```
#include "CalDigiAlg.h"

/// Gaudi specific include files
#include "GaudiKernel/MsgStream.h"
#include "GaudiKernel/AlgFactory.h"
#include "GaudiKernel/IDataProviderSvc.h"
#include "GaudiKernel/SmartDataPtr.h"

/// Glast specific includes
#include "Event/TopLevel/EventModel.h"
#include "GaudiKernel/ObjectVector.h"

// MC classes
#include "Event/MonteCarlo/McIntegratingHit.h"

#include "Event/Digi/CalDigi.h"
#include "CLHEP/Random/RandGauss.h"

/// for min and floor functions
#include <algorithm>
#include <math.h>

// std stuff
#include <map>
#include <string>

// Define the factory for this algorithm
static const AlgFactory<CalDigiAlg>  Factory;
const IAlgFactory& CalDigiAlgFactory = Factory;

// Algorithm parameters which can be set at run time must be declared.
// This should be done in the constructor.

CalDigiAlg::CalDigiAlg(const std::string& name, ISvcLocator* pSvcLocator) :
Algorithm(name, pSvcLocator) {

   // Declare the properties that may be set in the job options file
}


StatusCode CalDigiAlg::initialize() {
   // Purpose and Method: initialize the algorithm. Set up parameters from detModel
   // Inputs: detModel parameters

   MsgStream log(msgSvc(), name());
   log << MSG::INFO << "initialize" << endreq;
```

```
// extracting int constants from detModel. Store into local cache - member variables.

double value;
typedef std::map<int*,std::string> PARAMAP;
PARAMAP param;
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");

// now try to find the GlastDevSvc service

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


for(PARAMAP::iterator it=param.begin(); it!=param.end();it++){
   if(!detSvc->getNumericConstByName((*it).second, &value)) {
      log << MSG::ERROR << " constant " <<(*it).second <<" not defined" << endreq;
      return StatusCode::FAILURE;
   } else *((*it).first)=value;
}

int nTowers = m_xNum * m_yNum;

// extracting double constants from detModel. Store into local cache - member variables.

typedef std::map<double*,std::string> DPARAMAP;
DPARAMAP dparam;
```

```
    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");
    dparam[&m_thresh]=std::string("cal.zeroSuppressEnergy");

    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;
       }
    }

    Perhaps we should have a global utility for such conversions
    // scale max energies and thresholds from GeV to MeV
    for (int r=0; r<4;r++) m_maxEnergy[r] *= 1000.;
    m_thresh *= 1000.;

    return StatusCode::SUCCESS;
}


StatusCode CalDigiAlg::execute() {
    // Purpose and Method: take Hits from McIntegratingHit and perform the following steps:
    //  for deposit in a crystal segment, take into account light propagation to the two ends and
    // apply light taper based on position along the length.
    //  keep track of direct deposit in the diode.
    // Then combine diode (with appropriate scale factor) and crystal deposits and add noise.
    // Then, add noise to 'unhit' crystals.
    //  Finally, convert to ADC units and pick the range for hits above threshold.
    // TDS Inputs: Event::MC::McIntegratingHit
    // TDS Output:


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

    // get McIntegratingHit collection. Abort if empty.

    SmartDataPtr<Event::McIntegratingHitVector>
McCalHits(eventSvc(),EventModel::MC::McIntegratingHitCol );
//"/Event/MC/IntegratingHitsCol");

    if (McCalHits == 0) {
```

```
        log << MSG::DEBUG << "no calorimeter hits found" << endreq;
        return sc;
    }

    //Take care of insuring that data area has been created
    DataObject* pNode = 0;
    sc = eventSvc()->retrieveObject( EventModel::Digi::Event /*"/Event/Digi"*/, pNode);

    if (sc.isFailure()) {
        sc = eventSvc()->registerObject(EventModel::Digi::Event /*"/Event/Digi"*/,new
DataObject);
        if( sc.isFailure() ) {
            log << MSG::ERROR << "could not register " << EventModel::Digi::Event /*<<
/Event/Digi "*/ << endreq;
            return sc;
        }
    }



    //  clear signal array: map relating xtal signal to id. Map holds diode and crystal responses
    //  separately during accumulation.

    typedef std::map<idents::CalXtalId,XtalSignal> SignalMap;
    SignalMap signalMap;
    I find it confusing to have a variable name that only differs by case from the type name

    // loop over hits - pick out CAL hits

    for (Event::McIntegratingHitVector::const_iterator it = McCalHits->begin(); it != McCalHits-
>end(); it++) {


        //   extracting hit parameters - get energy and first moment
        Maybe we can explain in the Doxygen comments (or the comments in this method how we
think the volumeIds are organized as far as the CAL is concerned.
        idents::VolumeIdentifier volId = ((idents::VolumeIdentifier)(*it)->volumeID());
        double ene = (*it)->totalEnergy();
        HepPoint3D mom1 = (*it)->moment1();

        //   extracting parameters from volume Id identifying as in CAL

        if (volId[fLATObjects] == m_eLatTowers &&
            volId[fTowerObjects] == m_eTowerCal){

            int col = volId[fCALXtal];
```

```
int layer = volId[fLayer];
int towy = volId[fTowerY];
int towx = volId[fTowerX];
int tower = m_xNum*towy+towx;

idents::CalXtalId mapId(tower,layer,col);
XtalSignal& xtalSignalRef = signalMap[mapId];



if(volId[fCellCmp] == m_eDiodeMLarge)
   xtalSignalRef.addDiodeEnergy(ene,0);

else if(volId[fCellCmp] == m_eDiodePLarge)
   xtalSignalRef.addDiodeEnergy(ene,1);

else if(volId[fCellCmp] == m_eDiodeMSmall )
   xtalSignalRef.addDiodeEnergy(ene,2);

else if(volId[fCellCmp] ==    m_eDiodePSmall )
   xtalSignalRef.addDiodeEnergy(ene,3);

else if(volId[fCellCmp] ==  m_eXtal ){

   int segm = volId[fSegment];

   // let's define the position of the segment along the crystal

   double relpos = (segm+0.5)/m_nCsISeg;


   // in local reference system x is always oriented along the crystal
   double dpos =  mom1.x();

   // to correct for local reference system orientation in Y layers
   if (volId[fMeasure] == m_eMeasureY) dpos = -dpos;
   relpos += dpos/m_CsILength;



   // take into account light tapering

   double norm = 0.5+0.5*m_lightAtt; // light tapering in the center of crystal
(relpos=0.5)
Make it clear which is POS and which is NEG face
   double s2 = ene*(1-relpos*(1-m_lightAtt))/norm;
```

```
            double s1 = ene*(1-(1-relpos)*(1-m_lightAtt))/norm;

            // set up a XtalMap to act as key for the map - if there is no entry, add
            // add one, otherwise add signal to existing map element.


            xtalSignalRef.addSignal(s1,s2);



        }
    }
}


const    double ePerMeVinDiode = 1e6/3.6;
   // number of electrons per MeV of direct enegy deposition in a diode
   //  3.6 eV - energy to create 1 electron in silicon
   //  this is temporary - constant should be moved probably to xml file
Add this to the todo list?

   // add electronic noise to the diode response and add to the crystal
   // response. The diodeEnergy becomes the readout source.

   for(SignalMap::iterator mit=signalMap.begin(); mit!=signalMap.end();mit++){
      XtalSignal* xtal_signal = &(*mit).second;

      for ( int idiode =0; idiode < 4; idiode++){
         int diode_type = idiode/2;
         int face  = idiode%2;
         double signal = xtal_signal->getSignal(face);

         double diode_ene = xtal_signal->getDiodeEnergy(idiode);


         // convert energy deposition in a diode to
         // the equivalent energy in a crystal
         diode_ene *= ePerMeVinDiode/m_ePerMeV[diode_type];

         // add crystal signal - now diode energy contains
         // the signal at given diode in energy units
         // (equivalent energy deposition at the crystal center)
         diode_ene += signal;

         // add poissonic fluctuations in the number of electrons in a diode
         diode_ene += sqrt(diode_ene/m_ePerMeV[diode_type])*RandGauss::shoot();
```

```
            // add electronic noise
            diode_ene += RandGauss::shoot()*m_noise[diode_type]/m_ePerMeV[diode_type];

            //store modified diode signal in the signal map
            xtal_signal->setDiodeEnergy(diode_ene,idiode);

        }
    }

    //  adding electronic noise to the channels without signal,
    //      storing it in the signal map if  one of crystal faces
    //      has the noise above the threshold

    double noise_MeV = double(m_noise[0])/double(m_ePerMeV[0]); // noise in MeV for the
large diode
    for (int tower = 0; tower < m_xNum*m_yNum; tower++){
        for (int layer = 0; layer < m_CalNLayer; layer++){
            for (int col = 0; col < m_nCsIPerLayer; col++){
                double eneM = noise_MeV*RandGauss::shoot();
                double eneP = noise_MeV*RandGauss::shoot();
                idents::CalXtalId mapId(tower,layer,col);
                if((eneM > m_thresh || eneP > m_thresh) &&
                    signalMap.find(mapId) == signalMap.end()){
                    XtalSignal& xtalSignalRef = signalMap[mapId];
                    xtalSignalRef.addDiodeEnergy(eneM,0);
                    xtalSignalRef.addDiodeEnergy(eneP,1);
                }
            }
        }
    }
    /*
    log << MSG::DEBUG << signalMap.size() << "calorimeter hits in signalMap" << endreq;
    for( mit=signalMap.begin(); mit!=signalMap.end();mit++){
    log << MSG::DEBUG << " id " << (*mit).first
    << " s0=" << (*mit).second.getSignal(0)
    << " s1=" << (*mit).second.getSignal(1)
    << " d0=" << (*mit).second.getDiodeEnergy(0)
    << " d1=" << (*mit).second.getDiodeEnergy(1)
    << " d2=" << (*mit).second.getDiodeEnergy(2)
    << " d3=" << (*mit).second.getDiodeEnergy(3)
    << endreq;
    }
    */

    Event::CalDigiCol* digiCol = new Event::CalDigiCol;
```

```
// iterate through the SignalMap to look at only those Xtals that had energy deposited.
// if either side is above threshold, then select the appropriate ADC range and
// create a readout

for(SignalMap::iterator nit=signalMap.begin(); nit!=signalMap.end();nit++){
   XtalSignal* signal = &(*nit).second;

   if(signal->getDiodeEnergy(0) > m_thresh
      || signal->getDiodeEnergy(1) > m_thresh) {

      idents::CalXtalId xtalId = (*nit).first;

      char rangeP,rangeM;
      unsigned short adcP,adcM;

      // loop over the plus and minus faces of the crystal
      Can we use the CalXtalId::XtalFace values for the loop?
      for (int face=0; face<2; face++){
         double resp;
         int br;

         // loop over the diodes and fetch the response
         for (int r = 0;r<4;r++){
            int diode_type = r/2;
            int diode = 2*diode_type+face;
            resp = signal->getDiodeEnergy(diode);


            br=r;
            if( resp < m_maxEnergy[r]) break;

         }

         // set readout to rail if saturated

         if(resp > m_maxEnergy[br])resp = m_maxEnergy[br];

         // convert energy to ADC units here

         unsigned short adc = (resp/m_maxEnergy[br])*(m_maxAdc-m_pedestal)+m_pedestal;

         // assign the plus/minus readouts

         if(face == idents::CalXtalId::POS){ adcP=adc; rangeP=br;}
         else { adcM=adc; rangeM=br;}
```

```
            }
            /*
            log << MSG::DEBUG <<" id=" << xtalId
            << " rangeP=" << int(rangeP) << " adcP=" << adcP
            << " rangeM=" << int(rangeM) << " adcM=" << adcM << endreq;
            */
            Event::CalDigi::CalXtalReadout read = Event::CalDigi::CalXtalReadout(rangeP, adcP,
rangeM, adcM);
            Event::CalDigi* curDigi = new Event::CalDigi(idents::CalXtalId::BESTRANGE,
xtalId);
            curDigi->addReadout(read);
            digiCol->push_back(curDigi);
        }
    }

    sc = eventSvc()->registerObject(EventModel::Digi::CalDigiCol, digiCol);


    return sc;
}

StatusCode CalDigiAlg::finalize() {

    MsgStream log(msgSvc(), name());
    log << MSG::INFO << "finalize" << endreq;

    return StatusCode::SUCCESS;
}
CalDigiAlg::XtalSignal::XtalSignal() {
    // Purpose and Method: default constructor setting signals to zero
    m_signal[0] = 0;
    m_signal[1] = 0;

    for(int i=0; i<4; i++)m_Diodes_Energy.push_back(0.);

}

CalDigiAlg::XtalSignal::XtalSignal(double s1, double s2) {
    // Purpose and Method: constructor setting signals to those input
    m_signal[0] = s1;
    m_signal[1] = s2;

    for(int i=0; i<4; i++)m_Diodes_Energy.push_back(0.);

}
void CalDigiAlg::XtalSignal::addSignal(double s1, double s2) {
```

```
    // Purpose and Method: add signals s1, s2 to already existing signals.
    m_signal[0] += s1;
    m_signal[1] += s2;
    return;
}
```