Classes for Reading Great-format Data

Nigel Warr

August 2014

Abstract

For debugging purposes, it is useful to have a class, which can be used via root to access the Great-format data. This makes it possible to write scripts, which can perform simple tasks on an *ad hoc* basis, as and when they are needed.

The *libGreat* library provides four C++ classes for root to this end: *GreatFile*, *GreatBuffer*, *GreatWord* and *GreatHit*.

A few examples are also provided.

Contents

1 Introduction **4**

2 The Great format **5**
2.1 The block header 5
2.2 The data words 6
2.2.1 ADC words 6
2.2.2 Trace header words 6
2.2.3 Trace sample words 7
2.2.4 Information words 7
2.3 Determining the block size 8
2.4 Determining the byte order 8

3 The GreatFile class **10**
3.1 The Constructor 10
3.2 The Destructor 10
3.3 GreatFile::Open 10
3.4 GreatFile::Close 10
3.5 GreatFile::GetFileSize 11
3.6 GreatFile::GetBlockSize 11
3.7 GreatFile::GetNBlocks 11
3.8 GreatFile::GetBlock 11
3.9 GreatFile::Show 11
3.10 GreatFile::DetermineBlockSize 11

4 The GreatBuffer class **12**
4.1 The Constructor 12
4.2 GreatBuffer::Set 12
4.3 GreatBuffer::GetNWords 12
4.4 GreatBuffer::GetWord 12
4.5 GreatBuffer::Show 12
4.6 GreatBuffer::Swap32 13
4.7 GreatBuffer::SwapWords 13
4.8 GreatBuffer::Swap64 13

5 The GreatWord class **14**
5.1 The Constructor 15
5.2 GreatWord::Set 15
5.3 GreatWord::GetWord 15
5.4 GreatWord::GetKey 15
5.5 GreatWord::IsSample 15
5.6 GreatWord::IsTraceHeader 15
5.7 GreatWord::IsInfo 15
5.8 GreatWord::IsADC 15
5.9 GreatWord::GetLowTimestamp 16

5.10 GreatWord::HasExtendedTimestamp . . . 16
5.11 GreatWord::SetExtendedTimestamp . . . 16
5.12 GreatWord::GetFullTimestamp . . . 16
5.13 GreatWord::GetInfoField . . . 16
5.14 GreatWord::GetInfoCode . . . 16
5.15 GreatWord::GetInfoModule . . . 16
5.16 GreatWord::GetADCConversion . . . 17
5.17 GreatWord::GetADCID . . . 17
5.18 GreatWord::GetADCChannel . . . 17
5.19 GreatWord::GetADCBE . . . 17
5.20 GreatWord::GetADCModule . . . 17
5.21 GreatWord::GetADCVeto . . . 17
5.22 GreatWord::GetADCFail . . . 17
5.23 GreatWord::GetTraceNSamples . . . 17
5.24 GreatWord::GetTraceID . . . 17
5.25 GreatWord::GetTraceChannel . . . 18
5.26 GreatWord::GetTraceBE . . . 18
5.27 GreatWord::GetTraceModule . . . 18
5.28 GreatWord::GetTraceRawFlag . . . 18
5.29 GreatWord::Show . . . 18

6 The GreatHit class **19**
6.1 The Constructor . . . 19
6.2 GreatHit::Set . . . 19
6.3 GreatHit::AddSample . . . 19
6.4 GreatHit::GetID . . . 19
6.5 GreatHit::GetTimestamp . . . 19
6.6 GreatHit::GetConversion . . . 19
6.7 GreatHit::GetFail . . . 19
6.8 GreatHit::GetVeto . . . 19
6.9 GreatHit::GetNSamples . . . 19
6.10 GreatHit::GetSample . . . 20
6.11 GreatHit::GetTrace . . . 20
6.12 GreatHit::operator< . . . 20
6.13 GreatHit::Show . . . 20

7 Examples **21**
7.1 show.C . . . 21
7.2 stats.C . . . 21
7.3 proj.C . . . 22
7.4 baselines.C . . . 23
7.5 show_supercycle.C . . . 23
7.6 check_SYNC.C . . . 23
7.7 check_SYNC2.C . . . 23
7.8 timestamping.C . . . 24
7.9 build_tree.C . . . 24

1 Introduction

The first data acquired for the ISOLDE Decay Station (IDS) were obtained without a merger program. Consequently, they are not time ordered. The data in each buffer is time ordered, but each buffer only has data from a single module.

There is no synchronisation of buffers. This is different from the Xia modules, where each module starts a new buffer at the same time and when one is full, they all finish their buffer and start a new one together. This makes sorting **much** simpler, because it is just a matter of matching buffers with the same buffer timestamp and then sorting the events in them. The data acquired for IDS have no such syncronisation. This means that if, say, module 1 has a high count rate (16 clover channels) and module 2 has a low count rate (proton supercycle, proton impact, beam gate etc.) with module 3 having something in between (TACs and three $LaBr_3$(Ce) detectors), you get the situation that buffers fill up very fast for module 1, so the first event in module 2 might be contemporary with event in the first buffer from module 1, but the last event from module 2's buffer might be contemporary with the 14^{th} buffer from module 1. There is no way *a priori* to determine how much data to look through to be sure of getting everything, unless we start looking at individual timestamps.

There is no coincidence requirement on the hardware, so we have inordinate quantities of useless singles data (we are, after all, looking for at least triple coincidences between two detectors and a TAC for fast timing). Consequently, the data acquisition system generates 2 gigabyte files in 10 to 15 minutes and most of the events are of no interest whatsoever. This means that any sorting method has to be efficient. However, the poor design of the data format means that this is not possible[1].

[1] In fairness, the design was probably fine for its original purpose many many years ago, but it hasn't scaled well to the technological changes.

2 The Great format

The files consist of blocks of fixed length (though that length can vary from file to file), each of which starts with a 24-byte header.

2.1 The block header

The data are in blocks, each of which has a 24-byte header, which is explained at *http://ns.ph.liv.ac.uk/MTsort-manual/TSformat.html*. It is a structure given the name *DATA_HEADER* which is defined in *GreatHeader.hh*.

Type	Name	Meaning
Char_t	id[8]	8-byte string "EBYEDATA"
UInt_t	sequence	Number of the buffer within the file
UShort_t	stream	Data acquisition stream number (1...4)
UShort_t	tape	Always zero
UShort_t	MyEndian	Written as a native 1 by the tape server
UShort_t	DataEndian	Written as a native 1 by the hardware
UInt_t	dataLen	Number of bytes of data following the header

The types refer to *root* types. A *Char_t* is an 8-bit signed character, a *UShort_t* is a 16-bit unsigned integer and a *UInt_t* is a 32-bit unsigned integer. The sizes are guaranteed by *root* irrespective of the system.

It should be noted that the *sequence* isn't that useful, because we know this from the position in the file. Perhaps this was needed for tapes.

The *stream* is always 1 for IDS. Perhaps this has something to do with systems with multiple crates?

The *tape* is always zero, though the documentation says one. Probably, it is one if the data were really written to tape and zero for disk and IDS writes to disk.

The *MyEndian* is used to determine the endianness of the header (which is different to that of the data!). The machine writing the header writes one here. So if you get something else back, you have to swap the bytes as 32-bit words from big- to little-endian (or vice versa). Note that the term "tape server" refers to the program which writes the data to disk. The only tapes at IDS are the ones the beam is implanted in, but the data acquisition system goes back to days when people used to store data on tape.

The *DataEndian* is used to determine the endianness of the data. The hardware writes one here, so if you get something else, you have to swap the 64-bit words. Unfortunately, there are several ways to do this and the choice of one is rather poor, because it doesn't distinguish between them.

Nothing indicates the size of the block, which is larger than the 24-byte header plus dataLen. The rest of the block after the data is padded with the character 0x5E, which is some obscure ANSI tape padding byte from IBM Format-D, which goes back quarter of a century.

2.2 The data words

There are four types of data possible, which are indicated by the two most significant bits of the 64-bit word, which form a key:

Key	Meaning
0	Trace samples
1	Trace header
2	Information
3	ADC data

Each of these has its own structure.

2.2.1 ADC words

Bits	Meaning
0-27	28 least-significant bits of timestamp
28-31	Always zero
32-47	ADC data
48-51	Channel number (counting from zero)
52	Baseline (=1) or energy (=0)
53-59	Module number
60	Veto bit
61	Fail bit
62-63	Key = 3

Bits 48 to 59 may be viewed together as a channel ID.

2.2.2 Trace header words

Bits	Meaning
0-27	28 least-significant bits of timestamp
28-31	Always zero
32-47	Number of samples in trace
48-51	Channel number (counting from zero)
52	Baseline (=1) or energy (=0)
53-59	Module number
60-61	Always zero
62-63	Key = 1

Bits 48 to 59 may be viewed together as a channel ID.

2.2.3 Trace sample words

Bits	Meaning
0-13	Sample N+3
14-15	Always zero
16-29	Sample N+2
30-31	Always zero
32-45	Sample N+3
46-47	Always zero
48-61	Sample N
62-63	Key = 0

Note, that the manual explicitly warns, that the bits which are always zero and the key, might actually be non-zero under some circumstances, so it should be assumed that a certain number of samples follows a trace header and that trace header indicates how many.

2.2.4 Information words

Bits	Meaning
0-27	28 least-significant bits of timestamp
28-31	Always zero
32-51	Field
52-55	Code
56-61	Module number
62-63	Key = 2

The meaning of the field depends on the code.

For codes 2, 3, 4 and 7, they are the 20 most-significant bits of the 48-bit timestamp. So such words are important as they are the only ones which provide the **entire** 48-bit timestamp in one word. Unfortunately, there is no guarantee, that the file starts with such a word, so there may be ADC conversions and/or traces before the first full timestamp. Code 4 (SYNC100) is particularly important, because it should come every 65536 ticks = every 655.36 μs. i.e. at 1525.9 Hz, on all modules simulaneously.

Codes 1, 9, 10, 11 and 12 have the channel number (counting from zero) in the field. These are all error messages which should apply to the previous conversion. That conversion should also have its fail bit set.

Code	Meaning	Field
0	Unidentified data	
1	Pile-up	Channel number (counting from zero)
2	Pause timestamp	Timestamp bits 28-47
3	Resume timestamp	Timestamp bits 28-47
4	SYNC100 timestamp	Timestamp bits 28-47
7	Extended timestamp	Timestamp bits 28-47
9	Over-range	Channel number (counting from zero)
10	Under-range	Channel number (counting from zero)
11	Overflow	Channel number (counting from zero)
12	Underflow	Channel number (counting from zero)

Codes 5 and 6 are not documented. Code 8 has two meanings, neither of which are relevant for IDS. Codes 13-15 are also not relevant for IDS.

2.3 Determining the block size

The determination of the block size needs some explanation. It should be understood that there is nothing in the header to indicate the block size. This is rather poor design, but the reason is that it goes back to the times when people wrote their data to magnetic tape and the blocks in question corresponded to the blocks on the magnetic tape. This was something which was set when the tape was formatted. Consequently, you needed to already know the block size in order to be able to read anything. So merely being able to access the header, implies that you already know the block size. This is the reason Vic Pucknell gave for not including the block size in the header. So we have to apply some heuristics to determine the block size.

The way we do it is to use the fact that the first eight bytes in every header are the string "EBYEDATA". So the procedure is as follows:

- Check that the file begins with "EBYEDATA". If it doesn't, the file is not in the Great format, so we throw an exception.
- Guess a block size which is some power of two (we start with 2^8) and look that far into the file. If we find the "EBYEDATA" string, this indicates that we guessed correctly. In fact the initial guess is chosen to be always too small, so the first attempt will always fail.
- Then we double the block size and try again and keep doubling until we find the correct value, or reach the end of the file, in which case we throw an exception.

2.4 Determining the byte order

Unfortunately, the header also doesn't indicate the byte order unambiguously. There are, in fact, two different byte orderings to consider:

- The byte order for the header, which is determined by the tape server.
- The byte order for the data, which is determined partly by the hardware and partly by the merger software.

The byte order of the header is simple. The tape server writes 1 into the 16-bit *MyEndian* field, so if we read 0x0001, the computer reading has the same endianness as the tape server and does not need to swap. If, instead, we read 0x0100, the endianness is different and we need to swap bytes of the header words to change the endianness. This operation is symmetric (i.e. big $\$rightarrow$ little is the same procedure as the other way round). The only fields which need to be swapped are the *sequence* and *dataLen* ones, both of which are 32 bit. The *GreatBuffer::Swap32* method does this.

The byte order for the data is not simple. Again there is the 16-bit *DataEndian* word, into which the hardware writes 1. Again, if we read back 0x0001, it means that the computer has the same endianness as the hardware, and if we get 0x0100, the endianness differs. The data words are 64-bit words, and the *GreatBuffer::Swap64* method is used to change the endianness of such words. Again, the operatation is symmetric, so applying this method will change big to little endian or vice versa.

However, that does not exhaust the degrees of freedom. The hardware works in pairs of 32-bit words and we can either have the most-significant 32-bits first then the least-significant ones, or the other way round, regardless of the endianness. Both are possible and both can occur. Moreover, there is no way to determine which case we have from the header.

The only thing we have to go on is the fact that for ADC words, trace headers and information words, bits 62 and 63 are never both zero, while bits 30 and 31 are always zero. For samples, notwithstanding the caveat given in the manual, that we shouldn't rely on bits 30, 31, 62 and 63 being zero, they normally are. So the method used is to check through a buffer and if bits 30 or 31 are set anywhere, the 32-bit halves of the 64-bit word must be swapped. The *GreatBuffer:SwapWords* method is used to swap them back.

3 The GreatFile class

The *GreatFile* class is used to open a Great-format file, map it into virtual memory and determine the block size. It is defined entirely in *GreatFile.hh*.

3.1 The Constructor

The constructor can be called with or without a filename. If called without, it just sets various things to zero. If called with, it does that and then calls the *GreatFile::Open* method.

3.2 The Destructor

The destructor just calls the *GreatFile::Close* method.

3.3 GreatFile::Open

The *GreatFile::Open* method does five things:

- Calls the *GreatFile::Close* method to make sure any previous file is closed.
- Opens the file with *fopen*
- Determines the number of bytes in the file by using *fseek* to the end and then *ftell*.
- Maps the file into virtual memory with *mmap*
- Then it calls *GreatFile::DetermineBlockSize* to figure out the block size using the heuristics explained in section 2.3.

It throws exceptions if it can't open or map the file and GreatFile::DetermineBlockSize throws exceptions if the file is bad.

3.4 GreatFile::Close

The GreatFile::Close method closes a file if one was open. If no file was open, it does nothing.

It does the following:

- If something was mapped, it unmaps it with *munmap*.
- It zeroes various parameters.
- If a file was open it closes it with *fclose*.

3.5 GreatFile::GetFileSize

The *GreatFile::GetFileSize* method returns the size of the file in bytes. This was determined by *GreatFile::Open*. If no file is open it returns zero.

3.6 GreatFile::GetBlockSize

The *GreatFile::GetBlockSize* method returns the size of a block in bytes. This was determined by *GreatFile::DetermineBlockSize* which was called from *GreatFile::Open*.

3.7 GreatFile::GetNBlocks

The *GreatFile::GetNBlocks* method determines the number of blocks. This is just the file size divided by the block size.

3.8 GreatFile::GetBlock

The *GreatFile::GetBlock* method returns a pointer to the memory-mapped n^{th} block of the file or NULL, if the block doesn't exist, or no file was open. If called without a parameter, it returns the first block.

3.9 GreatFile::Show

The *GreatFile::Show* method is for debugging purposes only. It writes some information in human-readable form about the state of the class.

3.10 GreatFile::DetermineBlockSize

The *GreatFile::DetermineBlockSize* method is private and may not be called from outside the class. It is called by *GreatFile::Open* and is used to determine the block size. It throws exceptions if the first eight bytes of the file are not the string "EBYEDATA" or if it can't find the second occurrence of this string (i.e. the start of the second block).

It only tries powers of two starting from 2^8. The starting value is chosen to be definitely too small in all cases. If we started with a value which is too large, we could find the n^{th} block rather than the first or fail.

4 The GreatBuffer class

The *GreatBuffer* class handles the manipulation of the blocks of data and access to the words in the data buffer. It takes care of understanding the block header (see section 2.1). It is defined entirely in *GreatBuffer.hh*.

4.1 The Constructor

The constructor can be called with or without a pointer to a block of data. It merely calls the *GreatBuffer::Set* method, passing the pointer (which may be *NULL*).

4.2 GreatBuffer::Set

The GreatBuffer::Set method is used to set up a buffer. It is passed a pointer. First of all it zeroes a few items and if the pointer is *NULL* it returns. Otherwise, it sets up pointers of the appropriate types to the header and data.

Then it checks that the header is valid (it must start with the 8-byte string "EBYEDATA") and throws an exception if it isn't.

Finally, it determines the number of 64-bit words of data in the buffer. To do this it looks at *dataLen* in the header (which is in bytes), but if *MyEndian* from the header is not one, it has to swap the 32-bit *dataLen* from the endianness of the acquisition computer to that of the computer running the code. It divides the length in bytes by sizeof(ULong_t) to get a number of 64-bit words.

4.3 GreatBuffer::GetNWords

The *GreatBuffer::GetNWords* method returns the number of 64-bit words in the buffer if one was set, or zero otherwise. This is the value which was determined by *GreatBuffer::Set* using the *dataLen* from the header.

4.4 GreatBuffer::GetWord

The *GreatBuffer::GetWord* method is used to get the n^{th} word from the buffer. If the value of n is too large it returns zero.

The method automatically performs the necessary byte-swapping.

4.5 GreatBuffer::Show

The *GreatBuffer::Show* method is for debugging purposes only. It writes some information in human-readable form about the state of the class.

4.6 GreatBuffer::Swap32

The *GreatBuffer::Swap32* method is private and may not be used outside of this class. It swaps 32-bit integers between big- and little-endian. i.e. 0x1234 → 0x4321. It is used for swapping the data length in the header.

4.7 GreatBuffer::SwapWords

The *GreatBuffer::SwapWords* method is private and may not be used outside of this class. It swaps two 32-bit words within a 64-bit one. i.e. 0x12345678 → 0x45670123. It is used to swap the data. This is needed if the hardware has the same endianness as the computer running this code and is to do with the fact that the hardware represents the 64-bit integer as two 32-bit ones.

Question: does this implicitly imply that the code is running on a little-endian machine? I have no way of testing.

4.8 GreatBuffer::Swap64

The *GreatBuffer::Swap64* method is private and may not be used outside of this class. It swaps 0x12345678 → 0x43218765.

Question: is this correct? I have no way of testing.

5 The GreatWord class

The *GreatWord* class is used to interpret the data from a single word of data in the Great format. It is defined entirely in *GreatWord.hh*. The class should hide all this format from the user. Note, however, that when parsing a buffer of data, it is important to call the constructor only once and then use the *GreatWord::Set* method for each word in the buffer, in the order they appear in the buffer. In this case, the class will correctly handle the extended part of the timestamp.

The format is documented at *http://npg.dl.ac.uk/documents/edoc504/edoc504.html*. The data are in 64-bit words. The two most significant bits serve as a key to indicate what type of data the word contains:

Key	Meaning
0	A sample from a trace
1	A trace header
2	An information word
3	An ADC conversion

The ADC conversions contain two bits for *fail* and *veto*, 12 bits for the channel ID, 16 bits for the ADC conversion and the 28 least-significant bits of the timestamp (10 ns ticks).

The trace header words contain 12 bits for the channel ID, 16 bits for the sample length, and the 28 least-significant bits of the timestamp (10 ns ticks). Note that the sample length indicates the number of samples, not the number of sample words. There are four samples per word.

It should be noted that the specification specifically discourages reliance on the key bits for samples as stray data could end up in the two most significant bits. The trace header indicates how many samples will follow (note that 4 samples are packed into a single word).

The information words contain a 6 bits module number, a 4 bit information code, a 20-bit information field and the 28 least-significant bits of the timestamp (10 ns ticks). The contents of the field depends on the code. For codes 2, 3, 4 and 7, it is the 20 most-significant bits of the 48-bit timestamp (10 ns ticks). The code 4 (SYNC100) should come every 65536 ticks = 655.360 μs or at 1525.88 Hz. Extended timestamp words (code 7) are also common. Sometimes there are pile-up (code 1), ADC over range (code 8) or under range (code 9) or ADC overflow (code 11) or underflow (code 12). The Pause (code 2) and resume (code 3) should not occur if the system is working properly.

The channel ID is made up from the module number multiplied by 32 plus the channel number minus 1. where the module numbers are 1, 2, 3 and the

channels are 1... 16. This means there is an extra free bit which is indicated as 0 = energy, 1 = baseline, but it never seems to be set.

5.1 The Constructor

The constructor can be called either with or without a data word. It sets the extended timestamp to zero and then calls the *GreatWord::Set* method, passing the word.

5.2 GreatWord::Set

The *GreatWord::Set* method is called either with or without a word of data. It checks if the word is an information word containing an extended timestamp (i.e. codes 2, 3, 4, 7) by calling *GreatWord::HasExtendedTimestamp* and if it does, it gets the extended timestamp from the info field using *GreatWord::GetInfoField*.

5.3 GreatWord::GetWord

The *GreatWord::GetWord* method returns the word set with *GreatWord::Set*.

5.4 GreatWord::GetKey

The *GreatWord::GetKey* method returns the key, which is the two most significant bits of the word.

5.5 GreatWord::IsSample

The *GreatWord::IsSample* method returns true if the key is zero (trace sample) and false otherwise.

5.6 GreatWord::IsTraceHeader

The *GreatWord::IsTraceHeader* method returns true if the key is one (trace header) and false otherwise.

5.7 GreatWord::IsInfo

The *GreatWord::IsInfo* method returns true if the key is two (information word) and false otherwise.

5.8 GreatWord::IsADC

The *GreatWord::IsADC* method returns true if the key is three (ADC sord) and false otherwise.

5.9 GreatWord::GetLowTimestamp

The *GreatWord::GetLowTimestamp* method returns the 28 least-significant bits of the timestamp, except for trace samples, where it returns zero.

5.10 GreatWord::HasExtendedTimestamp

The *GreatWord::HasExtendedTimestamp* returns true if the word is an information word and has code 2, 3, 4 or 7, or false otherwise.

5.11 GreatWord::SetExtendedTimestamp

The *GreatWord::SetExtendedTimestamp* method sets the 20 most-significant bits of the 48-bit timestamp. Normally, the extended part of the timestamp is set automatically, whenever *GreatWord::Set* is called with a word containing this information. As long as the whole buffer is parsed sequentially with a single instance of *GreatWord* it should not be necessary to call this method. However, there may be occasions, when the user needs to set the extended part of the timestamp explicitly. In this case, this is the method to do this.

5.12 GreatWord::GetFullTimestamp

The *GreatWord::GetFullTimestamp* method returns the full 48-bit timestamp, except for trace sample words, where it returns zero. It reconstructs this full timestamp using the last extended timestamp (20 most-significant bits) it obtained from an information word and the 28 least-significant bits from the current word.

5.13 GreatWord::GetInfoField

The *GreatWord::GetInfoField* method returns the field part of an information word, or zero if it is not an information word. This corresponds to bits 32 to 51.

5.14 GreatWord::GetInfoCode

The *GreatWord::GetInfoCode* method returns the code part of an information word, or zero if it is not an information word. This corresponds to bits 52 to 55.

5.15 GreatWord::GetInfoModule

The *GreatWord::GetInfoModule* method returns the module number part of an information word, or zero if it is not an information word. This corresponds to bits 56 to 61.

5.16 GreatWord::GetADCConversion

The *GreatWord::GetADCConversion* method returns the ADC conversion part of an ADC word, or zero if it is not an ADC word. This corresponds to bits 32 to 47.

5.17 GreatWord::GetADCID

The *GreatWord::GetADCID* method returns the channel ID part of an ADC word, or zero if it is not an ADC word. This corresponds to bits 48 to 59.

5.18 GreatWord::GetADCChannel

The *GreatWord::GetADCChannel* method returns the channel number part of an ADC word, or zero if it is not an ADC word. This corresponds to bits 48 to 51. We add one to make sure it is in the range $1 \ldots 16$.

5.19 GreatWord::GetADCBE

The *GreatWord::GetADCBE* method returns the energy/baseline flag, which is encoded in the ID. This corresponds to bit 52.

5.20 GreatWord::GetADCModule

The *GreatWord::GetADCModule* method returns the module number part of an ADC word, or zero if it is not an ADC word. This corresponds to bits 53 to 58.

5.21 GreatWord::GetADCVeto

The *GreatWord::GetADCVeto* method returns the veto part of an ADC word, or zero if it is not an ADC word. This corresponds to bit 60.

5.22 GreatWord::GetADCFail

The *GreatWord::GetADCFail* method returns the fail part of an ADC word, or zero if it is not an ADC word. This corresponds to bit 61.

5.23 GreatWord::GetTraceNSamples

The *GreatWord::GetTraceNSamples* method returns the number of samples part of a trace header word, or zero if it is not a trace header word. This corresponds to bits 32 to 47.

5.24 GreatWord::GetTraceID

The *GreatWord::GetTraceID* method returns the channel ID part of a trace header word, or zero if it is not a trace header word. This corresponds to bits 48 to 59.

5.25 GreatWord::GetTraceChannel

The *GreatWord::GetTraceChannel* method returns the channel number part of a trace header word, or zero if it is not a trace header word. This corresponds to bits 48 to 51. We add one to get it into the range 1...16.

5.26 GreatWord::GetTraceBE

The *GreatWord::GetTraceBE* method returns the energy/baseline flag, which is encoded in the ID. This corresponds to bit 52.

5.27 GreatWord::GetTraceModule

The *GreatWord::GetTraceModule* method returns the module number part of a trace header word, or zero if it is not a trace header word. This corresponds to bits 53 to 58.

5.28 GreatWord::GetTraceRawFlag

The *GreatWord::GetTraceRawFlag* method returns the raw flag part of a trace header word, or zero if it is not a trace header word. This corresponds to bit 52.

5.29 GreatWord::Show

The *GreatWord::Show* method is for debugging purposes only. It writes some information in human-readable form about the state of the class.

6 The GreatHit class

The *GreatHit* class is used to represent the data from a single hit (including traces), which may derive from more than one *GreatWord*. It is entirely defined in *GreatHit.hh*.

6.1 The Constructor

The constructor takes the ID, timestamp, ADC conversion, fail and veto flags as optional parameters and simply calls *GreatHit::Set* with those parameters.

6.2 GreatHit::Set

The *GreatHit::Set* method stores the parameters passed to it in the class and clears any existing trace.

6.3 GreatHit::AddSample

The *GreatHit::AddSample* method adds a sample to the trace.

6.4 GreatHit::GetID

The *GreatHit::GetID* method returns the channel ID.

6.5 GreatHit::GetTimestamp

The *GreatHit::GetTimestamp* method returns the timestamp.

6.6 GreatHit::GetConversion

The *GreatHit::GetConversion* method returns the ADC conversion.

6.7 GreatHit::GetFail

The *GreatHit::GetFail* method returns the fail bit.

6.8 GreatHit::GetVeto

The *GreatHit::GetVeto* method returns the fail bit.

6.9 GreatHit::GetNSamples

The *GreatHit::GetNSamples* method returns the number of samples in the trace.

6.10 GreatHit::GetSample

The *GreatHit::GetSamples* method returns the n^{th} sample from the trace, or zero if there are less than n samples.

6.11 GreatHit::GetTrace

The *GreatHit::GetTrace* method returns a pointer to the trace.

6.12 GreatHit::operator<

The *GreatHit::operator<* is a comparison operator used for sorting. Classes having this operator can be sorted using *std::sort*. This method sorts by timestamp.

6.13 GreatHit::Show

The *GreatHit::Show* method is for debugging purposes only. It writes some information in human-readable form about the state of the class.

7 Examples

The examples (in the examples directory) were primarily written to debug certain features of the DAQ. To use them, you need to have loaded the *libGreat.so* library and set the include path to use the normal root include path and also the directory with the headers. e.g.

```
gSystem->SetIncludePath("-I/path/to/headers");
gSystem->Load("/path/to/library/libGreat.so");
```

You might want to put that in a rootlogon.C file. In the examples directory, there's a rootlogon.C which assumes the libGreat headers and library are in the parent directory.

The code needs to be compiled to work, so start root then:

```
.L example.C++
example(whatever parameters);
```

7.1 show.C

The *show.C* example dumps the entire contents of the file in human readable form. It takes a single parameter, which is the name of the data file.

```
Word: 0x81405CC10069001A Timestamp: 0x05CC1069001A Type: INFO          Module: 1 Code: 4 SYNC100  Field: 0x05CC1
Word: 0x82405CC10069001A Timestamp: 0x05CC1069001A Type: INFO          Module: 2 Code: 4 SYNC100  Field: 0x05CC1
Word: 0x83405CC10069001A Timestamp: 0x05CC1069001A Type: INFO          Module: 3 Code: 4 SYNC100  Field: 0x05CC1
Word: 0xC021094F00691D55 Timestamp: 0x05CC10691D55 Type: ADC           Module: 1 Channel: 2  E Conversion: 0x094F Veto: 0 Fail: 0
Word: 0xE02004BF00691D56 Timestamp: 0x05CC10691D56 Type: ADC           Module: 1 Channel: 1  E Conversion: 0x04BF Veto: 0 Fail: 1
Word: 0x8110000000691D56 Timestamp: 0x05CC10691D56 Type: INFO          Module: 1 Code: 1 Pile-up  Field: 0x00000
Word: 0xC02201D800691D58 Timestamp: 0x05CC10691D58 Type: ADC           Module: 1 Channel: 3  E Conversion: 0x01D8 Veto: 0 Fail: 0
Word: 0xC023158200691F6B Timestamp: 0x05CC10691F6B Type: ADC           Module: 1 Channel: 4  E Conversion: 0x1582 Veto: 0 Fail: 0
etc. etc.
```

First comes the word read from the file. Note that this is after bit-swapping, so this is a test if the bit swapping is correct. Then comes the full 48-bit timestamp. Note that as the top 20 bits come from the SYNC word, these will be zero for the events in a file up to the first SYNC.

7.2 stats.C

The *stats.C* example goes through the file and writes out statistics for each ID in that file. It takes a single parameter, which is the name of the data file.

```
Acquisition time: 591.505 seconds
ID    Total      Pileup     Overrange  Underrange Overflow   Underflow  Rate [/s]
32    853501     8761       396        0          0          59         1442.931
33    943724     11597      378        0          0          321        1595.462
34    951393     11062      415        0          0          84         1608.428
35    965842     11584      373        0          0          234        1632.855
```

```
36    1121674   15400     379       0         0         41        1896.305
37    1063291   13562     396       0         0         248       1797.603
38    984700    11668     447       0         0         12        1664.737
39    1153932   15921     417       0         0         143       1950.841
40    255       0         0         0         0         0         0.431
41    17        0         0         0         0         0         0.029
47    59150     0         0         0         0         0         99.999
64    981720    12217     391       0         0         164       1659.699
65    914446    10521     400       0         0         378       1545.965
66    962751    12173     411       0         0         57        1627.630
67    980516    11691     403       0         0         318       1657.663
68    1189529   18038     362       0         0         134       2011.021
69    1155700   17589     397       0         0         321       1953.830
70    1184367   17835     397       0         0         335       2002.294
71    1011813   13939     367       0         0         51        1710.574
72    4251660   89436     0         0         0         45100     7187.868
73    3099014   48178     0         0         0         129409    5239.202
74    7460623   284176    0         0         0         103752    12612.950
79    59150     0         0         0         0         0         99.999
103   59150     0         0         0         0         0         99.999
109   2688820   83852     890       0         0         29546     4545.727
110   2372282   61929     775       0         0         104       4010.587
111   2781062   88883     759       0         0         68004     4701.671
```

The total number is the number of ADC words, including failed conversions. The "pileup", "overrange", "underrange", "overflow" an "underflow" are obtained from the information words with codes 2, 9, 10, 11, and 12, respectively. The rate is just the total divided by the acquisition time, which is determined by comparing the first and last SYNC100 timestamps.

7.3 proj.C

The *proj.C* example generates the projection histograms for each ID in the file. These are written to a root file (by default proj.root). The first parameter is the data file. The second is the output root file into which the histograms are written ("proj.root" by default). The third parameter is the calibration file ("online.gains" by default). This has the same format as the calibration file used by grain. e.g.

```
# Clover 101
32= 1.890477 0.28199703 2.622174E-7 0
33= -1.759081 0.28671525 0 0
34= 1.379345 0.284008880 0 0
35= -0.637727 0.280889620 0 0
```

which gives the calibrations for IDs 32 to 35. The first three values after the equals sign are the offset, slope and quadratic term of the calibration. The fourth term is a time offset, which is not used by this code, but is included for compatibility with grain.

7.4 baselines.C

The *baselines.C* is a simplified version of *proj.C* which only generates the baseline histograms (if the baseline data is in the file). We change the IDs, however, so that the histogram has the ID of the corresponding energy histogram. i.e. h0032 is the baseline histogram corresponding to module 1 channel 1. In *proj.C* this baseline histogram would be h0048 and h0032 would be the energy histogram of that channel.

7.5 show_supercycle.C

The *show_supercycle.C* example creates a single histogram showing the time between an event with ID 41 (assumed to be the start of the proton supercycle = PS) and one with ID 40 (assumed to be the proton impact on the target = T1). i.e. it shows the structure of the supercycle. There is a small offset from the PS to the first possible proton pulse, then the proton pulses are 1.2 seconds apart, but we won't get all of them. It takes a single parameter, which is the name of the data file.

7.6 check_SYNC.C

The *check_SYNC.C* example shows the timestamps for SYNC events for each module in the first and last 200 ms on a canvas. The pattern it generates is meaningless and is an artefact of the binning. The common zero is determined by the first SYNC timestamp in the file. If everything is working correctly, the pattern should be the same for each module. If the modules are out of synch, one module might start later and there would be a gap at the start. It takes a single parameter, which is the name of the data file.

7.7 check_SYNC2.C

The *check_SYNC2.C* example is an alternative way of checking the SYNCs. It looks for the first SYNC of each module in the file and compares the timestamps. It takes a single parameter, which is the name of the data file.

```
0 0x05CC1068001A 0.000 s
1 0x05CC1068001A 0.000 s
2 0x05CC1068001A 0.000 s
```

The first number is the module number, the second is the timestamp and the third is the difference relative to the lowest timestamp. If they are in synch, the three timestamps should be the same. A bad file might give:

```
0 0x204E5876001A 59.992 s
1 0x204D991A001A 27.887 s
2 0x204CF2E2001A 0.000 s
```

This is what happens, when each module starts its clock on booting and they are never resynched. The time difference is the time needed to boot.

7.8 timestamping.C

The *timestamping.C* example generates timestamp-difference histograms between the pairs of channels. It takes the data file as the first parameter and the output root file where it writes the resulting histograms as the second (default = "timestamping.root").

With this you can determine the offsets in the timestamps, provided you have coincident events. Note, that the technique of reading up to a highwater mark and then sorting the events by timestamp and processing the first half of them, is designed to work even if the data are not properly time-ordered. This should no longer be needed! However, it shouldn't be a problem.

7.9 build_tree.C

The *build_tree.C* example creates a root tree of N'tuples (2048'tuples, in fact), corresponding to the conversions of each ID. i.e. the array is indexed using the ID.

The tree can be accessed with code like:

```
#define MAXID 0x1000
   UInt_t energies[MAXID];

   TFile *f = TFile::Open("input_file", "read");
   TTree *t = (TTree *)f->Get("great");
   t->SetBranchAddress("energies", energies);

   for (UInt_t i = 0; i < t->GetEntries(); i++) {
      t->GetEntry(i);

      // Here, energies is filled out for that event
   }
```

At the point labelled "Here", the energies array is set up for that event. So, for example, energies[32] contains the ADC value corresponding to ID 32 (= module 1, channel 1, energy).

The IDS 0 to 6 have a special meaning.

ID	Meaning
0	Time between supercycle start and proton impact [ms]
1	Time between proton impact and previous proton impact [ms]
2	Time between supercycle start and beam gate open [ms]
3	Time between supercycle start and beam gate close [ms]
4	Time between beam gate open and close [ms]
5	Time between supercycle start and tape movement start [ms]
6	Beam gate open [boolean]