Specification For Offline Interface to DAQ FormatReader

Author: Jeff Landgraf (jml@bnl.gov)

Defining The Event

The user of this program is responsible for using the TAG database to locate the event. The event can be either stored in a file or directly in memory. If the event is stored in a file, the DAQ format reader requires an open file descriptor to the file along with the offset to the event's DATAP bank. If the event is stored in memory, the DAQ format reader requires a pointer to the event's DATAP bank.

The first step for using the Format Reader is to select the event. This is accomplished by creating an *EventReader* object:

```
      EventReader *er = getEventReader(fd, offset);
 (or) EventReader *er = getEventReader(&event);
```

The event reader is treated like a file descriptor by the calling program. Creating the Event Reader is analogous to opening the event.

If a file the *EventReader* is created using a file then it is possible specify whether the file is read into memory all at once, or read piecemeal. The default is to read the data from the file as it is needed. To create an *EventBuilder* that reads the entire event into a memory buffer at creation use:

```
      EventReader *er = getEventReader(fd, offset, 0);
```

From the perspective of the format reader, the *EventReader* object provides all of the i/o routines, and eventually, will provide support for saving and managing buffers of already formatted data.

Defining The Detector

The second step is to select the detector by obtaining a *DetectorReader* object. The *DetectorReader* class is a pure virtual class from which readers for each different detector are derived. Each *DetectorReader* implementation will contain exactly the same interface. The function of this interface is to create *SectorReader* classes that read the actual data. The user creates the *DetectorReader* as follows (for the TPC version 1.0 reader):

```
      DetectorReader *dr = getDetectorReader(er, “TPCV1.0”);
```

dr is now a pointer to a *TPCV1P0Reader* object which is specifically implemented for the TPC.

The *DetectorReader* object has two roles. First, it contains factories that produce all of the varied sector readers. Second, it will eventually provide support for saving and managing the formatted sector data.

Defining A Sector Reader

The third step is to get a reader object for the sector you wish to address. There is a different reader object for each type of data that can be requested. The different sector reader objects are created using methods from the *DetectorReader*:

```
class DetectorReader
{
  virtual ZeroSuppressedReader *getZeroSuppressedReader(int sector)=0;
  virtual ADCRawReader *getADCRawReader(int sector)=0;
  virtual PedestalReader *getPedestalReader(int sector)=0;
  virtual PedestalRMSReader *getPedestalRMSReader(int sector)=0;
  virtual GainReader *getGainReader(int sector)=0;
  virtual CPPReader *getCPPReader(int sector)=0;
  virtual BadChannelReader *getBadChannelReader(int sector)=0;
  virtual ConfigReader *getConfigReader(int sector)=0;
}
```

If a reader object is requested for a data type that does not exist in the event, the request returns the NULL pointer and the detector's errno(), and errstr() function will return one of:

BANK_NOT_AVAILABLE	-	No data of this type in the sector
BANK_NOT_IMPLEMENTED	-	This data type has not been implemented for this detector
BANK_CORRUPT	-	This bank is corrupt in the data file
NO_MEMORY	-	The buffer's spaces could not be allocated

For example, to obtain a zero suppressed data reader for sector N, one executes the following command:

```
ZeroSuppressedReader *zsr = dr->getZeroSuppressedReader(N);
if(!zsr)
{
  cout << dr->errstr() << endl;

  . . .
}
```

The creation of the sector reader object is the expensive step in reading data. During this step, any physical coordinate based indexes are read and populated, any necessary byte swapping or uncompressing is done, and disk based data is stored in a memory buffer.

The *SectorReader* object is a pure virtual class. The specific implementations will be highly dependent upon detector and possibly on the raw format versions. Only the *DetectorReader* implementation knows which implementation of the *SectorReader* object is actually created.

Eventually, if performance becomes an issue, these buffers will be stored and managed by the detector reader and event reader objects through the *InformBuffers()* and *AttachBuffers()* function calls. When that occurs, the second time a sector reader is created, the sector readers will attach to the existing buffers, saving this overhead. The buffers are then destroyed when the *DetectorReader* and *EventReader* objects are deleted. To support this eventuality, each implementation should call attempt *AttachBuffers()* before allocating and reading the file. If attach buffers is successful, the reading/allocating need not be done. Likewise, after the buffers have been read, *InformBuffers()* should be called. For the first version, these functions will simply return false, doing nothing.

The buffer lifetime for the sector readers:

The specific SectorReaders are described in the next section. In general each reader has functions which provide pointers to buffers that are managed by the format reader program. The lifetime of the buffers is the same as the lifetime of the reader object that produced the pointer.

For example, the full code to read the zero suppressed sequences from a few specific pads follows,

```
// Assume event in open file fd at offset offset
EventReader *er = getEventReader(fd, offset);
if(!er) { // handle error };

DetectorReader *dr = getDetectorReader(er, “TPCV1.0”);
if(!dr) { // handle error };

ZeroSuppressedReader *zsr = dr->getZeroSuppressedReader(sector);
if(!zsr) { // handle error };

int nSeq, nSeq2;
Sequence *Seq, *Seq2;

int ret = zsr->getSequences(padrow, pad, &nSeq, &Seq);
      // Now the sequence’s can be accessed as
      // Seq[1. . . nSeq]

ret = zsr->getSequences(padrow2, pad2, &nSeq2, &Seq2);
      // Now Seq[1. . . nSeq] contains the sequences for pad
      //     Seq2[1 . . .nSeq2] contains the sequences for pad2

ret = zsr->getSequences(padrow3, pad3, &nSeq, &Seq);
      // nSeq may have changed from its previous value
      // Now Seq[1. . .nSeq] contains the sequences for pad3
      //     Seq2[1. . .nSeq2] contains the sequences for pad2
      // The containing the sequences for pad is still allocated
      // although I no longer have a pointer to it

// Here’s a do nothing loop that reads all the ADC values from
// pad3 into a variable and throws them away
unsigned char ADC;
for(int i=1; i<=nSeq; i++)
{
  unsigned char *p = Seq[i].FirstAdc;
  for(int j=1; j<=Seq[i].Length; j++)
  {
    ADC = *(p++);
  }
}

delete zsr;
      // The memory for all three pads is unallocated
      // if the InformBuffers() and AttachBuffers()
      // capabilities are implimented, the actual data buffers
      // are still allocated internally, but the only way to access them
      // is to reattach the buffers by re-creating the sector
      // reader.

//************************************ INVALID!!!!!!!
// The same do nothing loop now returns garbage or worse
unsigned char ADC;
for(int i=1; i<=nSeq; i++)
{
  unsigned char *p = Seq[i].FirstAdc;
  for(int j=1; j<=Seq[i].Length; j++)
  {
    ADC = *(p++);
  }
}
//*******************************************************
```

The Sector Reader Classes

The following are the explicit interface definitions for each kind of *SectorReader* currently present for the TPC.

ZeroSuppressedReader:

The data is returned using the Sequence structure as follows. The Pad and PadRow structures are used internally by the format reader.

```
// Each sequence contains one hit
struct Sequence
{
  unsigned short startTimeBin;
  unsigned short Length;
  unsigned char* FirstAdc;
};

// A pad contains the array of hits for that pad in this event
// (Internal Only)
struct Pad
{
  unsigned char nSeq;
  Sequence* Seq;
}

// A pad row contains the array of pads contained in the padrow
// (Internal only)
struct PadRow

{
  int nPads;
  Pad* Pad;
}
```

The data are read by using the interface defined below

```
class ZeroSuppressedReader
{
public:
  virtual int getPadList(int PadRow, unsigned char **padList)=0;
      // Fills (*padList)[] with the list of pad numbers containing hits
      // returns number of pads in (*padList)[]
      // or negative if call fails

  virtual int getSequences(int PadRow, int Pad, int *nSeq,
                           Sequence **SeqData)=0;
      // Fills (*SeqData)[] along with the ADC
      // buffers pointed to by (*SeqData)[]
      // Set nSeq to the # of elements in the (*SeqData)[] array
      // returns 0 if OK.
      // or negative if call fails

  virtual int MemUsed()=0;
  virtual ~ZeroSuppressedReader() {};
}
```

ADCRawReader:

The data are read by using the following interface. The data is returned via an array of 8 bit values.

```
class ADCRawReader
{
public:
  virtual int getPadList(int PadRow, unsigned char **padList)=0;
      // As for Zero suppressed data, this returns
      // the list of pads for which data can be obtained
      // Therefore, the padList will always contain all of the
      // pads in the specified PadRow regardless of the data

  virtual int getSequences(int PadRow, int Pad, int *nArray, uchar **Array)=0;
      // Fills (*Array)[] with Raw data
      // Fills nArray with the # of elements in (*Array)[] (512 bytes for TPC)
      // returns 0 if OK.
      // returns negative if call fails

  virtual int MemUsed()=0;
  virtual ~ADCRawReader() {};
}
```

PedestalReader:

The data are read by using the following interface. The data is returned via an array of 8 bit values.

```
class PedestalReader
{
public:
  virtual int getPadList(int PadRow, unsigned char **padList)=0;
      // As for Zero suppressed data, this returns
      // the list of pads for which data can be obtained
      // Therefore, the padList will always contain all of the
      // pads in the specified PadRow regardless of the data

  virtual int getSequences(int PadRow, int Pad, int *nArray, uchar **Array)=0;
      // Fills (*Array)[] with Pedestal data
      // Fills nArray with the # of elements in Array (512 bytes for TPC)
      // returns 0 if OK.
      // returns negative if call fails

  virtual int getNumberOfEvents()=0;
      // returns the number of events the pedestal run based on

  virtual int MemUsed()=0;
  virtual ~PedestalReader() {};
}
```

PedestalRMSReader:

The data are read using the following interface. The data is returned via an array of 8 bit values. These values must be divided by 16 to obtain the actual floating point RMS values.

```
class PedestalRMSReader
{
public:
  virtual int getPadList(int PadRow, unsigned char **padList)=0;
      // As for Zero suppressed data, this returns
      // the list of pads for which data can be obtained
      // Therefore, the (*padList)[] will always contain all of the
      // pads in the specified PadRow regardless of the data

  virtual int getSequences(int PadRow, int Pad, int *nArray, uchar **Array)=0;
      // Fills (*Array)[] with Pedestal RMS data * 16
      // Fills nArray with the # of elements in (*Array)[] (512 bytes for TPC)
      // returns 0 if OK.
      // returns negative if call fails

  virtual int getNumberOfEvents()=0;
      // returns the number of events the pedestal run based on

  virtual int MemUsed()=0;
  virtual ~PedestalRMSReader() {};
}
```

GainReader:

The data are read using the following interface. The data are returned via a gain structure:

```
struct Gain
{
  int t0;         // t0 * 16
  int t0_rms;     // t0_rms * 16
  int rel_gain;   // rel_gain * 64
}

class GainReader
{
public:
  virtual int getGain(int PadRow, int Pad, struct Gain **gain)=0;
      // sets (*gain) to a valid gain structure pointer
      // returns 0 if OK
      // returns negative if call fails

  virtual int getMeanGain()=0;
      // returns mean gain

  virtual int getNumberOfEvents()=0;
      // returns the number of events the calculation is based upon

  virtual int MemUsed()=0;
  virtual ~GainReader() {};
}
```

CPPReader:

Returns the raw cluster pointers provided by the ASICS.

```
struct ASIC_Cluster
{
  short start_time_bin;
  short stop_time_bin;
}

class CPPReader()
{
public:
  virtual int getClusters(int PadRow, int Pad,
                          int *nClusters, struct ASIC_Cluster **clusters)=0;
      // sets (*clusters) to beginning of array of clusters
      // sets nClusters to the length of the array
      // returns 0 if OK
      // returns negative if call fails

  virtual int MemUsed()=0;
  virtual ~CPPReader() {};
}
```

BadChannelReader:

Returns a boolean value for whether a channel is bad.

```
class BadChannelReader
{
public:
  virtual int IsBad(int PadRow, int Pad)=0;
      // returns true if the pad is bad.
      // returns false if the pad is not bad.

  virtual int MemUsed()=0;
  virtual ~BadChannelReader()=0;
}
```

ConfigReader:

Returns the FEE_id for the pad.

```
class ConfigReader
{
public:
  virtual int FEE_id(int PadRow, int Pad) = 0;
      // returns FEE_id

  virtual int MemUsed()=0;
  virtual ~ConfigReader() {};
}
```

Appendix: Class headers for event reader and detector reader.

The EventReader Class:

```
// These are the functions used to create the Event Reader
EventReader *getEventReader(int fd, long offset,  int mmap=1);
EventReader *getEventReader(void *event);

struct EventInfo
{
  // Not yet specified.  Contains:
  // 1.  A list of detectors in the event
  // 2.  A list of sectors for each detector
  // 3.  Anything else of use from DATAP
}

class EventReader
{
public:
  EventReader()
  void InitEventReader(uint fd, long offset);  // Constructor Given File, offset
  void InitEventReader(void *datap);          // Constructor Given pointer


  EventInfo(EventInfo *);            // returns a structure defining event
      // The memory for this structure is allocated by the client!
      // There is no internal buffer created for this information as
      // It is obtained by crawling the pointer banks down to the SECP level
      // The EventReader class only buffers data down to the DATAP level

  ~EventReader();

  int errno();                      // get the last err number
  string errstr();

  int MemUsed();

protected:                            // only accessible by the format reader
  // IO functions
  void *DATAP;                       // Pointer to datap
  int event_size;                    // size of event in words

  // Detector Buffering Functions
  InformBuffers(DetectorReader *);  // Does nothing in initial versions
                                     // Later,
                                     // gives ownership of buffers from
                                     // detectorreader to eventreader
  AttachBuffers(DetectorReader *);  // Does nothing in initial versions
                                     // Later, attaches formated buffers
                                     // to detector reader if they've
                                     // already been read
private:
  uint fd;              // -1 if event in memory
  void *event;          // NULL if event in file
  int errno;

  // Eventually storage for the detector buffers
  // (probably a vector of pointers indexed by detector name)
};
```

DetectorReader Class:

```
class DetectorReader
{
  friend class EventReader;

public:
  virtual ZeroSuppressedReader *getZeroSuppressedReader(int sector)=0;
  virtual ADCRawReader *getADCRawReader(int sector)=0;
  virtual PedestalReader *getPedestalReader(int sector)=0;
  virtual PedestalRMSReader *getPedestalRMSReader(int sector)=0;
  virtual GainReader *getGainReader(int sector)=0;
  virtual CPPReader *getCPPReader(int sector)=0;
  virtual BadChannelReader *getBadChannelReader(int sector)=0;

  DetectorReader(EventReader *);
  virtual ~DetectorReader() {} ;

  virtual int MemUsed()=0;

  int errno();
  string errstr();

protected:

  // Buffer and index functions for the various readers.
  // Initially these will do nothing.  Add functionality
  // to increase performance
  virtual int InformBuffers(ZeroSuppressedReader *,int sector)=0;
  virtual int InformBuffers(ADCRawReader *, int sector)=0;
  virtual int InformBuffers(PedestalReader *, int sector)=0;
  virtual int InformBuffers(PedestalRMSReader *, int sector)=0;
  virtual int InformBuffers(GainReader *, int sector)=0;
  virtual int InformBuffers(CPPReader *, int sector)=0;
  virtual int InformBuffers(BadChannelReader *, int sector)=0;
  virtual int InformBuffers(ConfigReader *, int sector)=0;

  virtual int AttachBuffers(ZeroSuppressedReader *, int sector)=0;
  virtual int AttachBuffers(ADCRawReader *, int sector)=0;
  virtual int AttachBuffers(PedestalReader *, int sector)=0;
  virtual int AttachBuffers(PedestalRMSReader *, int sector)=0;
  virtual int AttachBuffers(GainReader *, int sector)=0;
  virtual int AttachBuffers(CPPReader *, int sector)=0;
  virtual int AttachBuffers(BadChannelReader *, int sector)=0;
  virtual int AttachBuffers(ConfigReader *, int sector)=0;

  int errno;
private:
  EventReader *er;

  // Storage for the reader buffer’s
};
```