The Monitoring Program for the ATLAS Tile Calorimeter

P. Adragna, A. Dotti, C. Roda

***Abstract*— The Tile Calorimeter is a steel-scintillator device which constitutes the central hadronic section of the ATLAS Calorimeter. About 12% of the Tile Calorimeter has been exposed to test beams, at CERN, in order to determine the electromagnetic scale and to evaluate its uniformity. During the 2003 calibration periods, an online monitoring program has been developed to ease the setup of correct running condition and the assessment of data quality. The program, developed in C++ and based on the ROOT framework, has been built using the Online Software Service provided by the ATLAS Online Software group and is equipped with a graphical user interface. This application has been intensively used throughout the calibration sessions in 2003 and also during the setup and test with cosmic rays carried out, at the beginning of 2004, on a few modules of TileCal during the pre-assembly of the extended barrel. After a brief overview of ATLAS DAQ and Services, the architecture and features of the Tile Calorimeter monitoring program are described in detail. Performances and successive integrations are discussed in the last part of the paper.**

***Index Terms*— ATLAS, Tile Calorimeter, hadronic calorimeter, online monitoring program, test beam**

I. Introduction

THE monitoring program described in this paper has been developed in the framework of the calibration periods with beams carried out on the ATLAS Tile Calorimeter [1] along the CERN H8 SPS line. The ATLAS calorimetric system is a composite detector, which exploits different techniques inside different rapidity regions to optimize the calorimeter performances while maintaining a high enough radiation resistance. The Tile sampling calorimeter (TileCal) is the central hadronic section of this system. It is composed by one central barrel and two lateral (extended) barrels consisting of 64 wedges each. Every wedge consists of three longitudinal samplings of steel plates and scintillating tiles. Two edges of the tiles are air coupled to wavelength shifting fibers, which collect the scintillating light and bring it to photomultiplier tubes. The signal generated by each tube is shaped and amplified. The amplified signal is then sampled and digitised. The final output of the overall procedure are 9 signal samples for each photomultiplier. During the three-year-long calibration program, about 12% of TileCal has been exposed to test beams.

A picture of the setup exploited during the test performed along the H8 beam line is shown in fig. 1: four calorimeter wedges were placed on a movable table and exposed to beam. The bottom module (referred as *Module 0*) is the prototype barrel module and has been a reference throughout all data taking periods. On top of Module 0, one barrel module and two extended barrel modules are stacked; the extended barrel wedges are placed one next to the other. This configuration assure a complete containment of the lateral shower produced by pions interacting inside the central barrel module. The movable table allows the beam to impinge on the modules in any point of the lateral side or of the inner edge of the wedge, as it will happen with particles produced by beam interactions in ATLAS.

The monitoring program here described has been developed for the 2003 test beam period and has been in operation until the spring of 2004. This application, based on ROOT [2] and on the software developed by the ATLAS Online Software group [3], monitors both Tile Calorimeter modules and detectors equipping the beam line. The aim of the monitoring program was to allow people in the control room to verify the functionalities of the employed instrumentation, both from the *operational* (through a simple inspection of system work) and from the *physical* (through a cross check of acquired data) point of view. To achieve these objectives, sampled data from beam detectors and calorimeter modules are decoded and summarized into histograms, to be displayed in a simple way.

P. Adragna, A. Dotti, C. Roda are with Università di Pisa and INFN Sezione di Pisa, Via Buonarroti 2, Edifi cio C, 56127, Pisa, Italy.

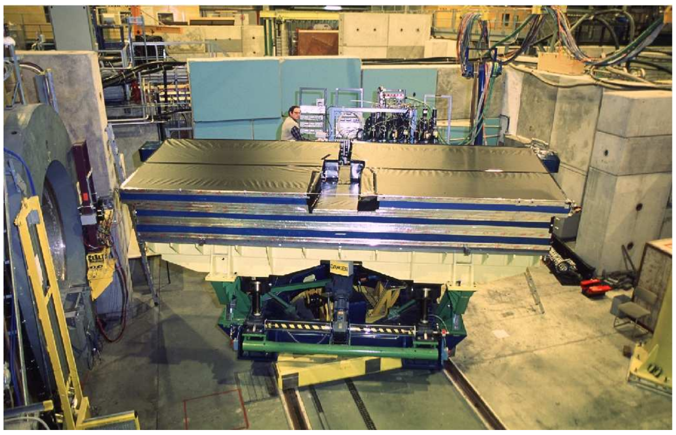


Fig. 1. Tile Calorimeter setup during standalone test beam. The beam impinges on the modules from the left side.

A. Simplified Architecture of the Acquisition System

A simplified outline of the data flow through the ATLAS DAQ chain is shown in fig. 2 [4].

Detector readout electronics produces digitised data. If the Level 1 Trigger accepts them, they enter the *Read Out Driver* (ROD), first stage of the data flow chain. At this stage simple calculations are performed on the data inside the ROD (for example calibration constants are applied); then the data are formatted with an header and a trailer enclosing additional information (source identifier, trigger type, ...). This block of data is moved to the *Read Out Buffer* (ROB), where it is buffered while the Level 2 Trigger decides if data are acceptable. In this case, the *Read Out System* (ROS) sends the data fragment to the Event Builder, which assembles a complete event. The last stage in the data flow is the Event Filter, where an event is reconstructed and a final selection is applied. If the Event Filter selection is satisfied, the event is recorded on disk.

B. The Event Monitoring Service

The ATLAS Online Software provides a number of *services* which can be used to build a monitoring program [4]. Their main task is to carry requests about monitoring data (e.g. request of event blocks, request of histograms...) from monitoring destinations to monitoring sources and then the actual monitored data (e.g. event blocks, histograms...) back from sources to destinations.

Our application exploits the Event Monitoring Service (EMS), which is responsible for the transportation of physics events or event fragments, sampled from well-defined points in the data flow chain.

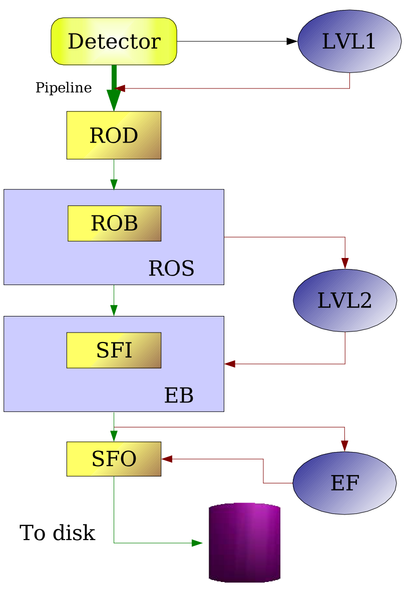


Fig. 2. A simplified outline of the ATLAS Acquisition System. Only one ROD/ROB/ROS is shown. Both ROS and EB can receive multiple inputs.

It is necessary to implement two interfaces with the Event monitoring Service to gain physical access to sampled data: the **EventSampler** and the **EventReceiver**. The first interface samples the events from a certain point of the data flow, the second one accounts for handling the events delivered by the EMS. An event sampler is a separate process: one sampler must be up and running in every node where events are requested.

II. Requirements

Our program fulfils the following requirements:

1) the program is able to retrieve information produced by every detector plugged into the Data Acquisition System;
2) his configuration is fully transparent to the end user;
3) his execution does not overload the Data Acquisition System;
4) the program provides the intermediate user with some basic function while the expert can have the possibility to extend the application's functionalities;
5) the application is easily extendable, to monitor different detectors;
6) the information collected is represented through a graphical user interface.

III. Program Architecture

The TileCal Monitoring Program [5] is an object oriented application developed in C++. It is based on the ROOT framework: in particular the data storage, the graphical user interface and the event handling fully exploit ROOT classes and methods. The program has been developed under the Linux operating system for the i686 architecture. Our program workflow is illustrated in the chart of fig. 3.

Input data can be either a raw data file, saved on disk, or real time sampled events provided by an event sampler. In both cases data are expected to be in the standard ATLAS format [6].

Events are copied inside a buffer, where they are unpacked and interpreted. Event unpacking proceeds through detector independent methods up to the localization of the Read Out Driver (ROD)

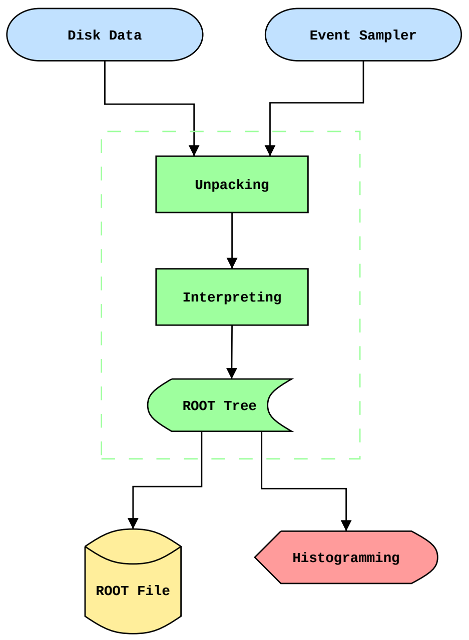


Fig. 3. Flow chart of the TileCal monitoring program execution.

fragments. Detector dependent methods are implemented to extract the data inside this fragment. The events, produced during all the tests, contains both Tile Calorimeter and beam detectors (Cerenkov counters, scintillator counters and wire chambers) data. All relevant information is extracted, event by event, and stored in a ROOT Tree residing in memory, while raw data are discarded and the buffer is freed. From this point on, the analysis is performed using only the data stored inside the ROOT Tree. Histograms produced during the analysis can be immediately displayed using the presenter included with the Graphical User Interface (GUI).

The possibility of reading raw data files from disk not only greatly simplifies the debugging process, but also allows to run simple analysis tasks on just acquired data.

As shown in the dependency diagram of fig. 4, the abstract structure of the program is divided into three blocks[1]:

- ***Data Source***, responsible for retrieving an event from the selected data source; it consists of two classes, **TC_DataFile** and **TC_Consumer**;
- ***Unpacking and Decoding***, interfaced with data source through the class **TC_DataConnection**, which transfers the memory address where the event is stored; the task of this block is extracting and interpreting the requested fragments;
- ***Presenter***, responsible for displaying the histograms produced by the class **TC_DataBase**, organizing them inside multiple windows; this block is also responsible for handling user requests.

The classes in the central part of the diagram perform management tasks and serve as a connection among the different parts of the program. The main class is **TC_MainFrame**. The class **TC_DataConnection** is responsible for the transmission of information among the various blocks and **TC_MainFrame**. **TC_DataBase** contains all the produced histograms. Finally, an istance of TTree (shown in fig. 4 with the name *ROOT Tree*) is the storage for the information extracted from the events.

[1]All our classes are recognizeable from the TC prefix (omitted in fig. 4) from the name of the adronic calorimeter (TileCal).

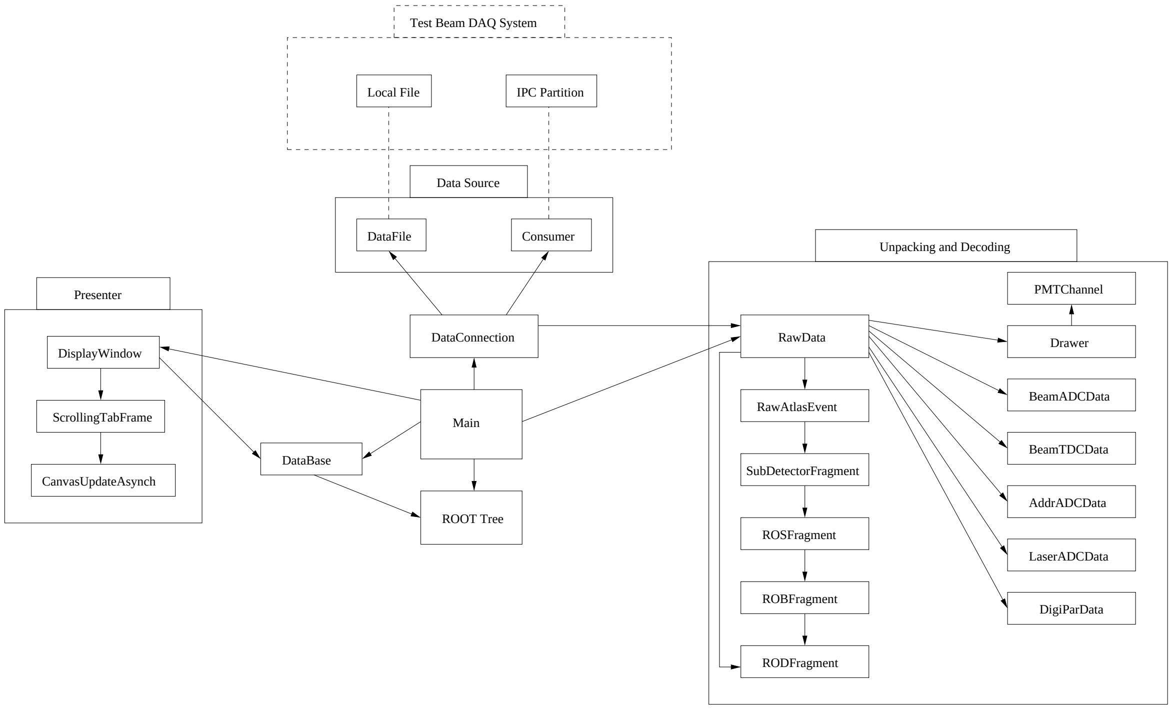


Fig. 4. Dependency diagram for the classes composing the monitoring program. A label, indicating the general role of a particular group of classes, identifies each block.

IV. The Interface with the Event Monitoring Service

The main purpose of our application is to examine raw data produced by the Acquisition System in real time; these data are sampled from the data flow [4]. The **EventReceiver** interface implementation is the class **TC_Consumer**. **TC_Consumer** traces the events through an iterator, making them available.

After the creation of an istance of **IPCPartition**, which is necessary to connect and to interact with the Inter Process Communication system [7], an object of class **Monitoring** [8] is created. This object is the agent between our program and the monitoring service. The operation is completed through a choice of suitable event selection criteria, of the address where data are sampled[2] and of the iterator type. These components uniquely determine the interface status (represented by the class **Monitoring::Status**). At this point it is possible to start examining the events. For example, the user can ask for events produced by a single detector section by defining the appropriate ROD Crate's sampling address. On the other hand, he can ask for a completely formatted event: in this case he must submit the Event Builder address.

V. Reading Data from Disk

As previously discussed, besides retrieving online sampled events it is possible to read raw data file from disk. The dedicated class is **TC_DataFile**.

It exploits the **EventStorage** library [9], provided by ATLAS DataFlow group, to read bytestream raw data from disk. This happens through an istance of the class **DataReader**.

The function **TC_DataFile::NewEvent** retrieves the events, identifying the correct region of memory through **DataReader::getData**. At the end, the program can access a complete event with a pointer to his first 32 bit word.

[2]In a distributed system different information providers can coexist, each with his specific address. Inside ATLAS DAQ systen, the address is composed by three fields: the name of the detector, the name of the dedicated electronics and the name of a particular electronic module.

VI. Data Unpacking

Each event is structured according to the ATLAS event format: the detector data are encapsulated into 5 layers of headers, all having similar structure. In order to easily reach these data, five classes have been developed, one for each hierachy level:

- **TC_RawAtlasEvent**;
- **TC_SubDetectorFragment**;
- **TC_ROSFragment**;
- **TC_ROBFragment**;
- **TC_RODFragment**.

The algorithms of the mentioned classes allow to identify the corresponding blocks of data and their fields. These classes are implemented in such a way that a call to the method **TC_RawAtlasEvent::ReadFromMem** triggers the complete event decoding, calling in cascade the corresponding **ReadFromMem** function of all the nested Fragment classes, up to the ROD level.

At the end of the unpacking chain, all the blocks are mapped inside a tree structure[3], as depicted in fig. 5. Among the data fields of each class there is a vector of pointers to the blocks of the lower level; for example, **TC_RawAtlasEvent** has a member **SubDetectorFragment**, a vector of pointers to **TC_SubDetectorFragment** istances.

The unpacking methods up to the ROD fragment are general and do not contain any reference to the particular detector that created the data.

VII. Detector Data Interpretation

Once the unpacking chain is succesfully executed, pointers to all ROD fragments are stored in a vector. Detector specific data are then decoded by the function **TC_RawData::Decode**, which determines the correct algorithm for every subsystem on the basis of the ROD identifier value. Every ROD, and consequently every data block type,

[3]More than one SubDetector, ROS and ROB fragments may be nested inside an event.

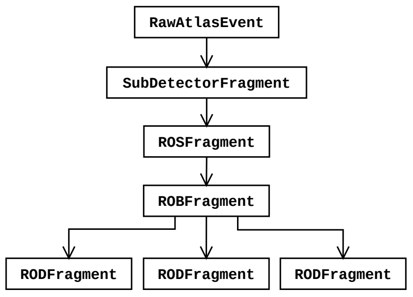


Fig. 5. Tree structure of unpacked data blocks.

is identified by a unique number, the *source identifi er*, member of the class **TC_RODFragment**.

The class that traces Tile Calorimeter information is **TC_Drawer**. It contains methods that allow to calculate, for each electronic channel, the amplitude of the 9 time samples of the signal, and correlated quantities such as the instant of the signal peak or the estimate of the corresponding energy deposit.

VIII. The ROOT Tree and the Integration

The possibility of easily extending our application to monitor different detectors is one of the fulfilled requirements. For this purpose, it was necessary to plan the data decoding and the storage with the largest possible flexibility.

Since the decoding phase is detector-dependent, it is natural to create appropriate functions to extract ROD blocks from an event. The data contained in each ROD block are interpreted and temporarily stored, to be used in the analysis phase. The implementation requires a class for the decoding and a structure to hold the reconstructed data. While for the first step some expertise is necessary, for the second step we found a simple solution, exploiting the ROOT Tree [10].

Using the ROOT Tree as an interface between different parts of our program, and not as a simple mass storage, is one the most important feature of the program we developed. Thanks to this structure, a new detector is integrated inside the program writing a class, with decoding functions and member data to store temporaly physical quantities, and creating a branch with the indication of the object type; whenever the method **TTree::Fill** is called, the Tree is filled.

The Tree is created on the basis of the event structure. Since this is not known in detail before the start of a run (i. e. the number of modules in a detector can be different from run to run) Branches and Leaves [10] are automatically built following the structure of the first event identified.

If the Tree is used only as an interface, it is possible to reset it as soon as the interesting data have been analysed. For example, the user is usually not interested in cumulating the events inside the structure. In this case it is possible to erase Tree's content after having filled the suitable histograms. However, the user can always choose to cumulate the events inside the Tree and to save the entire Tree on a file at the end of the run, in order to carry a more in-depth analysis on.

If the events are accumulated, the contents of the structure can be examined through a graphical interface provided by ROOT (**Tree-Viewer**), while the data acquisition is still in progress. The interface allows straight away booking and filling of histograms, in one or more dimensions, with every variable stored: this makes possible the study of simple correlations.

IX. The Database

Event by event, the data stored inside the ROOT Tree are used to fill all the necessary histograms. The class **TC_DataBase** manages the creation, the filling and the updating of these histograms. The class acts as a container of pointers to the single plots. The various graphs are created through the function **BookHistos** and filled, event by event, through the function **FillHistos**.

The total number of plots, as well as their type, is fixed and it is not possible to change them at run time. The only way of adding or removing the creation or the filling of a plot is acting on the source code and then recompile the modified class.

This feature did not greatly limit the performance of our application, since the experience with previous data taking allowed to know the histograms needed to monitor the TileCal performance in advance. However, in view of a more general use of our application, we would like to avoid this limitation. Therefore a *dynamical* creation and filling of histograms is currently under study.

We would like to point out that our architecture does not prevent the user from creating an histogram *on the fly*: when the data are organized inside a ROOT tree, it is possible to create an histogram using the **TTree::Draw** method. However, a plot created with **TTree::Draw** cannot be organized inside our graphical interface but it must be displayed as a separate window.

X. The Signal-Slot Mechanism

The user can interact with our program through a graphical user interface (GUI). The GUI is built with ROOT graphical classes. User actions, required through various panels, are executed through the Signal-Slot Mechanism [11], [12].

The Signal-Slot Mechanism manager is the ROOT class **TQObject**. If we want a user defined class to benefit from this mechanism, it must inherit from **TQObject**. Whatever public function of whatever instance of whatever class derived from **TQObject** can be called, as a *slot*, in answering an event: it is sufficient to *connect* the signal to the slot using the method **TQObject::Connect**.

XI. The Presenter

Histograms are displayed by a simple graphical window. Different sections, containing related information, are enclosed inside the same frame and organized into tabs. Tabs allow cohexistence of different drawing surfaces inside the same window, subdividing it into graphical pages. Thanks to the double buffering tecnique, images drawn inside tabs are stored in memory also when they are not on focus, so that, when selected, their appearence is extremely fast.

The graphical section, among which histograms are divided, are:

- ***Beam***, where the transverse profile of the beam, as provided by multiwire proportional chambers, is shown (fig. 6);
- ***Counters***, which contains the histograms of signal amplitude from the scintillating counters placed on the beam line;
- ***Cerenkov***, which contains the histograms of signal amplitude from Cerenkov threshold counters;
- ***Drawer***, which contains seven histograms for the total energy deposit inside the modules exposed to the beam (fig. 7). The two barrel modules (cp. I) are divided into two parts, one laying in the region of negative pseudorapidity (indicated with N0 and N1) and one laying in the region of positive pseudorapidity (indicated with P0 and P1); the histograms N2 and P2 refer to the two extended barrel modules; the last plot shows the total charge collected inside the whole apparatus;
- ***Event Display***, which shows the energy deposited inside every cell of the three calorimeter modules. Every calorimeter module is represented by a bidimensional histogram, while every cell is depicted by a parallelepiped with the height proportional to the energy deposit; the pseudorapidity is reported on the horizontal axis. As an example, in fig. 8 is represented a 300 GeV electron,

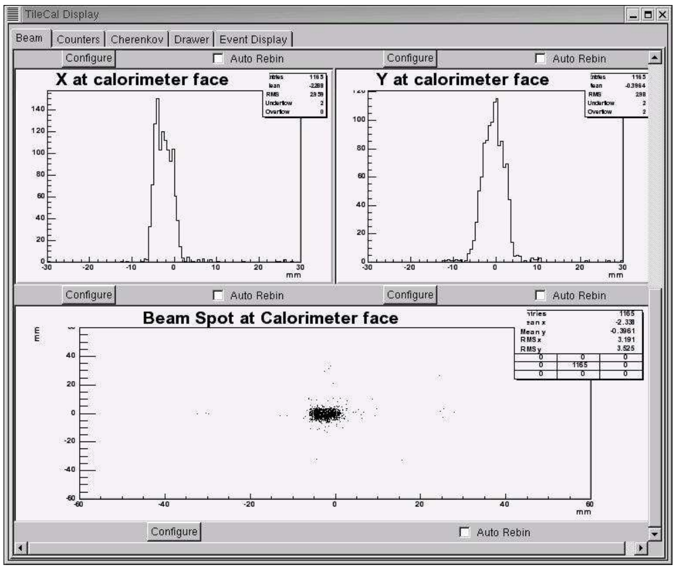


Fig. 6. Horizontal and vertical coordinates of beam impact point (top plots) and their correlation (bottom plot).

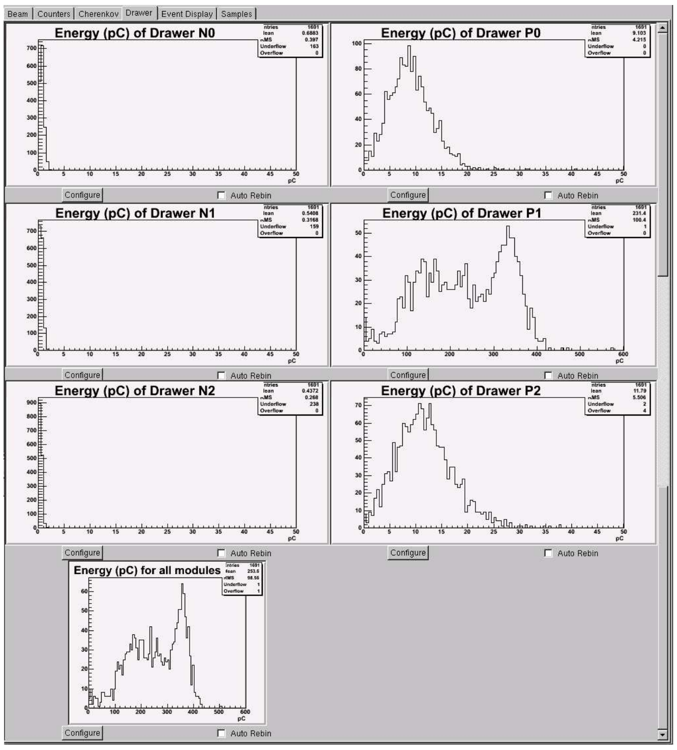


Fig. 7. Distributions of the charge collected inside the various calorimeter modules exposed to a beam impinging on the central module, at positive pseudorapidity.

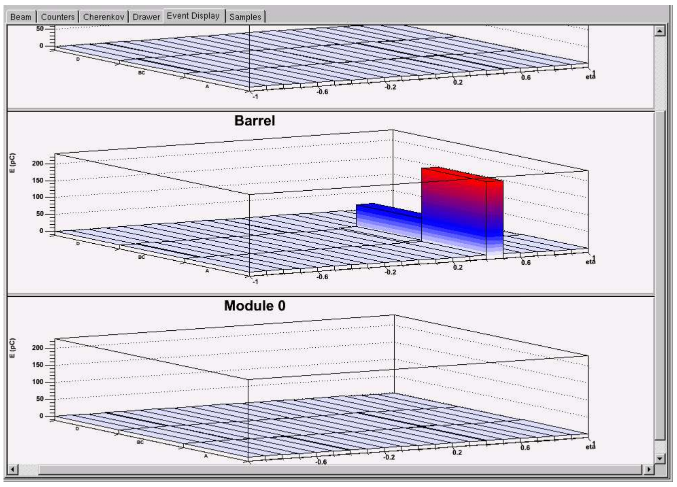


Fig. 8. Example of Event Display. The plot shows a representation of energy released inside calorimeter modules by an electron, impinging with $\eta = 0.45$ and an energy of 300 GeV, with signal production in cells A5, BC5.

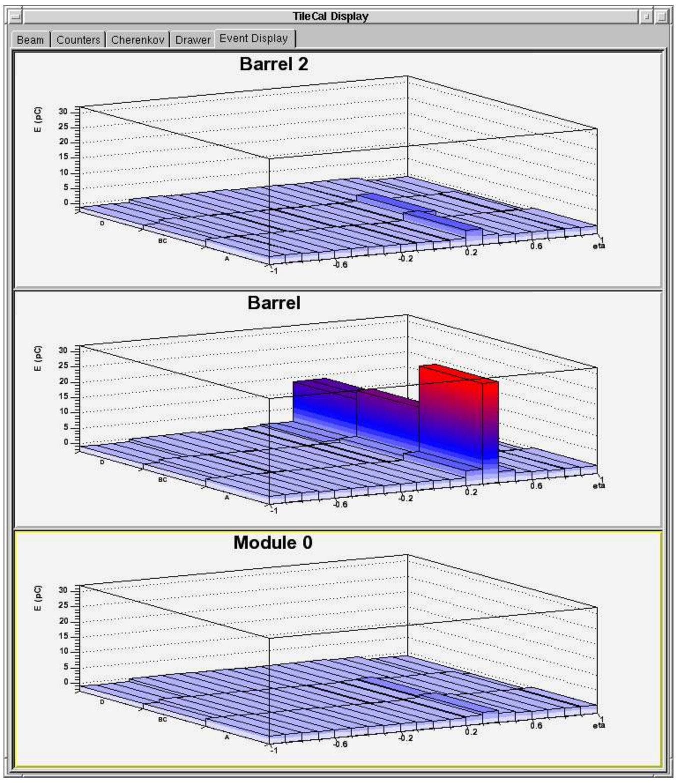


Fig. 9. Example of Event Display. The plot shows a representation of energy released inside calorimeter modules by a pion, impinging with $\eta = 0.35$, with signal production inside the cells A4, BC4, D4.

while fig. 9 shows the energy release of a pion: both particles are impinging on the central module;

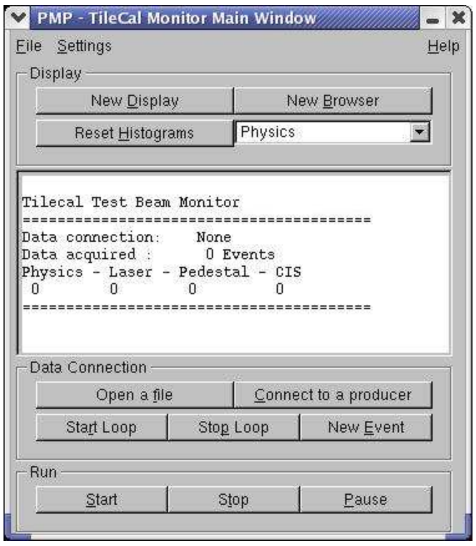
- ***Samples***, where the nine signal samples from each photomultiplier tube can be examined.

XII. Main Control

The class **TC_MainFrame** is responsible for the global administration of program functions. This class, besides drawing the main command panel (fig. 10) is also responsible for event handling and for triggering the actions connected with the events.

From the main panel the user can completely drive the application: choosing the data source, the data trigger type; opening the histogram presenter; etc. The main panel contains twelve buttons and one combo box for trigger selection. When the buttons are pushed a signal **clicked()** is emitted. Instead, the combo box emits a signal **Selected(Int_t)** when the trigger type is changed; the signal argument is an integer and identifies the required trigger type.

The signal **clicked()** is connected with the member function **HandleButton** of **TC_MainFrame**, as highlighted in fig. 11. This function determines which button sent the signal and triggers the proper actions.



```
Tilecal Test Beam Monitor
========================================
Data connection:    None
Data acquired :       0 Events
Physics - Laser - Pedestal - CIS
 0        0        0         0
========================================
```

Fig. 10. The main control panel of the program.

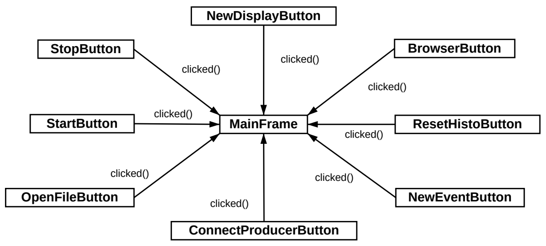


Fig. 11. Collaboration diagram for the acquisition of **clicked()** signal.

The buttons **New Display** and **New Browser** of the group *Display* open the presenter and the ROOT Class Browser, respectively. The combo box is useful to select the trigger type of the acquired events. It is possible to choose physics events (produced with particles), calibration events (laser beam or charge injection) or pedestal events.

The buttons of the group *Data Connection* allow the user to drive event retrieving. **Open a file** opens a dialog window, where it is possible to choose the raw data file; **Connect to a producer** starts the interface with the Data Acquisition System. The other buttons starts and stops the sampling action.

XIII. Real Time Update

One of the main feature of our program is the continuos updating of histograms as soon as the data are acquired. To perform this task, our program relies on signal-slot mechanism, introduced in section X. Histogram updating is not performed at every event, because this would be too much CPU time consuming, lowering the performances; the updating is performed at fixed time interval.

When the phase of decoding and histogram filling is finished, the class **TC_DataBase** emits a signal **Update** (fig. 12). The signal is catched by the class **TC_MainFrame**, which checks how much time is elapsed from the last update. If the elapsed time is greater than a certain, user-defined amount, then **TC_MainFrame** emits an **UpdateAll** signal. This signal, catched by the presenter, will trigger the updating of all the plots who are displayed on the currently selected pad. This simple but effective mechanism allows to monitor the data acquisition as it goes on. A similar mechanism is used to update histograms when, for example, the user decides to change their scale: in this last case the signal **UpdateAll** is emitted directly, so the updating is immediate.

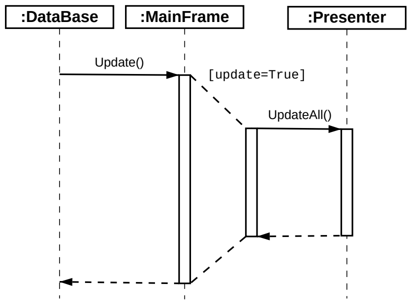


Fig. 12. Sequence diagram for the actions connected with real time updating.

XIV. Program Performance

The monitoring program has been extensively tested to check the usage of CPU and memory during the data taking at test beam. An excessive resource exploitation implies a decreasing functionality of DAQ system, since in our setup the monitoring and the acquisition tasks shares, during the test beam, the same machines.

The test has been performed on a PentiumIII-class machine at 1 GHz with 256 MB of RAM running Linux RedHat 7.3 (CERN version).

Our program can read and completely decode 250 events per seconds on average, with a mean charge of 41% on the CPU, while the mean memory usage is of 18%.

The total memory (physical plus swap) usage is shown in fig. 13 as a function of running time. The steep increase of used memory, shown on the plot around 550 seconds, represents the opening of the presenter GUI.

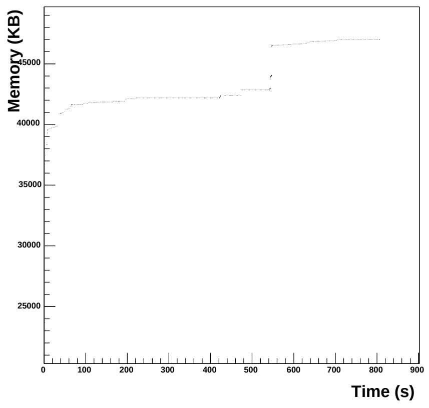


Fig. 13. Memory usage (physical plus swap) as a function of running time.

The memory usage watch has been revealed an important tool to find and correct memory leaks during the development of the program. Thanks to this kind of check we succeded in reducing remarkably the amount of physical memory used.

The memory usage is rather high but this can be understood: our application must manage, at the same time, large events, a high number of histograms and a composite graphics. On the other hand the CPU load is a little bit too much demanding. We solved the last problem through a fine tuning of process priority with the system command *nice*. In this way the program mantains a good functionality without loading excessively the acquisition system.

XV. Successive Integrations

The first upgrade to our program to monitor a larger set of detector has been tried during the combined test beam runs of September

2003. The first detectors to be monitored after TileCal were the muon spectrometer (Monitored Drift Tubes chambers, MDT) [13] and the central tracker (SemiConductor Tracker, SCT) [14].

The unpacking and interpreting of MDT and SCT were simply included using their standard data decoding routines. Adding new branches inside ROOT tree was equally simple. Last, a new graphical page was added inside the presenter to organize the new produced histograms. The availability of data from different subdetectors allowed to obtain online histograms showing the correlated information from different detectors. An example is shown in fig. 14.

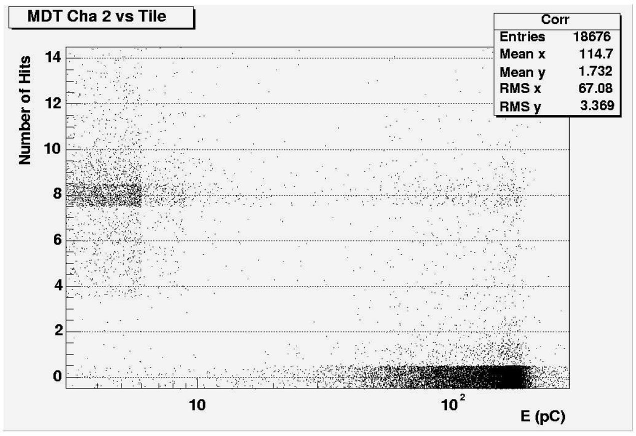


Fig. 14. Plot of the hits recorded by the first MDT chamber versus the total energy deposited inside the Tile Modules, obtained from data recorded during the Combined test beam 2003. The two populated regions (top left and bottom right) corresponds to muons and pions respectively.

The plot shows the number of hits recorded by the first MDT chamber versus the total energy deposited in the module of the hadronic Tile Calorimeter obtained from data recorded during the Combined test beam. Along the beam line, where tests are performed, the MDT chambers are separated from TileCal modules by a thick shield of cement, impenetrable for pions. A beam of pions and muon with energy of 180 GeV interacts first with the TileCal modules and then, after the cement shield, with the MDT chambers. The top left region of the plot is populated by muons, that have a low energy release inside the calorimeter and can also penetrate the shield; the bottom right region is populated by pions, that cannot pass over the shield and do not hit the chambers.

Once the data acquisition with test beams was finished, our application was further used during the set up and test with cosmic rays carried out, at the beginning of 2004, on a few modules of TileCal during the pre-assembly of the extended barrel. The three barrels composing the calorimeter are pre-assembled in surface to check the strict geometrical tolerance (a few millimiters over a few meters) and the integration with services (cables, crates, ...) and othe detectors. During the pre-assembly, a system has been set up using scintillating counters to trigger on cosmic muons traversing three of the TileCal modules. Our monitoring program has been slightly modified to account for the different electronic mapping and for the different list of histograms to be filled and displayed. The program was then used to monitor both physics and calibration data.

XVI. Conclusions

The online monitoring program we developed has succesfully run throughout the calibration sessions of 2003. It has proved to be a powerful tool, helping the shift crews to identify faulty detector states. Its modular design has permitted to easily extend its use to include, besides TileCal and beam-line detectors, modules of the vertex tracker SCT and chambers of the muon spectrometers MDT. Our application was usefully exploited during the pre-assembly of the extended barrel at the beginning of 2004, when few modules of the pre-assembled Extended Barrel of TileCal were tested with cosmic rays.

The experience gained during the tests with beams and with cosmic rays have allowed to evaluate the characteristics of our program in a realistic environment posing the bases for the design of an upgraded version of the monitoring program.

Acknowledgments

We would like to thank all the Tile Calorimeter group for their advices and useful comments as first users of our application. We are also grateful to the tecnicians who helped with the calorimeter assembly, to the Test Beam Coordinator B. Di Girolamo and to the SPS staff for allowing us to collect test beam data. Finally, we would like to thank Lorenzo Santi for giving us the code he wrote for the monitoring of the pixel detector, which was the starting point of our project, and J. E. Garcia Navarro, S. Gonzalez, R. Ferrari, W. Vandelli for helping us with the integration of SCT and MDT detectors in the monitoring program during September 2003 Combined Test Beam.

References

[1] ATLAS Collaboration. (1996). ATLAS Tile Calorimeter Technical Design Report. CERN. Genève, CH. [Online]. Available: http://atlas.web.cern.ch/Atlas/SUB_DETECTORS/TILE/TDR/TDR.html
[2] R. Brun, F. Rademakers. (1997). ROOT – An Object Oriented Data Analysis Framework. *Nucl. Inst & Meth. in Phys. Res. A 389*, pp. 81 – 86.
[3] S. Kolos et al. (2003). Online Monitoring software framework in the ATLAS experiment. CERN. Genève, CH. [Online]. Available: http://cdsweb.cern.ch/?c=ATLAS
[4] ATLAS Collaboration. (2003). ATLAS High-Level Trigger, Data Acquisition and Controls Technical Design Report. CERN. Genève, CH. [Online]. Available: http://atlas-proj-hltdaqdcs-tdr.web.cern.ch/atlas-proj-hltdaqdcs-tdr
[5] P. Adragna, A. Dotti, C. Roda. (2004). The ATLAS Tile Calorimeter Test Beam Monitoring Program. CERN. Genève, CH. [Online]. Available: http://cdsweb.cern.ch/?c=ATLAS
[6] C. Bee, D. Francis, L. Mapelli, R. McLaren, G. Mornacchi, J. Petersen, F. Wickens. (2003). The Raw Event Format in the ATLAS Trigger & DAQ. CERN. Genève, CH. [Online]. Available: http://cdsweb.cern.ch/?c=ATLAS
[7] S. Kolos. (2001). Inter Process Communication Design and Implementation. CERN. Genève, CH. [Online]. Available: http://cdsweb.cern.ch/?c=ATLAS
[8] S. Kolos. (2000). Online Monitoring User's Guide. CERN. Genève, CH. [Online]. Available: http://cdsweb.cern.ch/?c=ATLAS
[9] A. Dos Anjos. (2003). Event Format Library Analysis and Design. CERN. Genève, CH. [Online]. Available: http://cdsweb.cern.ch/?c=ATLAS
[10] S. Panacek ed. (2004), *ROOT User Guide*. [Online]. Available: http://root.cern.ch/root/doc/RootDoc.html
[11] Trolltech. (2003). Qt White Paper. Trolltech., Oslo, N. [Online]. Available: http://www.trolltech.com/products/whitepapers.html?cid=20
[12] V. Onuchin. TQObject Class in R. Brun et al. (2004) *ROOT Reference Guide*. [Online]. Available: http://root.cern.ch/root/Reference.html
[13] ATLAS Collaboration. (1998). ATLAS Muon Spectrometer Technical Design Report. CERN. Genève, CH. [Online]. Available: http://atlas.web.cern.ch/Atlas/GROUPS/MUON/TDR/Web/TDR.html
[14] ATLAS Collaboration. (1997). ATLAS Inner Detector Technical Design Report. CERN. Genève, CH. [Online]. Available: http://atlas.web.cern.ch/Atlas/GROUPS/INNER_DETECTOR/TDR