Online Monitoring for the Silicon Tracker of the LHCb Experiment

Master Thesis

Mathematisch-naturwissenschaftliche Fakultät der

Universität Zürich

Nicola Chiapolini

Prof. Dr. Ulrich Straumann
Dr. Olaf Steinkamp
Dr. Jeroen van Tilburg
Dr. Mark Tobin

Zürich
Juli 2009

Abstract

While the LHCb experiment acquires data, the detector and the data quality are continuously monitored. The first part of this thesis describes a package that was developed for managing monitoring pages. This package is a subdetector independent tool originally developed for the online monitoring system of the Silicon Tracker.

In the second half of this thesis, methodes to calculate the common mode subtracted noise in the Silicon Tracker are compared. Different possible adjustments are evaluated and possible improvements presented.

Contents

Chapter 1

Introduction

When the LHC restarts in a few months, the largest physics experiment in history will produce large amounts of data. While acquiring data it is important to get fast feedback that the detectors are working correctly and the data produced is of good quality.

To achieve this, a subset of the data produced by the detectors is continuously aggregated into statistical representations. These can then be monitored by scientists on shift. To allow for fast feedback on the data quality the monitoring system is run online. This means it is run on the data while it is collected.

As part of this thesis, the online monitoring system for the Silicon Tracker of the LHCb experiment was extended and improved. A tool was developed to automate the creation and configuration of the large number of histogram pages required to monitor the detector performance. The tool was developed such that it is detector independent and can be used by all LHCb subdetectors. In the second part of this thesis, different common mode subtraction algorithms were studied to improve the monitoring of common mode subtracted noise.

1.1 The Silicon Tracker

The LHCb experiment and the physics goals of this experiment have been described in detail in [9].

The Silicon Tracker consists of two detectors. One is the Tracker Turicensis (TT) and the other is the Inner Tracker (IT). Both detectors use silicon microstrip technology for the tracking of charged particles through the experiment.

1.1.1 Readout

Figure 1.1 shows a schematic of the readout process used for the Silicon Tracker. A detailed description of the readout hardware can be found in [15].

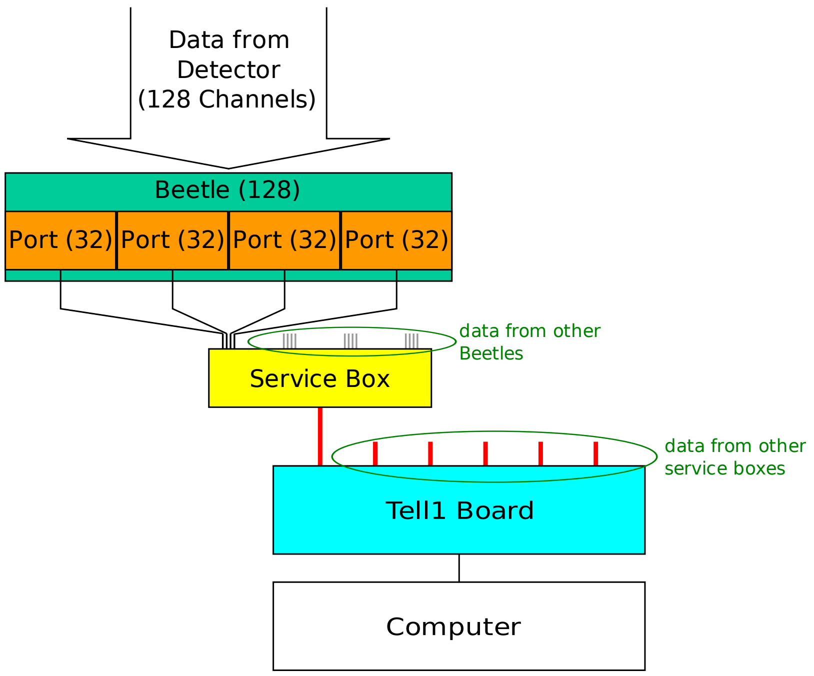


Figure 1.1: The readout process.

The electrical signal from the detector is read out by an electronic component called hybrid. Depending on the detector a hybrid contains either 3 (IT) or 4 (TT) Beetle chips. Every Beetle chip has 4 readout ports. Each port transfers the data of 32 detector channels to a Service Box. There the analog pulse height is converted into a digital signal. The digitised data is then sent to the Tell1 readout board through an optical link.

Table 1.1 lists the numbers of components in each of the detectors.

	TT	IT
channels in total	143360	129024
Beetle chips in total	1120	1008
Tell1s in total	48	42

Table 1.1: Numbers of components for the TT and IT

Beetle Chip

At the LHC, proton-proton-collisions occur with a bunch-crossing frequency of 40 MHz. The Beetle chip stores pulse-height information from the last 160 bunch crossings in an analogue pipeline memory. This buffer ensures a nearly dead time free readout. The position of an event in the pipeline is identified by the 8-bit pipeline column number (PCN). If the readout of an event is triggered, the PCN will be encoded into the first two bits sent by each of the 4 ports of a Beetle chip. Together with an unused third and fourth bit this is the header sent before the data of a port. Details about the Beetle chip and this encoding can be found in [10].

Tell1 Board

Each Tell1 board processes the output from 24 Beetle chips corresponding to 3072 detector channels in total. The Tell1 board synchronises the data from the different Beetles, calculates pedestals and common mode and subtracts these from the data. After these steps, a cluster finding algorithm detects hits and the data is zero-suppressed. All these calculations on the Tell1 boards are implemented on FPGAs.

In the last step, the Tell1 board encodes the data in raw banks and sends these to the data acquisition computers. There are three different raw bank formats ([4], [5], [6]):

zero-suppressed (ZS) Contains the cluster positions and the ADC values of the corresponding detector channels. This is the bank format used for normal data taking.

non zero-suppressed (NZS) Contains the ADC values of all channels in the detector. Neither pedestal nor common mode will be subtracted for this data. This bank format is used, for example, to verify the data processing on the Tell1.

error The error bank data provides detailed information for events that have shown synchronisation errors. It is sent automatically if an error is detected by the Tell1.

1.1.2 Noise

The readout electronics registers an ADC value for each detector channel, even when no particle has passed through the detector. This output value fluctuates around a constant pedestal. The rms of this fluctuation is called the raw noise.

Pedestal

The pedestal is different for each detector channel but is constant over time. Therefore, it can be determined by calculating the mean value from a set of events without hits.

As discussed in [7] it was found that the pedestal depends on the parity of the pipeline column number. Events with an odd PCN have a systematically higher pedestal than events with an even PCN. In addition, the PCN bits sent in the header influence the pedestal of the first detector channels in a port (header crosstalk). As there are 4 different configurations for the header bits and 2 parity states for the PCN, this gives in total 8 possible configurations. The brute force approach to remove these effects, is to calculate 8 separate pedestals, one for each configuration. This was tested in the second part of this thesis but is not yet implemented in the hardware or software used for data taking or monitoring.

Common Mode Noise

A second contribution to the output signal of the Beetles, is the common mode. The effect from the common mode changes from event to event but is correlated for all detector channels in one port. The common mode is calculated by fitting a straight line to all signal values from a given port.

After the pedestal and the common mode are subtracted, the output signal from a given detector channel still show a random fluctuation around zero. The rms of this distribution is called common mode subtracted noise (CMS noise).

1.1.3 Analysis Software

The LHCb software is based on the Gaudi framework described in [2]. It was developed for high energy physics applications and provides functionality for different contexts, such as file access, run-time configuration or histogram creation. The run-time configuration is based on job option files. For histogram creation Gaudi uses the ROOT data analysis framework [17].

The LHCb software is grouped into packages and projects. As explained in [16], a package is a set of files, organised according to some directory structure, which provides some well-defined, circumscribed functionality. Most LHCb packages contain code that can be compiled into one or more libraries. A project on the other hand is a set of packages that are grouped together according to some functionality.

The most important projects for this thesis are the Online and the Lbcom components projects. The Online components project groups together components and infrastructure software needed for running data processing applications in the Online computer farm where the monitoring is run. The

Lbcom components project groups together components shared by data processing applications as for example the different algorithms used for data acquisition or monitoring.

Decoding of Raw Banks

Before the data sent by a Tell1 board can be analysed, the raw bank is decoded. This is done by a set of algorithms that fill C++ objects containing the information from the raw bank. An example of the decoding can be found in [11] where the decoding of zero-suppressed data is described. For the monitoring the following data sets are important:

LiteClusters Contains the positions of the clusters (no ADC values) [11].

Clusters Contains the full information available from the zero-suppressed raw bank. In addition to the position of the cluster this contains the ADC values of each detector channel in a cluster [11].

Summary Contains information on Tell1 boards that are missing or have problems, together with the PCN [11].

Errors Contains the information transferred in the error banks [6].

Full Contains the information transferred in the non zero-suppressed raw banks [5].

Tell1 Emulator

A special component of the Silicon Tracker analysis software is the Tell1 emulator. This application provides a bit-correct emulation of the processing performed on the Tell1 boards. Using the emulator, one can reconstruct data at different stages in the Tell1 processing or generate the zero-suppressed information from non zero-suppressed data.

Chapter 2

Monitoring Framework

2.1 Basic Architecture

This section contains a brief description of the monitoring framework. A full description is given in [1].

The monitoring system is run on a dedicated computer farm and consists of three main parts; The monitoring algorithm, a database (HistDB) and a graphical user interface (Presenter). The monitoring is controlled by the same control software (PVSS) which is used to control the detector hardware.

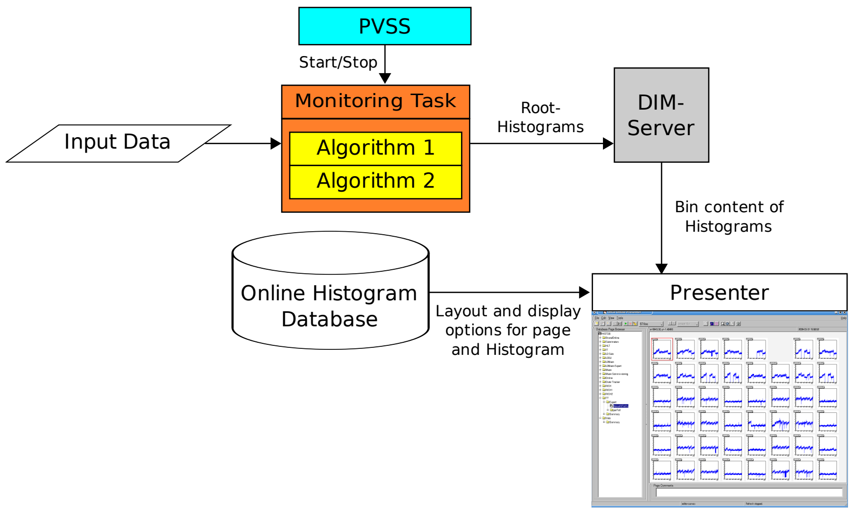

PVSS
Start/Stop
Monitoring Task
Algorithm 1
Algorithm 2
Input Data
Root-Histograms
DIM-Server
Bin content of Histograms
Online Histogram Database
Layout and display options for page and Histogram
Presenter

Figure 2.1: The monitoring system.

Figure 2.1 shows how the different parts of the monitoring system are connected. The detector data sent to the monitoring farm is processed by a monitoring algorithm. This algorithm then publishes a set of ROOT histograms that can be displayed by the Presenter. The information on how to display the histograms is stored in the HistDB.

To start an algorithm automatically when the detector starts taking data, a monitoring task has to be added in the control software of the corresponding detector. A task can run several algorithms and the control software can run several tasks for each subdetector.

A detailed description of how an algorithm is added to the monitoring can be found in the Section A.1.

2.1.1 Presenter

The Presenter is a graphical user interface used to display the ROOT-based histograms published by the monitoring algorithms. To display histograms, a page has to be defined. A page contains one or more histograms in a given layout and with given display settings. When the page is selected in the presenter, the page and histogram configuration are loaded from the HistDB. Then the histogram data published by the monitoring algorithm is read and the pages are displayed. As all display properties are stored in the HistDB, the Presenter loads only the bins and their content from the histograms published by the algorithms. The display properties specified by the algorithm will be ignored. The Presenter is described in [19].

2.1.2 The Online Histogram Database

The online histogram database (HistDB) is an SQL based database storing the display settings and configurations for all histograms and pages. All histograms published by an algorithm need to be registered in the HistDB. Histograms are identified by the combination of task name, algorithm name and histogram name. To allow easy adjustment of display properties for several histograms, histograms can be grouped into sets with common display options. A set is defined by the occurrence of `_$` in the histogram name.

The database can be accessed through a web interface[1] as well as through the presenter.

The OnlineHistDB package contains the SQL definition of the HistDB itself and provides the source code for the different interfaces to the database. The package contains the C++ interface used by the Presenter or HistDBPython (see Section 3) as well as the PHP code needed for the web interface. The OnlineHistDB package is described in detail in [3].

2.2 The ST Monitoring Code

The monitoring of the Silicon Tracker is split into different algorithms depending on the data they process. All monitoring algorithms can be found in the STMonitors package, which is part of the Lbcom components project. At the time of writing, it contains the algorithms shown in Table 2.1.

[1]The web interface can be found at: `http://lbhistogramdb.cern.ch/`

Algorithm	Control-Tasks
STClusterMonitor	Monitoring of zero-suppressed data with clusters
STErrorMonitor	Monitoring of error banks
STLiteClusterMonitor	Simple monitoring of zero-suppressed data with clusters
STNZSMonitor	Monitoring of NZS data
STSummaryMonitor	Monitoring of the information in the event summary class
STTAEClusterMonitor	Monitoring of time aligned events (TAE) used for detector calibration
STZSMonitor	Legacy algorithm for monitoring clusters

Table 2.1: Algorithms in the STMonitors package.

The three algorithms STNZSMonitor, STSummaryMonitor and STErrorMonitor were developed or modified as part of this thesis. They are described in the following sections.

2.2.1 STNZSMonitor

The STNZSMonitor is used to calculate the raw and CMS noise from non zero-suppressed data. The algorithm calculates the noise for each channel and publishes one histogram for each Tell1 board.

Calculating the Noise

The noise is defined as the RMS of the distribution of the ADC values and can be calculated as

$$\sigma = \sqrt{\frac{1}{N} \cdot \sum_{i=1}^{N} (adc_i^2) - \left(\frac{1}{N} \cdot \sum_{i=1}^{N} adc_i\right)^2}, \tag{2.1}$$

where N is the total number of events processed. For each channel, the STNZSMonitor needs to store the mean of all ADC values ($\frac{1}{N}\sum adc$) as well as the mean of all squared ADC values ($\frac{1}{N}\sum (adc^2)$). After a given number of events the noise can then be calculated using relation 2.1.

The sensitivity of such a noise calculation to the last event will decrease with the increasing number of processed events. To avoid this the STNZSMonitor implements a moving average as

$$m_i = m_{i-1} \cdot \frac{n-1}{n} + adc_i \cdot \frac{1}{n} \qquad \text{with } n = \begin{cases} i & \text{if } i < N \\ N & \text{if } i \leq N \end{cases},$$

where m_i is the pedestal after the i^{th} event and N is the following period. After i reaches N, this formula gives an exponentially decreasing weight to old events. The moving average is calculated in the same way for the squared ADC values.

Subtracting Common Mode

Originally, the STNZSMonitor produced only one set of noise histograms. It could be used to monitor the raw noise when running on the raw data. To monitor the CMS noise, the raw data had to be preprocessed by the Tell1 emulator to calculate the common mode subtracted values.

As running the Tell1 emulator significantly increased the complexity of the monitoring it was decided to add a simple common mode subtraction algorithm to the STNZSMonitor. This allows the calculation of the CMS noise without having to use the Tell1 Emulator. Different common mode subtraction algorithms were evaluated. This is described in the second part of this thesis (Section 4). The final implementation of the selected algorithm could not be finished in the timescale of this thesis.

Speed Optimisation

While evaluating the different algorithms it became evident that the ST-NZSMonitor was too slow and had to be optimised for speed. Running the algorithm in a profiler[2] showed that the way the ADC values were stored and accessed had a huge impact on the performance.

Change	no CMS [s/event]	with CMS [s/event]
initial	0.167	0.374
store in vector	0.075	0.186
iterator access	0.039	0.080

Table 2.2: Performance of STNZSMonitor before and after optimisation

Table 2.2 shows the estimate for the time needed to process one event at different optimisation stages. Originally the Tell1 data was stored in *maps*[3]. Replacing the maps by vectors gave a performance gain of a factor of two. Another factor of two could be achieved by replacing the direct element access[4] by iterator access.

Replacing the maps by vectors has the disadvantage that the Tell1 boards can no longer be accessed by the original Tell1 identifiers. Vector indices need

[2]A tool for software performance analysis.

[3]Maps are containers from the C++ Standard Template Library. Maps store elements by the combination of a key value and a mapped value.

[4]This is done by using brackets (`[]`)

to be consecutive integers starting with 0 and the original Tell1 identifiers are not consecutive. The downside of using the iterator access is that memory access errors are harder to find.

2.2.2 STSummaryMonitor and STErrorMonitor

The STSummaryMonitor and STErrorMonitor algorithms analyse the information from Tell1 boards that have errors. The STSummaryMonitor publishes histograms based on the summary information. Since the summary information is filled as part of the zero-suppressed decoding, the histograms are only available for zero-suppressed data. STSummaryMonitor publishes three histograms:

- The PCN distribution allows to check that no position in the pipeline is selected systematically more often than the other positions.
- The total data size allows to check that the expected amount of data is transferred for each event.
- The error information counts the accumulated number of errors.

For every event processed, one entry is added to each of these histograms. As an example, Figure 2.2 shows a screenshot from the presenter displaying the PCN distribution histogram.

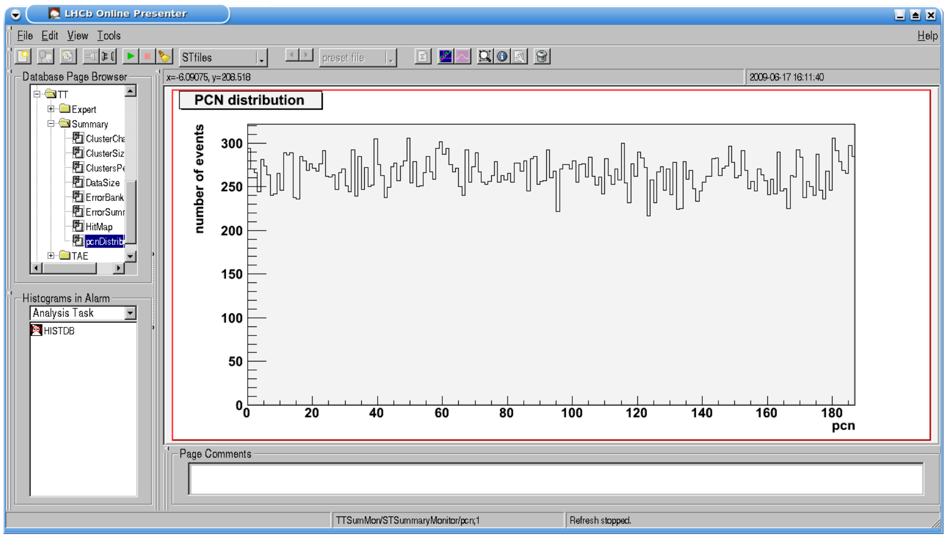


Figure 2.2: The PCN distribution histogram as displayed in the presenter.

The STErrorMonitor processes the error information in the error bank and publishes two types of histograms. Firstly, an overview histogram is published that counts the number of error banks per Tell1. In addition, a

2D histogram is published for each Tell1 that sends an error bank. These histograms display the error type versus the optical link number.

2.3 The ST Monitoring Tree

As explained in Section 2.1, histograms are displayed on pages. These pages are organised in a tree of pages and folders. The organisation of this tree should allow for simple and fast access to the needed information.

The monitoring tree is split into two separate top levels, one for IT and one for TT. Below this top level, each tree is split into two branches as shown in Table 2.3. The summary branch contains histograms used by the person on shift to check the status of the detector. In case of problems, the expert can access the pages in the expert branch to get more detailed information about a specific Tell1 board or detector region. The initial lists of pages for the two branches are given in 2.3. Both branches are likely to be expanded based on the experiences gained from running the detector.

Summary
- ClusterCharge
- ClusterSize
- Clusters per TELL1
- ClustersPerEvent
- DataSize
- ErrorBanks
- ErrorSummary
- HitMap
- pcnDistribution

Expert
- NoiseAllTell1s
- perTell
    - tell1
        - ErrorType
        - Noise
    - tell2
        - ErrorType
        - Noise
    - · · ·

Table 2.3: The TT monitoring tree.

For detector commissioning and debugging a third branch is available in each tree. This third branch is the TAE branch, containing pages for monitoring time aligned data. In time alignment mode several consecutive bunch crossings are read out for each trigger.

Chapter 3

Managing Pages with HistDBPython

As shown in the previous section, the monitoring tree contains hundreds of pages. Often, these pages are very similar and it would be tedious to set up each of them manually using the presenter or the web interface.

The OnlineHistDB package contains C++ classes that allow pages to be set up and configured by implementing a C++ algorithm. The downside of this approach is that one needs to write a separate C++ algorithm for every set of pages and has to recompile the source code every time something in the definition of these pages changes.

To avoid these downsides, generic algorithms were written that set up and configure pages according to a given definition. These algorithms are collected in the HistDBPython package. HistDBPython makes use of the power that GaudiPython offers when writing option files. GaudiPython is a Python interface to the Gaudi framework and allows the use of Python code inside the option files. Therefore, one can use variables and loops to simplify the definition of similar pages.

The remaining part of this Section will document the HistDBPython package and its algorithms. Appendix A.2 contains a step-by-step guide for this package and extensively commented example files.

3.1 Example Use Case

This section illustrates the use of the HistDBPython package, using the Noise pages for TT as an example.

To monitor the raw noise, STNZSMonitor produces a histogram for each Tell1 board. These histograms are called `noise_$tell1`, `noise_$tell2` and so on up to `noise_$tell48`. The `_$` in the names allows to group all these histograms into one set. This allows the adjustment of display properties for all histograms in the set at once. Each histogram is displayed on the `Noise` page in the directory `/TT/Expert/perTell/tellX`, where X is the sequential number of the Tell1. This is the same number as used in the histogram name (e.g. `noise_$tellX`).

Only few parameters change between these pages. This means that all these pages can be set up by running the `CreateHistDBPages` algorithm of HistDBPython. When doing so, only one option file needs to be configured. Since these options are written in Python, one can use counters and loops to define all the similar pages. The file `example_page.py` in Appendix A.2.4 shows the code needed to achieve this. In addition, the code in this file adds a second histogram `cms_$tellX` for monitoring CMS noise to the page.

For each of the other algorithms an example file is included in Appendix A.2.4. The example files belong to different histograms and pages. They were selected as they use all the available features in HistDBPython.

3.2 Preparing HistDBPython for use

The HistDBPython package is part of the Online components project. It can be found at Online/HistDBPython[1].

The properties of the different algorithms in this package are explained below. General properties can be put into separate files that can be included by other option files. This allows to reuse these settings for different sets of pages.

Once the algorithms are configured, they can be run by calling `gaudi-run.py` with the specified option file.

3.2.1 Specifying the password

Since HistDBPython writes to the Histogram Database, one needs to provide a username and password. Since the password is needed for every access to the database, it was no option to ask the user to type in the password everytime an algorithm is run. On the other hand, the password could not be hard-coded in the source code of HistDBPython, as it would then appear in the source version control system (CVS) and be visible on the internet.

The algorithm is therefore configured to read the password from an additional file. This file is not included in CVS. It needs to be created by the user before running an algorithm. This solution allows to store the password without having to put it into the source code.

[1] `http://isscvs.cern.ch/cgi-bin/viewcvs-all.cgi/Online/HistDBPython/?root=lhcb&pathrev=cms`

3.3 Creating Pages

The algorithm to create pages is `CreateHistDBPages`. A page is defined by three main parts: a name and path, the number of histograms to show, and a layout that specifies how the histograms should be placed on the page.

A design choice for `CreateHistDBPages` was that one option file can only create pages with the same layout. Each option file therefore contains exactly one page layout defined. To store the page definitions, the option files contain two lists with the same number of entries. The first list (`PageNames`) gives the page names (including the path) of all pages that should be created. The second list (`HistoNames`) is two dimensional and contains a list of histograms for each page. This solution had to be used as there is no easy way in Gaudi Python to translate complex python structures into C++ objects.

The properties of this algorithm are:

PageNames The names of all pages to create. The names can be given with a path and the values of this list will be prefixed with the content of `PageBase`. The order of the pages in this list is important for `HistoNames`.

PageDoc The page documentation strings. The page documentation is displayed in the presenter and can be used to store information for the users. If the length of this list does not match the number of pages, the first string will be used for every page.

HistoNames The histograms that should be displayed on the different pages. For each page, a list with histogram identifiers is given. All pages need to have the same number of histograms as the page layout (`PageLayout`).

HistoLinks The matching of pages and histograms used for linking. The presenter can display a new page when a histogram is clicked upon. This property is used to define these links. Not all histograms need to have a link. As the linked pages are a property of the histogram, only one page can be linked with a given histogram even if the histogram is displayed on different pages.

PageLayout The layout used to display the histograms on the pages. For each histogram a list with 4 values gives X_{min}, Y_{min}, X_{max} and Y_{max}. One entry needs to exist for each histogram on the pages. If X_{min} is smaller than 0, the histogram will be drawn into the same frame as the previous histogram. This allows to overlay two histograms. As the same layout will probably be used for different pages this property is best defined in a separate layout file.

PasswordFile The location of the password file to use. The file can be given with absolute or relative path. The property is best defined in a separate file containing property common to multiple sets of pages.

PageBase A prefix for all pages. `PageBase` can be used to define the correct main folder for the project. This is especially useful if defined in a separate file containing properties common to multiple sets of pages.

HistoBase A prefix for all histogram names. `HistoBase` can be used to define a fixed part for all histogram names. This is especially useful if this part is common for different sets of pages and might be changed in the future (e.g. task and algorithm name). The property is therefore best defined in a separate file.

3.4 Configuring Histograms

Histograms are configured by adjusting the display options stored in the OnlineHistDB. These display options provide access to most properties of ROOT histograms. The data type of all display options is either integer, float or string. As an example, the option named `NDIVX` contains an integer specifying the number of divisions for the X-axis A full list of all display options available for the histograms and their data type can be found in [3].

The algorithm to set display options in the histogram database is `Set-HistDisplayOptions`. To define the display option, the algorithm uses a separate python dictionary for each of the three data types. Each dictionary maps the names of display options to their respective values.

In addition to the display options that should be set, the histograms for which these options are to be set need to be given. For this, an additional dictionary is used. This dictionary maps the histogram name to a second boolean value. If the boolean value is true, the display options given will be applied for the whole set of histograms.[2]

The properties of SetHistDisplayOptions are:

HistoNames The histograms or histogram sets that should be updated. For each entry, one can specify if only this histogram or the whole set should be adjusted.

UnsetOptions If this boolean is set to true the display options will be reset to defaults before applying the new options given in the file.

intOptions/floatOptions/stringOptions The display options of the corresponding type[3].

[2]As mentioned in Section 2.1.2 histograms can be grouped into sets. This allows convenient management for a large number of similar histograms.

[3]For a full list of available display options and their data type see [3].

xLabels/yLabels The labels to use for the bins on the X/Y-axis. Each bin can have its own label. `example_dopts.py` uses this to name the bins for the different error types (see Appendix A.2.4).

PasswordFile/HistoBase *as described in Section 3.3.*

3.5 Adding Histograms

To add new histograms to the database, `AddNewHistToDB` is used. As adding histograms is normally done only once, it is often easier to add histograms to the database using the presenter. The downside of using the presenter is that it can only access histograms that are published at the time. Histograms are only published if the corresponding data is available. Therefore, missing histograms cannot be added to the database using the presenter.

The monitoring task, the monitoring algorithm and the histograms can be added using `AddNewHistToDB`. This algorithm can only add histograms that belong to the same task and monitoring algorithm and are of the same type.

The properties of AddNewHistToDB are:

TaskName The name of the task running the algorithm.

AlgorithmName The name of the algorithm publishing the histograms.

HistoNames The histograms to be added to the database.

HistoType The type of the histograms. Only histograms of the same type can be added to the database with the same option file.

PasswordFile *as described in Section 3.3.*

3.6 Removing Pages

In some cases it is necessary to delete pages from the monitoring tree. The presenter and the web interface offer an efficient and safe way to remove a small number of pages. On the other hand this is a slow and painful process if one or more branches should be removed. For this, HistDBPython provides the `RemovePages` algorithm. To remove branches from the monitoring tree, one simply adds the top folders of each branch to the list of start folders of `RemovePages`. The branch will then be parsed recursively and all pages and subfolders are selected for removal.

Removing is always a dangerous task. Therefore different measures were taken to avoid accidental deletes. First of all the `RemovePages` has an option to simulate execution. This prints a list of all pages selected for removal but does not remove anything. In addition, the algorithm makes sure that a given startpage is neither the root of the monitoring tree nor a top level directory. The latter check can be switched off if necessary.

Properties of RemovePages are:

startFolders The root of each branch that should be deleted.

dryRun If this boolean is set to true (default), nothing will be written to the database. This should always be active in the first run to prevent accidental deletions. Special care needs to be taken when adjusting an existing option file.

protectTopLevel If this boolean is set to true, top level directories in `startFolders` will be ignored. Normally this should not be deactivated. Often it is safer to use the presenter for deleting an empty top level directory.

PasswordFile *as described in Section 3.3.*

Chapter 4

Common Mode Subtraction

As part of the monitoring of non zero-suppressed data, the common mode subtracted noise (CMS noise) is of interest. As mentioned in Section 1.1.3, the common mode subtraction can in principle be done using the Tell1 emulator. However, this requires installing additional software on the monitoring cluster and increases the complexity of the monitoring.

Therefore, the existing noise monitor STNZSMonitor (see Section 2.2.1) was modified to add a common mode subtraction algorithm. Different algorithms were implemented and their performance compared.

This chapter first explains the modifications made to the STNZSMonitor and describes the different common mode subtraction algorithms implemented. In Sections 4.3 and 4.4 the different algorithms are compared. The data used consists of non zero-suppressed data collected during detector commissioning. This data contains no hits from particles. As the LHC is not running yet, no data with hits is available. Therefore, Monte Carlo generated hits were added to the measured data in order to test the behaviour of the common mode algorithms in the presence of hits. All figures are shown for data from Tell1 board 15 which is a typical example. For figures showing the noise as a function of the channel numbers, channels 1500 to 1600 were selected. This range covers three ports (with 32 channels each) from two Beetle chips.

4.1 Calculation of Noise

4.1.1 Main Structure of STNZSMonitor

The main structure of the extended STNZSMonitor is shown in Figure 4.1. Compared to the initial version of the algorithm as described in Section 2.2.1, three main changes were implemented:

- Superposition of Monte Carlo generated hits on existing data was added.

- Calculation of separate pedestals for each PCN configuration (PCN-pedestals) was added.
- Simultaneous calculation of raw noise and CMS noise was implemented.

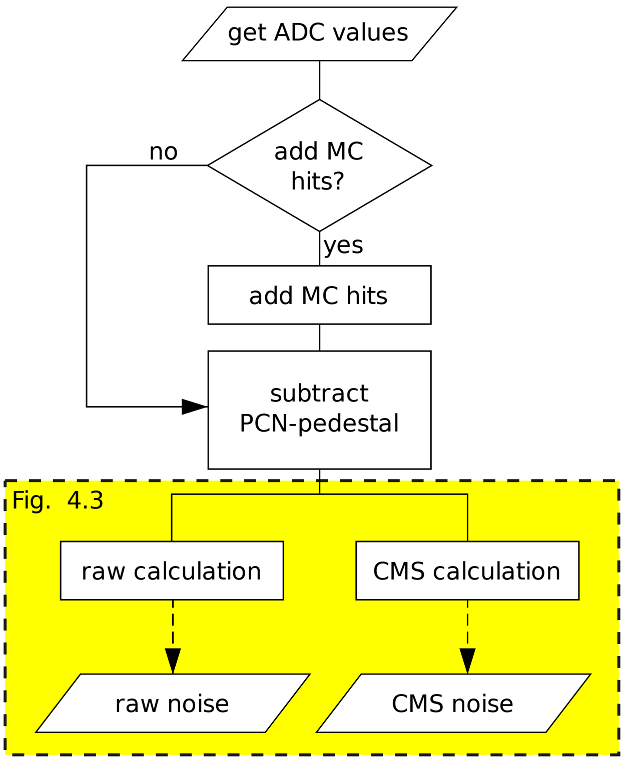


Figure 4.1: The main structure of the noise algorithm.

Simulating Hits

Due to lack of data with real hits, Monte Carlo hits are added for occupancy studies. The desired strip occupancy has to be specified for the simulation. The algorithm then adds the corresponding number of clusters, with a cluster-size distribution derived from a full Monte Carlo simulation of the LHCb detector.

Clusters are added by processing the non zero-suppressed input data channel by channel. Each channel has a probability p to be part of a cluster, where p is calculated from the strip occupancy:

$$p = \frac{\Omega}{\sum_{i=1}^{4} (f_i \cdot i)},$$

with Ω being the strip occupancy and f_i the fraction of clusters having size $i \, (i \in \{1, \ldots, 4\})$.

The fractions f_i were determined from a full LHCb Monte Carlo simulation of $B_s \rightarrow J/\psi\,\phi$ decays [14]. Table 4.1 shows the numbers used.

cluster size i	f_i (TT)	f_i (IT)
1	0.175	0.291
2	0.507	0.516
3	0.295	0.178
4	0.023	0.014

Table 4.1: Distribution of the cluster size.

For each channel in a cluster, 30 ADC counts are added to the non zero-suppressed input data[1].

PCN-dependant Pedestals

As described in Section 1.1.2, two effects on the raw noise (the header crosstalk and the PCN-parity dependent pedestals) can be removed by using separate pedestals depending on the PCN of the event. To study the effect this has on the CMS noise, the calculation of 8 PCN-dependant pedestals was added to the STNZSMonitor algorithm as an option. If this is activated, the pedestal corresponding to the PCN of the event (PCN-pedestal) gets subtracted before the noise calculation. The calculation of the PCN-pedestals is described in Section 4.1.2.

Calculating the Noise

After these preparations, the data is duplicated. One set of data is used to calculate the raw noise. The common pedestal and the common mode are subtracted from the second set of data and the CMS noise is then calculated. The basic concept of the noise calculation has been explained in Section 2.2.1.

A detailed description of the process used to calculate the CMS noise is given in Section 4.1.3 and in the descriptions of the different common-mode algorithms (4.2).

4.1.2 Initialisation of Pedestals and Raw Noise

Before the common mode can be subtracted, the pedestals and the raw noise for each channel have to be known. For this, the first $N + S$ events are used where

N =	number of events used to calculate the 8 PCN-pedestals.
S =	number of events used for the calculation of the common pedestal and the raw noise.

[1]30 ADC counts is to the expected signal height for a single-strip cluster

These events should not contain any hits as no hit removal is performed for these calculations. As Figure 4.2 shows, the first N events are used to determine the PCN-pedestals. First, the PCN configuration is determined for each event. The PCN-pedestals are then calculated as simple arithmetic means of the corresponding ADC values. There is no check that each PCN-pedestal is calculated from a minimal number of events.[2]

In the next S events, the PCN-pedestal is subtracted and the common pedestal and the raw noise for each channel are calculated as described in Section 1.1.2. If PCN-dependant pedestals are used, the common pedestal should be zero. When running the STNZSMonitor algorithm as part of the monitoring, these values need to be prepared beforehand.

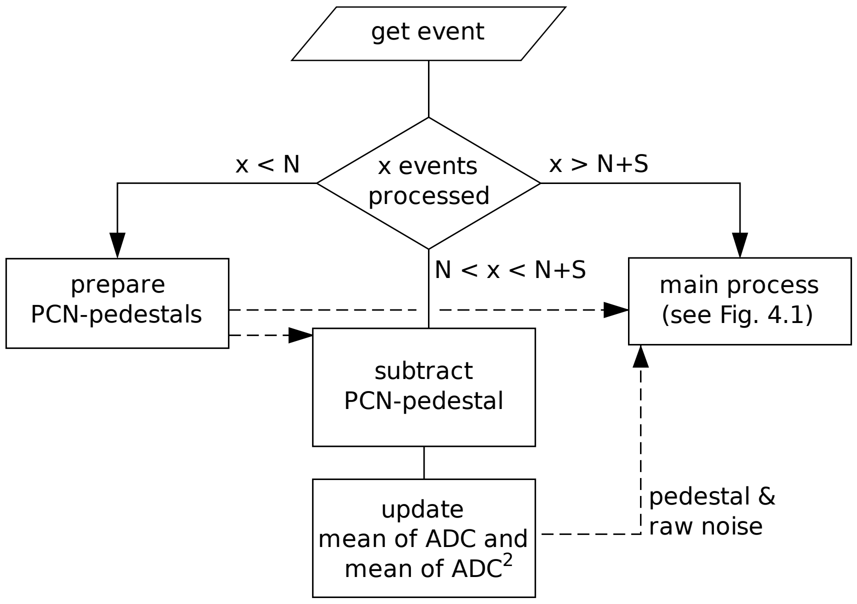


Figure 4.2: Preparation steps.

4.1.3 Calculation of CMS Noise

The CMS noise calculation is shown in Figure 4.3.

First, the common pedestal is subtracted from the data used for the common mode calculation. In the pedestal subtracted data, outliers are identified by an ADC count larger than 3 times the raw noise for the corresponding channel. Outliers are marked in the data so that the CMS algorithm can use or ignore this information depending on the options set. The data set is

[2]The frequencies with which the different PCN-configurations appear are not exactly equal. For example, In the first port only 4 of the 8 PCN configurations are possible and, therefore, occur with twice the frequency.

passed to the common-mode algorithm which calculates and subtracts the common mode.

From then on the raw data and the CMS data are treated in the same way. The noise is calculated based on moving averages (Eq. 2.1). Channels that have been marked are set to the current moving average of the ADC values for the corresponding channel. This approach was chosen as the algorithm would otherwise need to store the number of processed events separately for each channel instead of each Beetle. Thus, lower memory consumption and faster processing can be achieved.

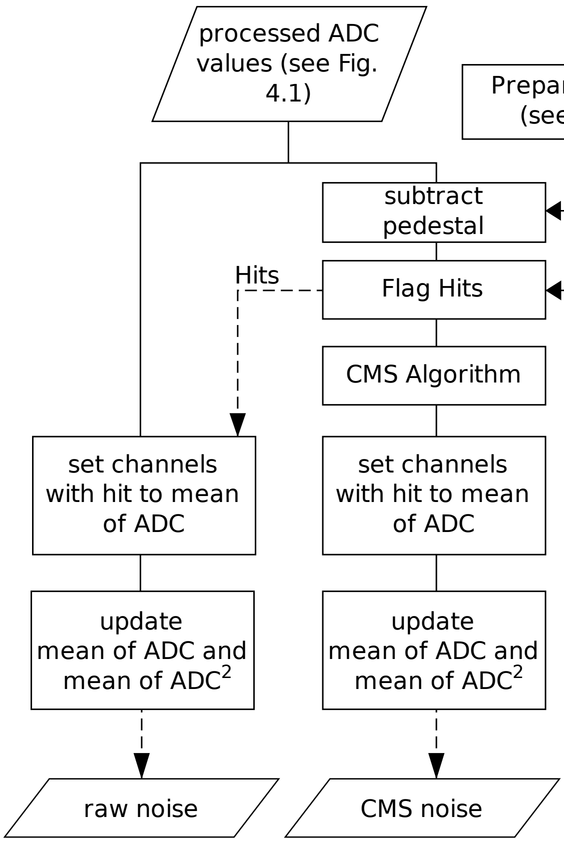

processed ADC values (see Fig. 4.1)
Preparation Steps (see Fig. 4.2)
subtract pedestal
pedestal
Hits
Flag Hits
noise
CMS Algorithm
set channels with hit to mean of ADC
set channels with hit to mean of ADC
update mean of ADC and mean of ADC^2
update mean of ADC and mean of ADC^2
raw noise
CMS noise

Figure 4.3: Processing ADCs for noise calculation.

4.2 CMS Algorithms

Three CMS algorithms were implemented for comparison. The first is the linear common mode subtraction algorithm as it is currently used on the Tell1 boards for the Silicon Tracker (ST algorithm). This is a simplified version of the algorithm used by the Vertex Locator (VELO). The full VELO algorithm was implemented as a second option. The VELO algorithm is described in [8]. As a third option, an algorithm originally proposed by Achim Vollhardt (two-point algorithm or 2P algorithm) was added. All three algorithms calculate a linear common mode for the 32 channels in a port.

4.2.1 ST Algorithm

The following list summarises the processing steps of the ST algorithm. The steps are explained in the text below. Figure 4.4 shows the ADC values for one port at different stages in this algorithm.

1. calculate mean of the 32 ADC values
2. subtract mean
3. detect hits ($>$13 ADC counts)
4. set hits and neighbours to zero
5. calculate new mean of the 32 ADC values
6. substract new mean
7. fit straight line to the 32 ADC values
8. substract line

First, the mean over all 32 ADC values in the port is calculated and subtracted. After this, hits are detected as ADC values above a given threshold. The implementation on the Tell1 board determines an individual threshold for each channel based on the raw noise of that channel. These threshold values are stored in a database and loaded into the memory of the Tell1 board. Thresholds usually lie between 12 and 14 ADC counts. For this study the same threshold of 13 ADC counts was used for all channels. Using the hits marked after the pedestal subtraction[3] was tested as well. As the simulated hit amplitude of 30 ADC counts per channel provides a clear signal, the impact of this change was found to be negligible.

ADC values for channels with hits and the neighbouring channels are set to zero. Then, the mean of the 32 channels is calculated and subtracted again. A simplified least-squares fit is then used to determine and subtract the linear common mode. The details of this fit are described in the documentation for the VELO algorithm [8].

The design of the ST algorithm was strongly influenced by the limited resources on the FPGAs used on the Tell1 boards. As the FPGAs do not allow for floating point arithmetic, only integers can be used and divisions, except by powers of two, are costly. Therefore, the least-squares fit had to be simplified for implementation on the Tell1 board. A question to be answered by this study was therefore how large the difference is between noise calculated with integer values and noise calculated with double-precision values.[4] To answer this question the ST algorithm was implemented with

[3]see Section 4.1.3

[4]see Section 4.3.2 for results.

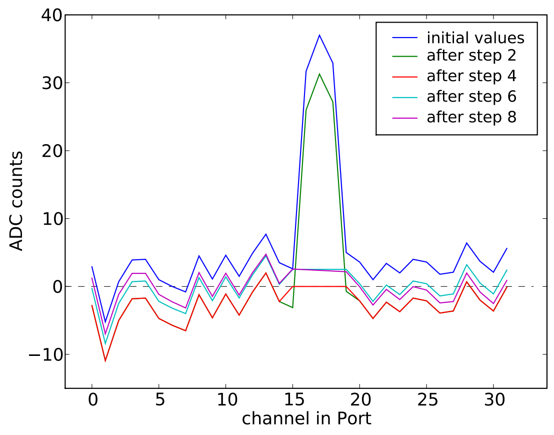


Figure 4.4: ADC values at different processing stages of the ST algorithm.

both data types. The integer version is implemented using the simplified least-squares fit, while the double version uses an exact least-squares fit.

4.2.2 VELO Algorithm

The algorithm implemented by the VELO is described in [8]. The basic concepts are the same for the VELO and the ST algorithm. Since the VELO reads out a smaller number of detector channels per Tell1 board, a more complex algorithm could be adopted. The following list summarises the processing steps of the VELO algorithm.

1. calculate mean of the 32 ADC values
2. substract mean
3. fit first straight line to the 32 ADC values
4. substract first line
5. detect hits ($>$ 3 times rms of the 32 ADC values)
6. set hits to zero (leave neighbours)
7. calculate new mean of the 32 ADC values
8. substract new mean
9. fit second line to the 32 ADC values
10. substract second line

The VELO fits a straight line before and after hit detection instead of only subtracting the mean before hit detection. After subtracting the first line, the ADC distribution on a port should be flat. The distribution of the 32 ADC values in a port can be used to detect hits. The rms of the 32 ADC values is calculated and any channel with an ADC count higher then 3 times this rms is considered a hit.

4.2.3 Two-Point Algorithm

The least-squares fit used in the ST algorithm and the VELO algorithm, gives a larger weight to strips at the border of the fit range [13]. Therefore the fitted straight line will tend to be closer to the ADC values in these channels. After subtraction of the linear common mode, ADC values in these channels will fluctuate less, i.e. these channels will have smaller CMS noise then channels in the middle of the fit range. In order to avoid this artefact, an alternative approach was suggested. The following list summarises the processing steps of the two-point algorithm (2P algorithm).

1. detect hits (>3 times raw noise of the channel)
2. split channels in two sets (channels 1-16, 17-32)
3. calculate mean ADC counts and mean strip number for each set
4. calculate slope and offset of the line through the two points
5. substract line

The data is split into two groups of 16 channels each (channels 1-16 and channels 17-32). In each group the mean over the ADC counts and over the strip numbers is calculated. These means define two points to which a line is fitted.

In the implementation of this algorithm the hit information available from the common mode data[5] was used to remove hits. As explained in Section 4.2.1, however, the impact of using either the marked hits or a fixed threshold is negligible.

4.3 Studies Without Hits

The performance of the different algorithms is presented in the following sections. This section presents the results of studies on measured data without hits. In a second phase simulated hits were added to the data. The results of the studies with hits are presented in Section 4.4.

4.3.1 Comparing two Algorithms

All the following histograms compare the CMS algorithms pairwise. Unless otherwise stated the ST algorithm with double-precision, no PCN-pedestals and no hits is used as the reference configuration.

For both algorithms under investigation the CMS noise is calculated. Figure 4.5 shows, as an example, the raw noise and the CMS noise as a function of channel number. The noise is calculated using the reference algorithm as explained above and the ST algorithm with integer-precision. Next, for each channel the difference of the two resulting CMS noise values is calculated.

[5]see Section 4.1.3

A histogram is then created with these differences. The histogram resulting for the configuration above is shown in Figure 4.6.

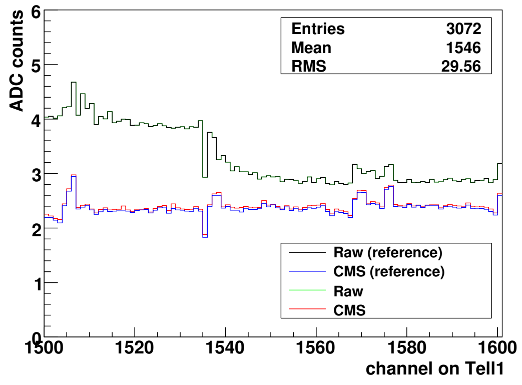


Figure 4.5: Noise calculated using the integer and double (reference) implementation of the ST Algorithm.

4.3.2 Studying Different Settings for the ST Algorithm

Comparing Integer and Double Implementation

The difference between the CMS noise calculated using double-precision and integer-precision is shown in Figure 4.6. On average, the difference is 0.03 ADC counts. This agrees with the expected value of 0.02 ADC counts. The

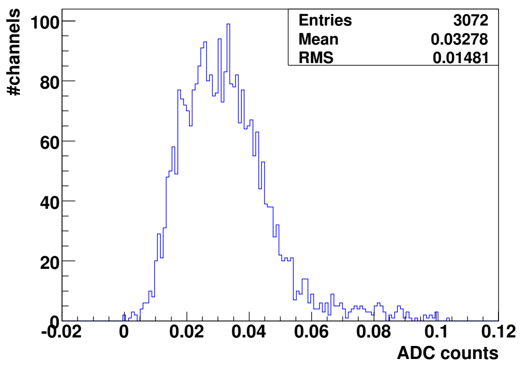


Figure 4.6: Distribution of $CMS_{int} - CMS_{double}$.

expected value can be determined from the fact that the error between the double and the integer value is expected to be equally distributed between 0 and 1. This gives a root mean square of

$$\sigma = \frac{1}{\sqrt{12}} \approx 0.29$$

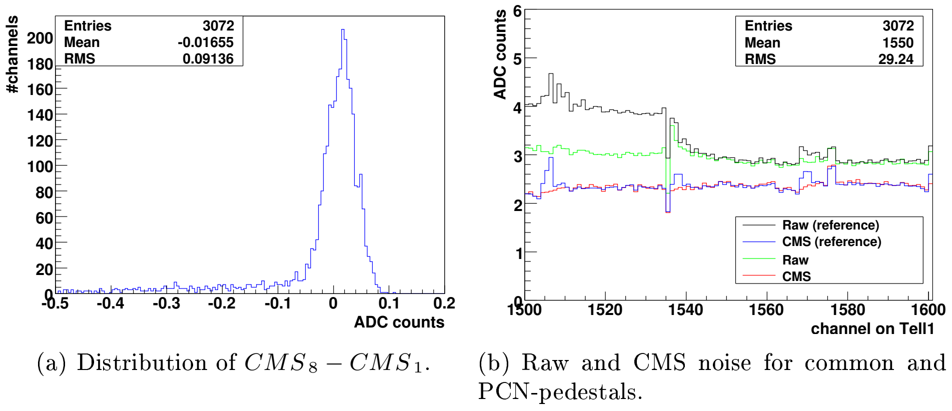
This error then needs to be combined with the common mode subtracted noise of approximately 2.3 ADC counts (see Figure 4.6). Using basic error propagation, this gives

$$\sqrt{2.3^2 + 0.29^2} - 2.3 = 0.02$$

Effect of Using PCN-dependent Pedestals

The effect on the CMS noise using the PCN-pedestals was studied. As Victor Hangartner shows in his master thesis [7] this removes the effects of the header crosstalk and the PCN-parity dependent pedestals in the raw noise[6].

Figure 4.7(a) shows the distribution of the difference between the CMS noise calculated with common pedestal and with PCN-pedestals. The main peak of the distribution is centred around 0.02 ADC counts, showing that using PCN-pedestals slightly increases the CMS noise for most channels. However, the tail below zero shows that this modification significantly improves the noise of some channels. As expected these are the channels affected by header crosstalk (e.g. channels 1504 - 1506 in Figure 4.7(b)).



(a) Distribution of $CMS_8 - CMS_1$.

(b) Raw and CMS noise for common and PCN-pedestals.

Figure 4.7: Using PCN-pedestals.

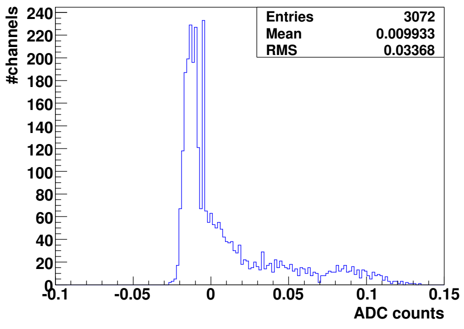
4.3.3 Comparison between Different Algorithms

After studying the different effects on the ST algorithm, the effects on the other two algorithms were studied and the results were compared. Figure 4.8 shows the resulting distributions for the 2P algorithm and the VELO algorithm for both common pedestal and PCN-pedestals.

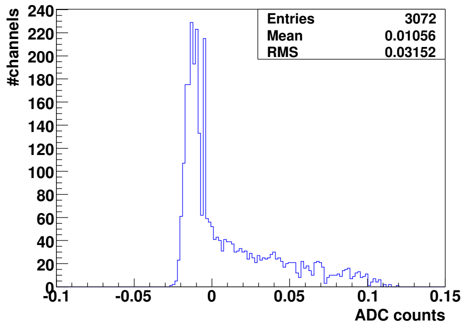
VELO Algorithm

The noise calculated by the VELO algorithm is slightly lower than the one from the ST algorithm, as shown in Figure 4.8(c). This is mainly due to the 3σ cut in the hit detection this algorithm uses (see Section 4.2.2). When the same fixed threshold hit detection is used for the VELO algorithm and the ST algorithm, the results are the same. As there are no hits in the data

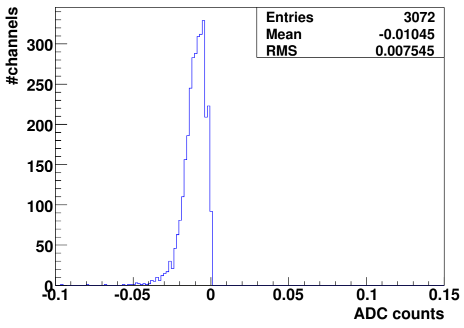
[6]see Section 1.1.2 for an explanation of these effects.



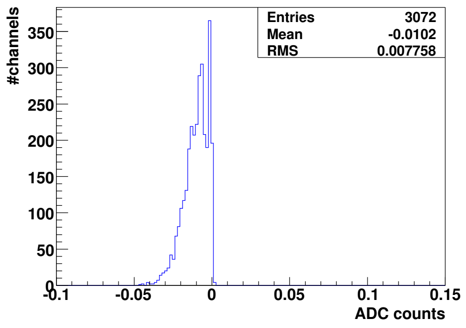
(a) 2P algorithm, common pedestal



(b) 2P algorithm, PCN-pedestal



(c) VELO algorithm, common pedestal



(d) VELO algorithm, PCN-pedestal

Figure 4.8: Comparing VELO algorithm and 2P algorithm to the ST algorithm. Figures 4.8(a) & 4.8(c) compared to the ST algorithm using the common pedestal, 4.8(b) & 4.8(d) compared to the ST algorithm using the PCN-pedestal.

analysed, the VELO hit detection seems to cut into the noise distribution. However, this still needs to be studied in detail.

2P Algorithm

The 2P algorithm shows a smaller noise for most channels and a tail to positive values. Figure 4.9 shows that the channels with increased noise are primarily the channels close to the edge of each port. As shown in Section 4.3.2, using PCN-pedestals removes the header crosstalk effects. For the first and the last channels of each port, the ST algorithm shows a decrease of the noise. This is due to the already mentioned artefact from the least-squares fit which gives a higher weight to the first and last channels (see Section 4.2.3). The flatter noise resulting from the 2P algorithm can therefore be considered an improvement.

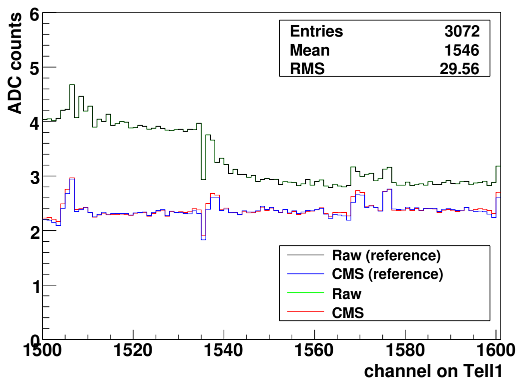


Figure 4.9: Raw noise and CMS noise for the ST algorithm (reference) and the 2P algorithm.

As shown in this study, comparing the algorithms using PCN-pedestals gives very similar results as comparing them using a common pedestal.[7] Further studies, therefore, use only either the common or the PCN-pedestal setting.

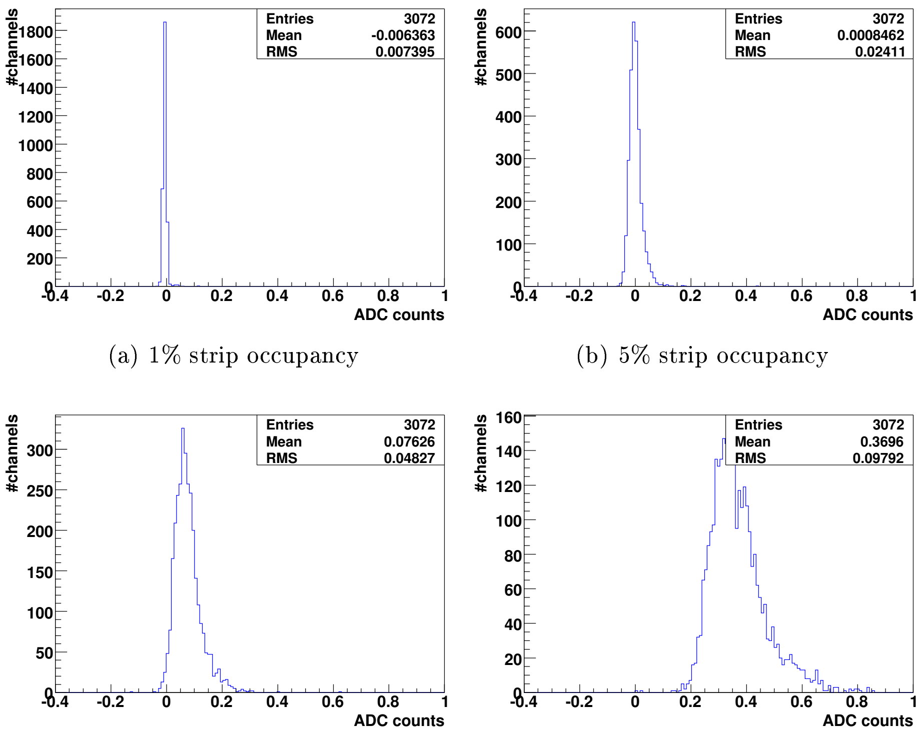
4.4 Studies With Hits

An important property of the common mode subtraction algorithm is its performance in the presence of hits. This was studied for all three algorithms for strip occupancies up to 20%. For operation at nominal LHCb luminosity strip occupancies up to 2-3% are expected [12].

4.4.1 Occupancy Study for the ST Algorithm

Figure 4.10 shows the development of the CMS noise for increasing strip occupancies. The mean of the noise distribution is stable for strip occupancies

[7]e.g. 4.8(a) is similar to 4.8(b) and 4.8(c) is similar to 4.8(d).



(a) 1% strip occupancy

(b) 5% strip occupancy

(c) 10% strip occupancy

(d) 20% strip occupancy

Figure 4.10: Difference of the CMS noise calculated with the ST algorithm without hits (reference) and with the ST algorithm for different strip occupancies.

up to 5%. The width of the distribution however starts to increase. For higher occupancies, the width of the noise distribution continues to increase while at the same time the mean shifts towards higher values. Figure 4.10(d) shows that the average CMS noise has increased by approximately 0.4 ADC counts for a strip occupancy of 20%.

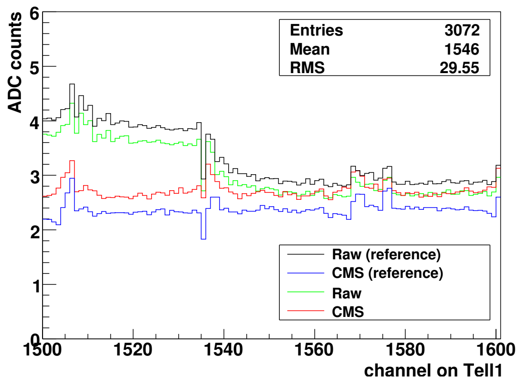


Figure 4.11: Noise from the ST algorithm with 20% strip occupancy.

Decreasing Raw Noise

Figure 4.11 displays the raw noise and the CMS noise as a function of strip number for 20% strip occupancy. This figure shows that the raw noise decreased at the same time as the CMS noise increased. This is an artefact of the way the STNZSMonitor handles hits. As explained in Section 4.1.3, channels that contain hits are set to the moving average of the ADC values. For this reason they do not influence the moving average. They do however reduce the fluctuations around the average and thereby decrease the calculated noise value.

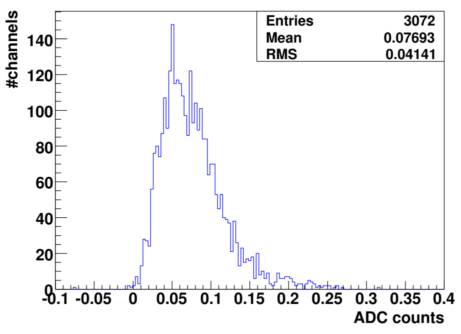
Increasing CMS Noise

Independent of the CMS algorithm used, the effect described above would be expected for the CMS noise as well. The real increase of the CMS noise due to the increasing occupancy must therefore be larger than shown in Figure 4.10. The increase of the CMS noise for the ST algorithm results from setting hits to zero. This effect is studied in detail in Section 4.4.4.

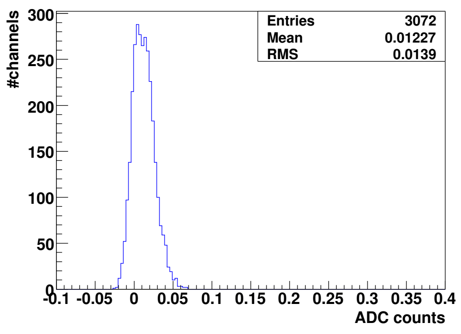
4.4.2 Occupancy Study for the VELO Algorithm

As shown in Section 4.3.3, the hit removal seems to bias the CMS noise calculated by the VELO algorithm even when there are no hits. When hits are added to the data, the hit removal leads to a rapid increase of the CMS noise.

Figure 4.12(a) shows that the CMS noise increases by almost 0.08 ADC counts for 1% strip occupancy. This is the same noise increase as the ST algorithm shows for 10% strip occupancy. That this effect is entirely due to the hit detection algorithm is demonstrated in Figure 4.12(b). There, a fixed threshold is used for hit detection and the VELO algorithm gives results comparable to the ST algorithm.



(a) default hit detection



(b) fixed threshold hit detection

Figure 4.12: Different hit detection methods for the VELO algorithm. For both plots 1% strip occupancy was used.

The reason why the default hit detection method leads to high CMS noise values is illustrated in Figure 4.13. The figure shows the ADC values of all 32 channels in one port at different stages of the common mode processing. The port contains one cluster with strips 16, 17 and 18. As explained in

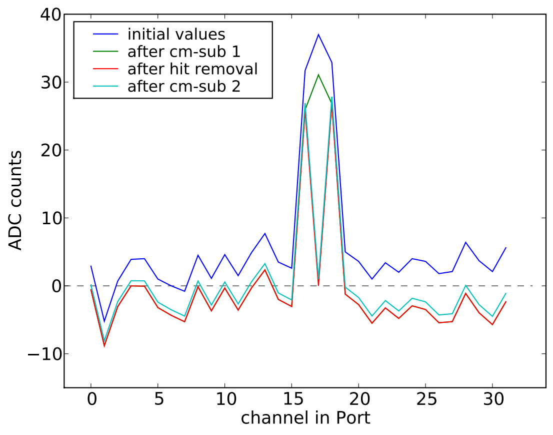
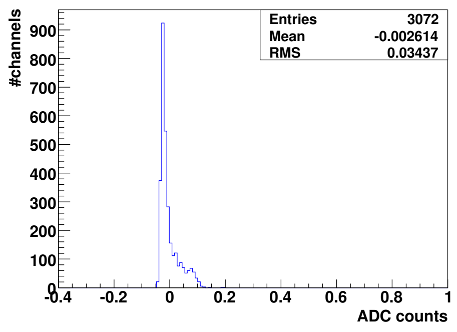


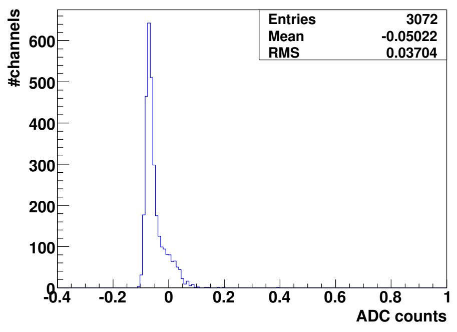
Figure 4.13: Common mode processing for one port.

Section 4.2.2 the hit threshold is determined from the rms of the 32 ADC values after the first linear subtraction. Detected hits are then set to zero. The rms is large if the data contains hits. In the example, the calculated hit threshold falls exactly between the ADC values of the outer channels and the ADC value of the central channel in the cluster. Only the central channel is set to zero and the other two escape detection. The second iteration of the linear fit then gives the wrong result, which leads to a high noise value for all channels.

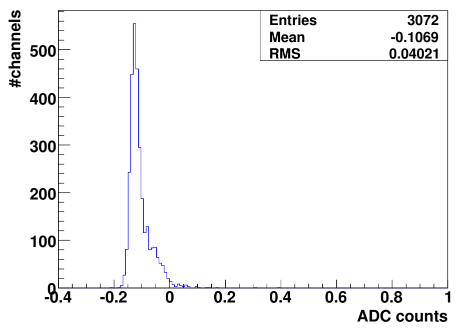
4.4.3 Occupancy Study for 2P Algorithm



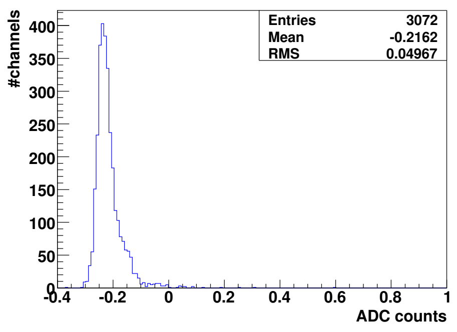
(a) 1% strip occupancy



(b) 5% strip occupancy



(c) 10% strip occupancy



(d) 20% strip occupancy

Figure 4.14: Difference of the CMS noise calculated by the 2P algorithm with the given strip occupancies and the noise for the ST algorithm without hits.

The 2P Algorithm shows the best behaviour even for high occupancies. Figure 4.14 shows the development of the CMS noise for strip occupancies of 1%, 5%, 10% and 20%. In contrast to the corresponding plots for the ST algorithm (Figure 4.10), the CMS noise decreases with increasing occupancy. This is the expected effect of setting hits to the moving average before the noise calculation (see Section 4.4.1).

4.4.4 Comparing ADC Values

The reason why the 2P algorithm gives better results than the ST algorithm is illustrated in 4.16. The figure shows the output of one Beetle chip for the ST algorithm and the 2P algorithm. Shown are the common mode

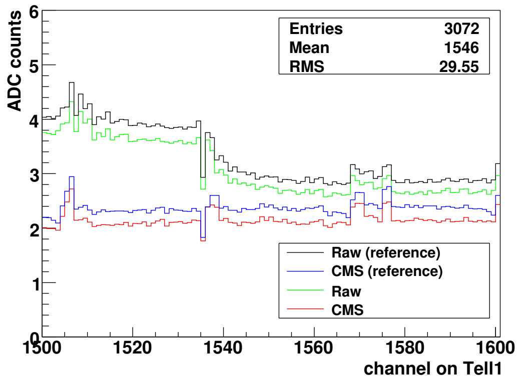
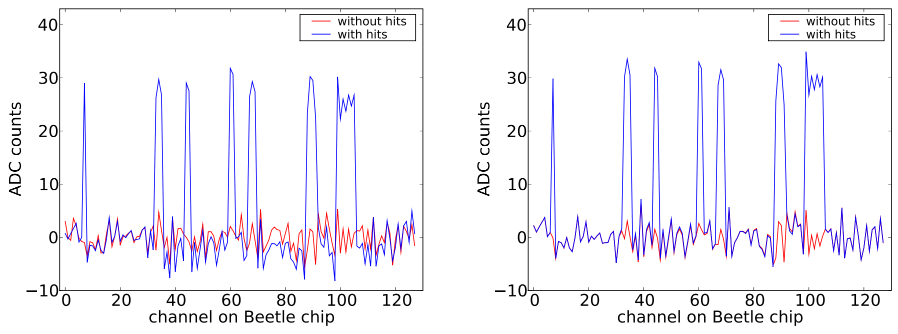


Figure 4.15: Raw noise and CMS noise for the 2P algorithm with 20% strip occupancy. As reference the ST algorithm is shown.

subtracted ADC values of both algorithms when for data without hits and for the same data with hits added. It is evident that in the presence of hits the 2P algorithm returns the original values almost perfectly while the ST algorithm sometimes shows large differences between results with and without hits. The ST algorithm therefore gives a too high noise.



(a) ST algorithm (b) 2P algorithm

Figure 4.16: Comparing common mode subtracted ADC values with and without MC hits.

The reason for this difference lies in the way hits are handled by the two algorithms. The ST algorithm sets detected hits and their neighbours to zero before calculating and subtracting the mean for the second time. This potentially distorts the common mode. Figure 4.17 and Table 4.2 illustrate the problem. If the hit in the original data is set to zero the calculated common mode correction will be too low.

Setting hits to zero is necessary for the ST algorithm. Masking them correctly would break the central assumptions on which the simplified straight line fit[8] is based, namely that 32 channels are processed. The 2P algorithm as implemented here makes no such assumption. Hits are masked correctly, i.e. they are not taken into account in the calculation of the averages from

[8]Step 7 in Section 4.2.1.

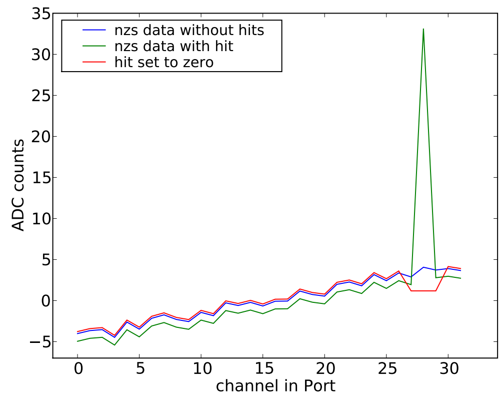


Figure 4.17: ADC values of the ST algorithm before calculating slope.

data used	calculated slope of fitted line
without hit	0.28
with hit	0.41
hit set to zero	0.24

Table 4.2: Setting hits to zero distorts the common mode.

which the straight line parameters are determined (see Section 4.2.3 For the calculation of the correct averages, divisions with integer values between 1 and 16 are needed. If the 2P algorithm can be implemented on the Tell1 board, therefore, depends on the possibility to carry out these divisions on the FPGAs. Studying such an implementation in detail was not possible in the timescale of this thesis.

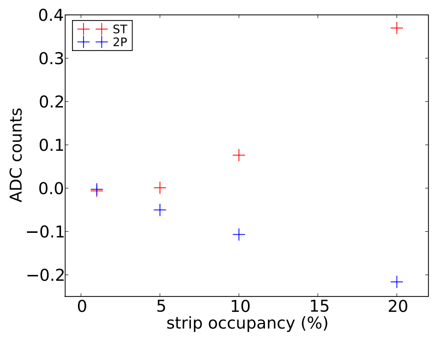
4.5 Summary and Outlook

Table 4.3 summarises the results for the three common mode algorithms compared in this thesis.

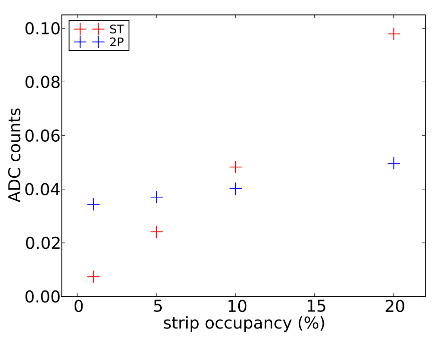
The study of these different common mode algorithms gave a number of important insights. First, the study demonstrated that the current STNZS-Monitor calculates a biased noise if the strip occupancy is high. To fix this, fundamental changes in the algorithm are needed. Whether this is necessary or not will be discussed and decided by the LHCb Silicon Tracker group.

A second result is that using the PCN-dependant pedestals removes the header crosstalk as expected but leads to a small increase in average CMS noise and a large increase in memory consumption.

Finally, Figure 4.18 clearly shows that masking the hits correctly makes an algorithm a lot more robust at high occupancies. As the 2P algorithms is simpler than the ST algorithm, it might be possible to implement the 2P



(a) Mean



(b) RMS

Figure 4.18: Mean and RMS for the noise distributions.

algorithm on the Tell1 board with correct hit-masking. The main obstacle for the implementation in the FPGA is the division needed to calculate the mean values. An implementation of the 2P algorithm on the Tell1 board therefore needs to be studied.

Algorithm	ST	VELO	2P
integers	+0.03	n/a	n/a
PCN-pedestals	+0.02 removes header crosstalk	similar to ST algorithm[a]	similar to ST algorithm[a]
CMS noise without hits	reference	−0.01	−0.01 increase for channels with header crosstalk
CMS noise 1% strip occupancy	< 0.01	+0.08 stable if ST hit detection is used	< 0.01
CMS noise 10% strip occupancy	+0.08[b]	n/a	−0.1[b]

[a] see Section 4.3.3
[b] see Section 4.4.4

Table 4.3: Comparison between algorithms. All numbers given are mean values of the differences in ADC counts when compared to the ST algorithm with double-precision, no PCN-pedestals and no hits.

Bibliography

[1] O. Callot et al, “Online Data Monitoring in the LHCb Experiment”, Proc. of CHEP 2007.

[2] G. Barrand et al, “GAUDI – A Software Architecture and Framework for building HEP Data Processing Applications”, Proc. of CHEP 2000

[3] G. Granziani, “Histogram DB for Online Monitoring – User’s Manual”, `/afs/cern.ch/user/g/ggrazian/www/lhcb/histdbdoc.pdf`.

[4] G. Haefeli, A. Gong, “ST zero suppressed bank data format”, EDMS 690583.

[5] G. Haefeli, A. Gong, “VELO and ST non-zero suppressed bank data format”, EDMS 692431.

[6] G. Haefeli, A. Gong, “VELO and ST error bank data format”, EDMS-note 694818.

[7] V. Hangartner, “Noise Investigations for the Tracker Turicensis of the LHCb Experiment”, Master Thesis.

[8] P. Koppenburg, “Contribution to the Development of the LHCb vertex locator and study of rare semileptonic decays”, CERN Thesis 2002-010.

[9] The LHCb Collaboration, “The LHCb Detector at the LHC”, J. Instrum. 3 (2008) S08005.

[10] S. Löchner, M. Schmelling, “The Beetle Reference Manual”, LHCb note 2005-105.

[11] M. Needham, “Silicon Tracker zero-suppressed data model and decoding”, LHCb note 2008-047.

[12] M. Needham, “Silicon Tracker Occupancies and Clustering”, LHCb note 2007-024.

[13] O. Steinkamp, private communication.

[14] J. van Tilburg, private communication.

[15] A. Vollhardt, “An Optical Readout System for the LHCb Silicon Tracker”, CERN Thesis 2005-025.

[16] `https://twiki.cern.ch/twiki/bin/view/LHCb/LHCbSoftwareTrainingBasics#Organisation_of_LHCb_software`

[17] `http://root.cern.ch`

[18] `http://lhcb-release-area.web.cern.ch/LHCb-release-area/DOC/lbcom/`

[19] `https://lbtwiki.cern.ch/bin/view/Online/HistogramPresenter`

[20] `http://lhcb-release-area.web.cern.ch/LHCb-release-area/DOC/online/`

Appendix A

User Guide for the Online Monitoring

A.1 Adding an Algorithm to the Monitoring

A.1.1 Preparing the code

1. Set up the environment for the monitoring:

```
$ export CMTCONFIG=$CMTDEB
$ export User_release_area=/group/st/sw/cmtuser
$ SetupProject Online
```

2. Source the `setup.sh` file in the most recent release.
3. Compile if necessary (`cmt br make`).
4. Generate the `*.vars` files.

```
$ source /group/online/dataflow/scripts/shell_macros.sh
$ crsetup
```

5. Create an option file for the new algorithm
   (e.g. /group/st/sw/scripts/TTNZSMon.opts).

A.1.2 Adding Task and assigning job options file

1. Login in to ttecs01 (or any other machine with PVSS installed).
2. Run

```
$ cd /group/online/dataflow/scripts
$ ./pvssui -JOBOPTIONS
```

3. Click on *Show Task Types*.
4. Right click in list of known task types and select *new*.
5. Choose task name and click *Create*.

6. Give a name to the task (e.g. TTNZSMon).
7. Double click on the newly created task (e.g. TTNZSMon).
8. Specify the script and tick the *Require Defaults* box.
9. Double click on the task.
10. Specify options the file
    (e.g. `#include "/group/st/sw/scripts/TTNZSMon.opts"`).

A.1.3 Running Monitoring Task in Online Monitoring

1. Run

   ```
   $ ./pvssui -stream
   ```

2. Select *TTECS: TT_RunInfo*.
3. Select *Monitoring*.
4. Add a line for the new task
   (e.g. TTNZSMon/1/Stream1).

A.2 Step by Step Guide to HistDBPython

1. Install the HistDBPython package
   - Make sure HistDBPython is part of the project
   - Prepare a working directory
   - Add example files
   - Create a password file (default `password`)
2. Configure the algorithms
3. Run `gaudirun.py [-nv] FILE`

A.2.1 Installation

1. To use the HistDBPython package add the following line to the requirements file of your project:

   ```
       use HistDBPython      *     Online
   ```

2. Create a directory where you plan to store your Python option files. This directory can get crowded so it might be a good idea to create a subdirectory for each of the four algorithms provided by HistDBPython.

3. When this directory is ready copy the example files from the documentation directory of the HistDBPython package[1] into your working directory.

[1]http://isscvs.cern.ch/cgi-bin/viewcvs-all.cgi/Online/HistDBPython/doc/?root=lhcb

4. Create a password file in the root of your working directory. The file should contain only the password for HIST_WRITER. The password can be obtained from Giacomo Graziani. Make sure there is no empty line at the end of the password file. This will lead to an error.

A.2.2 Configuration

The four algorithms are controlled by different option files. For each algorithm an extensively commented example file exists that document the different options available for each algorithm. All example files can be found in Section A.2.4.

A.2.3 Running

After configuring an algorithm it can be run by calling gaudirun.py with the corresponding option file. If you want to run the option file `page.py` you would therefore call:

```
$ gaudirun.py page.py
```

Hint: Use `gaudirun.py -vn ...` to check the configuration file. This command just parses the option file without executing any algorithm.

If you want to run several option files you can use a shell script like runOptionFiles.sh in Section A.2.4. This file is part of the HistDBPython package as well.

A.2.4 Example Files

runOptionFiles.sh

```
#!/bin/bash
##############################################
#                                            #
#    Script to run multiple option files     #
#       Nicola Chiapolini, 03.06.2009        #
#                                            #
##############################################

# print help text if necessary
if [[ $# -eq $startPageList ]]
then
  echo "  Usage: ./runOptionFiles.sh file1.py file2.py ..."
  exit 0
fi

# run the gaudi app for each page file specified on the commandline
for file in $@
do
        if [ -f $file ]
        then
                gaudirun.py $file
        else
                echo "$file does not exist in search path!"
        fi
done
```

example_add.py

```
from Gaudi.Configuration import *
from Configurables import AddNewHistToDB
addHistAlg = AddNewHistToDB("addHistAlg")

# set task and algorithm name of the histograms
#   if needed task and algorithm will be added to the DB as well
addHistAlg.TaskName      = "TTSumMon"
addHistAlg.AlgorithmName = "STErrorMonitor"

# ***********************************************
# defining helper variables to generate the histogram names
histName  = "error-types_$tell"
histNum   = range(1, 49)

# ***********************************************
# setting the type for all histograms
#   only histograms of the same type can be generated at the same
#   time
# possible values are (OnlineHistDB v5r0):
#   H1D, H2D, P1D, P2D, CNT
addHistAlg.HistoType  = "H2D"

# filling HistoNames
#   HistoNames is a list with the names of all histograms that
#   should be created
addHistAlg.HistoNames = []
for num in histNum:
        addHistAlg.HistoNames.append(histName + str(num))

# ***********************************************
# setting the Output level
addHistAlg.OutputLevel = INFO

# specifying the password file
#   this sting points to the file containing the password for
#   HIST_WRITER the file can be given either relative to the run
#   directory or with an absolut path.
#   the default value is 'password'
#addHistAlg.PasswordFile = "password"

# Add the algorithm to the application manager from Gaudi
app = ApplicationMgr()
app.TopAlg.append(addHistAlg)
```

example_page.py

```
from Gaudi.Configuration import *
from Configurables import CreateHistDBPages
createHistAlg = CreateHistDBPages("createHistAlg")

# ***********************************************
# defining helper variables to create the page
# and histogram names

# page info
folderBase = "Expert/perTell/tell"
page    = "Noise"

# histogram info
histoSet1  = "Raw/noise_$tell"
histoSet2  = "CMS/cms_$tell"
histoNum   = range(1, 49)


# ***********************************************
# setting the options for the algorithm

# defining the pages that should be created
#   PageNames is a list of strings that contains one entry for each
#   page to be created.
#   The page names will be prefixed with PageBase
createHistAlg.PageNames    = []

# define the documentation string for each page
#   PageDoc is a list of strings with the same ordering as PageNames
#   depending on the size of the list, the follwing actions will be
#   taken:
#   - if the list is empty, no documentation will be added.
#   - if the list has one entry for each page, these will
#     be used.
#   - otherwise the first documentation will be used for
#     every page.
createHistAlg.PageDoc    = []
```

continued on next page

example_page.py - continued

```
# define the histograms that should be shown on each page
#   HistoNames is a list of lists. Each inner list is a list of
#   strings containing the histogram names for one page.
# the structure is [pageNumber][histogram]
#   e.g.: HistoNames => {
#            page1 => { histo1A, histo1B }
#            page2 => { histo2A, histo2B }
#         }
# Conditions:
# - each page vector needs to have the same number of entries as the
#   layout definition.
# - the page vectors need to be sorted according to PageNames above.
createHistAlg.HistoNames = []

# define the pages histograms should be linked with
#   HistoLinks is a dictionary with structure { string : string }
#   *) the first string is the name of a histogram in HistoNames.
#   *) the second string is the full path of the page that should be
#      displayed when one clicks on the histogram.
#   The page to display is a property of the histogram, so only one
#   page can be defined per histogram.
createHistAlg.HistoLinks = {}

# loop filling the variables defined above
for num in histoNum:
        id = str(num)
        createHistAlg.PageNames.append("/"+folderBase+id+"/"+page)

        folderHistos = []
        folderHistos.append(histoSet1+id)
        folderHistos.append(histoSet2+id)
        createHistAlg.HistoNames.append(folderHistos)

        # this is useless and just for demonstartion
        createHistAlg.HistoLinks[histoSet1+id] =
        createHistAlg.HistoLinkshistoSet1+id =
          "/TT/Expert/NoiseAllTell1s/"

# ************************************************
# importing common configuration files

# layout definition
importOptions("example_layout.py")

# general options
importOptions("example_genCreate.py")
```

The last two lines of example_page.py include additional option files. These could also be added directly to the file but putting them into a separate file allows to reuse them for multiple pages.

example_layout.py

```python
from Gaudi.Configuration import *
from Configurables import CreateHistDBPages
createHistAlg = CreateHistDBPages("createHistAlg")

# define the layout for this page type
#   this is a list containing one entry for each histogram. each
#   entry is itself a list with four double values giving the
#   position of the histogram
# the structure is:
#   PageLayout => {
#       histo1 => { Xmin, Ymin, Xmax, Ymax }
#       histo2 => { Xmin, Ymin, Xmax, Ymax }
#   }
# if Xmin < 0 the histogram will be overlayed on the previous
# histogram.
createHistAlg.PageLayout = []
createHistAlg.PageLayout.append([0.01, 0.01, 0.99, 0.99])
createHistAlg.PageLayout.append([-1.0, 0, 0, 0])
```

example_genCreate.py

```
from Gaudi.Configuration import *
from Configurables import CreateHistDBPages
createHistAlg = CreateHistDBPages("createHistAlg")

# specifying the password file
#   this sting points to the file containing the password for
#   HIST_WRITER the file can be given either relative to the run
#   directory or with an absolut path.
#   the default value is 'password'
#createHistAlg.PasswordFile = "password"

# define a fixed part for all page names
#   This will be prepended to all page names and should be used to
#   define the correct main folder for the project
createHistAlg.PageBase = "/TestDir"

# define a fixed part for all histogram names
#   This will be prepended to all histogram names and can be used to
#   specify task and algorithm name
createHistAlg.HistoBase = "TestMon"

# set the output level for the message service
#   use one of:
#     DEBUG (2), INFO (3), WARNING (4), ERROR (5), FATAL (6)
#   additionally there are:
#     NIL (0), VERBOSE (1), ALWAYS (7), NUM_LEVELS (8)
createHistAlg.OutputLevel = INFO

# Add the algorithm to the application manager from Gaudi
app = ApplicationMgr()
app.TopAlg.append(createHistAlg)
```

example_dopts.py

```
from Gaudi.Configuration import *
from Configurables import SetHistDisplayOptions
setDisplay = SetHistDisplayOptions("setDisplay")

# Setting up the dictionaries for the display options.
#   There is one dictionary for each data type with the structure
#   { string : <datatype> }
#   (floatOptions is internally stored in a <string, double>-map
#    but the values are cast to float before setting)
setDisplay.intOptions = {}
setDisplay.floatOptions  = {}
setDisplay.stringOptions = {}


# ***********************************************
# defining the histograms to edit
#   HistoNames is a dictionary with strucutre { string : boolean }
#   *) the string is the identifier of the histogram, the string
#      will get prefixed with the content of HistoBase
#   *) if int is true, saveHistoSetDisplayOptions() will be called
#      i.e. the changes will be applied for the whole set
setDisplay.HistoNames = {
  "/error-types_$tell1" : True
}

# Should all display options for these Histograms be unset
# before the changes below are made?
setDisplay.UnsetOptions = True

# Filling the display option dicitonaries with entries
setDisplay.stringOptions["LABEL_X"]   = "Optical Link"
setDisplay.intOptions["NDIVX"]        = 424

setDisplay.floatOptions["LAB_X_SIZE"] = 0.04
setDisplay.floatOptions["LAB_Y_SIZE"] = 0.04

setDisplay.stringOptions["DRAWOPTS"]  = "box"
```

continued on next page

example_dopts.py - continued

```
# ************************************************
# Define custom bin labels
#   for each axis there is a list with one
#   string for each bin
#setDisplay.xLabels = [];
setDisplay.yLabels = []
setDisplay.yLabels.append("None");
setDisplay.yLabels.append("CorruptedBank");
setDisplay.yLabels.append("OptLinkDisabled");
setDisplay.yLabels.append("TlkLinkLoss");
setDisplay.yLabels.append("OptLinkNoClock");
setDisplay.yLabels.append("SyncRAMFull");
setDisplay.yLabels.append("SyncEvtSize");
setDisplay.yLabels.append("OptLinkNoEvent");
setDisplay.yLabels.append("PseudoHeader");
setDisplay.yLabels.append("WrongPCN");


# ************************************************
# application manager and messag service settings
importOptions("example_genSet.py")
```

As with example_page.py an additional option file is included in the file above. The code below could be included directly in the file as well, but as the same algorithm usually publishes different histograms these settings can be reused if put into a separate file.

example_genSet.py

```python
from Gaudi.Configuration import *
from Configurables import SetHistDisplayOptions
setDisplay = SetHistDisplayOptions("setDisplay")

# specifying the password file
#   this sting points to the file containing the password for
#   HIST_WRITER the file can be given either relative to the run
#   directory or with an absolut path.
#   the default value is 'password'
#setDisplay.PasswordFile = "password"

# define a fixed part for all histogram names
#   (e.g. task and algorithm name)
setDisplay.HistoBase = "TestMon"

# set the output level for the message service
#   use one of:
#     DEBUG (2), INFO (3), WARNING (4), ERROR (5), FATAL (6)
#   additionally there are:
#     NIL (0), VERBOSE (1), ALWAYS (7), NUM_LEVELS (8)
setDisplay.OutputLevel = DEBUG

# Add the algorithm to the application manager from Gaudi
app = ApplicationMgr()
app.TopAlg.append(setDisplay)
```

example_remove.py

```
from Gaudi.Configuration import *
from Configurables import RemovePages
rmAlg = RemovePages("rmAlg")

# defining the root of the branches to delete
#   StartFolders is a list of stings
#   !! All pages and subfolders will be delete
rmAlg.StartFolders = [ "/TestFolder/subfolder1",
                       "/TestFolder/subfolder2" ]

# simulate deletion?
#   If DryRun = True all pages affected by this configuration will
#   be printed but no changes will be written to the OnlineHistDB.
# This should be used to check which pages really get deleted.
rmAlg.DryRun       = True

# specifying the password file
#   this sting points to the file containing the password for
#   HIST_WRITER the file can be given either relative to the run
#   directory or with an absolut path.
#   the default value is 'password'
#rmAlg.PasswordFile = "password"

# Protect top level directories from deletion
# ----------------------------------------
# !! do not change unless you know what !!
# !! you are doing!                     !!
# ----------------------------------------
#   if ProtectTopLevel is True, top level directories (e.g. "/TT"
#   or "/VELO") in StartFolders will get ignored
#rmAlg.ProtectTopLevel = True

# set output level
rmAlg.OutputLevel  = INFO

app = ApplicationMgr()
app.TopAlg.append(rmAlg)
```