ATLAS TDAQ DataCollection

Format of the data files written by EventStorage library of ATLAS TDAQ

Document NoteNumber: 066
Document Version: 6
Document EDMS ID: ATL-DQ-EN-0021, 580290
Document Date: 01 April 2011
Document Status: Draft

Abstract

This note documents the format of RAW data files produced by the *EventStorage* library of the ATLAS TDAQ system. Such files contain events, which comply to the eformat specs, and meta-data, which are described in this note.
The *EventStorage* package is part of *tdaq-common* releases and is available to ATLAS software compiled in *tdaq* and *offline* releases. The library provides APIs for writing and reading the event data and metadata. DAQ applications are the main custormer of the writing API, whereas the reading API is used mostly by Athena services in the offline environment. Dynamically loaded "plugins" allow the reading of raw data files directly from Castor and dCache mass storage systems. In the main body of this note we describe the raw file format, the file name convetion (changed in *tdaq-common-01-10-00*), the libraries and their functionality.

In the Appendix we describe a new feature of the library: concatenation of many individual raw files into superstructures, called *merged raw data files*. The format of such files is defined here. Compliance to this format, guaratnies that merged files can be read by the usual *EventStorage* library tools in a transparent fashion for the user. There is also tools to unpack or just look at the file-content of a merged raw file.

Authors:
Hans Peter Beck, Szymon Gadomski, Kostas Kordas, Cyril Topfel, Wainer Vandelli

Institute:
LHEP, University of Bern CERN, Geneva

Table 1: Document Change Record

Title: ATLAS TDAQDataCollection Format of the data files written by EventStorage library of ATLAS TDAQ			
ID: TDAQ-DataCollection-66			
Issue	Revision	Date	Comment
1	0	2003	HP : creation
2	0	07-Apr-2005	Szymon: major update
3	0	10-Nov-2006	Szymon: addition of user-defined meta-data
4	0	04-Apr-2007	Szymon: A major update
5	0	05-Mar-2009	K. Kordas, C. Topfel: major update: redesgin interfaces, introduce merged files, new naming convention
6	0	01-Apr-2011	W.Vandelli: introduce data compression support

1 Raw data files in ATLAS

The RAW data files, which are called "byte stream" in the context of the offline processing, are written by ATLAS TDAQ applications. At the end of the data acquisition chain, after the Event Filter selection, the Sub-Farm Output (SFO) application of the data-logger system writes events into RAW data files. These files are then sent Offline for permanent storage, full reconstruction and further processing. For commissioning and testing purposes also other TDAQ applications, such as the Readout System (ROS) and the Event Builder (Sub-Farm Input, a.k.a SFI) can write RAW data files using the *EventStorage* library.

The Offline Software of ATLAS also uses the library to produce the "byte stream" files containing simulated ATLAS data in the RAW format. The library provides also the means to read RAW files, with the offline, via Athena services, being the main client of the related API.

2 Format of the data files

2.1 History of data file formats

In this note we describe the RAW file format version 5. Version 1 was used by the DAQ system for the data collected in the beam tests of 2003. The format version 2, was introduced in January 2004. That format was used in DataFlow release DF-00-07-00 and in all the following releases, including those used in the Combined Beam Test of ATLAS in 2004.

From version 2, we arrived to the current format (version 5, described here) in a smooth evolution proccess. In February 2006 a block of user-defined metadata was added to the file format. In the TDAQ system the user-defined metadata is introduced by the operator in a window provided by the Interactive Graphical User Interface (IGUI) of the ATLAS Online Software. The block of metadata strings was initially optional, i.e. it may or may not have been present in the data files. The change was introduced without changing of the format version number.

Since January 2007, a "Global Unique ID" (GUID) is always added to the files (see Section 2.3). The addition was done by making use of the existing block of metadata strings, avoiding changes of the file format; the only "side-effect" was that the block of metadata strings was always present.

In January 2008, the detector bit-mask (in the "run parameters" block) was expanded from one 32-bit word to become two 32-bit words. The API returns one 64-bit integer when asked for the DetectorMask value corresponding to the file.

In October 2008, three changes were introduced: i) support for writing and reading "merged RAW files" was added. Merged files are multiple RAW files concatenated and prepended with a header describing the content (see Appendix A.1); ii) the *RawFileName* class replaced the *AgreedFileName* function, in order to facilitate the implementation of the RAW filename convention (agreed with the Tier0) for all EventStorage client applications; iii) the projectTag, StreamTag, and the LuminosityBlockNumber strings where added (if available) to the metadata strings.

2.2 The current format

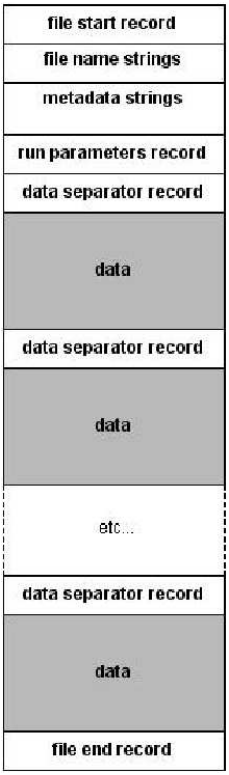

Figure 1: Positions of meta-data records in files. All files have the same structure.

Figure 1 presents the file structure in the current file format version 5. The fields shown in grey represent the event data, encoded using the eformat library [1]. For the EventStorage library though, the content of these "data blocks" is irrelevant. The fields shown in white represent metadata, shown in detail in Table 2.

Comments on the metadata table:

Table 2: Details of the metadata

Field	Type	Comment or value
file start record		
marker	u32	0x1234aaaa (1)
record_size	u32	size of this record (1)
version	u32	value is 5 in this version
file_number	u32	number of the file in a file-sequence (see Section 3)
date	u32	when the file was opened (2)
time	u32	when the file was opened (2)
sizeLimit_dataBlocks	u32	maximum file size in data blocks (3)
sizeLimit_MB	u32	maximum file size in MB (3)
file name strings		
marker	u32	0x1234aabb
length	u32	size of the following string
appName	string	as in file name (4 & Section 3)
length	u32	size of the following string
fileNameCore	string	as in file name (4 & Section 3)
metadata strings (5)		
marker	u32	0x1234aabc
length	u32	number of free metadata strings
length	u32	length of next string
GUID	string	Global Unique Identifier of the file (5)
length	u32	length of next string (6)
metadata string	string	(<tag>=value) as published in the online Information Service (IS)
length	u32	length of next string (7)
streamTag	string	as in the file name in the form StreamName_StreamType (7)
length	u32	length of next string (7)
projectTag	string	as in the file name (7)
length	u32	length of next string (7)
LuminosityBlockNumber	string	as in the file name (7)
length	u32	*optional* - length of next string (9)
Compression	string	*optional* - tag reporting the compression type (9)
run parameters record		
marker	u32	0x1234bbbb
record_size	u32	size of this record
run_number	u32	from IS
max_events	u32	from IS
rec_enable	u32	from IS
trigger_type	u32	from IS
detector_mask_1of2 (LSB)	u32	from IS
detector_mask_2of2 (MSB)	u32	from IS
beam_type	u32	from IS
beam_energy	u32	from IS
data separator record		
marker	u32	0x1234cccc
record_size	u32	size of this record
data_block_number	u32	number of the following block of data within this file sequence
data_block_size	u32	size of the following block of data in bytes
file end record		
marker	u32	0x1234dddd
record_size	u32	size of this record
date	u32	when the file was closed
time	u32	when the file was closed
events_in_file	u32	in this file
data_in_file	u32	in this file, in MB
events_in_run	u32	in this run
data_in_run	u32	in this run, in MB
status	u32	not zero if this file is the last in a sequence (8)
end_marker	u32	0x1234eeee

1. Every record starts with a marker word (4 bytes). The word following the marker gives the total size of the record, including the marker, in units of 1 word.

2. Date and time are encoded in two decimal integers as DDMMYYYY and HHMMSS respectively. This is the system time of the PC writing the data obtained with a `time` system call.

3. The DataWriter interface for writing files enables to set limits on the file size in two units: i) number of "data blocks" (typically "events"), and ii) the total volume of data. A file is closed and the next one is opened when either the max. number of events is reached, or the file size limit is exceeded. In a TDAQ setup the limits are specified in the configuration database as parameters of the SFO (or of other applications which may write files to disk, e.g. the SFI).

4. Two strings which are used to make file names (see below) are recorded in the file_name_strings structure. In order to preserve the 4-byte alignment of data in files, the strings are appended with 1 to 3 spaces if necessary. The length given before each string does not count the extra spaces.

5. Since release *tdaq-common-01-06-00*, the metadata stored in free strings is always populated with at least the GUID, and since release *tdaq-common-01-10-00*, also with the streamTag, the projectTag and the Luminosity Block Number, if available. Anything extra is optional. This block of metadata can not be present in any data files written with TDAQ or DataFlow software releases compiled before March 2006, when this feature was implemented. In a TDAQ setup the optional metadata strings can be entered by a shifter using the IGUI of his/her setup. One can produce *<tag>=value* pairs with arbitrary names of tags and values. Once defined, the values are available in a pull-down menu in order to avoid typing errors. These are then published to the online Information Service (IS) and they get fetched from there in order to be put in the metadata strings of the file.

6. This information is repeated N times, where N is the number of metadata strings that are stored.

7. If the fileNameCore is not complying to the convention described in Section 3, then it's not interpretable and the streamTag, projectTag and Luminosity Block Number strings are not present. If the user wants to have such info in the metadata strings, he/she has to put them by other means into the optional "metadata strings" above, and use the DataWriter constructor which accepts free metadata strings as an argument.

8. This status word indicates if there is an additional file with a higher file number, which hosts events with the same charcteristics (see Section 3). Note that, when writing events to files, we may have to respect various types of limits (e.g., maximum number of events in a file, file size limit) and thus, events with the same characteristics may be hosted in a series of consecutively numberd files.

9. Starting from *tdaq-common-01-17-01*, the data segments can be optionally compressed. In this case the data_block_size field in the corresponding data separator record will report the compressed block size. In any case, only the data blocks are compressed, not the file records, header and trailer. For files containing compressed data, the Compression tag in the free strings indicates the utilized compression technique. For back-compatibility reasons, the Compression tag is not necessarily set in non-compressed files.

Compressed files, on the other hand, must have a Compression tag. Currently, supported values for the Compression tag are: *none, reserved, unknown, zlib [2].*

Names of the records and names of their variables are given in Table 2 above for reference, but the users will normally not be exposed to them. They are internal to the library. The library offers an API for extracting the information, as described in Section 4.

2.3 The unique ID of the raw data files

As of January 2007 (release *tdaq-common-01-06-00* and later) the data files contain a Global Unique Identifier (GUID). The identifier will enable to track provenience of data in the offline processing. It can be used for navigation from data files in AOD and ESD format back to the raw data.

The GUID is the same as the one used by POOL. It is a string representation of the system call `uuid_generate_time`, which uses the current time and the local ethernet MAC address (if available) in order to produce a universally unique identifier.

Internally the GUID is stored as the first string in the metadata strings. The API for reading of files provides a way to get the GUID directly (see Section 4).

3 File naming convention

The DAQ applications which write the data files are supposed to use the convention of file names agreed with the offline software representatives in January 2007 (for releases up to *tdaq-common-01-09-03*) and November 2008 (for releases from *tdaq-common-01-10-00* onwards). In this later convention, which unified the online and Tier0 naming conventions, the file names have the following form:

<projectTag>.<runNumber>.<streamType_streamName>.daq.RAW._lb<Luminosity Block Number>._<AppName>._<file_number>.data

for example:

data08_cos.00090943.physics_RPCwBeam.daq.RAW._lb0007._SFO-2._0021.data

Comments:

- The <streamType>, <streamName> and the <Luminosity Block Number> have been added in release *tdaq-common-01-06-00*. The combination <streamType_streamName> is refered to as a <streamTag>. The projectTag was added in release *tdaq-common-01-10-00*. The other fields were already present in earlier releases, even though the exact spelling was different (e.g. "LB" instead of "_lb").
- The <projectTag> is a string defined by the shifter from a pull-down menu in the IGUI of the DAQ system.
- The <runNumber> and the <Luminosity Block Number> are defined by the online instrastructure and are also found in the header of every event [1].

- The <streamType> and the <streamName> are strings defined in the Trigger Configuration data base. They are also present in the event data (in the StreamTag object that can be obtained from the "full event header", see documentation of the eformat library for more information [1]). The expected values of <streamType> are defined in the eformat library (e.g., "physics", "calibration", "debug", "unknown"). The EventStorage libraries do not enforce any policy on the <streamType> or the <streamName>.

- The file_number is indicating the sequential file numbering (starting from 1) within a file sequence with the same characteristis. Such files have the same fileNameCore. E.g., in the filename example above, this is the 21^{st} file containing events with the same characteristics/fileNameCore.

- fileNameCore is the first part of the file name, leaving out the "_.<file_number>" and the extension (i.e., excluding "_.0021.data" in the example above).

- All the numbers are padded with "0" to a fixed length for the needs of alphabetical sorting. We estimate that four digits are enough for the LumiBlock number and for the file number. The run number was padded to seven digits till *tdaq-common-01-09-03* (even if the maximum range is $O(2^{31})$ and would therefore require 10 digits); since release *tdaq-common-01-10-00* it is padded to eight digits for compatibility with the dataset names [3]. If any of the numbers exceeds the allowed range, the number of digits gets increased automatically, so there is no hard limit here.

On user request, the EventStorage libraries do not enforce the file naming convention, neither at writing time nor when reading. However, the *RawFileName* class is provided in the EventStorage package, so that the TDAQ applications can produce the agreed file names easily (this replaced the function *AgreedFileName* which was used till *tdaq-common-01-09-03*). It is expected that the applications will obey to the file naming convention. The authors can ensure this only in case of the SFO, SFI, ROS and test applications in the EventStorage package.

In case some of the parameters are not set, e.g. in a test setup or in commissioning runs, the DAQ applications are encouraged to have the following default values:

- the default <streamType> can be taken by the `eformat::UNKNOWN_TAG`. Since there is no restriction on the <streamName>, a good default is "None";

- the default of the luminosity block number is zero (0), which leads to "._lb0000" in the data file names. In such a case, the file contains events irrespective of which luminosity block they belong to. **A real luminosity block number in the TDAQ system is expected to be always greater than zero.** Files with "._lbN" in their filename, with N not "0000", contain events from luminosty block "N" only.

Since *tdaq-common-01-12-00*, python binding of the *RawFileName* class is implemented via the eformat python bindings. After you setup the release, just type:
% *tdaq_python*
>>> *from eformat.stream import EventStorage*
>>> *type(EventStorage.RawFileName)*

4 The EventStorage package

In this section we present the libraries used for writing and for reading of event data into/from files. An overview of the functionality is given, and the classes with their relations

are briefly discussed. A detailed description of the APIs is available via Doxygen pages available from the software releases page [4] or via comments in the code [5, 6, 7].

The API for writing of the data files is *EventStorage/DataWriter.h*. The implementation is a single class, which is responsible for writing a sequence of a data files. A maximum size of a data file can be specified in Mega Bytes or in number of events. Files larger than 2 GB are possible on 64 bit systems (tested on Scientific Linux 4). On certain file systems the file size is limited to be less than 2 GB. A *DataWriter* object can be constructed by means of either of the two available constructors: one accepts an initial set of meta-data strings, while the other not.

The helper class *EventStorage/RawFileName.h* can be used to produce file names according to the convention defined in Section 3. An example of use of the API for data writing is in a test program of the EventStorage package named *test/testWritingSpeed.cxx*. This test program is compiled and installed as `testWritingSpeed` in the tdaq-common releases. In addition to being an example of code, it can be used to measure disk I/O performance.

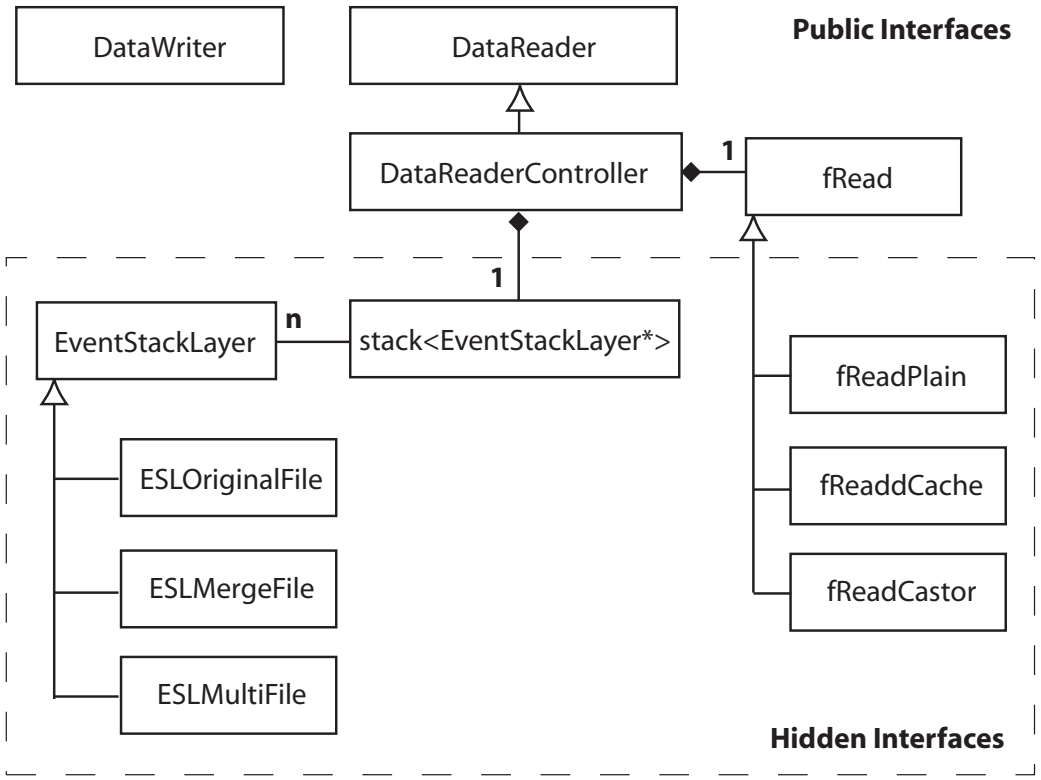

Figure 2: Class diagram of the EventStorage.

The API for reading is defined in *EventStorage/DataReader.h*. As of release *tdaq-common-01-06-00*, the API allows random navigation to events by specifying offsets. The offsets of different events can be obtained while reading a data file for the first time, e.g. during the first pass reconstruction. The data files do not contain lookup tables.

The implementation involves several classes as shown in Figure 2. The interface named *fRead* (implemented by the concrete classes *fReadPlain*, *fReadCastor*, and *fReaddCache*) is needed to encapsulate basic file reading operations for different file storage systems:

- *fReadPlain* - uses a standard C++ library, which can read disk files,

- *fReadCastor* - uses the *shift* library, which can be used to read files directly from the Castor system at CERN, or from another Castor system,
- *fReaddCache* - uses the *dcap* library, which can be used to read files directly from a dCache system.

While Castor is used at CERN, dCache is used at many GRID sites which store ATLAS data. It is therefore necessary to support different file I/O libraries in addition to the different file formats ($2 \rightarrow 5$). A simple use of inheritance would lead to a multiplication of the concrete classes to support all the combinations of file format and file I/O library. In order to avoid this problem, the design shown in Figure 2 was used. This is an example of the "bridge" design pattern as described in [8].

The classes that inherit from *fRead* are compiled in separate libraries, which are dynamically loaded. This is useful because *shift* and *dcap* libraries are not installed on all the systems where the software needs to run. Until the *tdaq-common-01-04-00* release the libraries *libfReadCastor.so* and *libfReaddCache.so* were part of the tdaq-common releases. As this was introducing unwanted dependencies, the code of the *fReadCastor* and *fReaddCache* plugins was moved to the offline repository [7]. These libraries make part of offline releases starting with release 13.

With the introduction of the merged raw files in release *tdaq-common-01-10-00*, the design has been considerably changed. The API header file *DataReader.h* has been preserved in order to avoid the need to change the existing code (both online and offline). The design involves a stack of objects implementing the *EventStackLayer* interface. A *ESLMultiFile* object serves as a starting point and handles file-sequences, if used. It is able to determine if the files are individual (original) or merged raw files (or neither). According to this information, a *ESLMergeFile-* or *ESLOriginalFile-* object is instantiated and put on the stack. A *ESLMergeFile*-object itself parses the header data and puts a *ESLOriginalFile*-object on the stack. As soon as this process has finished, the user can call the *readData(...)* method from the *DataReaderController*, which itself calls *readData(...)* from the topmost stack object. *DataReaderController* transparently detects the end of any file type (a stand-alone original, an original within a merged file, or a merged file) and proceeds to the next file unit. Objects are therefore pushed into and popped out of the stack while data is being read. Apart from the necessary interfaces, header files are hidden for encapsulation and patching reasons.

It is convenient and common to use a helper function defined in *EventStorage/pickDataReader.h*, which takes care of the dynamic loading of the plugins and tries to read a file with one of them. The helper function returns an object of type *DataReaderController*, which can be used to read data and metadata from the files. *DataReaderController* transparently handles individual (original) or merged raw files. Sequences are suported via the `enableSequenceReading` method of the DataReaderController object. Examples are shown in test programs of the EventStorage package: *test/readData.cxx, test/readMetaData.cxx, test/testReadingSpeed.cxx*. These are compliled and installed as executables in the tdaq-common releases under the names `readData`, `readMetaData` and `testReadingSpeed`, respectively.

A Appendix

A.1 Merged raw data files

At the end of the Data Acquisition chain, just after the Event Filter selection, several SFOs are writing out events into RAW data files. In doing so, each SFO is respecting various boundaries (e.g., maximum number of events, file size limit, luminosity block boundary, stream type and name) independently from the other SFOs. In this mode of operation, it then happens that the actual size of individual raw data files can vary considerably, leading to small (< 1 GB) files. Dealing with too many small data files could become a complication for the file management and distribution in the offline world, so we could be faced with the need to have our data contained in bigger, "merged", files. In this appendix we define the format of such merged files.

The least intrusive way for merging the content of several raw data files is to just concatenate them, with the simple addition of a special "merged file header" which describes briefly the file content and helps to retrieve the individual raw data files.

This type of merging can be done by a simple script, provided it respects the merged file format defined in Section A.3. A C++ class describing the header and methods to act on it is provided in the EventStorage package. An example for merging and retrieving the individual raw files is also provided.

A.2 Rules for merging raw data file

The policy on which files to merge is a data handling issue. Here we define the merged file format, assuming that the files to be merged, have the same triplet of:

i) run number (and therefore, also the same projectTag),

ii) streamTag (i.e., both streamType and steamName), and

iii) luminosity block number.

When a user is trying to concatenate individual files into a merged file, there is the possibility that the merged file becomes very big, i.e., above an acceptable size limit[1]. In such a case, several merged files, consecutively numbered, will have to hold the individual raw data files which satisfy the merging criteria. Thus, we could have a sequence of merged files hosting files with the same characteristics, in analogy to a sequence of individual raw files hosting similar events. Opening one of these merged files, the user should be able to understand if there is any more fiels in the sequence. Thus, the maximum size limit and identifiers for the previous and the next file in the sequence are recorded in the merged file itself.

A.3 Merged file format

In Table 3 below we see the detailed file structure, as it is implemented in the *MergedRawFile* class: a "merged file header" (presented in detail), followed by the individual raw files, concatenated one after the other (a merged file will contain only whole individual raw files; thus, no files will be split across various merged files):

[1]This could happen (or be prevented) depending on the policy used for file merging. E.g., for files in the express, calibration or debug streams (where the luminosity block boundaries are not respected), or even for very populous physics streams (which always respect luminosity block boundaries), it can happen that the list of files to concatenate is such that the merged file becomes over-sized.

Table 3: Format of merged RAW files

file start marker	top-header-part
version	
header size	
file size (part 1: LSB)	
file size (part 2: MSB)	
file size limit (MB)	
number of contained events	
number of events in sequence so far	
number of contained files = q	
file open date	
file open time	
run number	
luminosity block number	
total number of files in sequence	
this file sequence number	
previous file sequence number	
next file sequence number	
j GUID words	GUID
GUID word 0	
...	
GUID word $j-1$	
k filename words	merged file name
filename word 0	
...	
filename word $k-1$	
m Project Tag words	Project Tag (if available)
Project Tag word 0	
...	
Project Tag word $m-1$	
n Stream words	Stream Tag
Stream word 0	
...	
Stream word $n-1$	
e extra/user information words	extra/user info
extra/user information word 0	
...	
extra/user information word $e-1$	
p words about the q contained files	
location (offset) of contained file 0 (part 1: LSB)	info about contained file 0
location (offset) of contained file 0 (part 2: MSB)	
size of contained file 0 (part 1: LSB)	
size of contained file 0 (part 2: MSB)	
r filename words for contained file 0	
filename word 0 for contained file 0	
...	
filename word $r-1$ for contained file 0	
...	
location (offset) of contained file $q-1$ (part 1: LSB)	info about contained file $q-1$
location (offset) of contained file $q-1$ (part 2: MSB)	
size of contained file $q-1$ (part 1: LSB)	
size of contained file $q-1$ (part 2: MSB)	
t filename words for contained file $q-1$	
filename word 0 for contained file $q-1$	
...	
filename word $t-1$ for contained file $q-1$	
individual raw file 0	
...	
individual raw file $q-1$	

`uint32_t` file start marker	0x1ba2baba
`uint32_t` version	this is 1 for this iteration
`uint32_t` header size	the size of the header in 32-bit words
`uint32_t` file size	the size of the whole merged file in words, till the last word of the last attached individual raw file. Part 1 hosts the less-significant ("lower") 32-bits of the file size, while part 2 hosts the most significant ("upper") 32-bits of the file size
`uint32_t` file size limit (MB)	the maximum acceptable size of the whole merged file, in MB. This is a configuration parameter for the merging procedure. If the merged file becomes larger than this limit, a new merged file will be opened to host the rest of the individual raw data files provided
`uint32_t` number of contained events	the sum of the number of events in individual raw files contained in this merged file
`uint32_t` number of events in sequence	the number of events so far in the sequence (cummulated from the first file and including this one)
`uint32_t` number of contained files	the number of individual RAW files contained in this merged file
`uint32_t` file open date	the date as DDMMYYYY from a system call `time` on the computer writing the merged file
`uint32_t` file open time	the time as HHMMSS from the system call `time` on the computer writing the merged file
`uint32_t` run number	this is the run number of the first individual raw file contained in this merged file
`uint32_t` Luminosity Block number	the Luminosity Block of the first individual raw file contained in this merged file
`uint32_t` number of files in sequence	total number of files in sequence (since the files to merge are known up front, the calculation of how many merged files are needed is known since the begining)
`uint32_t` this file sequence number	sequence number of current file

`uint32_t` previous file sequence number	sequence number of previous file
`uint32_t` next file sequence number	sequence number of next file
`uint32_t` j GUID words	number of GUID words to follow
`uint32_t` GUID words	one word per character
`uint32_t` k filename words	number of filename words to follow; thus, if n=0, there is no filename words following
`uint32_t` filename word	one word per character
`uint32_t` m words describing the Project Tag	number of project tag words to follow; if n=0, there is no Project Tag words following; the Project Tag put here is the Project Tag of the first individual raw file contained in this merged file
`uint32_t` Project Tag word	one word per character
`uint32_t` n Stream Tag words	number of Steam Tag words to follow; the Stream Tag put here is the Stream Tag of the first individual raw file contained in this merged file
`uint32_t` Stream Tag word	one word per character
`uint32_t` e "extra information" words	number of words to follow with any extra information; for now, this is a space-holder allowing to add info in the feature, without changing the merged-file format
`uint32_t` "extra information" word	one word per character
`uint32_t` p words about the q contained files	number of relevant words to follow
`uint32_t` location of contained file	local offset (in number of words) from the start of the merged file, i.e., starting from (and including) the merged file marker. Part 1 hosts the less-significant ("lower") 32-bits of the offset, while part 2 hosts the most significant ("upper") 32-bits of the offset
`uint32_t` size of contained file	the size of the contained file in words; Part 1 hosts the less-significant ("lower") 32-bits of the file size, while part 2 hosts the most significant ("upper") 32-bits of the file size

`uint32_t` *r* filename words for contained file	number of words to follow, describing the contained file's name
`uint32_t` filename word of contained file	one word per character

A.4 File naming convention of the merged raw data files

Individual raw files produced online follow the agreed filename convention described in Section 3. Merged files can also follow the same naming convention, at least for the filenameCore. So, we can have names complying to the Tier0 and dataset naming convention [3] (`http://cdsweb.cern.ch/record-restricted/1070318/`)

<projectTag>.<runNumber>.<stream>.<productionStep>.<dataType>.
lb <lumiBlockNumber>. <applicationName>._ <partitionNr>._ <userTag>

for example a possible name could be:

data08_cos.00090943.physics_RPCwBeam.daq.RAW.
_lb0007._merger._0001._ATLAS

A.5 Creating a merged raw file

A test progam (`testMergedFile`) is instaleld in the realase. It works by passing it the following arguments: i) a file containing the list of files to merge, ii) a merged-file limit (MB), and iii) optionally, the output merged-file name-core. If the merged-file limit is 0, this means that all given files should be merged into a single merged file, without respecting any size limit.

Furthermore, a Tier-0 transformation python script is provided in the package (`share/MergeRawFileTier0Transformation.py`). This script is used by Tier-0 to perform file merging. It is called with the option `--argdict=<dictfile>`, where `dictfile` is a pickled map containing the required info. More information can be found by executing this python script without argument.

A.6 Reading and unpacking a merged raw file

As explained in Section 4, the *DataReader* understands, behind the scenes, if the file provided is an individual or a merged raw file, and then provides the caller with the usual result (e.g., the raw events), without the user noticing the difference. For the end user, the *pickDataReader(...)* helper function is still used to get such a *DataReader* object. The `readData` test executable is an example installed in the tdaq-common release, and can be used to demonstrate that merged or individual RAW files are read transparentely for the user.

The `testDeMerge` program is also installed in the release; it parses a merged file and decomposes it into its orginal components. It's used like:
% *testDeMerge <mergefile.data> [onlylist]*
If the argument *onlylist* is given, the program shows the contents of the merged file and exits.

Note also that, when a user interogates a merged file for meta-data, run parameters etc., he/she will be provided with the information found in the first contained individual raw file.

References

[1] *Web page of the "ATLAS event format library"*. https://twiki.cern.ch/twiki/bin/view/Atlas/AtlasComputing?topic=EventFormat. 3, 6, 7

[2] *zlib*. http://zlib.net/. 6

[3] *"ATLAS Dataset Nomenclature" - Internal Note*. http://cdsweb.cern.ch/record/restricted/1070318. 7, 14

[4] *Web page of the "ATLAS tdaq and tdaq-common releases"*. https://atddoc.cern.ch/cmt/releases/. 8

[5] *DAQ/DataFlow/EventStorage package in the CVS repository of TDAQ software*. http://isscvs.cern.ch/cgi-bin/viewcvs-all.cgi/?cvsroot=atlastdaq. 8

[6] *EventStorage library in tdaq-common-01-12-00 release*. installed/afs/cern.ch/atlas/project/tdaq/cmt/tdaq-common-01-12-00/. 8

[7] *offline/Event/ByteStreamStoragePlugins package in the CVS repository of offline software*. http://atlas-sw.cern.ch/cgi-bin/viewcvs-atlas.cgi/. 8, 9

[8] E.Gamma, R.Helm, R.Johnson, and J.Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley. ISBN 0201633612. 9