University of Glasgow | Experimental Particle Physics

GLAS-PPE/2009-11
28th October 2009

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

VETRA - offline analysis and monitoring software platform for the LHCb Vertex Locator

Tomasz Szumlak[1]

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

Abstract

The LHCb experiment is dedicated to studying CP violation and rare decay phenomena. In order to achieve these physics goals precise tracking and vertexing around the interaction point is crucial. This is provided by the VELO (VErtex LOcator) silicon detector. After digitization, FPGAs are employed to run several algorithms to suppress noise and reconstruct clusters. This is performed by an FPGA based processing board. An off-line software project, *VETRA*, has been developed which performs a bit perfect emulation of this complex processing in the FPGAs. This is a novel development as this hardware emulation is not standalone but rather is fully integrated into the LHCb software to allow the reconstruction of full data from the detector. This software platform facilitates the development and understanding of the behaviour of the processing algorithms, the optimization of the parameters of the algorithms that will be loaded into the FPGA and monitoring of the detector performance. This framework has also been adopted by the Silicon Tracker detector of LHCb. This processing framework was successfully used with the first 1500 tracks of data in the VELO obtained from the first LHC beam in September 2008. The software architecture and utilisation of the *VETRA* project will be discussed in detail.

17th International Conference on Computing in High Energy and Nuclear Physics
21 - 27 March 2009 Prague, Czech Republic

1 Introduction

The *VETRA* project was created to facilitate the development and commissioning of the TELL1 [1] board processing algorithms. The TELL1 board performs the off detector read-out and data processing (zero suppression) of the LHCb VELO, the silicon strip vertex detector, using programmable FPGA processors. In time VETRA became a much more versatile tool and an essential part of the VELO [2] software; it has been used for laboratory testing during construction, test-beam operation and detector commissioning. It can emulate the functionality of the TELL1 board (with respect to the data processing) and produce the output that is of the same format as the one produced by the TELL1.

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 [3].

The details of the processing algorithms and analysis are not described here, this paper focuses on the structure and capabilities of the *VETRA* software project.

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 some of the C++ implementation details are given and the base-line emulation described. A summary and conclusions are presented in 6.

The functionality of *VETRA* is complimentary to that of the existing LHCb projects [4], its purposes are:

- Decoding of the Non-Zero Suppressed banks [1)] (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 NZS data stream sent out by each VELO sensor consists of 2048 numbers that represent digitized raw signals (this corresponds to the 2048 physical channels on each sensor). For the whole VELO detector this sums up to approximately 180000 numbers in the NZS data stream for every event (i.e. each time the detector is read-out).

The distinct difference between *VETRA* and the other presently available projects is that *VETRA* makes use of NZS data streams, that is the full unprocessed data from the detector. The final output of *VETRA* - the emulated Zero Suppressed cluster bank - is identical to the bank that is produced by the data acquisition board during normal physics running. This is arguably the most important feature of this application since it allows the direct comparison of the emulated and real data banks produced by the TELL1 boards. The emulated ZS bank can subsequently be used in just the same way as the TELL1 ZS bank to reconstruct tracks and vertices.

Using the structure of the LHCb software it is also possible to perform functions that would normally be performed in other projects, such as the VELO tracking. 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 Description of *VETRA* project

VETRA provides a mechanism to emulate each of the algorithms that would normally be executed in the TELL1 data processor board to perform the zero suppression of the raw (NZS) data. 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. Additional computing resources are available within the FPGA processors 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 [5] objects. Each VeloTELL1Data object contains the digitized signals from all 2048 strips from one VELO silicon sensor (see section 3 for more details).

[1)] Each software structure that contains encoded data that come from the VELO detector is called the data bank. Each type of the data bank can be broken into two parts - the bank header and the bank body. The header is used when the event building is performed, the bank body contains the actual data.

The emulation is broken up into a number of distinct phases that directly correspond to the real processing performed by FPGA (Field Programmable Gate Array) chips on the TELL1 boards. More information on this can be found in section 4. 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. This capability is also available in the emulation - no inter-stage monitoring is available on the TELL1 boards.

The remaining parts of this section provides a short description of the structure and functionality of the *VETRA* component packages.

The *VETRA* project is a relatively large one (it contains around 100 k lines of code). This forces the creation of a logical structure that aids maintainability and makes the project more transparent for the end-user. All the components can be divided (according to their functionality) into the following categories:

- Core packages
- Utility packages
- Monitoring packages
- External packages

The core packages contain the executable code and option files needed for the correct configuration of a job. Also, all the base classes, standard algorithms (inherited from the GAUDI [6] framework components) and plain classes (*e.g.* emulation engine code) used by other packages and tool interfaces are implemented within the core packages. The emulation engine code is a high level language (C) model of the VHDL firmware that is run within FPGA processors.

The utility packages contain the implementation of the TELL1 board emulator algorithms. These algorithms provide the interfaces to the engine classes. 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 in specially formatted memory that is accessible from within every processing algorithm. Each wrapper class uses its engine counterpart as a plug-in. Using this wrapper-plugin approach the GAUDI environment is directly bound with the software library that was developed by the TELL1 programmers. This feature makes the emulation follow as closely as possible the hardware and firmware structure.

The utility packages also contain software that is used to calculate and optimise the parameters of the TELL1 board processing algorithms. All algorithms related with NZS data analysis, other than monitoring, are located in this package.

The monitoring packages contain both low and high level monitoring algorithms. The low level ones provide monitoring of the NZS data, pedestal bank and noise. There also exist a set of monitoring algorithms designed especially for the VELO commissioning making use of the full data in order to perform tests of the detector cabling, verifying the mapping between the TELL1 boards and the silicon sensors and determine the time alignment of the VELO detector. The high level monitoring algorithms make use of information accessible via both VELO cluster's and track's interfaces. It is possible to produce both a collection of histograms and NTuples for further more specific analysis if necessary.

The external packages (related to the *VETRA* project but released as a part of the LHCb software core framework) are related mainly with the VELO raw banks decoding. The ZS bank (the output of the processing board) is decoded to the VELO clusters that are subsequently used in the pattern recognition and track reconstruction. The ZS data decoding sequence is common for the whole LHCb detector and is executed at the beginning of the track reconstruction phase in order to translate packed banks into detector specific software objects used in this phase. The decoding of the NZS data from the full banks is discussed in more details in section 3 of this paper.

3 The VELO Non-Zero Data Stream Handling

The primary input for *VETRA* is the decoded NZS data, for the VELO this comes from decoding the `VeloFull` raw data bank. In addition data from the `VeloPedestal` and `VeloError` banks [7] 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 Experiment Control System). The

`VeloError` bank contains information on synchronization 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 stored in a specially formated memory (the so called Transient Event Store) (see Fig. 1). This transformation is done automatically by the framework conversion services 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 a 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. 2). The pedestal data is not subjected to any further processing and is used for monitoring purposes only.

The `VeloFull` bank contains the raw ADC data, *i.e.* the digitized charge signals collected from the strips of the VELO silicon sensors.

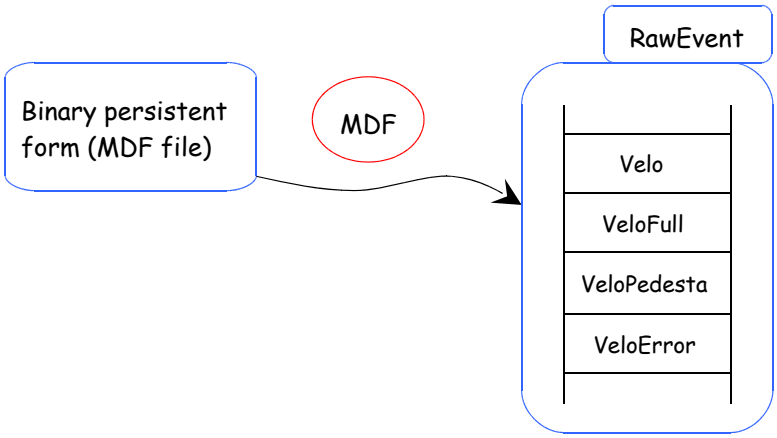

Figure 1: A binary file created by the Event Builder is transformed into the RawEvent structure using the converter service (called MDF converter).

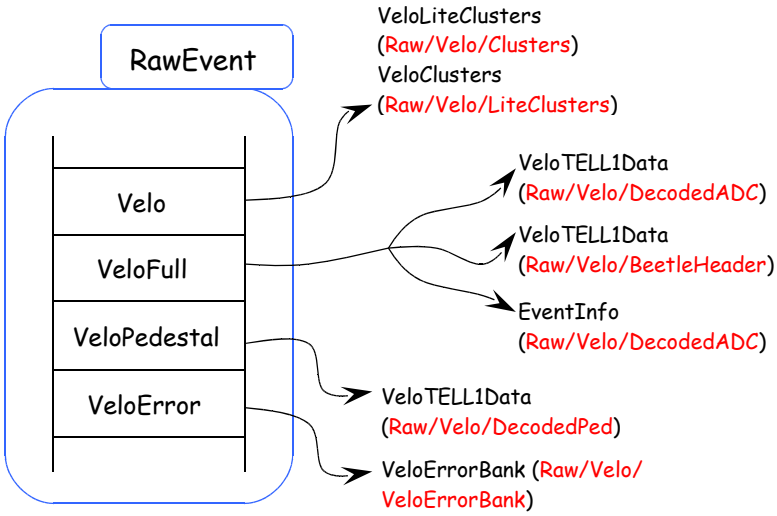

Figure 2: Decoding of the VELO raw banks. The final VELO data objects for each bank are shown with the TES location at which they are stored.

4 The VELO Acquisition Board Emulation

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

4.1 The Emulator

After a positive trigger decision is obtained the data from the VELO detector is read out for pre-processing by the TELL1 acquisition electronic boards. The pre-processing sequence is performed by programmable FPGA 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* software library 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, which is described in more detail in section 5.

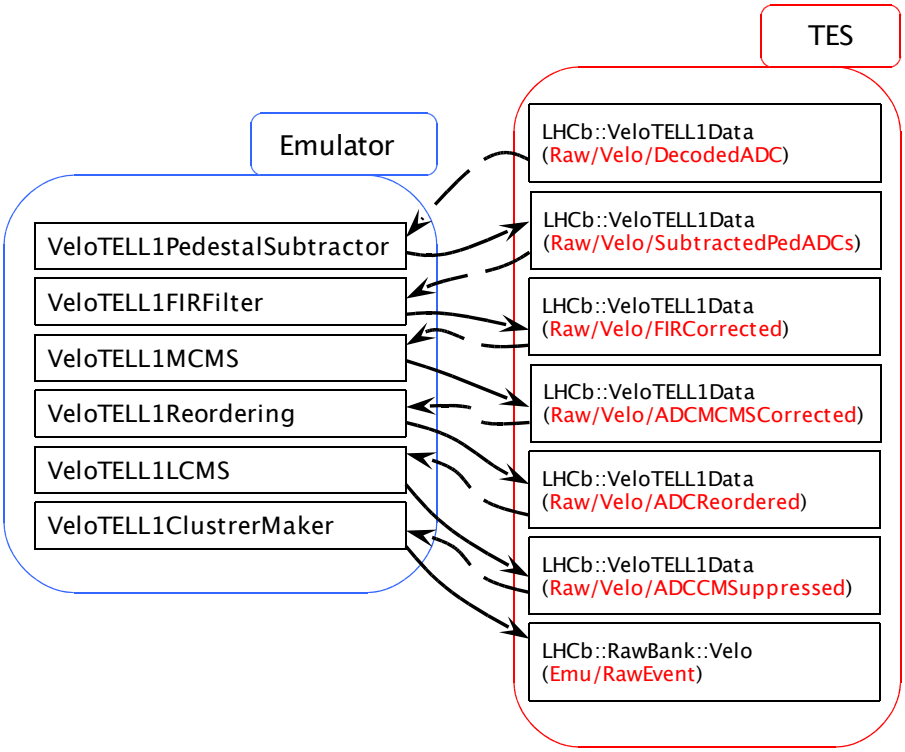

Figure 3: The algorithm sequence of the base-line TELL1 emulation for the VELO, shown together with the memory locations of the stored data in the Transient Event Store (TES) .

The emulation sequence presently implemented in *VETRA* for the VELO is shown in Fig. 3. 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.
- 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 to counteract saturation effects in the front end read-out chip's channels caused by deposition of large charges in those channels.
- Reordering - procedure used to reorder the channels. The channels are reordered from the 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.

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

- Linear Common Mode Subtractor - removes common mode noise
- Zero Suppression (clusterization) - at this step clusters are formed from 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.

4.2 NZS Data Preparation and Processing

The content of the NZS raw bank for one silicon sensor is presented schematically in Fig. 4. The same structure is produced for each VELO sensor and contains data from the 2048 strips on the sensor and 256 headers of the Beetle front-end chip. The Beetle headers samples are used for diagnosing the front-end chip state and time alignment of the detector).

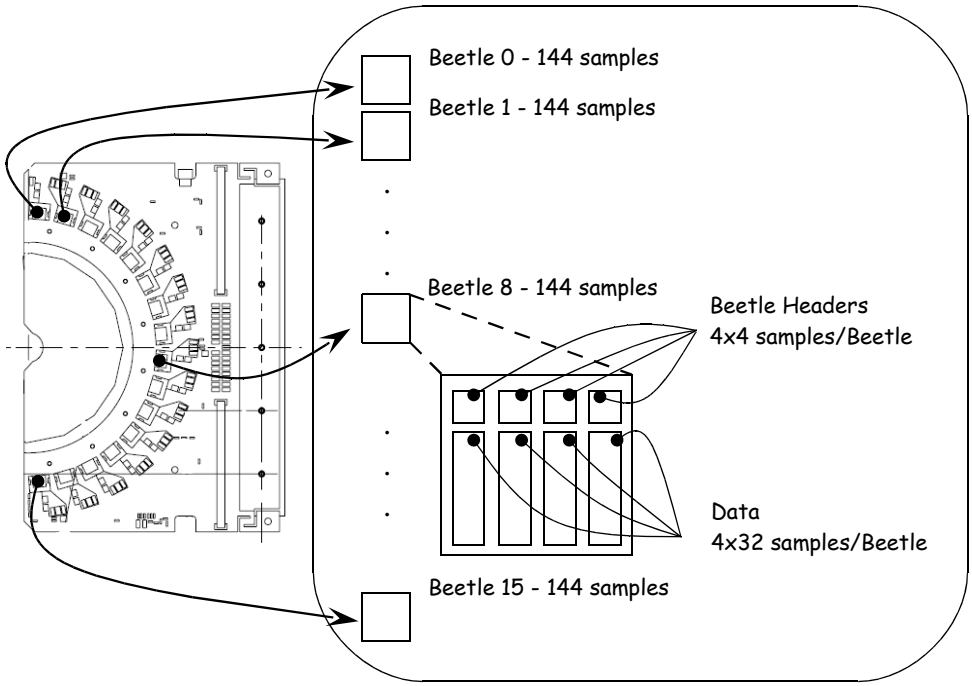

Figure 4: Contents of the VeloFull raw bank as created by the Event Builder. The data sent out by the Beetle chips over 36 clock cycles consists of the Beetle headers (4 values) and data samples (32 values).

The raw non-zero suppressed data from each sensor to the corresponding TELL1 board is 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

This section discusses implementation details of the TELL1 emulator for the VELO detector, to assist users in adding their own algorithms. The term 'engine' is used for a class that encapsulates the appropriate C-module (see section 4). By analogy the term 'wrapper' is applied to the algorithm 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

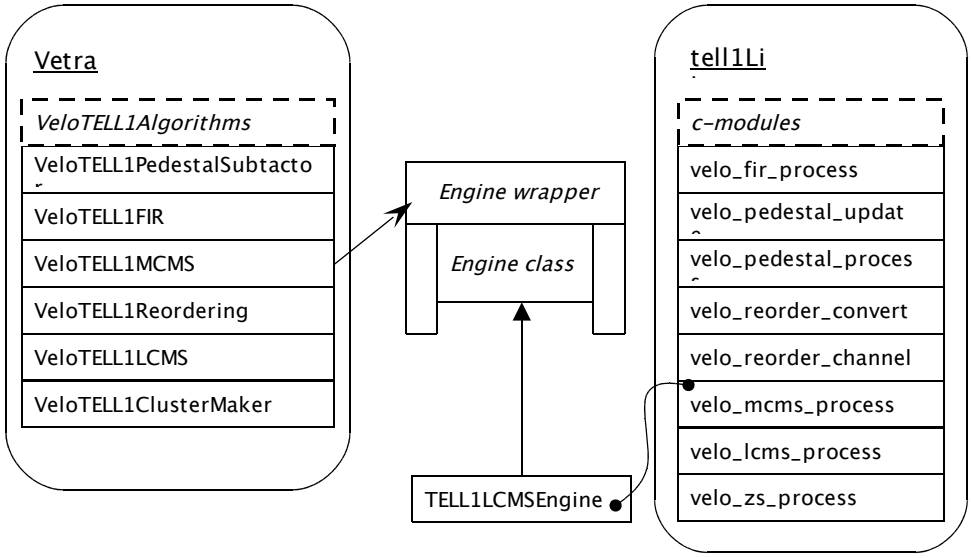

Figure 5: Dependency between the engine classes and the wrappers. Each engine class encapsulates the appropriate module from the *tell1Lib* and is used as plug-in by the corresponding wrapper algorithm.

The relationship between an engine class and a wrapper is depicted schematically in Fig. 5. 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 integer number)
- data scaling mode flag (a single integer number)
- zero suppression enable flag (a single integer number)
- header correction enable flag (a single integer number)
- header correction thresholds (two integer numbers)
- header correction values (two integer numbers per analogue link)
- pedestal masks - to enable or disable pedestal correction for a given channel (2048 integer 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. These parameters can be retrieved in two ways:

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

All the data that are used by the emulator's algorithms are stored inside the TES memory. 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 successfully demonstrated 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 base class from which in turn each engine class inherits. 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. 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];
```

The data flow between each engine and wrapper is as follow:
- initialization 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⟨signed int⟩`) 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 perform processing we will continue to use the pedestal subtraction algorithm as an example. In this case the *Tell1PedestalProcessEngine* class is used to remove pedestal offsets from the input data by the *VeloTELL1PedestalSubtractor* algorithm. One processing object is created for each TELL1's data stream. The unique configuration applied for each engine allows individual parameters to be used, so that the pedestal of each channel is subtracted.

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;
```

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 Summary and Conclusions

This paper describes the *VETRA* project which is dedicated to the analysis and monitoring of non-zero suppressed (NZS) data for the LHCb silicon detectors. 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 modeling of the firmware is provided as a set of C-modules that correspond to the processing stages performed in the FPGA processors.

The *VETRA* software has been successfully used to process and analyse data taken during the VELO test beam, data taken in the laboratory, and initial commissioning data in the experiment.

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

References

[1] Haefeli G., Bay A. and Gong A. 2006 Nucl. Inst. and Meth. in Phys. Res. A **569** p 119-122

[2] *The LHCb detector at the LHC* 2008 **JINST 3:S08005**
FPGA-based signal processing for the LHCb silicon strip detectors

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

[4] Mato P. et al. *GAUDI LHCb Data Processing Application Framework* LHCb 98-064 COMP

[5] Szumlak T. and Parkes C. *VELO Event Model* LHCb-2006-054

[6] https://twiki.cern.ch/twiki/bin/view/LHCb/LHCbSoftwareTutorials

[7] Haefeli G. and Gong A. *VELO and ST error bank data format* EDMS note 694818
Haefeli G. and Gong A. *VELO and ST pedestal bank data format* EDMS note 695007