```
#ifndef CalRecon_CalXtalRecData_H
#define CalRecon_CalXtalRecData_H 1

#include <iostream>
#include <vector>
#include "idents/CalXtalId.h"

#include "GaudiKernel/Kernel.h"
#include "GaudiKernel/StreamBuffer.h"

#include "GaudiKernel/ObjectVector.h"
#include "GaudiKernel/ContainedObject.h"
#include "Event/TopLevel/Definitions.h"
#include "geometry/Point.h"



extern const CLID& CLID_CalXtalRecData;

namespace Event
{

/**
* @class   CalXtalRecData
*
* @brief TDS class containing reconstructed data for
* one calorimeter crystal
*
* This class stores the reconstructed data -in the vector
* of CalRangeRecData objects (data member m_recData).
* This vector contains 1 or 4 elements,
* depending on- readout mode defined by private data member
* m_mode (BESTRANGE or ALLRANGE). Defined in CalXtalId
* The first element of the vector always contains information
* for the best range, as it was defined during readout process.
*
* Data member m_xtalId contains crystal identification.
*
* ObjectVector of CalXtalRecData is typedefed as CalXtalRecCol
* and used to store crystal reconstructed data for whole calorimeter.
*
* @author  A.Chekhtman
*
* $Header: /nfs/slac/g/glast/ground/cvs/Event/Event/Recon/CalRecon/CalXtalRecData.h,v 1.5
2002/06/12 00:04:40 chehtman Exp $
```

```
*/
   class CalXtalRecData : virtual public ContainedObject {

   public:

   /** @class   CalRangeRecData
   *
   * @brief   This is tehe nested class in CalXtalRecData,
   *          holding reconstructed data for one readout range.
   *
   *          The class contains reconstructed position
   *          and for both POSitive and NEGative faces of a crystal
   *          the reconstructed energy and the range number used
   *          to obtain this energy.
   *
   *
   * @author  A.Chekhtman
   */
      class CalRangeRecData {

      public:

         /// constructor initializing energies and ranges
         /// and setting position to zero.
         CalRangeRecData(char rangeP,
                         double eneP,
                         char rangeM,
                         double eneM) :
            m_rangeP(rangeP),
            m_eneP(eneP),
            m_rangeM(rangeM),
            m_eneM(eneM),
            m_pos(Point(0.,0.,0.))
          {};

         ~CalRangeRecData() {};

         /// The separate setting function for reconstructed position
         /// is needed, because position reconstruction is performed
         /// later, than energy reconstruction and is based
         /// on reconstructed energies
         void setPosition (Point pos) { m_pos = pos;}
Could we use a single initialize routine instead that sets all data members?

         /// retrieve position value
         const Point& getPosition() const { return m_pos;}
```

```
    /// retrieve energy from specified face
    inline double getEnergy(idents::CalXtalId::XtalFace face) const
    {return face == idents::CalXtalId::POS ? m_eneP : m_eneM;}

    /// retrieve energy range from specified face
    inline char getRange(idents::CalXtalId::XtalFace face) const
    {return face == idents::CalXtalId::POS ? m_rangeP : m_rangeM;}


private:


  /// reconstructed energy for POSitive face
  double m_eneP;

  /// reconstructed energy for NEGative face
  double m_eneM;

  /// reconstructed position
  Point m_pos;

  /// energy range for POSitive face
  char m_rangeP;

  /// energy range for NEGative face
  char m_rangeM;

};

/// default constructor
CalXtalRecData() {};

/// constructor with parameters initializing crystal
/// identification and readout mode
CalXtalRecData(idents::CalXtalId::CalTrigMode mode,
               idents::CalXtalId CalXtalId) :
               m_mode(mode), m_xtalId(CalXtalId){};

virtual ~CalXtalRecData() { };

/// function initializing crystal identification and readout mode
void initialize (idents::CalXtalId::CalTrigMode m,
                 idents::CalXtalId id)
{m_mode = m; m_xtalId = id; }
```

```
/// Retrieve readout mode
inline const idents::CalXtalId::CalTrigMode getMode() const
{ return m_mode; };

/// Retrieve crystal identifier
inline const idents::CalXtalId getPackedId() const
{ return m_xtalId; };

/// Add one more readout range to the reconstructed data vector
inline void addRangeRecData(CalRangeRecData r) { m_recData.push_back(r); }

/// Retrieve energy range for selected face and readout
/// returns -1 if readout with requested index doesn't exist
inline char getRange(short readoutIndex,
                     idents::CalXtalId::XtalFace face) const
{
   return (readoutIndex < m_recData.size()) ?
         ((m_recData[readoutIndex])).getRange(face) : (char)-1;
}

/// Retrieve energy for selected face and readout
/// returns -1 if readout with requested index doesn't exist
inline double getEnergy(short readoutIndex,
                        idents::CalXtalId::XtalFace face) const
{
   return (readoutIndex < m_recData.size()) ?
         ((m_recData[readoutIndex])).getEnergy(face) : (short)-1;
}


/// Retrieve average energy of two faces for the best range
inline double getEnergy()
{
   return (getEnergy(0,idents::CalXtalId::POS)
         +getEnergy(0,idents::CalXtalId::NEG))/2;
}

/// Retrieve average energy of two faces for the best range
/// (for const objects)
inline double getEnergy() const
{
   return (getEnergy(0,idents::CalXtalId::POS)
         +getEnergy(0,idents::CalXtalId::NEG))/2;
}

/// Retrieve the position for the best range
```

```
inline Point getPosition()
{
   return getRangeRecData(0)->getPosition();
}

/// Retrieve the position for the best range (for const objects)
inline const Point& getPosition() const
{
   return getRangeRecData(0)->getPosition();
}

/// Retrieve reconstructed data from both ends of selected readout
inline CalRangeRecData* getRangeRecData(short readoutIndex)
{
   if ( readoutIndex < m_recData.size() )
      return &(m_recData[readoutIndex]);
   else
      return 0;

}
/// Retrieve reconstructed data from both ends of selected readout
/// (for const objects)
inline const CalRangeRecData* getRangeRecData(short readoutIndex) const
{
   if ( readoutIndex < m_recData.size() )
      return &(m_recData[readoutIndex]);
   else
      return 0;

}

/// Retrieve energy from selected range and face
inline double getEnergySelectedRange(charidents::CalXtalId::AdcRange range,
            idents::CalXtalId::XtalFace face) const
{

   // get number of ranges as the size of m_recData vector
   char nRanges = (char)m_recData.size();

   // if there is only one range
   if (nRanges == 1)

      // and the range number corresponds to what is requested
      // return energy,
      // otherwise return -1
      return (range == ((m_recData[0])).getRange(face)) ?
```

```
                ((m_recData[0])).getEnergy(face) : -1.0;

        // if there are 4 ranges - they are stored in increasing order,
        // starting with best range and returning to 0 after 3
        else
            // calculate the index for requested range and return the energy
            return ((m_recData[(nRanges + range -
            ((m_recData[0])).getRange(face)) % nRanges])).getEnergy(face);
    }

    friend std::ostream& operator << (std::ostream& s, const CalXtalRecData& obj)
    {
        return obj.fillStream(s);
    };

    /// Fill the ASCII output stream
    virtual std::ostream& fillStream( std::ostream& s ) const;

private:

    /// Cal readout mode is based on trigger type
    idents::CalXtalId::CalTrigMode m_mode;

    /// crystal ID
    idents::CalXtalId m_xtalId;

    /// ranges and pulse heights
    std::vector<CalRangeRecData> m_recData;

};

typedef ObjectVector<CalXtalRecData> CalXtalRecCol;

}


#endif
```