Production of Compressed and Uncompressed rawdata for ALICE Data Challenge

Author:
....
D. Favretto, A .K . Mohanty
CERN, CH-1211 Geneva 23, Switzerland

Abstract:

The ALICE Data Challenge will require simulated raw data in a format as close as possible to the actual detector readouts. This document describes strategy and code that has been implemented to generate raw data both in compressed and uncompressed mode for two ALICE sub-detectors: the Inner Tracking Systems (ITS) and the Time Projection Chamber (TPC).
Key words: Data Challenge, Raw data format, Mapping, ITS, TPC, Data Compression, Huffman.

Contents

1 Introduction

The main purpose of the ALICE experiment is to study the quark-gluon plasma using beams of heavy ions such as those of lead. The particles in the beams will collide thousands of times per second and each collision will generate an event containing thousands of charged particles. Thus, every second, thousands of particles have to be recorded. The central and minimum bias events are expected with a relatively low rate of about 10 events per second with a high data volume of 10 to 40 Megabytes per event. Dielectron events with partial read out will produce a higher data stream (about 100 events per second) for a smaller data volume of 1 to 4 Megabytes per event. There will be also events with Dimuon trigger which may generate about 1500 events per second with a data size of 200 to 750 Killobytes per event. These are the three types of events, each contributing to about one third of the final data volume going through the ALICE data acquisition system. The data flow between the data acquisition system and the permanent data storage will be limited to a throughput of 1.25 Gigabytes per second, for a grand total of 1 Petabytes produced during the lead beam period. Another half a Petabyte will be created during the proton beam run period, amounting to a yearly production (maximum achievable time in a year) of 1.5 Petabytes of raw data to be recorded and made available for later processing.

Since 1998, the ALICE experiments and the IT division have jointly executed several large-scale high throughput distributed computing exercise: the ALICE Data Challenges. The goals of these regular exercises are to prototype the data acquisition and computing systems, to test hardware and software components of these system in realistic condition and to realize an early integration of the overall ALICE computing infrastructure. An important step in the ALICE Data Challenge is to produce simulated raw data for different ALICE detectors in a format as close as possible to the real detector readouts. This doument, therefore, describes the strategy and the software implementation to create raw dat both in compressed and uncompressed mode for two of the ALICE sub-detectors (ITS and TPC).

2 Data Acqusition System Architecture

The ALICE Data Acquisition System (DAQ) architecture will be based on a data-driven approach. Under the control of a three level trigger systems, the Front End Electronics (FEEs), located as closed as possible to the detectors, will readout, format and validate the event rawdata at a local level (ranging from a complete detector to a sector or a sub-sector of the same). All accepted events will be shipped via a customed-designed point-to-point optical link called Detector Data Link (DDL) to a Local Data Concentrator (LDC), a commodity PC located a few hundred meters away from the interaction point. The LDC will validate the event, eventually perform local event building (for LDCs with multiple incoming DDLs), run data compression and other data analysis functions and finally move the raw data to the event builder running on a Global Data Collector (GDC). The ALICE DAQ environment is shown in Figure 1.

The Permanent Data Storage (PDS) system will perform data recording from the GDC and will provide access to the event data for all successive analysis

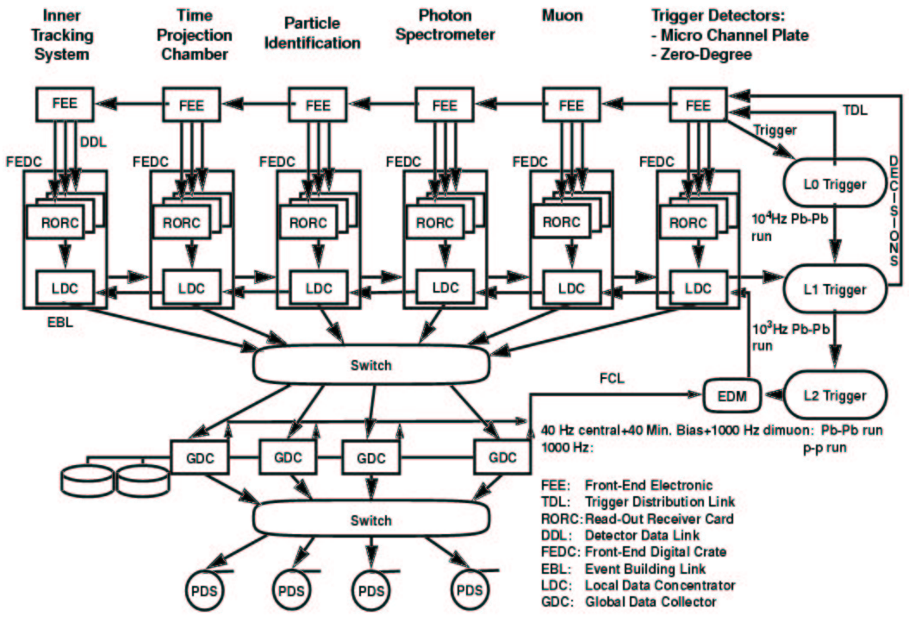


Figure 1: ALICE DAQ system.

stages.

3 General Strategy

The number of DDL per LDC is not fixed and keeps on changing during each data challence exercise. Therefore, it is desirable to produce raw data in a basic unit of a DDL. A LDC file can be generated by combining appropriate numbers of DDL files. Since the raw data is generated in a unit of a DDL, required number of LDC files can be generated during each data challenge without regenerating the raw data.

It is proposed to follow an uniform DDL file structure for all the ALICE sub-detectors as shown in Figure 2.

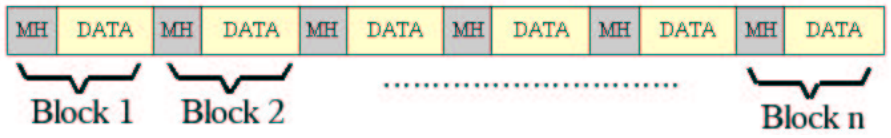


Figure 2: File structure.

Each block is a DDL file which contains a mini header (MH) followed by the DDL raw data. A LDC file contains several such blocks whose number is decided by the number of available LDCs. As an example, the TPC sub-detector has 216 DDLs. If a given data challence uses 12 LDCs, 18 DDL blocks are required to be packed into a single LDC file. The Table 1 shows the number of LDCs assigned to various sub-detectors and the corresponding DDL blocks per LDC file going to be used during a forthcoming data challenge exercise.

Detector	LDCs number	DDLs number	Blocks per file
ITS Drift	4	12	3
ITS Pixel	2	20	10
ITS Strip	2	16	8
TPC	12	216	18

Table 1: LDCs assigned to TPC and ITS.

The above assignment is rather arbitrary. Since a DDL block is fixed, the generation of LDC file is rather trivial. Figure 3 shows an example of an ITS sub-detector with 2 LDCs files having 8 DDL blocks per file and 4 LDC files having 4 DDL blocks per file.

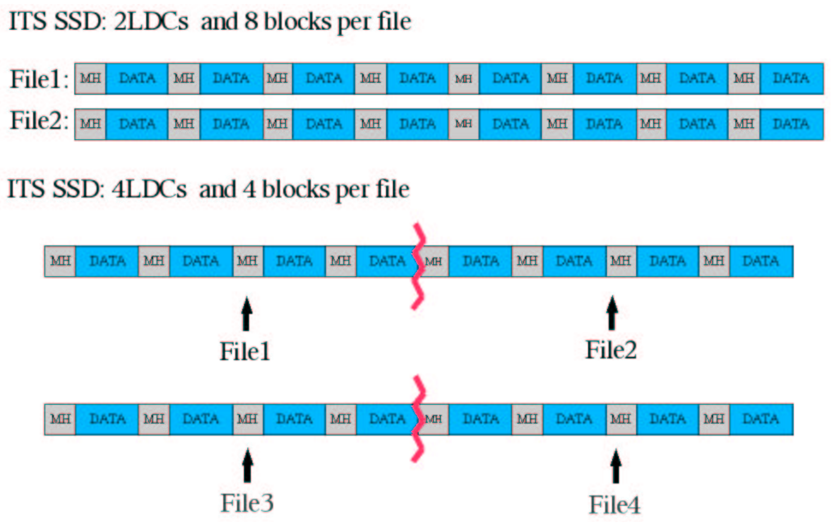

ITS SSD: 2LDCs and 8 blocks per file
File1:
File2:
MH
DATA
ITS SSD: 4LDCs and 4 blocks per file
File1
File2
File3
File4

Figure 3: File rearrangement.

The MH identifies the DDL uniquely and contains the following fields (Total 12 bytes):

- **Size of the raw data** (4 bytes).
- **Magic Word** (3 bytes) A hexadecimal pattern 123456 is used to distinguish between garbage and real data.
- **Detector ID** (1 byte) Used for sub-Detector identification.
- **DDL ID** (2 byte) Used for DDL identification.
- **Flag Compressed/Uncompressed** (1 byte) Indicates if the raw data block is compressed or uncompressed.
- **Version** (1 byte) Mini Header version.

In the following, we describe the classes and methods to be used for raw data production of ITS and TPC sub-detectors.

4 Inner Tracking System

The Inner Tracking System (ITS) consists of 240 Silicon Pixel Detectors (SPD), 260 Silicon Drift Detectors (SDD) and 1698 Silicon Strip Detectors (SPD). These detector modules are linked to the LDC by 48 DDLs as shown in the appendix.

4.1 SPD Raw Data Format

The Silicon Pixel Detector has 240 sensitive modules [1]. Four modules mounted on a planar structure make one full stave. There are 20 staves in the internal layer, and 40 staves in the external one. The total number of modules is $20 * 4 + 40 * 4 = 240$ which are linked by 20 DDLs each containing 12 modules. Each module has 5 read out chips each with 256 by 32 cells. As shown in Figure 4, raw data is formatted considering an half stave as minimal block of size 32 bits. Information within an half stave is organized in a structure called half stave frame shown in Figure 5.

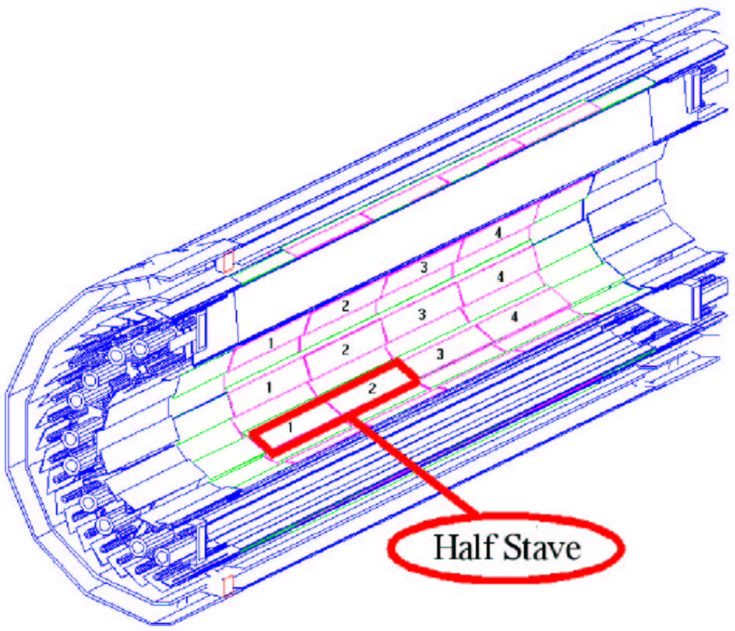


Figure 4: Silicon Pixel Detector.

The word length within the half stave frame is 16 bits. However the half stave frame makes use of 32 bits data words being accepted by the DDL by sending the 16 bits words in both the lower and the upper 16 bit word of the DDL

[1]Note that for SPD, what is refered as module in the Aliroot frame work is called as a ladder as per TDR description.

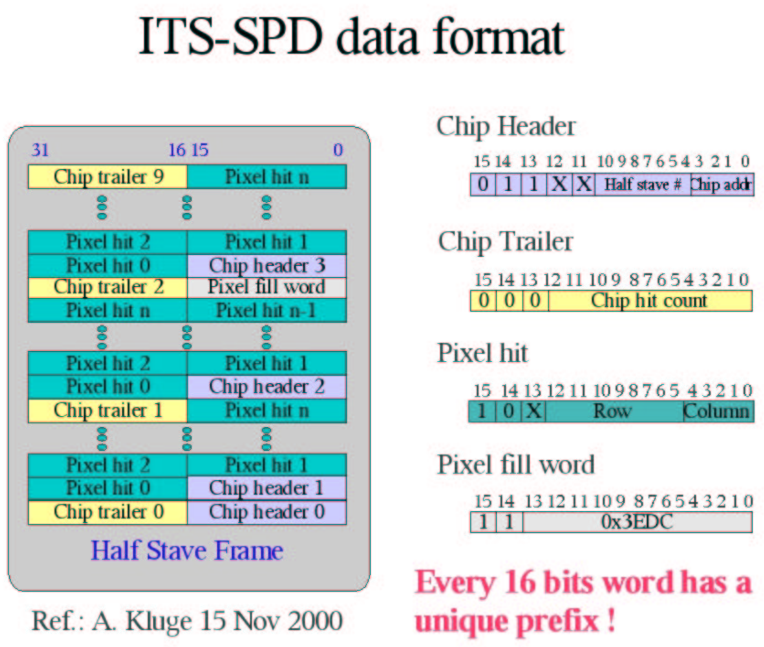


Figure 5: ITS-SDP data structure.

format alternatively. The half stave frame starts with the *Chip Header*. The *Chip Header* is always located in the first 16 bits (bits 0 to 15). The *Chip Header* is followed by the *Pixel Hit* word, that corresponds to one single hit. The data block of a chip is terminated by the *Chip Trailer*. In order to facilitate frame error detection the *Chip Trailer* is always sent to the higher 16 bits (bits 16 to 31). In case the number of hits in one chip is even, the *chip trailer* would be sent to the lower 16 bits of the 32 bits word . Thus a *Pixel Fill Word* is sent before the chip trailer. Note that in case a chip has not been hit at all, still both the *Chip Header* and the *Chip Trailer* have been sent. The *Chip Header* contains the chip number (form 0 to 9) and the half stave number (0 to 120). The *Chip Trailer* contains the total number of hits on the associate chip. The *Pixel hit* contains column and row of an hit in a chip. A Chip is represented by matrix of 256 by 32 cells. The last two bits (14 and 15) are used to identify the *Chip Trailer*, *Chip Header*, *Pixel Hit* and *Pixel Fill Word*.

4.2 SDD Raw Data Format

The SDDs (modules) are mounted on linear structure called ladders, each holding six detectors for the internal layer and eight for the external one. The layers are composed of 14 and 22 ladders, respectively. So, total detector modules are $14 * 6 + 22 * 8 = 260$. Each module will be read by 512 anodes and 256 time buckets with a 8 bits ADC signal. The Figure 6 shows the raw data format which is a simple 32 bits word.

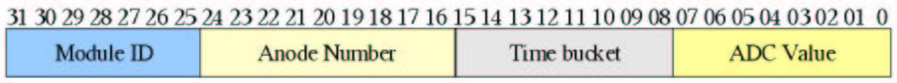


Figure 6: ITS-SDD data structure.

Note that with the above scheme, it is not possible to store the absolute value of the module numbers (range 240:499) using remaining 7 bits (from bit 25 to bit 31). However, since the DDL numbers are stored in the mini header, it is sufficient to store the relative module number within a given DDL as shown in the appendix.

4.3 SSD Raw Data Format

The SSD has 34 ladders in the internal layer and 38 ladders in the external one. Each ladder of the internal layer has 22 detector modules, while there are 25 detector modules in the ladder of the external layer. So, total number of modules is $34*22+38*25 = 1698$. Each module has 768 strips on either side (N/P). Like SDD, the raw data format of SSD is also a 32 bit word as shown in Figure 7, In this case, the first 10 bits are used for the amplitude value, next 10 bits for

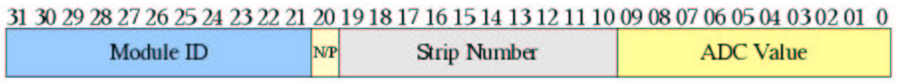


Figure 7: ITS-SSD data structure.

the strip number, 1 bit for the strip type (N or P) and remaining 12 bits can be used for module identification. Even in this case the module identification is a relative number with respect to a given DDL.

4.4 Macro for ITS Raw Data Production

AliITSDDLRawData.C is the macro used to generate ITS raw data files [2]. This macro makes use of the methods of the **AliITSDDLRawData** class to create the required DDL files. This class has three main methods: *RawDataSPD()*, *RawDataSDD()*, and *RawDataSSD()*. All these methods are very similar in implementation and is based on the following schema:

1. Initializations
2. Loop over DDLs
   - Writes the mini header at the begining. Since at the begining, the size of the data block is not knwon, this is only dummy mini header which reserves 12 bytes at the begining.

[2] In the present implementation, a 10 bit ADC value is assumed. So make sure that the line **if(fResponse-¿Do10to8()) signal = Convert8to10(signal)** in the method *AddDigit()* of the class **AliITSsimulationSDD** is commented. Besides, this macro requires a digit file which can be obtained using the macro **AliHits2SDigits.C** and **AliSDigits2Digits.C**.

- Loops over modules filling a buffer as per the sub detector format.
- Writes the buffer into the output file.
- Empties the buffer.
- Writes real mini header.

3. Creates a new DDL file.

In the initialization part, the DDL mapping is read from the look up table: a matrix in which rows represent DDL number and columns respresent detector modules. Three different methods: *GetDigitsSPD()*, *GetDigitsSDD()*, and *GetDigitsSSD()* for SPD, SDD and SSD are used to fill a 32 bit buffer. Last two methods are very simple, they just loop over digits of a given module to fill the buffer. However, the method *GetDigitsSPD()* is bit involved as data must be formatted according to the half stave frame structure as described before. The original digit file (galice.root) is zero suppressed i.e. the chips without any pixel hits are not read. However, in the present SPD format, a chip header and a chip trailer are written for all the chips eventhough a given chip has no pixel hits. The chip numbering is done considering two consecutive modules of an half stave. For each module, a private class variable *fHalfStaveModule* is set to either 0 or to 1 alternatively in order to indicate if the current module is the first or second part of an half stave: this information is used to calculate the chip number (0 to 4 or 5 to 9).

The formatted data is stored in a buffer. This buffer is saved every time a new module is considered. All the methods of this class use directly or indirectly the method *PackWord()*, that inserts a word in a buffer of 32 bits, specifying the start and stop bits.

4.5 Steps to Follow

1. Create *galice.root*
2. Create Digits *AliITSHits2Digits.C*. As mentioned before, for SDD, the ADC signal is assumed to be 8 bits. So, before executing this macro make sure that the line **if(fResponse-¿Do10to8())signal=Convert8to10(signal)** in the method *AddDigit()* of the class **AliITSsimulationSDD** is commented.
3. Execute Macro *AliITSDDLRawData.C*. The methods *RawDataSPD*, *RawDataSDD* and *RawDataSSD* assume 2, 4 and 2 LDCs (by default) for SPD, SDD and SSD respectively. However, number of LDC's can be specified by the last arguments of the above methods. The corresponding LDC files are strored as **xxxslicen** in binary format where *xxx* stands for SPD, SDD and SSD respectively and *n* for the LDC number. The text files **xxxslice.txt** are created for debugging purposes (for verbose level 2 only).

5 Time Projection Chamber (TPC)

The TPC has 36 inner read out sectors (IROS) (0 : 35) and 36 outer read out sectors (OROS) (36 : 71). The IROS has 5504 pads over 63 pad rows (0 : 62). Similarly, the OROS has 9984 pads over 96 pad rows (0 : 95). For read out, each

IROS is divided into two sub-sectors and each OROS is divided into four sub-sectors that make 6 Detector Data Links (DDL) per sector (total 216 DDLs). However, this division is row-wise uneven (see Figure 8 and Fifure 9).

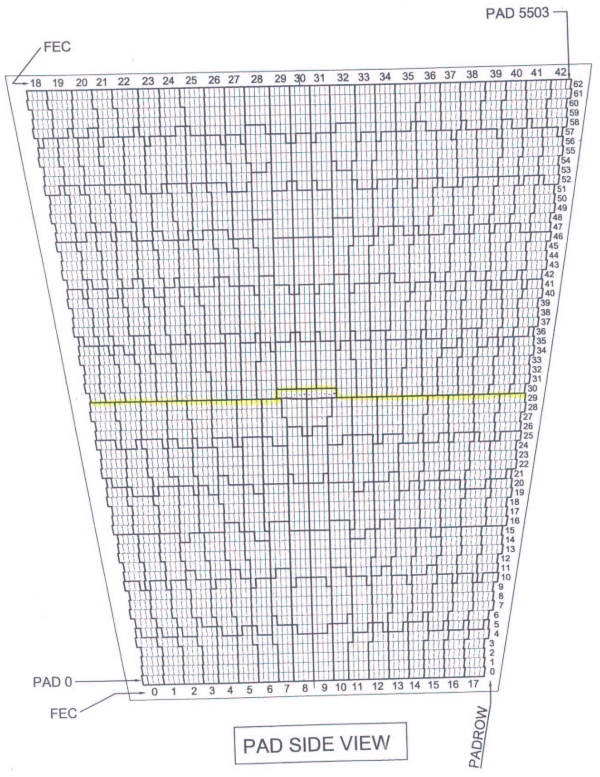


Figure 8: IROS pad side view

For example, the sub-sector 0 of IROS contains all the pads of the rows from 0 to 29 whereas the sub-sector 1 contains the pads of the rows from 31 to 62. The pads of the 30^{th} row are divided between sub-sector 0 and sub-sector 1 as follows: (i) Sub-sector 0 contains pads 37 to 48 of the 30^{th} row, (ii) Sub-sector 1 contains pads 0 to 36 and pads 49 to 85 of the 30^{th} row. Similarly, the four sub-sectors of OROS are composed as follows: (i) Sub-sector 0 contains all the pads of the rows from 0 to 26 and partly the pads of the 27^th row (pads 0 to 42 and 47 to 89), (ii) Sub-sector 1 contains pads 43, 44, 45 and 46 of the 27^{th} row and all the pads of the rows from 28 to 53, (iii) Sub-sector 2 contains pads of all the rows from 54 to 75 and partly the pads of the row 76 (pads 0 to 32 and 89 to 121) (iv) Sub-sector 3 includes pads from 33 to 88 of the 76^{th} row and all the pads of the remaining rows from 77 to 95.

5.1 ALTRO raw data format

As discussed before, a TPC DDL contains several read out pads. The digitized data from each pad is processed and formatted by an Application Specific Integrated Circuilt (ASIC) called ALTRO (ALICE TPC Read Out). The basic

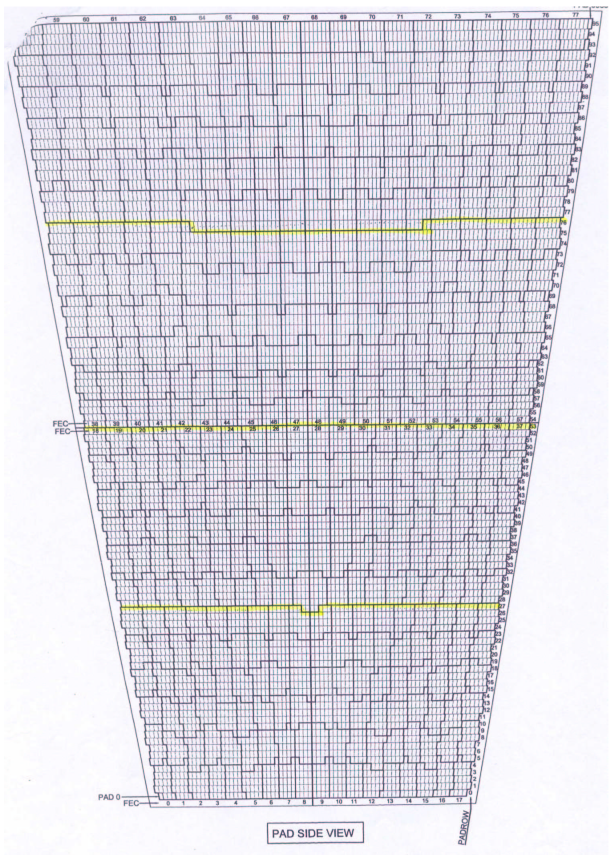


Figure 9: OROS pad side view

pulse detection scheme is fixed thresholding: samples of value smaller than a constant decision level are considered as noise and are rejected. Figure 10 shows the plot of over threshold samples as a function of time bins. Since the signal is sampled with a frequency $\sim$ 5.66 MHZ which divides the total drift time of $88\mu s$ into about 500 time bins. In the ALTRO data format, zero suppressed data is recorded for each pad over all the time bins. This means, if one calls *bunch* a group of adjacent over threshold samples coming from one pad, the signal can be represented bunch by bunch. Since the data is zero suppressed, it is sufficient to record three types of 10 bits data, the sample amplitude in a given bunch (Si), the time bin of the last sample of a given bunch (TB) and the number of samples per bunch (BL).
As shown in figure 11, the ALTRO raw data is composed of a sequence of variable size blocks, each corresponding to a single read-out pad. A block corresponding to a given pad is composed of a sequence of 10 bits words arranged in groups of 4 (packed into 40-bits word). If some data is missing to complete a 40-bit word, an "A" hexadecimal pattern is used. A trailer completes the data packet, which is the last 40-bit word of the data structure. The trailer contains the information about the pad identification (pad, row and sector numbers) and the block size i.e. the number of 10-bits words in the block. The last three words can be used

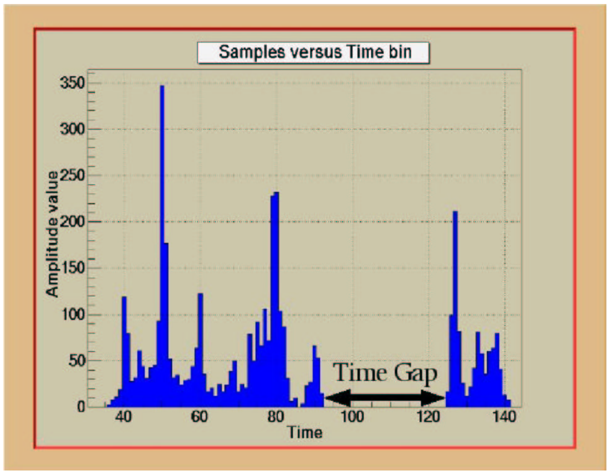
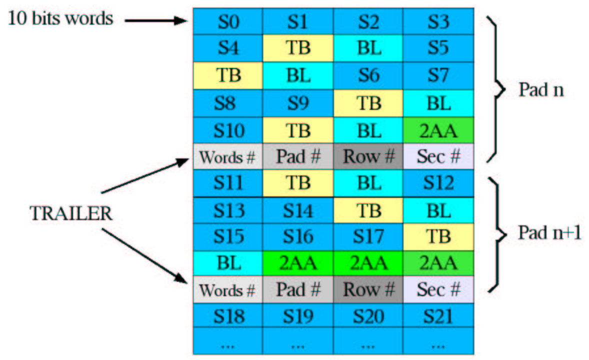


Figure 10: Time bin vs over threshold Samples



S0	S1	S2	S3
S4	TB	BL	S5
TB	BL	S6	S7
S8	S9	TB	BL
S10	TB	BL	2AA
Words #	Pad #	Row #	Sec #
S11	TB	BL	S12
S13	S14	TB	BL
S15	S16	S17	TB
BL	2AA	2AA	2AA
Words #	Pad #	Row #	Sec #
S18	S19	S20	S21
...	...	...	...

Figure 11: Example of an Altro Format.

to extract the exact channel address [3].

5.2 AliTPCBuffer160 class

As the name suggests, this class manages an internal buffer of size 160 bits. Since we use an integer array where the maximum integer size is 32 bits, a 160 bits buffer size is the minimum requirement to pack 16 ALTRO words (10 bits each). Internally, it is managed by an integer array of dimension 5 ($ULong\_t\ fBuffer[5]$). This buffer class can be initialized both in read in write mode (0 read mode and 1 write mode). The $FillBuffer()$ method is invoked to fill the buffer (Buffer[0] to Buffer[4]) with the ALTRO words starting from

[3]The specification of the 40-bits trailer contents are slightly different than what is mentioned in [?]. The basic purpose of the last 40-bits is to know the block size (maximum 10-bits) and to specify the geographical location of the data packet unambiguously. For the purpose of simulation, we have filled the last three words (30 bits) with the pad, row and sector numbers which specifies a channel content rather uniquely. It is also possible to use less number of bits as the mini header of the tape also records the DDL address.

MSB as shown in Figure 12. Finally, the buffer is streamed out into a file when it is full.

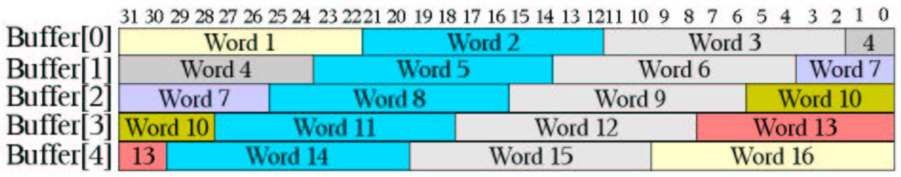


Figure 12: Structure of the internal buffer.

In write mode, follwoing methods can be used.

- `Int_t GetFreeCellNumber()`

  To count the number of words required to fill the buffer.

-
  ```
  void WriteTrailer(Int_t WordsNumber,Int_t PadNumber,
                    Int_t RowNumber,Int_t SecNumber)
  ```

  To write a trailer into the output file.

-
  ```
  void WriteMiniHeader(ULong_t Size,Int_t SecNumber,
                       Int_t SubSector,Int_t Detector,Int_t Flag)
  ```

  To write a mini header; this will be explained later in the section dedicated to the Data Challenge.

- `void SetVerbose(Int_t val)`

  To set the verbose level: 0 silent, 1 output messages.

- `void FillBuffer(Int_t Val)`

  To insert an integer inside the buffer and streams out the buffer contents when it is full.

- `void Flush()`

  Called by the **AliTPCBuffer160** destructor. If the last buffer is incomplete, it is filled with the hexadecimal pattern **2AA** before it is streamed out into the file.

In the read mode, the following methods are used. in the file:

- `Int_t GetNext()`

  This method retrives an ALTRO word from the buffer (in the forward direction) which is filled with 16 ALTRO words from the file at a time. Once all the 16 words are read, the buffer is zeroed and filled again. When the end of the file is reached -1 is returned.

- `Int_t GetNextBackWord()`

  This method retrieve a 10 bits word from the buffer. It works in the similar way as the previous one, but input file is read backward starting from the end. The advantage is, it is now possible to interpret the ALTRO word which is read each time. When the beginning of the file is reached -1 is returned.

- `Int_t ReadTrailerBackward(Int_t &WordsNumber,Int_t &PadNumber,`
  `                              Int_t &RowNumber,Int_t &SecNumber)`

  Reads a trailer. This method has to be used when the internal file pointer points to the begining of a trailer.

5.3 Macros for TPC Raw Data Production

The first step is to get the digits by executing the macro **AliTPCHits2Digits.C** which creates a new branch containing the digits in the galice.root file. The digits although arranged row wise are not DDL compatible.

5.3.1 AliTPCDDL.C

This macro generates a binary file (ALiTPCDDL.dat) as per the DDL mapping. The AliTPC.DDL file contains a sequence of records each containing the following informations:

- Sector number
- Sub sector number
- Row number
- Pad number
- Amplitude value
- Time bin value

In this arrangement, consecutive records having the same sub-sector numbers correspond to a given DDL. The change of sub-sector number indicates end of a DDL. Although in a given DDL, record entries like sector, sub-sector, row and pad numbers are repetative, this file is an intermediate storage and only used to generate DDL files with appropriate ALTRO data structure. This macro makes use of the method *WriteRowBinary()* of the **AliTPCBuffer** class.

5.3.2 AliTPCDDLRawData.C

This macro creates both compressed as well as uncompressed LDC files. The method *RawData(LDCsNumber)* can be used to produce uncompressed raw-data. The *LDCsNumber* specifies the number of LDC need to be produced (default value is 12). the uncompressed LDC files are named as TPCslice1, TPCslice2 and so on. For compression, a simple Huffman coding is applied to the ALTRO raw data. As per ALTRO format, raw data is stored block by block. Since the block size is written at the end, the raw data file is read blockwise backward. Each block is compressed using different Huffman tables for different data types. The trailer contents are not compressed, copied directly to the compressed file. Similarly, fill words are also skipped because they can be re-generated when decompressing the file. So what has been considered for the compression is the sequence of 10 bits words of each block which falls into three different categories as (1) Samples, (2) Time Bins and (3) Bunch Length. For convenience, the time bin is not coded directly. It is substituted by the time gap between two adjacent bunches. Further, the samples are also divided into three different categories: (i) Isolated Samples (One sample bunch) (ii) Border Samples (the first and last samples of each bunch) and (iii) Central Samples. Therefore, total five Huffman tables are required for data compression. The following macros are used for compression. The method *RawDataAltro* of the **AliTPCDDLRawData** class creates a file *AltroFormatDDL.dat*. This file is created for two purposes. First, it is used to build the frequency tables for 5 different data categories from which the Huffman tables are generated by calling the *CreateTables()* method of the **AliTPCCompression** class. Second, it can also be used for debugging purposes. For example, this ALTRO file can be converted into a text file using the method *ReadAltroFile()* of the **AliTPCCompression** class. To create the files with compressed data the macro executes the method *RawDataCompDecompress()* of the class **AliTPCDDLRawData** (files are named TPCslice1.comp, TPCslice2.comp and so on). The same method is also used to decompress the files, just setting to 1 the second input parameter (the first parameter indicates the number of files to be created). The *CreateTables("AltroFormatDDL.dat",NumTable)* of the **AliTPCCompression** class is used to generate *NumTable* of Huffman Tables which are used during compression. This method builds 5 frequency tables from the raw data file *AltroFormatDDL.dat* and builds the Huffman tables in binary format. The table *Table0.dat* is for the bunch length distribution, *Table1.dat* is for time gap distribution, *Table2.dat* is for one sample bunch, *Table3.dat* is for central sample and *Table4.dat* is for border sample bunch. For detail see the appendix. These Huffman binary tables can also be generated by using the method *CreateTablesFromTxtFiles(NumTable)*. This method requires 5 normalized frequency tables *Tablen.txt* where n varies from 0 to 4 and these tables need to be supplied.

A brief description of various methods of **AliTPCDDLRawData** class are given below.

5.3.3 AliTPCDDLRawData class

This class is used to produce rawdata files to be used during future Data Challenges. It contains the following public methods:

- `void  RawDataAltro()`

  This method creates a binary file *AltroFormatDDL.dat* in Altro Format from the temporary DDL file *AliTPCDDL.dat*. A program reads each data record and builds the ALTRO block by identifying the amplitude values in bunches which belong to a given pad. Since the data is zero suppressed, a group of amplitudes belonging to the same pad (same sector, sub sector, pad and row number) form a bunch if and only if the time bin values are continuous. A discontinuity in the time bin either indicates the begining of a new bunch in the same pad (same ALTRO block) or the beginning of a new bunch in a new pad (new ALTRO block). Accordingly, the trailer is invoked when an ALTRO block ends.

- `void  RawData(Int_t LDCsNumber)`

  This method works in a similar way as that of previous one. In addition, it creates a set of files containing DDL rawdata. A DDL contains a mini header followed by a set of ALTRO data blocks coming from one DDL. A DDL is a set of ALTRO blocks having same sector and sub-sector addresses. Although, a mini header appears at the begining of each DDL, in practice, it is filled after the ALTRO blocks are written into the DDL in order to know the size of the data blocks. Therefore, the to create a DDL file, the following steps are followed:

  1. Skip a block of 12 bytes (dimension of the mini header).
  2. Write all the ALTRO data blocks of a given DDL .
  3. Go back to the begining of the DDL and write the mini header.
  4. Jump to the end of the DDL.

  If the number of LDC are same as the number of DDL, each LDC file contains a DDL block. However, if LDC number **LDCsNumber** is less than the DDL number **216**, each LDC file contains **216/LDCsNumber** of DDL blocks.

- `Int_t RawDataCompDecompress(Int_t LDCsNumber,Int_t Comp=0)`

  This method is used to create DDL files in the compressed mode ($Comp = 0$). It can also be used in $Comp \neq 0$ mode to decompress the compressed data. In the compressed mode, each ALTRO data block is copied into a temporary file and is compressed using the *CompressDataOptTables()* method of the compression class as described in this note. Finally, compressed output is copied into the output file. In the last step, the mini header is also written into the file. The decompression works in the same way which uses the method *DecompressDataOptTables()*.

- `void  RawDataAltroDecode(Int_t LDCsNumber,Int_t Comp=0)`

  This method is used for debugging purposes. Depending on the compression mode, it loops over all the LDC files (uncompressed $Comp = 0$ or compressed , $Comp = 1$) and merges all into one binary file *AltroDDLRecomposed.dat* or *AltroDDLRecomposedDec.dat* depending on the *Comp* flag after removing all the mini headers. In case of uncompressed LDC files, the output file should be equivalent to the binary file created by *RawDataAltro* method. If this doesn't happen, there could be an error in the code.

5.4 Steps to Follow

1. Create *galice.root*
2. Create Digits *AliTPCHits2Digits.C*
3. Create DDL Mappinig, Execute Macro *AliTPCDDL.C("galice.root")*. Creates the files **AliTPCDDL.dat**.
4. Execute Macro *AliTPCDDLRawData.C*.

6 Appendix A:Huffman Compression

The compression algorithm for TPC data is based on lossless Huffman coding [?]. The codes generated using this procedure are called Huffman codes. These codes are prefix codes and are optimum for a given model. The Huffman procedure is based on two observations regarding optimum prefix codes.

1. In an optimum code, symbols that occur more frequently (have higher probability of occurrence) will have shorter codewords than symbols that occur less frequently.
2. In an optimum code, the two symbol that occur less frequently will have the same length.

Compression is based on two elements, a model and a compression algorithm; what is important for a good compression, is to have a model that describes the nature of the data. The better data is described by the model, the better is the compression factor. As shown in Figure 13, the TPC data are distributed in bunches. Therefore, the ALTRO data can be divided into 5 categories, three for Samples, one for bunch length and one for time bins or time gap between two successive bunches.

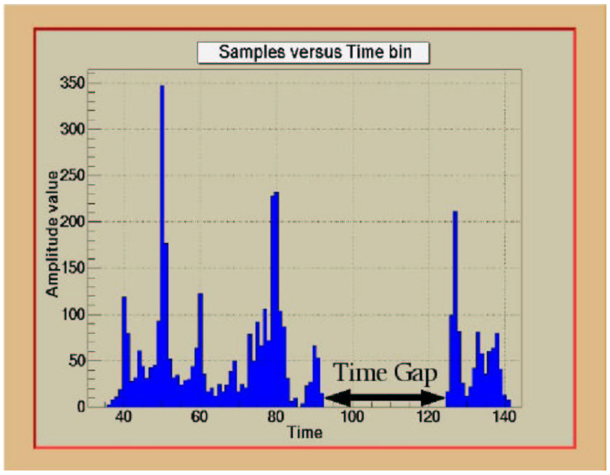


Figure 13: Bunches VS Time bin.

The 5 Huffman tables used are as follows:

1. Bunch Length
2. Time (Gap between two successive bunch)
3. Isolated Sample (Sample of single bunch)
4. Central Samples
5. Border Samples

6.0.1 AliTPCHOptimizedTables.C

This macro can be used to compress an AlTRO file (AltroFormat.dat) using the Huffman tables created with one of the following options:

1. Either calling the method *CreateTables*. This method builds 5 frequency tables from the AltroFormat.dat file which are finally used to generate the Huffman coding tables.
2. Or calling the method *CreateTablesFromTxtFiles*. This method builds the Huffman tables using the frequency tables provided by the user. This is useful to create Huffman tables using an single set of optimized tables.
3. Or using some formula.

The useful classes and methods are described below.

6.0.2 AliTPCHNode class

The class **AliTPCHNode** is the base element for building an Huffman tree. The trees are the crucial elements of Huffman algorithm from which the look up tables are derived. Each object of this class represent a node of the tree and contains the following information: Symbol, Frequency of the associate symbol, a pointer to the left child and a pointer to the right child. An Huffman tree is a binary tree in which each internal node has always two children, as shown in Figure 14.

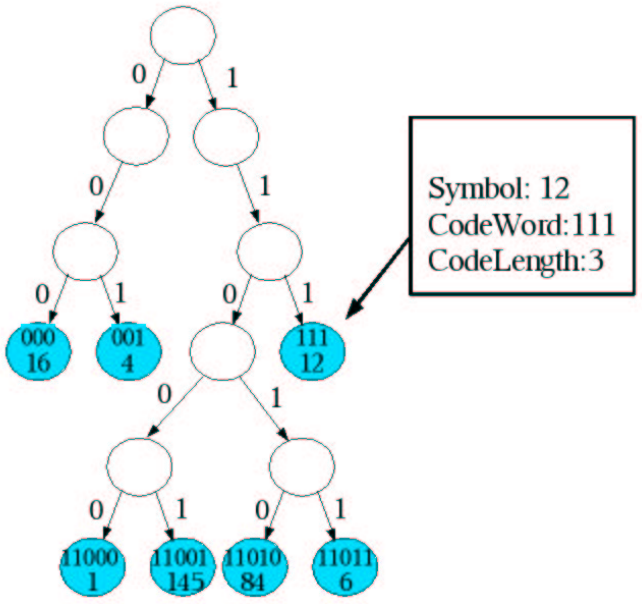


Figure 14: Huffman tree.

6.0.3 AliTPCHTable class

This class contains the implementation of the Huffman algorithm, that makes the following steps for each data class:

1. Get the frequencies of the symbols.
2. Create the tree.
3. Visit the tree and create the look up table.

In our case, each symbol is a 10 bits word, so the range goes from 0 up to $2^{10}-1$, for a total of 1024 symbols. Every table has as many rows as the number of symbols. Each symbols has a codeword and code length. Basically an object of this class represents a table whose dimension is specified by the input parameter of the constructor. Some of the useful methods of this class are as follows.

- `void PrintTable()`

  This method prints on the screen the content of the table (symbol,codeword and code length) in a user-friendly style.

- `Int_t GetFrequencies(const char* fname)`

  This method is used to get the frequencies of the symbols directly from the input file which must be Altro format compliant.

- `Int_t SetFrequency(const Int_t Val)`

  This method is used to increase by one the frequency value of the symbol specified in input.

- `Int_t StoreFrequencies(const char *fname)`

  This method is used to store the frequencies of the symbols in the specified binary file.

- `Int_t BuildHTable()`

  This method, first builds the Huffman tree starting from the frequencies and then creates the corresponding table.

- `Double_t GetEntropy()`

  This method is used to get the entropy value, and it must be used after *GetFrequencies()* or *SetFequency()* and before *BuildHTable()*. This is because the entropy is calculated on the ground of the frequencies; frequencies values that are not any more available after the execution of the *BuildHTable()* method.

- `void SetVerbose(Int_t val)`

  This method is used to print out on the screen some messages during the program execution. Verbose level can be set in two different position: 0 silent, 1 output messages.

6.0.4 AliTPCCompression class

The methods of this class are used either to compress or decompress the Altro-File using the model based on five different tables.

- `Int_t CompressData(AliTPCHTable* table[],Int_t NumTable,`
  `const char* fSource,const char* fDest)`

  This method is used to compress an Altro file specified in input (fSource) using specific tables calculated considering the file itself. To make the decompression possible, the tables are stored at the beginning of the compressed file. If the frequency distribution is known a priori, it is not necessary to store the corresponding Huffman tables in the files.

- `Int_t CompressDataOptTables(Int_t NumTable,const char* fSource,`
  `const char* fDest)`

  This method compress the input file, using a variable number of static tables that are a priori calculated and optimized for compressing data coming from the TPC detector. The implemented algorithm is parametrized according to the number of tables, but it works only if the input file is Altro format compliant.

- `Int_t DecompressData(Int_t NumTables,const char* fname,`
  `char* fDest="SourceDecompressed.dat")`

  This method decompress a file that has been compressed using the *CompressData()* method.

- `Int_t DecompressDataOptTables(Int_t NumTables,const char* fname,`
  `char* fDest="SourceDecompressed.dat")`

  This method decompress a file that has been compressed using the *CompressDataOptTables()* method.

- `Int_t FillTables(const char* fSource,AliTPCHTable* table[],`
  `const Int_t NumTables)`

  This method is called also in *CreateTables()* and it is used to get the frequencies of the symbols for all the necessary tables.
  Besides this method creates a text file, named *Statistics* which contains many information, like for instance, the number of word, number of trailer, and the size in term of bytes of each category. This file is then used to store compression information like entropy and different compression factors.

- `Int_t CreateTables(const char* fSource,const Int_t NumTables)`

  This method is used to create and stored the tables. It has to be executed if the optimized tables are not available.
  The frequencies are calculated considering the input file and the number of tables.
  This methods also, stores in the *Statistics* file the value of the entropy for each category.

- `void SetVerbose(Int_t val)`

  This method is used to set the verbose level. It can assume three different values: 0 silent, 1 some messages, 2 pedantic output.

- `void ReadAltroFormat(char* fileOut,char* fileIn)`

  This method read an Altro format file (FileIn parameter) and creates a text file (fileOut parameter) which contains the same information but stored in a user-friendly mode. This feature is useful for code debugging and know, which data, has been processed, but pay attention that the text file usually is quite big (several MB) and takes a while be be generated.

Talking about the implementation it has to be said, that all the methods work directly or indirectly with the Altro format and this makes this class absolutely inappropriate to compress/decompress data of a different format. As it has been said, that compression is based on Huffman algorithm with five different tables; to compress a word using this method it is necessary to know, which table has to be utilized. The answer is provide by the *NextTable()* private method, that according to the real state, gives the number of the table that has to be used to compress the next word.
Now the tables are numbered from 0 to 4 as follow:

- 0 for Bunch length
- 1 for Time Bin
- 2 for 1-sample bunch
- 3 for Central samples
- 4 for Border samples

So a bunch length value has to be coded, using the table 0 and the next value will be a time bin value (Compression is done backward wise) which will be coded using the table 1. Now according to the bunch length information, we know, if after the time bin there is a bunch of one sample (table 2), or the beginning of multiple sample bunch (table 4). The mechanism goes on till the end of the bunch, and then it starts again from the bunch length table (Information is stored bunch by bunch).
So exploiting the specific Altro structure, it is possible to know which table has to be used each time.
The *NextTable()* method is used extensively around the code, and it has to be rewritten in case the model change.
Let's see now, how the *FillTables()* method works. Even this method is model dependent, and as it has been said, it fills all the tables with the frequencies of the data contained in the input file. To get the frequencies the input file, has to be read backward wise, but in this way, we get the bunches in the opposite order to respect to the time bin evolution. So to calculate the time gap starting from the fist bunch, it is necessary to save all the values coming from one packet (data from one pad), in a temporary buffer (named packet) and then scan it starting from the end, using the auxiliary array *TimePos* that provides direct access to the time bin cells of the buffer (Figure 15).
Now the same work for the time gap calculation, is done also in the compression methods (*CompressDataOptTable()* and *CompressData()*) making them model dependent. So this suggests to create a new class to manage a variable number of tables and make the compression methods independent from the specific number of tables and model.
The implementation of the two methods for compressing a file is very similar, the only thing that is different is that using the *CompressData()* method all the tables are stored in the compressed file; so let's analyze only the *CompressDataOptTable()* method that is the one actually used.

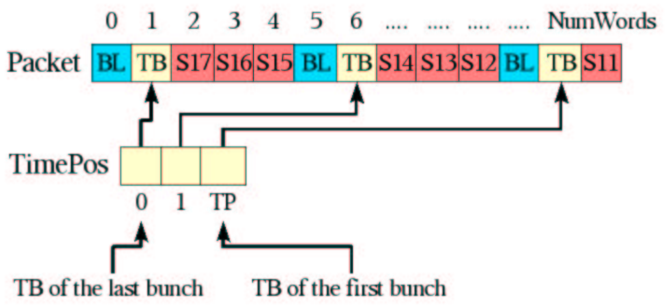

0 1 2 3 4 5 6 .... NumWords
Packet
BL TB S17 S16 S15 BL TB S14 S13 S12 BL TB S11
TimePos
0 1 TP
TB of the last bunch
TB of the first bunch

Figure 15: Time bin calculation.

The first thing that this method does, is to restore in memory the tables from the files, calling the method *RetrieveTables()*. This last method has the particularity that every codeword is not retrieved as it is stored into the file, but it is retrieved inverting the order of its digits. For instance if the codeword is 12345 it is retrieved 54321. This is done because, 54321 will be stored in the compressed file, and not the original codeword, making the decompression routine faster in the codeword recognition phase.
In fact even the decompression routine works backward wise to reestablish the original order in the decompressed file. Reading backward cannot be avoided using memory buffers due to the size of the Altro file.
Once that the tables are restored the compression continues reading the input file from the end to the beginning and compressing the data working pad by pad. This means that all the data coming from one pad are stored in an internal buffer called Packet, then all the time gaps are calculated using the same technique previously described and finally every word is coded and stored in the compressed file using the method *StoreWord()*.
The trailer, that complete a packet, is stored directly without compression.
The method *StoreWord()* doesn't store the codewords directly in the output file, but it stores them in an internal buffer of 32 bits. This is necessary because each codeword, expressed by an ULong_t variable, has to be stored in the output file using only the number of bits specified by the associate code length variable. So all the codewords are packed using an internal buffer who works as follow:
The codeword are inserted directly in the buffer till there is space, then when the buffer is full it is saved in the output file and emptied. If there isn't enough space left in the buffer for a new codeword, the codeword is splitted in two parts in such a way that the first part fits exactly in the buffer, that becomes full and it is saved in the output file. Then the buffer is emptied and used to store the second part of the codeword which goes at the beginning of the buffer.
The *CompressDataOptTable()* method store also some useful information in the *Statistics* file. For instance it stores the compression factor, calculated as ratio of the original file size and the size of the compressed file.

6.1 Decompression of Raw Data

AliTPCCompression class provides also two methods which can be used to decompress a file. They are *DecompressData()* and *DecompressDtaOptTable()*; the first one is used to decompress a file created using the method *CompressData()* while the second one is used to decompress a file created using the *CompressDataOptTable()* method.
The difference between them is that while *CompressDataOptTable()* expects the tables for decoding the words as a sequence of binary file, *DecompressData()* expects all the tables stored at the beginning of the compressed file.
Now, since the compression methods are very similar, even these methods for uncompressing are similar, so here only the *CompressDataOptTable()* method is described.
To decompress a file it is necessary to know the tables that have been used to codify the words, so the first step of the decompression routine is to get all the tables. This is done, calling the method *CreateTreesFromFile()*, as the name suggests instead of creating look up tables it creates an Huffman tree for each table (Figure 16. Note that only the leaves contain symbols).

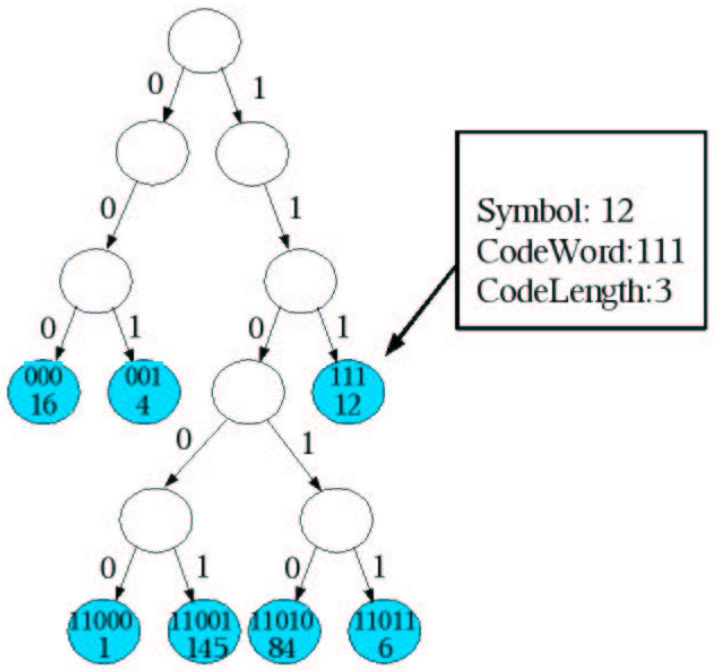


Figure 16: Huffman tree.

Using trees the decompression is faster than using tables, because the routine doesn't know anything about the length the codewords stored in the compressed file and using tables it has to look for all possible prefixes of every codeword in all the tables. This is a time consuming operation that is avoided using trees. After that, trees have been created the decompression routine makes some initializations and then it works looping over the packets (the number of packets

is stored at the end of the file).
For each packet the first step is to read the trailer (it is not coded) getting so the number of words contained inside the packet. At this point the method loops over the words of the packet decoding and saving them in a new Altro data file created in the initialization phase.
The decoding of words is implemented in the *GetDecodeWord()* method who analyzes bit by bit the information read from the compressed file. To see how this is done let's suppose that we have to decode a bunch length codeword and this is the only information that we have, at this stage we don't know anything about the length. So first a pointer to the root of the tree associates with the bunch length table is set, and then one bit from the input file is read; now according to the value of this bit, the pointer is moved to the right or left child; than an other bit is read and the pointer is updated using the same rule. This is repeated till a leaf is reached. The decoded symbol can be read directly from the reached leaf.
Since we are reading the file backward, this explain why codeword are stored in the inverse way.

7 Appendix B

DDL's mapping for ITS.

DDL number	Modules
0	0, 1, 4, 5, 80, 81, 84, 85, 88, 89, 92, 93
1	2, 3, 6, 7, 82, 83, 86, 87, 90, 91, 94, 95
2	8, 9,12,13, 96, 97,100,101,104,105,108,109
3	10,11,14,15, 98, 99,102,103,106,107,110,111
4	16,17,20,21,112,113,116,117,120,121,124,125
5	18,19,22,23,114,115,118,119,122,123,126,127
6	24,25,28,29,128,129,132,133,136,137,140,141
7	26,27,30,31,130,131,134,135,138,139,142,143
8	32,33,36,37,144,145,148,149,152,153,156,157
9	34,35,38,39,146,147,150,151,154,155,158,159
10	40,41,44,45,160,161,164,165,168,169,172,173
11	42,43,46,47,162,163,166,167,170,171,174,175
12	48,47,50,51,176,177,180,181,184,185,188,189
13	50,51,54,55,178,179,182,183,186,187,190,191
14	56,57,60,61,192,193,196,197,200,201,204,205
15	58,59,62,63,194,195,198,199,202,203,206,207
16	64,65,68,69,208,209,212,213,216,217,220,221
17	66,67,70,71,210,211,214,215,218,219,222,223
18	72,73,76,77,224,225,228,229,232,233,236,237
19	74,75,78,79,226,227,230,231,234,235,238,239

Table 2: Silicon pixel detector mapping.

DDL number	Modules
20	240,241,242,246,247,248,252,253,254,258,259, 260,264,265,266,270,271,272,276,277,278
21	243,244,245,249,250,251,255,256,257,261,262, 263,267,268,269,273,274,275,279,280,281
22	282,283,284,288,289,290,294,295,296,300,301, 302,306,307,308,312,313,314,318,319,320
23	285,286,287,291,292,293,297,298,299,303,304, 305,309,310,311,315,316,317,321,322,323
24	324,325,326,327,332,333,334,335,340,341,342, 343,348,349,350,351,356,357,358,359,364,365
25	328,329,330,331,336,337,338,339,344,345,346, 347,352,353,354,355,360,361,362,363,368,369
26	366,367,372,373,374,375,380,381,382,383,388, 389,390,391,396,397,398,399,404,405,406,407
27	370,371,376,377,378,379,384,385,386,387,392, 393,394,395,400,401,402,403,408,409,410,411
28	412,413,414,415,420,421,422,423,428,429,430, 431,436,436,438,439,444,445,446,447,452,453
29	416,417,418,419,424,425,426,427,432,433,434, 435,440,441,442,443,448,449,450,451,456,457
30	454,455,460,461,462,463,468,469,470,471,476, 477,478,479,484,485,486,487,492,493,494,495
31	458,459,464,465,466,467,472,473,474,475,480, 481,482,483,488,489,490,491,496,497,498,499

Table 3: Silicon Drift Detector mapping.

DDL number	Modules
32	500, 501, 502, 503, 504, 505, 506, 507, 508, 509, 510, 522, 523, 524, 525, 526, 527, 528, 529, 530, 531, 532, 1204,1205,1206,1207,1208,1209,1210,1211,1212,1213,1214, 1226,1227,1228,1229,1230,1231,1232,1233,1234,1235,1236, 1248,1249,1250,1251,1252,1253,1254,1255,1256,1257,1258,1259, 2098,2099,2100,2101,2102,2103,2104,2105,2106,2107,2108,2109, 2123,2124,2125,2126,2127,2128,2129,2130,2131,2132,2133,2134, 2148,2149,2150,2151,2152,2153,2154,2155,2156,2157,2158,2159, 2173,2174,2175,2176,2177,2178,2179,2180,2181,2182,2183,2184
33	566, 567, 568, 569, 570, 571, 572, 573, 574, 575, 576, 588, 589, 590, 591, 592, 593, 594, 595, 596, 597, 598, 610, 611, 612, 613, 614, 615, 616, 617, 618, 619, 620, 1273,1274,1275,1276,1277,1278,1279,1280,1281,1282,1283,1284, 1298,1299,1300,1301,1302,1303,1304,1305,1306,1307,1308,1309, 1323,1324,1325,1326,1327,1328,1329,1330,1331,1332,1333,1334, 1348,1349,1350,1351,1352,1353,1354,1355,1356,1357,1358,1359, 1373,1374,1375,1376,1377,1378,1379,1380,1381,1382,1383,1384
34	632, 633, 634, 635, 636, 637, 638, 639, 640, 641, 642, 654, 655, 656, 657, 658, 659, 660, 661, 662, 663, 664, 676, 677, 678, 679, 680, 681, 682, 683, 684, 685, 686, 698, 699, 700, 701, 702, 703, 704, 705, 706, 707, 708, 720, 721, 722, 723, 724, 725, 726, 727, 728, 729, 730, 1398,1399,1400,1401,1402,1403,1404,1405,1406,1407,1408,1409, 1423,1424,1425,1426,1427,1428,1429,1430,1431,1432,1433,1434, 1448,1449,1450,1451,1452,1453,1454,1455,1456,1457,1458,1459, 1473,1474,1475,1476,1477,1478,1479,1480,1481,1482,1483,1484
35	742, 743, 744, 745, 746, 747, 748, 749, 750, 751, 752, 764, 765, 766, 767, 768, 769, 770, 771, 772, 773, 774, 786, 787, 788, 789, 790, 791, 792, 793, 794, 795, 796, 808, 809, 810, 811, 812, 813, 814, 815, 816, 817, 818, 1498,1499,1500,1501,1502,1503,1504,1505,1506,1507,1508,1509, 1523,1524,1525,1526,1527,1528,1529,1530,1531,1532,1533,1534, 1548,1549,1550,1551,1552,1553,1554,1555,1556,1557,1558,1559, 1573,1574,1575,1576,1577,1578,1579,1580,1581,1582,1583,1584, 1598,1599,1600,1601,1602,1603,1604,1605,1606,1607,1608,1609
36	830, 831, 832, 833, 834, 835, 836, 837, 838, 839, 840, 852, 853, 854, 855, 856, 857, 858, 859, 860, 861, 862, 874, 875, 876, 877, 878, 879, 880, 881, 882, 883, 884, 896, 897, 898, 899, 900, 901, 902, 903, 904, 905, 906, 1623,1624,1625,1626,1627,1628,1629,1630,1631,1632,1633,1634, 1648,1649,1650,1651,1652,1653,1654,1655,1656,1657,1658,1659, 1673,1674,1675,1676,1677,1678,1679,1680,1681,1682,1682,1684, 1698,1699,1700,1701,1702,1703,1704,1705,1706,1707,1708,1709, 1723,1724,1725,1726,1727,1728,1729,1730,1731,1732,1733,1734
37	918, 919, 920, 921, 922, 923, 924, 925, 926, 927, 928, 940, 941, 942, 943, 944, 945, 946, 947, 948, 949, 950, 962, 963, 964, 965, 966, 967, 968, 969, 970, 971, 972, 984, 985, 986, 987, 988, 989, 990, 991, 992, 993, 994, 1748,1749,1750,1751,1752,1753,1754,1755,1756,1757,1758,1759, 1773,1774,1775,1776,1777,1778,1779,1780,1781,1782,1783,1784, 1798,1799,1800,1801,1802,1803,1804,1805,1806,1807,1808,1809, 1823,1824,1825,1826,1827,1828,1829,1830,1831,1832,1833,1834, 1848,1849,1850,1851,1852,1853,1854,1855,1856,1857,1858,1859

Table 4: Silicon Strip Detector mapping.

DDL number	Modules
38	1006,1007,1008,1009,1010,1011,1012,1013,1014,1015,1016, 1028,1029,1030,1031,1032,1033,1034,1035,1036,1037,1038, 1050,1051,1052,1053,1054,1055,1056,1057,1058,1059,1060, 1072,1073,1074,1075,1076,1077,1078,1079,1080,1081,1082, 1094,1095,1096,1097,1098,1099,1100,1101,1102,1103,1104, 1873,1874,1875,1876,1877,1878,1879,1880,1881,1882,1883,1884, 1898,1899,1900,1901,1902,1903,1904,1905,1906,1907,1908,1909, 1923,1924,1925,1926,1927,1928,1929,1930,1931,1932,1933,1934, 1948,1949,1950,1951,1952,1953,1954,1955,1956,1957,1958,1959
39	1116,1117,1118,1119,1120,1121,1122,1123,1124,1125,1126, 1138,1139,1140,1141,1142,1143,1144,1145,1146,1147,1148, 1160,1161,1162,1163,1164,1165,1166,1167,1168,1169,1170, 1182,1183,1184,1185,1186,1187,1188,1189,1190,1191,1192, 1973,1974,1975,1976,1977,1978,1979,1980,1981,1982,1983,1984, 1998,1999,2000,2001,2002,2003,2004,2005,2006,2007,2008,2009, 2023,2024,2025,2026,2027,2028,2029,2030,2031,2032,2033,2034, 2048,2049,2050,2051,2052,2053,2054,2055,2056,2057,2058,2059, 2073,2074,2075,2076,2077,2078,2079,2080,2081,2082,2083,2084
40	511, 512, 513, 514, 515, 516, 517, 518, 519, 520, 521, 533, 534, 535, 536, 537, 538, 539, 540, 541, 542, 543, 1215,1216,1217,1218,1219,1220,1221,1222,1223,1224,1225, 1237,1238,1239,1240,1241,1242,1243,1244,1245,1246,1247, 1260,1261,1262,1263,1264,1265,1266,1267,1268,1269,1270,1271,1272, 2110,2111,2112,2113,2114,2115,2116,2117,2118,2119,2120,2121,2122, 2135,2136,2137,2138,2139,2140,2141,2142,2143,2144,2145,2146,2147, 2160,2161,2162,2163,2164,2165,2166,2167,2168,2169,2170,2171,2172, 2185,2186,2187,2188,2189,2190,2191,2192,2193,2194,2195,2196,2197
41	555, 556, 557, 558, 559, 560, 561, 562, 563, 564, 565, 577, 578, 579, 580, 581, 582, 583, 584, 585, 586, 587, 599, 600, 601, 602, 603, 604, 605, 606, 607, 608, 609, 621, 622, 623, 624, 625, 626, 627, 628, 629, 630, 631, 1285,1286,1287,1288,1289,1290,1291,1292,1293,1294,1295,1296,1297, 1310,1311,1312,1313,1314,1315,1316,1317,1318,1319,1320,1321,1322, 1335,1336,1337,1338,1339,1340,1341,1342,1443,1344,1345,1346,1347, 1360,1361,1362,1363,1364,1365,1366,1367,1368,1369,1370,1371,1372, 1385,1386,1387,1388,1389,1390,1391,1392,1393,1394,1395,1396,1397
42	643, 644, 645, 646, 647, 648, 649, 650, 651, 652, 653, 665, 666, 667, 668, 669, 670, 671, 672, 673, 674, 675, 687, 688, 689, 690, 691, 692, 693, 694, 695, 696, 697, 709, 710, 711, 712, 713, 714, 715, 716, 717, 718, 719, 731, 732, 733, 734, 735, 736, 737, 738, 739, 740, 741, 1410,1411,1412,1413,1414,1415,1416,1417,1418,1419,1420,1421,1422, 1435,1436,1437,1438,1439,1440,1441,1442,1443,1444,1445,1446,1447, 1460,1461,1462,1463,1464,1465,1466,1467,1468,1469,1470,1471,1472, 1485,1486,1487,1488,1489,1490,1491,1492,1493,1494,1495,1496,1497
43	753, 754, 755, 756, 757, 758, 759, 760, 761, 762, 763, 775, 776, 777, 778, 779, 780, 781, 782, 783, 784, 785, 797, 798, 799, 800, 801, 802, 803, 804, 805, 806, 807, 819, 820, 821, 822, 823, 824, 825, 826, 827, 828, 829, 1510,1511,1512,1513,1514,1515,1516,1517,1518,1519,1520,1521,1522, 1535,1536,1537,1538,1539,1540,1541,1542,1543,1544,1545,1546,1547, 1560,1561,1562,1563,1564,1565,1566,1567,1568,1569,1570,1571,1572, 1585,1586,1587,1588,1589,1590,1591,1592,1593,1584,1595,1596,1597, 1610,1611,1612,1613,1614,1615,1616,1617,1618,1619,1620,1621,1622

Table 5: Silicon Strip Detector mapping.

DDL number	Modules
44	841, 842, 843, 844, 845, 846, 847, 848, 849, 850, 851, 863, 864, 865, 866, 867, 868, 869, 870, 871, 872, 873, 885, 886, 887, 888, 889, 890, 891, 892, 893, 894, 895, 907, 908, 909, 910, 911, 912, 913, 914, 915, 916, 917, 1635,1636,1637,1638,1639,1640,1641,1642,1643,1644,1645,1646,1647, 1660,1661,1662,1663,1664,1665,1666,1667,1668,1669,1670,1671,1672, 1685,1686,1687,1688,1689,1690,1691,1692,1693,1694,1695,1696,1697, 1710,1711,1712,1713,1714,1715,1716,1717,1718,1719,1720,1721,1722, 1735,1736,1737,1738,1739,1740,1741,1742,1743,1744,1745,1746,1747
45	929, 930, 931, 932, 933, 934, 935, 936, 937, 938, 939, 951, 952, 953, 954, 955, 956, 957, 958, 959, 960, 961, 973, 974, 975, 976, 977, 978, 979, 980, 981, 982, 983, 995, 996, 997, 998, 999,1000,1001,1002,1003,1004,1005, 1760,1761,1762,1763,1764,1765,1766,1767,1768,1769,1770,1771,1772, 1785,1786,1787,1788,1789,1790,1791,1792,1793,1794,1795,1796,1797, 1810,1811,1812,1813,1814,1815,1816,1817,1818,1819,1820,1821,1822, 1835,1836,1837,1838,1839,1840,1841,1842,1843,1844,1845,1846,1847, 1860,1861,1862,1863,1864,1865,1866,1867,1868,1869,1870,1871,1872
46	1017,1018,1019,1020,1021,1022,1023,1024,1025,1026,1027, 1039,1040,1041,1042,1043,1044,1045,1046,1047,1048,1049, 1061,1062,1063,1064,1065,1066,1067,1068,1069,1070,1071, 1083,1084,1085,1086,1087,1088,1089,1090,1091,1092,1093, 1105,1106,1107,1108,1109,1110,1111,1112,1113,1114,1115, 1885,1886,1887,1888,1889,1890,1891,1892,1893,1894,1895,1896,1897, 1910,1911,1912,1913,1914,1915,1916,1917,1918,1919,1920,1921,1922, 1935,1936,1937,1938,1939,1940,1941,1942,1943,1944,1945,1946,1947, 1960,1961,1962,1963,1964,1965,1966,1967,1968,1969,1970,1971,1972
47	1127,1128,1129,1130,1131,1132,1133,1134,1135,1136,1137, 1149,1150,1151,1152,1153,1154,1155,1156,1157,1158,1159, 1171,1172,1173,1174,1175,1176,1177,1178,1179,1180,1181, 1193,1194,1195,1196,1197,1198,1199,1200,1201,1202,1203, 1985,1986,1987,1988,1989,1990,1991,1992,1993,1994,1995,1996,1997, 2010,2011,2012,2013,2014,2015,2016,2017,2018,2019,2020,2021,2022, 2035,2036,2037,2038,2039,2040,2041,2042,2043,2044,2045,2046,2047, 2060,2061,2062,2063,2064,2065,2066,2067,2068,2069,2070,2071,2072, 2085,2086,2087,2088,2089,2090,2091,2092,2093,2094,2095,2096,2097

Table 6: Silicon Stip Detector mapping.