```
// File and version Information:
//   $Header: /nfs/slac/g/glast/ground/cvs/CalRecon/src/CalXtalRecAlg.cxx,v 1.7 2002/06/13
20:40:59 chehtman Exp $
//
// Description:
//    CalXtalRecAlg is an algorithm to reconstruct calorimeter
//    information in each individual crystal
//
// Author: A.Chekhtman



#include "CalXtalRecAlg.h"
#include "Event/Recon/CalRecon/CalXtalRecData.h"
#include "GaudiKernel/MsgStream.h"
#include "GaudiKernel/AlgFactory.h"
#include "GaudiKernel/IDataProviderSvc.h"
#include "GaudiKernel/SmartDataPtr.h"
#include "idents/VolumeIdentifier.h"
#include "CLHEP/Geometry/Transform3D.h"
#include "geometry/Point.h"
#include "Event/Digi/CalDigi.h"
#include "Event/TopLevel/EventModel.h"
#include <map>


static const AlgFactory<CalXtalRecAlg>  Factory;
const IAlgFactory& CalXtalRecAlgFactory = Factory;

using namespace Event;

// constructor
CalXtalRecAlg::CalXtalRecAlg(const std::string& name, ISvcLocator* pSvcLocator):
Algorithm(name, pSvcLocator) {

}


StatusCode CalXtalRecAlg::initialize()
Nice, clear explanation
// Purpose and method:
//         This function sets values to private data members,
//         representing the calorimeter geometry and digitization
//         constants. Information  from xml files is obtained using
//         GlastdetSvc::getNumericConstByName() function.
//         To make this constant extraction in a loop, the pairs
```

```
//          'constant pointer, constant name' are stored in
//          std::map container.
//          Double and integer constants are extracted separatly,
//          because constants of both types are returned
//          by getNumericConstByName() as double.
//


{
   MsgStream log(msgSvc(), name());
   StatusCode sc = StatusCode::SUCCESS;


   // extracting int constants
   double value;  // intermediate variable for reading constants from
            // xml file as doubles and converting them to interger
   typedef std::map<int*,std::string> PARAMAP;
   PARAMAP param; // map containing pointers to integer constants to be read
            // with their symbolic names from xml file used as a key

//    filling the map with information on constants to be read

   param[&m_xNum]=       std::string("xNum");
   param[&m_yNum]=       std::string("yNum");
   param[&m_eTowerCAL]=   std::string("eTowerCAL");
   param[&m_eLATTowers]=  std::string("eLATTowers");
   param[&m_CALnLayer]=   std::string("CALnLayer");
   param[&m_nCsIPerLayer]=std::string("nCsIPerLayer");
   param[&m_nCsISeg]= std::string("nCsISeg");
   param[&m_eXtal]=      std::string("eXtal");
   param[&m_eDiodeMSmall]=std::string("eDiodeMSmall");
   param[&m_eDiodePSmall]=std::string("eDiodePSmall");
   param[&m_eDiodeMLarge]=std::string("eDiodeMLarge");
   param[&m_eDiodePLarge]=std::string("eDiodePLarge");
   param[&m_eMeasureX]=std::string("eMeasureX");
   param[&m_eMeasureY]=std::string("eMeasureY");
   param[m_noise]=std::string("cal.noiseLarge");
   param[m_noise+1]=std::string("cal.noiseSmall");
   param[m_ePerMeV+1]=std::string("cal.ePerMeVSmall");
   param[m_ePerMeV]=std::string("cal.ePerMevLarge");
   param[&m_pedestal]=std::string("cal.pedestal");
   param[&m_maxAdc]=std::string("cal.maxAdcValue");
   param[&m_thresh]=std::string("cal.zeroSuppressEnergy");

   // now try to find the GlastDevSvc service
```

```
//    IGlastDetSvc* detSvc;
    sc = service("GlastDetSvc", detSvc);

        // loop over all constants information contained in the map
        for(PARAMAP::iterator it=param.begin(); it!=param.end();it++){

        //  attempt to get teh constant value via the method of GlastDetSvc
        if(!detSvc->getNumericConstByName((*it).second, &value)) {

            // if not successful - give the error message and return
            log << MSG::ERROR << " constant " <<(*it).second
                <<" not defined" << endreq;
            return StatusCode::FAILURE;

        //  if successful - fill the constant using the pointer from the map
        } else *((*it).first)=value;
    }

    // extracting double constants

    typedef std::map<double*,std::string> DPARAMAP;
    DPARAMAP dparam; // map containing pointers to double constants to be read
                    // with their symbolic names from xml file used as a key

    dparam[m_maxEnergy]=std::string("cal.maxResponse0");
    dparam[m_maxEnergy+1]=std::string("cal.maxResponse1");
    dparam[m_maxEnergy+2]=std::string("cal.maxResponse2");
    dparam[m_maxEnergy+3]=std::string("cal.maxResponse3");
    dparam[&m_lightAtt]=std::string("cal.lightAtt");
    dparam[&m_CsILength]=std::string("CsILength");

    for(DPARAMAP::iterator dit=dparam.begin(); dit!=dparam.end();dit++){
        if(!detSvc->getNumericConstByName((*dit).second,(*dit).first)) {
            log << MSG::ERROR << " constant " <<(*dit).second << " not defined" << endreq;
            return StatusCode::FAILURE;
        }
    }
```

A Global Utility would seem to be useful here

```
    // conversion from GeV to MeV
    for (int r=0; r<4;r++) m_maxEnergy[r] *= 1000.;


            return sc;
}
```

```
StatusCode CalXtalRecAlg::execute()

//  Purpose and method:
//    This function is called to do reconstruction
//    for all hitted crystals in one event.
//    It calls retrive() method to get access to input and output TDS data.
//    It iterates over all elements in CalDigiCol and calls
//    computeEnergy() and computePosition() methods doing real reconstruction.
//
//   TDS input:
//          CalDigiCol* m_CalDigiCol - private class member containing
//          a pointer to the calorimeter digi callection
Show full path in TDS (perhaps just the constant defined in EventModel
//
//   TDS output:
//          CalXtalRecCol* m_CalXtalRecCol - private class member containing
//          a pointer to the calorimeter crystal reconstructed data collection
//
//

{
        StatusCode sc = StatusCode::SUCCESS;
      MsgStream log(msgSvc(), name());

      sc = retrieve(); //get access to TDS data collections

Should check the  StatusCode from retrieve before continuing

      // loop over all calorimeter digis in CalDigiCol
         for (Event::CalDigiCol::const_iterator it = m_CalDigiCol->begin();
        it != m_CalDigiCol->end(); it++) {
        idents::CalXtalId xtalId = (*it)->getPackedId();

        // create new object to store crystal reconstructed data
        Event::CalXtalRecData* recData =
              new Event::CalXtalRecData((*it)->getMode(),xtalId);


        // calculate energy in the crystal
           computeEnergy(recData, *it);

        // calculate position in the crystal
           computePosition(recData);

        // add new reconstructed data to the collection
```

```
        m_CalXtalRecCol->push_back(recData);
         }

         return sc;
}


StatusCode CalXtalRecAlg::finalize()
// empty function: required by base class (Algorithm)
{
         StatusCode sc = StatusCode::SUCCESS;
         return sc;
}



StatusCode CalXtalRecAlg::retrieve()

// Purpose and method:
//           This function provides access to the TDS input and output data
//           by setting the private data members m_CalDigiCol
//           and m_CalXtalRecCol
//
//   TDS input: CalDigiCol
//   TDS output: CalXtalrecCol
//
//

{

   MsgStream log(msgSvc(), name());
   StatusCode sc = StatusCode::SUCCESS;

   m_CalXtalRecCol = 0;

   //  create output data collection
   m_CalXtalRecCol = new CalXtalRecCol;

   DataObject* pnode=0;

   // search for CalRecon section of Event directory in TDS
   sc = eventSvc()->retrieveObject( EventModel::CalRecon::Event, pnode );

   // if the required directory doesn't exist - create it
   if( sc.isFailure() ) {
      sc = eventSvc()->registerObject( EventModel::CalRecon::Event,
```

```
                                new DataObject);
        if( sc.isFailure() ) {

            // if cannot create the directory - write an error message
            log << MSG::ERROR << "Could not create CalRecon directory"
                << endreq;
            return sc;
        }
    }


    // get a pointer to the input TDS data collection
    m_CalDigiCol = SmartDataPtr<Event::CalDigiCol>(eventSvc(),
                            EventModel::Digi::CalDigiCol);

    //register output data collection as a TDS object
    sc = eventSvc()->registerObject(EventModel::CalRecon::CalXtalRecCol,
                                          m_CalXtalRecCol);
    return sc;
}




void CalXtalRecAlg::computeEnergy(CalXtalRecData* recData, const Event::CalDigi* digi)

//   Purpose and method:
//              This function calculates the energy for one crystal.
//              It makes a loop over all readout ranges (1 or 4, (BESTRANGE or ALLRANGE)
depending
//              on readout mode), and converts adc values for both crystal
//              faces into energy, using constants contained in private data
//              members m_maxEnergy, m_pedestal, m_maxAdc.
//
//
//  Input: CalDigi* digi - pointer to digitized calorimeter data for
//                      one crystal
//
//  Output: CalXtalRecData* recData - pointer to reconstructed data for
//                                this crystal



{
    MsgStream log(msgSvc(), name());
```

```
    const Event::CalDigi::CalXtalReadoutCol& readoutCol = digi->getReadoutCol();

    // loop over readout ranges
    for ( Event::CalDigi::CalXtalReadoutCol::const_iterator it = readoutCol.begin();
                    it !=readoutCol.end(); it++){

        // get readout range number for both crystal faces
        int rangeP = it->getRange(idents::CalXtalId::POS);
        int rangeM = it->getRange(idents::CalXtalId::NEG);

        // get adc values
        double adcP = it->getAdc(idents::CalXtalId::POS);
        double adcM = it->getAdc(idents::CalXtalId::NEG);

        // convert adc values into energy
        double eneP = m_maxEnergy[rangeP]*(adcP-m_pedestal)/(m_maxAdc-m_pedestal);
        double eneM = m_maxEnergy[rangeM]*(adcM-m_pedestal)/(m_maxAdc-m_pedestal);


        // create output object
        CalXtalRecData::CalRangeRecData* rangeRec =
            new CalXtalRecData::CalRangeRecData(rangeP,eneP,rangeM,eneM);

        // add output object to output collection
        recData->addRangeRecData(*rangeRec);
    }

}


void CalXtalRecAlg::computePosition(CalXtalRecData* recData)

// Purpose and method:
//             This function calculates the longitudinal position
//             for each crystal from light asymmetry.
//
//  Input: CalXtalRecData* recData - pointer to the reconstructed crystal data
//  Output: the same object, the calculated position is stored using
//            public function SetPosition()



{
        MsgStream log(msgSvc(), name());
```

```
// get crystal identification
idents::CalXtalId xtalId = recData->getPackedId();

// unpack crystal identification into tower, layer and column number
   int layer = xtalId.getLayer();
   int tower = xtalId.getTower();
   int col   = xtalId.getColumn();

// create Volume Identifier for segment 0 of this crystal
   idents::VolumeIdentifier segm0Id;
   segm0Id.append(m_eLATTowers);
   segm0Id.append(tower/m_xNum);
   segm0Id.append(tower%m_xNum);
   segm0Id.append(m_eTowerCAL);
   segm0Id.append(layer);
   segm0Id.append(layer%2);
segm0Id.append(col);
   segm0Id.append(m_eXtal);
   segm0Id.append(0);

HepTransform3D transf;

//get 3D transformation for segment 0 of this crystal
   detSvc->getTransform3DByID(segm0Id,&transf);

//get position of the center of the segment 0
Vector vect0 = transf.getTranslation();


// create Volume Identifier for the last segment of this crystal
idents::VolumeIdentifier segm11Id;

// copy all fields from segm0Id, except segment number
   for(int ifield = 0; ifield<fSegment; ifield++)segm11Id.append(segm0Id[ifield]);
   segm11Id.append(m_nCsISeg-1); // set segment number for the last segment

   //get 3D transformation for the last segment of this crystal
detSvc->getTransform3DByID(segm11Id,&transf);

//get position of the center of the last segment
   Vector vect11 = transf.getTranslation();

   Point p0(0.,0.,0.);

// position of the crystal center
```

```
    Point pCenter = p0+(vect0+vect11)*0.5;

  //normalized vector of the crystal direction
  Vector dirXtal = 0.5*(vect11-vect0)*m_nCsISeg/(m_nCsISeg-1);


  //extract energy for the best range for both crystal faces
    double eneNeg = recData->getEnergy(0,idents::CalXtalId::NEG);
    double enePos = recData->getEnergy(0,idents::CalXtalId::POS);
    double asym=0;

  //calculate light asymmetry
  if(enePos>0 && eneNeg>0)asym = (enePos-eneNeg)/(enePos+eneNeg);

  // calculate slope - coefficient to convert light asymmetry into the
  // deviation from crystal center
    double slope = (1+m_lightAtt)/(1-m_lightAtt);

  // calculate average position of energy deposit in this crystal
    Point pXtal = pCenter+dirXtal*asym*slope;


  // store calculated position in the reconstructed data
  // for the best readout range
  (recData->getRangeRecData(0))->setPosition(pXtal);


}
```