Diana User and Developer Tutorial

M. Pallavicini - M. Vignati

January 22, 2007

1 Document History

Added Implementing a Module and Events structure 22-1-2007 First preliminary and incomplete version 5-1-2007

2 Document Content

This document is a brief tutorial to the use of Diana, the reconstruction and analysis software of the Cuore experiment. The document is structured as follows: section 3 lists the system and environment settings that are required to compile and run Diana; section 4 says how to get the code from SVN repository and compile it; section 5 tells how to run Diana; section 6 is a brief and general description of the internal structure of the Diana framework and of the main software components of this framework; section 7 lists the coding convention used throughtout the code; section 8 gives some hints and examples on how to write a new Diana module; section 9 describes the structure of the internal diana event; section 10 gives a detailed description of the root event; finally, section **??** describes the policies that **Diana** developers and users should comply with.

3 System and environment requirements for running Diana

The code is written in C++ and is developed and debugged in Linux platforms. The most used distributions in the Cuore developers community are Debian and RedHat, but several other platforms have been used with no major problems. The code has been successfully compiled and run under MAC Darwin UNIX; although we are willing to do some effort to maintain the Makefiles also for Darwin, there is no official support on MAC. The kernel version should not matter, provided that it is not too old and that the g++ version is new enough (see below). Usually we work with kernel 2.4 or newer.

The code is maintained in a *Subversion* (SVN) server (http://svnbook.org) installed in Milano. A WEB server is available to browse the source code (http://crio.mib.infn.it/wsvn/). The code is also documented by means of **doxygen** tool (http://www.stack.nl/ dimitri/doxygen). If you do not have *doxygen* installed, you can anyway compile and run the code but you will not be able to produce the HTML documentation of the code. See section 4 for details about using SVN.

All scripts are written using standard shell **sh** or **perl**. Developers are strongly encouraged to avoid using other scripting languages, unless there are good reasons to do so.

The currently used C++ compiler is **g++** version 3.2. Versions 4.x are not supported yet. The code might not compile with older versions. Type **g++ −version** to discover which is the version installed in your system. Ask your system administrator to upgrade it, if it is too old.

The GNU Scientific Library (GSL, http://www.gnu.org/software/gsl) is used. Current version is 1.8. Older versions might not work. The script **gsl-config** must be in the PATH variable and work properly.

The root package (http://root.cern.ch) is used. Current version is 5.12 or above. The ROOTSYS environment variable must be properly set, and the PATH variable should include $ROOTSYS/bin/. Contact your system administrator for details.

The SQL data base server is **postgreSQL**. It is free and normally distributed in all Linux platforms. To compile and run you do not need a server up and running, but just the C++ libraries. Current version is 7.3. See http://www.postgreSQL.org for details about postgreSQL.

4 Getting and Compiling Diana

The simplest way to get diana is to download the code from SVN server. Just type:

svn co http://crio.mib.infn.it/svn/cuore/

and you will get a new *cuore* directory with all code in it. In this way you get the whole Cuore software repository which currently includes **Diana** and **Apollo** (the data acquisition and online monitor software). If you are not interested to online (**Apollo** is NOT documented here) enter directory cuore and type:

make diana

If everything runs smoothly, you'll get a binary file **bin/diana**. To run diana just type:

./bin/diana [options]

Next section will give you the details on how to customize Diana behaviour to your needs.

5 Running Diana

As described in next section (6) **Diana** is based on a flexible framework that is basically only capable of running a list of *sequences*, each of them being a coherent list of *modules*.

The exact meaning of *module* and *sequence* is described in section 6. Here is important just to say that a sequence is a list of modules that contain at least a *reader* and a *writer* (plus any number of additional modules) and are run on all events in a given file (or other data source) and a module is a piece of code that acts on a single event and compute come quantity (i.e. a module could take a Cuore pulse and compute its FFT and store the result into the event). We recall here that an *event* is the smaller data unit provided by data acquisition, typically a triggering pulse, a record of event infos and an optional list of nearest neighbor pulses.

How does **Diana** know which module should be run ? From a configuration file that by default is called cfg/diana.cfg. This file, through a simple syntax, tell **Diana** which modules should be run, which modules should be grouped in *sequences* and which input parameters are to be given to each module. See 6.3 to have more details about the configuration file.

Besides the configuration file **Diana** accepts a few inline options:

-h Type help

-C file set diana config file name [default cfg/diana.cfg]

-g run diana with GUI for configuring parameters (not implemented yet)

-G n n=1 starts event display; n=2 starts event filter; n=3 starts sequence filter; these are interactive graphic mode of diana (not implemented yet)

-f file set input file name or id depending on reader in use

-F file set input file list

-e n maximum number of events to be processed

-s n number of events to be skipped

Figure 1: Basic strucuture of **Diana** framework

6 Brief description of Diana framework

6.1 Framework, Modules and Sequences

The structure of **Diana** is depicted in Figure 1. The class QFrameWork is the main engine of the system. It gets a list of QSequences from the QModuleFactory. Each QSequence is a consistent list of QModule (actually classes that inherit from QModule). Each QSequence has a unique *reader*, a unique *writer* and the list of *modules*.

Besides some interaction with the Graphic User Interfaces (which is not described here because it is still very preliminary and not implemented yet), the QFrameWork::Run() method has only the function of executing the QSequence::Run() method for each defined QSequence. The QSequence has the capability to decide whether it should be re-run or not in order to allow iterative algorythms.

In the QSequence::Run() method, at the beginning the virtual member function Init() is called for the *reader*, the *modules* and the *writer*.

In the Init() function each module is supposed to get the parameters (see 6.3 for details) thorugh standard methods provided by the base class QModule, and to perform all initializations. For example, a *reader* will probably open the file or open a network connection or any other activity needed to prepare the data source to provide events; the modules will do any initialization activity like creating histograms; the writer will create the output files and so on. The only important thing to be known is that Init() is called before beginning the loop over the events.

After Init(), the QSequence enter the main loop, in which an empty QEvent (created by the QSequence itself) is filled by the *reader* method Do(QEvent*). Then the event is passed to all modules through their specific Do(QEvent*). While passing from one module to the following one ,the events get richer and richer in informations, until it reaches the *writer* that will write it on disk.

After the end of the main event loop, the method Done() is called for *reader*, *modules* and *writer*.

In case more than one *sequence* is defined, the framework automatically guarantees that the output of the first sequence is given as input to the second, and so on. Therefore, the input file (or input data source) is meaningful for the first sequence only.

6.2 Data base interface

To be done.

6.3 Configuration file and parameters

In order to run **Diana** , a configuration file is mandatory. Unless otherwise specifiied with option -C, **Diana** will look for file cfg/diana.cfg The syntax for this file is very simple. The file can define any number of sequences, each of them containing any number of modules. Each module can have any number of input parameters.

A sequence is defined by the two keywords **sequence** *seqname* and **endseq**, being *seqname* an arbitrary string (no spaces, no special characters!) that will define the name of the sequence.

Within a sequence, a *reader* and a *writer* ar mandatory; they are defined by the keywords **reader** *readername* and **endmod** and **writer** *writername* and **endmod**; furthermode any list of additional modules can be defined using the keywords **module** *modname* and **endmod**. A module can be included in the same sequence more than once, and can be present in more than one sequence. Remember that in this case ALL parameters should be written in all module sections, because for the framework parameter A in module B of sequence C, is DIFFERENT from parameter A of module B in sequence D, or of another occurrence of module B in the same sequence C. The parameters "enable" and "verbosity" are mandatory for all modules. "enable" just tell if the module should be run or not. The verbosity set the minimum level of messages will be print on screen and into log file.

Within a module you can add any number of parameters with the simple syntax **parname = parvalue**. parname can be any string with no spaces. parvalue can be a string, an integer number, a double (float) value and boolean. The type will be automatically defined by the parser: it first checks whether the parameter is a boolean (true or false), then if it is an integer, than a double and otheriwise will treat it as a string. Be careful that this means that if you right 3. the parameter will be understood as double, while 3 will be understood as integer. Other parameter types might be added in the future if needed (for example a type vector, meaning a list of numbers). Right now only bool, int, double and string are implemented.

7 Coding standards and code structure

Both **Apollo** and **Diana** are written following a set of coding standards that should be followed strictly[1] throughout the code.

1. Each class name should have the general form QClassName, without _ and using capital letters where appropriate. The only exceptions are classes that inherit from class QModule (i.e. the *modules*) which should have an M instead of Q.

2. Class headers are in a separate file called QClassName.hh and class implementation should be in file QClassName.cc. Each file contains code for one class only.

3. Class variables should always be private. Class variables should begin with letter f and using capital letters where appropriate. Example: fFileName. static member variables should begin with g instead of f

4. Each class header should include as first cuore header file QCuore.hh.

5. The code should be well documented using *doxygen* keywords. Take as an example class QVector in pkg/mathtools/ to learn how[2].

6. Each **Diana** module is generally kept in a separate package (i.e. directory). However, in case of modules that are strictly linked together, we admit reasonable exceptions to this rule.

The following are some additional points that explain the code structure and define the role of the software manager, of the package coordinators, of the developers and of the users.

1. The code is structured in several packages, i.e. sub-directories of the main pkg directory.

2. Each package has a package coordinator, whose role is to maintain this piece of code, discussing further developments with other developers and software coordinator.

3. Each developer is free to create and add new classes to its own package. If the class can be of general use, the developer should contact the software manager to see if it is appropriate to locate the new class in a more general package.

4. Each developer is free to commit new code into its own package(s). To add or modify code in other packages, the developer should get in contact first with the relevant package coordinators and in case of significant change with the software manager.

5. Whatever it is written in the previous, the most important one is always common sense and flexibility. As far as the Cuore software developers group remains reasonable small, any well motivated and exceptional violation of the above rules will be accepted. This is not to encourage violations, but not to get stuck around the rules when needed.

[1] Yes, we do know some rules were not followed very strictly eveywhere, but we encourage to do so from now on as strictly as possible!

[2] This rule is the almost ignored right now because we have introduced *doxygen* very recently. We encourage each developer to start including as much documentation as possible into the code.

8 Implementing a module

Modules are atomic units designed to perform tasks on events. They can read all the information contained in Event and, if needed, they can also modify the Event piece that the software administrator allowed to write. We encourage the developer not to write fat modules with thousands lines of code, all the leading code should be arranged in *service classes* to ensure code reusability. Therefore modules have to be intended as interfaces between **Diana** and service classes performing all the leading operations.

Modules have to implement at least the three following methods:

- Init: It is executed before the event loop, put here all the code you need before event processing. It should be intended as a constructor operator, so we suggest to put here all the initialization code, leaving the constructor empty. You should put here things like members initialization, file opening, config file parameters reading etc...
- Do: It is executed for each event in the event loop. Here you can get all the Event members and use them within service classes. You can also set the Event members that this module has permission to write (see 9).
- Done: It is executed at the end of the event loop. It should be intended as a destructor operator, so put here all the delete and free instructions, file closing etc... leaving the destructor empty.

Modules can control the **Diana** flow by executing again the Sequence they belong to, this feature has been designed for algorithms that need multiple iterations. In order to perform this operation use the following methods:

- SetRunAgain: Set that the Sequence have to be executed another time or not (default).
- GetRunAgain: Get that the Sequence is being executed another time or not.
- GetIteration: Get the number of times the Sequence has been executed so far.

Modules can save temporary data, i.e. objects that will not be written in output files and live only inside the **Diana** run. These object are useful to share informations beetwen modules that doesn't need to be dumped on output files:

- Event AuxData: use the Event method AuxData to access temporary data stored in each event. These data live only in the event loop and get lost even if the Sequence is run another time.
- Sequence AuxData: use the Module method SeqAuxData to access temporary data stored in the current sequence. These data remains alive even if the Sequence is run another time.

Instead of using the usual printf or cout output stream functions use the following commands that support the printf syntax, each of them will print messages on the screen only if the verbose level set in the config file is greater equal than the used one:

- Debug: code debugging
- Info: general information
- Warn: an error that **Diana** can handle and fix.
- Error: an error that **Diana** cannot handle but that doesn't cause the stop of **Diana** run.
- Panic: an error that **Diana** cannot handle that causes the stop of the **Diana** run.

Modules can read configuration parameters from config file. if the parameter is not found the default value provided by the user is used. Available methods are: GetInt, GetDouble, GetString, GetBool. Below is a config file entry example for a generic module:

```
module ModuleName
##mandatory parameters
# set verbosity level
verbosity = info
# set if the module have to be loaded
enable = true
# set if the module have save data into the event or not.
storedata = true
##module specific parameters
myvarint = 1
myvardouble = 1.0
myvarstring = hellocuore!
myvarbool = true
endmod
```

9 Diana event structure

The **Diana** event contains all the event by event data. Every datum or group of data are stored in EventData classes that every module can read. Write access is given only to the module that owns an EventData member. We expect a one-one correspondence between modules and EventData members. Regardless the type or completeness of input data, the **Diana** Event will be always the same, with all its members, some of them even uninitialized if the corresponding module has not been applied yet.

10 ROOT event structure

Events are stored on disk in form of ROOT files. ROOT events are converted in **Diana** events during the **Diana** loop and converted back in ROOT file when writing on disk. These events are also suitable for interactive use in the ROOT shell. At the moment we have three types of events:

- Cuoricino: Contains all the data dumped by the Milano-DAQ system (rawdata).
- Apollo: Contains all the data dumped by the Genova-DAQ system (rawdata).
- Diana: Contains the same data of the **Diana** event. After the first loop of Diana on rawdata the dumped events will be of this third type, regardless they were in origin Cuoricino or Apollo events.

11 List of package coordinators

Package Name	Coodinator e-mail	Institution	Note
apollobase	marco.pallavicini@ge.infn.it	Genova	
apollomain	marco.pallavicini@ge.infn.it	Genova	
apollotest	andrea.giachero@ge.infn.it	Genova / LNGS	
apollodaq	marco.pallavicini@ge.infn.it	Genova	
apollodb	andrea.giachero@ge.infn.it	Genova / LNGS	
apolloele	andrea.giachero@ge.infn.it	Genova / LNGS	
apollogui	andrea.giachero@ge.infn.it	Genova / LNGS	
apollomsg	andrea.giachero@ge.infn.it	Genova / LNGS	
apolloreader	sergio.didomizio@ge.infn.it	Genova	
apolloslow	andrea.giachero@ge.infn.it	Genova / LNGS	
apollosrvbase	marco.pallavicini@ge.infn.it	Genova	
apollotrigger	sergio.didomizio@ge.infn.it	Genova	
base	marco.vignati@roma1.infn.it	Roma	
comm	sergio.didomizio@ge.infn.it	Genova	
coretools	??	??	
dbbase	marco.pallavicini@ge.infn.it	Genova	
dianadb	marco.pallavicini@ge.infn.it	Genova	
dianaevent	marco.vignati@roma1.infn.it	Roma	
dianaevtgui	EGuardincerri@lbl.gov	LBL	
dianaframework	marco.vignati@roma1.infn.it	Roma	
dianagui	EGuardincerri@lbl.gov	LBL	
dianamain	marco.vignati@roma1.infn.it	Roma	
mathtools	riccardo.faccini@roma1.infn.it	Roma	
modcomputefft	EGuardincerri@lbl.gov	LBL	
moddaq	sergio.didomizio@ge.infn.it	Genova	
modfinddiscont	EGuardincerri@lbl.gov	LBL	
modoptimumfilter	riccardo.faccini@roma1.infn.it	Roma	
modtest	marco.vignati@roma1.infn.it	Roma	
modulefactory	marco.vignati@roma1.infn.it	Roma	
moduser	marco.vignati@roma1.infn.it	Roma	
modbasicparams	??	??	
modwienerfilter	??	??	
modtimereorder	??	??	
modaveragepulse	riccardo.faccini@roma1.infn.it	??	
modnoise	??	??	
modcalib	??	??	
modenergy	??	??	
modloadcurve	andrea.giachero@ge.infn.it	Genova / LNGS	
onlinemon	andrea.giachero@ge.infn.it	Genova / LNGS	
parser	marco.pallavicini@ge.infn.it	Genova	
readerdiana	marco.vignati@roma1.infn.it	Roma	
readerqino	EGuardincerri@lbl.gov	LBL	
rootevent	marco.vignati@roma1.infn.it	Roma	
writerapollo	sergio.didomizio@ge.infn.it	Genova	
writerqino	marco.vignati@roma1.infn.it	Roma	
writerdiana	marco.vignati@roma1.infn.it	Roma	

Table 1: List of packages under directory pkg. Some of them do not exist yet.

Module Name	Package	Module Goal
MARootFileReader	readerapollo	Reader for Apollo raw data root file
MRootFileReader	readerdiana	Reader for Diana output root file
MQinoDataReader	readerqino	Reader for Qino raw data files
MQinoNtupleReader	readerqino	Reader for Qino ntuples
MWriterApollo	writerapollo	Writer for raw data root files
MWriterDiana	writerdiana	Writer for Diana root files
MUserWriter	writerdiana	User defined simple and flat root file writer
MTimeReordering	modtimereorder	Order pulses according to time. Qino only.
MRemoveRetriggers	modretrigger	Remove double or multiple pulses. Qino only.
MComputeFFT	modcomputefft	Compute FFT of a pulse
MAveragePulse	modaveragepulse	Compute average pulse with heater and real pulses
MNoiseSpectrum	modnoise	Compute noise Fourier spectrum
MWienerFilter	modwienerfilter	Compute Wiener filter
MWienerFindPileUp	modwienerfilter	Search for pulse multiplicity and positions
MPrepulseAnalysis	modprepulse	Compute slope and shape of baseline before pulse
MPulseBasicParameters	modbasicparams	Compute height, rise time and decay time and others
MOptimumFilter	modoptimumfilter	Compute optimum filtered pulse
MOFPulseHeight	modoptimumfilter	Compute pulse height from OF pulse
MOFPulseShape	modoptimumfilter	Compute pulse shape variables from OF pulse
MFindTDiscontinuities	modfinddiscont	Search for Temp discont. based on thermometer
MFindCDiscontinuities	modfinddiscont	Search for Temp discont. based on heater pulses
MComputeCorrectParams	modfinddiscont	Compute parameters for gain correction
MCorrectGainDrift	modfinddiscont	Correct amplitude for gain drift
MCheckStabilization	modfinddiscont	Check stabilization quality
MSelectEventXXXX	modselectevent	Select events with features defined by XXXX
MCalibParameters	modcalib	Compute calibration parameters for each channel
MEnergy	modenergy	Compute energy of a pulse using calibration data
MLoadCurveReader	modloadcurve	Special reader that accepts load curve files
MLoadCurveVI	modloadcurve	Compute load curve online using baselines
MFindWorkPoint1	modloadcurve	First work point approximation (inversion point)
MFindWorkPoint2	modloadcurve	Work point as maximum amplitude of heater pulse
MFindWorkPoint3	modloadcurve	Work point as maximum signal / noise using OF
MTestModule	modtest	Simple test module with no specific purpose
MUserModule	moduser	Empty user module that any user can fill freely

Table 2: Preliminary list of modules. Most of them do not exist yet, and are given here for future reference.