Chamber 26
  - 3.3.4 Time of Propagation Detector 27
  - 3.3.5 Electromagnetic Calorimeter 28
  - 3.3.6 K_L^0 and μ detection 29
- 3.4 Global Belle II data acquisition (DAQ) system and trigger 30
  - 3.4.1 Trigger 30
  - 3.4.2 Belle II DAQ 31

3.5 Data reduction with the ONSEN-system 33

4 Generating, unpacking and monitoring of clustered PXD data 37
4.1 The Belle II analysis framework - basf2 38
4.2 Data flow - processing of cluster data in basf2 39
4.3 Data formats 41
4.3.1 Outgoing ONSEN frames 41
4.3.2 DHC sub event formats 43
4.3.3 DHE event formats 45
4.3.4 DHE cluster format 47
4.4 Generation of test data 48
4.5 Cluster related modules and classes 48
4.5.1 PXD Unpacker Module 49
4.5.2 Raw cluster class 49
4.5.3 Hardware cluster unpacker Module 49
4.5.4 Hardware cluster class 50
4.5.5 Data Quality Monitoring 50

5 Data consistency check 51
5.1 Generation of test patterns 51
5.1.1 Square clusters 52
5.1.2 V-shaped clusters 53
5.1.3 Discussion of the test results 54
5.2 Summary and Conclusion 60
5.3 Outlook 62

6 Appendix 63

Bibliography 85

Chapter 1

Motivation

In order to motivate this work, I will start with the discoveries of the former Belle experiment which was the predecessor experiment of Belle II, on which this thesis is written. During its 11 years of data taking Belle accumulated an enormous amount of B-meson decays which are analyzed until today. Two of the discoveries made at Belle were the X(3872), which is an exotic charmonium state, and the measurement of CP-violation in B-meson decays (chapter 2.7), to name only the most cited ones. Since these processes are very rare it is necessary to have high luminosity (chapter 3.0.5.1). At Belle the peak luminosity reached $\mathcal{L} = 2.11 \cdot 10^{34} cm^{-2} s^{-1}$ in 2009 [Col13, p. 2] and still represents the world record.

In 2010 it was decided to upgrade the Belle experiment to Belle II which will start taking data in 2018. Since most of the phenomena that will be studied at Belle II are related to extremely rare decays one needs to accumulate an enormous amount of data. This can be realized with an upgrade of the KEKB collider ring to SuperKEKB, with a design luminosity of $\mathcal{L} = 10^{38}$ which is 40 times higher than the one at Belle. Due to this high luminosity, B-meson decays which violate CP-conservation can be studied more detailed than in Belle. Like the former KEKB collider, SuperKEKB will be an asymmetric collider. This asymmetry is needed in order to measure the time dependent CP-violation in B-meson decays precisely. Due to the asymmetric collision energies of electrons (7GeV) and positrons (5GeV) the B-meson system will gain a Lorentz-boost in the laboratory system. In case of CP-violation, both B-mesons will decay at different times which can be measured as spatial separation. For high resolution purposes a pixel detector (PXD) (chapter 3.3.1) will be installed.

The PXD consists of approximately 8 million pixels made out of semi conducting material that can measure a particles energy deposition. To dissolve the separation of decay vertices precisely the PXD is installed close to the Interaction Point (IP). Since the PXD is the innermost Belle II detector it faces high background originating from Touschek scattering, e^+, e^- pair production or radiative bhabha scattering [DU10, p.66] leading to a data rate of up to $20GB/s$. Therefore a complex data acquisition (DAQ) system that reduces the amount of stored data by filtering physically interesting events was invented (chapter 3.5).

During readout, data transmitted from the PXD will be processed in several steps of the readout chain. Each step will add an additional header to the data format with important information for the analysis of this event. The innermost data format refers to single pixels or cohesive pixel structures; the clusters. They occur when a particle deposits energy in neighboring pixels which have at least one common edge. In case of clusters the format contains information about all pixel inside the cluster. In this thesis, the PXD cluster format unpacking and decoding was implemented into the Belle II analysis framework (basf2) and tested for the first time.

This thesis will start with a short introduction to the Standard Model of particle physics (SM) and some phenomena which will be studied at Belle II. After that the Belle II experiment will be discussed starting with a short explanation of the SuperKEKB collider upgrade before giving an overview of the detectors that will be installed. A chapter dealing with generating, unpacking and monitoring of PXD cluster data will follow. There, a detailed overview of the data formats in use and the code written in order to finish this thesis is given. At the end of this chapter results of a data streaming and consistency test shall be discussed. The last chapter gives an outlook of what has to be done in order to test the code with the whole readout chain.

Chapter 2

The Standard model of Particle Physics

The standard model of particle physics (SM) is a quantum-field theory which describes the structure and forces of the universe. So far there is no experimental result which is contradicting SM predictions, which is why the SM is extremely successful. Still there are problems with the theory originating from the fact that there are 18 ”free” parameters which can not be calculated from theory and therefore have to be measured in experiments. Another problem is the absence of gravitation in the SM. Since the gravitational force at small distances is extremely weak compared to the other three forces of nature, in the present energy regions it is not really important. Till now there is no way to include gravitation, so the SM only deals with the three remaining forces of nature:

- **strong force:** The strongest of all four forces is described in quantum chromodynamics (QCD) [SB11, p. 107] and appeals only between color charged objects. It is mediated by gluons which carry color and anti-color charge. Since there are three colors and three anti-colors one can combine 9 different gluons. The ninth combination combination corresponds to a colorless object. Since this object wouldnt interact, due to the lack of color charge, it could travel infinitely. This can not be due to the fact that atoms have, which are confined via the strong force, no infinite potential.

- **weak force:** The weak interaction was discovered during studies on the β-decay of nuclei [SB11, p.103]. It can interact between two quarks, two leptons or quarks and leptons. There are some processes where only the weak force can interact for example all processes involving neutrinos. The SM describes the weak force as interchange of $W^{\pm}$ or Z^0 bosons.

- **electromagnetic force:** This is, besides gravitation, the most obvious force in our daily life. Electromagnetism is mediated via the photons, particles taking part in electromagnetic reactions need to carry charge, which is a conserved quantity in the SM.

As described before the forces are transmitted via particles, the so called gauge-bosons. Besides them there are some other particles which are elementary and build up all composed particles described in the SM. An illustration of these is shown bellow:

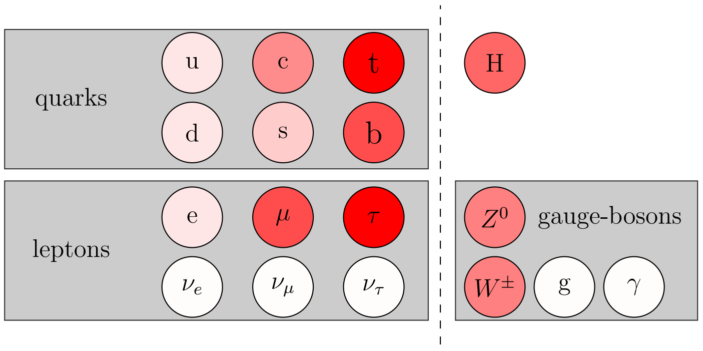


Figure 2.1: **Left:** SM particles which build up the known matter of the universe. **Right:** The gauge-bosons which transmit the forces and the Higgs-boson which introduces mass are drawn. The red shades symbolize the mass differences between these particles.

2.1 Fermions

Most of our surrounding matter is made out of these elementary particles called fermions. They all carry a half integer spin and split up into two families which are made up of three different generations according to their increasing masses. The biggest difference between these two families is the way they interact with the forces of nature.

- **Leptons:** Leptons cannot interact via the strong force since they do not carry color charge. One can further categorize them into charged and neutral particles. The charged ones, e, μ and τ can interact with the weak and the electromagnetic force. In addition to these three generations of charged leptons there are three corresponding generations of neutral leptons, the so called neutrinos; ν_{e^-}, ν_μ and ν_τ. Since they do not carry electric or color charge they can only interact via the weak force which puts high requirements on the detectors of neutrino experiments.

- **Quarks:** Quarks are the only fermions which interact via all forces of nature and, besides the gluons, the only elementary particles which can interact via the strong force. Due to this fact they have to carry electromagnetic and color-charge. According to the three generations of leptons the SM includes three generations of quarks with two different flavors for each of them. The first and lightest generation consists of up, and down-quark (u, d), the second of strange, and charm-quark (s, c) and the third of bottom, and top-quark (b, t) ordered according to the increasing masses.

  A very important property of quarks is the confinement. It describes the absence of free quarks in the SM where quarks can only exist in bound states of quark and anti-quark (mesons) or three quarks or anti-quarks (baryons). Theoretically there are also states with four, five or even more quarks possible and under discussion. Some candidates for these states where also found experimentally (X,Y,Z-states) [ea03][eaBC05][eaBC13] but still physicists are not sure what they

- **Baryons:** In comparison to the other fermions, baryons are composed of elementary particles, in particular three quarks, but also carry half integer spin which classifies them as fermions. Combining three quarks with spin $s = 1/2$ leaves you with 2 possibilities for their combined spin in the ground state, $s = 1/2$ or $s = 3/2$.

  Like leptons also the baryons have their own quantum number, 1 for baryons, -1 for anti-baryons, 1/3 for quarks, -1/3 for anti-quarks and 0 for all other particles. The baryon number is an absolute conserved quantity in the SM.

2.2 Bosons

In the SM, bosons can be devided into gauge-bosons which transmitt the forces and the Higgs-boson introducing mass to all other particles. In addition there are not-elementary bosons, the mesons, which consist of a quark anti-quark pair. All bosons have integer spin.

- **Gauge-Bosons:** These are all elementary spin 1 particles transmitting the forces described in the SM. Photons (γ) transmit the electromagnetic force, gluons (g) are the interacting particles of the strong force, W-Bosons ($W^{\pm}$) and Z-Bosons (Z) transmit the weak interaction, the W are charged and the Z are neutral.
- **Higgs-Boson:** These particles introduce mass to the other SM-particles. Like the $W^{\pm}$ and Z they are unstable particles which carry mass ($m_H \approx 125GeV$). The Higgs was first predicted from Peter Higgs in 1964 [Hig64] and discovered at CERN in 2012 [Col12, p.1-29].
- **Mesons:** There are plenty of other bosons which can be distinguished from the gauge-bosons by the simple fact that they are not elementary and therefore a combination of elementary particles. What unites both is their integer spin. A huge boson family are the so called mesons. They consist of a quark and an anti-quark so their total spin adds up to 0 or 1 in the ground state.

All the particles described in the SM have their own anti-particle which is identical in mass but switches charge, lepton or baryon number to the exact opposite. Particles like the γ or the Z are there own anti-particles.

2.3 Conservation laws

Like in all other fields of physics conservation laws are also very important for particle physics. Besides the classical ones like energy, momentum or angular-momentum conservation the SM needs additional ones which are related to the quantum numbers of particles and states. Three of them shall now be briefly discussed.

- **Parity conservation:** Parity conservation predicts that, if a physical process is taking place in a completely mirrored world, it would still elapse the same way. Parity is conserved in strong and electromagnetic processes but violated in weak interactions. In 1956 Chien-Shiung Wu proved the existence of parity violation experimentally [LEE57] which was only theoretically predicted before [LY56, p.254] [HH57].

- **Charge conjugation:** This process describes the transformation of a particle in its anti-particle and therefore switches its properties like charge, lepton number, baryon number or color charge. Like parity, charge conjugation is conserved in electromagnetic and strong interactions but violated in weak processes.

- **CP-violation:** CP-invariance, which was assumed to be an universal law, would suggest the outcome of a process with interacting anti-particles taking place in a mirrored world to be the same as particles interacting in the unmirrored world. In 1964 Cronin and Fitch could disprove this universality [JHCJWCVLF64, 138]. After this first evidence for CP-violation in the SM also Belle and BaBar made discoveries of CP-violation in B^0-meson decays [ea04].

Since CP violation is a fundamental process related to the imbalance of matter and anti-matter in our universe, a deeper understanding of this phenomena is an important task for both theorists and experimentalists. For that reason many experiments tried to perform better measurements with higher statistics. The next generation experiments such as Belle II will further focus on this issue [DU10].

2.4 Cabibo-Kobayashi-Maskawa Matrix

To understand CP-violation one needs to study the CKM-Matrix, an extension to the 2×2 Cabibbo-Matrix which describes a rotation around the Cabibbo-angle and was first introduced by Nicola Cabibbo in 1963 [Cab63]. This rotation is physically related to the mixing of quark generations. These new states are mixtures of the quark mass eigenstates which couple to the weak force.

In 1973 Kobayashi and Maskawa published there work on CP-violation and the Cabibbo-matrix which is since then known as Cabibbo-Kobayashi-Maskawa-

Matrix [uTM73, p. 652]. The new CKM-Matrix was now also including the recently discovered quark generations b and t along with a complex phase giving rise to CP-violation. A convenient and popular way to write down the CKM-matrix is:

$$M_{CKM} = \begin{pmatrix} V_{ud} & V_{us} & V_{ub} \\ V_{cd} & V_{cs} & V_{cb} \\ V_{td} & V_{ts} & V_{tb} \end{pmatrix} \tag{2.1}$$

If one knows the values of each element, where the diagonal ones are almost one and the rest almost zero, the transformation probability of quarks proportional to the square of the corresponding matrix element. The entries represent free parameters of the SM which cannot be derived from theory [SB11, 219] and have to be measured in experiments. Determining the elements of this matrix gives rise to a specific probability sequence for quarks changing their flavor, which is illustrated in the picture below:

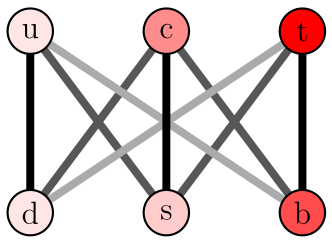


Figure 2.2: Transition between quarks will be more suppressed on light gray lines and more likely on black lines. The red shades indicate the mass difference between these quarks.

2.5 The unitary triangle

The unitary triangle is a direct consequence of the CKM-matrix, which is itself unitary and therefore $M \cdot M^{\dagger} = I$, where I is the unitary matrix is fulfilled. With this relation one can derive the following equation:

$$V_{ud}V_{ub}^{*} + V_{cd}V_{cb}^{*} + V_{td}V_{tb}^{*} = 0$$

This can be illustrated as vectors drawn in the complex plain.

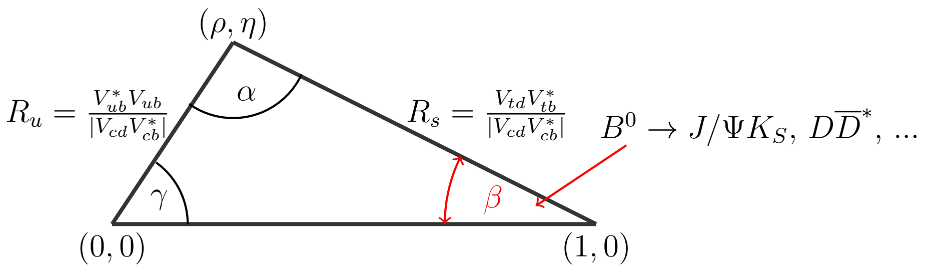

(ρ,η)
$R_u = \frac{V_{ub}^* V_{ub}}{|V_{cd}V_{cb}^*|}$
α
$R_s = \frac{V_{td}V_{tb}^*}{|V_{cd}V_{cb}^*|}$
$B^0 \to J/\Psi K_S,\ D\overline{D}^*,\ ...$
γ
β
$(0,0)$
$(1,0)$

Figure 2.3: The unitary triangle in the complex-plain. β can be derived from the decay channels next to its corner [SB11, 359].

The lines not located on the real-axis R_u and R_s are normalized to map the triangle between zero and one. Through the measurement of α, β and γ one can directly calculate the sites of the triangle which are related to the probability of quark-transformations. The upper point (ρ, η) can also be calculated with the angles and gives rise to CP-violation in the SM.

As already mentioned CP-violation is a fundamental process in order to explain the imbalance of matter and anti-matter in the universe. After the big bang one could naively think that matter and anti-matter should be distributed in equal densities over the whole universe. Taking a look at our environment this cant be the case! To explain these imbalance three criteria, first introduced from Andrei Sakharov [Sak67], are necessary.

- **Baryon number violation**
- **Thermal non-equilibrium**
- **CP violation, possibly accompanied by C violation**

Until now there is no experimental result for baryon number violation, nevertheless it is one of the three fundamental conditions related to the matter anti-matter imbalance and therefore needs to be possible under certain conditions. These conditions were achieved directly after the big bang when the energy density was high enough. In so called Grand Unified Theories (GUT) the forces of nature were combined and had the same couplings to every particle during this first 10^{-37} seconds. All processes were allowed which leads to the prediction of leptoquarks, states of combined quarks and leptons. When these states where decaying the products could be either pure hadronic or as well a combination of quarks and leptons. The probability to decay into a pure hadronic state is slightly higher, which after some time leads to an imbalance of matter and anti-matter.

Thermal non-equilibrium in the primordial universe propably occured due to a very fast expansion (inflation, Alan Guth, 1981 [Gut81]). In the inflation phase, distances between particles increased too rapidly, so that scattering processes could not ensure thermal equilibrium anymore.

The last criterion, CP-violation, was already described on page 7. Since the impact of CP-violation in K-systems is too weak to contribute alone to the third Sakharov criterion, there has to be another source of CP-violation. Therefore physicists began to investigate B-systems. An explanation of how this works is given in the next chapters.

2.6 Υ-resonances produced in e^+e^--collisions

Since leptons do not couple to the strong force, production of particles in e^+e^--collisions is transmitted via the weak or electromagnetic force. In case of produced quarks, the final state is a bound hadronic state of quark and anti-quark, a meson.

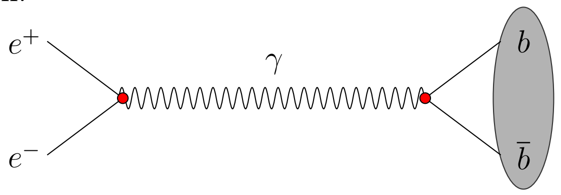


Figure 2.4: Electron (e^-) and positron (e^+) interacting via the electromagnetic force producing a virtual photon (γ) which then forms a confined bottom, anti-bottom quark state.

Figure 2.4 shows a Feynman diagram where electron and positron annihilate to form a $b-\overline{b}$-state via virtual photon exchange. The produced $b-\overline{b}$-states can be plotted as production cross section ($\sigma(e^+e^- \rightarrow hadrons)[nb]$) as function of the mass in GeV which is shown in figure 2.5.

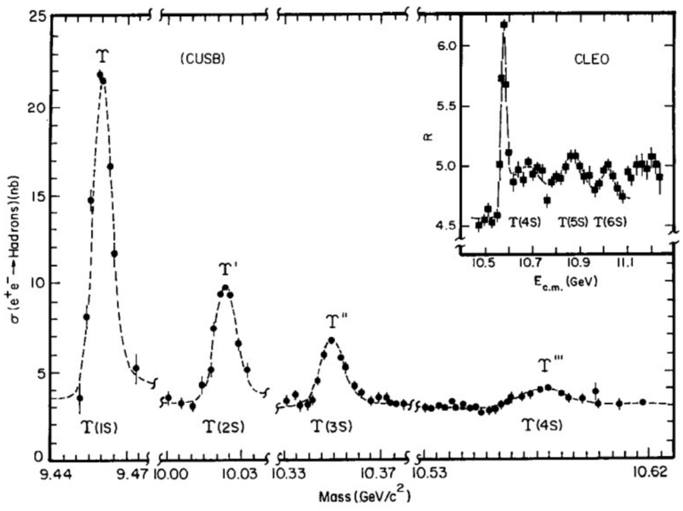


Figure 2.5: The cross section of electron positron annihilation to hadrons ($\sigma(e^+e^- \rightarrow Hadrons)$) is plotted against the mass in GeV. The peaks correspond to the indicated Υ-resonances up to the $\Upsilon(4S)$. [Fra78]

As one can see in figure 2.5, the $\Upsilon(4S)$ mass is slightly heavier than the combined mass of a $B^0 - \overline{B^0}$-system which adds up to $m_{B^0 - \overline{B^0}} = 10.56 GeV$. For that reason the $\Upsilon(4S)$ decays to $> 96\%$ into $B^0\overline{B^0}$. Due to this "open beauty" decay, the width of the $\Upsilon(4S)$ is very broad compared to the 1S, 2S and 3S state corresponding to a short life time.

Since one of the main research topics of Belle II will be the measurement of CP-violation in B decays, the $\Upsilon(4S)$-resonance is a good candidate to gain a huge data sample of B-mesons which is essential for the upcoming precision studies. A Feynman diagram of $\Upsilon(4S)$ decaying into B mesons is shwon in figure 2.6.

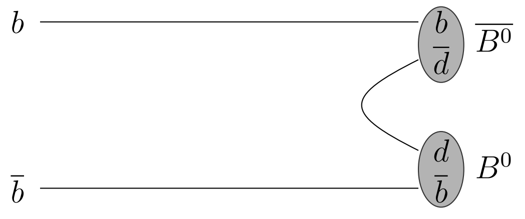
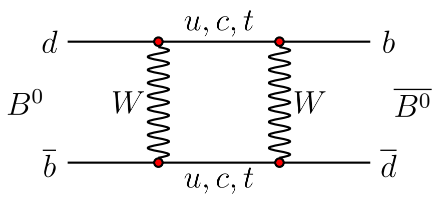


Figure 2.6: $\Upsilon(4S)$ decaying into $\overline{B^0}$ B^0.

2.7 Time dependent CP-violation in B-meson decays

As mentioned in section 2.5 the angles of the unitary triangle can be measured in experiment and are related to the CP-violating phase of the CKM matrix. The main measurements at Belle have been performed in B^0-decays and investigated the β-angle. These measurements are related to the time difference between the B^0 and the $\overline{B^0}$ decay. In the SM this effect can be described as interference of direct and indirect CP-violation. The $B^0-\overline{B^0}$ mixings responsible for indirect CP-violation are shown in the following Feynman diagrams.



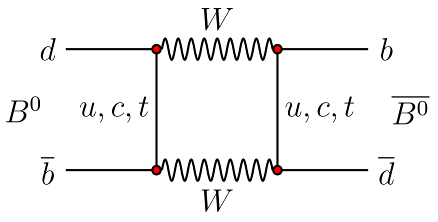


Figure 2.7: Feynman diagram showing the mixtures of $B^0\overline{B^0}$-mesons via intermediate W-bosons and quarks.

Belle measured the time dependent CP-violation via the process illustrated in the figure 2.8.

The time value t can be measured by tagging the primary decaying B^0-meson and reconstructing the second by its decay products. The difference of this time value in B^0 and $\overline{B^0}$ decays can then be plotted which is shown in figure 2.9.

The amplitude of the difference of the two curves in 2.9 is proportional to $\sin(2\beta)$, where β is the angle from the unitary triangle which determines the position of (ρ, η) in the complex plane. The value of $\sin(2\beta)$ is confirmed to be 0.731 ± 0.056 [Ber06, p. 433] which is in good comparison to the predicted SM value. Nevertheless a better measurement of this value, which can be achieved by accumulating more data, would proof the SM prediction to be right or reveal some new physics if the value differs from the predicted one. To further investigate this phenomena will be one research focus of the future Belle II experiment.

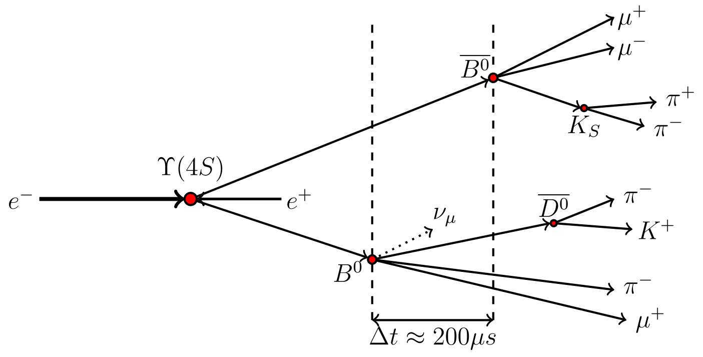


Figure 2.8: Electron and positron annihilating and forming a $\Upsilon(4S)$ which then decays to a B^0 and $\overline{B^0}$. The first decaying particle is then tagged and identified as B^0. At this time the other B-meson has to be an $\overline{B^0}$. If the $\overline{B^0}$ decays to $K_S J/\Psi$ the CP-violating V_{td} matrix element from formula 2.1 contributes to the calculation on the decay-vertex. In the above decay the J/Ψ can be reconstructed from $\mu^+\mu^-$.

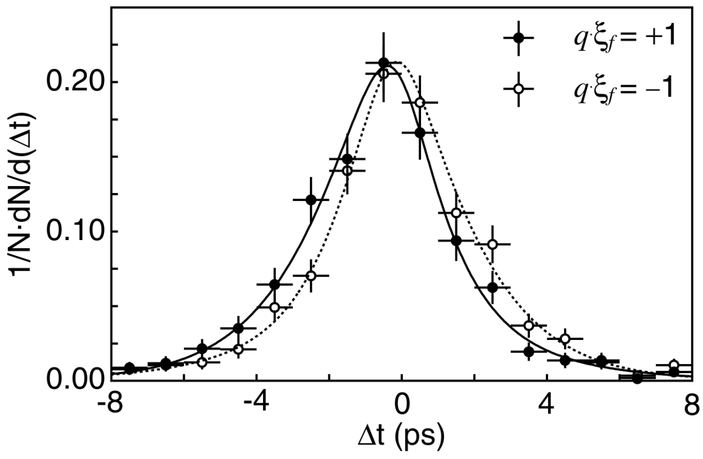


Figure 2.9: Distribution of the time value Δt for B-decays. On the y-axis the derivative $\frac{dN}{d(\Delta t)}$ of the number of decays N to the time value Δt is normalized to the number of decays $\frac{1}{N}$. The x-axis shows the time value Δt in pico seconds ($10^{-12}s$). B^0 decays are shown as $q\xi_f = +1$ (solid line) and $\overline{B^0}$ are shown as $q\xi_f = -1$ (dotted line) [Abe01].

Chapter 3

The Belle II Experiment

This section will give an introduction to the future Belle II experiment, its detectors and the SuperKEKB accelerator which provides the collisions. All informations given in this section are taken from the Belle II technical design report. Since the experiment will first run in 2018 small changes in hard and software are still possible but the overall experimental setup will remain as described here.

3.1 Motivation for the upgrade to Belle II

Belle, which was hosted at the KEKB accelerator in Tsukuba Japan, was incredibly successful during its 11 years runtime. KEKB was an antisymmetric electron positron collider which was running at a center of mass (CMS) energy of $E_{CMS} = 10.58 GeV$, the mass of the $\Upsilon(4S)$-meson. Since this resonance is dominantly decaying into B mesons the whole facility belongs to a sort of particle colliders which are known as B factories. In 2009 Belle accumulated a total integrated luminosity of $\mathcal{L} = 952 fb^{-1}$ which is until now the world record. With an upgrade of collider and detector one can reach much higher precision in the measurement of rare SM effects and may also have the possibility to study effects which correspond to physics behind the SM. For that reason it was decided in 2010 that KEKB and Belle should be upgraded to SuperKEKB and Belle II respectively.

3.2 The SuperKEKB accelerator

The Belle II experiment will be hosted at the SuperKEKB accelerator at KEK institute Tsukuba, Japan. This asymmetric ring collider is an upgrade of the former KEKB accelerator which was providing collisions for Belle. Since Belle II is mainly investigating very rare processes one needs high statistics. This can be achieved with a high collision rate and large data samples which have to be handled by the hard- and software of the detectors. The first run of Belle II will most probably run on the $\Upsilon(5S)$ energy and later go down to $\sqrt{s} = 10.58 GeV$ which is the mass of the $\Upsilon(4S)$ since this resonance is decaying dominantly ($> 96\%$) into B-mesons [Gro14, p. 142].

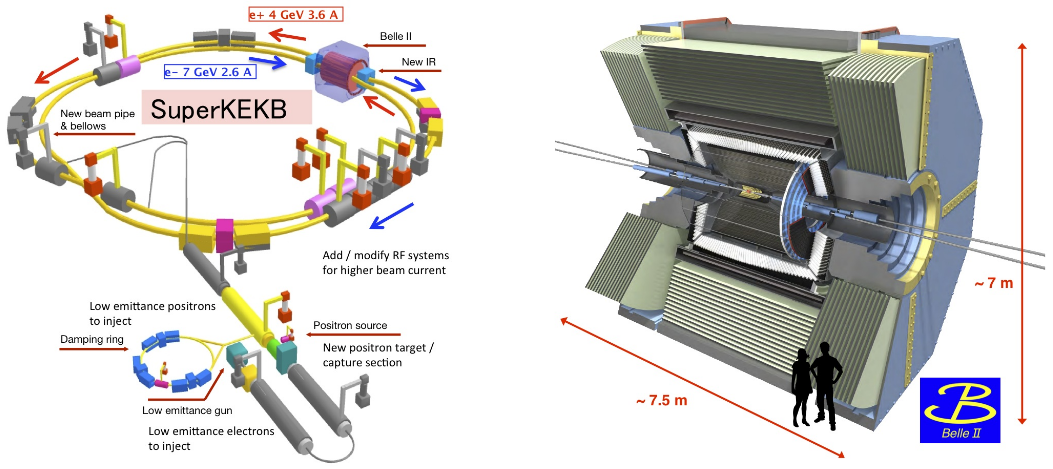

e+ 4 GeV 3.6 A
Belle II
New IR
e− 7 GeV 2.6 A
SuperKEKB
New beam pipe & bellows
Add / modify RF systems for higher beam current
Low emittance positrons to inject
Damping ring
Positron source
New positron target / capture section
Low emittance gun
Low emittance electrons to inject
~ 7 m
~ 7.5 m
Belle II

Figure 3.1: **Left:** Sketch of the SuperKEKB accelerator. **Right:** The Belle II detector [Ins10].

3.2.1 Luminosity

One of the most important parameters to characterize a particle collider is the Luminosity $\mathcal{L}$, which is schematically shown in figure 3.2. It is defined as the number of particles (N_1, N_2) interacting with each other in a certain region (cross section $\rho_x, \rho_z[cm^2]$) per unit time $(f[1/s])$.

$$\mathcal{L} = f\frac{N_1 N_2}{4\pi\rho_x\rho_z}\left[\frac{1}{cm^2 s}\right] \tag{3.1}$$

It is also directly related to the event rate N:

$$\tfrac{dN}{dt} = \mathcal{L}\cdot\sigma \quad\Longrightarrow\quad N = \sigma\int\mathcal{L}dt$$

N, which is the cross section times the luminosity integrated over time, represents the total amount of events collected in a certain period of time.

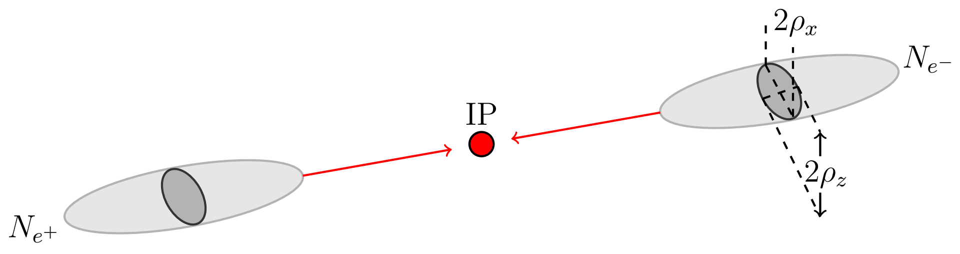


Figure 3.2: Two particle bunches one with N_{e^+} positrons the other with N_{e^-} electrons collide at the IP. The interacting beam cross sections ρ_z and ρx indicate the indicate the effective interaction areas of the beams. Luminosity can then be calculated according to formula 3.1

Up to now the world record in luminosity $\mathcal{L} = 2.11\cdot10^{-34}cm^{-2}s^{-1}$ is still held by the former Belle experiment. In order to increase this by a factor of ~ 40 for the future Belle II experiment one can tune three parameters in formula 3.1, the frequency f, the number of particles N_1, N_2 and the bunch cross section $\rho_x\rho_z$. For Belle II it was decided to decrease the bunch cross sections which is described in the next chapter.

3.2.2 The nano-beam-scheme

The future SuperKEKB collider is build for a peak luminosity of $\mathcal{L} = 80 \cdot 10^{-35} cm^{-2} s^{-1}$. To gain this increase of a factor 40, of the former Belle luminosity, one can tune the three parameters of formula 3.1. For Belle II the bunch cross section $\rho_x \rho_z$ is tuned.

The parameters $\rho_x \rho_z$ can be described as $\rho_i = \sqrt{\epsilon_i \beta_i}$, where ϵ_i is the emittance e.g. the spread of particle coordinates in position and momentum phase-space and β_i the beta function related to the transverse size of a particle beam. The emittance ϵ_i can be derived from $\epsilon = d\phi \cdot dq$, here $d\phi$ is the angle divergence and dq the cross section area of one beam. According to the Liouville-Theorem [Sch06], which implies that the phase-space distribution function is constant along a systems trajectories, the emittance is a conservation value.

With this knowledge increasing luminosity comes with decreasing the beta function. This happens via the relation $\beta_z < d$ which connects the beta function with the beam overlap d. The following scheme illustrates the crossing of two particle bunches at the IP.

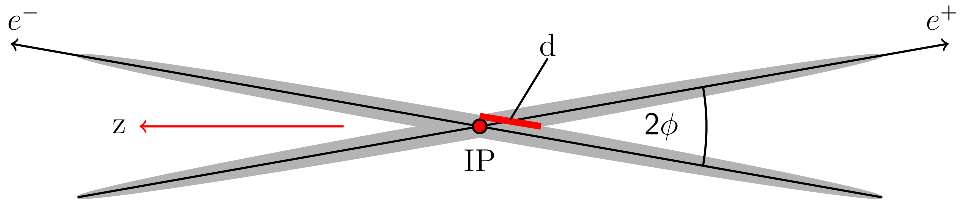


Figure 3.3: The picture illustrates the bunch crossing of electron (in positive z-direction) and positron beams at the IP. d, which indicates the effective overlap length, limits the magnitude of the β-function at the interaction point.

As one can see in figure 3.3 a smaller d is induced by increasing ϕ. At the Belle II IP ϕ will be $41.5 mrad$ corresponding to $\sigma_x = 10.2 \mu m$ for electrons and $\sigma_z = 7.75 \mu m$ for positrons which leads to the proposed luminosity. Different values of σ_x for positrons and electrons are related to the different beam energies $E_{e^-} = 7 GeV$ and $E_{e^+} = 4 GeV$.

3.3 Experimental setup

The Belle II detector consists of 6 sub-detectors which can be divided in three fields of functionality; particle identification, track reconstruction and energy measurement. This chapter will first describe the two types of detectors mentioned and then give an overview of each sub detector mainly concentrating on the pixel detector (PXD) which is topic of this work.

- **track reconstruction:** The main purpose of this detector type is the reconstruction of particle trajectories. Belle II will host three detectors of this type, the PXD, the Silicon Vertex Detector (SVD) and the Central Drift Chamber (CDC). Combined they will cover the polar angle region from $17° < \Theta < 150°$.

- **particle identification:** These detector type performs identification of particles and is represented by the Time Propagation Detector (TOP), the Electromagnetic Calorimeter (ECL) and the K_L^0 and μ detector (KLM). These three form the outermost detector layers and will be surrounded by the magnet-yoke.

- **energy measurement:** Energy measurement is performed by the ECL and KLM which measure the energies of photons, kaons, electrons and muons which are produced in high numbers at Belle II.

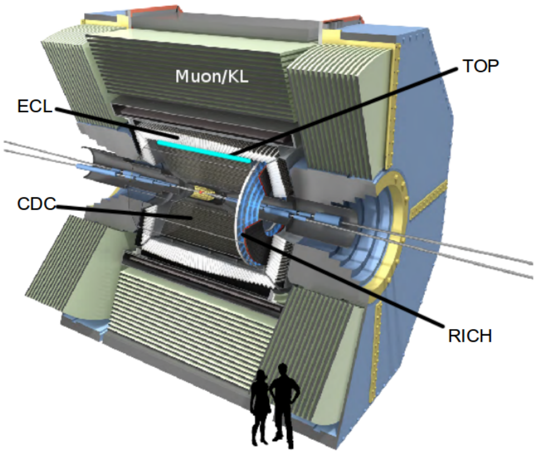


Figure 3.4: Assembling of the Belle II sub detectors inside the magnet-yoke [DU10]. The PXD and SVD can be seen in yellow and red directly around the IP.

3.3.1 Pixel detector

Consisting of two layers placed 14mm and 22mm around the IP, the PXD is the innermost Belle II detector. Together with the SVD the PXD forms the Vertex Detector (VXD). It's task is to reconstruct the vertices of particles decaying near the IP. Picture 3.5 shows the layout of the PXD.

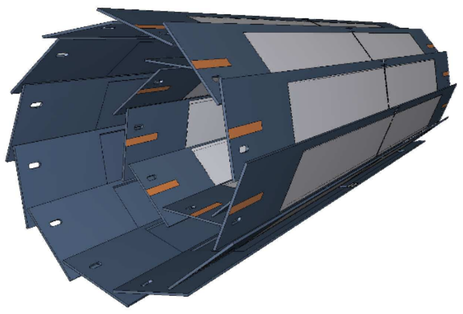


Figure 3.5: Setup of the PXD, consisting of inner and outer layer with 16 and 24 half-ladders respectively. The gray colored rectangles indicate the sensitive detector area [DU10, p.77].

As already mentioned the main focus of Belle II will be the investigation of processes related to B-meson decays. The reconstruction of these decay vertices is, due to the high occupancy more or less impossible with strip detectors. For the PXD pixels are used instead of stripes which comes with a much higher number of channels and therefore less occupancy. Still one needs the SVD, which is placed 38mm around the IP, to reconstruct vertices with sufficiently high resolution. This combination of pixel and strip detectors was successfully implemented at the LHC. Still the Belle II PXD will set new standards since the layers of the LHC-PXD with $200 - 250\mu m$[Kei01, p.40] would be too thick for the measurements performed at Belle II.

The thickness of the sensitive layers is a major criterion for the behavior of a particle when flying through the sensor. If the material is too thick, it will cause multiple scattering which makes the reconstruction of decay-vertices impossible. For this reason the sensitive area of the Belle II PXD consists of a new transistor type, the DEpleted P-channel Field Effect Transistor (DEPFET), which can be thinned down to $75\mu m$.

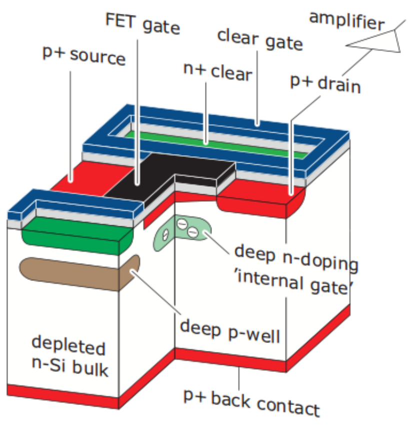


Figure 3.6: Scheme of a DEPFET transistor forming one pixel of the PXD [DU10, p.79]. In the depleted silicon bulk (white) electrons and holes are produced by particles flying through. These electrons and holes are then propagating to the oppositely charged poles deep p-well and p+ back contact respectively.

As shown in the previous picture, the major part of a DEPFET pixel consists of depleted silicon bulk which is grounded on a positive contact and topped with a p-channel MOSFET (Metal Oxide Semiconductor Field Effect Transistor) or JFET (Junction gate Field-Effect Transistor). The positive ground-plate causes full electron depletion in the silicon which in turn charges the p-channel ground-plate negative. Beneath the FET gate a potential minimum evolves which accumulates the energy set free by a particle ionizing the silicon-soil when flying through the DEPFET. The amount of energy deposited can be approximately derived from the Bethe-Bloch formula, which reflects the energy transfer between a charged particle and the matter it is flying through. This ionization process causes electrons and holes in the silicon bulk which then propagate to the oppositely charged poles. A potential difference between deep p-well, where electrons gather and the p+ back contact, where the holes are accumulated is the result. This leads to a channel current which is directly proportional to the energy loss of the particle passing the DEPFET pixel.

3.3.1.1 PXD readout

As shown in figure 3.5 the whole PXD will be made up of 40 half-ladder. Each containing 768×250 of the discussed DEPFET pixels corresponding to a total amount of ≈ 8 million pixels for the whole PXD. A scheme of such a half ladder is shown in figure 3.7.

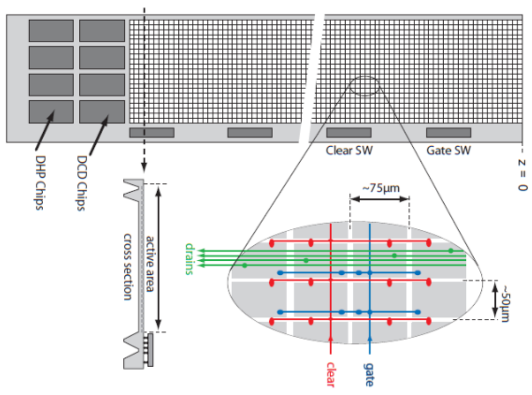


Figure 3.7: Scheme of a PXD half-ladder [DU10, p.90]. On the left the four DHP and DCD, which read out this half-ladder, are located. In the lower picture one sees a cross section cut of the half-ladder and how gate, clear and drain is connected between the single pixels.

Pixels are read out by four DCDs (Drain Current Digitizer) combined with four DHPs (Data Handling Processor). The DCDs digitize the currents so that the DHP can produce a data format (discussed in chapter 4.3) which can then be processed. In order to minimize readout time the whole half-ladder is segmented in four parts with 768 rows and 64 columns. Each of these segments is read out by one DCD-DHP pair. This is perfomred column by column in blocks of four rows and is regulated by the switches shown on the bottom of the half-ladder in figure 3.7. The readout of one half-ladder will take $20\mu s$ which corresponds to a readout rate of $50kHz$.

Four of these DHP correspond to one DHE (Data Handling Engine) which is responsible to send the data of one half-ladder to the DHC (Data Handling Concentrator). The DHC combines data of five DHE and sends it out via four optical links to the ONSEN (ONline SElection Node) system. This system represents the core of the PXD's data reduction and will be discussed in chapter

(3.5). As mentioned before with an occupancy of $\approx 3\%$ mostly originating from background effects the PXD DAQ (Data AcQuisition) will be faced with an enormous amount of data which cannot be completely stored and therefore needs to be reduced. These background sources are:

- **Synchrotron radiation (SR) from upstream:** Synchrotron radiation originating from the high energy beam. The IP chamber is designed to avoid direct SR hits from the high energy beam.
- **Backscattering of SR from downstream:** This background source originates from the bending magnet which separates the beams behind the IP. At SuperKEKB a much lower SR from downstream than at KEKB is expected.
- **Beam gas scattering:** Bremsstrahlung and Coulomb scattering hanging the momenta of beam particles which then hit the walls of vacuum chambers and magnets. This produces shower particles which are one of the major beam-induced background sources.
- **Touschek scattering:** An effect of intra-bunch scattering which can force particles to hit vacuum chamber and magnet walls producing shower particles.
- **Radiative Bhabha scattering:** Interacting electrons and positrons produce photons which propagate along the beam axis and interact with the iron of the magnets. In these interactions neutrons are produced via the giant photo-nuclear resonance mechanism [ea08]. These photons are the main background source for the KLM.
- **Electron-positron pair production via two photon process:** Electron-positron pairs are produced in two-photon processes $e^+e^- \to e^+e^-e^+e^-$. This is the major background source for the PXD which is faced with 900 to 14000 e^+e^- pairs in each event in the first layer, based on monte-carlo studies [DU10, p. 67].

To reduce the data in the ONSEN-system so called ROIs (Regions Of Interest) are selecting physically interesting data which does not originate from these background effects. These ROIs are defined as areas on the PXD surface which contain physically interesting information related to trajectories calculated from the High Level Trigger (HLT). If the ONSEN-system receives data for a given ROI this data will be kept, otherwise it will be discarded.

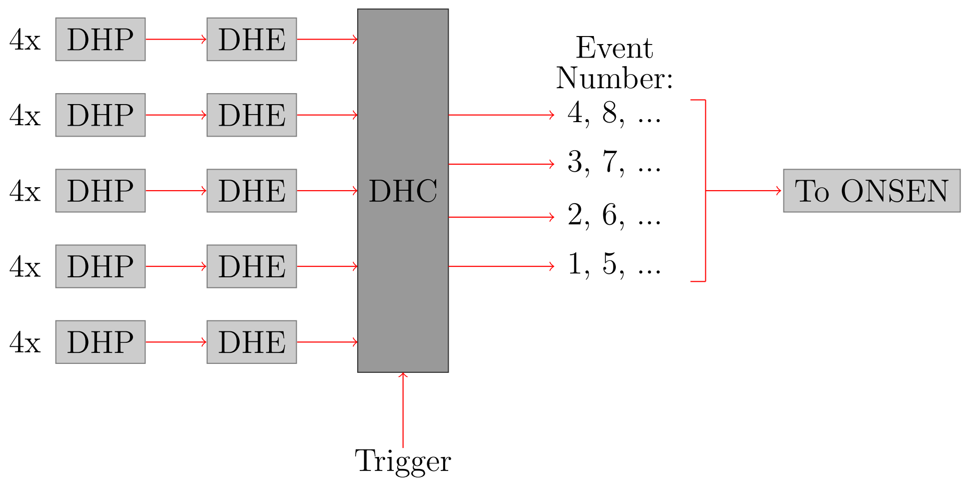


Figure 3.8: The picture shows the readout of 5 PXD half-ladder (DHE) which transmit their data to one DHC. Data is then ordered in event number and further transmitted to the ONSEN-system.

3.3.2 Silicon Vertex Detector

The second innermost detector and outer part of the VXD is the SVD, a strip detector consisting of p-doped silicon stripes on one side and orthogonally arranged n-doped stripes on the other. These stripes are assembled on ladders which surround the IP at increasing radii. The first SVD layer consists of 7 ladders, the second of 10, the third of 12 and the fourth and last of 16. Main purpose of the SVD is the reconstruction of particle tracks flying through its layers. The DATa CONcentrator (DATCON), which gets the SVD data after it's readout chain, calculates ROIs for the ONSEN-system.

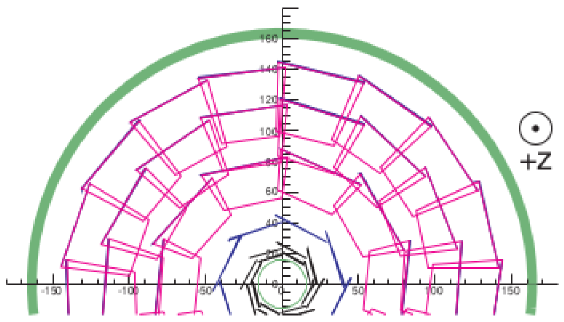


Figure 3.9: Illustration of the VXD surrounding the IP viewed in negative z direction. In blue and purple the SVD layers in black the PXD layers [DU10, p.142].

As in case of the PXD, detection of particles in the SVD is based on ionization. Electrons and holes will propagate to the oppositely charged strip and induce a current which can be read out. In case of one particle flying through one SVD strip in a certain time window a distinct assignment of the position can be made. If there are two or more particles flying through the same strip in the same time interval the positioning is not distinctive anymore due to so called ghost hits illustrated in figure 3.10.

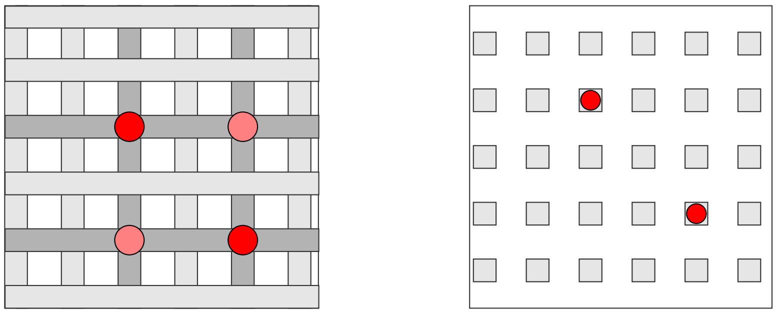


Figure 3.10: **Left:** Ghost hits (light red dots) produced on the SVD layers. Dark red dots indicate the real hits. **Right:** Avoidance of these ghost hits on the PXD.

3.3.3 Central Drift Chamber

The VXD is surrounded by the CDC, responsible for tracking and momentum measurement. In addition it provides triggers and particle identification for charged particles flying through. The chamber is filled with 50% He and 50% C_2H_6. Charged particles will ionize the surrounding gas-mixture and produce electrons and ions. The electrons will gain additional velocity due to the electric field obtained in the chamber and further ionize the gas molecules. This causes an avalanche of new electrons generating a measurable signal which is directly proportional to the amount of electrons produced by the primary particle flying through.

To measure this signal the chamber is traversed by $30\mu m$ thick wires with positive potential to attract the electrons set free by ionization. A particle track can then be calculated via the distance d between the track of the incoming particle and the wire which detects it. This distance is related to the time electrons need to drift to the wire; the drift-time. Since the wires are very close to each other electrons will always drift to two neighboring wires leading to two different drift-times. With these drift-times, the constant drift-speed and the distance between the wires, one can calculate the distance d in which particles are flying by. With this distance one can "draw" a circle around the wire. A particle track represents one of an infinite number of tangents to this circle. To have a distinct reconstruction of a particle path one needs to calculate the distance d of many wires to the particle track. A tangent which hits all calculated circles is then describing the particles trajectory.

3.3.4 Time of Propagation Detector

The detection principle of the TOP detector is based on Cherenkov radiation which is emitted from charged particles moving in a medium faster then the speed of light in the same medium [PR14, p.371]. The angular distribution of photons emitted during this process is given by:

$$\Theta = arccos\left(\frac{1}{\beta n}\right)$$

Θ is the angle between the particles trajectory and the wave front made up of the emitted photons. $\beta = \frac{v}{c}$ is the ratio of particle speed to the speed of light and n the index of refraction of the material. Radiation is emitted in a cone around the particles trajectory, an illustration of this is shown below.

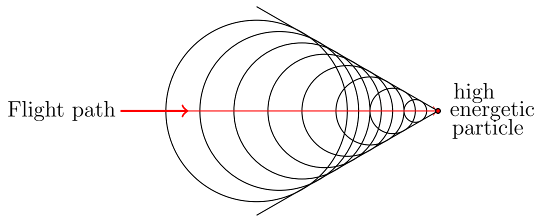


Figure 3.11: Cherenkov cone of a particle flying through a medium faster then the speed of light in the same medium. Photons are emitted perpendicular to the wave fronts.

The TOP detector at Belle II consists of a quartz-plate generating the Cherenkov radiation when a particle is flying through. One side of this plate is equipped with a micro-channel-plate photon multiplier (MPC) for photon detection. When reaching the edge of the quartz plate photons are reflected which causes them to exit on the side equipped with the MPC. In order to determine the time a particle needs to travel through the quartz one measures the time difference between the reflected Cherenkov Photons in the material. The following picture illustrates this procedure:

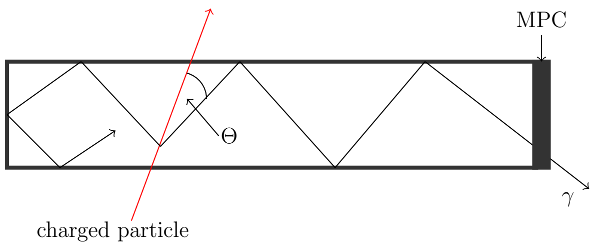


Figure 3.12: Side view of a TOP quartz-plate. The photons emitted are traveling in both ways through the quartz and get reflected each time they reach an edge until they are detected in the MPC.

3.3.5 Electromagnetic Calorimeter

In the final state of B-meson decays, which are produced at Belle II in high numbers, roughly 1/3 of the decay products are neutral particles like π^0 or K^0 leading to a huge number of photons with a broad energy spectrum ($20MeV$ - $4GeV$). In order to detect these photons with high energy precision one needs to have a high resolution electromagnetic calorimeter, which can manage the following tasks. Detection of photons and determination of their momenta and angular distribution, electron identification, measurement of luminosity via Bhabha scattering, generation of suitable triggers as well as K_L^0 detection together with the KLM.

For the detection material of the Belle II ECL major parts of the former Belle ECL will be used, the major difference to the former ECL are the readout electronics. The detecting material is made up of 8736 caesium-iodide crystals doped with thallium (CsI(Tl)). CsI(Tl) is a scintillating material. This means that incoming particles will produce photons that can be read out by photo multipliers at the end of each crystal.

The new readout electronics allow wave form analysis and suppression of fake photons detected by a factor of 7 while still keeping the efficiency for true photons at 93% or higher for photons above $20MeV$.

3.3.6 K_L^0 and μ detection

The KLM, which surrounds all previous mentioned detectors, consists of a barrel and an end cap KLM enclosed by the magnet yoke. The barrel part provides angular acceptance from 45° to 125° which is extended by the end cap part from 20° to 155°. Therefore the KLM can detect all particles which propagate through its layers under an angle of more than 20° in respect to the beam axis. It is made up of alternating iron plates and active RPC detector layers. RPC stands for glass Resistive Plate Chamber, a glass-electrode plate which provides good spatial and timing resolution. The detecting material in the end cap KLM will be renewed since the old material would only reach an efficiency of 50% due to the higher background at Belle II.

3.4 Global Belle II data acquisition (DAQ) system and trigger

This chapter gives an overview of the global Belle II data acquisition (DAQ) and trigger system. Starting with the triggers which are essential to understand the DAQ chain used at the future Belle II experiment.

3.4.1 Trigger

For the Belle II physics program there are many interesting events with triggers corresponding to their specific decay signature. These triggers lead to characterstic responses in the Belle II sub-detectors. The trigger rates and total cross sections for several physically interesting processes at the $\Upsilon(4S)$-resonance, assuming the goal luminosity of $8 \cdot 10^{35} cm^{-2} s^{-1}$ are listed in the table below. Bhabha and $\gamma\gamma$ have very large cross section, therefore their triggers are prescaled by a factor of 100 or more which is no problem due to their distinct signature. These events are also used to measure luminosity and calibrate the detectors.

Physics process	**Cross section [nb]**	**Rate [Hz]**
$\Upsilon(4S) \rightarrow B\overline{B}$	1.2	960
Hadron production from continuum	2.8	2200
$\mu^+\mu^-$	0.8	640
$\tau^+\tau^-$	0.8	640
Bhabha ($\Theta_{lab} \geq 17°$)	44	350
$\gamma\gamma$ ($\Theta_{lab} \geq 17°$)	2.4	19
2γ processes ($\Theta_{lab} \geq 17°$, $p_t \geq 0.1 GeV/c$)	80	15000
Total	**130**	**20000**

For this trigger system the following requirements are necessary:

- high efficiency for hadronic events
- maximum average trigger rate of 30kHz
- fixed latency of about maximum $5\mu s$

- timing precision of less then 10ns
- two-event separation of 200ns or better
- flexible and robust trigger configuration

The Belle II trigger system works in two steps in order to fulfill these requirements. In the first step each sub-detector provides his own sub-trigger. This is done by using Field Programmable Gate Arrays (FPGA) which provide reconfiguration support and high speed serial data transmission. Based on these decisions the Global Decision Logic (GDL) sends a final trigger decision signal. Triggers of the sub-detectors are listed below:

- **CDC trigger:** This trigger is split ub in two sub-triggers, 2D and 3D. For the 2D trigger the CDC data is processed in perpendicular projections. It first searches for circular shaped tracks and transforms the via conformal transformation into lines originating from the vertex in conformal space. Second step is to find these tracks via Hough transformation and transmit this 2D trigger to the GDL. In addition the data is used to perform a r-ϕ-fit and r-z-fit to find an axial coordinate of the track which is the result of the 3D trigger.

- **ECL trigger:** This trigger is based on FPGA hardware which analyzes the data transmitted from the ECL. The so called fast shaper analyzes the signal's pulse hight and time length which are forwarded to the GDL.

- **PID trigger:** The PID trigger uses data from the Cherenkov detector, which was not used before, in particular position and time and transmits it to the GDL.

- **KLM trigger:** The last trigger comes from the KLM. Its main purpose is to trigger on existing μ-pairs in the event by performing a 3D tracking on the data.

3.4.2 Belle II DAQ

Goal of the Belle II DAQ is to store all physically interesting data and discard most of the background. Therefore the detectors are read out according to the level 1 (L1) trigger decision which is provided by the trigger-system explained in the last chapter. Data is then transferred from the front end electronics through several steps of data processing to Event Builder 2 (EVB2) and the storage system. An overview of this global DAQ architecture and its main components, the unified data link (Belle2Link), the Common Readout Platform (COPPER), the event Builder system (EVB1 & EVB2) and the High Level Trigger (HLT) is given in figure 3.13.

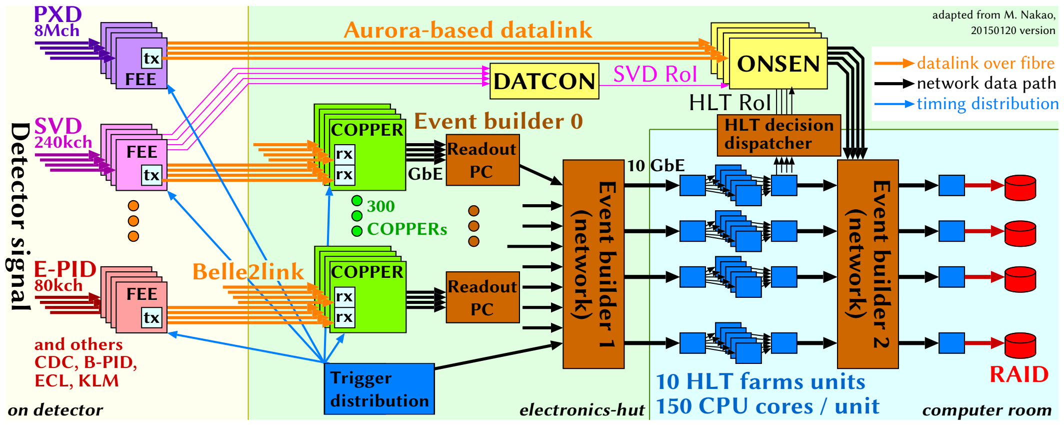


Figure 3.13: Design of the Belle II DAQ. All detectors are read out by there own front end electronics (FEE) based on FPGAs. Data is then trasnmitted according to the trigger distribution (blue) via Belle2Link to the COPPER boards except for the PXD which transfers data via optical link to the ONSEN system. The SVD transmits data via Belle2Link to COPPER as well as via optical link to the DATCON. COPPER boards are read out via readout PCs and the EVB1 which transmits the data to the HLT farm which performs trigger and ROI calculation for the PXD and forward data to the EVB2. On EVB2 all data is collected and stored to the RAID.

The data is transmitted via Belle2Link from the front end readout boards, which are placed near or inside the detectors, to COPPER. Simple data reduction is performed on each front end board or the COPPER receiver module. Formatting and module-level event building is done on the COPPER's on baord CPU. Further event building and data reduction is done on the readout PCs and event builder. Data is then finally processed by the HLT farms for software event selection.

3.5 Data reduction with the ONSEN-system

The ONSEN-system represents an integral part of the PXD readout chain where it provides the connection to Event Builder 2 (EVB2) and the data store. The system is build out of 9 carrier boards, assembled in an ATCA-shelf (Advanced Telecom Computing Architecture), where each can carry 4 smaller xFP cards. Main part of the xFP cards is a Xilinx Virtex 5 FPGA (Field Programmable Gate Array) where the firmware for ROI merging and ROI selecting is running. All cards are equipped with 4 GB DDR 3 RAM which is necessary to store pixel data until the HLT ended calculation time (up to 5 seconds) and ROIs arrive. For data transmission SiTCP [Uch12], an ethernet data transmission protocol based on TCP, and AURORA (which is a Xilinx specific data transmission protocol [Xil15]) is used. Each board is equipped with 2 SFP+ (small form-factor pluggable) transceiver slots and one ethernet connector on the front side. The ethernet connector is used for slow control of the board via EPICS (Experimental Physics and Industrial Control System). SFP+ slots can also be used as SiTCP transceivers but their main purpose, on selecting xFP cards, is receiving pixel data from the DHCs and send reduced data out to EVB 2. ROIs are transmitted from the merger node to the 32 selector nodes via the carrier board's backplane. A scheme of how xFP cards are arranged in an ATCA shelf can be found below.

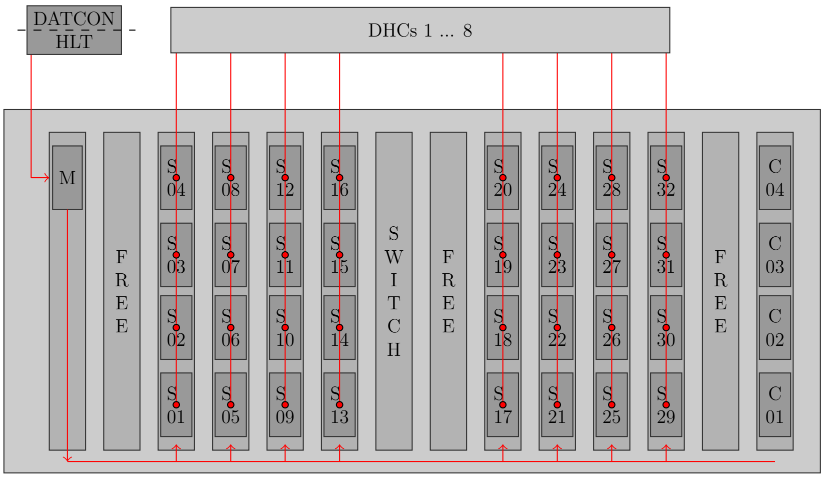


Figure 3.14: Schematic illustration of an ATCA shelf with 9 carrier boards. The merger (M) on the left side distributes ROIs according to their trigger numbers to the selector nodes (S01 ... S32) via backplane AURORA. Pixel data is coming from the DHCs to each selector node's front optical link.

To transmitt data to EVB2, three options are under discussion and shall be briefly explained:

- **First and so far only tested option:** Send data to EVB2 via 32 ethernet links, one for each selector node. Problem with that option is the limited programmable storage on each FPGA. To transmit the data via ethernet one needs to implement an SiTCP core to the FPGA's firmware which would occupy approximately 20% of the chips programmable storage. This problem could be solved with one of the two following options.

- **Second option:** Send data via backplane AURORA to four combiner nodes (C 01 ... 04 in figure 3.14) which concentrate the data and send it via SiTCP to EVB2. This option needs SiTCP cores only on the combiner nodes which is no problem since additional firmware on this nodes would not occupy much of the FPGA's resources.

- **Third option:** Send data via backplane Gbit ethernet to a 10Gbit ethernet switch and from there via SiTCP to EVB 2. This option is under investigation and would be the most convenient since it would not need additional firmware development.

So far only option one is tested and considered as fall back solution. Discussions on the other solutions are still ongoing and will be clarified when first results of the third option are achieved. A picture of one fully equipped ONSEN carrier board and one compute node (CN) can be found below.

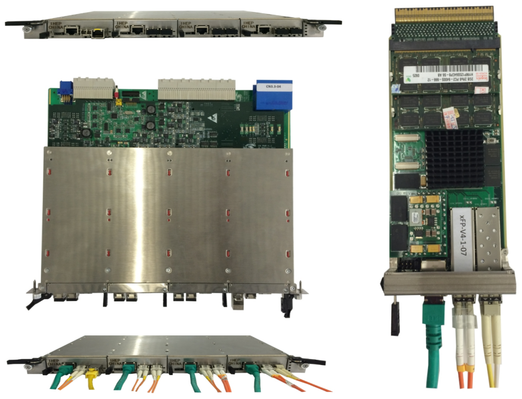


Figure 3.15: **Left:** Picture of a fully equipped carrier board. **Right:** Top view of an xFP daughter board card.

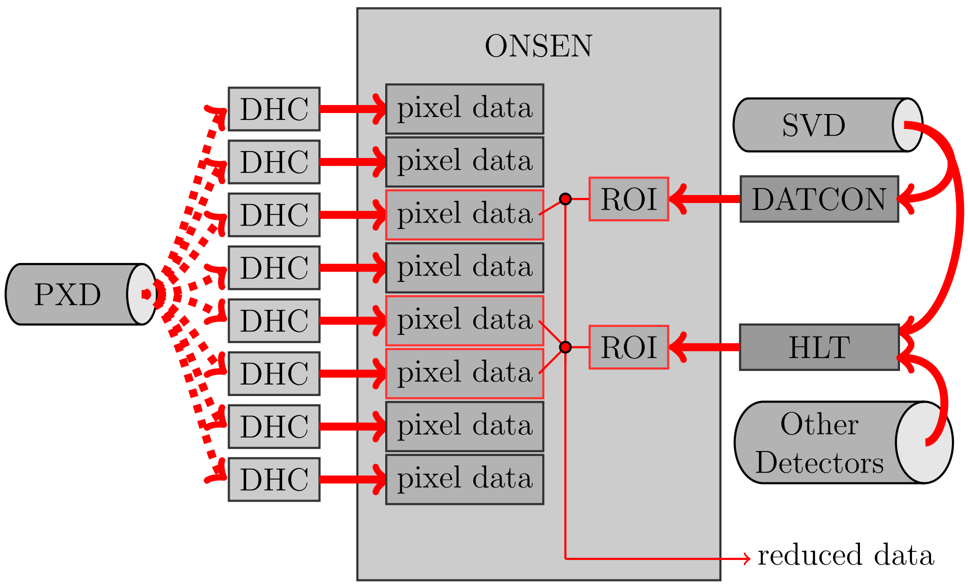


Figure 3.16: Schematic picture of how data is reduced within the ONSEN-system. Pixel data is transmitted from the DHEs to the DHCs as shown in figure 3.8. DATCON ROIs and pixel data are then buffered for 5 seconds in the ONSEN-system until the ROI decision from HLT arrives. Both ROI sources are then merged according to the trigger number. Pixel data which corresponds to matched ROIs is then transmitted to EVB2.

Figure 3.8 shows how pixel data is read out from the PXD, in figure 3.16 this part is symbolized by the dashed red arrows. The ONSEN-system receives pixel data from 8 DHCs. On the other hand two sources of ROIs are streamed to the ONSEN, the DATCON which calculates ROIs only from SVD data and the HLT which calculates ROIs with data from all detectors except the PXD itself. The merger is responsible for matching these two ROI sources and transmit the matched ones to the selectors. Within these selector nodes ROIs are selecting physically interesting pixel data which has the same trigger number, VXD ID and DHP ID combination (numbers which identify one half-ladder) and originates from pixels inside this ROI. After ROI selection the background in pixel data is reduced by a factor of 30 which makes it convenient to store on hard disk. An illustration of a ROI pointed to by a particle trajectory is given in figure 3.17

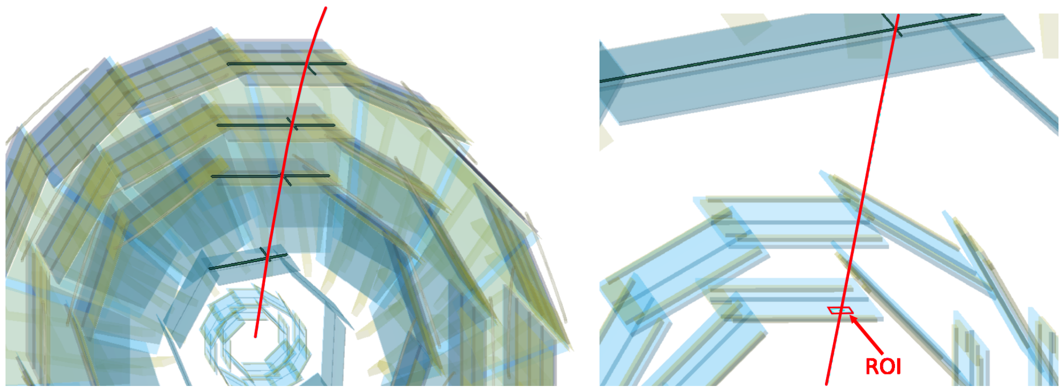


Figure 3.17: **Left:** A particle track, drawn in red, crosses the layers of the VXD and points back to its vertex near the IP. **Right:** Zoom on the PXD surface. The ROI which refers to the particle track is indicated in red.

Chapter 4

Generating, unpacking and monitoring of clustered PXD data

This Chapter will explain the work which was done within this thesis. The first part will give a short introduction to the Belle 2 analysis framework (basf2), which represents the software environment where most of this thesis's work is based uppon. After that a detailed picture of the specific data flow of cluster data in basf2 is given. The data format as well as the generation of test data in the given format for testing purposes is discussed afterwards. With this information the modules and classes, written in C++ and registered in basf2, which access and store the incoming data and can be understood. Finally the result of some generated test patterns is discussed to show consistency between incoming and outgoing data. This last chapter is then closed with a discussion of the achieved work and an outlook to future tests and installation.

4.1 The Belle II analysis framework - basf2

The Belle II analysis framework is the software framework of the Belle II experiment. It provides classes and modules which can analyze data accumulated by the detectors online (directly during data taking) and offline (analysis of pre filtered and stored data).

Basf2 is a modular structured, ROOT-based framework written in C and $C{+}{+}$ using python-scripts to execute modules in a certain, user specific order. There are global modules and detector specific modules which can all be organized according to the following tasks: calibration, reconstruction, unpacking, generation and geometry simulation.

All modules and classes in basf2 consist of a source (`.cc`) and a header (`.h`) file. The header file includes libraries and other header files and declares constants, variables, functions and classes which are needed for the specific module or class. In the source file the declared objects are defined which means that functions get a specific task and variables, constants or classes are assigned with different values according to the functions they are involved in. The created module can then be registered in basf2 and used as part of a module chain to analyze data. An illustration of such a module chain is made in figure 4.1.

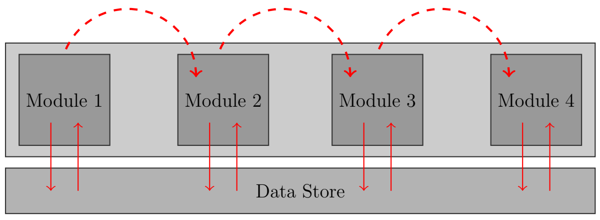


Figure 4.1: The picture illustrates a module chain with 4 modules in basf2. All modules can access the data store to take and store data according to their functions. The modules are executed from left to right (number 1 to 4).

The registered modules with specific parameters given by the user is now accessed event by event, which means that a module chain which shall modify 1000 events would process 1000 times. Since there are far more than 1000 events to process in most of the cases one can use parallel processing in order to shorten calculation time. This method is called multi-threading and is illustrated in figure 4.2.

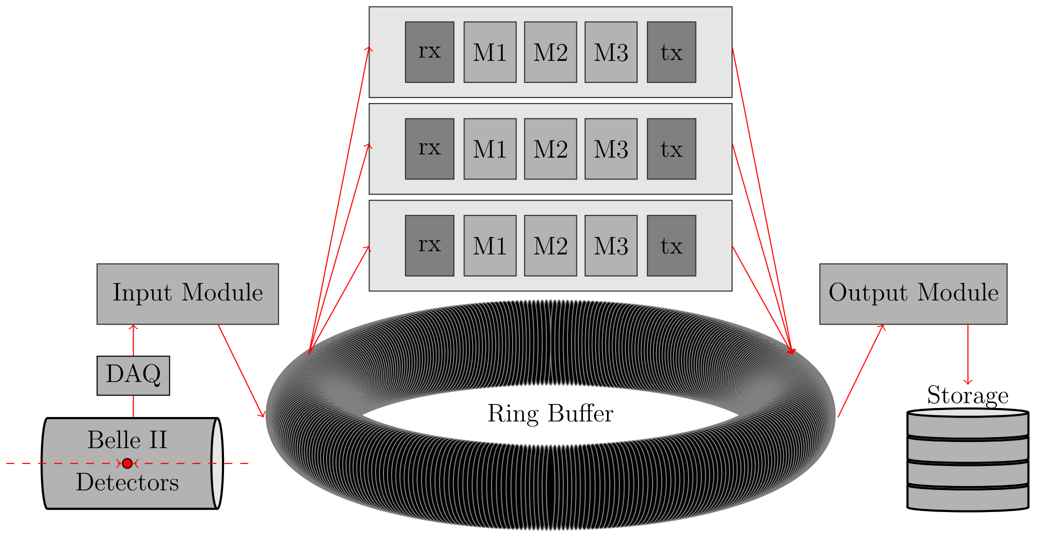


Figure 4.2: Scheme of a multi threading process in basf2. Data is received by the input module which stores it to the ring buffer. From there a module chain is processed event by event, modifies data and writes it back to the ring buffer which is then accessed by an output module writing the data to storage. Receiving data is abbreviated by rx and transmitting by tx.

The data written to the ring buffer or stored in storage is structured in objects. These are class definitions in C++ which allow the user to store data with certain informations which is necesary in order to further analyze it. Some objects are basf2 unique others are derived from the underlying ROOT framework.

4.2 Data flow - processing of cluster data in basf2

As already mentioned in chapter 3.3.1 and chapter 3.5, data which is read out from the DCDs and accumulated from the DHPs and DHCs is send via optical link to the ONSEN system. From there the reduced data is transmitted to EVB 2 where it can be accessed for further analysis. Before data can be analyzed basf2 needs to know how to treat the incoming binary data stream. Therefore important information has to be unpacked and stored. This process is illustrated in the picture below.

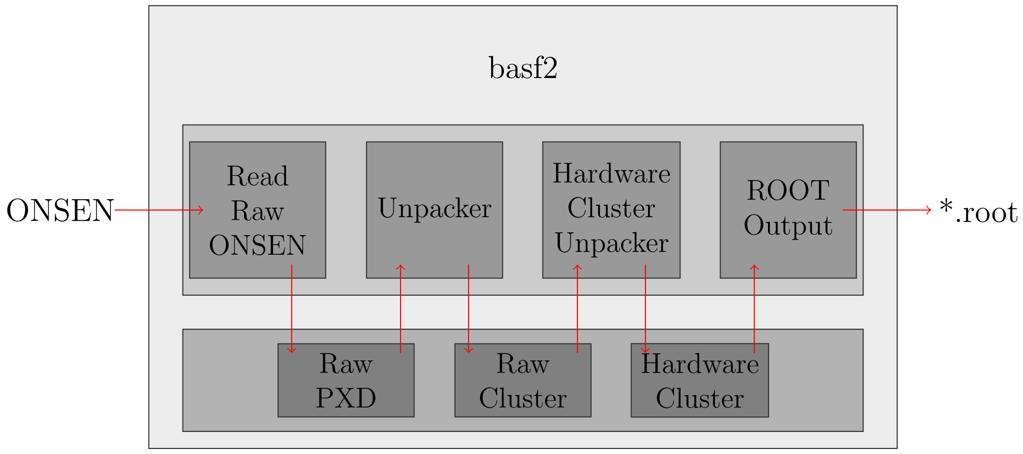


Figure 4.3: Specific module chain for unpacking of cluster data in basf2. Each module is accessing a certain class from the data store in which it reads or saves data. The final output is a .root file.

A python script which represents the upper module chain can be found in listing 6.1 of the appendix.

4.3 Data formats

For each detector at the Belle II experiment a dedicated data format was introduced in order to store important information efficiently in the binary data stream. The specific data formats for the PXD clusters starting with the final ONSEN format are described in this section. As mentioned in the motivation chapter, clusters are cohesive pixel structures which have at least one common edge. An illustration of these clusters is given in figure 4.4.

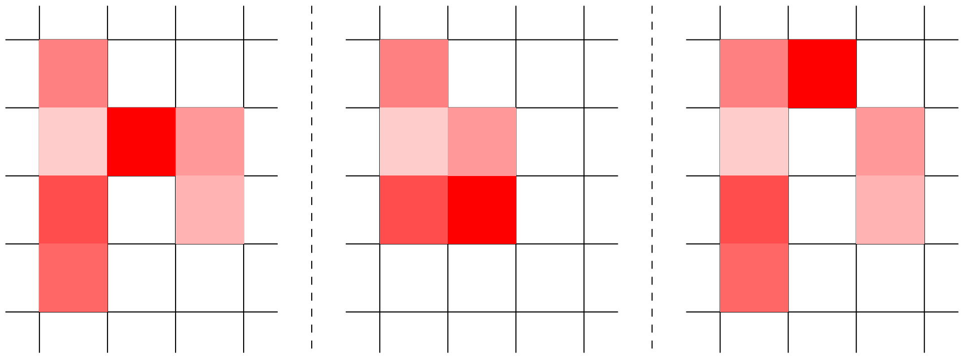
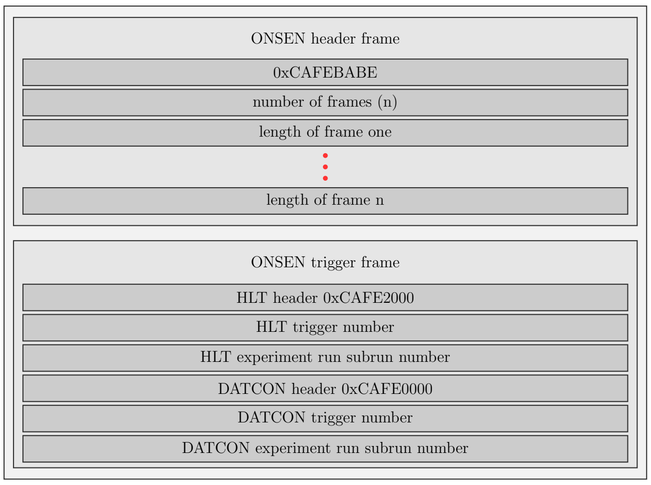


Figure 4.4: Schematic picture of four different clusters, the shades in red indicate the different pixel charge. **Left:** V-shaped cluster. **Middle:** Standard cluster without gaps. **Right:** This object is treated as two different clusters since the two pixels on the right only have a common corner (and no edge) with the rest of the structure.

4.3.1 Outgoing ONSEN frames

The final ONSEN frames which are transmitted to EVB2 are made up of DHC start and end of sub event frames which envelope DHE start and end frames including the data frames. The ONSEN header and trigger frame is illustrated on top in figure 4.5 whereas an example of how frames for one event can be arranged in a data stream is shown on bottom of figure 4.5.



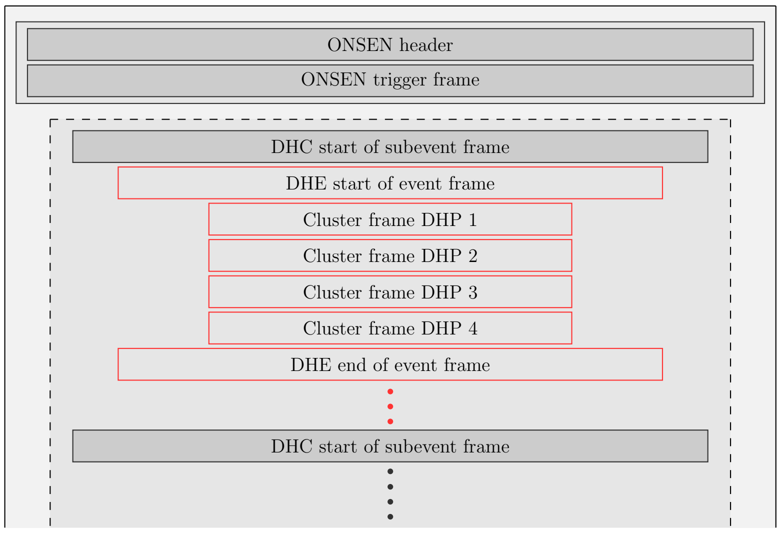


Figure 4.5: **Top:** The ONSEN header and trigger frame. Starting with `0xCAFEBABE`, the ONSEN header frame contains information about how many frames are transmitted and the length of each frame in bytes.**Bottom:** Example for the final ONSEN output including cluster data frames. After the ONSEN header and trigger frame a DHC specific start of sub event frame is transmitted. The DHE start frame marks the start of one DHE (half-ladder) readout. Before the DHE end frame n cluster frames with up to 4 different DHP numbers can be enveloped. This structure (in red) can repeat up to 5 times before the DHC end of sub event frame. One ONSEN frame can contain up to 8 DHC start/end of sub event structures (dashed box).

4.3.2 DHC sub event formats

These sub event formats represent the last data format changes before the ONSEN system. The DHC start and end of sub event structures can reapeat up to 8 times in one ONSEN frame and envelop up to 5 DHE start/end structures.

4.3.2.1 DHC start of sub event format

This start of sub event frame marks the start of one DHC frame and is illustrated below.

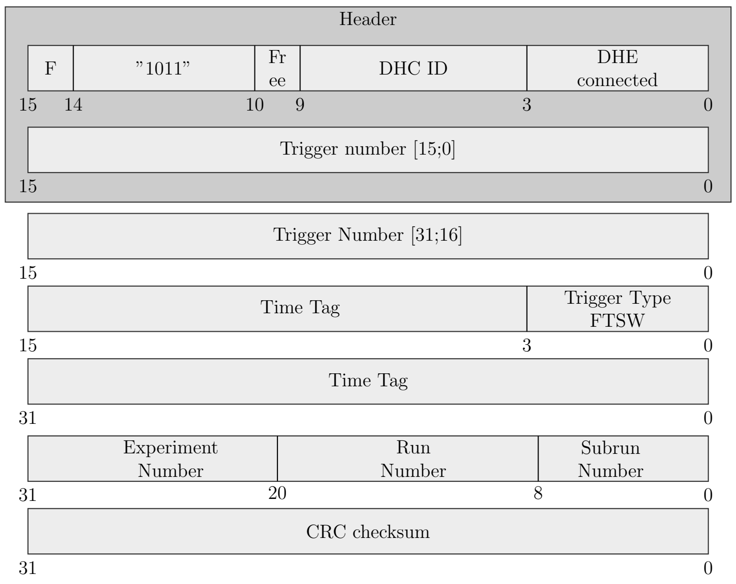


Figure 4.6: This picture illustrates the DHC start of sub event frame. Starting with the header, which contains information about the frame type "1011", the DHC ID, how many DHE are connected and the lower bits of trigger number. After that the upper bits of the trigger number as well as the time tag is transmitted. The frame end with experiment, run, and subrun number and an 32bit checksum.

The header of a DHC start of subevent frame starts with a Flag and "1011" indicating the frame type. The next bit is unused and followed by the DHC ID giving a distinctive number for the DHC in use. The second 16 bit word of the header is occupied by the first 16 bit of the trigger number which is completed by the third 16 bit word. The next three 16 bit words are made up of a time tag from the Front End Timing SWitch (FTSW) and the corresponding trigger type. Before ending the frame with a 32 bit checksum the experiment, run and subrun -number are given.

4.3.2.2 DHC end of sub event format

This frame type marks the end of a DHC frame which can envelope up two 5 DHE frames embracing the data frames.

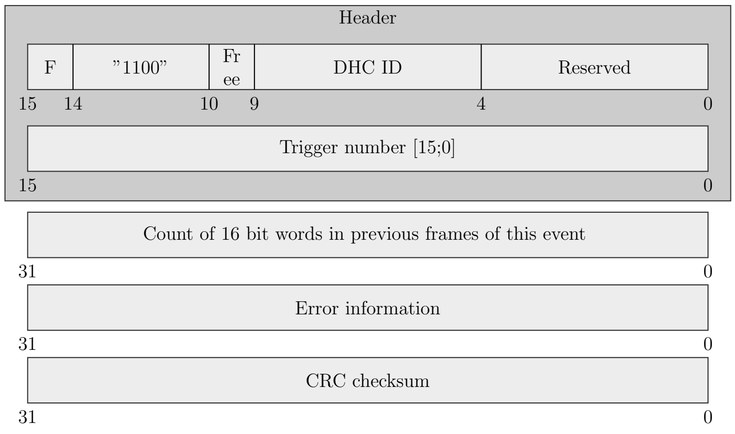


Figure 4.7: This picture illustrates the DHC end of sub event frame. Starting with the header, which contains information about the frame type "1100", the DHC ID and the lower bits of trigger number. After that a 16 bit word count, which counts the 16 bit words in the previous event is transmitted before the frame ends with error information and checksum.

As in the other frames before the header is made up of two 16 bit words, the first with the frame type "1100" and the DHC ID, the second with the lower 16 bit of the trigger number. The three last 32 bit words start with a counter for all 16 bit words in the previous frames of this event. The second last word is for error information and the last one a 32 bit checksum.

4.3.3 DHE event formats

Each DHE, which is responsible for one half-ladder (40 in total), will contribute with a start and end of event frame to the outgoing data scheme which is discussed in 4.3.1.

4.3.3.1 DHE start of event format

The DHE start of event frame will mark the start of pixel data frames. In case pixel data is coming in cluster frames there will be up to 4 cluster frames between DHE start and DHE end frame corresponding to the 4 DHPs on one half-ladder.

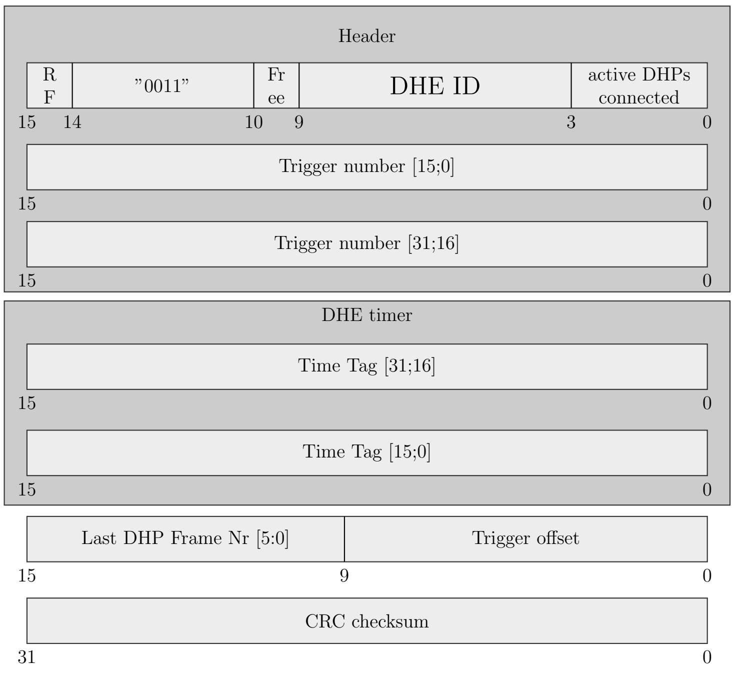


Figure 4.8: This picture shows the DHE start of event frame. Starting with the header, which contains information about the frame type "0011", the DHE ID, how many DHPs are connected and the trigger number. The next block consists of the 32 bit time tag before the last DHP frame number and the trigger offset is given. It also ends with a 32 bit checksum.

As one can see in the upper illustration the header is made up out of one 32 bit word describing the data class and two 32 bit words for the trigger number. The first word starts with the reformatted flag "RF" which indicates weather the data inside the frame was reformatted or not. The second part consists of the frame types identification number "0011", the third part is one free bit and after that the DHE ID is placed. This ID characterizes the half-ladder this frame is read out from. The last four bits correspond to the number of active DHPs which where red out for this frame. The DHE timer is the next block after the header. It provides a time tag for the processed event so that further data analysis can refer to the event. A 16 bit word shows the upper 6 bit of the last DHP frame's number and the trigger offset in respect to the start of frame strobe before the frame ends with a 32 bit checksum.

4.3.3.2 DHE end of event format

The DHE end of event frame format marks the end of one half-ladder readout for one event and is structured like shown below.

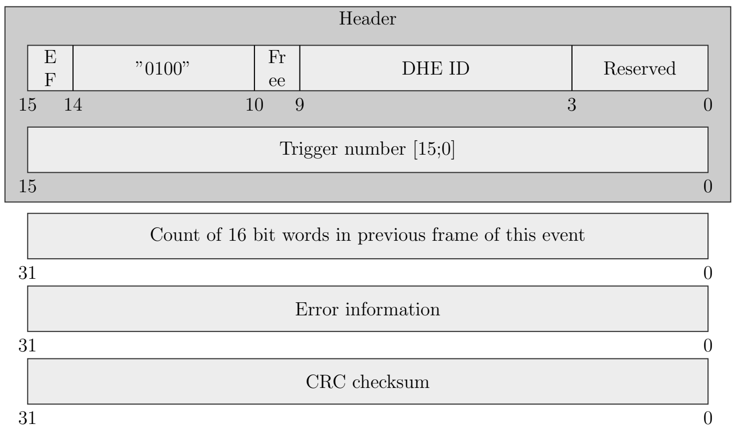


Figure 4.9: The picture shows the DHE end of event frame. Beginning with the header, which contains information about the frame type "0100", the DHE ID and the lower 16 bits of the trigger number. The next 32 bit count all 16 bit words in the previous frame before a 32 bit error information is given. It also ends with a 32 bit checksum.

As in case of the DHE start of event frame this one also starts with a flag "EF", but here this shows weather there was an error in the data frames or not. Second part of the first 16 bit header word is the type number "0100" then 1 bit free and the DHE ID which of course has to match the DHE ID in the start of event frame. The last 4 bits are reserved. In the second 16 bit of the header the lower 16 bit of the trigger number is given. The last three 32 bit words start with a counter for all 16 bit words in the previous data frames of this event before a detailed information on errors is given when "EF" equals 1. The last word is like in the DHE start of event frame a 32 bit checksum.

4.3.4 DHE cluster format

The DHE cluster format is the innermost data structure of the final cluster data stream. It is build on the DHH FPGA which is why it is also called FPGA Clustering Engine (FCE) format.

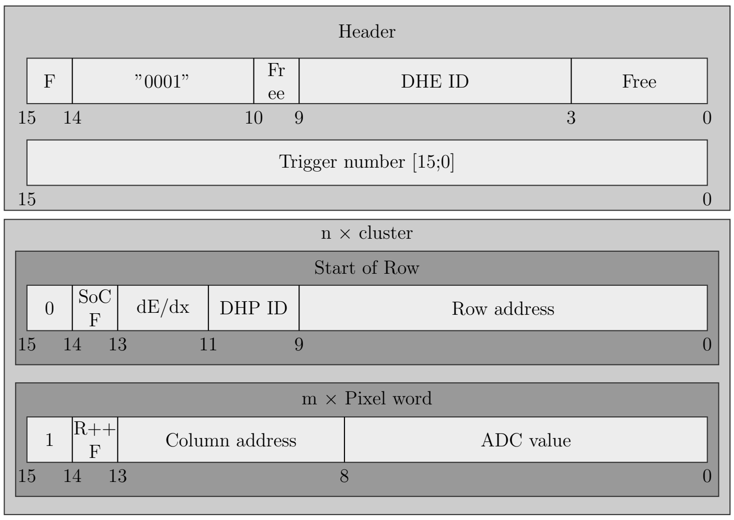


Figure 4.10: The DHE cluster format also starts with a header where type "0001", DHE ID and the lower 16bit of the trigger number is provided. After the cluster header up to n cluster frames starting with a start of row word which contain information about the DHP ID the deposited energy (dE/dx) and the row address can be transmitted. Each cluster can contain m pixel words containing column address and adc value of the pixel. If the pixel is located in the neighboring row, the R++ flag is set to "1". A new cluster always starts with a start of row word, where the SoC (start of cluster) flag is set to "1".

In the upper picture "0001" is the type number of a cluster frame. The Field DHE ID includes the number of the DHE which is read out. In the lower box one cluster is shown which is basically made up out of a start of row / start of cluster word and m pixel words. Between two header frames, which means two different half-ladders or trigger numbers, can be n clusters each one starting with a start of row word where the field SoC (start of cluster flag) is set to one. One cluster can contain more then one start of row word which is for example the case when dealing with V-shaped clusters (5.1.2). All pixel words related to one cluster are then confined between two start of row words where the start of cluster flag is turned to "1". The pixel words themselves contain the important information about column and charge (ADC value), the row is given in the start of row word and will be increased by the readout code (unpacker) when the increment row flag (R++) is set to "1".

4.4 Generation of test data

The last chapter illustrated the frame types which where defined to transport data from the PXD over the ONSEN system to EVB2. In order to test an unpacking module which should read out the important properties of the received data, it is necessary to produce data according to these given formats. Since the cluster unpacker should unpack data arriving from the ONSEN system it has to handle the arriving data stream as outgoing ONSEN frames with all underlying sub frames generated by DHC and DHE. A detailed explanation how this was realized in C++ can be found in listings 6.2 to 6.7 of the appendix.

By modifying the code shown in listing 6.5 in the appendix one can generate all kinds of cluster shapes from randomly organized clusters with different number of pixels to defined cluster patterns. Chapter 5.1 will introduce the cluster patterns used for testing the software written in this thesis.

4.5 Cluster related modules and classes

The last chapters introduced the cluster data format and how data is generated in order to satisfy the given format. In this chapter the modules and classes which interpret this data within basf2 shall be explained.

4.5.1 PXD Unpacker Module

The first module used for unpacking cluster data in basf2 is the PXD unpacker module. It is written to unpack all data formats which can be send from ONSEN to EVB2. For this thesis a new function which allows the module to unpack cluster data was implemented. The C++ code describing this function can be found in listing 6.8 of the appendix.

In order to unpack data to the `PXDRawCluster` class, the incoming data stream has to be interpreted according to the formats presented in chapter 4.3. All frames except the DHE cluster frames in figure 4.4 are discarded in this process. Nevertheless the unpacker needs to understand all frames in order to interpret the data structure correctly. To do so, the code in listing 6.8 is executed. The input of this function is a data stream which is treated as a pool of `RawPXD` objects each one related to one event. In case of incoming cluster data the output of the unpacker module will be the ”`PXDRawCluster`” object which is presented in the next chapter.

4.5.2 Raw cluster class

The raw cluster object (`PXDRawCluster`) is called in the unpacker module. It stores cluster with a given length out of a data stream and saves the size of this cluster and all data in its frames without unpacking it. In the appendix, listing 6.9 and 6.10 show parts of this object’s code.

4.5.3 Hardware cluster unpacker Module

The ”`PXDHardwareClusterUnpackerModule`” is processed after the ”PXDUnpackerModule” in the module chain of a basf2 python script. It is responsible for the interpretation of ”PXDRawCluster” which is the object it gets as input. In order to do so the module needs to unpack data according to the structure presented in chapter 4.3.4. In addition to the interpretation of `PXDRawCluster` objects it also provides functions to find the seed pixel, e.g. the pixel with the highest charge, as well as the total cluster charge. The unpacked and calculated information is then stored as `PXDHardwareCluster` object which will be topic of the next chapter. Parts of the code written for this module can be looked up in listings 6.11 to 6.15 in the appendix.

4.5.4 Hardware cluster class

In the last lines of listing 6.12 in the appendix, the `unpack_fce` function is called to store data in the `PXDHardwareCluster` class. This class saves important information of each cluster. Therefore it uses arrays which have the same size as the number of pixels in the corresponding cluster. These arrays are then filled with row, column and charge of each pixel of a certain cluster. In addition address information of the seed pixel and its charge are saved as well as the total charge deposited in this cluster. Since the half-ladder can be derived from vxd-id also this number is stored. A detailed description of how that was realized in C++ can be found in listing 6.16 and 6.17 in the appendix.

4.5.5 Data Quality Monitoring

In order to have a live monitoring of the processed data a so called Data Quality Monitoring (DQM) module was written. It provides ROOT histograms and fills them with data stored in "`PXDRawCluster`" and "`PXDHardwareCluster`" during the unpacking process. One of the most important values to monitor is the cluster size and the charge deposited in each pixel of a cluster. For the charge a 2 dimensional histogram which shows a PXD half-ladder is filled with information about row and column of each pixel, the charge deposited is indicated as the pixel's color. An example of such a histogram will be shown in chapter 5.1.3

Chapter 5

Data consistency check

To test the written code it is important to perform an input/output consistency check. This was done with the generation of test patterns where the exact shape of transmitted data is known. By plotting the outgoing data to root diagrams one can show that in and outgoing data matches.

5.1 Generation of test patterns

For the generation of test patterns the code presented in listings 6.2 to 6.7 was used. Two different patterns where used to test the unpacking chain; the simplest one a square cluster and a more complicated V-shaped cluster. Both where produced on all 4 DHPs. Since the column information in the cluster frames has only 6 bit, the number of columns is limited to 63 (starting from 0) for each DHP. The ”`HardwareClusterUpacker`” then adds 64 $\times$ nr_dhp columns to the addressed column in order to represent the whole half-ladder.

5.1.1 Square clusters

Square clusters where generated in the following pattern. The dark red points in the middle symbolize the clusters seed charge i.e. the pixel with the highest charge.

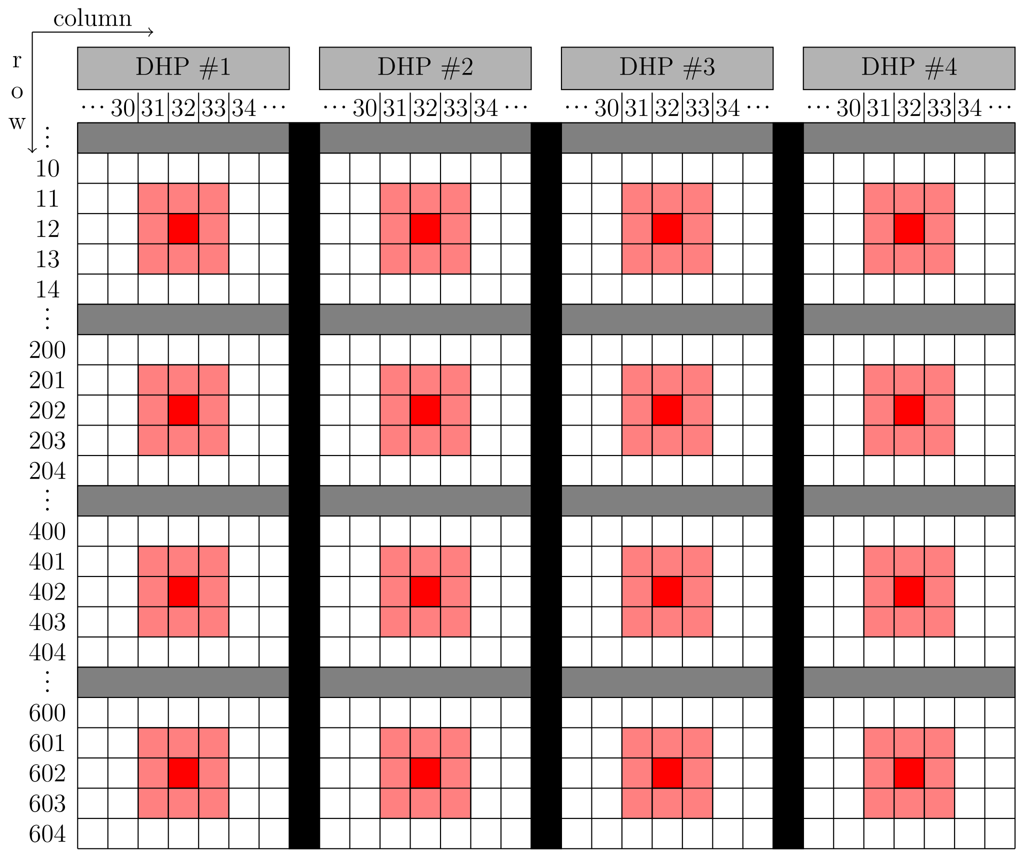


Figure 5.1: DHE half-ladder with square cluster patterns the dark red squares symbolize the pixel with the highest charge.

To generate the upper cluster pattern the code listed in 6.18 in the appendix was used.

5.1.2 V-shaped clusters

V-shaped clusters are special since they need an additional start of row word with start of cluster flag "0". This behavior comes due to the fact that the clustering algorithm on the FPGA starts with a certain row and then increments this row address by setting "increment_row_flag" to "1". Therefore it cannot decrease a row and a new start of row word has to be putted in, in order realize to a "gap" in the cluster. The illustration bellow will clarify this procedure.

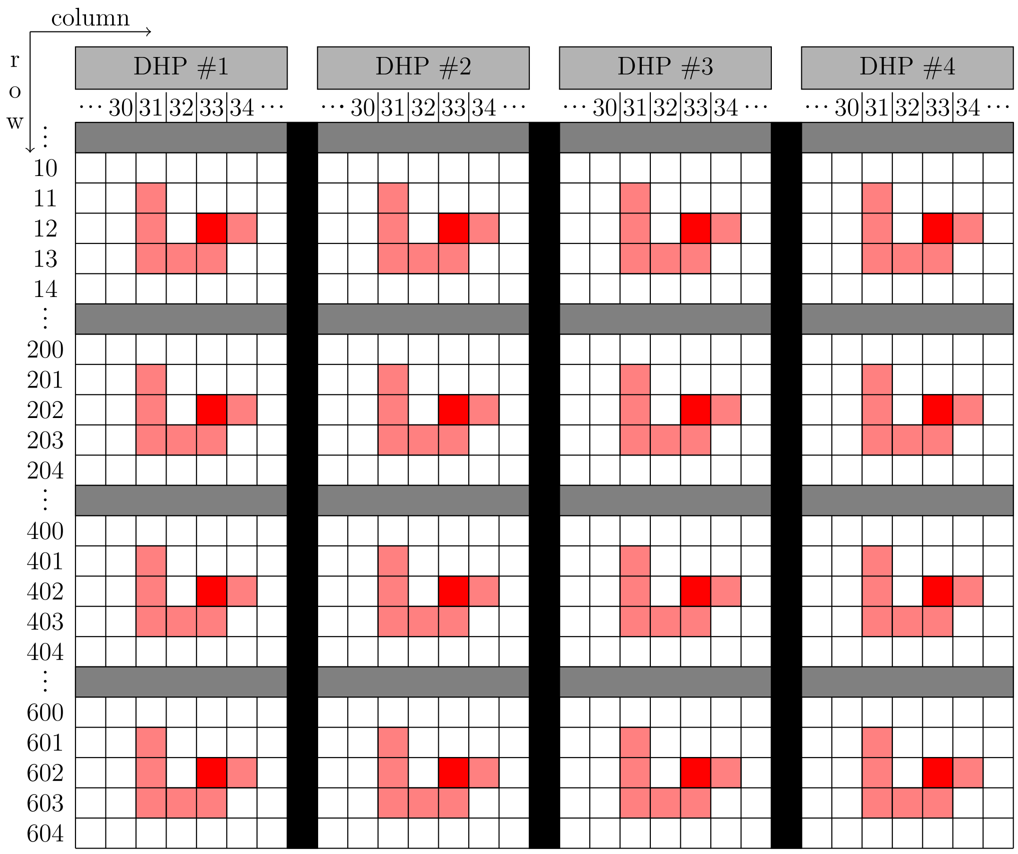


Figure 5.2: DHE half-ladder with V-shaped cluster patterns the dark red squares symbolize the pixel with the highest charge.

Listing 6.19 in the appendix shows the code generating the pattern in the upper figure.

5.1.3 Discussion of the test results

In this chapter the results of the cluster pattern test are presented within plots and histograms. The histograms verify the right number of cluster data, whereas the plots show the geometrical distribution of the clusters on one half-ladder. The next two tables represent a clustered data set of 10 million events for squared shaped as well as V-shaped clusters.

Parameter	**Generated**	**Unpacked**
Cluster length in 16 bit words	10	10
Number of pixel per cluster	9	9
Pixel charge	$128 \cdot 10^6 \times 30$ $16 \cdot 10^6 \times 50$	$128 \cdot 10^6 \times 30$ $16 \cdot 10^6 \times 50$
Charge of seed pixel	50	50
Total cluster charge	290	290
Coordinates of seed pixel [row/column] DHP 1	12/32 12/96 12/160 12/224	12/32 12/96 12/160 12/224
DHP 2	212/32 212/96 212/160 212/224	212/32 212/96 212/160 212/224
DHP 3	412/32 412/96 412/160 412/224	412/32 412/96 412/160 412/224
DHP 4	612/32 612/96 612/160 612/224	612/32 612/96 612/160 612/224

The upper table compares a data set of 10 million squared shaped generated cluster events with the unpacked numbers of the root histogram. 10 million events with 16 cluster per event (4 per DHP) and 9 pixels in each cluster correspond to a total amount of $144 \cdot 10^6$ pixels. Since each cluster has one seed pixel (with charge 50) the amount of clusters with charge 30 should be $8/9 \times 144 \cdot 10^6 = 128 \cdot 10^6$ and $16 \cdot 10^6$ pixels should have charge 50. This is shown in the upper table under the parameter pixel charge.

The next table shows the comparison between the parameters of 10 million generated V-shaped clusters and the corresponding unpacked parameters.

Parameter	**Generated**	**Unpacked**
Cluster length in 16 bit words	9	9
Number of pixel per cluster	7	7
Pixel charge	$96 \cdot 10^6 \times 30$ $16 \cdot 10^6 \times 50$	$96 \cdot 10^6 \times 30$ $16 \cdot 10^6 \times 50$
Charge of seed pixel	50	50
Total cluster charge	230	230
Coordinates of seed pixel [row/column] DHP 1	12/33 12/97 12/161 12/225	12/33 12/97 12/161 12/225
DHP 2	212/33 212/97 212/161 212/225	212/33 212/97 212/161 212/225
DHP 3	412/33 412/97 412/161 412/225	412/33 412/97 412/161 412/225
DHP 4	612/33 612/97 612/161 612/225	612/33 612/97 612/161 612/225

As in case of the square shaped cluster data sample, the pixel charge divides on pixels with charge 30 and seed pixels with charge 50. Here the total amount of pixels can be calculated as $10^6 events \times 16 \frac{cluster}{event} \times 7 \frac{pixel}{cluster} = 112 \cdot 10^6 pixels$. According to this calculation there should be $1/7 \times 112 \cdot 10^6 = 16 \cdot 10^6$ seed pixels with charge 50 and $6/7 \times 112 \cdot 10^6 = 96 \cdot 10^6$ pixels with charge 30. This can be seen under the parameter pixel charge in the upper table. The first two parameters are related to the length of each cluster and the number of contained pixels. Since in case of V-shaped clusters one needs to add two start of row words, the number of words in the cluster has to be the number of pixels plus 2.

The next plots show the result of square unpacked clusters, arranged like the pattern in figure 5.1, which were transferred to EVB2.

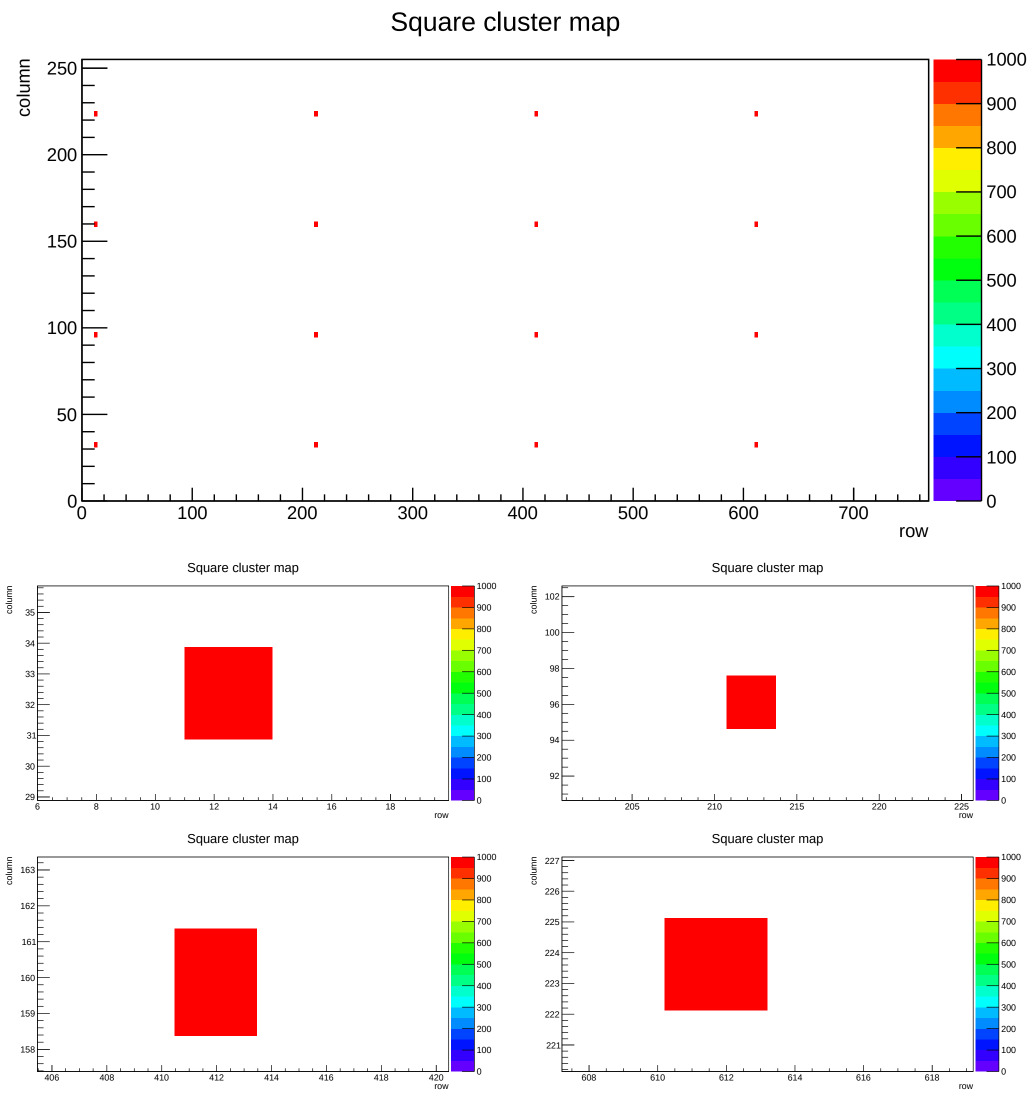


Figure 5.3: **Top:** This picture shows one half-ladder populated with square shaped clusters. The column is plotted on the y-axis and the row on the x-axis, color indicates how often one pixel was hit. **Bottom:** Zoomed in parts of the upper plot. For each DHP one square shaped cluster is shown.

Now also the results of the V-shaped cluster (figure 5.2) test, which have a more complicated data structure, shall be shown.

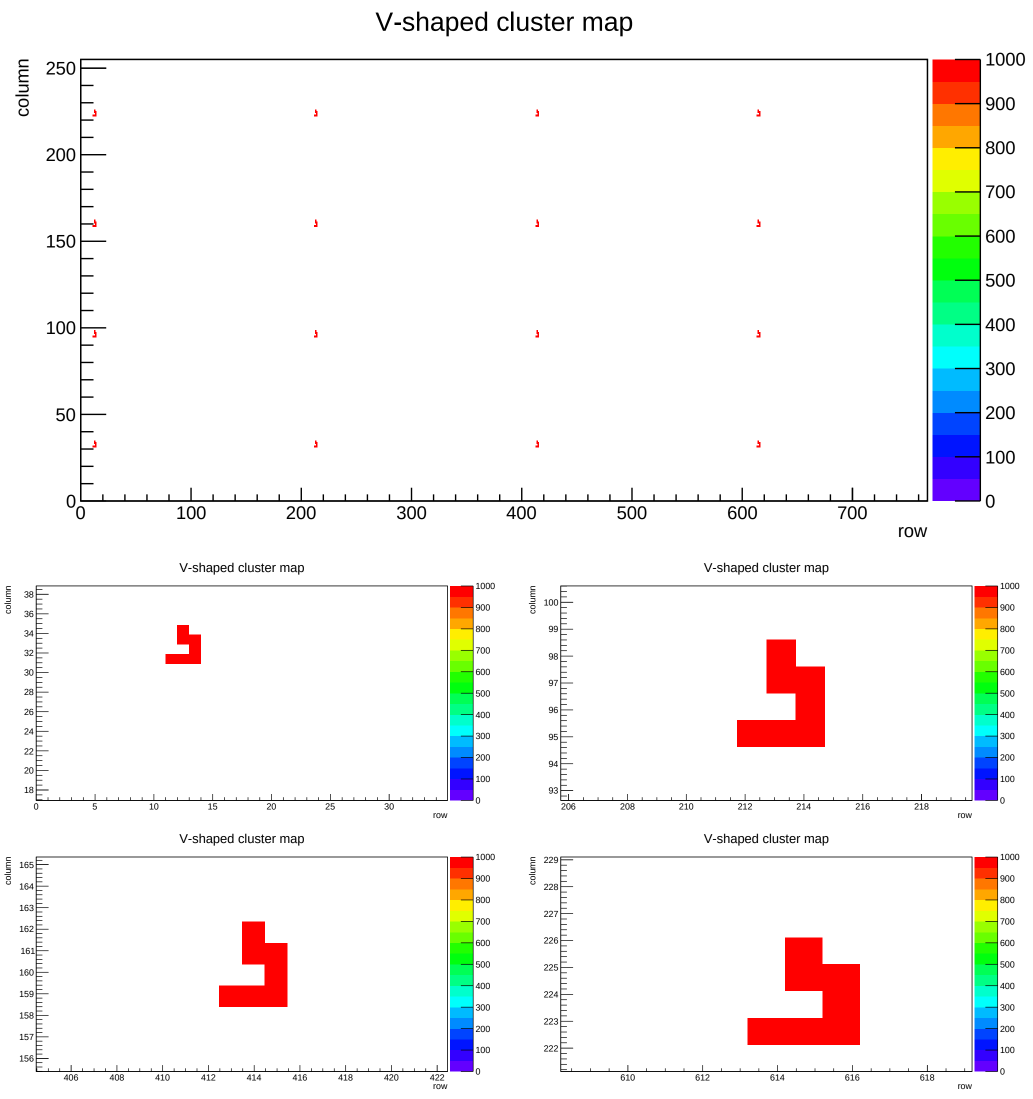
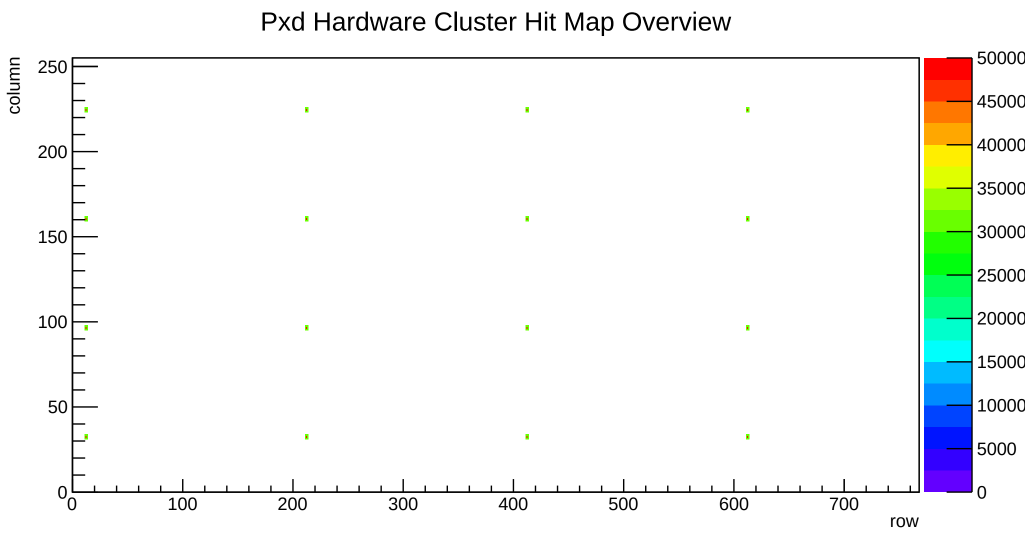


Figure 5.4: **Top:** This picture shows one half-ladder populated with V-shaped clusters. The column is plotted on the y-axis and the row on the x-axis, color indicates how often one pixel was hit. **Bottom:** Zoomed in part of the upper plot. One cluster on each DHP is shown therefore the column number has to be increased by 64 × *nr_dhp*.

The next two figures show an overview of square and V-shaped clusters as well as a zoomed cluster of both types. All four histograms are generated with the DQM module, the charge of each pixel is indicated with colors.



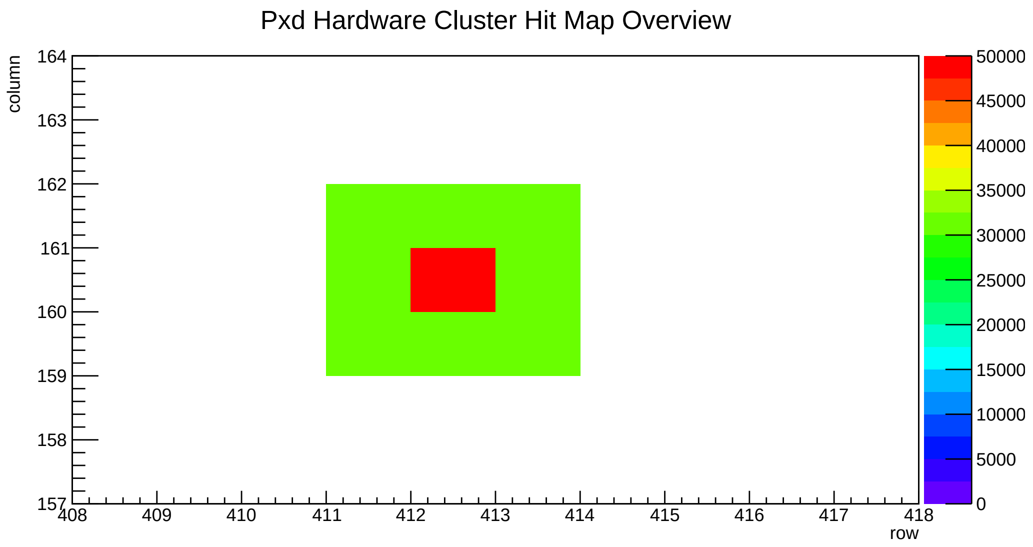


Figure 5.5: Square cluster pattern shown on the whole half-ladder (top) zoomed in part (bottom). Columns are shown on the y-axis and rows on the x-axis. The bottom part shows a square cluster with its seed pixel (red) in the middle.

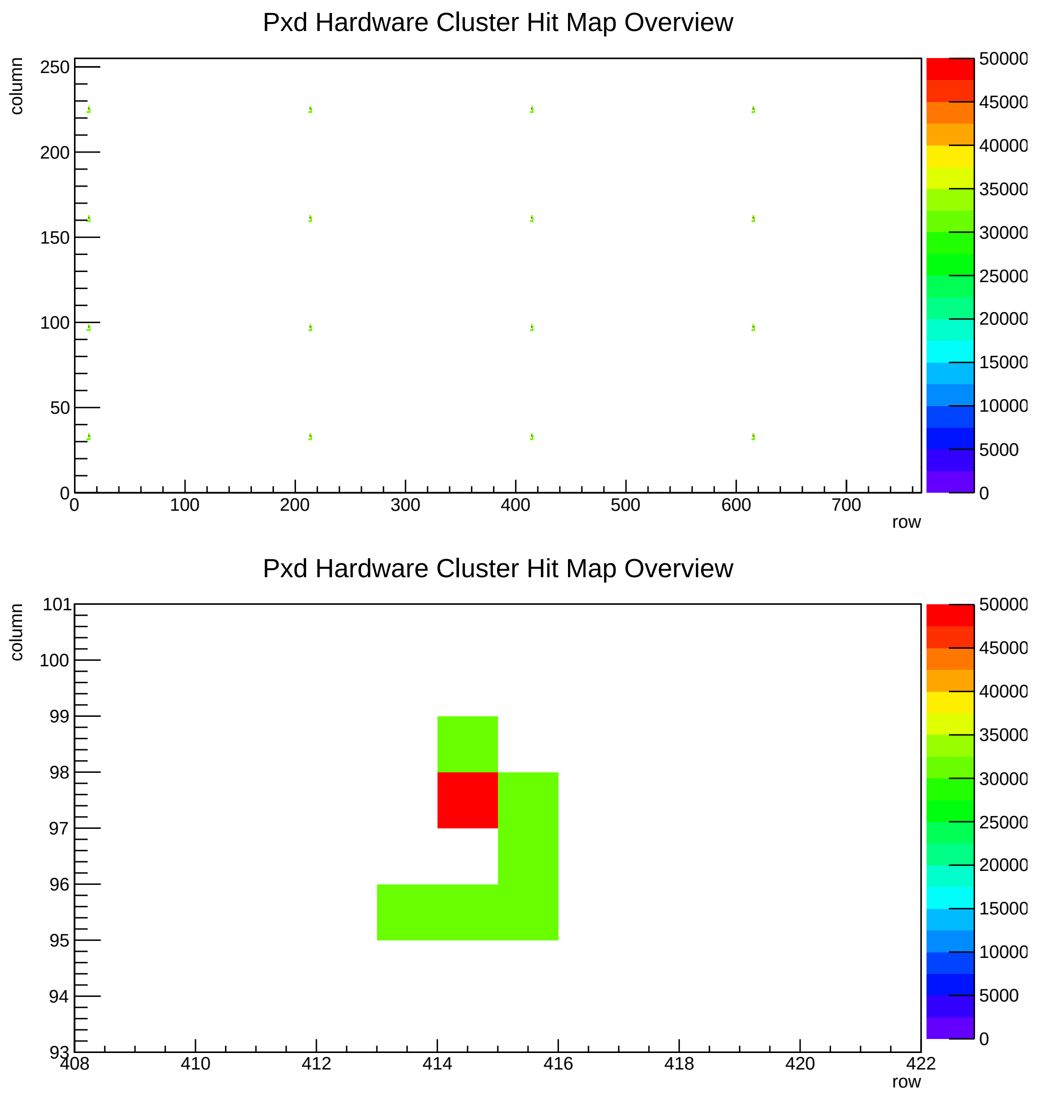


Figure 5.6: V-shaped cluster pattern on the whole half-ladder (top) zoomed in part (bottom). As in case of the square cluster, the red pixel indicates the seed charge pixel.

5.2 Summary and Conclusion

Goal of this thesis was the implementation of a whole new data format to the Belle II analysis framework basf2. The implemented cluster format is generated from the DHE which reads out one half-ladder of the pixel detector. These clusters are cohesive structures of DEPFET pixels which were hit by a particle depositing charge in it. Since, up to now, there is no available cluster data from the DHE, data had to be generated according to the formats given in section 4.3 of this thesis.

Clusters are defined as pixel structures which have at least one common edge. Due to that, there are various of possible cluster shapes. Nevertheless it is possible to break down the almost infinite number of different shapes to two main categories, V-shaped and non V-shaped clusters. An example for this two types is shown in figure 4.4. Special about the V-shaped clusters is their additional header word, the start of row word. It is necessary in order to form the "gap" which is characteristic for these cluster type. For that reason the written data generator had to be "flexible" in the way it generates cluster shapes.

Since the data which is transmitted to basf2 originates from the ONSEN system, the generator also needs to add all necessary headers shown in figure 4.5, in order to be processed in the right order from the unpacking modules implemented in basf2. For this unpacking process the existing PXD unpacker module in basf2 needed expansion.

If the type field in the header of figure 4.10 indicates a DHE cluster frame, the unpacker saves the length and content of each cluster in this frame as member of the new implemented PXDRawCluster class. In this class only the length and the packed content is saved with no additional information about row, column or charge of the cluster's pixels.

To further unpack each cluster and get the physically interesting information it is further unpacker by the PXDHardwareClusterUnpackerModule. This module takes one PXDRawCluster object from the data store (figure 4.3) and unpacks it according to the format shown in figure 4.10. The unpacked data is saved as PXDHardwareCluster object which was also implemented in this work. This class provides arrays to save row, column and charge of each cluster as well as variables for seed and cluster charge. These two parameters are calculated in the PXDHardwareClusterUnpackerModule. The seed charge corresponds to the pixel with the highest charge whereas the cluster charge indicates the total charge of the unpacked cluster. The data is in both cases (PXDRawCluster and PXDHardwareCluster) saved in .root files which can be accessed and analyzed.

In addition to this unpacking routine, which is processed in a sequence indicated in figure 4.3, a Data Quality Monitoring (DQM) module was implemented. This module provides live histograms during the unpacking process. These histograms show parameters like cluster size in short words, pixels per cluster, total cluster charge or seed pixel charge. In addition 2D plots which indicate one half-ladder are produced. These plots, the so called cluster maps, show how clusters are distributed on one half-ladder. The row is shown on the x-axis and the column on the y-axis. Charge is indicated with colors corresponding to the value shown on the right side of each plot.

To verify that everything works, a data consistency check was performed. Two data samples with 10 million events were generated, one for square shaped and one for V-shaped clusters. Both samples where unpacked to basf2 and the received and transmitted clusters were compared. The result of this test is shown in the tables shown in section 5.1.3. During processing no error was observed in the unpacking routine, which means that all unpacked cluster patterns where stored in therefore developed data classes and show the same pattern in the final root output as they have in the data generated file. The average data rate for the unpacking process on a 2.4GHz PC is $2.5MB/s$.

The code was committed into the repository of basf2 to provide global access for Belle II members who want to make tests with clustered pixel data.

5.3 Outlook

The code developed during this master thesis provides basf2 with the ability of unpacking and storing pixel data which is accumulated in clusters. In order to use this software with pixel data coming from the DHCs the cluster format has to implemented to the ROI selection firmware which runs on the FPGA of the selector xFP cards. If this is done the ONSEN system can place its header on top of the DHC start of sub event frames to satisfy the data structure shown in figure 3.10. In october 2015 will be a first test with a reduced ONSEN system and a DHC at KEK, Japan. Till now it is unclear when the cluster format will be implemented in the firmware of the ONSEN, nevertheless this work shows that clusters can be read out and saved from a data stream correcly in order to use the included data for further physically interesting analysis.

Chapter 6

Appendix

Python script for processing of cluster data

Listing 6.1: Python script which corresponds the module chain shown in figure 4.3. At first the modules are declared by giving them a specific name which is related to the module via the `register_module()` function. For each module one can define parameters with `MODULE_NAME.param('PARAMETER_NAME', 'VALUE')`, for module2 the parameter `DoNotStore` is set to 'False' since data shall be stored. When all modules are registered, a specific sequence is declared. This is done in line 20-24 with the `add_module()` function. The last line gives the command for executing this module chain.

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import os
5  import shutil
6  from basf2 import *
7
8  from basf2 import Module
9          module1 = register_module('ReadRawONSEN')
10         module1.param('FileName', 'input.dat')
11
12         module2 = register_module('PXDUnpacker')
13         module2.param('DoNotStore',False);
```

```

        module3 = register_module('PXDHardwareClusterUnpacker')
        module3.param('histoFileName', 'histogram_of_input.root')

        module4 = register_module('RootOutput')
        module4.param('outputFileName', 'output.root')
main = create_path()
        main.add_module(ReadRawONSEN)
        main.add_module(PXDUnpacker)
        main.add_module(PXDHardwareClusterUnpacker)
        main.add_module(RootOutput)
process(main)
```

Data generator code

Listing 6.2: Main function of the data generator which calls the `sender()` function (line 8) as long as `i` is smaller than the chosen amount of triggers (`trigger_number`). In line 4 the data format variable `data_output_format` is set to 'CLUSTER'.

```
 1  int trigger_counter=0;
 2  int main()
 3  {
 4    data_output_format = CLUSTER;
 5    int i = 0;
 6    while(i < trigger_number){
 7      i++;
 8      if(!sender()) break; //Calls the send function in listing 6.3
 9      trigger_counter++;
10    };
11    printf(stderr,"Status Send: Frames %8d, TrigNr  %08X\n", send_frames,
        send_trig);
12    return EXIT_SUCCESS;
13  }
```

Listing 6.3: `sender` function which calls `generate_data_for_trigger` with the current `trigger_number` and the data format one wants to process; in this case `CLUSTER`. Line 4 and 5 clear the header and payload in order to have free variables for the next frame.

```
 1  int sender(void)
 2  {
 3    //Clears header and payload before new frame is processed.
 4    m_onsen_header.clear();
 5    m_onsen_payload.clear();
 6
```

```cpp
  //Create data frames
  return generate_data_for_trig(trigger_counter, data_output_format);
}
```

Before moving to the actual data generating function, generate_data_for_trigger, three other important functions which do often appear shall be explained.

Listing 6.4: The append_int16() and append_int32() functions, for generating correct data word types. These two functions get an integer or short as input and shift the entries to the right sequence in order to transform the data stream into big endian which can in turn be read by the unpacker. If this would not be done the output of 0xCAFE2000 in line 9 of listing 6.5, which is part of the ONSEN trigger frame, would transform to 0x0020FECA and therefore interpreted wrong from the unpacker.

```cpp
void append_int16(unsigned short word)
{
  m_current_frame.push_back((unsigned char)(word >> 8));
  m_current_frame.push_back((unsigned char)(word));
}

void append_int32(unsigned int word)
{
  m_current_frame.push_back((unsigned char)(word >> 24));
  m_current_frame.push_back((unsigned char)(word >> 16));
  m_current_frame.push_back((unsigned char)(word >> 8));
  m_current_frame.push_back((unsigned char)(word));
}
```

Listing 6.5: add_frame_to_payload function. This function is called each time a frame is finished and shall be added to payload. In line 3 it checks weather the frame is 32 bit aligned or not. If not it reports an error since the unpacker is written to receive 32 bit aligned frames. Line 7 to 9 add the checksum which is part of each frame before adding the length of the frame in bytes (line 11) and its content (line 12) to the payload.

```cpp
1  void add_frame_to_payload(void)
2  {
3    if (m_current_frame.size() & 0x3) {
4      cerr << "Frame is not 32bit aligned, unsupported by Unpacker!"<<endl;
5    }
6    //Add checksum frame
7    dhe_crc_32_type current_crc;
8    current_crc.process_bytes(m_current_frame.data(), m_current_frame.size());
9    append_int32(current_crc.checksum());
```

```
  //And add it
  m_onsen_header.push_back(m_current_frame.size());
  m_onsen_payload.push_back(m_current_frame);
}
```

Listing 6.6: Generator function: generates cluster data for each trigger number processed in `main()` (listing 6.2). Line 6 to 15 generates the ONSEN trigger frame of the outgoing ONSEN format (figure 4.5) starting with the frame type and the trigger number. Line 18 to 20 declare variables connected to the VXD ID such as the DHE ID (`dhe_ids_list`), the DHC ID and the number of active DHEs per DHC which can vary from 1 to 5. The next frame is the DHC start of sub event frame, which is created in lines 22 to 32. For each of these DHC start/end of sub event structures up to 5 DHE start/end of event frames can be generated. In line 36 this is done via a for loop, in this case only for one DHE. Line 37 sets the number of active DHPs for his DHE to 4 which is maximum and corresponds to a full half-ladder. The DHE ID is set according to the number in the `dhe_ids_list` array in line 18. This number is given determined by the for loop. Line 42 to 49 produce the DHE start frame. Now the data frames which contain the cluster data are generated in lines 54 to 99. This is done for all four DHPs (line 57). For each DHP one cluser header is generated (line 58-61) which increases the word count by 2 (line 62). This is important since it has to be checked if the frame is 32 bit aligned. From line 66 to 94 the cluster are generated, five per DHP as shown in the for loop (line 66). Line 72 produces 9 pixel for every cluster. The loop repeats 10 times since the first 16 bit word of each cluster is the start of row word (figure 4.10) which is generated in lines 74 to 76. If it is not the first word, pixel words are generated from line 77 to 94. Line 81 to 84 forfec the generator to produce square shaped cluster like shown in figure 5.1. Since the middle pixel of each cluster shall have the highest charge line 86 sets the charge value to 50. Line 95 to 98 check weather the frame i 32 aligned or not. If not the last start of row word will be added. Then the DHE end frame is produced in lines 105 to 110, followed be the DHC sub of vent frame produced in line 114 to 119. The last lines call the `write_out_data()` function which is explained in the next listing and increase the number of frames sent as well as the trigger number.

```
1 int generate_data_for_trig(unsigned int trigger_nr, data_output_format_type
      mode)
2 {
3   int crc_value;
4   int global_word_count=0;
5   //Places ONSEN trigger frame on top of the generated data
6   start_frame();
```

```
  crc_value=(DHC_FRAME_HEADER_DATA_TYPE_ONSEN_TRG << 27) | (trigger_nr & 0xFFFF);
  append_int32(crc_value); // Frame type and trigger number
  append_int32(0xCAFE2000);// HLT HEADER
  append_int32(trigger_nr);// HLT Trigger Nr
  append_int32(0x00400101);// HLT Run Nr etc.
  append_int32(0xCAFE0000);// DATCON HEADER
  append_int32(trigger_nr);// DATCON Trigger Nr
  append_int32(0x00400101);// DATCON Run Nr etc.
add_frame_to_payload();

// DHC Start
int dhe_ids_list[5]={0x02,0x03,0x04,0x08,0x22};
int dhc_id=1; //DHC ID, can vary from 1 to 8
int dhe_active=0x1;//Max 5 DHE per DHC

start_frame();
  crc_value = (DHC_FRAME_HEADER_DATA_TYPE_DHC_START << 27) | (dhc_id & 0xF) << 21 | (dhe_active & 0x1F) << 16 | trigger_nr & 0xFFFF ;
  append_int32(crc_value);
  crc_value = (trigger_nr & 0xFFFF0000 )>>16;
  append_int16(crc_value);
  append_int32(0x00000000);// TimeTag 11-0 | Type | TimeTag 27-12
  append_int16(0x00000000);// TimeTag 43-27
  append_int16(0x0000);// run number 0, subrun number 0
  crc_value = (0x1) << 8 | 0x00; // exp number 1, run number 0
  append_int16(crc_value);
add_frame_to_payload();

//nr_dhe_frames = number of dhe_start/dhe_end structures between dhc_start and dhc_end frame
//Max 5 for each DHE connected to one DHC
for(int nr_dhe_frames =0 ; nr_dhe_frames < 1 ; nr_dhe_frames++) {
  int dhp_active=0xF;//If 0xF all four DHP channels are connected
  int dhe_id = 0;
  dhe_id = dhe_ids_list[nr_dhe_frames];

  //DHE Start
  start_frame();
    crc_value = 0x18000000 | (dhe_id & 0x3F) << 20 | (dhp_active & 0xF) <<16 | trigger_nr & 0xFFFF ;
    append_int32( crc_value);
    append_int16(trigger_nr >> 16);  // Trigger Nr High
    append_int16(0x00000000);  // DHE Timer Low
    append_int16(0x00000000);  // DHE Timer High
    append_int16(0x00000000);  // Last DHP Frame Nr 5-0, Trigger Offset 9-0
  add_frame_to_payload();

  //Now the data frames...
```

```

//DHP numbering valid on one DHE (half-ladder)
int dhp_ids_list[4]={0x0,0x1,0x2,0x3};
int word_count = 0;//in 16 bit words
//nr_dhp_frames = number of DHP frames per DHE. Max 4, one for each DHP!
for(int nr_dhp_frames = 0 ; nr_dhp_frames < 4 ; nr_dhp_frames++){
 start_frame();
  //FCE Header, one per DHP
  crc_value=(0x0 << 31) | (DHC_FRAME_HEADER_DATA_TYPE_FCE_RAW << 27) | (dhe_ids_list[0] & 0x3F) << 20 | (0x00 << 16) | (trigger_nr & 0xFFFF);
 append_int32(crc_value);
  word_count = word_count + 2;//since header is a 32 bit word
  int row_address = 10;
  int dEdx = 2;
  //Generates nr_cluster Cluster for one DHP
  for(int nr_cluster = 0 ; nr_cluster < 5 ; nr_cluster++){
   int start_of_row_flag = 0, start_of_cluster_flag = 0, pixel_word_flag = 0, increment_row_flag = 0, column_address = 0;
   int adc_value = 0;
   column_address = 20;
   start_of_cluster_flag = 1;
   //Generates pixel in cluster
   for (int pixel_in_cluster = 0 ; pixel_in_cluster < 10 ; pixel_in_cluster++){
    if(pixel_in_cluster==0 ){
     crc_value = (0 << 15) | (1 << 14) | (dEdx & 0x3) << 12 | (dhp_ids_list[nr_dhp_frames] & 0x3) << 10 | (row_address & 0x3FF);
     append_int16(crc_value);
     word_count++;
    } else {
     adc_value = 30;
     pixel_word_flag = 1;
     increment_row_flag = 0;
     if(pixel_in_cluster == 4 || pixel_in_cluster == 7 ){
      increment_row_flag = 1;
      column_address = column_address - 3;
     }
     if(pixel_in_cluster==5){
      adc_value = 50;//One pixel with high charge for seed pixel unpacking
     }
     crc_value = (pixel_word_flag << 15) | (increment_row_flag << 14) | (column_address & 0x3F) << 8 | (adc_value & 0xFF);
     append_int16(crc_value);
     word_count++;
     column_address++;
    }
```

```
     }
    }
    if((word_count % 4) != 0){//If frame is not 32bit aligned the last 16 bit word will be identical with the last start of row word
     crc_value = (0 << 15) | (1 << 14) | (dEdx & 0x3) << 12 | (dhp_ids_list[nr_dhp_frames] & 0x3) << 10 | (row_address & 0x3FF);
     append_int16(crc_value);
    }
   add_frame_to_payload();
   //16 bit word count for DHE end of event frame
   global_word_count = global_word_count + word_count;
   word_count = 0;
  }
  //DHE end of frame
  start_frame();
   crc_value = 0x20000000 | (dhe_id & 0x3F) << 20 | trigger_nr & 0xFFFF;
   append_int32(crc_value);
   append_int32(global_word_count);/ 16 bit word count
   append_int32(0x00000000);// Error Flags
  add_frame_to_payload();
  global_word_count = global_word_count + 96;
 }
 //DHC end of sub event frame
 start_frame();
  crc_value = 0x60000000 | (dhc_id & 0xF) << 21 | trigger_nr & 0xFFFF ;
  append_int32(crc_value);
  append_int32(global_word_count);// 16 bit word count
  append_int32(0x00000000);// Error Flags
 add_frame_to_payload();

 //Done
 WriteOutData(m_onsen_header,m_onsen_payload,mode);
 send_frames++;
 send_trig=trigger_nr;
 return 1;
}
```

When one event is fully processed the data is written out by the `WriteOutData` function which is shown in the next listing. `cout.write(POINTER, LENGTH);` streams the generated data to a file of which the name is consigned as an command line argument to the generator program.

Listing 6.7: `WriteOutData` function to stream the processed data to a file. Line 11 to 20 add the ONSEN header frame from figure 4.5, starting in line 11 and 12 with the ONSEN header magic (`0xCAFEBABE`). Then the number of generated frames followed by there length is written out in line 14 to 20. Now

the content of each frame, with the actual cluster data, is written out in line 22 to 26.

```
 1 void WriteOutData(std::vector <unsigned int>& header, std::vector <std::vector
       <unsigned char> >& payload, data_output_format_type mode)
 2 {
 3   // Create header from payload
 4   int nr_frames = header.size();
 5   int payload_size = 0; // in 32 bit words
 6   for (auto & it : header) {
 7     payload_size += (it + 3) / 4; // in 32 bit word, rounded up
 8   }
 9   unsigned int w;
10   //Write ONSEN header Magic to stream
11   w = endian_swap((unsigned int)0xCAFEBABE);
12   cout.write((const char *)&w, 4);
13   //Write number of frames in this event to stream
14   w = endian_swap((unsigned int)nr_frames);
15   cout.write((const char *)&w, 4);
16   //Append length of each frame to w and write it to stream
17   for (auto & it : payload) {
18     w = endian_swap((unsigned int)it.size());
19     cout.write((const char *)&w, 4);
20   }
21   //Write actual data to stream
22   for (auto & it : payload) {
23     cout.write((const char *)it.data(), it.size());
24     //Checks if data is always 32bit aligned
25     if( (it.size()) & 0x3) cerr << "Frame is not 32 bit boundary!" <<endl;
26   }
27 }
```

Pixel data unpacker

Listing 6.8: Unpack cluster data function: `unpack_fce`. This function is called from the `unpack_event` function which receives one ONSEN frame structure as shown in figure 4.5. The length and pointer to first word of this frame are then forwarded to the `unpack_fce` function shown below. Data is then written to the `ubig16_t` variable `cluster` which is a 16 bit big endian variable. In line 12 to 33 data is read out and saved to the Raw cluster class. Since the function gets a pointer to the first word of the actual cluster frame (figure 4.10), the header and trigger number can be skipped (line 12). The if condition in line 15 searches for a start of row frame with start of cluster flag set to one. The found cluster is then stored as `RawCluster` object in line 17 and `words_in_cluster` is set to 0 to count words for the next cluster. Line 20 to 22 searches for

start_of_row_frame. If the last word in the whole cluster frame is equal to the last start_of_row_frame (happens when generated data was not 32bit aligned), line 28 will overwrite this word with zeros to make checking easier for the PXDHardwareClusterUnpacker.

```
 1  void PXDUnpackerModule::unpack_fce(unsigned short* data, unsigned int length,
        VxdID vxd_id)
 2  {
 3    //initializes cluster pointers with the 16 bit data words
 4    ubig16_t* cluster = (ubig16_t*)data;
 5    int nr_words; //Words in DHP frame
 6    unsigned int words_in_cluster = 0; //Counts 16bit words in cluster
 7    nr_words = length / 2; //length in bytes, nr_words in 16bit words
 8    ubig16_t start_of_row_frame;
 9    //Loops over i which is representing all entries in the array pointed to
10    //By cluster starting with 2 since first two 16 bit words are header and
11    //Trigger number which can be skipped
12    for (int i = 2 ; i < nr_words ; i++) {
13        //Searches for start of row frame with start of cluster flag = 1
14        //=> new cluster
15        if ((((cluster[i] & 0x8000) == 0) && ((cluster[i] & 0x4000) >> 14) == 1)
            ) {
16          //Stores information in new raw cluster object with length
                words_in_cluster
17          m_storeRawCluster.appendNew(&data[i-words_in_cluster],
                words_in_cluster, vxd_id);
18          words_in_cluster = 0;
19        }
20       if((cluster[i] & 0x8000) == 0){
21         start_of_row_frame = cluster[i];
22       }
23       words_in_cluster++;
24       //if frame is not 32bit aligned last word will be the last start of row
             word
25       if ((cluster[nr_words - 1] & 0xFFFF) == (start_of_row_frame & 0xFFFF)) {
26         //overwrites the last redundant word with zero to make checking easier
               in
27         //PXDHardwareClusterUnpacker
28         cluster[nr_words - 1] = 0x0000;
29       }
30      if (i == nr_words - 1) {
31        m_storeRawCluster.appendNew(&data[i-words_in_cluster+1],
              words_in_cluster, vxd_id);
32      }
33    }
34  }
```

Raw cluster class

Listing 6.9: Functions declared in the PXDRawCluster class which become important for storing and further unpacking.

```
//Constructor of the PXDRawCluster class
PXDRawCluster():
  m_length(0), m_cluster(0), m_vxdID() {};
  /**
  * @param length  //length of cluster frame in shorts
  * @param data  //pointer to the first word in cluster frame
  * @param vxdID //Vertex Detector ID of the half ladder the cluster originates
  */

//Destructor of the PXDRawCluster class, called last to clean requested memory space
~PXDRawCluster();

//Actual storing function defined in the source file Listing 3.9
PXDRawCluster(unsigned short* data, unsigned int length, VxdID vxdID);

//Length of cluster in short words
unsigned int getLength() const { return m_length; }

/** Get static pointer to data.
* @param j Index of pointer in data
* @return pointer.
*/
unsigned short getData(unsigned int j) const { return (m_cluster[j]); }

/** Get the Vertex Detector ID of the half ladder the cluster originates from.
* @return Vertex Detector ID.
*/
VxdID getVxdID() const { return m_vxdID; }
```

Listing 6.10: Parts of the source code of the PXDRawCluster class. The class constructor is called in line 2 and 3. It assigns the values it gets from the unpacker (listing 6.8 line 17) to the class own variables `m_length` and `m_vxdID`. `m_cluster` is initialized as array with length `m_length` in line 6. Line 10 to 13 write the data in `d[i]` which was assigned in line 8 to the array elements of `m_cluster`. At the end the destructor ~`PXDRawCluster()` is called which deletes the memory space allocated for `m_cluster` in order to allocate new one for the next event.

```
1  //Storing function which was declared in Listing 3.8
```

```cpp
PXDRawCluster::PXDRawCluster(unsigned short* data, unsigned int length,  VxdID vxdID):
  m_length(length), m_cluster(0), m_vxdID(vxdID) //variables get initialized
{
  //Requests new memory space for m_cluster array with length m_length
  m_cluster = new unsigned short[m_length];
  //Boost's endianess and data type of 'unsigned short* data' to 'ubig16_t* d'
  boost::spirit::endian::ubig16_t* d = (boost::spirit::endian::ubig16_t*)data;
  //Writes all data stored in the array pointed to by 'd' into 'm_cluster'
  for (unsigned int i = 0 ; i < m_length ; i++) {
    m_cluster[i] = d[i];
  }
};

PXDRawCluster::~PXDRawCluster()
{
    delete m_cluster;
};
```

Hardware cluster unpacker

Listing 6.11: This function parses all PXDRawCluster objects in one event to the `unpack_event` function. In line 4 the object `RawCluster` is declared which which contains entries of the `PXDRawCluster` class. The lines 12 to 15 save the important event data of the event where the cluster is contained. Now the `RawCluster` objects are parsed to the `unpack_event` function in lines 18 to 23 and number of unpacked events is increased in line 24.

```cpp
1  void PXDHardwareClusterUnpackerModule::event()
2  {
3    //Registers PXDRawCluster object as RawCluster
4    StoreArray<PXDRawCluster> RawClusters;
5    StoreObjPtr<EventMetaData> evtPtr;
6    //Gets Entries in RawCluster
7    int nRaws = RawClusters.getEntries();
8    if (verbose) {
9      B2INFO("PXD Hardware Cluster Unpacker Info  --> RawClusters Objects in
           event: " << nRaws);
10   };
11
12   m_meta_event_nr = evtPtr->getEvent();
13   m_meta_run_nr = evtPtr->getRun();
14   m_meta_subrun_nr = evtPtr->getSubrun();
15   m_meta_experiment = evtPtr->getExperiment();
16
17   //Hands out each RawCluster object to the unpack_event function
18   for (auto& it : RawClusters) {
    if (verbose) {
      B2INFO("PXD Hardware Cluster Unpacker Module --> Unpack Objects");
    };
    unpack_event(it);
  }
  m_unpackedEventsCount++;
}
```

Listing 6.12: The `unpack_event` function checks weather data has the right size (line 6 to 9) and parses a full PXDRawCluster to the `unpack_fce` function (line 23). The lines 15 to 20 check weather a frame ended with a zero word (happens when produced data was not 32 bit aligned) and discards it.

```
 1  void PXDHardwareClusterUnpackerModule::unpack_event(PXDRawCluster& cl)
 2  {
 3    int fullsize;
 4    unsigned short data[cl.getLength()];
 5    VxdID vxdID;
 6    if (cl.getLength() <= 0 || cl.getLength() > 16 * 1024 * 1024) {
 7      B2ERROR("PXD Hardware Cluster Unpacker Module --> invalid packet size (32
          bit words) " << hex << cl.getLength());
 8      return;
 9    }
10
11    fullsize = cl.getLength();
12    vxdID = cl.getVxdID();
13
14    //Writes the data stored in PXDRawCluster to data[i], an array of 16 bit
          words.
15    for (int i = 0 ; i < fullsize ; i++) { data[i] = cl.getData(i); }
16
17    //Checks weather the last 16 bit word is zero. If so fullsize will be
          decreased by one
18    //16 bit word. This comes if produced data was not 32 bit aligned and an
          additional word
19    //had to be added in order to align it.
20    if((data[fullsize - 1] & 0xFFFF) == 0){ fullsize = fullsize - 1; }
21
22    //Calls the unpack_fce function.
23    unpack_fce(&data[0], fullsize, vxdID);
24    B2INFO("unpack Cluster frame with size " << fullsize);
25  }
```

Listing 6.13: The `unpack_fce` function unpacks one PXDRawCluster object according to its data structure illustrated in figure 4.10. After declaring variables and initializing the arrays with zero, the main unpacking code starts in line 21. Here it is checked weather the word is a start of row word or a

pixel word. If it is a pixel word, the number of pixels is incremented and a `increment_row_flag` is searched. If this is one, the `row_address` written out ion line 24 is incremented by one. Now the column address is written out in line 32 and it is checked on which DHP the cluster is located. According to the DHP number an offset of 64·`dhp_id` is added. All characterizing pixel values (row, column, charge) are then written out to the corresponding arrays (line 38 to 40). The for loop starting in line 45 checks weather that the arrays do not have entry zero which would correspond to a second start of row word in the cluster. If a the entries are zero, the arrays gets shifted by one entry. In line 67 the `calc_cluster_charge` function is called, which calculates the total cluster charge. Line 69 calls the `find_seed_pixel` function which searches the clusters seed pixel. The last lines store the unpacked data to an `PXDHardwareCluster` object.

```
 1  void PXDHardwareClusterUnpackerModule::unpack_fce(void* data, unsigned int
      nr_words,
 2                                                                VxdID vxdID)
 3  {
 4    unsigned short* dhe_fce = (unsigned short*)data;
 5    unsigned int increment_row_flag = 0, all, dhp_id = 0, index[nr_words],
        nrPixel = 0;
 6    unsigned short rows[nr_words], cols[nr_words], row_address = 0,
        column_address = 0;
 7    unsigned char adcs[nr_words], adc_value = 0;
 8    //Declares a variable of the type seed_pixel defined as struct in
 9    //PXDHardwareClusterUnpacker.h
10    seed_pixel addr;
11
12    //Initializes declared arrays with zero
13    for (unsigned int i = 1; i < nr_words ; i++) {
14      rows[i] = 0;
15      cols[i] = 0;
16      adcs[i] = 0;
17      index[i] = 0;
18    }
19
20    //Checks weather short word is a start of row or a pixel word
21    for (unsigned int i = 0; i < nr_words; i++) {
22      if ((dhe_fce[i] & 0x8000) == 0 && i != nr_words - 1) {
23        //Start Of Row
24        row_address = (dhe_fce[i] & 0x3FF);
25        dhp_id = (dhe_fce[i] & 0xC00) >> 10;
26        index[i] = i;
27      } else {
28        //Pixel word
29        nrPixel++;
```

```
      increment_row_flag = (dhe_fce[i] & 0x4000) >> 14;
      if (increment_row_flag ==  1) { row_address++; }
      column_address = (dhe_fce[i] & 0x3F00) >> 8;
      //Adds 64*nr_dhp to column address if dhp_id != 0
        if (dhp_id == 1) { column_address = column_address + 64; }
        if (dhp_id == 2) { column_address = column_address + 64 + 64; }
        if (dhp_id == 3) { column_address = column_address + 64 + 64 + 64; }
      adc_value = (dhe_fce[i] & 0xFF);
      adcs[i] = adc_value;
      cols[i] = column_address;
      rows[i] = row_address;
    }
  }
  //Checks that the arrays adcs, cols, rows do not have entry 0 which would
  //correspond to a Start of Row word from the previous loop
  for (unsigned int j = 1 ; j < nr_words ; j++) {
   if (index[j] != 0 ) {
      for (unsigned int l = j ; l < nr_words; l++) {
        cols[l] = cols[l + 1];
        rows[l] = rows[l + 1];
        adcs[l] = adcs[l + 1];
      }
    }
  }

  unsigned short r[nrPixel], c[nrPixel];
  unsigned char q[nrPixel];

  //Shifts the arrays rows cols adcs so that they don not have entries with
      value 0 at the end
  //which can derive from more than one Start of Row word per cluster
  for (int k = 1; k <= nrPixel ; k++) {
    r[k - 1] = rows[k];
    c[k - 1] = cols[k];
    q[k - 1] = adcs[k];
  }

  //Calls the calc_cluster_charge function described in listings 6.14
  all = calc_cluster_charge(dhe_fce, nr_words);
  //Calls the find_seed_pixel function described in listings 6.15
  addr = find_seed_pixel(dhe_fce, nr_words, dhp_id);

  if (!m_doNotStore) { //Stores one PXDHardwareCluster object to the data
      store
    m_storeHardwareCluster.appendNew(&r[0], &c[0], &q[0], nrPixel, addr.charge
        , addr.row, addr.col, all, vxdID, dhp_id);
  }
}
```

In order to calculate the total cluster charge and search the seed pixel, two other functions are necessary. Both are called in the `unpack_fce` function in listing 6.13. The next two listings show how this functions work.

Listing 6.14: `calc_cluster_charge` calculates the integral charge of one cluster. In the for loop (lines 5 to 10) a pixel word is identified by its flag (figure 4.10) and the charge is written to the value `charge`. This value is then added to `cluster_charge` for all pixels in the cluster. Line 11 returns the value of `cluster_charge`.

```
 1  unsigned int PXDHardwareClusterUnpackerModule::calc_cluster_charge(unsigned
        short* data, unsigned int nr_pixel_words)
 2  {
 3    unsigned char charge = 0;
 4    unsigned int cluster_charge = 0;
 5    for (unsigned int i = 1 ; i < nr_pixel_words ; i++) {
 6      if ((data[i] & 0x8000) != 0) { //Checks if short word is a pixel word
 7        charge = (data[i] & 0xFF);
 8        cluster_charge = cluster_charge + charge;
 9      }
10    }
11    return cluster_charge;
12  }
```

Listing 6.15: `find_seed_pixel` searches address and charge information of the clusters seed pixel. The `seed_charge` is initialized with the first pixels charge (line 5). Then it is checked for every pixel weather itscharge is higher or lower then the actual value of `seed_charge`. Every time `seed_charge` is changed, the row and column of the pixel is written out. Line 13 writes out the `index` i of the pixel in order to get information of its DHP ID in lines 18 to 21. Here the pixels column is customized according to the DHP the pixel originates from. The last lines set the variables of the `addr`-object and return it.

```
 1  PXDHardwareClusterUnpackerModule::seed_pixel PXDHardwareClusterUnpackerModule
        ::find_seed_pixel(unsigned short* data, unsigned int nr_pixel_words,
        unsigned int dhp_id)
 2  {
 3    seed_pixel addr;
 4    unsigned short row_addr = 0, col = 0, index = 0, row = (data[0] & 0x3FF);
 5    unsigned char seed_charge = (data[1] & 0xFF) >> 8;
 6    for (unsigned int i = 0 ; i < nr_pixel_words ; i++) {
 7      if ((data[i] & 0x8000) == 0) { row = (data[i] & 0x3FF); }
 8      if ((data[i] & 0x8000) != 0) {
 9        if (((data[i] & 0x4000) >> 14) == 1) { row++; }
10        if ((data[i] & 0xFF) > seed_charge) {
11          seed_charge = (data[i] & 0xFF);
```

```
          row_addr = row;
          index = i;
        }
      }
    }
    //Sets the column address according to the DHP the cluster emerges from
      if (dhp_id == 0) { col = (data[index] & 0x3F00) >> 8; }
      if (dhp_id == 1) { col = ((data[index] & 0x3F00) >> 8) + 64; }
      if (dhp_id == 2) { col = ((data[index] & 0x3F00) >> 8) + 64 + 64; }
      if (dhp_id == 3) { col = ((data[index] & 0x3F00) >> 8) + 64 + 64 + 64; }
    addr.col = col;              //column of seed pixel
    addr.row = row_addr;         //row of seed pixel
    addr.charge = seed_charge;   //charge of seed pixel
    return addr;
  }
```

Hardware cluster class

Listing 6.16: Constructor and destructor function of the `PXDHardwareCluster` class. These two functions will be executed every time the class is called. The constructor at the beginning and the destructor at the end to free requested memory space. The lines 4 to 6 assign the values parsed to the class from the unpacker (listing6.13 line 72) to the cluster own variables. Line 11 starts the for loop which writes each pixels column, charge and row values to the arrays allocated in line 8 to 10. The ∼`PXDHardwareCluster` destructor, called in line 19, frees the memory space allocated for the arrays.

```
1  PXDHardwareCluster::PXDHardwareCluster(unsigned short* pixelU, unsigned short*
        pixelV, unsigned char* pixelQ, unsigned int nrPixel, unsigned char
       seedCharge, unsigned short seedU, unsigned short seedV, unsigned int
       clusterCharge, VxdID vxdID, unsigned int chipID):
2      //Initializes the variables declared in this class with the parsed ones
           from the
3      //function call in the PXDHardwareClusterUnpacker lisitng 3.12
4      m_nrPixel(nrPixel), m_pixelU(0), m_pixelV(0), m_pixelQ(0), m_seedCharge(
           seedCharge),
5      m_seedU(seedU), m_seedV(seedV), m_clusterCharge(clusterCharge), m_vxdID(
           vxdID),
6      m_chipID(chipID)
7  {
8    m_pixelU = new unsigned short[m_nrPixel];
9    m_pixelV = new unsigned short[m_nrPixel];
10   m_pixelQ = new unsigned char[m_nrPixel];
11   for (unsigned int i = 0 ; i < m_nrPixel ; i++) {
12     m_pixelU[i] = pixelU[i];
13     m_pixelV[i] = pixelV[i];
```

```
    m_pixelQ[i] = pixelQ[i];
  }
};

//Deletes all memory space requested for this classes arrays.
PXDHardwareCluster::~PXDHardwareCluster()
{
  delete m_pixelU;
  delete m_pixelV;
  delete m_pixelQ;
};
```

To access the so saved data in ROOT one needs to define functions which just return the variables data was saved to. This is done in the header file of this class. Part of this file are shown in listing 6.17.

Listing 6.17: A section of the `PXDHardwareCluster` classes header file listing the declared variables and the functions needed to access them.

```
  protected:
    unsigned int m_nrPixel, m_clusterCharge, m_chipID;
    unsigned short* m_pixelU; // [m_nrPixel] /**< Buffer of size m_nrPixel
        shorts  */
    unsigned short* m_pixelV; // [m_nrPixel] /**< Buffer of size m_nrPixel
        shorts  */
    unsigned char* m_pixelQ;  // [m_nrPixel] /**< Buffer of size m_nrPixel
        shorts  */
    unsigned char m_seedCharge;
    unsigned short m_seedU, m_seedV;
    VxdID m_vxdID;

    //returns the number of pixel in the cluster
    unsigned int getNrPixels() const { return m_nrPixel; }

    //returns the row coordinate of pixel j
    unsigned short getPixelU(unsigned int j) const { return (m_pixelU[j]); }

    //returns the column coordinate of pixel j
    unsigned short getPixelV(unsigned int j) const { return (m_pixelV[j]); }

    //returns the charge of pixel j
    unsigned char getPixelQ(unsigned int j) const { return (m_pixelQ[j]); }

    //returns the seed charge of the cluster
    unsigned char getSeedCharge() const { return m_seedCharge; }

    //returns the seed pixels row address
    unsigned short getSeedU() const { return m_seedU; }

    //returns the seed pixels column address
    unsigned short getSeedV() const { return m_seedV; }

    //returns the total cluster charge
    unsigned int getClusterCharge() const { return m_clusterCharge; }

    //returns the VXD ID of the cluster
    VxdID getVxdID() const { return m_vxdID; }

    //returns the DHP ID of the cluster
    unsigned int getChipID() const { return m_chipID; }
```

Square shaped cluster generator

Listing 6.18: Essential part of the C++ code which generates a square cluster shaped pattern. The first for loop starting in line 2, loops over all 4 DHPs. In line 4 the first frame starts and the cluster header is added in line 6 and 7. The cluster is build of 9 pixels. Since the first word before the pixel words is a start of row word, the loop in line 17 repeats 10 times (one start of row and 9 pixel words). The if condition in line 27 forces the column address to be decreased by 3 and the increment row flag to be one when the fourth or the seventh pixel is reached. This behavior assures that the cluster shape is a square.

```
 1  //Generate data on all 4 DHP
 2  for(int nr_dhp_frames = 0 ; nr_dhp_frames < 4 ; nr_dhp_frames++){
 3    int row_add = 11;
 4    start_frame();
 5      //FCE Header, one per DHP
 6      crc_value=(0x0 << 31) | (DHC_FRAME_HEADER_DATA_TYPE_FCE_RAW << 27) | (
          dhe_ids_list[0] & 0x3F) << 20 | (0x00 << 16) | (trigger_nr & 0xFFFF);
 7      append_int32(crc_value);
 8      word_count = word_count + 2;
 9      int dEdx = 2;
10      //Generate 4 cluster per DHP
11      for(int nr_of_cluster = 0 ; nr_of_cluster < 4 ; nr_of_cluster++){
12        int increment_row_flag = 0, column_address = 0;
13        int adc_value = 0;
14        column_address = 31;
15
16        //Generate square pattern with 9  pixel, loop over 10 due to the start
            of cluster word
17        for (int pixel_in_cluster = 0 ; pixel_in_cluster < 10 ; pixel_in_cluster
            ++){
```

```
        //Start of cluster word
        if(pixel_in_cluster==0 ){
          crc_value = (0 << 15) | (1 << 14) | (dEdx & 0x3) << 12 | (dhp_ids_list[nr_dhp_frames] & 0x3) << 10 | (row_address & 0x3FF);
          append_int16(crc_value);
          word_count++;
        //Pixel words
        }else{
          adc_value = 30;
          increment_row_flag = 0;
          if(pixel_in_cluster == 4 || pixel_in_cluster == 7 ){
            increment_row_flag = 1;
            column_address = column_address - 3;
          }
        //Seed pixel charge
        if(pixel_in_cluster==5){
          adc_value = 50;
        }
        crc_value = (1 << 15) | (increment_row_flag << 14) | (column_address & 0x3F) << 8 | (adc_value & 0xFF);
        append_int16(crc_value);
        word_count++;
        column_address++;
        }
      }
      row_address = row_address + 200;
    }
    //Payload adder in order to stay 32bit aligned
    if((word_count % 4) != 0 && (word_count % 2) != 0){
      crc_value = (0 << 15) | (1 << 14) | (dEdx & 0x3) << 12 | (dhp_ids_list[nr_dhp_frames] & 0x3) << 10 | (row_add & 0x3FF);
      append_int16(crc_value);
    }
    word_count = 0;
  add_frame_to_payload();
```

V-shaped cluster generator

Listing 6.19: Section of the C++ code that generates a V-shaped cluster pattern. The main functions are already explained in listing 6.6. The main difference is the second start of row word in the cluster (line 24). It generates the clusters row for the clusters seed pixel (figure 5.2 dark red pixel). In line 40 the column address for the pixels is customized to form a V-shaped pattern.

```
1 //Generates data for all 4 DHPs
2 for(int nr_dhp_frames = 0 ; nr_dhp_frames < 4 ; nr_dhp_frames++){
```

```
int row_add = 11;
start_frame();
  //FCE Header, one per DHP
  crc_value=(0x0 << 31) | (DHC_FRAME_HEADER_DATA_TYPE_FCE_RAW << 27) | (
      dhe_ids_list[0] & 0x3F) << 20 | (0x00 << 16) | (trigger_nr & 0xFFFF);
  append_int32(crc_value);
  word_count = word_count + 2;
  int dEdx = 2;
  //Generates 4 cluster per DHP
  for(int nr_of_cluster = 0 ; nr_of_cluster < 4 ; nr_of_cluster++){
    int increment_row_flag = 0, column_address = 31;
    int adc_value = 0;
    //Generates 7 pixels for each cluster pattern, loops to 9 due to start
        of cluster and start of row words
    for (int pixel_in_cluster = 0 ; pixel_in_cluster < 9 ; pixel_in_cluster
        ++){
      if(pixel_in_cluster==0 || pixel_in_cluster==6){
        //Start of cluster word
        if(pixel_in_cluster==0){
          crc_value = (0 << 15) | (1 << 14) | (dEdx & 0x3) << 12 | (
              dhp_ids_list[nr_dhp_frames] & 0x3) << 10 | (row_address & 0
              x3FF);
          append_int16(crc_value);
          word_count++;
        }
        //Start of row word
        if(pixel_in_cluster==6){
          row_address = row_address + 1;
          crc_value = (0 << 15) | (0 << 14) | (dEdx & 0x3) << 12 | (
              dhp_ids_list[nr_dhp_frames] & 0x3) << 10 | (row_address & 0
              x3FF);
          append_int16(crc_value);
          word_count++;
        }
      }else{
        adc_value = 30;
        increment_row_flag = 0;
        if(pixel_in_cluster == 2 || pixel_in_cluster == 3 ){
          increment_row_flag = 1;
        }
        //Seed pixel charge
        if(pixel_in_cluster==7){
          adc_value = 50;
        }
        if(pixel_in_cluster==4 || pixel_in_cluster==5 || pixel_in_cluster
            ==8){
          column_address++;
        }
        //Pixel word
```

```
          crc_value = (1 << 15) | (increment_row_flag << 14) | (column_address
               & 0x3F) << 8 | (adc_value & 0xFF);
          append_int16(crc_value);
          word_count++;
        }
      }
      row_address = row_address + 200;
      }
      //Payload adder in order to stay 32bit aligned
      if((word_count % 4) != 0 && (word_count % 2) != 0){
        crc_value = (0 << 15) | (1 << 14) | (dEdx & 0x3) << 12 | (dhp_ids_list
            [nr_dhp_frames] & 0x3) << 10 | (row_address & 0x3FF);
        append_int16(crc_value);
      }
    word_count = 0;
  add_frame_to_payload();
}
```

Bibliography

[Abe01] K. et al. Abe. Observation of large cp violation in the neutral b meson system. *Phys. Rev. Lett. 87, 091802*, 2001.

[Ber06] C. Berger. *Elementarteilchenphysik, 2. Auflage.* Springer, 2006.

[Cab63] Nicola Cabibbo. Unitary symmetry and leptonic decays. *Phys. Rev. Lett. 10, 531*, 1963.

[Col12] ATLAS Collaboration. Observation of a new particle in the search for the standard model higgs boson with the atlas detector at the lhc. *Physics Letters B, Vol. 716, Issue 1*, 2012.

[Col13] Belle Collaboration. Kekb accelerator achievements of kekb. *Prog. Theor. Exp. Phys., 03A001*, 2013.

[DU10] Z. Dolezal and S. Uno. *BELLE II Technical Design Report Motivation Part.* Belle II Colaboration, 2010.

[ea03] S. K. Choi et al. Observation of a narrow charmoniumlike state in exclusive $b^{\pm} \rightarrow k^{\pm}\pi^{+}\pi^{-}j/\psi$ decays. *Phys. Rev. Lett. 91, 262001*, 2003.

[ea04] B. Aubert et al. Direct cp violating asymmetry in $b_0 \rightarrow k^{+}\pi^{-}$ decays. *Phys. Rev. Lett. 93, 131801*, 2004.

[ea08] C. Amsler et al. Particle data group. *Phys. Lett. B667, 1*, 2008.

[eaBC05] B. Aubert et al. (BABAR Collaboration). Observation of a broad structure in the $\pi^+\pi^- j/\psi$ mass spectrum around $4.26 gev/c^2$. *Phys. Rev. Lett. 95, 142001*, 2005.

[eaBC13] M. Ablikim et al. (BESIII Collaboration). Observation of a charged charmoniumlike structure in $e^+e^- \to \pi^+\pi^- j/\psi$ at $\sqrt{s} = 4.26 gev$. *Phys. Rev. Lett. 110, 252001*, 2013.

[Fra78] P. Franzini. Upsilon resonances. *Ann. Rev. of Nuclear and Particle Science*, 1978.

[Gro14] Particle Data Group. *Particle Physics Booklet.* Chinese Physics C, 2014.

[Gut81] Alen H. Guth. Infiationary universe: A possible solution to the horizon and fiatness problems. *Physical Review D Vol 23*, 1981.

[HH57] C. S. Wu E. Ambler R. W. Hayward D. D. Hoppes and R. P. Hudson. Experimental test of parity conservation in beta decay. *Phys. Rev. 105, 1413–1415*, 1957.

[Hig64] Peter Higgs. Broken symmetries, massless particles and gauge fields. *Phys. Rev. Lett. 12, 2*, 1964.

[Ins10] KEK Insitute. *http://www.desy.de/sites2009/site www-desy/content/e421/e55046/e157108/e157121/SuperKEKB-BelleII ger.jpg.* Belle II Collaboration, 2010.

[JHCJWCVLF64] R. Turlay J. H. Christenson J. W. Cronin V. L. Fitch. Evidence for the 2π decay of the k_2^0 meson. *Phys. Rev. Lett. 13, 138*, 1964.

[Kei01] Markus Keil. Pixeldetektoren aus silizium und cvd-diamant zum teilchennachweis in atlas bei lhc. *Dissertation*, 2001.

[LEE57] TSUNG DAO LEE. Weak interactions and nonconservation of parity. *Nobel Lecture*, 1957.

[LY56] T. D. Lee and C. N. Yang. Question of parity conservation in the weak interactions. *Phys. Rev. Lett. 104, 254*, 1956.

[PR14] Scholz Zetsche Povh Rith. *Teilchen und Kerne 8. Auflage.* Springer, 2014.

[Sak67] A.D. Sakharov. Violation of cp invariance, c asymmetry, and baryon asymmetry of the universe. *Pisma Zh.Eksp.Teor.Fiz.*, 1967.

[SB11] Maurizio Spurio Sylvie Braibant, Giorgio Giacomelli. *Particles and Fundamental Interactions An Introduction to Particle Physics.* Springer, 2011.

[Sch06] Franz Schwabl. *Statistische Mechanik.* Springer, 2006.

[Uch12] Tomohisa Uchida. Sitcp manual. *SiTCP_eng.pdf*, 2012.

[uTM73] Makoto Kobayashi und Toshihide Maskawa. Cp-violation in the renormalizable theory of weak interaction. *Progress of Theoretical Physics. Band 49, Nr. 2*, 1973.

[Xil15] Xilinx. Aurora 64b/66b v10.0. *pg074-aurora-64b66b.pdf*, 2015.

Eidesstattliche Versicherung

Ich versichere, dass ich die vorliegende Arbeit selbstständig geschrieben und deren Inhalt wissenschaftlich erarbeitet habe. Außer den im Literaturverzeichnis angegebenen Quellen habe ich keine weiteren Hilfsmittel verwendet.

Klemens Lautenbach, Gießen den
September 24, 2015

Danksagung

Besonders danken möchte ich Herrn Prof. Dr. Wolfgang Kühn und Herrn Dr. Jens Sören Lange AR, die es mir ermöglicht haben, diese Arbeit an einem so großen Experiment wie dem Belle II zu schreiben und mir mit Rat und Tat zur Seite standen.

Außerdem danken möchte ich Herrn Dr. Björn Spruck der mich während der Programmierarbeiten mit konstruktiver Korrespondenz unterstützte.

Weiterer Dank geht an die Mitglieder der AG Kühn die zu einer sehr angenehmen Arbeitsatmosphäre beitrugen und auch außerhalb der Arbeit, in einer abendlichen Runde, für gute Unterhaltung sorgten.

Des weiteren möchte ich meiner Familie danken, die immer ein offenes Ohr für mich hatte und mich während des Studiums unterstützte.