Roadmap for CMS Calorimetry Software

Jeremiah Mans, Francesca Cavallari, Robert Harris,
André Holzner, Paolo Meridiani, Shahram Rahatlou,
Christopher Tully, Richard Wilkinson

May 23, 2005

Version 0.9

1 Indexing

The first task is define a logical cell index which is useful for most reconstruction tasks. This requires a close mirror of the subdetector topology and simple algorithms for converting between subdetectors where possible. For compactness, we choose to use a bit-packed unsigned 32-bit integer.

The preshower is not yet covered in this document as we need input from the preshower group to determine the most appropriate structures for the preshower.

1.1 Packing

	Bits	**Usage**
	(31:24)	Subdetector id (also functions as a version id)
HCAL		
	(23:16)	Unused (0)
	(15:14)	Depth index
	(13)	Eta sign bit (1=positive, 0=negative)
	(12:7)	Eta index (1-41)
	(6:0)	Phi index (1-72)
EB		
	(23:17)	Unused (0)
	(16)	Eta sign bit (1=positive, 0=negative)
	(15:9)	Eta index (1-85)
	(8:0)	Phi index (1-360)
EE		
	(23:15)	Unused (0)
	(14)	Eta sign bit (1=positive, 0=negative)
	(13:7)	X index (1-100)
	(6:0)	Y index (1-100)
Trigger		
	(23:14)	Unused (0)
	(13)	Eta sign bit (1=positive, 0=negative)
	(12:7)	Eta index (1-32)
	(6:0)	Phi index (1-72)

Conversion between HCAL and EB is very easy.

$$
\begin{aligned}
i\eta_{\text{tower}} &= \frac{i\eta_{\text{crystal}} - 1}{5} + 1 \\
i\phi_{\text{tower}} &= \frac{i\phi_{\text{crystal}} - 1}{5} + 1
\end{aligned}
$$

The Trigger Towers in the ECAL Barrel are made of units of 5x5 crystals, which also correspond to elementary read-out towers, and the TT assignment can be also obtained with a simple computation. There is a one-to-one correspondence between EB Trigger Towers and HB/HE Trigger Towers.

In the ECAL endcap the correspondence between crystals and trigger towers is drawn by the HCAL Endcaps scintillators, but limited by the geometry of the EE crystals. Thus, conversions between HCAL and EE cannot be as easily performed, but some can be done with trigonometry. The trigger tower assignment in EE cannot be peformed in a simple manner and must be obtained from a table. The EE numbering uses the range 1-100 rather than -50 to 50 to avoid an unnatural boundary at X=0 or Y=0. By convention, we propose that the signed range (-1)-(-100) be used for the negative EE and (1)-(100) be used for the positive EE.

1.2 Subdetector id assignments

Id	Logical name	Subdetector
1	EcalBarrel	Barrel ECAL (EB), uses EB structure in cell id field
2	EcalEndcap	Endcap ECAL (EE), uses EE structure in cell id field
3	HcalBarrel	Barrel HCAL (HB), uses HCAL structure in cell id field
4	HcalEndcap	Endcap HCAL (HE), uses HCAL structure in cell id field
5	HcalOuter	Outer barrel HCAL (HO), uses HCAL structure in cell id field
6	HcalForward	Forward Calorimeter (HF), uses HCAL structure in cell id field
7	EcalTriggerTower	RCT trigger tower from ECAL, uses Trigger structure in cell id field
8	HcalTriggerTower	RCT trigger tower from HCAL, uses Trigger structure in cell id field

The subdetector id field can be used also as a versioning field. If the numbering must be updated for a detector (take EE as an example), the original assignment can be retained as EcalEndcap_v1=2, and the new (active) assignment becomes EcalEndcap=EcalEndcap_v2=11.

1.3 CellId Classes

While the persistent representation of the cell id should be simple (e.g. a bit-packed 32-bit unsigned integer), we may wrap this unsigned integer in classes which can decode the various bit fields very quickly. Using the newer versions of ROOT (ROOT v4), which will be required by the EDM in any case, it will possible to persist objects which do not inherit from TObject. Thus, the CellId class can be supported both in full ORCA and ORCA-Lite[1].

We propose to have *separate* CellId classes for each different subdetector, except that HB and HE should be considered a single subdetector for CellId purposes. These classes will have parallel structures, but *will not* form an Object-Oriented tree structure. This choice allows the avoidance of virtual function calls and allows the compiler to optimize the use of these simple decoding methods (mostly shifts and masks). CellId classes should also implement equality operators and any necessary operators to enable the use of STL maps and algorithms. Draft versions of these classes are attached to this document in Appendix A.

[1] Although there are different definitions of ORCA-Lite going around, one convenient definition is "any functionality which does not require database access".

2 Data Structures

2.1 Introduction and General Comments

This document will consider four tiers of data in the reconstruction path.

0 Raw Format set by hardware, common collection and presentation for all subdetectors. Used as input by Calorimetry software (and potentially produced by Calorimetry simulation), but not specified by this document.

1 Digi Represents same level of data (ADC samples) as the raw format, but with electronics-to-logical map applied. This tier includes the precision readout ADC counts as well as separate structure representing the trigger primitives. At the raw level, these two may be combined into a single data block.

2 RecHit Represents the energy (in units of GeV) deposited into a cell of the calorimeter and has the same granularity as the Digi. This tier is the most widely used tier in higher-level reconstruction.

3 Composite Combines multiple RecHits in a manner convenient for reconstruction, such as a projective tower collecting the HCAL and ECAL energies and used primarily for jet reconstruction.

The data structures described here are simple structs. These will be collected in vector-type containers for the EDProduct. Where different structures are used for different subdetectors, the structures are the same for all common items, but different for any items which cannot be interpreted the same. This allows for templated generic programming to be used in many cases.

2.2 Digis

In all digis, the `sz` field indicates the number of samples actually present in the fixed size buffer. The `hcalPresamples` field indicates where within the buffer the triggered bunch crossing occurs. Both fields are likely to be same for all digis in a collection. They could be moved to the collection level. ROOT is likely to compress these values to nearly zero size on disk, however, so it is unlikely to have much impact even if the fields are in the struct.

2.2.1 Precision Readout Digis

```
struct HBHEDataFrame {
  HBHECellId cellId;
  uint32 samples[10];
  int sz;
  int hcalPresamples;
  uint32 hcalElectronicsId;
  uint32 hcalOpticsId;
};

struct HODataFrame {
  HOCellId cellId;
  uint32 samples[10];
  int sz;
  int hcalPresamples;
  uint32 hcalElectronicsId;
  uint32 hcalOpticsId;
};
```

```
struct HFDataFrame {
  HFCellId cellId;
  uint32 samples[6]; // never need 10 samples here
  int sz;
  int hcalPresamples;
  uint32 hcalElectronicsId;
  uint32 hcalOpticsId;
};
```

The electronics id and optics id information for HCAL is a new addition, intended to aid analysis of potential correlations between noise or response and parts of the detector chain. For HCAL the data which will be packed into these integers includes the RBX number, the RM number, the card and QIE number within the RM, the HPD pixel number, and potentially high-voltage and low-voltage circuit ids.

```
struct EBDataFrame {
  EBCellId cellId;
  uint32 samples[10];
  int sz;
};

struct EEDataFrame {
  EECellId cellId;
  uint32 samples[10];
  int sz;
};
```

2.2.2 Trigger Primitive Digis

```
struct HcalTrigPrimDigi {
  HcalTriggerCellId cellId;
  uint32 packedPrimitives[10]; // Et + fine-grain
  int sz;
  int hcalPresamples;
};

struct EcalTrigPrimDigi {
  EcalTriggerCellId cellId;
  uint32 packedPrimitive;
  uint32 fullPrecisionPrimitive; // if available from TCC
};
```

2.3 RecHit

2.3.1 Precision Readout RecHits

```
struct HBHERecHit {
  HBHECellId cellId;
  float energy, time;
};
```

```
struct HORecHit {
  HOCellId cellId;
  float energy, time;
};

struct HFRecHit {
  HFCellId cellId;
  float energy, time;
};
```

An UncalibratedRecHit has a generic ADC-to-GeV conversion applied, while a calibrated RecHit has channel-by-channel gains applied along with other corrections. The UncalibratedRecHits would have the ADC linearity correction applied (which could be channel-by-channel). The separation between UncalibratedRecHits and calibrated RecHits is made to ensure that algorithms do not accidentally operate on UncalibratedRecHits, which are intended for later re-reconstruction/correction passes.

The commented-out `nCellHits` and `iThisHit` fields are intended to be added if needed when the ECAL reconstruction is able to separate multiple RecHits (at different times) from the dataframe of a single cell. These fields would allow the identification of a cell where multiple hits occurred and index the multiple hits involved. Potentially, a hit with a special number could be added as well to represent the (lower-χ^2) reconstruction of the whole dataframe as a single hit.

```
struct EBUncalibratedRecHit {
  EBCellId cellId;
  float energy, time;
  float chi2;
  // uint8 nCellHits, iThisHit;
};

struct EEUncalibratedRecHit {
  EECellId cellId;
  float energy, time;
  float chi2;
  // uint8 nCellHits, iThisHit;
  EcalTriggerCellId towerId; // identify trigger tower (not in cellId for EE)
};

struct EBRecHit {
  EBCellId cellId;
  float energy, time;
  float chi2, deltaPedestal;
  // uint8 nCellHits, iThisHit;
};

struct EERecHit {
  EECellId cellId;
  float energy, time;
  float chi2, deltaPedestal;
```

```
  // uint8 nCellHits, iThisHit;
  EcalTriggerCellId towerId; // identify trigger tower (not in cellId for EE)
};
```

2.3.2 Trigger Primitive RecHits

```
struct HcalTrigPrim {
  HcalTriggerCellId cellId;
  float eT;
  bool fineGrain;
};

struct EcalTrigPrim {
  EcalTriggerCellId cellId;
  float eT;
  bool fineGrain;
};
```

2.4 Compound Objects (Tier 3)

2.4.1 HF Combined Hit

The simplest compound object is a combination of the long and short fiber data from HF to create a combined hit. Different algorithms may be developed over time to perform this combination. In this case, it is cheaper in terms of memory space to copy the energy and time information from the RecHits than to use a reference.

```
struct HFCombinedHit {
  HFCellId cellId; // depth == 2 for combined hit?
  float energy, time;
  float energyShort, timeShort; // short fibers
  float energyLong, timeLong; // long fibers
};
```

2.4.2 Calorimeter Towers

For many jet and MET algorithms, it is convenient to group together all the ECAL and HCAL hits which make up a projective tower. Such a Tier 3 object might look like this:

```
struct CaloTower {
  CaloTowerCellId cellId;
  float energy, energyEM, energyHad, energyOuter;
  float eT_nominal;
  float eta_nominal, phi_nominal;
  // references to constituents
  std::vector< EDM::Reference<EBRecHit> > ebCrystals;
  std::vector< EDM::Reference<EERecHit> > eeCrystals;
  std::vector< EDM::Reference<HBHERecHit> > hbheDepths;
  EDM::Reference<HORecHit> hoHit;
  EDM::Reference<HFCombinedHit> hfCombinedHit;
};
```

2.4.3 Clusters as Level 3 objects

Basic clusters could be considered either as electron-id objects or as general calorimeter objects.

3 Collections and Views

For storage efficiency and robustness, the preferred persistent format for the collections of Digis and RecHits will be simple vectors. The granularity of these vectors is not fully specified. The largest fraction of the detector in any single vector would be a complete subdetector (EB,EE,HB/HE,HO,HF) and the smallest would be the data from a single FED.

For processing which requires topological navigation, however, vectors are quite slow. To support such processing, we propose to provide View classes which can be inserted into the Event as separate EDProducts or included directly with the primary container EDProduct. The concept of a View is illustrated in Figure 1, for the case of the ECAL endcap.

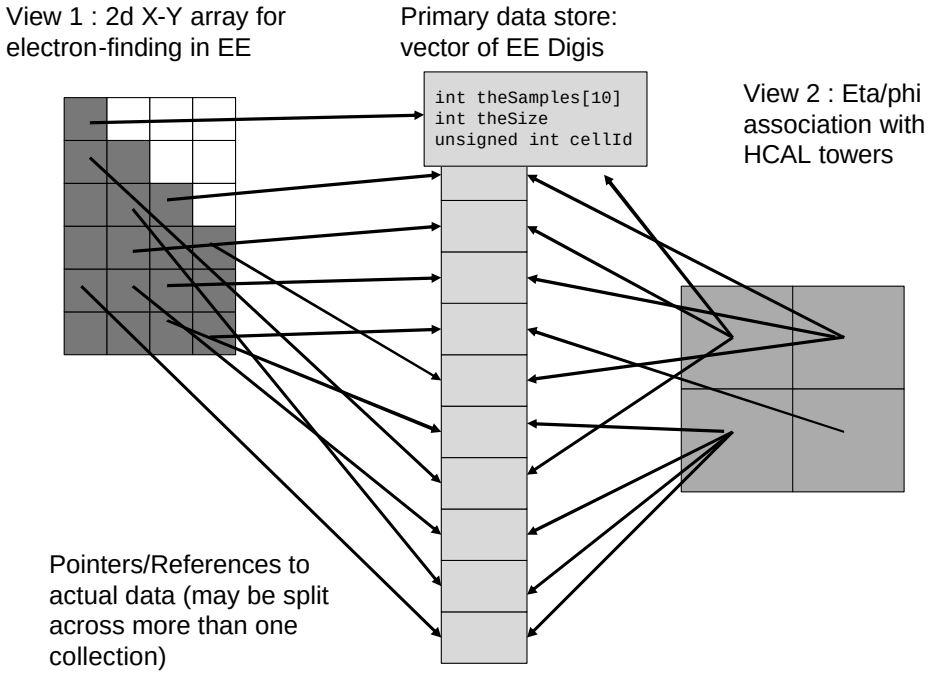

Figure 1: Concept of a view demonstrated for the case of the ECAL Endcap. The primary datastore is the vector in the center, while the left and right views contain only references, arranged in the most convienient manner for access.

Several Views are sketched out in the appendices.

4 Algorithms and Modules

It is important to keep the concepts of the algorithms and the modules seperate. Algorithms should not know about the EDM, but just the structs on which they need to work. Modules, on the other hand, should be very short pieces of code which operate simply to glue together algorithms, data products, and services. A

single algorithm could be used in several modules, depending on the inputs, outputs, and services (such as calibrations) needed.

Most algorithms can be generically named using the transformation which they apply.

Input	Output	Algorithm	Notes
FedRawData	DataFrame	CaloUnpacker	
DataFrame	FedRawData	CaloPacker	DAQ/Trigger/Hardware use
DataFrame	RecHit	CaloReconstructor	
RecHit	RecHit	CaloReReconstructor	For applying corrections later on

The Reconstructor algorithm does not need any knowledge of how the calibration information is obtained. The module which uses the algorithm will provided the calibration, generally from a database through the calibration Service. The Reconstructor algorithm should not use the Service directly. The calibration data is not event data, and could be cached by the calibration Service. Therefore, a simple CaloReconstructor could look like this:

```
struct HcalCalibrationInfo {
  float gain;
  float pedestal;
};

class HBHEReconstructor {
public:
  void reconstruct(const HcalDataFrame&, const HcalCalibrationInfo&, HBHERecHit&);
};
```

The module which wraps this reconstructor might look like:

The HcalCalibrationService would responsible for caching and quickly looking up the required calibration information.

A more complicated example would be the ECAL case, where several algorithms work together to reconstruct an event.

```
template<DataFrameClass>
class EcalLinearizer {
public
  void linearize(const DataFrameClass&, const EcalLinearizationData&, vector<float>& linearAdc);
};

typedef EcalLinearizer<EBDataFrame> EBLinearizer;
typedef EcalLinearizer<EEDataFrame> EELinearizer;

template<UncalibratedRecHitClass>
class EcalPulseAmplitudeMeasurer {
public
  void reconstruct(const vector<float>& linearAdc, UncalibratedRecHitClass&);
};

typedef EcalPulseAmplitudeMeasurer<EBUncalibratedRecHit> EBPulseAmplitudeMeasurer;
typedef EcalPulseAmplitudeMeasurer<EEUncalibratedRecHit> EEPulseAmplitudeMeasurer;
```

A Draft CellId Classes

```
class EBCellId {
public:
  EBCellId(int crystal_ieta, int crystal_iphi) throw (OutOfRangeException) {
    if (crystal_iphi<1 || crystal_iphi>360 || crystal_ieta<-85 || crystal_ieta>85 ||
crystal_ieta==0)
      throw new OutOfRangeException();
    if (crystal_ieta<0)
      ebCellId_=(EcalBarrel<<24)|((-crystal_ieta)<<9)|(crystal_iphi);
    else ebCellId_=(EcalBarrel<<24)|(1<<16)|(crystal_ieta<<9)|(crystal_iphi);
  }
  inline int zside() const { return (ebCellId_&0x10000)?(1):(-1); }
  inline int ieta() const {
    return (ebCellId_&0x10000)?((ebCellId_&0xFE00)>>9):(-((ebCellId_&0xFE00)>>9));
  }
  inline int iphi() const { return ebCellId_&0x7F; }
  inline int ieta_tower() const { return ((ieta()-zside())/5)+zside(); }
  inline int iphi_tower() const { return ((iphi()-1)/5)+1; }
private:
  uint32 ebCellId_;
};

class EECellId {
public:
  EECellId(int crystal_x, int crystal_y) throw (OutOfRangeException) {
    if (crystal_x<-100 || crystal_x>100 || crystal_x==0 ||
crystal_y<-100 || crystal_y>100 || crystal_y==0 || crystal_x*crystal_y<0)
      throw new OutOfRangeException();
    if (crystal_x<0) eeCellId_=(EcalEndcap<<24)|((-crystal_x)<<7)|(-crystal_y);
    else eeCellId_=(EcalEndcap<<24)|(1<<14)|(crystal_x<<7)|(crystal_y);
  }
  inline int x() const {
    return (eeCellId_&0x4000)?((eeCellId_&0x3F80)>>7):(-((eeCellId_&0x3F80)>>7));
  }
  inline int y() const {
    return (eeCellId_&0x4000)?(eeCellId_&0x7F):(-(eeCellId_&0x7F));
  }
private:
  uint32 eeCellId_;
};

class HcalCellId {
public:
  HcalCellId(CalDetector subdetector, int tower_ieta, int tower_iphi, int depth) throw (OutOfRangeException) {
    if (tower_iphi<1 || tower_iphi>72 || tower_ieta<-41 || tower_ieta>41 ||
tower_ieta==0 || depth<0 || depth>3)
      throw new OutOfRangeException(); // more tests possible with subdet-specific code
    if (tower_ieta<0)
      hcalCellId_=(subdetector<<24)|((-tower_ieta)<<7)|(tower_iphi);
    else hcalCellId_=(subdetector<<24)|(1<<13)|(tower_ieta<<7)|(tower_iphi);
  }
  inline int zside() const { return (hcalCellId_&0x2000)?(1):(-1); }
```

```
  inline int ieta() const {
    return (hcalCellId_&0x2000)?((hcalCellId_&0x1F80)>>7):(-((hcalCellId_&0x1F80)>>7));
  }
  inline int iphi() const { return hcalCellId_&0x7F; }
  inline int ieta_crystal_low() const {
    return ((hcalCellId_&0x1F80>(17<<7))?(0):((ieta()-zside())*5+zside()));
  }
  inline int ieta_crystal_high() const {
    return ((hcalCellId_&0x1F80>(17<<7))?(0):((ieta()-zside())*5+4*zside()));
  }
  inline int iphi_crystal_low() const {
    return ((hcalCellId_&0x1F80>(17<<7))?(0):((iphi()-1)*5+1));
  }
  inline int iphi_crystal_high() const {
    return ((hcalCellId_&0x1F80>(17<<7))?(0):((iphi()-1)*5+4));
  }

private:
  uint32 hcalCellId_;
};

typedef HcalCellId HBHECellId;
typedef HcalCellId HOCellId;
typedef HcalCellId HFCellId;
```

B Simple Endcap ECAL View

This draft ECAL Endcap view is very simple and does not require any geometry information for its creation. However, it is not appropriate for clustering near the EB/EE border.

```
template <class ViewedClass>
class EcalEndcapXYView {
public:
  EcalEndcapXYView() { clear(); }
  void addToView( EDVectorProduct<ViewedClass>& prod ) {
    for (EDVectorProduct<ViewedClass>::iterator i=prod.begin(); i!=prod.end(); i++)
      viewReferences_.push_back(EDM::Reference<ViewedClass>(prod,i));
  }
  const ViewedClass* get(int x, int y) const throw (OutOfRangeException) {
    if (x<-100 || x>100 || y<-100 || y>100 || x*y<=0)
      throw OutOfRangeException();
    return (x>0)?(viewPointers[0][x-1][y-1]):(viewPointers[1][-1-x][-1-y]);
  }
  void buildView() { // called when created and when "puffed" after loading
    std::vector< EDM::Reference<ViewedClass>::const iterator i;
    for (i=viewReferences_.begin(); i!=viewReferences.end(); i++) {
      const ViewedClass* c=*(*i);
      if (c.cellId.x()>0)
        viewPointers[0][c.cellId.x()-1][c.cellId.y()-1]=c;
      else
        viewPointers[1][-1-c.cellId.x()][-1-c.cellId.y()]=c;
    }
  }
private:
  void clear() {
    for (int i=0; i<2; i++)
      for (int j=0;j<100; j++)
        for (int k=0; k<100; k++)
          viewPointers_[i][j][k]=0;
  }
  std::vector< EDM::Reference<ViewedClass> > viewReferences_;
  const ViewedClass* viewPointers_[2][100][100]; // transient
};
```

C Comprehensive ECAL View

This draft ECAL View covers both the barrel and endcap and supports electron finding in the boundary region. This view could use geometry information to improve its use of nominal eta/phi locations.

```
template <class EBStruct, class EEStruct>
class WholeEcalView {
public:
  WholeEcalView() { clear(); }

  void addToView( EDVectorProduct<EBStruct>& prod ) {
    for (EDVectorProduct<EBStruct>::iterator i=prod.begin(); i!=prod.end(); i++)
      ebReferences_.push_back(EDM::Reference<EBStruct>(prod,i));
  }
  void addToView( EDVectorProduct<EEStruct>& prod ) {
    for (EDVectorProduct<EEStruct>::iterator i=prod.begin(); i!=prod.end(); i++)
      eeReferences_.push_back(EDM::Reference<EEStruct>(prod,i));
  }

  const EEStruct* getEE(int x, int y) const throw (OutOfRangeException) {
    if (x<-100 || x>100 || y<-100 || y>100 || x*y<=0)
      throw OutOfRangeException();
    return (x>0)?(eePointers_[0][x-1][y-1]):(ebPointers_[1][-1-x][-1-y]);
  }
  const EBStruct* getEB(int ieta, int iphi) const throw (OutOfRangeException) {
    if (ieta<-85 || ieta>=85 || ieta==0 || iphi<1 || iphi>360)
      throw OutOfRangeException();
    return ebPointers[ieta+85][phi];
  }
  void getInCone(const EBStruct* seed, double dR, vector<const EBStruct*>& eb,
 vector<const EEStruct*>& ee) {
    getInCone(nominalEta(seed),nominalPhi(seed),dR,eb,ee);
  }
  void getInCone(const EEStruct* seed, double dR, vector<const EBStruct*>& eb,
 vector<const EEStruct*>& ee) {
    getInCone(nominalEta(seed),nominalPhi(seed),dR,eb,ee);
  }
  void getInCone(double eta, double phi, double dR, vector<const EBStruct*>& eb,
 vector<const EEStruct*>& ee) {
    eb.clear(); ee.clear();

    // very, very simple-minded search.  Should be improved before real use.
    for (int ieta=-85; ieta<85; ieta++) {
      if (ieta==0) continue; // non existant eta=0
      for (int iphi=1; iphi<=360; iphi++) {
        const EBStruct* crys=ebPointers_[ieta+85][iphi-1];
        if (crys!=0 && toolbox::deltaR(eta,phi,nominalEta(crys),nominalPhi(crys))<dR)
          eb.push_back(crys);
      }
    }
    int ez=(eta>0)?(0):(1); // minor optimization..
    for (int x=1; x<=100; x++)
      for (int y=1; y<=100; y++) {
```

```
        const EEStruct* crys=eePointers_[ez][x-1][y-1];
        if (crys!=0 && toolbox::deltaR(eta,phi,nominalEta(crys),nominalPhi(crys))<dR)
          ee.push_back(crys);
      }
  }

  void buildView() { // called when created and when "puffed" after loading
    std::vector< EDM::Reference<EBStruct>::const iterator i;
    for (i=ebReferences_.begin(); i!=ebReferences.end(); i++) {
      const EBStruct* c=*(*i);
      ebPointers[c->ieta()+85][c->iphi()-1]=c;
    }
    std::vector< EDM::Reference<EEStruct>::const iterator j;
    for (j=eeReferences_.begin(); j!=eeReferences.end(); j++) {
      const EEStruct* c=*(*j);
      if (c.cellId.x()>0)
        eePointers[0][c.cellId.x()-1][c.cellId.y()-1]=c;
      else
        eePointers[1][-1-c.cellId.x()][-1-c.cellId.y()]=c;
    }
  }
private:
  float nominalEta(const EBStruct* ebs) {
    return ebs->cellId.ieta()*1.7453e-2f-ebs->zside()*8.72664e-3f;
  }
  float nominalPhi(const EBStruct* ebs) {
    return ebs->cellId.iphi()*1.7453e-2f-8.72664e-3f;
  }
  float nominalEta(const EEStruct* ees) {
    // requires some constants...
  }
  float nominalPhi(const EEStruct* ees) {
    return atan2(abs(ees->y())-50.5,abs(ees->x())-50.5);
  }

  void clear() {
    for (int i=0; i<85*2+1; i++)
      for (int j=0;j<360; j++)
        ebPointers_[i][j]=0;
    for (int i=0; i<2; i++)
      for (int j=0;j<100; j++)
        for (int k=0; k<100; k++)
          eePointers_[i][j][k]=0;
  }
  std::vector< EDM::Reference<EBStruct> > ebReferences_;
  std::vector< EDM::Reference<EEStruct> > eeReferences_;

  const EBStruct* ebPointers_[85*2+1][360]; // transient
  const EEStruct* eePointers_[2][100][100]; // transient

};
```