LHCb note 2008-047

LPHE note 2008-010

Silicon Tracker zero-suppressed data model and decoding

M. Needham
École Polytechnique Fédérale de Lausanne

January 29, 2009

Abstract

The Silicon Tracker zero-suppressed data model and decoding are described. Examples of the use of the event model classes together with checks made of the data integrity in the decoding procedure are described.

Revision January 2009 (Brunel v34r1):

- Add description of **StripRepresentation** and related classes
- Update description of **IT** and **TTName** classes
- Update description of cluster classes

1 Introduction

The information sent from each TELL1 readout board is stored in so called **RawBanks** [1]. The format of the zero-suppressed cluster banks in the case of the Silicon Tracker is described in [2]. The bank consists of three parts (Fig. 1). The first part is a header that gives:

- The size of the bank in bytes.
- A unique identifier for the Tell1 board that sent the data [3].
- An error flag. This bit is set if the Tell1 board reported an error. In this case an error bank is also sent [4].
- The PCN of the event reported by the Tell1. This is the PCN of the first Beetle that is flagged as working in the Tell1 board configuration.
- The number of clusters in the bank.

The second section contains the channel numbers of the clusters whilst the final part contains the ADC values of the corresponding strips. For use in the track reconstruction, trigger and for monitoring this information is decoded into cluster objects. During this procedure the data integrity is checked and the online Tell1 channel numbering converted to the offline channel numbering described in [3].

This note is structured as follows. First the offline data model is described. Then the decoding algorithms are described. Both the data model and decoding software were developed in collaboration with the Velo and share many common elements with the software described in [5].

2 Data Model

Two clusters classes are provided. The first is the **STLiteCluster** class which gives access to the information contained in the first two sections of the **RawBank** format. The second is the **STCluster** class that allows access to the full information contained in the **RawBank**. The first class is used in the trigger and pattern recognition whilst the second is used by the track fit and monitoring. The source code for both classes is located in the

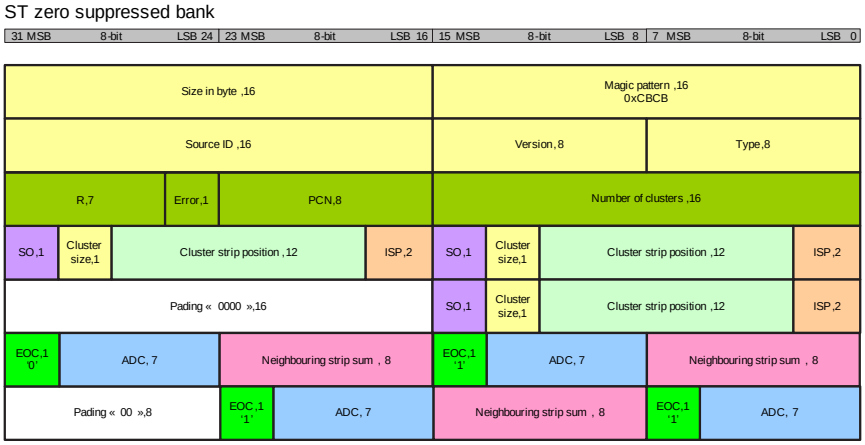

Figure 1: ST zero-suppressed data bank format (from [2]).

DigiEvent package. These classes together with the **STSummary** [1] and **STChannelID** [2] class are described in the following sections. The location of the source code for the classes in this note is summmarized in Table 1.

Class	Package	Header Location
STCluster	DigiEvent	Event/STCluster.h
STLiteCluster	DigiEvent	Event/STLiteCluster.h
STChannelID	LHCbKernel	Kernel/STChannelID.h
TTNames	LHCbTKernel	Kernel/TTNames.h
ITNames	LHCbKernel	Kernel/ITNames.h
STSummary	RecEvent	Event/STSummary.h
StripRepresentation	STKernel	Kernel/StripRepresentation.h
BeetleRepresentation	STKernel	Kernel/BeetleRepresentation.h
PPRepresentation	STKernel	Kernel/PPRepresentation.h

Table 1: Header file location.

[1] Located in **RecEvent**.
[2] Located in **LHCbKernel**.

2.1 STChannelID

Both cluster classes give access to a **STChannelID** object. This is a bit packed word that uniquely labels every channel in the Silicon Tracker. The following code fragments illustrate how to create and use objects of this type:

```
#include "LHCb/STChannelID.h
using namespace LHCb;

// construct an object of this type
unsigned int station, layer, detRegion, sector, strip
STChannelID chan(STChannelID::typeTT, station, layer, detRegion, sector, strip);

// get back the information
unsigned int station = chan.station();
unsigned int layer = chan.layer();
```

Often it is useful to convert part of the **STChannelID** into a string. To help in this two classes are provided in the **LHCbKernel** package - **ITNames** and **TTNames**. The use of these classes is shown below:

```
#include "Kernel/ITNames.h"
#include "Kernel/TTNames.h"
using namespace LHCb;

STChannelID chan;
if (chan.isTT()){

  // print the sector name
  std::cout << "unique sector name "
            << TTNames().uniqueSectorToString(chan) << std::endl;

}
else {

  std::cout << "unique layer "
            << ITNames().uniqueLayerToString(chan) << std::endl;

  std::string << "just the layer name "
              << ITNames().layers(chan) << std::endl;

}
```

These classes also allow to get lists of all valid names. For example, the code fragment below shows how to get a list of all valid TT layer names:

```
#include "Kernel/TTNames.h"
using namespace LHCb;
std::vector<std::string> layers = TTNames().allLayers();
```

2.2 StripRepresentation

The channels in the raw data bank are numbered from 0 to 3071. For type safety this numbering scheme is wrapped inside a class (**StripRepresentation**). In some circumstances it is useful to decompose this into a beetle and analogue port together with a relative strip number. Another use case is to decompose it into the Tell1 FPGA ('PP'), beetle, port and strip. To do this two helper classes are provided (**BeetleRepresentation**, **PPRepresentation**). Interconversion between the different representations is also supported. The following code fragments illustrate the use of the classes:

```
#include "Kernel/StripRepresentation.h"
#include "Kernel/BeetleRepresentation.h"
#include "Kernel/PPRepresentation.h"

using namespace STDAQ;

unsigned int strip; // 0 - 3071
StripRepresentation strip = StripRepresentation(chan);

// go to beetle rep
BeetleRepresentation beetleRep = BeetleRepresentation(strip);

// decompose into beetle [0-23], port [0-3], strip 0-31
unsigned int beetle, port, relStrip;
beetleRep.decompose(beetle,port,strip);

// get back the strip
strip = beetleRep.toStripRepresentation()

// constructor a new beetle representation
BeetleRepresentation beetleRep2 = BeetleRepresentation(beetle,port,relStrip);

// go to PP Rep
PPRepresentation ppRep = PPRepresentation(strip);

// decompose into pp [0-3], beetle [0-5], port [0-3] strip 0-31
unsigned int pp;
ppRep.decompose(pp,beetle,port,strip);

// get back the strip
strip = ppRep.toStripRepresentation()

// constructor a new pp representation
PPRepresentation ppRep2 = PPRepresentation(pp,beetle,port,relStrip);
```

2.3 STLiteCluster

This class provides access to the information in the second part of the data bank. It is used in the pattern recognition and the trigger where access to the ADC values is not necessary and the emphasis is on the decoding

speed. Internally an object of this type is represented as a 32-bit integer which ensures allocation. Externally users see an interface to the data which means they do not need to know the details of the internal bit-packing. The following code fragments illustrate how to use an object of this type:

```
#include "Event/STLiteCluster.h"
using namespace LHCb;
STLiteCluster clus;

// channelID
STChannelID chan = clus.channelID();

// short-cut to station etc are provided
unsigned int station  = clus.station();

// interstrip fraction: 0., 0.25, 0.5, 0.75
const double frac = clus.interStripFraction();

// cluster size
const unsigned int pSize = clus.pseudoSize();

// check if has high threshold
const bool hasHigh = clus.highThreshold();
```

Objects of this type are stored in a 'FastContainer' that follows the same philosophy of hiding the lightweight internal representation from the user. The container has similar functionality to that provided by the STL vector class:

```
using namespace LHCb;

// In STLiteClusters header is following typedef
// typedef FastClusterContainer < LHCb::STLiteCluster, int > STLiteClusters;

typedef STLiteCluster::STLiteClusters FastCont;
FastCont cont;
STLiteCluster clus;
cont.push_back(clus);

// iterate over the container
FastCont::const_iterator iter = cont.begin();
for (; iter != cont.end(); ++iter){
   // do something
}
```

A policy is also provided to find a **STLiteCluster** with a given channel in the **FastContainer**:

```
using namespace LHCb;
typedef STLiteCluster::STLiteClusters FastCont;

STLiteCluster clus;
FastCont fastCont;
// find it in the fast container
FastCont::iterator iter =
  fastCont->find<FastCont::findPolicy>(clus);
```

To allow binary searching the clusters in the **FastContainer** are sorted by increasing channel number [3].

For convenience short-cuts (that internally delegate to the **IT** and **TTNames** classes) are provided to convert the **STChannelID** into string format are provided:

```
using namespace LHCb;
STLiteCluster clus;

std::cout << "Station " << clus.stationName() << std::endl;
std::cout << "Layer " << clus.layerName() << std::endl;
std::cout << "Sector " << clus.sectorName() << std::endl;
```

2.4 STCluster

Access to the full information within the cluster bank is provided by the **STCluster** class. This class is used by the offline track fit and also for monitoring. Internally, this class contains a **STLiteCluster** (and hence provides the functionality described in the previous section) together with a list of the ADC values for the strips that contributed to the cluster:

```
#include "Event/STCluster.h"
using namespace LHCb;
STCluster* aCluster;

// total charge
const double charge = aCluster->totalCharge();

// vector of ADC values
const LHCb::STCluster::ADCVector& vec = aCluster->stripValues();

// adc value of given strip
unsigned int strip;
const unsigned int adc = aCluster->adcValue(strip);

// maximum adc value in the cluster
const unsigned int maxADC = aCluster->maxADCValue();

// channels that contributed to the cluster
std::vector<STChannelID> channels = aCluster->channels();

// first and last channel
const STChannelID firstChan = aCluster->firstChannel();
const STChannelID lastChan = aCluster->lastChannel();

// check if cluster contains a channel
STChannelID aChan;
bool isInCluster = aCluster->contains(aChan);
```

This information is used to re-calculate the inter-strip fraction offline. A method is also provided that gives access to neighbour sum information:

[3] This corresponds to sorting by increasing zyx [3].

```
using namespace LHCb;
STCluster* aCluster;
double sum = aCluster->neighbourSum();
```

For TAE running where more than one spill is read out and clusters from different spills are merged into one container access is provided to a spill identifer:

```
using namespace LHCb;
STCluster* aCluster;
std::cout << "Spill " << aCluster->spill() << std::endl;
```

Finally, the class provides access to the Tell1 board and online channel number the cluster corresponded to:

```
LHCb::STCluster* aCluster;
const unsigned int boardID = aCluster->sourceID();
const unsigned int strip = aCluster->tell1Channel();
```

Objects of this type are stored in a **KeyContainer** with the **STChannelID** as the key. This container provides sequential access via iterators together with the possibility of 'keyed' access:

```
using namespace LHCb;
// retrieve clusters [GaudiAlgorithm syntax]
const STClusters* clusterCont = get<STClusters>(STClusterLocation::TTClusters);

// sequential access
STClusters::const_iterator iterClus = clusterCont->begin();
for( ; iterClus != clusterCont->end(); ++iterClus){
  // do something
}

// keyed access
STChannelID chan;
STCluster* clus = clusterCont->object(chan);
if (chan != 0){
   // cluster is in the container
}
```

To allow binary searching the clusters in the **STClusters** container are sorted by increasing channel number.

2.5 STSummary

During the decoding step information on Tell1 boards that are missing or have problems is collected. This information together with the PCN of the event is stored in an **STSummary** class that is created and registered on the

store during the **STCluster** decoding. The code fragments below illustrate the use of this class:

```
// Event
#include "Event/STSummary.h"
using namespace LHCb;

// retrieve the event summary
const STSummary* summary = get<STSummary>(TSummaryLocation::TTSummary);

// some banks are lost
const std::vector<unsigned int>& lost = summary->missingBanks();

// some banks were corrupted
const std::vector<unsigned int>& corrupted = summary->banksWithError();

// some banks can be recovered [map containing source + fraction recovered]
const STSummary::RecoveredInfo& recovered = summary->recoveredBanks()

// pcn
const double pcn = summary->pcn();
bool sync = summary->pcnSync();
```

3 Decoding Algorithms

The **STLiteClusters** and **STClusters** are created by decoding the RawBanks. Two **GaudiAlgorithms** are provided for this purpose in the **STDAQ** package: **RawBankToSTLiteClusterAlg** and **RawBankToSTClusterAlg**. Since the decoding of both classes shares common function they derive from the **STDecodingBaseAlg**. The base class also provides functionality to decode the Tell1 error banks. This allows the possibility of recovering partially corrupted banks during the decoding procedure. A third algorithm - **STErrorDecoding** derives from the base class. This algorithm 'forces' the error bank decoding by calling the appropriate methods of the base class.

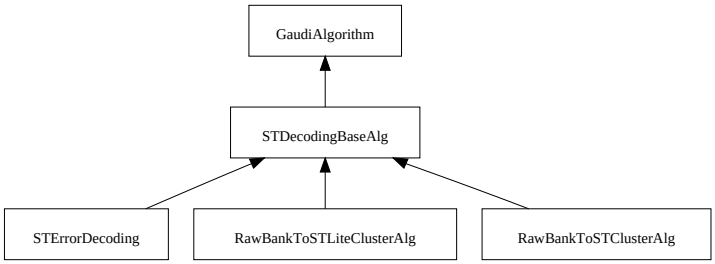

Figure 2: Class diagram for ST decoding algorithms.

The majority of the code in these algorithms is dedicated to converting the online channel number to the offline representation and checking that the integrity of the raw data. These tasks are discussed in detail in the next sections.

3.1 Channel Mapping

The mapping between the online Tell1 channel numbering and offline scheme [3] is performed by the **ISTReadout** interface. There is is one implementation of this tool for the Inner Tracker and one for the Trigger Tracker. A base class expresses the common functionality between the two detectors (Fig. 3). This class delegates much of the work to the **STTell1Board** class.

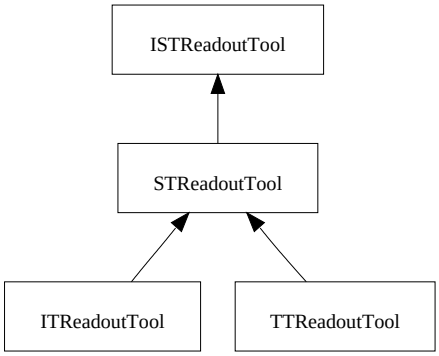

Figure 3: Class diagram for the ST readout tools.

The following code fragments demonstrate how the tool is used in the decoding:

```
#include "Kernel/ISTReadoutTool.h"
#include "Kernel/STTell1Board.h"
#include "Kernel/STTell1ID.h"
#include "Kernel/STChannelID.h"
#include "Kernel/StripRepresentation.h"
#include "Kernel/STDAQDefinitions.h"

using namespace LHCb;

// readout tool
ISTReadoutTool* readoutTool = tool<ISTReadoutTool>("TTReadoutTool");

// get the board
STTell1Board* aBoard = readoutTool->findByBoardID(STTell1ID((*iterBank)->sourceID()));

// check the channel is valid
```

```
unsigned int chan;
if (aBoard->validChannel(chan) == true){
 // valid - use it
 unsigned int fracStrip;
 STTell1Board chanPair =  aBoard->DAQToOffline(fracStrip, STDAQ::v4,
                          STDAQ::StripRepresentation(chan));
 std::cout << "Channel " << chanPair.first << std::endl;
}
```

The tool is also used in the encoding to convert the **STChannelID** to the corresponding Tell1 board and online channel:

```
#include "Kernel/STDAQDefinitions.h"

STChannelID chan; double isf;
STDAQ::chanPair aPair = m_readoutTool->offlineChanToDAQ(chan,isf);
std::cout << "board " <<  " channel " <<  aPair.first << aPair.second std::endl;
```

3.2 Data Integrity Checks

The following checks are made of the data integrity are performed:

- The sourceID of the TELL1 board should be valid [3].
- The magic pattern in the header of the RawBank should be correct.
- The error bit in the header should not be set. If the bank is flagged as having an error it is not decoded unless the recovery mode is activated. In the latter case the corresponding error bank is decoded and all Beetle ports that are not flagged as having an error are decoded [4].
- A majority vote of the PCNs of all boards that do not have errors is performed. Any board that does not have the same PCN as the board that wins the majority vote is discarded.
- The pseudo-size for each cluster written in the first half of the bank should be consistent with the number of ADC values found in the second half of the bank.
- The channel number of the cluster must correspond to a valid detector channel.
- The number of bytes read during the decoding process should be consistent with the value written in the bank header.

[4] By default this mode is activated.

- If two clusters with the same channel number are found only the first is kept. All these checks are currently performed by both decoding algorithms [5].

The number of missing, corrupted and recovered banks is kept track of and stored in the **STSummary** object. In addition, **GaudiAlgorithm** counters are incremented and printed out at the end of the job to give a summary of the decoding performance.

References

[1] O. Callot *et al.* Raw-data Format. EDMS-565851.

[2] G. Haefeli and A. Gong. ST zero suppressed bank data format. EDMS-note 690583/3.

[3] M. Needham and O. Steinkamp. Updated channel numbering and readout partitioning for the Silicon Tracker. LHCb-note 2007-137.

[4] G. Haefeli. VELO and ST error bank data format. EDMS-note 6948181/1.

[5] T. Szumlak and C. Parkes. Velo Event Model. LHCb-note 2006-054.

[5] In the future some of these checks could be dropped from the **STLiteCluster** decoding in order to save some CPU time.