LHCb-INT-2014-XXX
August 13, 2014

HLT monitoring in Run II

I. Chalkiadakis, D. Tonelli and E. Van Herwijnen

CERN, Geneva, Switzerland

Abstract

This is the documentation of the High Level Trigger monitoring and of a polling service developed to support it during LHCb Run II.

Contents

1 Introduction

This is an internal document about the High Level Trigger monitoring software (*VanDer-Meer/Monitor/HltMonitor* package).It starts by documenting a polling service developed inside the Gaudi framework to support the monitoring of the High Level Trigger of the LHCb detector. Its use is mandated by the change in the HLT architecture for the 2015 Run II. Its function is focused mainly on the last part of the trigger (post-HLT2). The note concludes with the documentation of the HLT I and II monitoring software.

This document is structured as follows: section *Conceptual design* describes the main agents involved in the project and the project requirements. It also gives a brief overview of what we want to achive with this project; section *Architecture* details the architecure and design of the two components that form the project : the polling service and the event selection service using it; section *Implementation and workflow* describes the steps of the execution of the polling service and the service using it (*EventSelector*); in section *How to run and test Poller* , a short description is given on the required pieces of code that should be put in a script in order to run the poller,and section *HLT monitoring* documents the HLT monitoring software.

2 Conceptual design

2.1 Main agents and concept

The main components referred to in the following documentation are the polling service and the EventSelector.

EventSelector : the Gaudi component that given a set of selection (physics) criteria can choose what events will be processed by an application.

Poller : the polling service developed.

In Run II the HLT will be split in two parts : HLT 1 and HLT 2. The output of the HLT 2 is the directory that keeps the files containing events which were accepted by the trigger. So far, the EventSelector was reading the files from that directory, taking for granted that the files were there. In Run II the poller will be scanning this directory periodically and when a new file arrives it will be given out to the EventSelector for reading. In that way, we can have control over the run that is processed at a given time : if the piquet decides that the histograms produced using a certain run are enough to classify it, then the poller can be used to stop giving out files from that run or, in contrast, give out more files from the same run for further processing.

2.2 Requirements

- The polling service should run constantly in the background.
- It should send out for reading N files from each run; N should be determined by the user.
- It should make sure that when it sends out a file, the recipient (an EventSelector) has received it and opened it successfully. If not, the user should be notified of the failure.
- The number of events that are read in total should be determined by the user. When the specified number is reached, the EventSelector should stop reading, connect to the poller and wait for another file to read from.
- The poller should keep track of the files that have been sent out and processed successfully.
- It is necessary to save the histogram sets for each run during the execution of the EventSelector and not at its finalization.

3 Architecture

3.1 Poller

3.1.1 Purpose and functionality

The poller is responsible for scanning a specified directory periodically, feeding the EventSelector with files of events and keeping a record of the files - and consequently the runs - it has given out for processing.

3.1.2 Interfaces

The poller implements the following interfaces:

Online Service (IOnlineService) : The poller is implemented as a service inside the Online environment for Gaudi. This implies that it is invoked and finalized by the main process in Gaudi, the ApplicationManager, and its function is based on the initialize()/start() and stop()/finalize() methods. It also provides the user with the ability to acquire the service's name and access its other interfaces through the queryInterface() method.

The TimerUtility class (DimTimer) : This is the element that adds the polling functionality. With it the user can set a timer and call the start() method to start it. Once the time interval is up, the timerHandler() method is called. It performs the main operations of the service, namely the polling and the file distribution, and restarts the timer [2].

The error handling interface (IAlarmHandler) : The issueAlarm() method that the poller implements allows it to handle errors arising from unsuccessful delivery and/or processing of the files by the EventSelector. The need for more than a printed message in such cases, called for the implementation of this interface instead of using the Gaudi MessageSvc.

The interface to handle a process connected to the poller (from now on called listener) (IHandleListener) : Through this interface a listener (an EventSelector) can notify the poller that it needs a new file to read from and the poller can control the listener that has asked for work, e.g remove it from the waiting list once it has been served. Furthermore it provides a method for printing out a list of the waiting listeners and a method for the listener to notify the poller that it has successfully opened (only opened not processed!) the file that it was given. This possibility of feedback is the reason why this interface was developed instead of using the Gaudi IncidentSvc, which has no feedback functionality.

The interface to handle a database for the record keeping (IOnlineBookkeep) : This interface allows the poller to create and connect to a database, update it and and check whether a file (and consequently a run) has been processed or is defect It also provides auxiliary methods to print the contents of the database and acquire the run rumber out of the complete path of the file.

Currently the database used is an SQLite database.Since there are no final plans yet for the monitoring during Run II, we should be able to handle every request of information about a run or events that belong to a certain run or file. Therefore we save in the database not only the run number but also the name of the file processed, the events we read from it and a status flag indicating the status of the process of reading the file.

3.2 EventSelector

3.2.1 Purpose and functionality

For the purpose and functionality of the EventSelector see the Architecture Design Document of Gaudi [3].

3.2.2 Interfaces

In addition to the interfaces described in the Architecture Design Document mentioned above, an EventSelector has to be modified and implement two more interfaces that will allow it to communicate with the polling service.

The modification consists of adding a locking mechanism in the next() method of the EventSelector, so that it can pause execution when it has finished reading a file and resume from the same point when the poller has provided it with a new file.

The extra interfaces that the EventSelector will have to implement are:

the *IAlertSvc* : Through it the poller gives the new file path to the EventSelector and makes it resume its operation, after having reset the input context/criteria.

the *ISuspendable* : It allows the *EventSelector* to suspend and resume its operation.

4 Implementation and workflow

4.1 Dependencies

The package *OnlineFileSelector* resides in the *Online* project. Furthermore it uses the following packages : *Event/DAQEvent*, *DAQ/MDF*, *LCG_Interfaces/sqlite* and *LCG_Interfaces/AIDA*.

4.2 Workflow

The ApplicationManager initializes the required services (for details see ADD [3]), among them the *Poller* and the *EventSelector*.

4.2.1 Poller

The initialization is completed by calling the *initialize()* method of the poller, which first connects to the database where it will store the run-related information (or creates a new database if one does not exist). The execution continues with the *start()* method, which sets the timer and starts the countdown until the next polling.

When the timer expires, the *timerHandler()* method gets called. It will first call the *poller()* method which takes as argument the directory to be scanned for new files. Then it will start distributing the file paths to the EventSelector and it will stop either when the listener has been served or when there are no more files to distribute.The process for distributing the files is as follows: First the file path is removed (*remFromfileNames()*) from the list of available file paths, which was filled during the last polling period.From that path we extract the run number (*getRunFileNumber()*) which we will use to check:

- if we had tried to open it but we encountered an error and the file should not be tried again (*isDefect()*),
- if we have already opened it and is currently being processed (*isBookKept()*), or
- if the file has already been processed (*isProcessed()*)

The run extraction method takes as argument a regular expression to match with the run number. If one of the above mentioned cases occurs we skip the processing of the current run, otherwise we pass the file path on to the listener,which is then removed from the waiting spot.

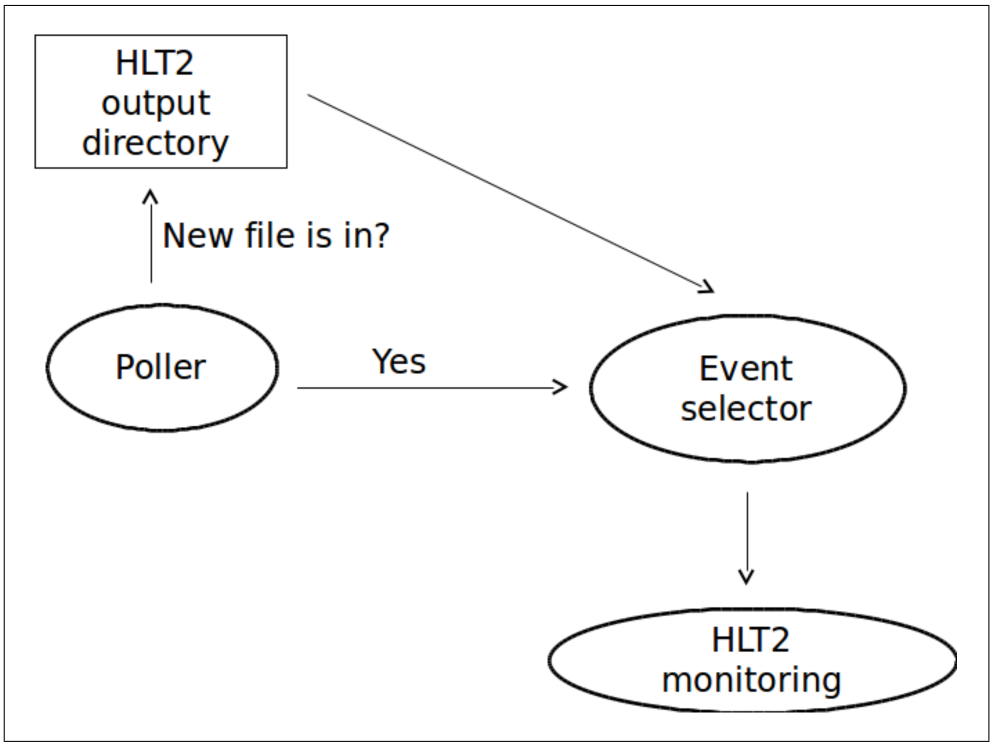
Regarding the removal of the listener there are three cases :

- the communication between the *Poller* and the *EventSelector* takes place (either successful opening of the received file or not) and then the listener is removed (from inside *alertSvc()*)
- the listener does not manage to acquire the waiting lock before going idle -erroneous behaviour, removed from the list from inside *goIdle()*
- the listener fails for some reason. To cover this case there is a removal taking place inside *timerHandler()*, after the delivery of the file. In normal cases the listener will have been already removed from *alertSvc()*

When the distribution of the files is over, we empty the vector holding the remaining file paths (if any) and the timer is reset.

At the end of the execution cycle, the *stop()* method is called which stops the service and the countdown of the timer. It is followed by a call to the *finalize()* closes the connection to the file database.

Figure 1: *The function of the polling service*



4.2.2 EventSelector

In the initialization of the *EventSelector* pointers to other services are acquired. What we need to add to an *EventSelector* to make it possible for it to use the poller are the following services, for which we acquire the necessary pointers : *FilePoller*, *HistogramDataSvc*, *RootHistSvc* and *HistogramPersistencySvc*. For details on the functionality of the latter three see [3]. Then we create the lock that will control the waiting/resuming of the *EventSelector* during its interaction with the poller, and we initialize its value to zero, which means that at the beginning of its execution the *EventSelector* will wait for input from the poller. Finally we initialize the counter *m_evtCount*, which counts the events read from a file and helps in deciding when the *EventSelector* will go idle. The *start()/stop()* methods are inherited from the base class *Service*.

The *EventLoopMgr* will use the *EventSelector* to acquire the next event for processing -the *next()* method of the *EventSelector* gets called. After two initial checks regarding saving the histograms and updating the database (see later), the *EventSelector* will check whether there is an input context (*pCtxt != 0*). If there is not, the *EventSelector* will return StatusCode::FAILURE.

If there is an input context, we first have to check if this is the first connection (meaning *m_firstConnection == 0*) to an input file that the *EventSelector* is trying to establish. Without this check we cannot safely call the *receiveData()* method to read the next event.

- If it is the first connection the *EventSelector* will finally go idle, that is it will connect to *poller* and wait for an input file path (by calling the *goIdle()* method).

- If it is not the first connection, the *EventSelector* will call *receiveData()* to read the next event; if the call is successful the *EventSelector* will increase the corresponding event counters and return StatusCode::SUCCESS. If it cannot read the next event (e.g. because there are no more events in the file), the *EventSelector* will go idle after updating the database with the number of events it has read from the last input file and saving the histogram sets.

For the next calls of the *next()* method there are two checks preceding the input context check.

The first *m_maxNoEvt == m_evtCount* checks whether we have reached the maximum number of events we want to read from one file. If we have, then the *EventSelector* updates the database with the number of events so far read, resets the event counter, saves the histograms and then goes idle, waiting for the next file.If we want to read until the end of the file, we should set the *m_maxNoEvt* to -1.

The second *m_runNum* $\neq$ *m_prevRun* checks whether the run has changed. If yes, it resets the event counter and changes the name of the file which will contain the next set of histograms.Then the *EventSelector* continues reading events from the new file (and thus run). This check covers the case where the minimum number of events to produce a set of histograms is bigger then the events we have read from a file before changing it (e.g. covers the case of a very small run).

After having gone idle, the *EventSelector* will wait at the corresponding lock (*m_suspendLock*) until the poller serves it with a file. In order for this to happen, the poller will call the *alertSvc()* method of the *EventSelector* passing as argument the file path to read from. This method before exiting will change the value of the *m_firstConnection* to 1, to signal that there has been at least one connection established.Next, it will get the current input context and update it with the new file path by calling *resetCriteria()*.If the context update is successful, the *EventSelector* reports back to the poller the *successful opening* of the new file using *statusReport()*. This means that the file and run will be added to the file database with a status flag of 0 : the file has been opened but we have not finished processing it.Then the poller will remove the *EventSelector* from the waiting spot, extract the new run number from the file path and generate accordingly the new histogram set name.It will finally resume the operation of the *EventSelector*, which will continue from the point where the last call of *goIdle()* was made.

At the end of the execution cycle, the inherited *stop()* method is called and then the *finalize()*. In the *finalize()* method we reset the counters, remove the acquired interfaces by setting the pointers to them to 0 and we delete the wait/resume lock of the *EventSelector*.

4.2.3 Details about Poller methods and member variables

- std::string m_scanDirectory : this is the directory that the poller will check periodically for files, which will then distribute to the listeners

- int m_alrmTime : the time interval between two successive pollings
- std::deque<std:string> m_fileNames : the list of the file paths that were found in the scanned directory during the last polling.

  The *std::deque* container was chosen for the above list as it implements a First In First Out structure, which is what we want especially for the listeners : the poller should serve first the listener who asked first for a new file. Furthermore, a *std::deque* performs better than a vector at the removal of its first element and grows more efficiently in size.

- mutable <IAlertSvc*> m_EvtSelector : the listener waiting for input from the poller. It is kept in the form of a pointer to its *IAlertSvc* interface.
- sqlite3* m_FileInfo : the handler of the database keeping the file/run information.

  The necessary accesor methods for the above private member variables are provided : *addTofileNames,remFromfileNames.*

  Some more implementation details about methods that were mentioned in the workflow:

- StatusCode poller(const std::string scan_path) : This is the method that performs the polling.It uses *readdir_r* to scan the directory. If it finds a file, it adds it to the file list whereas if it finds another directory it calls itself recursively to scan it (apart from the current (¨ . ¨) and previous (¨ .. ¨) directories).
- StatusCode statusReport(StatusCode status,const std::string file) : This method gives feedback to the poller about the *opening* of the file from the *EventSelector*.If the opening was unsuccessful the method *issueAlarm()* is called, which inserts an error record into the database containing the file name, the run number and the value of -1 in the event counter and status flag.If the opening was succesful, we insert the file name into the database, marking it as unprocessed/currently in process (*StatusFlag == 0*), using the method *markBookKept()*.
- const StatusCode issueAlarm(const std::string msg) : In case the *EventSelector* does not manage to open a file from the path it has received, it is not enough to print a message reporting it. This method provides support for this case by adding an entry in the database with the name of the file that failed to be opened, the run it belongs to and the value of -1 in the *StatusFlag* and '*TotalEvents* fields.
- StatusCode isBookKept(const std::string file) : It checks if the file passed as its argument is in the database with *StatusFlag == 0*,i.e. it is currently being processed.
- StatusCode isProcessed(const std::string file) : It checks if the file passed as its argument is in the database with *StatusFlag == 1*.

- StatusCode updateStatus(const std::string file, int events) : It updates the record in the database which corresponds to the file passed as argument, with the number of events passed as the second argument.

4.2.4 Details about EventSelector methods and member variables

- std::string m_HistoDirectory : this is the output directory for the histogram sets saved by the poller
- StatusCode goIdle() const : This method is called for the transition of the *EventSelector* from a running to an idle state.It first sets the *m_isWaiting* flag to true. Then the *EventSelector* is added to the listener spot of the poller by calling *addListener()*.If the registration to the poller is successful, the *EventSelector* waits at the *m_suspendLock*. If it cannot acquire the waiting lock it removes itself from the listener spot by calling *remListener()*. Either after its removal or when it resumes execution after having successfully waited, the *m_isWaiting* flag is set to false.
- StatusCode saveHistos() const : This method is responsible for saving the histogram sets. Normally these steps would be carried out when the event loop would have finished, but since the poller will be constantly running this will happen only at the end of the processing of all runs. The need to save the sets at the end of each run called for the integration of this code here. The histograms are traversed, conversed to their permanent representation (e.g. ROOT) and saved. In order to create a new file for each saveset and not overwrite the same saveset again and again, the histograms must be reset and a new file name has to be selected for the histogram file. Whenever we have to save the histograms (that is either when we have reached the necessary number of events or when the run has changed), the conversion service (*m_RootHistSvc*) is stopped and finalised. The methods *sysStop()* and *sysFinalize()* are called instead of the *stop()* and *finalize()* to avoid execution errors because these methods change the state of the machine according to their function from running to configure.If a new file is going to be opened next, that is if a new run is going to be processed, the name of the histogram file is changed. It is in this method (*setNewHistosName()*) that we reintialize and restart the conversion service.

5 Implementation and workflow of the FilePollerN-Files

5.1 Introduction

In addition to the simple poller described above, the following part describes the functionality of the *FilePollerNFiles*, which does not send a run for processing unless there are N files available for reading from that run.To support this functionality we had to add a table in the database, which holds the run number and the number of available files that belong to that run.As soon as this number is greater than or equal to N, we start distributing the files to the event selector.

5.2 Dependencies

–No extra dependencies–

5.3 Workflow

The ApplicationManager initializes the required services (for details see ADD [3]), among them the *Poller* and the *EventSelector*.

5.3.1 Poller

The initialization is completed by calling the *initialize()* method of the poller, which first connects to the database where it will store the run-related information (or creates a new database if one does not exist). The execution continues with the *start()* method, which sets the timer and starts the countdown until the next polling.

When the timer expires, the *timerHandler()* method gets called. It will first call the *poller()* method which takes as argument the directory to be scanned for new files. Then it will start distributing the file paths to the listener and it will stop when there are no more files to distribute.The process for distributing the files is as follows: First the file path is removed (*remFromfileNames()*) from the list of available file paths, which was filled during the last polling period.From that path we extract the run number (*getRunFileNumber()*) which we will use to check whether we have to process the run. First we acquire the counter of the files we have yet to read from this run.If the counter is -1, we do not need to read any more files from this particular run, so we 'free' the *EventSelector* that handled it (*clearRun(run)*), that is, the event selector is now able to be assigned a different run; then we skip the current file and move on to the next one.

Next, using the file name we check :

- if we had tried to open the file but we encountered an error and the file should not be tried again (*isDefect()*),
- if the file has already been processed (*isProcessed()*), or
- if we have already opened it and is currently being processed (*inProcess()*)

If one of the above mentioned cases occurs we skip the processing of the current file.All these methods check the *StatusFlag* of the corresponding file, which has four possible values :

- -1 : erroneous file
- 0 : the file has just been inserted in the database

- 1 : the file has already been processed
- 2 : the file is currently being processed

If none of the above methods returns *StatusCode::SUCCESS* we check whether we have available the minimum number of files to start processing the run.If that is the case, we mark the run as ready to be processed. A file whose run is 'ready' to be processed is then passed on to the listener.

When the distribution of the files is over, we empty the vector holding the remaining file paths (if any) and the timer is reset.

At the end of the execution cycle, the *stop()* method is called which stops the service and the countdown of the timer. It is followed by a call to the *finalize()* method which closes the connection to the file database.

5.3.2 EventSelector

In the initialization of the *EventSelector* pointers to other services are acquired. What we need to add to an *EventSelector* to make it possible for it to use the poller are the following services, for which we acquire the necessary pointers : *FilePoller*, *HistogramDataSvc*, *RootHistSvc* and *HistogramPersistencySvc*. For details on the functionality of the latter three see [3]. Then we create the lock that will control the waiting/resuming of the *EventSelector* during its interaction with the poller, and we initialize its value to zero, which means that at the beginning of its execution the *EventSelector* will wait for input from the poller. Finally we initialize the counter *m_evtCount*, which counts the events read from a file and helps in deciding when the *EventSelector* will go idle. The *start()/stop()* methods are inherited from the base class *Service*.

The *EventLoopMgr* will use the *EventSelector* to acquire the next event for processing -the *next()* method of the *EventSelector* gets called. After two initial checks regarding saving the histograms and updating the database (see later), the *EventSelector* will check whether there is an input context (*pCtxt != 0*). If there is not, the *EventSelector* will return StatusCode::FAILURE.

If there is an input context, we first have to check if this is the first connection (meaning *m_firstConnection == 0*) to an input file that the *EventSelector* is trying to establish. Without this check we cannot safely call the *receiveData()* method to read the next event.

- If it is the first connection the *EventSelector* will finally go idle, that is it will add itself to the listener spot of the *poller* and wait for an input file path (by calling the *goIdle()* method).
- If it is not the first connection, the *EventSelector* will call *receiveData()* to read the next event; if the call is successful the *EventSelector* will increase the corresponding event counters and return StatusCode::SUCCESS. If it cannot read the next event (e.g. because there are no more events in the file), the *EventSelector* will go idle after updating the database with the number of events it has read from the last input

file.If the file was the last one available from the particular run that is currently being processed, the histogram sets will be saved as well.

For the next calls of the *next()* method there are two checks preceding the input context check.

The first *m_maxNoEvt* == *m_evtCount* checks whether we have reached the maximum number of events we want to read from one run. If we have, then the *EventSelector* updates the database with the number of events read from the last file, saves the histograms and then goes idle, waiting for the next file/run.If we want to read until the end of every file of the run we are processing, we should set the *m_maxNoEvt* to -1.

The second *m_runNum* $\neq$ *m_prevRun* checks whether the run has changed. If yes, it resets the event counter and changes the name of the file which will contain the next set of histograms.Then the *EventSelector* continues reading events from the new run.

After having gone idle, the *EventSelector* will wait at the corresponding lock (*m_suspendLock*) until the poller serves it with a file. In order for this to happen, the poller will call the *alertSvc()* method of the *EventSelector* passing as argument the file path to read from. This method before exiting will change the value of the *m_firstConnection* to 1, to signal that there has been at least one connection established.Next, it will get the current input context and update it with the new file path by calling *resetCriteria()*.If the context update is successful, the *EventSelector* reports back to the poller the *successful opening* of the new file using *statusReport()*. This means that the file and run will be updated in the file database with a status flag of 2 : the file has been opened but we have not finished processing it.Then the poller will remove the *EventSelector* from the listener spot, extract the new run number from the file path and generate accordingly the new histogram set name.It will finally resume the operation of the *EventSelector*, which will continue from the point where the last call of *goIdle()* was made.

At the end of the execution cycle, the inherited *stop()* method is called and then the *finalize()*. In the *finalize()* method we reset the counters, remove the acquired interfaces by setting the pointers to them to 0 and we delete the wait/resume lock of the *EventSelector*.

5.3.3 Details about Poller methods and member variables

- std::string m_minFileNum : this is the minimum number of files of a run that should be available in order to start processing the corresponding run.
- std::string m_scanDirectory : this is the directory that the poller will check periodically for files, which will then distribute to the listeners
- int m_alrmTime : the time interval between two successive pollings
- std::deque<std:string> m_fileNames : the list of the file paths that were found in the scanned directory during the last polling.

  The *std::deque* container was chosen for the above list as it implements a First In First Out structure, which is what we want especially for the listeners : the poller should serve first the listener who asked first for a new file. Furthermore, a *std::deque* performs better than a vector at the removal of its first element and grows more efficiently in size.

- mutable <IAlertSvc*> m_EvtSelector : the listener waiting for input from the poller. It is kept in the form of a pointer to its *IAlertSvc* interface.
- sqlite3* m_FileInfo : the handler of the database keeping the file/run information.

The necessary accesor methods for the above private member variables are provided : *addTofileNames,remFromfileNames*.

Some more implementation details about methods that were mentioned in the workflow:

- StatusCode poller(const std::string scan_path) : In this implementation of the polling function we have added the functionality of inserting a run in the database or increasing the run's file counter.When we scan a file in the directory, we check the buffer to make sure we have not buffered it before (if (find(m_fileNames.begin(), m_fileNames.end(),n_path) == m_fileNames.end())).If we have not, then it is added to the buffer (*m_fileNames*).For this version of the poller, we proceed to check if the file is in the files table of the database and if it is not (*StatusCode::FAILURE == isBookKept(file_regexp)*) we insert it in the table *FileRecords* with the *StatusFlag* set to zero.Then we check if the run (that the file belongs to) is in the run database: if not, we insert it and if it is, and we still need to process the run (*if (-1 != getCounter(run))*), we increase the file counter.
- StatusCode statusReport(StatusCode status,const std::string file) : This method gives feedback to the poller about the *opening* of the file from the *EventSelector*.If the opening was unsuccessful the method *issueAlarm()* is called, which inserts an error record into the database containing the file name, the run number and the value of -1 in the event counter and status flag. If the opening was succesful, we insert the file name into the database, marking it as currently in process (*StatusFlag == 2*), using the method *SetInProcess()*.
- const StatusCode issueAlarm(const std::string msg) : In case the *EventSelector* does not manage to open a file from the path it has received, it is not enough to print a message reporting it. This method provides support for this case by adding an entry in the database with the name of the file that failed to be opened, the run it belongs to and the value of -1 in the *StatusFlag* and '*TotalEvents* fields.Since such a file will not be processed again, it also decreases the counter of available files of the particular run. If there are no more files available it also frees the event selector which was assigned to that run (*clearRun()*).

- StatusCode isBookKept(const std::string file) : It checks if the file passed as its argument is in the database with *StatusFlag == 0*,i.e. it has just been inserted into the database..
- StatusCode isProcessed(const std::string file) : It checks if the file passed as its argument is in the database with *StatusFlag == 1*.
- StatusCode updateStatus(const std::string file, int events) : It updates the record in the database which corresponds to the file passed as argument, with the number of events passed as the second argument.It also decreases the counter of available files of the particular run, since this method is called when we have finished processing one file. If it was the last available file, we mark the run as processed,i.e. its file counter is set to -1.
- StatusCode insertRun(const std::string run) : This method inserts the run into the corresponding table and initializes the file counter to 0.
- StatusCode existsRun(const std::string run) : This method checks whether the run passed as argument is present in the database.
- StatusCode distributeFile(const std::string run, const std::string path_name) : This method takes care of the distribution of a file to the event selector. If the current run is empty or is the same as the run that *path_ name* belongs to, we forward the file to the event selector. If it is not,then we return successfully as it is implied that the event selector is busy handling a different run.

5.3.4 Details about EventSelector methods and member variables

- std::string m_HistoDirectory : this is the output directory for the histogram sets saved by the poller
- StatusCode goIdle() const : This method is called for the transition of the *EventSelector* from a running to an idle state.It first sets the *m_isWaiting* flag to true. Then the *EventSelector* is added to the listener spot of the poller by calling *addListener()*.If the registration to the poller is successful, the *EventSelector* waits at the *m_suspendLock*. If it cannot acquire the waiting lock it removes itself from the listener spot by calling *remListener()*. Either after the removal from the list or when it resumes execution after having successfully waited, the *m_isWaiting* flag is set to false.
- StatusCode saveHistos() const : This method is responsible for saving the histogram sets. Normally these steps would be carried out when the event loop would have finished, but since the poller will be constantly running this will happen only at the end of the processing of all runs. The need to save the sets at the end of each run called for the integration of this code here. The histograms are traversed, conversed to their permanent representation (e.g. ROOT) and saved. In order to create a new file for each saveset and not overwrite the same saveset again and again, the

histograms must be reset and a new file name has to be selected for the histogram file. Whenever we have to save the histograms (that is either when we have reached the necessary number of events or when the run has changed), the conversion service (*m_RootHistSvc*) is stopped and finalised. The methods *sysStop()* and *sysFinalize()* are called instead of the *stop()* and *finalize()* to avoid execution errors because these methods change the state of the machine according to their function from running to configure.If a new file is going to be opened next, that is if a new run is going to be processed, the name of the histogram file is changed. It is in this method (*setNewHistosName()*) that we reintialize and restart the conversion service.

NOTE: The first histogram file containing actual content will be the one with the correct name (with respect to the run the histograms come from) and not whichever name is set in the python configuration script under *HistogramPersistencySvc().OutputFile.*

6 How to run and test Poller

6.1 Prerequisites

This section will give some instructions on how to run the polling service. It takes for granted that the *EventSelector* used will have been modified to use the poller, as described above.

The *OnlineFileSelector* package should be in the same *VanDerMeer* project directory, or the monitoring scripts should be able to "see" it by including it in the *requirements* file of their *cmt* directory.

.

6.2 Running

In order to run the *poller*, an instance of it and of a modified *EventSelector* should be instantiated and its variables initialized (scan directory, alarm timer). Next follow the code excerpts of a testing python script that do the job :

- Import : from Configurables import (

  LHCb_MDFOnlineEvtSelectorNFiles,

  LHCb_FilePollerNFiles,

)

- Initial output file : HistogramPersistencySvc().OutputFile = ""

- Poller instantiation and setting parameters :

  poller = LHCb_FilePollerNFiles('Poller')

  appMgr.ExtSvc.append(poller)

poller.scanDirectory = "/daqarea/lhcb/data/2014/RAW/FULL/LHCb1/TEST"
poller.MinimumFileNum = 2
poller.alarmTime = 3
poller.DbName = "./OnlineFileProcessing.db"

- EventSelector instantiation and setting parameters :
  selector = LHCb_MDFOnlineEvtSelectorNFiles('EventSelector')
  appMgr.ExtSvc.append(selector)
  selector.MaxNoEvents = 50000;
  selector.PrintFreq = 10000
  selector.HistogramFile = ""
  selector.EvtsForHist = 30000
  selector.SaveHistoDir = "./HLT2/"

Next follows a typical python script running the poller and producing the same histograms as *HltMassMonitor* program.The histograms will be saved in the directory specified in the script.The following procedure should be executed to run the script (provided the following packages are inside *VanDerMeer v6r1*:*OnlineFileSelector, Monitor/HltMonitor, Monitor/CommonMonitor*):

- run the *SetupVanDerMeer_v6r1.sh* script, which is inside *Monitor/CommonMonitor/job/*
- compile the *OnlineFileSelector* package by running *cmt make* in the *cmt* directory of the package *OnlineFileSelector*
- execute (*source*) the setup script in the *cmt* directory of *HltMonitor*
- compile the *HltMonitor* by running *cmt make* in its *cmt* directory
- run *gaudirun.py testPoller.py* inside *HltMonitor/scripts/*

Note: Make sure to compile and instantiate the poller and the corresponding event selector : FilePoller/MDFEvtSelector or FilePollerNFiles/MDFEvtSelectorNFiles.

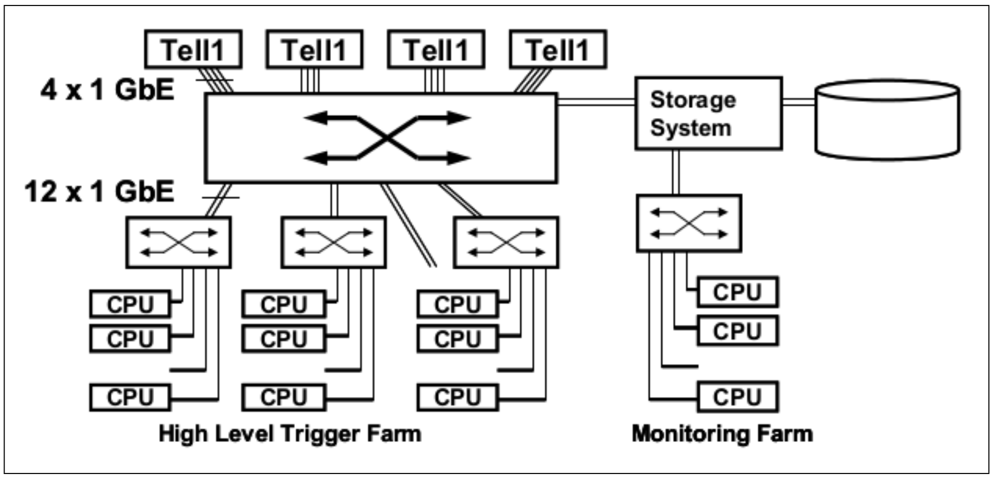
Figure 2: *Sample script*

```
#!/usr/bin/env python

from GaudiKernel.ProcessJobOptions import PrintOff, InstallRootLoggingHandler
import logging
InstallRootLoggingHandler(level = logging.CRITICAL)
from Gaudi.Configuration import *
import GaudiKernel.ProcessJobOptions
import Gaudi.Configuration as Gaudi
import Configurables as Configs
import OnlineEnv as Online
import os

from Configurables import ( LHCbApp,
                        ApplicationMgr,
                                LHCb__MDFOnlineEvtSelector,
                                LHCb__MDFOnlineEvtSelectorNFiles,
                                LHCb__FilePoller,
                                LHCb__FilePollerNFiles,
                                LHCb__RawDataCnvSvc,
                                GaudiSequencer,
                        HltMassMonitor,
                        HltRoutingBitsFilter,
                        HltSelReportsDecoder,
                        HltDecReportsDecoder,
                                HltVertexReportsDecoder,
                        HltLumiSummaryDecoder,
                                HltMonitor
                               )

app = LHCbApp()
app.Persistency="RAW"
app.EvtMax = -1

#Start ApplicationManager
appMgr = ApplicationMgr()
appMgr.EvtMax = -1
appMgr.HistogramPersistency = 'ROOT'
appMgr.SvcOptMapping.append('LHCb::FmcMessageSvc/MessageSvc')

EventPersistencySvc().CnvServices.append( LHCb__RawDataCnvSvc('RawDataCnvSvc') )
HistogramPersistencySvc().OutputFile = ''
HistogramPersistencySvc().Warnings = False
ApplicationMgr().HistogramPersistency = "ROOT"

HistogramPersistencySvc().OutputFile = ""#"testPol.root"

appMgr.TopAlg.append('StoreExplorerAlg')
StoreExplorerAlg.Load = 1
StoreExplorerAlg.PrintFreq = 100
StoreExplorerAlg.OutputLevel = 1;
```

Figure 3: *Sample script*

```
data = Online.evtDataSvc()
data.RootCLID = 1
data.ForceLeaves = 1
data.EnableFaultHandler = True

#Invoke poller
#poller = LHCb__FilePoller('Poller')
poller = LHCb__FilePollerNFiles('Poller')
appMgr.ExtSvc.append(poller)
poller.scanDirectory = "/afs/cern.ch/user/i/ichalkia/TEST"
 #"/home/ichalkia/2012" #"/daqarea/lhcb/data/2014/RAW/FULL/LHCb1/TEST"
poller.MinimumFileNum = 2
poller.alarmTime = 3
poller.DbName = "./OnlineFileProcessing.db"

#Invoke EventSelector
##selector = LHCb__MDFOnlineEvtSelector('EventSelector')
selector = LHCb__MDFOnlineEvtSelectorNFiles('EventSelector')
appMgr.ExtSvc.append(selector)
selector.MaxNoEvents = 150000;
selector.PrintFreq = 10000
selector.HistogramFile = ""#"testPollerPr.root"
#selector.EvtsForHist = 30000
selector.SaveHistoDir = "./HLT2/"

# The stuff to run
physFilter = HltRoutingBitsFilter( "PhysFilter", RequireMask = [ 0x0, 0x4, 0x0 ] )
dec = HltSelReportsDecoder(SourceID=2)
vdec =  HltVertexReportsDecoder()
lumidec = HltLumiSummaryDecoder()
monitor = HltMassMonitor()

monitor.Decisions  = { "Jpsi"         : "Hlt2DiMuonJPsiDecision",
                       "D0->Kpi"      : "Hlt2CharmHadD02HH_D02KPiDecision",
                       "D0->KK"       : "Hlt2CharmHadD02HH_D02KKDecision",
                       "D0->pipi"     : "Hlt2CharmHadD02HH_D02PiPiDecision",
                       "D->hhh"       : "Hlt2CharmHadD2HHHDecision",
                       "D->hhhh"      : "Hlt2CharmHadD02HHHHDecision",
                       "InclusivePhi" : "Hlt2IncPhiDecision" }
monitor.Histograms = { "Jpsi"         : [ 3005, 3051, 3141, 3186, 50 ],
                       "D0->Kpi"      : [ 1815, 1840, 1890, 1915, 50 ],
                       "D0->KK"       : [ 1815, 1840, 1890, 1915, 50 ],
                       "D0->pipi"     : [ 1815, 1840, 1890, 1915, 50 ],
                       "D->hhh"       : [ 1815, 1840, 1890, 1915, 50 ],
                       "D->hhhh"      : [ 1815, 1840, 1890, 1915, 50 ],
                       "InclusivePhi" : [  990, 1005, 1035, 1050, 30 ] }

# Top level sequence
topSeq = GaudiSequencer( "TopSequence" )
topSeq.Members = [ dec,  vdec, lumidec, monitor ]
appMgr.TopAlg = [ topSeq ]
appMgr.OutputLevel = 3;
```

Figure 4: *The hardware layout.Tell1 are the readout boards.* [1]



7 HLT monitoring

A brief introduction to the monitoring process and a description of the monitoring programs is given in this section. The main monitoring software is in *HltMonitor.cpp* (for HLT 1) and *HltMassMonitor.cpp* (for HLT 2).

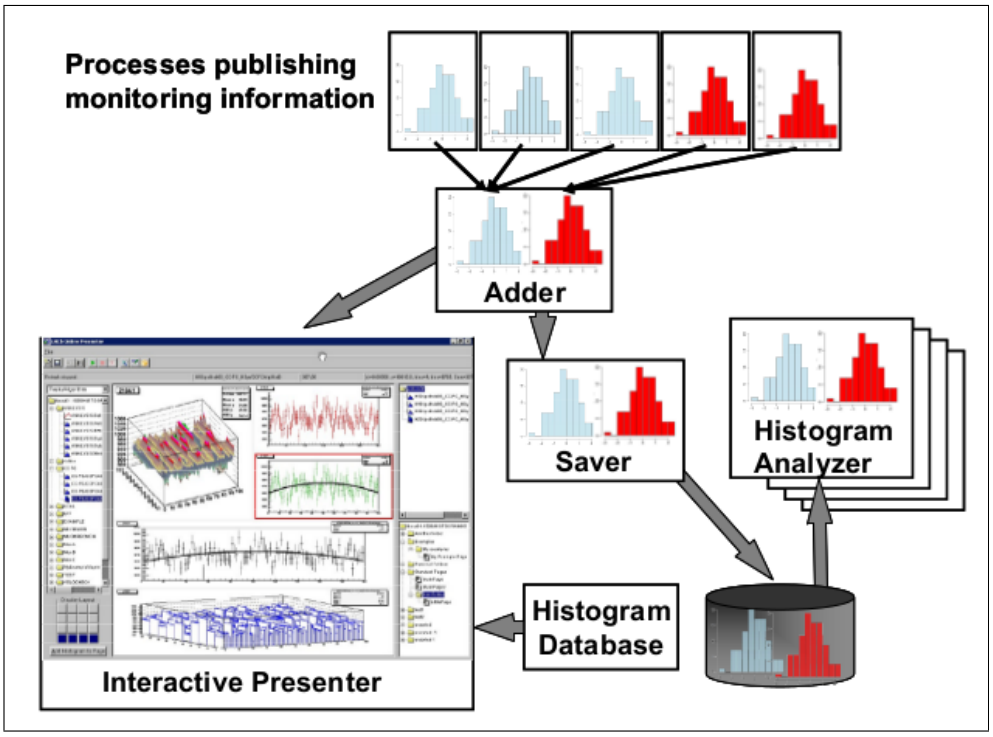
7.1 Online data monitoring [1]

The front-end readout electronics of the detector send the data collected during the collisions to the HLT farm nodes, where it is decided whether the event will be accepted.Events that are accepted from the HLT algorithms are sent to the storage and the GRID for further analysis.A fraction of these events is directed to the monitoring farm, which executes tasks which monitor the detector response in order to ensure data integrity and to detect malfunctioning components early (*see fig.4*).

Each task in the monitoring farm creates its own set of monitoring information.This implies that all data are distributed over all tasks and the monitoring information is not appropriate for analysis; first it must be collected and summed (*see fig.5*).

Dedicated tasks (*Adders*) sum the corresponding items published by a set of monitoring tasks and then they publish the summed information. Monitoring information is pushed to these tasks at regular intervals to obtain coherent snapshots of the published information. Such snapshots are created by the monitoring applications at regular intervals during the data taking activity and at a run change.In the event of irregularities an alarm is raised and displayed by the experiment control system. A program called the Presenter allows the user to view and process the summed monitoring information.

Figure 5: *The flow of the monitoring information.* [1]



7.2 HLT1 monitoring

7.2.1 HltMonitor.cpp

7.2.2 Introduction

The program flow follows the basic pattern of execution inside *Gaudi* : initialization (*initialize()*), execution (*execute()*) and finalization (*finalize()*). The execution method invokes all the actual monitoring methods.

First, some key objects present in the monitoring code will be presented and the description of the program flow will follow.

The first part of the documentation refers to the source file *HltMonitor.cpp* found in *VanDerMeer_v6r1/Monitor/HltMonitor/src*.

7.2.3 RawEvent,routing bits and other key objects in monitoring

The *RawEvent* can be considered as a persistent form of event storage containing packed and optimised data as close as possible to the detector's output. These packed data are called *RawBanks* and are of many different types depending on the subdetector [4].One of these banks is the ODIN RawBank,which contains information about the identity, the source and the quality of an event [5].

The *routing bits* can be read off a *RawEvent* object and are bits set by the Trigger according to what should happen to the event after the processing in the event filter farm [6].

The class *LHCb::HltDecReports* is the container of the HLT Trigger Decision Reports. It contains the Trigger Configuration Key (TCK) used for the configuration, the Decision Reports keyed by the trigger decision name and methods for manipulating the container [7].The *HltDecReports* are persisted as a RawBank in RawEvent.

7.2.4 Helper methods and member variables

- *monitorRoutingBit(int bit)* : this method takes as argument a bit and checks whether it has fired or not.First it acquires a RawEvent object from which it will then extract a vector containing all the bits that have fired, which will be examined for the specific bit we want to check. The raw event is acquired from the list of possible locations of the RawEvent object in the transient store, which are "pRec/DAQEvent" and "DAQ/RawEvent" unless otherwise given in the options.

- *bookHlt1Histos(),bookHlt2Histos()* : these methods first identify if there are any HLT1 error histogramms to be saved (based on the decision reports) and if yes, they book them. Note that only the histogram booking is done here, the filling is done inside the monitoring methods.

- *monitorErrors()* :This method fills the error histogramms of HLT 1 and 2.It starts by checking for the decision reports.If they are found and acquired (*if (decReports!=0)*), the methods for booking the HLT 1 and 2 histograms are called.When the histogram booking is over, we loop over the HLT lines filling the histogramms. The filling is done with the *fill()* method of the *AIDA::IHistogram1D* interface, which takes as argument a value and the corresponding weight. [8].

- *handle()* :This method is used during initialization to initialize certain variables as if there was a run change or the beginning of a new run.Normally the incident service (*IncidentSvc*) will notify the monitoring program about the run change.

- The method *ODINgpsTime()* returns the time in minutes since 1970. It is received via the Beam Synchronous Timing system to allow correlating the physics events to slow control events [5].

- *updateCondition()* :used to update the conditions from the Conditions database.

- *m_nbofcollbunches* :the number of colliding bunches, if it is greater than 0.It is used for the monitoring of the filling scheme.

- *m_startEvent* :the time at which an event took place.

- *m_startClock* :start measuring the time at the beginning of each new run.

- *timeset* :shows whether or not the clock has started, that is whether the variable *m_startClock* has been initialized.It is used for the monitoring of the filling scheme.

7.2.5 Initialization,Execution and Finalization

- The *initialize()* method begins by checking whether or not we run in "PassThrough", that is without HLT decision reports (*if (m_hlttype.substr(0,11)=="PassThrough")*). If we do, the method returns *StatusCode::SUCCESS* without further action, otherwise we proceed with the initialization of the algorithm by calling the corresponding method of the base class (*GaudiHistoAlg::initialize()*).

  Then we retrieve the Trending tool which is used to store time dependent information in an efficient way, so that it can be retrieved and displayed by the Presenter ([9]).The file name (*trendname*) has to be specified using the *partition* member variable which is retrieved from the JobOptions file (handled by the Application Manager, see [3]).

  Next, it is checked whether or not the trigger decisions are of the same number as the histograms.If not, an exception is thrown and *StatusCode::FAILURE* is returned. If yes, we proceed to build the wrapper(7.3) for each histogram, and the tags that are needed by the Trending tool.After having built the wrappers, the histograms are booked and the underlying ROOT object of each histogram is extracted from the AIDA pointer.

  Following the histogram booking is the registration to the *IncidentSvc*, so that the monitoring program is notified everytime one of the following events occurs : start of an Event, start of a Run or Run Change. Using the *handle()* method the member variables *m_startEvent,m_startClock* and *m_currentTime* are initialized as if there was a "Run Change" or "Begin Run" event.

  Finally, the conditions are updated from the conditions database (*CondDB*) and the histogram file that was created is opened for writing.

- The *execute()* method is where the monitoring methods are called. Again, if we run in "Pass through" mode (*if (m_hlttype.substr(0,11)=="PassThrough")*) then the method returns successfully without calling any monitoring method.Next it proceeds to call the monitoring methods *LumiSeenByJob()* and *monitorMu()*. At this point if the event is lumi-exclusive the *execute()* method will return successfully, otherwise it will move on to invoke the rest of the monitoring methods.

- The *finalize()* method is called when the execution of the algorithm ends.It starts by deleting the histogram wrappers and then it closes the trending file containing the histograms.Finally, the inherited from the base class *finalise()* method is called to finalize the execution of the algorithm.

7.2.6 Monitoring functions

- *monitorVertices()* :Provided there are available vertex reports this method fills the monitoring histograms for the vertices, as well as the corresponding profile histograms.Profile histograms are used to display the mean value of the Y axis and its error for each bin in the X axis.

- *monitorMasses()* :Provided there are HLT selection reports available, this method fills the mass plots through each histogram wrapper.
- *MonitorMu()* :This method begins by acquiring the ODIN RawBank from which we will check the source of the trigger and the bunch crossing type (*if ((odin->triggerType() == LHCb::ODIN::LumiTrigger) (odin->bunchCrossingType() == LHCb::ODIN::BeamCrossing))* respectively).If the trigger type is random trigger ([10]) and the bunch crossing type (*BXtype*) is beam crossing then we proceed to find the fraction of empty events, which are equally important to calculate the luminosity as the selected events [11].

  If the lumi counter *m_Counter* has the value corresponding to a random method (value == 21) we proceed to examine the *HltLumiSummary* ([12]).

  After checking that an *HltLumiSummary* RawBank exists, the method proceeds to acquire it and extract the information stored in it (*hltLumiSummary->extraInfo()*, see [13]).Based on this information, the method fills the *m_lumi_rate* or the *m_lumi_ee_rate* histograms.The execution concludes with the filling of the *m_mu_rate_root* histogram.
- *LumiSeenByJob* :This method follows the same execution pattern as *MonitorMu()*, but fills the *m_total_lumi_rate* histogram.

7.3 HistoWrapper.cpp

7.3.1 Introduction

HistoWrapper is a simple wrapper class which contains information about histograms.

7.3.2 Methods

- *HistoWrapper()* :The default constructor initializes the variables and books the following histograms for each measured quantity :*invariant mass,left,center,right,tmp,rate per* 10^6*ppinteractions*, $massvsSPDhits$, $signaloverbg$, $chi2perdof$, $chi2perdof\_bg$, *pT*.
- *fill(const LHCb::HltSelReports* selReports, double when, AIDA::IHistogram1D* m_total_lumi_rate, double scale ,AIDA::IHistogram1D* m_mu_rate, int SPDmultiplicity)* : First we check if the variables *m_massDef* and *m_rateDef* contain all the necessary information (have the correct size - *if (m_massDef.size() != 5* || *m_rateDef.size() != 3)*.Next, we check if the trigger decision name (*m_decision*) is present in the container *selReports* (*if (!selReports->hasSelectionName(decision()))*). If it is not, the method returns, otherwise it proceeds to acquire a pointer to the HLT selection report for the given trigger decision name : *selReport = selReports->selReport(decision())*.Then, we loop over all the objects on which the selection report was built and fill in the previously booked histograms.([14]).
- *getBinContent()* :gets the content of the previous bin.

7.4 ErrorHistoWrapper.cpp

7.4.1 Introduction

ErrorHistoWrapper is a simple wrapper class which contains information about error histograms.It is based on the previously discussed *HistoWrapper* class.

7.4.2 Methods

- *ErrorHistoWrapper()* :The constructor takes the necessary steps to initialize the HLT2 error histograms (book them, set axes, labels).
- *fill()* :It fills the error histograms for each of the HLT2 lines.

7.5 HltTupleMaker.cpp

7.5.1 Introduction

HltTupleMaker is a simple algorithm which puts some event info,Hlt2 candidate masses and occupancies in an ntuple. As a Gaudi algorithm it follows the usual execution pattern : initialize, execute, finalize.

7.5.2 Methods

- *initialize()* :The Algorithm subscribes itself to the *IIncidentSvc*, to be notified of a run change or the beginning of a new run.Next we fill the *m_occupancies* container with the required occupancy methods and acquire the *OTRawBankDecoder* tool.The occupancy methods are : *OTOccupancy()*, which returns the total number of hits in the Outer Tracker without decoding the modules.This is useful in pattern recognition to remove the full events; *ITClusters()*,which returns the size of the Silicon Tracker LiteClusters; *VeloClusters*, which return the size of the VELO LiteClusters and *SPDMultiplicity()*, which extracts information from the Level-0 decision unit report (*L0DUReport*) .
- *execute()* :The *execute* method retrieves event information from the ODIN raw bank and then continues to acquire the HLT decision and selection reports.
- *finalize()* :In this method open files are closed and pointers are deleted before finalising the Algorithm through the base class method (*GaudiAlgorithm::finalize()*).
- *createFile()* :It creates the tuple file we want to process. If the directory of the file is not there, we create it; if the file already exists, we open it with an *"update"* flag.
- *createTree()* :The method starts with inserting the candidate masses into the corresponding member variable (*m_masses*).Next, it continues to build the tree setting the masses and the occupancies to its branches.The way of building the tree depends

on whether the tree exists or not; in the first case we re-point the branches to their new contents (see [15]).

- *fillTree()* :This method fills in the tree with the required information :masses, acquired from selection reports, occupancies and routing bits.

7.6 HltMuonAsymMonitor.cpp

7.6.1 Introduction

The execution flow of the *HltMuonAsymMonitor* follows the same pattern as every Gaudi Algorithm : *initialize(), execute()* and *finalize()*.

7.6.2 Methods

.

- *initialize()* :The nethod starts by initializing the Algorithm using the base class initialization method.It then proceeds by booking the histograms,filling the Look-up Tables (LUT) and checking to make sure that the LUT will be filled and saved successfully.

- *execute()* :First we try to retrieve the L0DU report, the L0Muon candidates and the selection reports.To acquire the selection reports we have to have set the *m_decision* member variable in the options.Finally, we retrieve the corresponding report for the decision, get the run number from the ODIN raw bank and call the monitoring method *monitorAsym()* with the L0Muon candidates and the selection report as arguments.

- *finalize()* :In the finalization, we save the LUT and finalize the Algorithm using the base class method.

- *monitorAsym()* :In this method first we acquire the first HLT candidate and fill the J/Psi mass histogram.Then we loop over its daughters and their tracks and check for matches between muons and L0 candidates.Finally we loop over the muon pairs examining them and filling the necessary histograms.

7.6.3 Helper methods

There are special methods for handling LUTs (*openLUT,fillLut,getPTFromLUT,writeLUT*), as well as for getting the candidate M1 and M2 pads (*getCandidatePads*) and the time (*getTime()*).

7.7 HLT2 monitoring

7.7.1 HltMassMonitor.cpp

7.7.2 Introduction

The execution of the HLT 2 monitoring program is very similar to the corresponding for HLT 1.It follows the usual pattern: *initialization,execution,finalization.*

7.7.3 Methods

- *initialize()* :same as for HLT1 monitor, see 7.2.5.
- *execute()* :same as for HLT1 monitor (see 7.2.5), but now we just monitor the masses and Mu (not errors or vertices).
- *finalize()* :clears the wrappers and finalizes the Algorithm (see 7.2.5).

7.7.4 Monitoring methods

Most of the monitoring functions used in HLT 2 are the same as HLT 1 except for a few additions (7.2.6).

- *monitorMasses()* :Add the signal-to-background ratio to the histogram as well (apart from rate).
- *monitorMu()* :Same as HLT 1, but the content of one more bin is added to the *m_mu_rate_root* histogram (*m_trend->addValue(std::string("Mu") , m_mu_rate_root->GetBinContent(now-1))*).

References

[1] O. Callot *et al.*, *Online data monitoring in the LHCb experiment*, J. Phys. : Conf. Ser. **119** (2008) 022015.

[2] C. Gaspar, M. Dnszelmann, and P. Charpentier, *Dim, a portable, light weight package for information publishing, data transfer and inter-process communication*, Computer Physics Communications **140** (2001), no. 12 102 , {CHEP2000}.

[3] P. Mato, *Gaudi-architecture design document*, tech. rep., CERN-LHCb-98-064, 1998.

[4] *Raw Event wiki*, , https://twiki.cern.ch/twiki/bin/view/LHCb/RawEvent.

[5] *Odin raw bank*, , http://lhcb-online.web.cern.ch/lhcb-online/tfc/documents/ODIN_rawdata_edms704084_1.pdf.

[6] *Routing bits*, , https://twiki.cern.ch/twiki/bin/view/LHCb/HltRoutingBits, https://twiki.cern.ch/twiki/bin/view/LHCb/HltEfficiency.

[7] *Hlt decision reports*, , http://lhcb-release-area.web.cern.ch/LHCb-release-area/DOC/davinci /releases/latest/doxygen/d7/db8/class_1_h_cb_1_1_hlt_dec _reports.html#details.

[8] *Aida::IHistogram1D*, , http://aida.freehep.org/doc/v3.2.0/api/hep/aida/IHistogram1D.html.

[9] *Trending tool*, , https://lbtwiki.cern.ch/bin/view/Operation/UseTrending.

[10] *Odin fields*, , http://svn.cern.ch/guest/lhcb/LHCb/trunk/Kernel/LHCbAlgs /src/ODINCodecBaseTool.cpp.

[11] *Lumi Counters*, , https://twiki.cern.ch/twiki/bin/view/LHCb/FileSummary Record#LumiCounters.

[12] *Lumi Counters XML*, , https://svnweb.cern.ch/trac/lhcb/browser/LHCb/trunk/Event /LumiEvent/xml/LumiCounters.xml?rev=168574.

[13] *Lumi Summary XML*, , https://svnweb.cern.ch/trac/lhcb/browser/LHCb/trunk/Event /HltEvent/xml/HltLumiSummary.xml?rev=36838.

[14] *HltObjectSummary*, , https://svnweb.cern.ch/trac/lhcb/browser/LHCb/trunk/Event/HltEvent /xml/HltObjectSummary.xml?rev=36787.

[15] *TTree*, , http://root.cern.ch/root/html/TBranchElement.html.