The design of the MEG offline code

Paolo Walter Cattaneo
INFN Pavia, Via Bassi 6, Pavia, I-27100, Italy
Email: Paolo.Cattaneo@pv.infn.it

June 2, 2004

Abstract

The requirements of the MEG offline software are presented with the design criteria of a FORTRAN implementation based on ZEBRA as IO format.
The implementation details are presented only for the simulation that is more advanced at this stage.
This note does not deal with computing resource and data processing requirements. These issues are largely independent of the offline development and should be treated separately.

1 Requirements for the MEG offline

The tasks of the MEG Offline software include the generation of physics event, their simulation through the detector, the reconstruction of simulated and real data, the analysis of reconstructed events.
Throughout these steps event display and data base access must be available.
The list of tasks can therefore be splitted as follows

- IO
- Geometry description
- Data base
- Data cards
- Event format
- Event display
- Histogramming
- Physics event generator
- Monte Carlo simulation
- Event reconstruction for MC and data events
- Analisys environment

1.1 Requirements for IO

The various programs need to read and write data on a permanent storage, e.g. disks.
This IO operation may take place in several stage of the processing chain. The IO may be based on standard libraries or on user defined formats.
The IO format may be different from that used for histogramming.
The format and organization of the data should be of relevance only to the IO capability, because they will be converted from and to internal formats for the actual handling of the programs, because they will be converted from and to internal formats for the actual handling of the programs.

1.2 Requirements for the geometry description

The detector geometry and the parameters required for the simulation and the reconstruction need to be available in a single place, where they can be accessed by all the programs. They can be stored in ASCII or in binary format as long as a standard access is provided.
Together with the default values, a mechanism for changing each of the parameters interactively at run time must be provided.
The geometric information must be grouped down to an adequate level of granularity (e.g. a single chamber versus the whole spectrometer).
Each of this geometric subunit must be mappable one to one to a programming block. The format used to access information at run time must be independent by that used to store it permanently.
The connection between the two formats is given by appropriate converters.
The output of each simulation/reconstruction run must contain all the geometric information so that these information are available to the following stage.

1.3 Requirements for the data base

A number of data, e.g. phototube gains, require access on a run by run basis through a data base.
The data base should be chosen between those publicly available, possibly free.
Data base access should be possible and trasparent in a stand alone mode, and from online and offline programs.

1.4 Requirements for data cards

The data cards are a location that can be accessed by the user before running the programs storing all the informations regarding detector geometry as well as processing parameters. They are accesed by the programs in the simulation/reconstruction chain at run time. The organization of the cards must map closely that of the programs. As a principle every element in the program (e.g. a geometric element, an

algorithm, a framework) must correspond to an element in the data card.

1.5 Requirements for the event format

The event is a entity that evolves throughout the simulation/reconstrcution chain.
Each step of the chain will start from a well defined event status and trasform it adding more information.
The key point consists in defining clearly the number of stages through which the event will go and the event formats for each of this stage.
This event formats correspond to a Temporary Data Store (TDS) that is used in the simulation/reconstruction chain. That must be detached from the IO format, that must be defined separately. The connection through the two formats is given by a converter.
With a good design the largest part of the code per forming this conversion can be written automatically starting from an event descriptio starting from an event descriptionn.

1.6 Requirements for the event display

The event display must be able to access the geometric as well as the event information and send on the screen or on file pictures of the detector with or without a simulated and/or reconstructed event.
The event should be displayed both during the simulation/reconstrcution process or after the event processing stored data.
It is important to keep open the possibility of using different graphic libraries. Therefore the simulation/recosntruction code should communicate with the graphic package through library independent interfaces.

1.7 Requirements for the histogramming

The histogramming is during the execution of the simulation-reconstruction chain is required to monitor the functionality of the programs. That is different from the issues of data format output. In principle the formats used to write the data and that used for the histograms can be different.
Histogramming (distinct from data output) should consist of a separate module with interfaces independent from the histogramming library used. The last requirement may be difficult to fulfill for the calls for booking and filling unless a full interface is developed. Even if this is not done, the library specific calls should be concentrated to make easy to switch histogramming library.

1.8 Requirements for the physics event generator

This task consists in the generation of physics event, both signal ($\mu \to e^+\gamma$) and all the possible background (single and multiple Michel decays, random electrons etc.).
The nature of the experiment implies that no general purpose event

generator library is required like LUND, PHYTIA or FLUKA. The event generator routines are written explicitly for MEG.
The event topology must be steerable through external cards and the simulation result must be writable to disk in a format readable by the following stage.
The code must be organized to make easy the addition of a new physics process and to keep separate the event generation library from a small set of steering routines for stand alone running.
The structure of the produced event is simple even if formulae may be very complicated.

1.9 Requirements for the MC simulation

The simulation task consists in fetching the output of the event generator phase either reading the output of the previous phase or running the generator internally. Then the event tracks are propagated through the detector interacting with the material and generating secondaries. This stage should take place within a standard detector simulation package. GEANT3 or GEANT4 seem the only options.
The hits and the digitization generated are stored and, together with the event truth from the previous stage, must be writable to disk in a format readable by the following stage.
The simulation must access the standard geometry with the possibility of steering each geometry and simulation parameters through external cards.
The simulation must be highly modularized so that each relevant part of the subdetector can be simulated separately for laboratory or beam tests.
The simulation requires only highly standard and well known processes, therefore implementation of alternative versions of detector simulation is not required.

1.10 Requirements for the reconstruction

The reconstruction task requires the capability of reading the raw data as well as the MC output. The acquisition system, based on MIDAS, is designed to write the raw data in midas format. This format is unlikely to be the choice for IO for the rest of the MEG offline software. Therefore there are the following choices

- Change the MIDAS output format
- Double reading option for the reconstruction
- Tool for converting the MIDAS output format

The next stage is the calibration that transforms the digitization counts in physical quantities. At this stage the access to the data base is required.
Then an iterative process is required, it consists of the following steps

- Hit/Cluster finding

- Single track reconstruction
- Event reconstruction

The first step is applied separately to each subdetector.
The second step builds the photon(s) from the calorimeter information and the positron(s) combining the results from the Drift Chambers and the Timing Counter.
The last step combines positron(s) and photon(s) information through the vertex to give a fully reconstructed event.
The results of the last iteration for all the steps must be writable to disk and readable by the following analysys stage.
The reconstruction is conceptually rather straightforward because it requires little interaction between the subdetectors, infact

- DCH reconstructs positron momentum
- TIC reconstructs positron timing
- XEC reconstructs photon momentum and timing

The positron momentum and timing information may then be combined together and the photon and positron information can be combined to reconstruct the event.
A crucial tool is the extrapolator, that offers the possibility of extrapolating tracks within the magnetic field, taking into account the energy loss and computing the multiple scattering due to material.
The most important point of the design is to define clearly the event format and the interfaces at the various phases of the reconstruction so that different algorithms can be tested and compared.

1.11 Requirements for the analysis

It is probably too early to define clear requirements for the analysis. This item is the one most sensible to the computing issues, that is how the data are stored and distributed.
We concentrate here only on the requiremet that some form of DST will be made available to the end user. That could be distributed also in a compact form suitable for interactive analysis. The most obvious choice (even if not necessarily the only ones) are nowdays PAW and ROOT. We should strive to produce output in multiple formats, if the users want it.
As a consequence the reconstruction program, or an additional program running on its output, will provide DST-like data in a variety of output formats (like PAW NTuples and ROOT Tree).

2 Design criteria

2.1 Module design

In the design of the software architecture, the main parameters are

- Modularity

- Module standardization
- Clear interface
- Indipendence on specific choices of external libraries

These goals can be achieved with different tools depending on the language of choice. OO languages provide built-in tools to implement the goals of price of incresed coding complexity.
In non-OO oriented languages, OO techniques can be applied with profit at price of increased coding discipline.
An approximate parallelism between OO elements and elements of a traditional procedurale language are

- Class ↔ Library
- Class data ↔ Data structure (F77 Common block, F90 module, C structure, etc..)
- Class interface ↔ restricted set of library routines
- Base Class ↔ Module standardization
- Virtual Class ↔ Alternate choice of libraries

The modules can be further classified into several subcategories

- Data Module
- Service Module
- Algorithm Module

In the data module, the only relevant item is the data. The routines in the module perform only simple manipulation on the data.
The module standard tasks consists in initializing, reading, manipulating and making the data available to other modules.
Some data module may consists of several submodules, which are structured in the same way.
Algorithm modules have some data on their own, steering the algorithms, but mostly access data modules, upgrading them.
Service modules are similar to data modules, but they are built around an external library (e.g. card reading, IO, Histogramming) making it available to the algorithm modules. Other classifications are possible, in particular in relation with geometric objects.
The standard set of routines for a generic module are

- xxxini Initialize xxx Module with default parameters
- xxxcar Define cards of xxx Module
- xxxset Check cards and define extra parameters for xxx
- xxxend End processing Module xxx (close files, print statistics)
- xxxhis Book histograms related to Module

Each Module may have its own specific routines.
In addition an algorithm module has

- xxxrun Run algorithms transforming data accessed by the xxx Module

The next level is the 'Geometric' Module.
More routines are required to describe the geometry and the materials

- xxxdraw Draw xxx
- xxxgeom Define xxx Geometry
- xxxinitmed Initialize tracking media for xxx
- xxxmate Define materials for xxx
- xxxtmed Define tracking media for xxx

Finally there is the next level of 'Detector' Module.
More routines are required to deal with hits and digitization

- xxxdefhit Define hits for xxx
- xxxdrawhit Draw hits for xxx
- xxxhit Fill hits for xxx
- xxxdigi Digitize hits for xxx

Each Module type employs a fixed (minimal) set of routines. It is therefore possible to create automatically a module of a given type through a script reading a configuration file where the module type is specified. That would avoid unnecessary code writing. Obviously the routines would be empty and to be filled.

2.1.1 Service Module design

Service modules implement functionalities supplied by external libraries e.g. histogramming.
Ideally for the other modules using this Service Module, there is a standard, library independent interface.
When multiple options are possible for a given functionality, the programmer should be able to switch easily from one to the other. The OO languages allow to switch at run time, otherwise there are other options:

- Linking separately the alternative libraries → need relinking
- Using preprocessor $\#ifdef$ to choose within the module→ need recompilation
- Using $ifthen$ → need always both libraries for linking

The possibility of supplying a library-independent intreface is not always possible (or not easy at least) so that another possibility is to concentrate all the calls of a module type to a service module in a single routine within the module and using one of the previous approaches for switching between libraries.

2.2 Sequencer design

The other main set of programs are the sequencers: program accessing a set of service and data modules, applying some algorithm module so to upgrade some data modules. The standard routines of a sequencer are

- xxxini Initialize Sequencer xxx Modules with default parameters
- xxxcar Define cards of Sequencer xxx Modules
- xxxset Check cards and define extra parameters for Sequencer xxx Modules
- xxxend End processing Sequencer xxx Modules (close files, print statistics)
- xxxhis Book histograms related to Sequencer xxx Modules
- xxxrun Run through events applying Sequencer xxx Algorithm Modules
- xxxrunfirst Perform operation at run start
- xxxrunlast Perform operation at run end
- xxxread Read event before processing
- xxxwrite Write event afte processing

A standard event loop is supplied within the run routine as well as standard read and write event routines.
A sequencer may use some Modules .
The sequencer may consist of a base with general purpose service modules, with the possibility of adding new modules.
This addition consisting in writing some Sequencer specific code, up to one routine for each of the previous routines, that should contains only calls to the corresponding Module routines. The only information required to do such an addition is the list and the nature of the modules from a configuration files. From this file, the Sequencer user calls can be created automatically.

3 Implementation in FORTRAN77

On the basis of the previous design criteria, an implementation based on FORTRAN77 as programming language is proposed.
The software tools employed by this implementation are

- F77
- FFREAD
- GEANT3
- CERNLIB
- FFREAD
- HBOOK
- PAW

- ZEBRA
- LAPACK

They are standard items tested since many years. The development has been frozen but they provide most of the required functionality.
In the forthcoming years F77 could be replaced by F95 that is backward compatible. As to the package from CERN, the most up to date statement about their future support is that even if formal support is expired, self made installation has been made easy and versions for Linux will keep appearing for the next future.

3.1 Directory structure

The directory structure is highly standardized to ease the creation and manipulation of new subpackages.
For packages requiring only library creation (no main associated to it) the subdirectories under the *package* directory are

- banks
- cards
- include
- src

The *banks* directory contains the files having the *.bank* extension with the DZDOC documentation of the ZEBRA banks used in this package.
The *cards* directory contains the files having the *.card* extension with the FFREAD card description of the cards required to steer this package.
The *include* directory contains the files having the *.inc* extension with the common blocks and the parameter definitions required by this package.
The *src* directory contains the files having the *.F* extension with the source code related to this package.
Another directory is created automatically by the Makefile during the build procedure containing the library produced by this package.
It has the name of the Operating System (Linux on a Linux machine).
In case the package has a main program associated at it, that is an executable is produced by the build procedure another directory is required

- *package*

It contains files having the *.F* extension with the source code needed for building the executable for this package.
Another directory is created automatically by the script running the executable, that is

- output

where the output of the execution is stored.

3.2 Implementation of the IO

The IO implementation is based on the CERN package ZEBRA, that provides a machine independent format and tools to input and output structured data.
The basic item of ZEBRA is a bank. A bank contains formatted data and is linked to other banks to build a structured data unit.
The main problem in using ZEBRA banks for data handling within the program is they are very cumbersome and error prone. Furthermore, for a given module, the data set required by the program may be larger than the minimal set required by IO.
The solution to this problem, that also guarantees some library independence, is to use a TDS (Transient Data Storage) within the programs, different from the IO format. The TDS are implemented by common blocks, each bank corresponds to a common block.
That means that each bank needs a routine for bank construction starting from common block value (build) for bank writing and a routine for filling the common block from bank value (filling) for bank reading, plus a routine for printing out common block values for debugging. That implies a lot of duplication of information that make maintenance difficult.
The best solution is using the bank format description format used for producing documentation with DZDOC and use a script to to read bank information and write automatically the code for the three previous routines. The AWK script is banktocode.awk.
Further extension of this mechanism should allow to minimize the library dependent code explicitly written.

3.3 Implementation of the geometry

The geometry description follows closely the idea of the previous section.
Each geometric element is described thorugh a ZEBRA bank for IO, is available in the program as common block and can be modified at run time through a data card. All the geometry information is written out in the run header and is recovered from the following stage at reading time. That guarantees uniformity of geometry information throughout the processing.

3.4 Implementation of the data base

At the moment there is no implementation of the data base. The main point is understanding exactly where it is needed in the simulation reconstruction chain and who will fill it. After that, the most likely implementation might consist in using MySQL provided with a standard C interface. Filling the data base can be done with a dedicated standalone program using again the C interface or a Web interface.

3.5 Implementation of the data cards

This module is based on the FFREAD package, available from CERNLIB.
This package allows easy access through editing to ASCII files for the modification of parameters at run time.
The cards can be organized, so that each basic programming element is steered by one card. Using the FFREAD READ istruction, the cards can be splitted in several files, so to match the program granularity (e.g. subdetectors).

3.6 Implementation of the event format

The event must be understood as a dinamyc entity evolving during the processing. Starting from a generic header, the various stages, Physics event generator, MC simulation, digitization and the various reconstruction steps, add information to the event.
Each stage will correspond in the IO to one or more bank linked to the banks of the previous stage. In the TDS, they will corrspond to common blocks.
The design of the event format consists of defining the common blocks to be def at the input of a processing stage and of the corresponding output.

3.7 Implementation of the event display

Within the GEANT3 simulation environment there is provision for a graphics library which have access to the internal representation of the geometry and event.
The GEANT3 graphics library requires, for each Geometric Modules, the definition of the drawing attributes (e.g. color) of each subvolume. That is done in the xxx.draw.F subroutines. In the xxxdrawhit.F subroutines the hit attribute are defined and the hits are displayed.
That is done while running the simulation or running on stored data.
The 'draw' Module incorporates most of the GEANT3 specific part of the package and all the screen and file handling.

3.8 Implementation of the histogramming

The histogramming are organized in the library histo aroung the package HBOOK.
This library shields most of the library specific calls. The booking calls for each module xxx are confined in the routine xxxhis.
The filling calls scattered in various routines are not yet library independent.
An extra effort in making all the calls library independent should not be difficult and should allow to use other histogramming packages (e.g. ROOT).

3.9 Implementation of the physics event generator

The implementation of the event generator is designed around the library libmegeve. This library provides $\mu^+ \to e^+\gamma$ events and numerous background sources, under the common interface of the routine *genermegeve*, steered by a flag and put the results in a common block.
Using a different library would require only providing the same interface plus a routine for writing out events. Adding an extra background source requires writing an routine filling the appropriate common and adding one line to *genermegeve* as an extra option.

3.10 Implementation of the MC simulation

In order to ease the development of the simulation, the MC code has been designed heavily structured and modularized, according to the prescription of the design section.
Each part of the simulation will belong to one of the previously defined categories and will be implemented through a separate directory generating a separate library.
Pure Modules are:

- bfield: magnetic field
- draw : detector drawing routines
- io : input/output routines
- card : steering cards
- histo : histogram routines

Geometric Modules are:

- frame: MEG frame
- magnet : magnet and cryostat geometry
- target : target and volume geometry

Detector Modules are:

- dch: Drift Chamber
- ticp : Timing Counter Phi measuring
- ticz : Timing Counter Z measuring
- xecal : Liquid Xenon Calorimeter

The simulation of the detector geometry and of the physical process within it are done with GEANT3. That is basically the only choice in a FORTRAN77 environment. That dictates the structure of the simulation that must follow the GEANT3 steps and use the GEANT3 variables available in COMMON block.
For each subdetector module, there are at least the following include files (extension .inc), each containing one common block

- xxxmate: Materials defined for xxx Module
- xxxtmed: Tracking media defined for xxx Module
- xxxgeom: Geometry parameters for xxx Module

- xxxhit: Hit for xxx Module
- xxxdigi: Digitization information for xxx Module

3.11 Implementation of the REM sequencer

The sequencer called REM (Readout Event for Meg) realize the design criteria detialed in the previous section in F77.
It provides calls to a basic set of service modules, like IO, data card reading, histogramming, a run routine with an event loop and a set of user functions for enlarging its functionality.
In order to use it, the user must check out the basic REM directory, add the calls to the required modules in the user routines, add in the Makefile the link to the appropriate library and redefine part or all of the IO routines to match them to the IO event type (deafult functions are in the IO library).
The routines that might require redefinition are

- buildrunheader
- fillrunheader
- writerunheader
- buildevent
- fillevent
- writeevent
- generevent

These routines must be added to the rem directory that produce the executable, to overwrite effectively those in the library.

3.12 Implementation of the event reconstruction

There is no actual implementation of the event reconstruction in F77 by now.
Nevertheless we can anticipate that the reconstrcution program will be built around the REM sequencer. The reconstruction steps will be naturally Modules, eventually splitted in submodules according to the detector structure.
The reconstruction steps will follow somehow the simulation steps in reverse order. First the raw digitized data will be trasformed in physical quantities (e.g photon fluxes), then there is the hit reconstruction, then the reconstrcution of the physics objects (photons, tracks) and then of the full event (this step may go into the analysis part).
The main difference with the simulation part is that some part of the reconstrcution stage is recursive the design need to support iterations. The extrapolator can be called reloading the GEANT3 geometry and using the GEANT3 extrapolation routines.
The main point is defining the interface between these sections through the reconstructed event stages. These formats will be available in common blocks as usual.

3.13 Implementation of the analysis environment

This part is not implemented and it is that mostly dependent on the requirements of the final user and on computing issues, like data distribution.
An implementation is therefore delayed to a future time.

4 Build package

The process of automatically compiling and linking the files when they are updated is called building. This task is performed under Unix by the make program. For a Linux system the GNU Make package (gmake) is the natural solution.
The direct use of gmake is possible but it requires the explicit writing of a makefile file for each directory. As the directory evolves the files need to be updated and that is cumbersome and error prone.
For this reason several automatic systems for code maintenance have been developed. One of such system is Rules [1] designed for small and medium size projects.
This system is well suited for MEG purpose and is proposed as build system for the offline code.

5 Man power

Man power request are limited to code development and maintenance and do not address the computing and production issues.
It incorporates the requests to the subdetectors

- Core offline group: 2.0
  - CVS management
  - Framework organization
  - Utilities
  - IO
  - Histogramming
  - Data card
  - Geometry
- Physics event simulation: 0.5
- MC simulation:
  - General: 0.5
  - TIC : 0.5
  - XEC : 1.0
  - DCH : 0.5
  - Trigger: 0.5
  - Other : 0.5
- Reconstruction:
  - General: 1.0

  - TIC : 0.5
  - XEC : 1.0
  - DCH : 1.0
- Test organization: 0.5

The total of 10.0 physicist equivalent can be probably somehow reduced if we choose a minimal strategy and some activities are put in series rather than in parallel. Under this condition the extimation could decrease to maybe 8. p.e.
These estimation is valid since now to the start of experiment. At that time, the offline tasks will be reorganized, the man power requirements on some items (e.g. MC development) will decrease and other topics like support for analysis and computing issues will increase.

References

[1] P.W. Cattaneo. The rules build system. Available from www.pv.infn.it/ cattaneo/Rules.ps, 2004.