University of Glasgow | Experimental Particle Physics

GLAS-PPE/2008-40
24th November 2008

Department of Physics and Astronomy
Experimental Particle Physics Group
Kelvin Building, University of Glasgow,
Glasgow, G12 8QQ, Scotland
Telephone: +44 (0)141 330 2000 Fax: +44 (0)141 330 5881

Description of the Vetra Project

T. Szumlak[1], C. Parkes[1]

[1] University of Glasgow, Glasgow, G12 8QQ, Scotland

Abstract

Vetra is the LHCb data reconstruction project which emulates the performance of the TELL1 readout board processing algorithms. This project is required for monitoring and commissioning the LHCb silicon detectors. A bit-perfect emulation of the TELL1 processing algorithms is performed. This project allows raw data (non-zero suppressed) to be processed to produce the standard zero suppressed cluster data, used by the LHCb reconstruction project Brunel.

The Vetra framework is used by the VELO and ST detectors in LHCb. This note provides a general description of the Vetra framework but concentrates on the VELO usage. Vetra is used to monitor the performance of the detector and the data acquisition board algorithms. The parameters that control the data acquisition boards are determined and optimised using Vetra. The project is used widely in the VELO and is used for testbeam and laboratory studies, including production testing for the modules.

Vetra is named in honour of the famous CERN physicist and creator (sic) of anti-matter Vittoria Vetra.

Contents

List of Figures

1 Introduction

The *Vetra* project was developed to facilitate the development and commissioning of the TELL1 [4] board processing algorithms. In time Vetra became a much more versatile tool and an essential part of the VELO software; it has been used for laboratory testing during construction, test-beam operation and detector commissioning.

The *Vetra* project will be used in the experiment during data taking as a monitoring tool and for parameter tuning. It will be used to calculate the values of parameters of the TELL1 processing algorithms such as cross talk coefficients or clusterization seeding and inclusion thresholds [2].

The details of the processing algorithms and analyses are not described here, this note describes the structure and capabilities of the *Ve* software framework.

A general description of *Vetra* is provided in section 2. This is followed by two chapters describing the most important components of *Vetra* - Non-Zero Suppressed (NZS) data handling (reading and decoding are presented in section 3) and the TELL1 software emulator (section 4). In section 5 the C++ implementation details are given and the base-line emulation described. A summary and conclusions are presented in 6.

This note focuses on the description of the *Vetra* from the VELO group's point of view because presently the full chain of the NZS data processing exists for this detector only and the application has been heavily used to process real data taken during the VELO test beam campaign. However, all the functionality that is currently provided by *Vetra* can, and is, being reused and extended to accommodate the ST silicon detector needs.

The functionality of *Vetra* is complimentary to that of the existing LHCb projects, its purposes are:

- Decoding of the Non-Zero Suppressed banks (including pedestal bank and error bank)
- TELL1 electronic board processing (zero suppression) emulation
- High level detector monitoring requiring Non-Zero Suppressed data (used in test beams, commissioning and standard running of the experiment)
- Processing parameter determination (used in commissioning and standard running of the experiment)

The distinct difference between *Vetra* and the other presently available projects is that the *Vetra* make use of NZS data streams (e.g. VeloFull bank or pedestal bank). The final output of *Vetra* - the emulated Zero Suppressed cluster bank - is identical to the bank that is produced by the data acquisition board. In this respect it is similar to the Boole application which also provides the cluster bank as its output but for simulated rather than real data. The emulated ZS bank can subsequently be used by the Brunel project to reconstruct tracks and vertices.

Naturally, it is also possible to run *Vetra* over the real ZS data and perform a detector specific tracking without the Brunel overhead. This functionality proved to be very useful during the VELO test beam when the alignment procedure, Velo tracking, and reconstruction software were all tested for the first time using the real data.

2 General Description

Vetra provides a mechanism to emulate each of the algorithms that would normally be executed in the TELL1 data processor board. A bit perfect emulation of the algorithms is provided and the output of the chain is the zero suppressed raw bank, that would normally have been produced by the TELL1 board. In addition a space is available to allow alternative algorithms to be developed and their performance compared with the base-line TELL1 algorithms.

For the VELO, the data is prepared by taking the VeloFull bank produced by the TELL1 and decoding this into VeloTELL1Data [3] objects. Each VeloTELL1Data object contains the digitized signals from all 2048 strips from one VELO silicon sensor (see section 3 for more details).

The emulation is broken up into a number of distinct phases (more information on this can be found in section 4) that directly correspond to the real processing performed by FPGA processors on the TELL1 boards. The output of each stage (up till the final clustering) is again stored in the VeloTELL1Data objects.

At each stage of the emulation process detailed monitoring of the output data can be performed.It is also possible to make a direct comparison of the output of the whole chain with the zero-suppressed data bank produced by the TELL1 for verification of the TELL1 processing, and to indicate problems with data processing on the TELL1 boards.

The remaining parts of this section provides a short description of the structure and functionality of each of the *Vetra* component packages. The structure is illustrated in Fig. 1. Only the packages directly related with the *Vetra* project are presented all other dependencies, for instance framework specific components, are omitted.

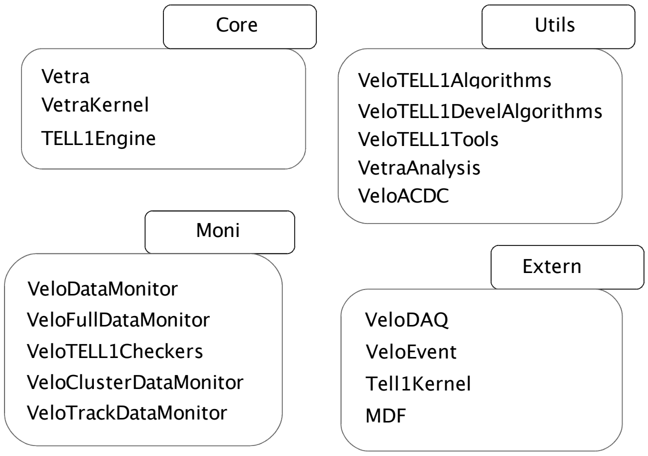

Figure 1: The component packages of the *Vetra* project. The essential external packages are also shown.

2.1 Core Packages

Tell1/Vetra

Short description: This is the core package of the application and contains the executable code and option files needed for the correct configuration of a job.

Tell1/VetraKernel

Short description: contains the base classes (e.g. `VeloTELL1Algorithm`), algorithms and plain classes used by other packages and tool interfaces. From the framework point of view the *Vetra*`Kernel` constitutes both the component and linker package.

Tell1/TELL1Engine

Short description: the most important part of the *Vetra* TELL1 emulation. This contains the so-called "engine classes" that perform the pre-processing of the specially prepared data. Each engine class implements part of the *tell1Lib* software library that is used to provide a bit perfect high level model of the `VHDL` machine code of the TELL1 firmware. Each engine represents one processing stage performed by the TELL1 boards.

The engine classes are decoupled from the `GAUDI` [6] framework in the sense they do not inherit from any of the standard base class. This eliminates any overhead introduced by the `GAUDI` framework and makes the classes simple and maintainable by the people supporting the *tell1Lib* software.

The *tell1Lib* software is written in C and can be compiled and run on the credit card PC of the TELL1 board. In addition, the interface of each engine class and the naming convention is identical to the one used in *tell1Lib* . This allows 'copy paste' functionality and facilitates rapid updates of the engine classes when new *tell1Lib* releases are provided.

2.2 Utility Packages

Velo/VeloTELL1Algorithms

Short description: These algorithms provide the interfaces to the TELL1 algorithm emulation performed by the engine classes of the `TELL1Engine` package. One of these interfaces or 'wrappers' is provided for each of the engine classes. They handle the preparation of the input data for the engines, run the processing, and store the output data. The data for each processing stage is retrieved and stored on the Transient Event Store.

Each wrapper inherits from `GaudiAlgorithm` and uses its engine counterpart as a plug-in.

Using this wrapper-plugin approach the `GAUDI` environment is bound with the stand alone *tell1Lib* software to make the emulation follow as close as possible the hardware and firmware structure.

Velo/VeloTELL1DevelAlgorithms

Short description: this package accommodates the development of alternative processing algorithms, which can then be compared with the performance of the base-line bit-perfect emulation software. This code is not based on the *tell1Lib* .

Velo/VeloTELL1Tools

Short description: contains implementation code of all tools used for NZS data analysis and monitoring. Each tool inherits from the appropriate interface in the `VetraKernel` package.

Velo/VetraAnalysis

Short description: this stores the analyses to calculate and optimise the parameters of the TELL1 board processing algorithms. All algorithms related with NZS data analysis, other than monitoring, are located here. At the time of writing it contains the cable cross-talk computer algorithm that calculates the correction coefficients for the FIR algorithm (see section 4) and a basic algorithm for analysis of the cluster bank (for both real and emulated ZS banks). In the future the package will be extended to incorporate further algorithms such as those to compute Beetle header cross talk correction coefficients, resolution parametrizations, and thresholds for clusterization process.

Velo/VeloACDC

Short description: this package is related with the VELO test beam Alignment Calibration Data Challenge (ACDC) campaign and contains a full description of the detector geometry used in all stages of the test beam. This package is required when using *Vetra* to analyse any data taken during the ACDC tests.

2.3 Monitoring Packages

Velo/VeloFullDataMonitor

Short description: provides monitoring of the NZS data, pedestal bank and noise. The contents of the ODIN bank are also analysed by a dedicated algorithm.

Velo/VeloDataMonitor

Short description: contains a set of monitoring algorithms designed especially for the VELO commissioning making use of the full data. The most important part of this package is an algorithm performing tests of the detector cabling and verifying the mapping between the TELL1 boards and the silicon sensors.

Velo/VeloTELL1Checkers

Short description: simple monitoring of the NZS data at each stage of the TELL1 emulation is provided. This provides information on the data quality and the Beetle chips condition (noisy or dead channels can be detected from an analysis of the produced histograms). Examples showing the recommended usage of the `VeloTELL1Data` class interface are also provided.

The following packages are released within the *Vetra* framework mainly for historical reasons. It is foreseen to move the software to Lbcom project in the near future.

Velo/VeloClusterDataMonitor

Short description: implements monitoring of all the information accessible via the VELO cluster's interface. It is possible to produce both a collection of histograms and NTuples for further more specific analysis if necessary. An additional algorithm `ClusterDataMoniUserHistos` is provided to facilitate more advanced user specific analysis. This package is designed to monitor clusters in data - no simulation (Monte-Carlo truth) information is available.

Velo/VeloTrackDataMonitor

Short description: this package is designed along the same lines as the cluster monitoring package. It provides monitoring for information retrieved from the track interface and can also determine both biased and unbiased residuals. Again no simulation information is used.

A `TrackDataMoniUserHistos` algorithm is also provided that can be used for non-standard user specific analysis.

2.4 External Packages

Velo/VeloDAQ

Short description: This package is part of the `Lbcom` component project and contains the decoding algorithms of all the VELO data banks. The ZS bank is decoded to `VeloLiteClusters` and `VeloClusters` that are subsequently used in the pattern recognition and track reconstruction in (`Brunel`). The decoding of the NZS data from the full banks is discussed in section 3 of this note.

Event/VeloEvent

Short description: contains all the VELO specific event classes used throughout the software.

DAQ/Tell1Kernel

Short description: contains all the definitions and typedefs related with the TELL1 emulation and NZS data decoding. These typedefs are gathered within the VeloTELL1 namespace.

DAQ/MDF

Short description: this package handles files containing real data created by the LHCb Event Builder machinery. The most important functionality of the package from the *Vetra* point of view is a set of converters to handle transformation of persistent binary MDF files into the RawEvent structure that is used in the software to access the raw banks.

3 Non-Zero Data Stream Decoding

The primary input for *Vetra* is the decoded NZS data, for the VELO this comes from decoding the `VeloFull` raw bank. In addition data from the `VeloPedestal` and `VeloError` banks [5] can be used. The `VeloPedestal` bank contains the pedestals currently used in the TELL1 board (the TELL1 board may have determined these in a processing algorithm or uploaded them from the ECS system). The `VeloError` bank contains information on synchronisation errors that have occurred during data processing. The decoding of these three VELO banks is discussed in this section.

Before the decoding procedure can be performed the binary data stream created by the Event Builder needs to be transformed into a `RawEvent` object and registered in the TES (see Fig. 2). This transformation is done automatically by the `RawDataCnvSvc` at the start of the *Vetra* job.

The `RawEvent` may be regarded as a collection of raw banks containing data from all the LHCb sub-detectors that can be accessed from within an GaudiAlgorithm using its standard interface. The `VeloError` bank is decoded into a dedicated `VeloErrorBank` class that provides a simple interface that can be used for monitoring the TELL1 behaviour. The `VeloFull` and `VeloPedestal` banks are both decoded into the `VeloTELL1Data` object class (see Fig. 3). The pedestal data is not subjected for any further processing and is used for monitoring purposes only.

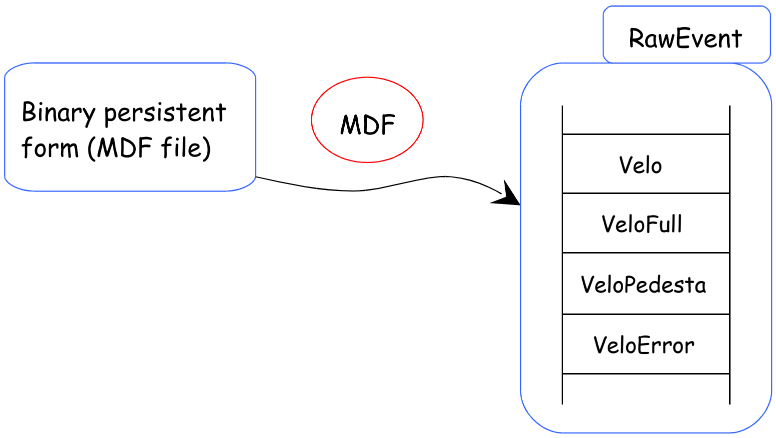

Figure 2: A binary file created by the Event Builder is transformed into the RawEvent structure using the converter service provided by the MDF package.

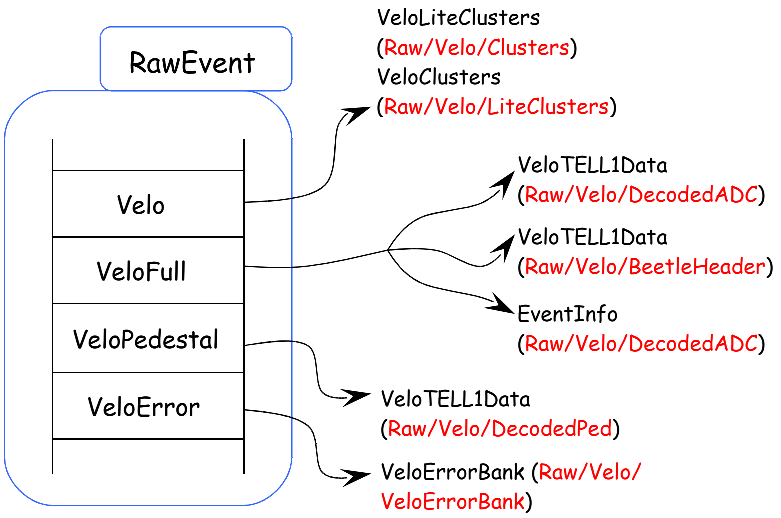

Figure 3: Decoding of the Velo raw banks. The final VELO data objects for each bank are shown with the TES location at which they are stored.

The `VeloFull` bank contains the raw ADC data, *i.e.* the digitized charge signals collected from the strips of the VELO silicon sensors. In addition the bank also contains header data from the Beetle front-end chips, this is sent together with the ADC samples and an Event Info block [1]. The Event Info block includes information from bunch counters and is decoded into a dedicated `EvtInfo` class. Four info blocks are provided for one VELO sensor (one per each processing unit of the TELL1 board). Both the headers and the ADC data are decoded into the `VeloTELL1Data` objects. The decoded ADC samples can then be processed by the TELL1 emulator.

4 TELL1 Acquisition Board Processing Emulation - VELO case.

This section provides a general description of the TELL1 emulation as implemented in *Vetra*.

4.1 The TELL1 Emulator

After a positive L0 trigger decision is obtained the data from the VELO detector is sent off for pre-processing by the `TELL1` acquisition electronic boards. The pre-processing sequence is performed by programmable FPGA [1)] processors and its purpose is to produce the ZS raw bank (VELO clusters).

Each pre-processing step is implemented as a separate algorithm. The suite of algorithms are executed by the processing units of the `TELL1` and are implemented in the low level VHDL language as a part of the `TELL1` firmware. The VHDL code is not easily human readable and hence is cumbersome to maintain, update and debug. Hence a high-level language model of the VHDL firmware has been created as a set of c-modules each of which represents one step of the pre-processing. The c-modules are a part of the *tell1Lib* and are meant to provide bit perfect results identical to those produced by the `TELL1` boards.

The c-modules are written in plain c, this makes it possible to run the code on the credit card PC module that is a part of the `TELL1` board. In order to create a reliable emulation executed within the standard LHCb software environment and keep the c-modules unchanged it was decided to adopt a wrapper-plugin approach, as described in section 5.

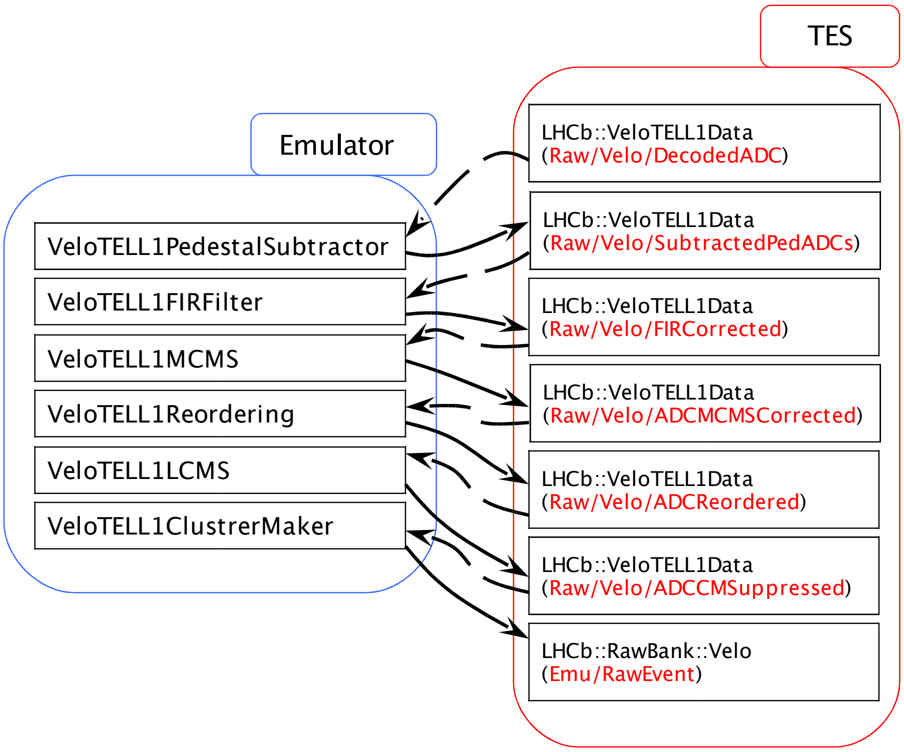

Figure 4: The algorithm sequence of the base-line TELL1 emulation for the VELO, shown together with the TES locations of the stored data.

The emulation sequence presently implemented in *Vetra* for the VELO is shown in Fig. 4. The list of algorithms constituting this base-line emulation is as follows:

- Pedestal Subtractor - subtract pedestal offset values for each channel.
- Pedestal Updater [2)] - refreshes the value of the pedestal estimate for each channel, this value is then used in the Pedestal Subtractor for the next event.

[1)]Field Programmable Gate Array.

[2)]The combined operation of Pedestal Subtraction and Pedestal Update is known as Pedestal Following.

- Digital FIR filter - responsible for removing the cross talk from the cable or other sources.
- Mean Common Mode Suppression is an algorithm that has been introduced after the ACDC3 test beam to counteract saturation effects in the Beetle chip's channels caused by deposition of large charges in those channels. It turn out that a very large charge (above 127 ADC counts [3)]) deposited on a given strip leads to severe Beetle's base line shift that affects adjacent channels. The base line shift can reach even 50 ADC counts.
- Reordering - procedure used to reorder the channels. The channels are reordered from the Beetle electronic channel order to the strip numbering order that follow the geometry of the R/ϕ VELO sensors. This sensor geometry order of the strips is required for the clusterization algorithm.
- Linear Common Mode Subtractor - removes common mode noise
- Zero Suppression (clusterization) - at this step clusters are formed from the the channels. Predefined cluster thresholds are used in the cluster finding algorithm, known as the high or seeding threshold and the low or inclusion threshold.

The sequence presented here is not the final one that will be used for the VELO, both the implementation of the algorithms and their order are expected to change in the future. However, the type of algorithms and the structure of the emulator is expected to remain, and will also be used for the ST (where, for example, the FIR algorithm and the Reordering algorithm are not required).

The performance of the algorithms are being studied and extensively tested. Verification studies have been performed [4)] using data created by using the Beetle front-end chip injection unit to produce test pulses in given electronic channels of the front-end chip. The performance is then tested using test beam data.

4.2 NZS Data Preparation and Processing

The content of the NZS raw bank for one silicon sensor is presented schematically in Fig. 5. The same structure is produced for each VELO sensor and contains data from the 2048 strips on the sensor and 256 Beetle front-end chip headers.

During the decoding process (described in section 3) the data samples and Beetle headers are separated, decoded into VeloTELL1Data objects and stored in the TES. One VeloTELL1Data object is created per sensor for both the data and the Beetle headers. The input stream of the VELO TELL1 emulation is a vector of these VeloTELL1Data objects.

Before the input data can be processed for the VELO dummy channels need to be added to the real data. This is necessary because the number of inner and outer strips on each VELO Φ sensor is an odd number and the FPGA processors are constructed to operate on packets of 32 channels. The addition of the dummy channels makes it possible to process the data from the Φ type sensors (In addition it is necessary for technical reasons to insert the dummy strips also in the data from the R type sensors).

In order to obtain bit-perfect agreement, the same procedure must also be performed for the *Vetra* TELL1 emulation. The details of the insertion of the dummy channels into the input data vector of the 2048 raw data samples (ordered according to electronic chip channel number) are shown in Fig. 6. The input data can be divided logically into 64 analogue links. Each analogue link consists of 32 read out (electronic) channels. The input data for each processing unit of the TELL1 board is made of 16 analogue links. The FPGA processors can process data in a number of parallel threads called processing channels each of which is responsible for the handling of two analogue links of data (64 samples). In order to conform properly to this hardware data processing model within the *Vetra* emulation 64 dummy channels need to be added at the end of the data stream for each FPGA processor.

The input data with added dummy channels is then formatted to interface to the c-modules as a 3-dimensional array of size [4][9][64], where the first index corresponds to the number of processing units, the second to the number of processing channels (threads) and the last one represents the 64 data samples to be processed in each FPGA thread. The correct formatting of the input data is critical for the behaviour of the reordering and clusterization algorithms.

5 Implementation of the TELL1 Emulator for the VELO

This section discusses implementation details of the TELL1 emulator for the VELO, to assist users in adding their own algorithms. The term 'engine' is used for a class that encapsulates the appropriate *tell1Lib* C module.

[3)] One ADC count unit corresponds to charge of 442 electrons.

[4)] See Appendix A.

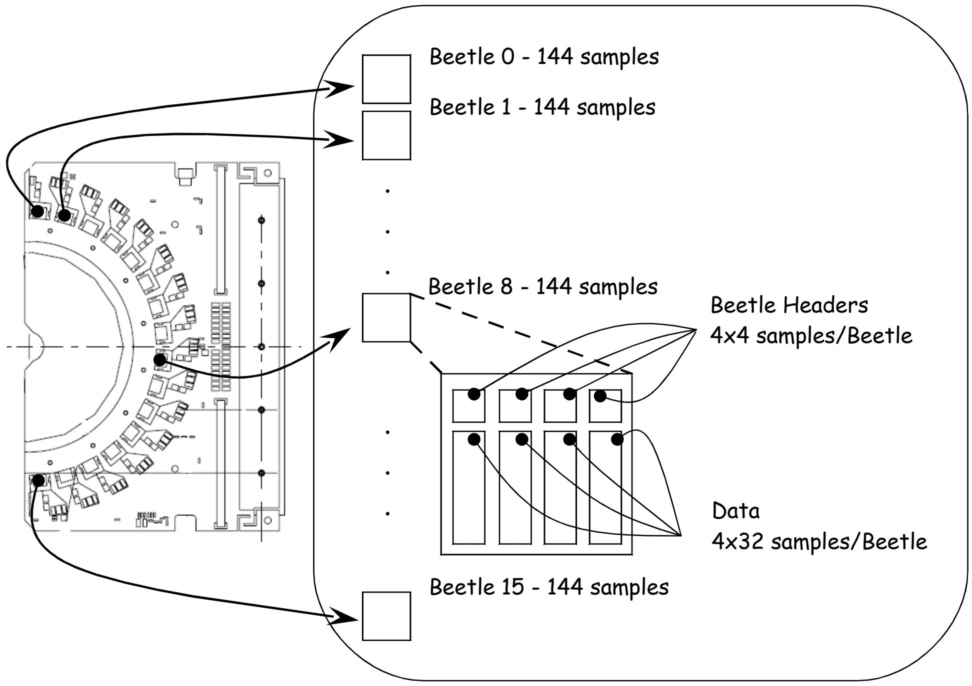

Figure 5: Contents of the VeloFull raw bank as created by the Event Builder. The data sends out by the Beetle chips over 36 clock cycles consists of the Beetle headers (4 values) and data samples (32 values).

By analogy the term 'wrapper' is applied to the *GaudiAlgorithm* that uses an engine to process the NZS data. The following sections describe the interaction between the engines and wrappers and the interfaces to the engine class. An example of how to use the engine class is provided.

5.1 Interaction between Engine and Wrapper

The relationship between an engine class and a wrapper is depicted schematically in Fig. 7. Each such class (derived from a $TELL1Engine$ base class) has its counterpart algorithm that can be executed within the $GAUDI$ framework. These classes perform the actual processing of the NZS data. The main tasks of the wrappers are to instantiate and configure the appropriate engines, format and feed the input data to them, run the processing and finally to retrieve and store the output data.

Each processing algorithm needs to be provided with a number of parameters to operate (engine configuration). For instance the c-module responsible for pedestal subtraction - velo_pedestal_process - needs to be provided with following set of parameters:

- pedestal algorithm enable flag (a single number)
- data scaling mode flag (a single number)
- zero suppression enable flag (a single number)
- header correction enable flag (a single number)
- header correction thresholds (two numbers)
- header correction values (two numbers per analogue link)
- pedestal masks - to enable or disable pedestal correction for a given channel (2048 numbers)

The number of the parameters needed for the full VELO setup for all algorithms is estimated to be of the order of 10^6. The parameters can be retrieved twofold:

- from the option files if the static configuration is chosen
- dynamically from the Condition Data Base

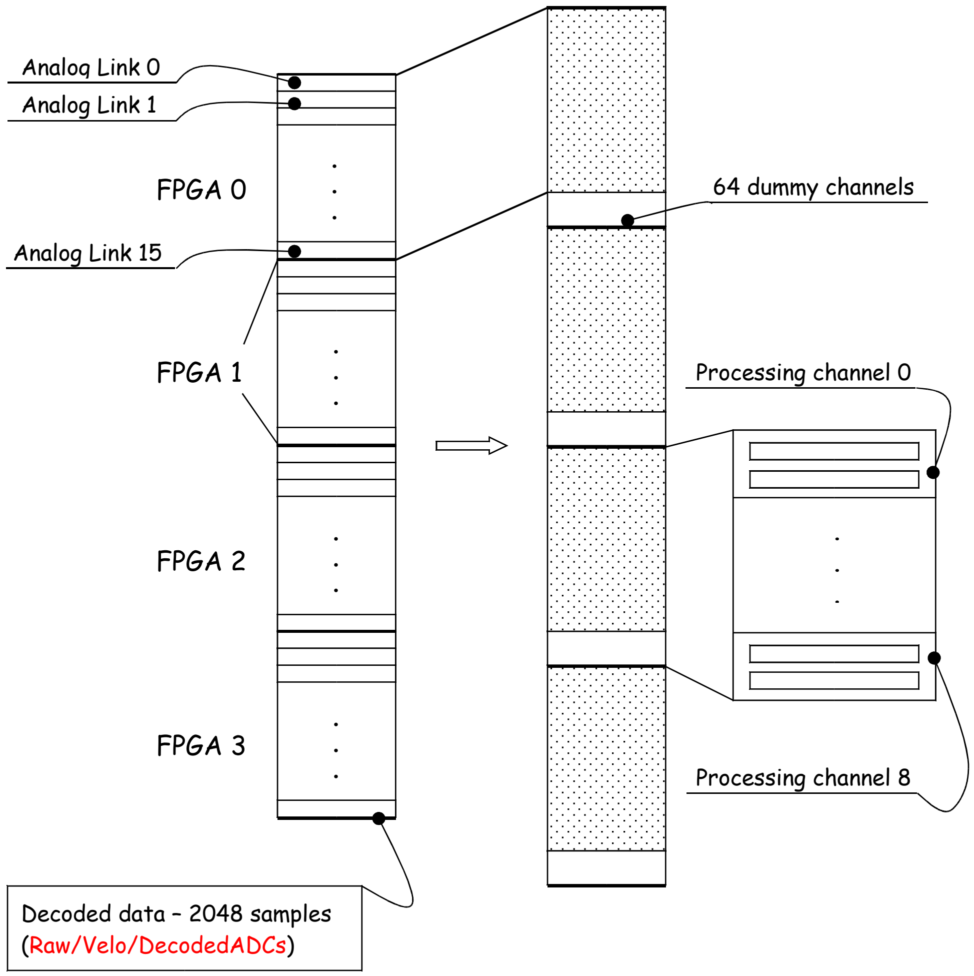

Figure 6: Preparation of the decoded NZS data for the TELL1 emulation. Before processing two additional analogue links of dummy channels need to be added to the data stream of one FPGA processor.

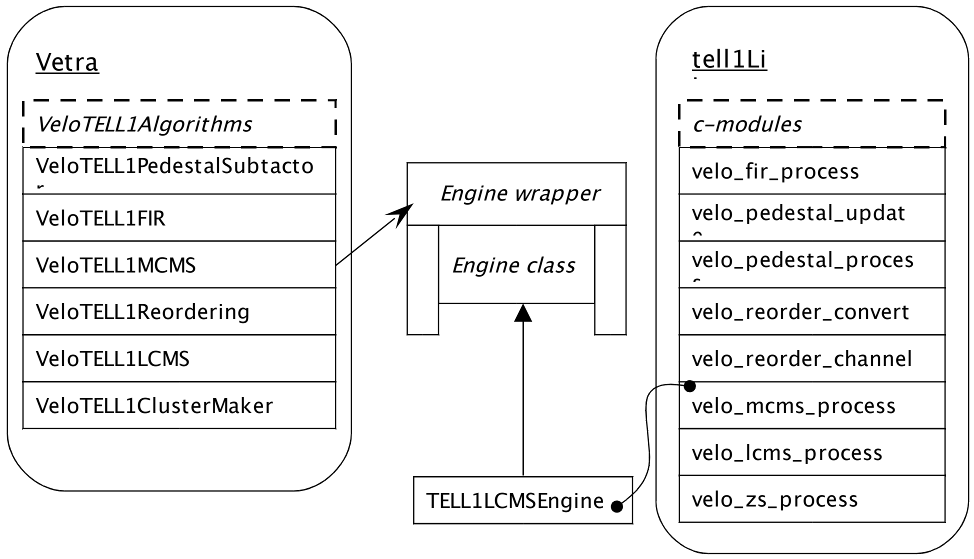

Figure 7: Dependency between the engine classes and the wrappers. Each engine class encapsulates the appropriate module from the *tell1Lib* and is used as plug-in by the corresponding wrapper algorithm.

Detailed description of the configuring procedure will be given in the last part of this chapter.

All the data that are used by the emulator's algorithms is stored inside the *TES*. As the TES is based on an abstraction of the standard template library (STL), and as it is not possible to use the STL containers directly in the C language, all the input data must be properly formatted to conform to an engine interface before it can be passed to the engine for processing. This simply requires a logical rearrangement of the data from the STL vectors into plain arrays. However, it is important to take care of the proper memory handling for these insecure table data types. A more detailed description of the engine modules data interface is given below.

The wrapper-engine pattern that has been used for the TELL1 emulator implementation has been proved successful with the VELO test beam data samples. The software created proved to be fast and stable. The separation of the *GAUDI* environment technical overhead from the actual code that models the TELL1 processing allows the developers of the *tell1Lib* software to focus on providing the C code only, without worrying for instance about problems with data storage technology used in the LHCb software.

5.2 Engine (c-module) Data Interface

The data interface of each c-module corresponds to the hardware implementation of the NZS data manipulation performed by the FPGA processors. As the data processing is done in parallel by a number of threads it is very convenient to align the data as a multi dimensional array. The number of all elements in the table is equal to a sum of the number of electronic channels (2048) of the VELO silicon sensor – each TELL1 board operates on the data from one VELO sensor – and the number of the dummy channels (256).

The appropriate functionality for the data transformation from a STL container into a plain array has been implemented inside the $TELL1Engine$ base class. The details of the implementation are discussed below using the pedestal subractor process as an example - the same pattern is used for all other modules and wrappers.

It was decided to use typedefs rather than explicit array objects to increase the security of the memory management. All the definitions needed are implemented in the $VeloTell1Core$ header file and placed in the $DAQ/Tell1Kernel$ package. Also, all the data manipulation that is done outside the c-modules are performed using the STL algorithms only to ensure that the data translation will not be corrupted. For instance the type of array used to pass the input data to the pedestal module has been aliased as:

```
typedef int Data [PP_FPGA][PROCESSING_CHANNELS][CHANNELS];
```

where the indices denotes the number of processing units per TELL1 board (4), the number of threads per FPGA processor (9) and the number of physical channels processed in one thread respectively (64).

For the sake of performance [5)] it was decided to use one raw memory buffer for each engine to store the data. Initialisation and copying of this chunk of memory is done by using special functions designed to operate on raw memory: *std::memset* and *std::memcpy* to initialize and copy the memory buffer respectively.

The data flow between each engine and wrapper is as follow:

- initialisation of the memory buffer that will hold the data to be processed (the buffer is of type Data)

```
std::memset(**m_cModuleData, 0, sizeof(Data));
```

- copying of the input data (of type $std :: vector\langle signed\ int\rangle$) to the buffer

```
std::memcpy((**m_cModuleData), &(*inData().begin()), sizeof(Data));
```

- after the processing the memory buffer contains the output data that needs to be transfered back to the wrapper

```
std::memcpy(&(*outData().begin()), (**m_cModuleData), sizeof(Data));
```

5.3 Implementation Example

In order to explain how a given engine class is employed to preform processing we will continue to use the pedestal subtraction algorithm as an example. In this case the *Tell1PedestalProcessEngine* class is used to remove pedestal noise from the input data by the *VeloTELL1PedestalSubtractor* algorithm. One processing object is created for each TELL1's data stream. Unique configuration applied for each engine allows to take into account differences in noise generated on different sensors (and in turn setting different clusterization thresholds), maps of dead strips, Beetle cross talk corrections *etc.* This approach makes the emulation to be much closer to the reality than in the case when single processing object would be used to process all the data.

The collection of processing object is defined as one of the wrapper's data members:

```
std::map<unsigned int, TELL1PedestalProcessEngine*> m_pedestalEngines;
std::map<unsigned int, TELL1UpdateProcessEngine*> m_updateEngines;
```

[5)] All the performance studies were done by Kurt Rinnert.

Instantiation and configuration is preformed once per job. All the parameters required by the engine can be set during that step using its public interface (all the needed parameters are retrieved by the wrapper). The following example shows how to create a new processing object and set its enable flag parameter (single number) and strips mask (an array of numbers).

```
m_pedestalEngines[tell1]=new TELL1PedestalProcessEngine();
m_pedestalEngines[tell1]->setProcessEnable(m_pedestalEnableMap[tell1]);
m_pedestalEngines[tell1]->setLinkMask(m_linkMaskMap[tell1]);
```

After the configuration each processing object can accept the input data. The main purpose of this preparation step is to set up all the parameters that are needed for the processing (see 5.1).

```
m_pedestalEngines[tell1]->setInData(rawADCs);
m_pedestalEngines[tell1]->runSubtraction();
subPedADCs=m_pedestalEngines[tell1]->outData();
```

where the rawADCs and subPedADCs are containers with the NZS data from one sensor before and after subtraction respectively.

6 Conclusions

This note describes the *Vetra* project which is dedicated to the analysis and monitoring of non-zero suppressed (NZS) data. The project contains NZS data bank decoding and complete TELL1 electronic board emulation. The processing allows the reproduction of the zero suppressed data bank that would result from the TELL1 board. This zero suppressed data bank is the standard input data for the LHCb reconstruction software. The emulator uses part of the *tell1Lib* library that models the firmware that is run on the acquisition boards. The software for the modelling of the firmware is provided as a set of C-modules that correspond to the processing stages performed in the FPGA processors.

At present, the software contains the full chain of raw data processing for the VELO detector only. However, the ST group is also preparing their software for VETRA release. If required, it would be possible accommodate any other detectors that use the TELL1 acquisition boards for data pre-processing.

The *Vetra* software has been used successfully to process and analyse data taken during the VELO test beam, data taken in the laboratory, and initial commissioning data in the experiment. At present the main development effort is focused on updating the *Vetra* for the commissioning purposes of the VELO detector. This includes work on interfacing the software with the condition data base in order to retrieve and update processing algorithms' parameters. The next update will also permit each TELL1 board to have a corresponding engine object created and individually configured.

In the final running of the experiment non-zero suppressed data will be written out at a low rate in addition to the standard zero-suppressed data. The VETRA software will then be used to determine the parameters required by the TELL1 processing algorithms and to monitor the performance of these algorithms.

References

[1] G. Haefeli, A. Gong, ‘VELO and ST non-zero suppressed bank data format‘, EDMS note 692431 v.2

[2] G. Haefeli, Contribution to the development of the acquisition electronics for the LHCb experiment, Thse EPFL, no 3054 (2004) (PhD thesis).

[3] T. Szumlak, C. Parkes, ’VELO Event Model’, LHCb-2006-054

[4] G. Haefeli at al, ’TELL1 - specification for a common read out board for LHCb’, LHCb-2003-007

[5] G. Haefeli, A. Gong, ‘VELO and ST error bank data format‘, EDMS note 694818 v.1
G. Haefeli, A. Gong, ‘VELO and ST pedestal bank data format‘, EDMS note 695007 v.1

[6] http://lhcb-comp.web.cern.ch/lhcb-comp/Frameworks/Gaudi/GaudiTutorial.htm

Appendix A

Verification procedure for the TELL1 emulation in *Vetra*.

The verification of the *Vetra* emulation is performed on the generated data produced by the charge injection unit of the TELL1 board. The purpose of this procedure is to verify the performance of the emulation against the actual firmware code run by the FPGA processors. The TELL1 board is operated with input data in pre-defined patterns (generated by putting test-pulses in the front-end chips of the detector). The signals are arranged to mimic single and multi strip clusters for both types of the VELO sensors geometry. The test data exercises every processing channel. Non-zero suppressed data and zero-suppressed data banks are produced by the TELL1. The non-zero suppressed data is then also processed through the emulator software. The zero-suppressed output bank of the emulator is then directly compared with that from the TELL1 board. It is possible to run this verification job with the processing parameters set in both static (taken form the option files) and dynamic (condition data base) modes.

Appendix B

Example of a *Vetra* job

This appendix provides an example of how to run a *Vetra* job with the baseline collection of processing algorithms. We assume here that user has access to the lxplus machine (of course it is perfectly possible to use the software on any machine with the lhcb software installed).

The first step is getpacking and building the *Vetra* application:

```
getpack Velo/Vetra v5r1
```

The default option file provided for base line processing is *options/Velo/VetraTELL1NZS.opts* and the default input file contains data taken during the ACDC3 VELO test beam. The default processing algorithms order is defined in the *options/Velo/TELL1Emulator.opts* file and for the present release it is as follow:

```
TELL1ProcessingVELOSeq.Members+={
                              "VeloTELL1EmulatorInit"
                              ,"dataTranslator"
                              ,"VeloTELL1PedestalSubtractor"
                              ,"VeloTELL1FIRFilter"
                              ,"VeloTELL1MCMS"
                              ,"VeloTELL1Reordering"
                              ,"VeloTELL1LCMS"
                              ,"VeloTELL1ClusterMaker"
                             };
```

Because during the test beam the VELO was only partially populated a special detector geometry description data base is needed. This is implemented inside *Velo/VeloACDC* package that is a part of the *Vetra* project. In order to use this geometry one needs to:

- define the data base location (via option file)

  ```
  DetectorDataSvc.DetDbLocation=$LOCATION;
  ```

  where location should be set to:

  ```
  location="$XMLDDDBROOT/DDDB/Velo/VeloACDC/ACDC3.xml"
  ```

- setup the following environmental variables:

  ```
  XMLDDDBROOT=$VELOACDCROOT/xml
  XMLCONDITIONSROOT=$XMLDDDBROOT
  ```

  by sourcing *../scripts/VeloACDC3.(c)sh* script from the *cmt/* folder of the *Vetra* project.

The *Vetra* job can be run using either static or dynamic configuration (by default the configuration is set to static). In the former case no more adjustments need to be done in the latter case one needs to edit the main option file and generate the *sqlite* file with the VetraCondition Data Base In order to create the data base one needs to run the following two scripts:

```
python/write_HP4_xml_cond.py
scripts/create_HP4_sqlite_file.sh
```

The first script will create a *xml* file with the *TELL1* conditions while the second one dump those conditions into a *sqlite* data base. After completing the above steps one needs to comment the following line from the *../options/Velo/VetraTELL1NZS.opts* file:

```
DetectorDataSvc.DetDbLocation="$XMLDDDBROOT/DDDB/Velo/VeloACDC/ACDC3.xml";
```

and uncomment this one:

```
#include "$VETRAROOT/options/Velo/VeloHP4CondDB.opts"
```

Also, this line must be uncomment in the *options/Velo/TELL1Emulator.opts*:

```
#include "$VETRAROOT/options/Velo/CondDBEmulation.opts"
```

Having completed this, one can now run a job by executing the following command from either of the *job/* or *cmt/* folders of *$VETRAROOT*:

```
../$CMTCONFIG/Vetra.exe ../options/Velo/VetraTELL1NZS.opts
```

This produces a root file and a digi file as the output. Inside the root file one can find a number of basic monitoring histograms with NZS data plotted against channel or strip number. The digi file contains real and emulated raw banks that can be subjected to further analysis (e.g track reconstruction, vertexing etc.). The monitoring stages can be controlled via TELL1Checkers.opts file.

One crucial parameter that needs to be set up is the convergence limit that indicates the number of events needed to train the pedestal following algorithm. This limit may be changed using option:

```
VeloTELL1EmulatorInit.ConvergenceLimit=2048;
```

The option can be found in the TELL1Emulator.opts file (it is recommended to run any ACDC3 data analysis with the convergence limit set to 4000).