Specification for the Offline Interface to DAQ: StDaqLib

Authors: ***Jeff Landgraf (jml@bnl.gov)***
M.J. LeVine (levine@bnl.gov)

Defining the event

The user of this library 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 stored in a file, the DAQ interface library requires an open file descriptor for 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 StDaqLib is to select the event. This is accomplished by creating an *EventReader* object:

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

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

If the *EventReader* is created using an input file then it is possible to 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, mmap=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.

A variant of the Event Reader which creates a diagnostic log file (strongly recommended) is invoked:

```
     EventReader *er = getEventReader(fd, offset,
          logfilename [, mmap]);
```

The log file is opened in append mode, so that the file name can be constructed from the data file name, if desired, and the diagnostic messages from successive events will be contained in a single file.

The potential user is directed to the sample user-level code in *client.cxx,* in StDaqLib/examples. There are two frameworks in which StDaqLib can be used. First it can be accessed via Root. Second, it can be used with a top-level program (replacement for client.cxx) and compiled and linked using the Makefile in StDaqLib/examples.

Numbering scheme

StDaqLib uses indices starting with 1 for physical entities: sector, pad row, pad. Abstract or derived quantities such as sequence number are numbered starting with 0.

Various detectors within STAR

The detectors whose readout resembles that of the TPC are characterized by a uniform interface, described in the next section. Other detectors have interfaces unique to each detector, described separately in a later section of this document.

Defining The Detector (TPC-like detectors)

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 the various **TPC-like** detectors 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 2.0 reader):

DetectorReader *dr = getDetectorReader(er, “TPCV2.0”);

dr is now a pointer to a TPCV2P0_Reader object which is specifically implemented for the TPC.

The DetectorReader object contains factories that produce all of the varied sector readers.

The TPC-like detectors are TPC, FTPC, SSD, SVT.

Defining A Sector Reader (TPC-like detectors)

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* interface:

```
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 corrupted 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 (N =1,…,24):

```
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 arrays are allocated and populated, any necessary byte swapping or uncompressing is done, and disk-based data are 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. The *DetectorReader* implementation determines which implementation of the *SectorReader* object is actually created. In the future, the specific version of the Detector Reader is expected to depend on the format version of the event itself.

Buffer lifetime for the sector readers:

The specific Sector Readers 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.

The data structures which support the sector readers are fully populated when the sector reader is constructed. For example, the array of Sequences for all pads are allocated and filled.

In the following example, the full code to read the zero suppressed sequences from a few specific pads is shown:

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

DetectorReader *dr = getDetectorReader(er, "TPCV2.0");
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);
     // The sequences can be accessed as Seq[0,...,nSeq-1]

ret = zsr->getSequences(padrow2, pad2, &nSeq2, &Seq2);
     // Now Seq[0,...,nSeq-1] contains sequences for pad
     //     Seq2[0,...,nSeq2-1] contains sequences for pad2
```

```
ret = zsr->getSequences(padrow3, pad3, &nSeq, &Seq);
     // nSeq contains the number of sequences for pad3
     // Seq[1,...,nSeq] contains the sequences for pad3
     // Seq2[1,...,nSeq2] contains the sequences for pad2
     // The array containing the sequences for pad is still
     // allocated although we no longer have a pointer to it

// Here's a do-nothing loop that reads all the ADC values
// from pad3, converts them to 10-bit values and discards
// them:
#include "StDaqLib/TPC/trans_table.hh"
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 = log8to10_table[*(p++)];
  }
}

delete zsr;  // The memory for all three pads is deallocated.
// any further reference to the arrays Seq[1,...,nSeq],
// Seq2[1,...,nSeq2], will access irrelevant data.
```

The Sector Reader Classes

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

ZeroSuppressedReader:

The zero suppressed reader attempts to locate the zero suppressed banks TPCADCD, TPCSEQD for all mezzanine cards in the specified sector. If these are missing for any mezzanine card, it attempts to construct them using the raw banks TPCADCR, TPCPADK, TPCCPPR. If these cannot be found, sequences for pads corresponding to the missing mezzanine are left unpopulated. This behavior, as opposed to returning an error status when the sector reader is initialized, is required in order to accommodate receiver boards that have been disabled in software due to hardware problems on the TPC readout board or in the receiver board itself.

The data are accessed using Sequence structure as follows.

```
// Each Sequence contains one hit on a single pad
struct Sequence
{
  unsigned short startTimeBin;
  unsigned short Length;
  unsigned char* FirstAdc;
};

struct Centroids {
  unsigned short x; // units: 1/64 pad
  unsigned short t; // units: 1/64 timebin
};

struct SpacePt {
  Centroids centroids;
  unsigned short flags;
  unsigned short q;
};
```

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 getSpacePts(int PadRow, int *nSpacePts,
          SpacePt **SpacePts)=0
     // gets SpacePts array from TPCMZCLD banks

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

ADCRawReader:

The ADCRaw reader attempts to locate the raw banks TPCADCR, TPCPADK. If these cannot be found, sequences for pads corresponding to the missing mezzanine are left unpopulated. This behavior, as opposed to returning an error status when the sector reader is initialized, is required in order to accommodate receiver boards that have been disabled in software due to hardware problems on the TPC readout board or in the receiver board itself.

The data are read by using the following interface. The data are 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 are 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 are 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;
}
struct ASIC_params
{
  unsigned char thresh_lo;
  unsigned char thresh_hi;
  unsigned char n_seq_lo;
  unsigned char n_seq_hi;
};
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 getAsicParams(ASIC_params *)=0;
  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 each pad.

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

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

Remaining STAR Detectors

RICH

Sample code to create the RICH reader follows:

```
RICH_Reader *drich = getRICHReader(er);
      if(!drich)
     cout << "Error creating RICH_Reader: " << er->errstr() <<
endl;
      else {
     printf("created RICH_Reader\n");
     // call additional methods defined for RICH
      }
```

Interface:

```
class RICH_Reader{
  void ProcessEvent(const Bank_RICP * RichPTR);

public:

  RICH_Reader(EventReader *er, Bank_RICP *pRICP);
  ~RICH_Reader(){};
  unsigned short GetADCFromCoord(int x,int y);
  unsigned short GetADCFromCramChannel(int cramBlock,
           int channelNum);
  unsigned int GetEventNumber();
  const char * GetBankType();
}
```

Appendix: Class headers for event reader and detector reader.

The EventReader Class:

```
struct EventInfo // return from EventReader::getEventInfo()
{ int EventLength;
  unsigned int UnixTime;
  unsigned int EventSeqNo;
  unsigned int TrigWord;
  unsigned int TrigInputWord;
  unsigned char TPCPresent;
  unsigned char SVTPresent;
  unsigned char TOFPresent;
  unsigned char EMCPresent;
  unsigned char SMDPresent;
  unsigned char FTPCPresent;
  unsigned char Reserved;
  unsigned char RICHPresent;
  unsigned char TRGDetectorsPresent;
  unsigned char L3Present;
};

// Functions used to create the Event Reader:
EventReader *getEventReader(int fd, long offset, int mmap=1);
EventReader *getEventReader(int fd, long offset,
                    char *logfilename, int mmap=1);
EventReader *getEventReader(void *event);

class EventReader
{
public:
  EventReader()
  EventReader(const char *logfilename)
  void InitEventReader(uint fd, long offset, int mmap=1);
                          // Constructor Given File, offset
  void InitEventReader(void *datap);
                          // Constructor Given pointer
  struct EventInfo getEventInfo();
     // returns a structure containing event information
     // (see above struct declaration)
  void printEventInfo();  // prints EventInfo on stdout
  void printEventInfo(File *); // prints EventInfo on File
  long NextEventOffset(); // returns offset to next event
  void setVerbose(int);   // 0 turns off all internal printout
  char * findBank(char *bankid); // navigates to specified
                  pointer bank below DATAP
  ~EventReader();
  int errno();            // get the last err number
  string errstr();
  int runno()             // returns run number
  FILE *logfd;            //file handle for log file
  int MemUsed();

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

private:
  uint fd;                // -1 if event in memory
  char *MMAP;             // Begining of memory mapping
  int runnum;
};
```

DetectorReader Class (factory class for all TPC-like detectors):

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

  int errno();
  string errstr();

protected:
  int errno;
  char errstr0[250];

private:
  EventReader *er;
};
```