The format of the run data produced by SAMBA

Introduction

The data produced by SAMBA are of several types. There are *the* main data (tracks from the detectors), and auxiliary data (information data that helps for the analysis of the former).

The principle is to have somehow self-documented files, in order to get rid of the changes of the actual contents, due to continuous new requirements.

For auxiliary data, SAMBA produce simple text files. Currently, SAMBA save values of ntuples computed for each event (extension _ntp) and a report of the evolution of the thresholds used for triggering (extension _seuils). For any of these files, a header made of a single line including a # (comment sign) as the first character gives a name for each subsequent data of the same column. They can be opened with a variety of software, including EXCEL if the fields are supposed to be separated by blanks.

SAMBA produces also a verbose log of the run (extension : _log). The contents of this pure text file should not be analysed by any automatic procedure, since it is subject to important changes without notice.

The case of the main data is a bit more complicated and is described from now.

The main data

Partitions

The main data may be divided into partitions. Currently, SAMBA save one file of main data every hour (extension is _*nnn* where *nnn* is the number of the partition, beginning at 0). However, each partition is usable as if it were a unique main file. Processing a whole run consists of using the first partition, then opening successively the following ones, jumping after the general header which is the same in all the partitions of a given run. The run is finished when opening a new partition returns the error « file not found ».

Headers

The main file contains *text information* mixed with *binary data*. This is the same as the FITS standard ; besides, the SAMBA output could conform exactly this standard if necessary ; however it has actually some real differences.

Text information is divided into blocks to be seen as headers. A header is made of variable-length, carriage-return terminated lines (a « carriage-return » is 0x0D in hexadecimal), so they can be read by *fgets*. A line contains a variable name, followed optionally by a sign '=', followed by its value, followed optionally by a comment beginning by a sign '#'. A line beginning by a sign '#' is totally a comment.

Variable names are strictly defined (excepted important releases), while the comments are totally free and should not be used for else than human explanation.

Some lines may begin with the sign '*' : they contain a short information (a tag for instance) not relevant to the header library. Particularly, all headers are terminated with the end-of-header line (' * ---------- ', 10 signs '-' followed by the carriage-return).

Structure of the general headers

The main file begins with the Setup header. It is followed by one header for each Detector known at the experiment level (not only the detectors managed by the current computer). These are followed by one header defining each channel; for instance, there are 3 channels : ‘chaleur’, ‘centre’ and ‘garde’ for each NTD detector. The number of Detector headers and Channel Definition headers to decode is given by the variables ‘Bolo.nb’ et ‘Voies.nb’ (respectively), which can be found in the Setup header seen previously.

The Channel Definition header may include, after its end-of-header line, the coefficients of the filter used in the post-buffer computation (included in the trigger and event building algorithm). In that case, the end-of-header line is followed by a ‘* Filtres’ tag line. The number of stages of the filter is written in a binary 32-bits integer. For each stage, the number of direct coefficients is written in a new binary 32-bits integer, followed by this number of 64-bits double-float direct coefficients. Then the reverse coefficients are added (one 32-bits integer number and as many 64-bits double-float reverse coefficients).

The last header is the Run header, which contains run-dependant information (terminated of course by the end-of-header line), and this header is followed by the ‘begin-of-data’ line (‘* Donnees’ terminated by the carriage-return).

Beyond this limit are the data, which are described below, depending of the type of the sauvegarde. **So, in any case, provided you are aware of the conditions of the run in some way, you may simply start retrieving the amplitudes just after decoding the string ‘* Donnees’**.

Data for a stream file

In a stream file, only the saved channels are described by a Channel Definition header (consequently, the Channel Definition set is only computer-wide). The data are put as binary 16-bits words in the same order as the Channel Definition header order, for a given time. After this, the data for the next time sample is written, and so on, until the end of file.

Data for events

Of course, the events are written in the order they arrive. Here is the structure for one event :

An event begins with the Event header, which gives the event-level, event-depending information. This information includes the number of channels which participated to the event, in the variable ‘Voies.nb’. The Event header is followed by this number of channel information, each one including a Event Channel header and the binary data. The index (in the list of Channel Definitions at the beginning of the file, which is experiment-wide) of the currently saved channel is given by the Event Channel header variable ‘Numero’.

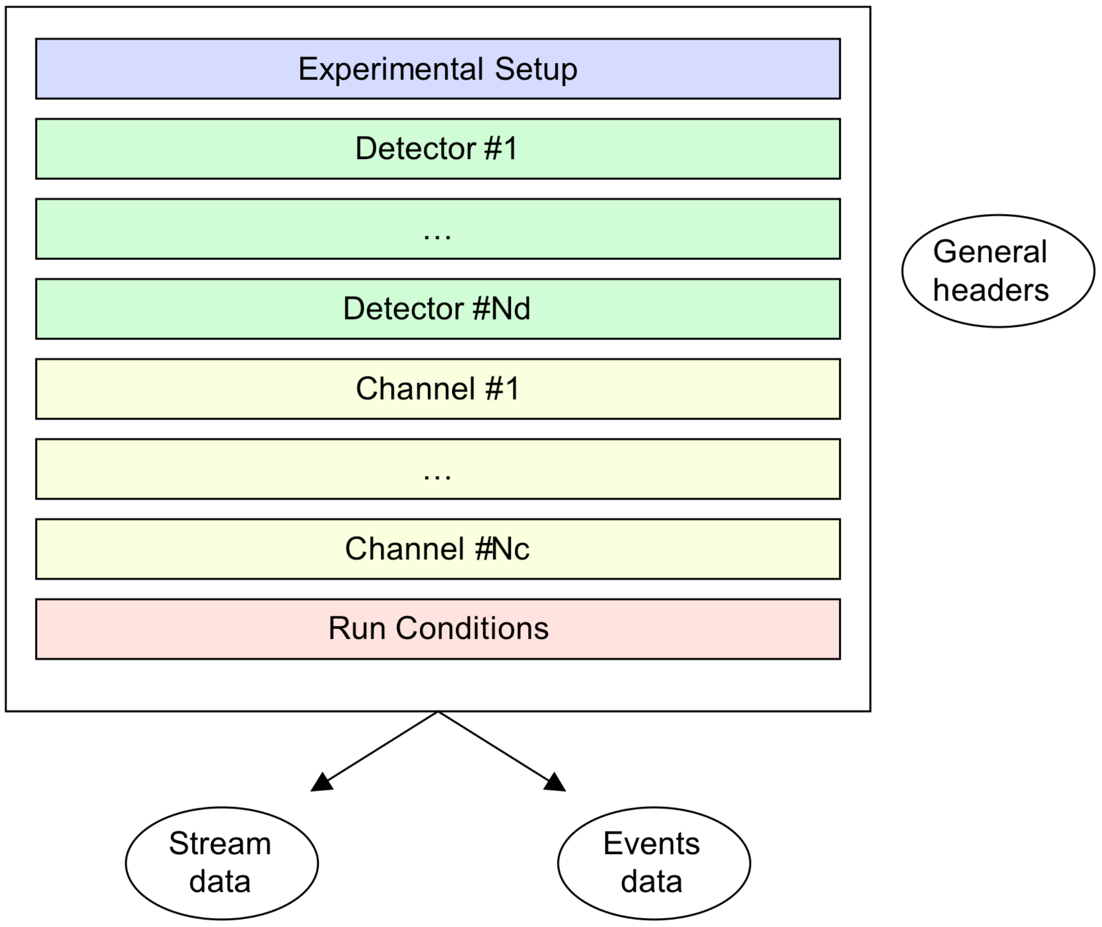
Depending on the value of the variable ‘Filtre.nb’ found in the Event Channel header, a few filtered data of the channel may be found immediately after the header. They are to be used as start values for a later filtering of the saved data, and are included as binary 64-bits double-float values.

In any case, the Event Channel header variable ‘Dimension’ give the number of the binary 16-bits integer values which follow, and that are part of the current event concerning this channel.

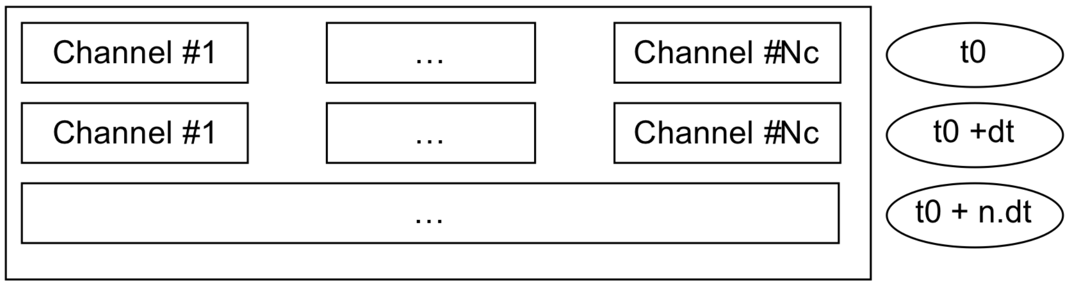
Graphical representation of the structure

In the following, coloured rectangles represent a header (optionally with few binary data), while white rectangles are the binary data from the detectors.

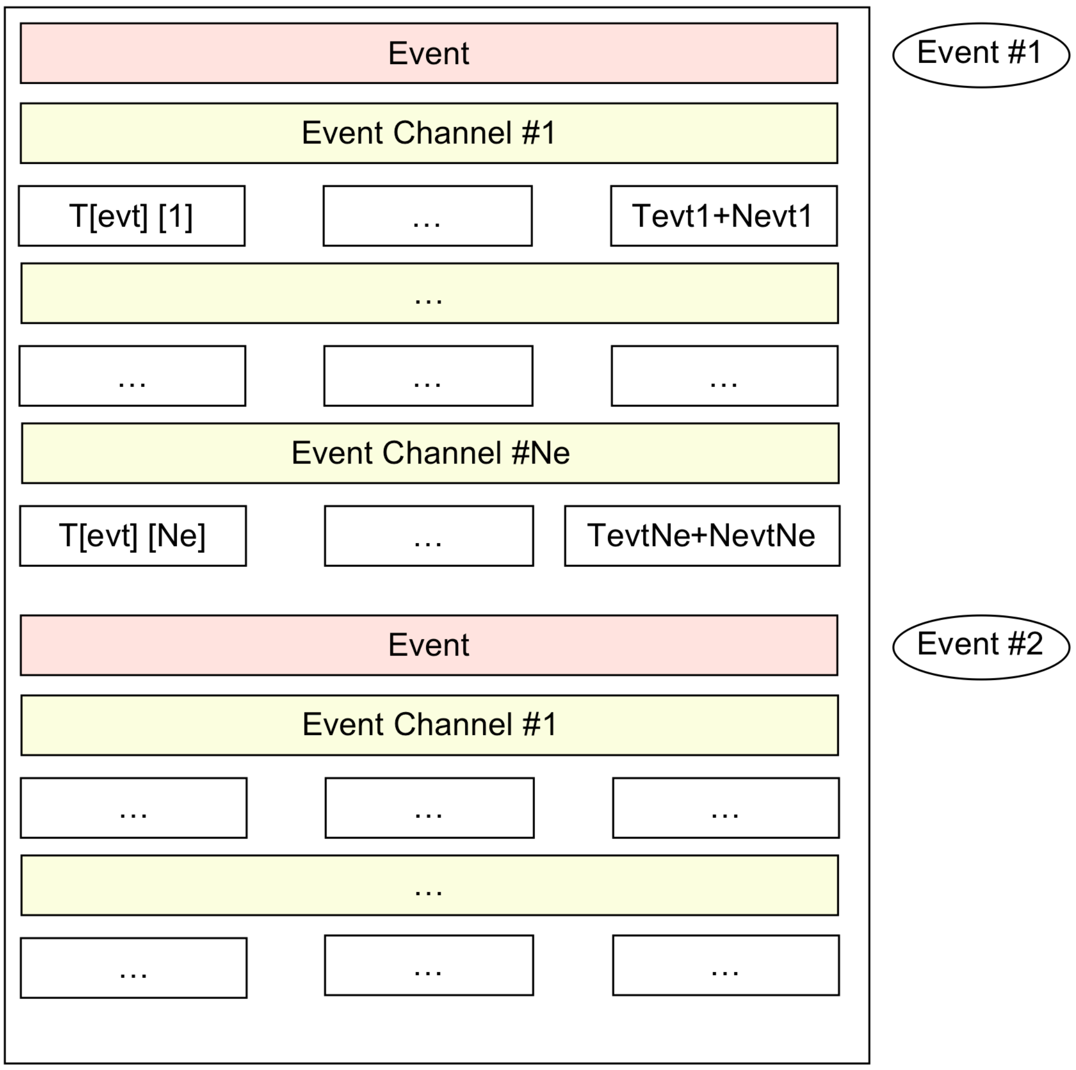
This first figure represents the structure of the general header:



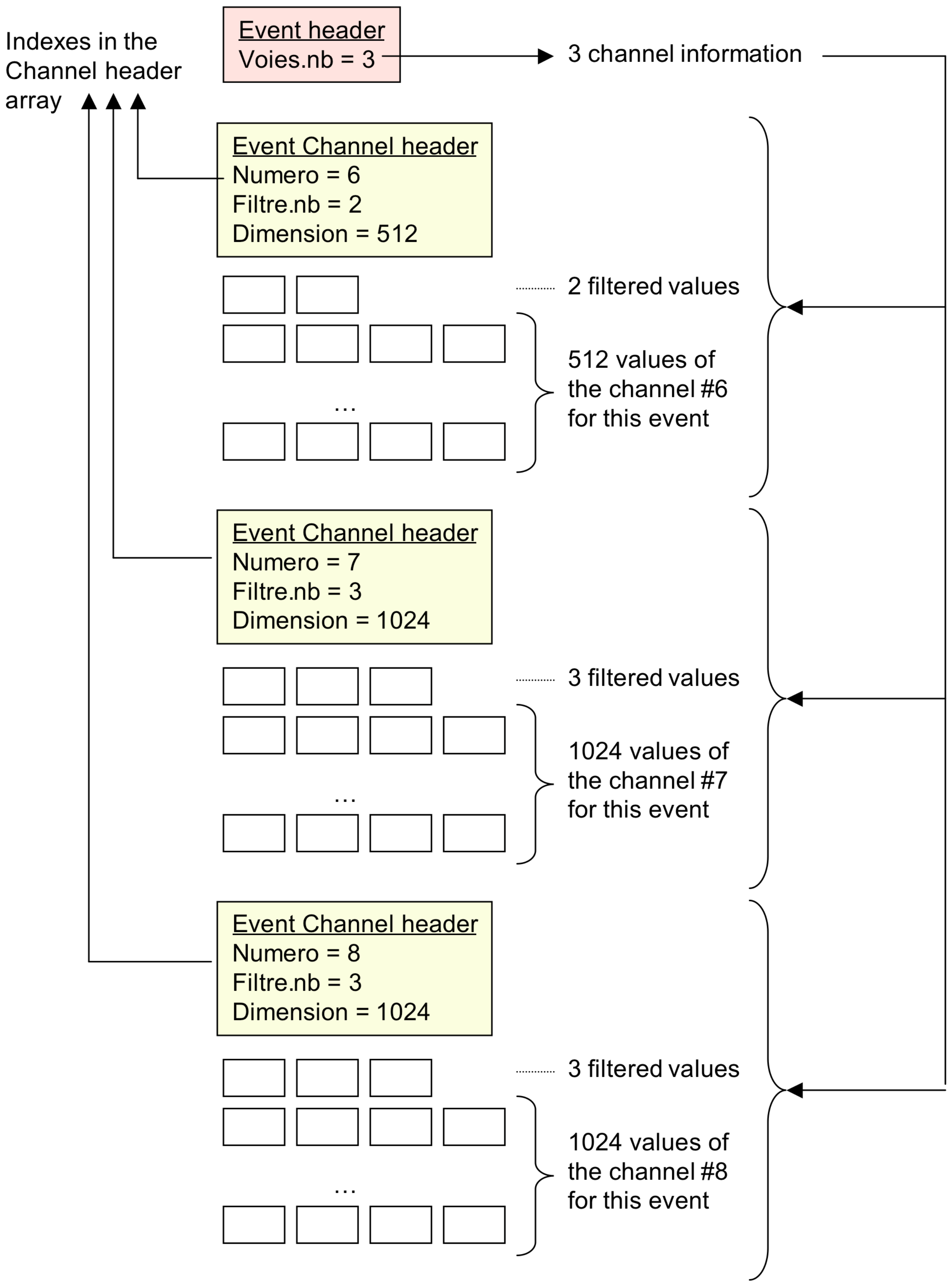
If the data are stream data, this structure follows:



If the data are events data, the structure is rather this one:



Links between data for one event:



Software support

A library, written in C, is available to get events from event files. The goal is that users have not to mind the file structure of all this information. By the side, the software has been designed to allow some forward and backward changes in the structure or in the variables. By using this software, users may in some cases directly use files of a version different from their software; in the worst case, they only relink or recompile their software when important changes occur.

The usage look like the following:
- include the files `rundata.h` and `archive.h` , which define the variables containing all the information for a given event ;
- query the access to the data with `ArchRunOpen(runname)`, which returns 0 if no file exists;
- get an event of this run with Arch`EvtGet(num)`, which returns false if there is less than *num* events.

The event (i.e. all the relevant information) is then available in a structure `TypeEvt` and other related structures, like the detector and the channel definition.

Installing the library

1. prepare a directory to keep this software, and download the '`libarch.tar`' tarfile into it;
2. untar this tarfile: a directory '`libarch`' is created and fully dedicated to reading SAMBA event files;
3. go to this directory, then enter '`make lib`' to build the library;
4. you may then enter '`make example`' to build and run the example.

Should a new release be needed, only the sources will be distributed.

Using the library

To retrieve events, you need 3 (perhaps 4) routines:

`char ArchReadInit(max_evts)`

Initialises the library. A maximum number of events is needed.

If the functions return a false (null) value, the initialisation didn't work. You may examine the standard global variable 'errno' to get an idea of the error.

`char ArchRunOpen(char *nom, char log, char *raison)`

Opens a run (whose name is given in '`nom`'). The run may have the form of a directory containing partitions; the routine logically concatenates the partitions as if there were a unique (big) file for all the run.

The '`log`' flag, if set, make the routine printing the general headers of the run.

If opening fails, the routine returns a false (null) value and the reason for this is given in the string '`raison`'.

Other runs may be concatenated to a previous run (or a previous set of runs) by just calling again this routine. The event numbers for a given run will be shifted by the total number of events of the previously open runs.

In any case, the total number of events to get is given by 'EvtNb'.

`char ArchEvtGet(int num, char *explic)`

Retrieves the given event of the open runs. If something fails, the routine returns a false (null) value and the reason for this is given in the string '`raison`'. See next section for how the use the information on the event.

`void ArchRunClose()`

Close all the files and reset all the counters. The status is the same as just after calling '`ArchReadInit`'; particularly, you may open a new set of runs.

Structure of the event

The event is available in predefined variables. To keep later the event in memory (for managing more than one a at time, for instance to compare two of them), you have to make an allocation of the same structure as used by the library.

Note that the different headers are defined in ‘`archive.h`’. This file defines also the main event variables names to be retrieved for each event; however the definitions of the corresponding structures, as well as the common variables found only in the various headers, are defined in ‘`rundata.h`’.

Event level

The variable is ‘`ArchEvt`’ and its type is ‘`TypeEvt`’. It is defined as follows:

```
typedef struct {
    int num;                /* number from the beginning of the readout (starts at 1)*/
    int sec,msec;           /* date of the maximum in seconds and microseconds       */
    int TMsec,TMmsec;       /* total dead time at the date of this event             */
    char regen;             /* true si regeneration was in place                     */
    int gigastamp,stamp;    /* timestamp of the maximum (nb samples 100kHz)          */
    off_t pos;              /* offset within the file                                */
    float delai;            /* delay since the previous event (seconds)              */
    unsigned int liste[4];  /* detectors above their threshold (1 per bit)           */
    int bolotrig;           /* detector with higher event (index in Evt[].voie)      */
    int voietrig;           /* channel with higher event (index in VoieEvent[])      */
    int voielocale;         /* channel with higher event (index in Evt[].voie)       */
    int nbvoies;            /* number of channel saved in this event                 */
    TypeVoieArchivee voie[MAXVOIESEVT]; /* data of the channels saved in this event */
} TypeEvt;
```

Channel level

So there are ‘`ArchEvt.nbvoies`’ channel saved in the event, and they are stored in ‘`ArchEvt.voie[i]`’, which is of type ‘`TypeVoieArchivee`’. This type is defined as follows:

```
typedef struct {
    int num;                /* numero de voie global                                 */
    int sauvee;             /* numero de voie sauvee                                 */
    int64 debut;            /* point de debut d'evenement                            */
    int dim;                /* nombre de points sauvegardes                          */
    float horloge;          /* duree d'un point (millisecs)                          */
    int avant_evt;          /* nb points memorises avant le trigger                  */
    short trig_pos;         /* amplitude minimum                                     */
    short trig_neg;         /* amplitude maximum si seuil negatif ou indifferent     */
    int sec,msec;          /* datation du debut en secondes et microsecondes        */
    float base;            /* ligne de base en ADU                                  */
    float bruit;           /* bruit sur la permiere moitie de la ligne de base      */
    float amplitude;       /* amplitude en ADU                                      */
    float montee;          /* temps de montee en millisecs                          */
    float integrale[MAX_EVTPHASES]; /* integrale du signal, par phase               */
    float total;           /* integral du signal total                              */
    float decal;           /* decalage avec la voie ayant trigge (millisecs)        */
    float seuil;           /* meilleur fit evt unite sur ligne de base              */
    float energie;         /* fit evt unite                                         */
    float temps;           /* difference avec debut                                 */
    TypeDonnee *filtrees;  /* tampon contenant les donnees traitees sur tampon (monit) */
    int max;               /* dimension du susdit                                   */
    int64 fin;             /* point de fin du meme                                  */
    double *demarrage;     /* 1ers points filtres pour recalcul                     */
    int nbfiltres;         /* nombre de ceux-ci                                     */
    int recalcul;          /* index dans l'evt ou le recalcul doit commencer        */
} TypeVoieArchivee;
```

'ArchEvt.voie[i].num' (let's call it 'voie') is the index in the array 'VoieManip', whose structure is given below. This array gives the general information about all the channels, including the detector index (in 'VoieManip[voie].det') in the array 'Bolo' which in turn gives all the information about the detectors (structure below).

The data of the event is to be found, for each channel, in 'VoieEvent[voie].brutes.val[i]', which is an array of 'short' and whose number of useful data is 'ArchEvt.voie[i].dim'.