Abstract

This document describes the software used to convert raw EMCAL data in PRDF format (Phenix Raw Data Format) to calibrated EMCAL data.

1 Overview

First step is to access the PRDF data : you do this using the ONCS Event package[Pur] and the famous `Eventiterator` class, which returns you an `Event` object. Only then the EMCAL software come into the game.

Then, you will ask an `emcRawDataAccessor` object (figure 1) to process the Event. In this step, the RawDataAccessor (RDA hereafter) will *decode* the Event, i.e. fetch the high-pre, high-post, low-pre, low-post and tac values for each and every channels, and store them into a `emcRawDataObject` (RDO hereafter). The channels are those belonging to one of the FEMs described in a *configuration file* which is needed to initialize the RDA.

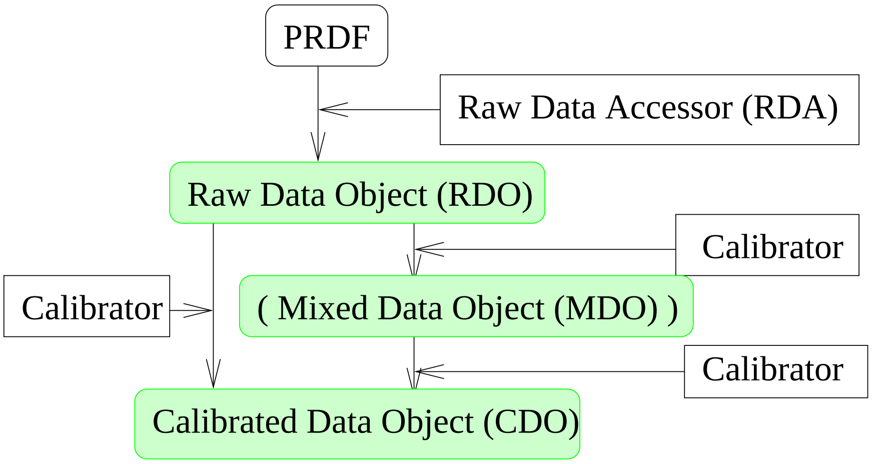

Figure 1: Overview of calibration process

You will then pass this RDO to an `emcCalibrator` which will convert it into an `emcMixedDataObject` (MDO hereafter), where you'll get only 2 values per channel : an ADC value and a TAC value, i.e. the Calibrator will make the pre-post subtraction and choose between high and low gain values.

The same calibrator can then convert the MDO into an `emcCalibratedDataObject` (CDO hereafter) where ADC and TAC values will be converted into GeV and ns. Note that the calibrator can go directly from the RDO to the CDO, with a slight lost in performance (see section 3 for details).

The calibrator needs to fetch some calibration data to perform its work. Those data are provided by the `emcDataManager`. They can come from files or from Objectivity Federation (except time-zero calibration which are by now only on files).

Once you get a CDO, you are free to do whatever you'd like with its data : fill another object (e.g. an old STAF table), clusterize it, or store it directly using PHOOL (the CDO is a child of TObject and as such can be put in a `PHIODataNode`).

In the following section we describe in some more (technical) details the above steps, giving code examples and some design considerations and issues. If you just need to know how to calibrate EMCAL raw data, see sections 2 to 4. Those are the only section you really need to know about. But if you would like to know more about how calibrations are stored into DB, see sections 5, or if you would like to fill yourself some calibration objects, see section 6. Section 7 gives insight into plugins, and section 8 gives an alphabetical index of classes used.

2 Where is the code ?

To play with the following examples, you'll need to access some libraries, which are located as any other offline libraries, i.e. in `$OFFLINE_MAIN/lib`.

The code itself is located in CVS /offline/packages/emc-calib directory. It is split into 5 packages, each package corresponding to one library :

- Static, to handle all the (static) index marshalling, and to access time-zero calibration data. Produces a libEmcStatic library.
- Dynamic, to handle dynamic mapping of FEM channels and so on, and to access raw data (e.g. emcRawDataAccessor and emcRawDataObject, see section 3) are defined there. Depends on Static package. Produces a libEmcDynamic library.
- DataManager, to define a unique interface for DB access. Produces a libemcDataManager library.
- Calib, to define the calibration data objects (section 6) and the Calibrator-related classes (section 4). Depends on Dynamic, Static and DataManager. Produces a libemcCalib library.
- DataManager_plugins, to define the plugins that store/retrieve objects to/from the Objectivity Federation and to retrieve(only) from files. Depends on Calib and DataManager. Produces (so far) a unique libemcOM library.

The inter-library dependencies are depicted in figure 2. Obviously some links are doubled. We are aware of that and it should be fixed some days...

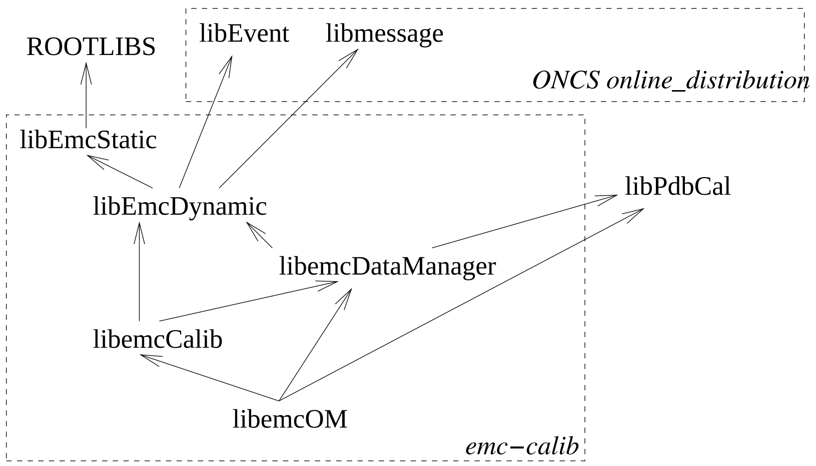

Figure 2: Inter-library dependencies

As the libemcOM library depends on all the other ones, all the examples given in this document should work (from Root prompt) as long as you first load it :

```
gSystem->Load(''libemcOM.so'');
```

NOTE: the code located in /offline/packages/emc-calib/ is a *stable* (and slowly evolving) version of what is developped and used online, located in /online/monitoring/emc/Common CVS directory.

In the following examples, we only use part of the interfaces of the classes we describe. Feel free to browse the include files to have a look at the complete interfaces. If you miss some methods, just ask for them ! (kistenev@bnl.gov or aphecetc@in2p3.fr or schutz@subatech.in2p3.fr or delagran@subatech.in2p3.fr or martinez@subatech.in2p3.fr or enterria@in2p3.fr).

3 The emcRawDataAccessor (RDA)

The `emcRawDataAccessor` is a *singleton*, which means that only one such object (i.e. one instance of this class) can exist at any time in your program. Here is how you can access this instance :

```
emcRawDataAccessor* rda = emcRawDataAccessor::GetInstance() ;
```

This will work only if the RDA has been initialized properly at least once :

```
int status ;
char* config = ''FEM.conf'' ;
emcRawDataAccessor* rda = emcRawDataAccessor::GetInstance(status, config) ;
```

The *configuration file* describes what and where EMCAL FEMs are. Here is how it looks like :

```
# pos    -> location of the crate within calorimeter Sector (SM144 number)
#           It is used to identify the packet (numbered like 8001+pos)
# pin    -> personal identification number of the Crate (MUST be written on a
#           sticker on the mother board). Range 0 - 255. Copied into user word
#           in the packet header (module Id) and used to verify the correctness
#           of the Crate-packet relations
# sector -> Sector Identifier
map: emcal
data: towers
fem: pos=0, pin=20, sector=w0
delay: tac=3 post=4
chan: 0-143
fem: pos=1, pin=22, sector=w0
delay: tac=3 post=4
chan: 0-143
...
```

NOTE: It is good practice to always check that the pointer you get back from the `emcRawDataAccessor::GetInstance()` method is not null, which would indicate an error.

Once you get a pointer to the RDA, the next step is to obtain a pointer to a RawDataObject :

```
emcRawDataObject* rdo = rda->GetRawDataObject() ;
```

NOTE: for performance reasons, the RDO pointer you obtain from the RDA is always the same (unless you re-initialize the RDA in the meantime) so the memory allocation for it is done only once. Its size is fixed by the number of FEMs present in the configuration file (i.e. it is *not* zero-suppressed).

The actual filling of the RDO is still governed by the RDA (you have no other way of filling it, anyway), by its `processEvent(Event*)` method. The reader is referred to the ONCS documentation about Event handling for more details about getting access to Event objects[Pur]. A loop over events looks like this :

```
int status ;
char* config = ''FEM.conf'' ;
emcRawDataAccessor* rda = emcRawDataAccessor::GetInstance(status,config) ;
if (!rda) {
  cerr << ''<E> Cannot get a RDA !'' << endl ;
}
else {
  emcRawDataObject* rdo = rda->GetRawDataObject() ;
  Eventiterator* it = new fileEventiterator(''data.prdf'') ;
```

```
  Event* event ;

  while ( event = it->getNextEvent() ) {
    if ( event->getEvtType() == DATAEVENT ) {
      rda->processEvent(event) ;
      // here you can play with the RDO if you need to ...
      cout << (*rdo) << endl ;
    }
  }
}
```

4 The emcCalibrator

4.1 Principle

You get access to a calibrator using a *calibrator factory*. The idea behind this design choice is to ease the move from one calibration strategy to another one with a minimal change in the reconstruction code. The factory must be initialized first with the desired strategy and then you can use it :

```
emcCalibratorFactory::Initialize(''emcRawDataCalibrator'') ;
emcCalibrator* calib = emcCalibratorFactory::GetCalibrator() ;
```

NOTE 1: By now, only one strategy (i.e. classname) can be used with the factory and is named `emcRawDataCalibrator` (figure 3). In the future, if alternative strategird are defined, you will just have to change the name in the Initialize call and that's all.

NOTE 2: In case the factory is not initialized, the second line in the above example will return a null pointer.

Before using the calibrator, you can set a number of its properties :

```
// ADC counts below the threshold are considered to be zero
calib->SetThresholdADC(10) ;
// Which High Post value to consider
// as the limit between High and Low gain ?
calib->SetHighLowLimit(1024) ;
// Select the calibration data sources
emcManageable::EStorage data_source = emcManageable::kDB_Objy ;
calib->SelectSource(''Gains'',data_source) ;
calib->SelectSource(''Pedestals'',data_source) ;
calib->SelectSource(''HLRatios'',data_source) ;
// Disable collection of Tof calibration data
calib->SelectSource(''Tofs'',emcManageable::kNone) ;
// The CDO will be zero-suppressed
calib->SetZeroSuppression(true) ;
// Print calibrator info
calib->Print() ;
```

We will explain more on data sources in section 5.

Once the calibrator is set up, here is how you use it, assuming you already have a RDO pointer at hand (see section 3) :

```
emcMixedDataObject mdo ;
emcCalibratedDataObject cdo ;
PHTimeStamp when(2000,6,24,0,0,0) ;

1 calib->Calibrate(*rdo,mdo,when) ;
  calib->Calibrate(mdo,cdo,when) ;

// Can be simplified to :
// 2 calib->Calibrate(*rdo,cdo,when) ;
// if you don’t need the information stored in MDO.

// Play here with your brand new calibrated data object
int i ;
for ( i = 0 ; i < cdo.GetSize() ; i++ ) {
  cout << cdo.GetSoftwareKey(i) << ‘‘ Energy=’’ << cdo.GetEnergy(i)
       << ‘‘ Time=’’ << cdo.GetTime(i) << endl ;
}
cout << cdo.GetTotalEnergy() << endl ;
...
```

NOTE: The short version (line 2 above) to go from RDO to CDO has a *small* disadvantage over the two-step version : it will recreate a MDO object at each call, whereas the two-step version uses the very same MDO each time (I’ve never really timed the difference though).

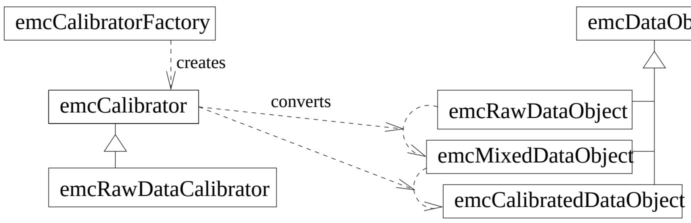

Figure 3: emcCalibrator and related classes

4.2 More on the emcRawDataCalibrator strategy

In this section we will describe what the emcRawDataCalibrator (RDC) is actually doing and what it needs.

When going from the RDO to the MDO, the RDC will :

1. choose between high and low gain, depending on the RDC GetHighLowLimit(), subtract the relevant pedestal and apply the proper high-to-low gain ratio. Schematically :

```
if ( hgpost < GetHighLowLimit() ) {
  adc = lgpre-lgpost ;
  adc -= pedestal_for_LG_Pre-Post(amupost) ;
  adc *= HLRatio ;
}
else {
  adc = hgpre-hgpost ;
  adc -= pedestal_for_HG_Pre-Post(amupost) ;
}
```

2. subtract tac pedestal :

```
tac -= pedestal_for_TAC(amupost) ;
```

Thus the RDC needs to access Pedestals and HLRatios values (see section 5)

When going from the MDO to the CDO, the RDC will calibrate energy and time, for each channel which is above the GetThresholdADC() value. To do this, the RDC needs to access Gains (relative normalization), Tofs and time-zero (absolute normalization) calibration data.

[be more specific here...]

Fine. Unless you are really interested in more details, you can safely stop reading here, and enjoy your CDO !

5 The emcDataManager, the plugins and ... the databases

The calibrator, as described in section 4, needs to access some calibration data. Apart from the time-zero calibrations which are only available on file so far and are accessed via the Static package, all other calibration data are accessed through the DataManager.

The DataManager was developped during a period when we did not know yet exactly what the calibration data will be. In the same time we wanted to be able to access the available ones and not to be obliged to re-code everything when new ones come into the game. Thus we defined a unique interface to access calibration data, and this interface is reflected by the `emcDataManager` class.

The DataManager can only handle objects which are of a subclass of a `emcManageable` class. It's the object by itself who instructs the DataManager where to fetch/put it. The DataManager is in fact doing nothing by itself, but delegates actual work to some Object Managers or *Plugins* which know how to read/write calibration data from/to database (figure 4). Using some trick in the code of the plugins, those are automatically known by the DataManager as soon as you load the library(ies) containing them (trick originally found in pdbcal code[Mes]).

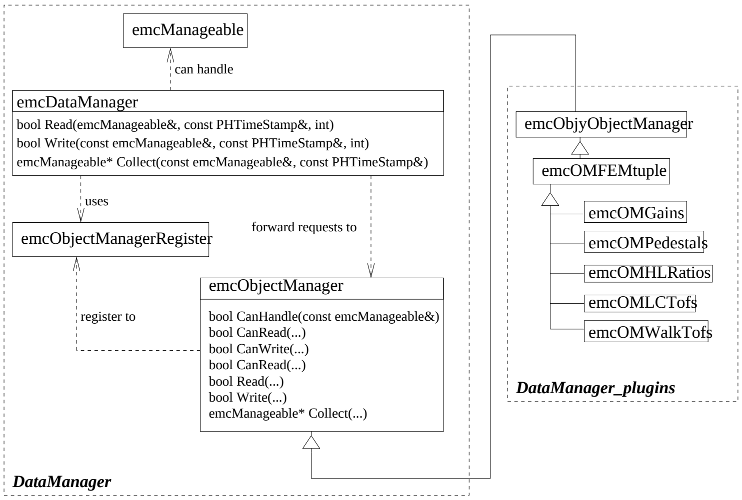

Figure 4: emcDataManager and related classes

The DataManager is a singleton, and its interface basically consists in 3 methods : Read, Write and Collect. Before explaining those, we must first explain how data are stored in the databases.

As stated in section 4, the EMCAL calibrator needs calibration data about Gains, Pedestals, HLRatios, and Tofs (Least-Count and Walk). We hereby call those different ones "calibration *flavours*". Each flavour is sitting in a different *database*, sitting in the very same *Objectivity Federation*. Each database consists in calibration *banks*, storing information about 1 FEM (144 channels). Each calibration bank has a header, which is sitting itself in a different database (the Tag one). This header contains a bank identifier, a validity period (2 `PHTimeStamps`) and an insertion time (1 `PHTimeStamp`). The bank themselves are vectors of calibration items (there might be one item or several items per channel, e.g. for Gains). Those items are objects of classes named PdbEmcXXX, indicating that they are PdbCal classes[Mes]. Table 1 indicates the relationship between calibration flavours, database names and PdbCal low-level class names. Those are given for the sake of completeness, but please note that, as a normal user, you are not supposed to deal directly with those classes.

We now return to the DataManager interface. The `emcDataManager::Read` method is used to fetch an object spanning one FEM (144 channels usually) only, i.e. one bank. The `Write` method can be given an object spanning any number of FEMs, the underlying plugin will take care of splitting it into the

Flavour	DB	PdbCal class
Gains	calib.emc.GainsCal calib.emc.GainsTag	PdbEmcTracedValue PdbCalHeader
Pedestals	calib.emc.PedestalsCal calib.emc.PedestalsTag	PdbEmcPedestal PdbCalHeader
HLRatios	calib.emc.HLRatiosCal calib.emc.HLRatiosTag	PdbEmcHLRatioVector PdbCalHeader
LCTofs	calib.emc.LCTofsCal calib.emc.LCTofsTag	PdbEmcLCTof PdbCalHeader
WalkTofs	calib.emc.WalkTofsCal calib.emc.WalkTofsTag	PdbEmcWalkTof PdbCalHeader

Table 1: Databases and PdbCal names for each EMCAL calibration flavour

relevant number of FEMs (banks). The `Collect` method is used to recover an object spanning those FEMs which are described in the configuration file. For example, if you would like to recover calibration data about gains for all the FEMs described in configuration file FEM.conf, for date 27-Jun-2000 00:00:00, try this :

```
 1 int status ;
 2 char* config = ‘‘FEM.conf’’ ;
 3 emcRawDataAccessor* rda = emcRawDataAccessor::GetInstance(status,config) ;
 4 PHTimeStamp when(2000,6,27,0,0,0) ;

 5 emcDataManager* dm = emcDataManager::GetInstance() ;
 6 emcGains gainDriver ;
 7 emcGains* gains ;

 8 gain_driver.SetSource(emcManageable::kDB_Objy) ;
 9 gains = dynamic_cast<emcGains*>(dm->Collect(gainDriver,when)) ;
10 if (gains) { //play with it... }
```

The 4 first lines are described in section 3. The RDA is initialized here not for real data access, but just for reading the configuration file, which is needed for the Collect.

In line 5 we get an instance of the DataManager.

Then we create in line 6 an `emcGains` object (for more details on this class, see section 6) and in line 7 a pointer to an emcGains. The need to create 2 such objects might seems strange to you (after all, you just want one object with all the information, right ?). In fact, this design decision has to do with performance and responsability. Thanks to the `gainDriver` object, the DataManager will be able to locate the relevant plugin that can handle this object, and will be able to know where to fetch him (line 8). This `gainDriver` is only used as a type (i.e. you don't need to fill it or whatever). The Collect method will return you a pointer to the full object you wanted. Doing this that way, it's the plugin responsability to actually access the DB if (and only if, thus saving some time in many cases) needed. You are *not* the owner of the returned pointer, thus you don't care about memory management for this object.

Line 9 does the actual job of returning you the object you wanted. In case you are not familiar with the `dynamic_cast`, it is the new C++ way of casting things. You might be familiar with this syntax :

```
gains = (emcGains*)(dm->Collect(gainDriver,when)) ;
```

The `dynamic_cast` has the advantage to dynamically (i.e. at runtime) check that the cast you request is a valid one (otherwise it returns null), thus it is safer.

There are in fact some other methods of the DataManager you may be aware of :

- `ls`, which outputs on screen the list of available plugins,
- `Reset`, which will reset all the plugins (i.e. after a call to Reset, they will behave as if they were freshly created, e.g. all the data that they might have collected just disappear upon Reset),
- `Set[Get]VerboseLevel`, which can be used to make the DB access much more verbose.

6 Calibration objects

In this section we describe the different calibration data objects that are used by the Calibrator, and that you can store/retrieve from the DB.

If you have read the preceding sections (and you'd better read them!), you know that we are dealing with different calibration *flavours* : Gains, Pedestals, HLRatios, LCTofs, WalkTofs. We also use time-zero calibrations, but those are treated in a different way so far, so we won't discuss them here.

All the flavours have a number of common points, which lead us to define some base classes, and derive from them to implement the specifics of each flavour. This is illustrated in the class diagram (figure 5).

What you can store/retrieve to/from the DB (using the DataManager) are subclasses of `emcFEMtuple` class (which is itself a subclass of the generic `emcManageable` class). Notice that the `emcGains` class we used in example of section 5 is indeed such a subclass.

6.1 Using the calibration objects

Let's assume you already have at hand an `emcFEMtuple` object, named `femtuple`. Regardless of its actual type (emcGains, emcPedestals, etc...), you can apply the following methods :

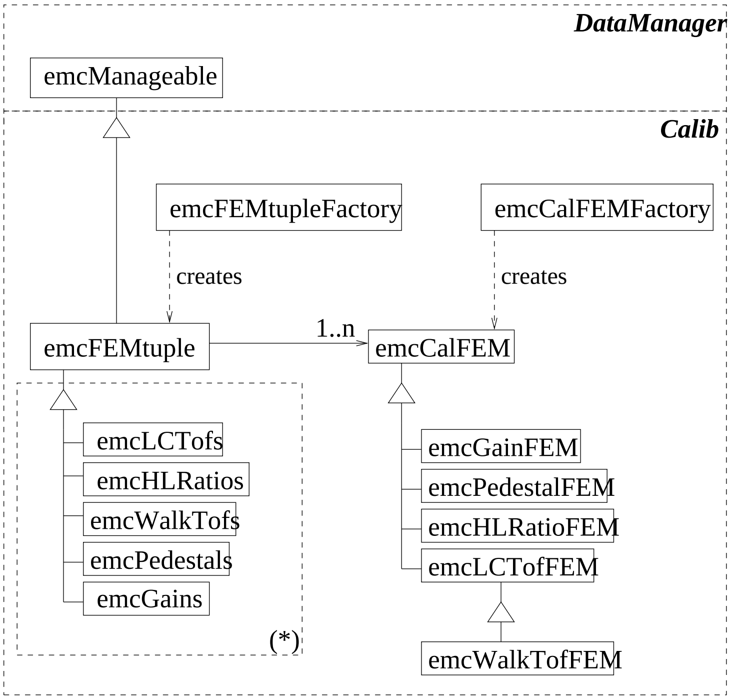

Figure 5: Class diagram for calibration data classes

```
// How many channels handled by femtuple ?
int n = femtuple.GetNumberOfChannels() ;
// How many FEMs handled by femtuple ?
int nfem = femtuple.GetNumberOfFEMs() ;
// AMU post-pre value for a given fem
int ifem = 2 ; // should be between 0 and nfem-1
int pp = femtuple.Get_post_pre(ifem) ;
int tp = femtuple.Get_tac_pre(ifem) ;
// Tells if femtuple is valid at time when
PHTimeStamp when ;
bool valid = femtuple.IsValid(when) ;
// Print
femtuple.Print(1) ;

// Get a pointer to a given FEM
emcCalFEM* fem = femtuple.GetFEM(ifem) ;
```

The last line gives access to a given calibration FEM. Again, regardless of its type (emcGainFEM, emcPedestalFEM, etc...), you can :

```
// Get the absolute position
int ap = fem->GetAbsPosition() ;
// Get the pin number
int pin = fem->GetPinNumber() ;
// Get the amu post-pre and tac-pre
int pp = fem->Get_post_pre() ;
int tp = fem->Get_tac_pre() ;
// Get the validity period
PHTimeStamp start,end ;
start = fem->GetStartValTime() ;
end = fem->GetEndValTime() ;
// Tells if fem is valid at time when
PHTimeStamp when ;
bool valid = fem->IsValid(when) ;
```

OK. Let's now turn to the real use of those objects, namely getting the calibration values for individual channels. The `emcFEMtuple` has a number of `getValue` methods for that very purpose :

```
/// get value method for time-independant objects
float getValue(int ichannel) const { return -9999.9 ; }

/// get value method for time-dependant objects
float getValue(int ichannel, int thetime) const { return -9999.9 ; }

/// get value method with a kind (e.g. for pedestals)
float getValue(int ichannel, int amucell,
               const char* kind) const { return -9999.9 ; }
```

Those methods in fact only call the corresponding `getValue` of the relevant `emcCalFEM` subclass (depending on the flavour), after proper marshalling of the `ichannel` parameter (which is running from 0 to the total number of channels in a femtuple, and -usually- from 0 to 143 in a calfem).

The first method is used for time-independent objects, as HLRatios, LCTofs and WalkTofs. The second one is used for time-dependant objects, as Gains ; the time parameter is an incremental time (zero=start-of-validity period of the object). The last one is a special case used for pedestals, where you must provide the amu (post) value and the pedestals type (LG_Pre-Post, HG_Pre-Post, TAC) you wish to access.

6.1.1 Gains

`emcGainFEM` is by far the most complicated one among all the emcCalFEM subclasses. For each channel, what is stored is a list of `emcTracedValue`, i.e. a list of line pieces (3 parameters=1 start time $t0$, 1 slope and 1 constant). Each piece of line is valid during a given time. When you request the gain value for a given channel at a given incremental time t, four cases have to be considered (figure 6) :

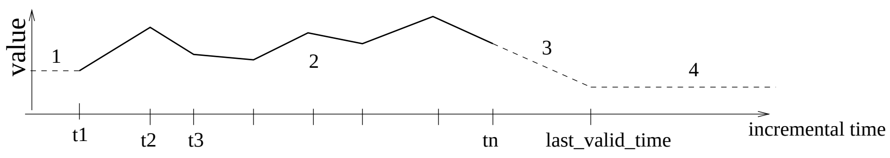

Figure 6: emcTracedValues = what is stored for Gains.

1. t is below the time of the first TracedValue. The value returned is then the constant of the first line.
2. t falls within one of the TracedValue. The value returned is computed by this TracedValue (=constant+$(t - t0)\times$slope)
3. t falls after the last TracedValue, but before a parameter (common to all channels in this FEM) called last_valid_time. The parameters of this last TracedValue are used to extrapolate until t
4. t is after last_valid_time : the value returned is the one returned by the last TracedValue, at time t =last_valid_time.

NOTE: as there might be several TracedValues per channel, the Gains banks might have more than 144 items (see section 5).

Storage organization of `emcGainFEM` object is depicted in figure 7, together with its relation to the Gains bank organization.

You can loop over all the TracedValue using an iterator-like procedure :

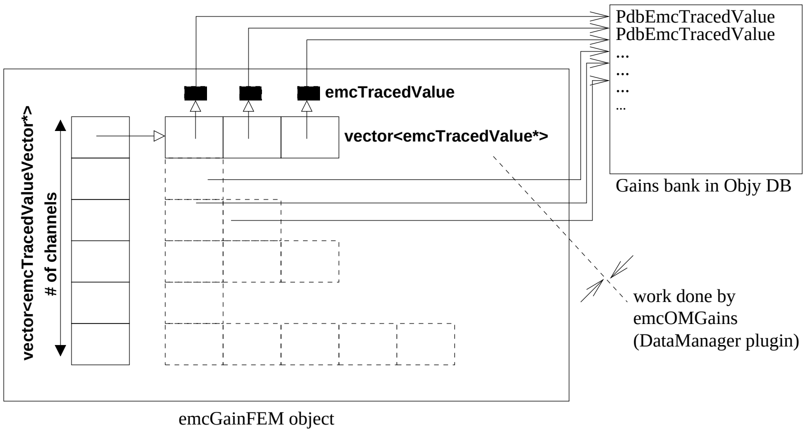

Figure 7: emcGainFEM object and its corresponding Objy bank

```
// Get a filled gain fem from a femtuple (e.g.)
emcGainFEM* gainfem = dynamic_cast<emcGainFEM*>(femtuple.GetFEM(0)) ;
int i ;
emcTracedValue* val ;
for ( i = 0 ; i < gainfem->GetNumberOfChannels() ; i++ ) {
  gainfem->FirstItem(i) ;
  while ( val = gainfem->NextItem() ) {
    cout << (*val) << endl ;
  }
}
```

6.1.2 Pedestals

Each channel has 3 types of pedestals : HG_Pre-Post, LG_Pre-Post and TAC. For each type, there are 64 integer values (one per AMU (post) cell) (figure 8). Thus `emcPedestalFEM` objects are the "fatter" calibration objects we deal with.

Access to actual value is done the following way :

```
// Get a filled pedestal object from a femtuple (e.g.)
emcPedestalFEM* pedfem = dynamic_cast<emcPedestalFEM*>(femtuple.GetFEM(0));
int ichannel=12;
int amupost=2;
int low = pedfem->getValue(ichannel,amupost,''LG_Pre-Post'') ;
int high = pedfem->getValue(ichannel,amupost,''HG_Pre-Post'') ;
int tac = pedfem->getValue(ichannel,amupost,''TAC'') ;
```

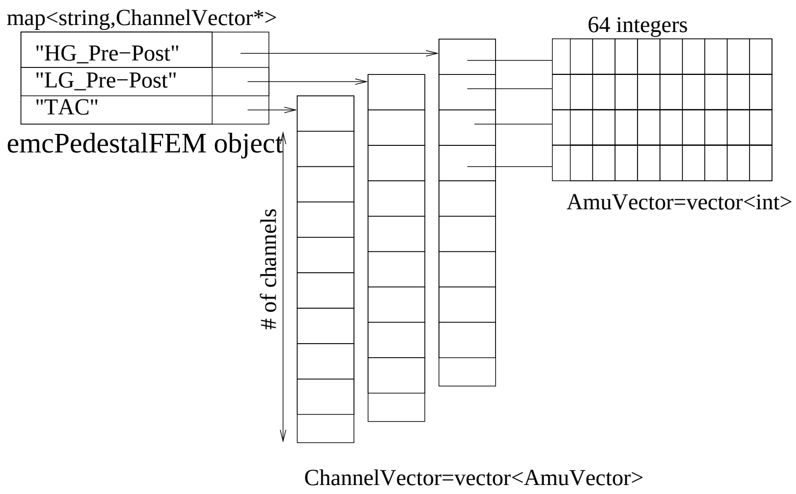

Figure 8: emcPedestalFEM calibration object

6.1.3 HLRatios

In principle, there should be only one High-to-Low gain ratio for each channel. But we keep four values for each channel : an average value (which is what we are returning with the `getValue` method so far), a rms value, and two parameters from a linear fit : a slope and an intercept (figure 9).

Note: It seems that the fitted value could be always better than the *global* average, in which case we would use the fitted value in `getValue` method [to be checked].

To access to actual values, use :

```
emcHLRatioFEM* hlrfem = .... ;
float av = hlrfem->GetAverage() ;
float rms = hlrfem->GetRMS() ;
float slope = hlrfem->GetSlope() ;
float inter = hlrfem->GetIntercept() ;
float globalAverage = hlrfem->GetGlobalAverage() ;
float globalRMS = hlrfem->GetGlobalFEM() ;
```

6.1.4 LCTofs and WalkTofs

The 2 classes emcLCTofFEM and emcWalkTofFEM are (so far) strictly identical (indeed, the second derives from the first). For each channel, two floating point values are kept. You can access them using `GetValue1,2(int ichannel)` methods.

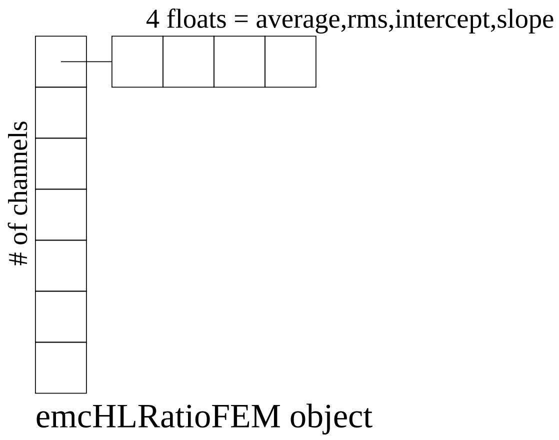

Figure 9: emcHLRatioFEM calibration object

6.2 Building the calibration objects

OK. If you have survived the reading up to this point, it means you really need those damned calibration objects ! Found a bug somewhere in a plugin and want to fix it ? Want to build you own objects ? Want to write you own plugin ? Fine. Here we go...

6.2.1 Overview

Remember that what you are trying to build are some sub-class of `emcFEMtuple` class. You start from an empty one, and you a) add some `emcCalFEM` to it b) fill the emcCalFEM objects you've added. For both type of objects, there are factories available that may be of some help (figure 5).

```
// Create an empty femtuple of flavour Gains
emcFEMtuple* tuple = emcFEMtupleFactory::Create(``Gains'') ;
// You could also write :
// emcGains* tuple = new emcGains ;
// Append one CalFEM to it and convert it to emcGainFEM type
emcGainFEM* fem = dynamic_cast<emcGainFEM*>(
  tuple->AppendFEM(absPosition,pinNumber,post_pre,tac_pre,begin,end)) ;
// Fill the fem object (see below)
...
```

The actual way of filling the `fem` object depends on the flavour (see below).

You might also want to selectively replace 1 FEM among all those stored in an emcFEMtuple object (e.g. the one which reach its end-of-validity time). Use the `emcFEMtuple::ReplaceFEM(int ifem, emcFEMtuple&)` method for that purpose (see section 7 for example).

6.2.2 Gains

`emcGainFEM` objects are filled with `emcTracedValue` objects. For a number of reasons (one being to stick to some internal conventions), you should put 144 channels into it.

```
int j ; // channel number (0..143)
for ( j = 0 ; j < 144 ; j++) {
  time = ... ;
  constant = ... ;
  slope = ... ;
  fem->AddNewItem(j,new emcTracedValue(time,constant,slope)) ;
}
fem->SetLastValidTime(last_valid_incremental_time) ;
```

Note the use of the `SetLastValidTime` method which spans the whole FEM (see section 6.1.1). In the above example we put only one TracedValue per channel. You are free to put more per channel, e.g. :

```
int j ;
for ( j = 0 ; j < 144 ; j++) {
  int i ;
  for ( i = 0 ; i < j ; i++ ) {
    time = ... ;
    constant = ... ;
    slope = ... ;
    fem->AddNewItem(j,new emcTracedValue(time,constant,slope)) ;
  }
}
fem->SetLastValidTime(last_valid_incremental_time) ;
```

6.2.3 Pedestals

For pedestals, you must feed the object with 3 pedestals types. For each type 64 values per channel must be given.

```
int j ; // channel number (0..143)
for ( j = 0 ; j < 144 ; j++ ) {
  emcPedestalFEM::AmuVector vlow,vhigh,vtac ;
  int amu;
  for ( amu = 0 ; amu < 64 ; amu++ ) {
    vlow.push_back(some_ped_low_value) ;
    vhigh.push_back(some_ped_high_value) ;
    vtac.push_back(some_ped_tac_value) ;
```

```
  }
  // it's important to group the following lines (as it's an
  // append for the same channel).
  fem->AppendOneChannel("LG_Pre-Post",vlow) ;
  fem->AppendOneChannel("HG_Pre-Post",vhigh) ;
  fem->AppendOneChannel("TAC",vtac) ;
}
```

6.2.4 HLRatios

Each channel must be fed with 4 floating point values.

```
int j ;
for ( j = 0 ; j < 144 ; j++ ) {
  fem->AppendOneChannel(some_average,some_rms,some_intercept,some_slope) ;
}
```

6.2.5 LCTofs and WalkTofs

Same interface as HLRatios but with only 2 values per channel (AppendOneChannel(value1,value2)).

7 More about plugins

OK. The latter the better. Here come the plugins ! The class diagram for plugins is the right part of figure 4. Plugin principles are quite general, but we will focus here on plugins managing emcFEMtuple objects. Adaptation to other kind of emcManageable should be trivial.

7.1 Advertising the plugin capabilities

The first duty of a plugin is to advertise what kind of objects it can read, write, collect or handle (handle=read OR write OR collect), i.e. it *should* implement the following methods (defined in the base `emcObjectManager` class and returning false by default) :

```
bool CanCollect(const emcManageable& object) ;
bool CanHandle(const emcManageable& object) ;
bool CanWrite(const emcManageable& object) ;
bool CanRead(const emcManageable& object) ;
```

By default, the `CanHandle` is defined as :

```
bool emcObjectManager::CanHandle(const emcManageable& object) {
    return
      CanWrite(object) ||
      CanRead(object) ||
```

```
    CanCollect(object) ;
}
```

A plugin that advertise Reading capability should be able to read a Manageable from DB and/or from file.

Here is a code sample on how you can do this. It's all about casting :

```
//_____________________________________________________
bool emcOMGains::CanRead(const emcManageable& object)
{
  // we advertize here that we can only handle a given
  // number of object types, e.g. emcGains only.

  bool rv = false ;
  const emcManageable* object_ptr = &object ;
  const emcGains* test = dynamic_cast<const emcGains*>(object_ptr) ;

  if (test) {
    rv = true ;
  }
  return rv ;
}
```

In the above example, the plugin can read emcGains objects, whatever their source is. You might want to selectively announce reading capability from one given source. You could do this like that :

```
//_____________________________________________________
bool emcOMGains::CanRead(const emcManageable& object)
{
  // we advertize here that we can only handle a given
  // number of object types, e.g. emcGains only.

  bool rv = false ;
  const emcManageable* object_ptr = &object ;
  const emcGains* test = dynamic_cast<const emcGains*>(object_ptr) ;

  if (test && object.GetSource()==emcManageable::kDB_Objy) {
    // can only read from Objy DB (not from files)
    rv = true ;
  }
  return rv ;
}
```

If the plugin announces Writing capability, it must be able to split the given emcFEMtuple into pieces spanning one FEM (=1 bank).

The CanCollect is usually equivalent to CanRead (i.e. if a plugin can Read, it should be able to Collect also).

7.2 The mother class emcOMFEMtuple

As many operations are common to all the calibration flavours, those operations have be gathered into the base plugin class `emcOMFEMtuple`. The Read,Write and Collect methods are defined there. Plugins deriving from this class must implement (assuming they can want to access Objy at all) :

- `FromPdbCalBank(emcCalFEM*,PdbCalBank*)` : to convert a PdbCalBank to an emcCalFEM
- `ToPdbCalBank(emcCalFEM*,PdbCalBank*)` : to convert an emcCalFEM to a PdbCalBank
- `char* GetPersistentClassName` : to know the name of the low-level PdbCal classname (e.g. PdbEmcTracedValue for flavour Gains)
- `Reset` : to reset its internals.

They might also redefine the following :

- `MakeBankID(PdbBankID& bankID,int absPosition,int pinNumber, int post_pre, int tac_pre)` which should construct the bankID from the four next parameters. There is a default implementation of this one (see below).
- `ReadFromFile(emcFEMtuple& femtuple, int absPosition, int pinNumber, int post_pre, int tac_pre)`, if the plugin is able to read from file.

A note on bankID : so far, the bankID internal value (which is an 4-bytes integer) is built from four characteristic values of the FEMs : absolute position, pin number, post-pre, tac-pre, as illustrated in figure 10.

31			0	
absPosition	pinNumber	post−pre	tac−pre	Pedestals,HLRatios,LCTofs,WalkTofs
absPosition	0	0	0	Gains

Figure 10: bankID for FEMs

7.3 The Collect procedure

The `emcOMFEMtuple` class maintains an internal (pointer to) object of class `emcFEMtuple` (actual type depends on the flavour), figure 11.

The first time the Collect is called (or after a Reset of the plugin), the pointer to the emcFEMtuple object is null. In that case the Collect is just a loop over the FEMs described in the configuration file. Each FEM is read from DB/File and then added to the internal emcFEMtuple object. Schematically :

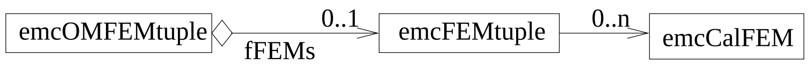

Figure 11: emcOMFEMtuple class

```
emcManageable* emcOMFEMtuple::Collect(const emcManageable& object,
  const PHTimeStamp& when) {

...

emcRawDataAccessor* rda = emcRawDataAccessor::GetInstance() ;
int nSM = rda->GetDynamicData()->getnSM() ;
const SuperModule* SMMap = rda->GetDynamicData()->getSMMap() ;

// Create an emcFEMtuple object of the right type
fFEMs = emcFEMtupleFactory::Create(object.GetCategory()) ;

int ifem ; // fem index
int code ;
bool ok = true ;
emcFEMtuple* oneFEM ;

// Loop over FEMs

for ( ifem = 0 ; ifem < nSM && ok==true ; ifem++ ) {

  // Here some code to skip the monitoring crates
  // (if any) for anything which is
  // not Pedestals of HLRatios
  ...

  // Create the ‘‘driver’’ object of the right type
  // This object is only needed to give the type and
  // the data source.
  oneFEM = emcFEMtupleFactory::Create(object.GetCategory()) ;
  oneFEM->SetSource(object.GetSource()) ;

  // Build the code for the FEM.

  code = emcCalFEM::FEMCode(SMMap[ifem].absPosition,SMMap[ifem].femPin,
                            SMMap[ifem].post_pre,SMMap[ifem].tac_pre) ;
```

```
  // Read this FEM
  ok = Read((*oneFEM),when,code) ;

  // Add this FEM to fFEMs
  ok = fFEMs->Add((*oneFEM)) ;

  // The delete free the pointer, but not the internal of it (note 1)
  delete oneFEM ;
 }
```

Note 1 : remember that an emcFEMtuple is a list of emcCalFEM objects. When you create an emcFEMtuple, it is owner of this list. As soon as you add an emcFEMtuple to another one, it looses its ownership on its internal emcCalFEMs, thus a delete will *not* erase the emcCalFEM objects (it will be deleted, in the above example, when fFEMs is deleted).

Once the Collect has been done once, subsequent call to it will behave slightly differently. First, when entering the Collect method, we check if the fFEMs object is still valid at time when, in which case we just return it :

```
emcManageable* emcOMFEMtuple::Collect(const emcManageable& object,
  const PHTimeStamp& when)
{
  if (fFEMs) {
    // If what we have already is still valid, returns it.
    if (fFEMs->IsValid(when)) return fFEMs ;
  }
}
```

Otherwise, we enter the loop as before, but we do some extra check (the assumption here is that only part of the emcCalFEMs might be out of the validity range) :

```
...
for ( ifem = 0 ; ifem < nSM && ok==true ; ifem++ ) {
...
  // fFEMs is already filled up, se we first check
  // that we really need to fetch new values for this FEM
  if ( ! fFEMs->IsValid(when,ifem) ) {
    ok = Read((*oneFEM),when,code) ;
    if (ok) {
      ok = fFEMs->ReplaceFEM(ifem,(*oneFEM)) ;
    }
  }
}
```

i.e. we only read FEMs which are not still valid. The ReplaceFEM will just exchange one pointer to another one (and delete the first one). Hope the explanation is clear...

OK. So far so good. The Read and Write methods are just trivially using pdbcal interfaces to access PdbCalBanks, and then the relevant To/FromPdbCalBank are called to fill actual emcCalFEM objects (see code for details).

7.4 Plugin trick

Before we end up, you should be aware on the mechanism used to convert a normal C++ class into a DataManager plugin. In the implementation file of your plugin, just call the following macro :

```
#include "emcObjectManagerRegister.h"
EMCALDM_REGISTER_OBJECTMANAGER(classname,title,comment)
// e.g. for Gains plugins :
EMCALDM_REGISTER_OBJECTMANAGER(emcOMGains,
  "emcOMGains",
  "Object to handle Gains and TofT0s objects")
```

Note that there is no trailing ; at the end of the macro call.

That's it. You should be able to debug or write a plugin by yourself now !

8 Index

In case you'd like to locate in which package a class is defined, here's an alphabetical index of the classes in packages Static, Dynamic, DataManager, Calib and DataManager_plugins. In case you are curious, this index is produced by a little perl script (index_classes.pl) located at /afs/rhic/phenix/users/aphecetc/script. Enjoy !

Classname	Package Name
EMCalFEE	Dynamic
EmcDynamicData	Dynamic
EmcIndexer	Static
EmcSector	Static
EmcStaticData	Static
EmcSuperModule	Static
PbGlCalibrationData	Static
PbGlIndexer	Static
PbGlSector	Static
PbGlSuperModule	Static
PbScCalibrationData	Static
PbScIndexer	Static
PbScSector	Static
PbScSuperModule	Static
emcBankIDConverter	Calib
emcCalFEM	Calib
emcCalFEMFactory	Calib

Classname (cont.)	Package Name (cont.)
emcCalibratedDataObject	Calib
emcCalibrator	Calib
emcCalibratorFactory	Calib
emcCheckDBUponFiles	DataManager_plugins
emcDataManager	DataManager
emcDataObject	Dynamic
emcFEMtuple	Calib
emcFEMtupleFactory	Calib
emcGainFEM	Calib
emcGains	Calib
emcHLRatioFEM	Calib
emcHLRatios	Calib
emcLCTofFEM	Calib
emcLCTofs	Calib
emcManageable	DataManager
emcMixedDataObject	Calib
emcNamed	DataManager
emcOMFEMtuple	DataManager_plugins
emcOMGains	DataManager_plugins
emcOMHLRatios	DataManager_plugins
emcOMLCTofs	DataManager_plugins
emcOMPedestals	DataManager_plugins
emcOMWalkTofs	DataManager_plugins
emcObjectManager	DataManager
emcObjectManagerRegister	DataManager
emcObjyObjectManager	DataManager_plugins
emcPedestalFEM	Calib
emcPedestals	Calib
emcRawDataAccessor	Dynamic
emcRawDataCalibrator	Calib
emcRawDataObject	Dynamic
emcTofT0FEM	Calib
emcTofT0s	Calib
emcTracedValue	Calib
emcWalkTofFEM	Calib
emcWalkTofs	Calib

References

[Mes] M. Messer. Pdbcal explained. http://www.phenix.bnl.gov/ messer/tutorials/pdbexplained.html.

[Pur] M. Purschke. The event handling library user guide. http://www.phenix.bnl.gov/ phoncs/oncs/code_documentation/Event/doc/index.html.