MED Data Structure

R. Lutter

October 17, 2018

Abstract

This document describes the structure of experimental data taken in MINIBALL experiments using the MARaBOU data acquisition system. **MED** is an abbreviation for "**M**BS **E**vent **D**ata" as this format is based on regular MBS data structures [1].
A detailed description of MBS data structures used as well as MARaBOU extensions to these structures will be given.

Contents

1 MED file format

A `.med` file contains a stream of MBS events of standard type [10,1] (known as "`VME Event`" inside MBS). In contrast to the MBS file format (`.lmd` format) no buffering is used during output: data are streamed out event by event by the generating program. As a consequence there is no event spanning across buffer boundaries making things easier for the reader. Note that there is neither a file header nor any buffer header, too. Each event contains a sequence of subevents all based on MBS subevent [10,1] (so-called "`CAMAC Subevent`"). There are several extensions to this subevent type to cover different hardware and software requirements within MAR$_a$BQU.
Fig. 1 shows the overall MED data structure, table 1 gives a list of subevent types used by MAR$_a$BQU applications [2].

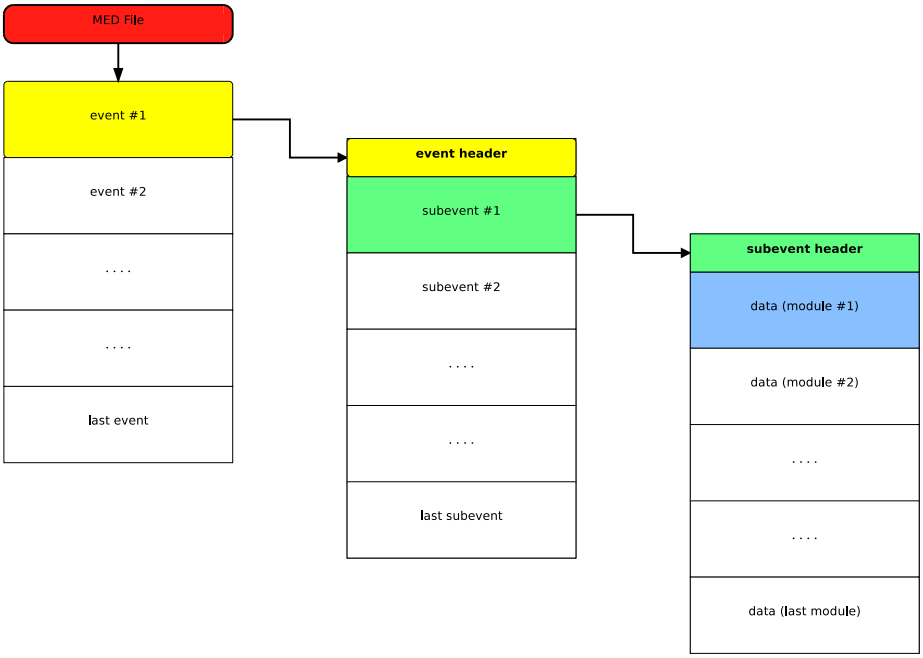

Figure 1: Overall structure of MED data

2 Event and subevent formats used by MAR$_a$BQU

Fig. 2 shows standard MBS event and subevent headers as used in a MAR$_a$BQU environment.

[type,subtype]	class name	format	mps [a]	comment
[10,1]	TMrbSubevent_10_1	zero-compressed, preceeded by channel number	1 only	universal MBS subevent
[10,11]	TMrbSubevent_10_11	data w/o channel numbers, zero-padded	1 only / any [b]	universal MARaBQQU subevent
[10,12]	TMrbSubevent_10_12	same as [10,1], including module headers	any	
[10,21]	TMrbSubevent_DGF_1	original XIA format	any	for XIA DGF-4C modules
[10,22]	TMrbSubevent_DGF_2 [c]	...	any	...
[10,23]	TMrbSubevent_DGF_3 [c]	...	any	...
[10,31]	TMrbSubevent_Silena_1	zero-compressed Silena format	any	for Silena 4418V/T modules
[10,32]	TMrbSubevent_Silena_2 [c]	...	any	...
[10,41]	TMrbSubevent_Caen_1	original CAEN format	any	for CAEN V785/V775 modules
[10,42]	TMrbSubevent_Caen_2 [c]	...	any	...
[10,43]	TMrbSubevent_Caen_3 [c]	...	any	...
[10,51]	TMrbSubevent_Sis_1	original SIS format	any	for SIS 3XXX modules
[10,52]	TMrbSubevent_Sis_2 [c]	...	any	...
[10,53]	TMrbSubevent_Sis_3 [c]	...	any	...
[10,54]	TMrbSubevent_Sis_33	special format	any	for SIS3302 tracing adc
[10,81]	TMrbSubevent_Mesytec_1	original MESYTEC format	any [d]	for MADC/MTDC/MQDC/MDPP16 modules
[10,82]	TMrbSubevent_Mesytec_2 [c]	...	any [d]	...
[10,83]	TMrbSubevent_Mesytec_3 [c]	...	any [d]	...
[10,91]	TMrbSubevent_Data_S	short (16 bit) data	–	universal data container
[10,92]	TMrbSubevent_Data_I	int (32 bit) data	–	...
[10,93]	TMrbSubevent_Data_F	float (32 bit) data	–	...
[9000,1]		time stamp	–	ppc clock, in steps of 100 μs
[9000,2]		dead time	–	contents of dead time scaler
[111,111]		default	–	default (empty) subevent

Table 1: Subevent types used by MARaBQQU

[a] modules per subevent

[b] As there is no module id in this format the sequence of modules has to be known to the reader. To avoid ambiguities it is recommended to store 1 module per subevent only.

[c] Note: formats 2 and 3 have **same** data structure on input but follow different output strategies

[d] Although any number of modules is possible it is recommended to store only 1 module per subevent.

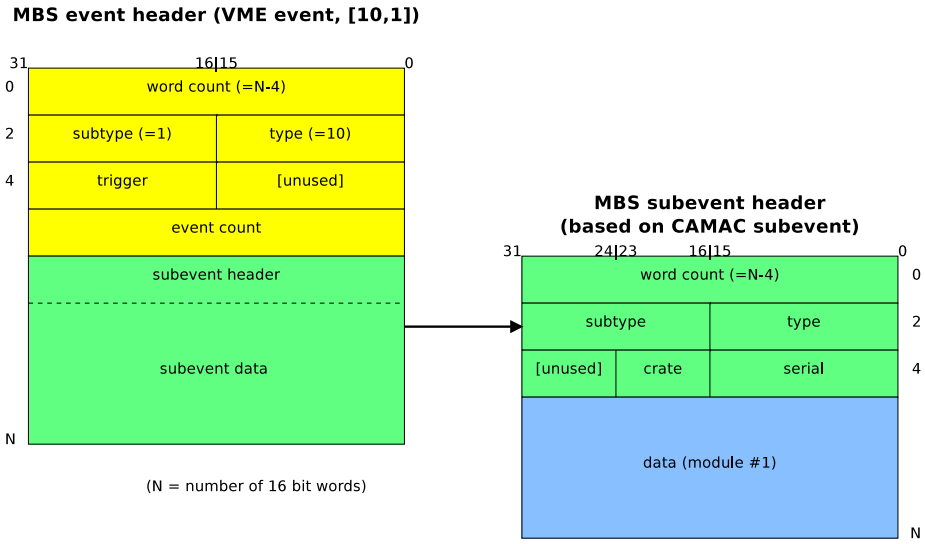

Figure 2: Standard MBS headers used in MARaBOU [1]

event header	
`word count`	number of 16 bit words for this event [a]
`subtype`	event subtype (=1)
`type`	event type (=10)
`trigger`	trigger number
`event count`	MBS event count
subevent header	
`word count`	number of 16 bit words for this subevent [a]
`subtype`	subevent subtype [b]
`type`	subevent type [b]
`crate`	crate number (VME=0, CAMAC=1,2,...)
`serial`	subevent serial number [c]

[a] excluding first 2 header words, thus event/subevent length is (N=wc+4) 16 bit words
[b] see table 1
[c] assigned sequentially during `Config.C` step

Note: Data have differently to be swapped on input depending on data type (8/16/32 bits)!

2.1 Universal data storage: subevent formats [10,1] and [10,11]

Subevent formats [10,1] and [10,11] are universal formats to store module data in a straightforward way. Format [10,1] contains zero-compressed data preceeded by channel numbers; it is therefore recommended for modules having a large number of channels, but only a few hits. Format [10,11] contains one data item per channel, missing channels are padded with a zero data value. Thus this format is more applicable to store module data where most of the channels have converted. As there is no module identification inside these formats it is recommended to store only one module per subevent. Data have to be aligned to 32 bit boundaries, so in case of an odd number of module channels there is a filler (0xFFFF) at end of data.

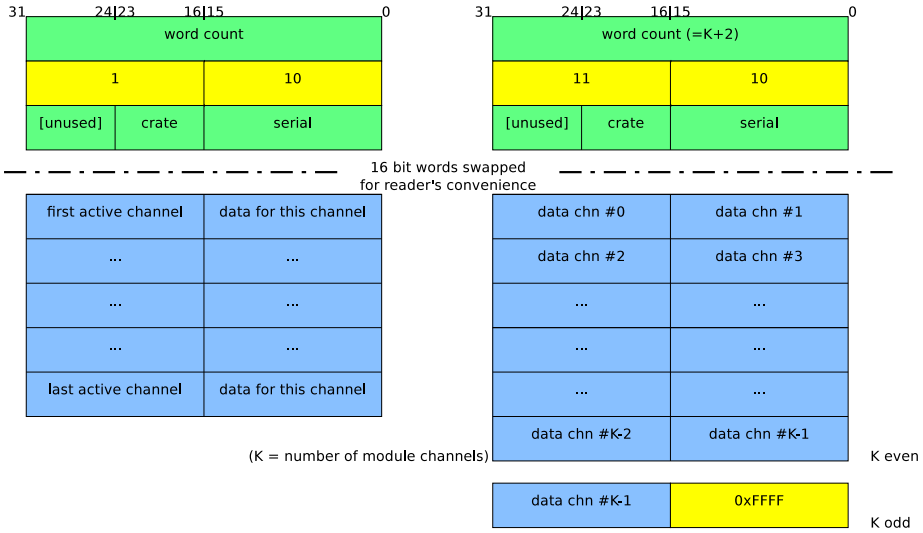

Figure 3: Subevent formats [10,1] and [10,11]: universal storage

2.2 Multi-module extension: subevent format [10,12]

Subevent format [10,12] is an extension to format [10,1]: zero-compressed data preceeded by channel numbers are written together with a module header. Thus several modules may easily be stored in one subevent.

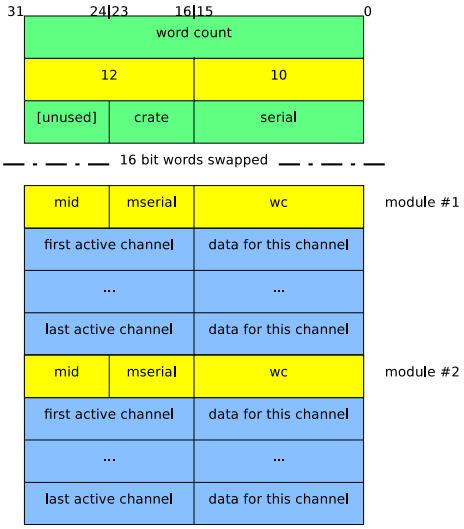

Figure 4: Subevent format [10,12]: data zero-compressed, any number of modules

`mid`	module id
`mserial`	module serial number [a]
`wc`	word count, including header words

[a] assigned sequentially during `Config.C` step

2.3 XIA DGF-4C data: subevent formats [10,21], [10,22], and [10,23]

Formats [10,2X] are used to store original buffers read from XIA DGF-4C modules [3].

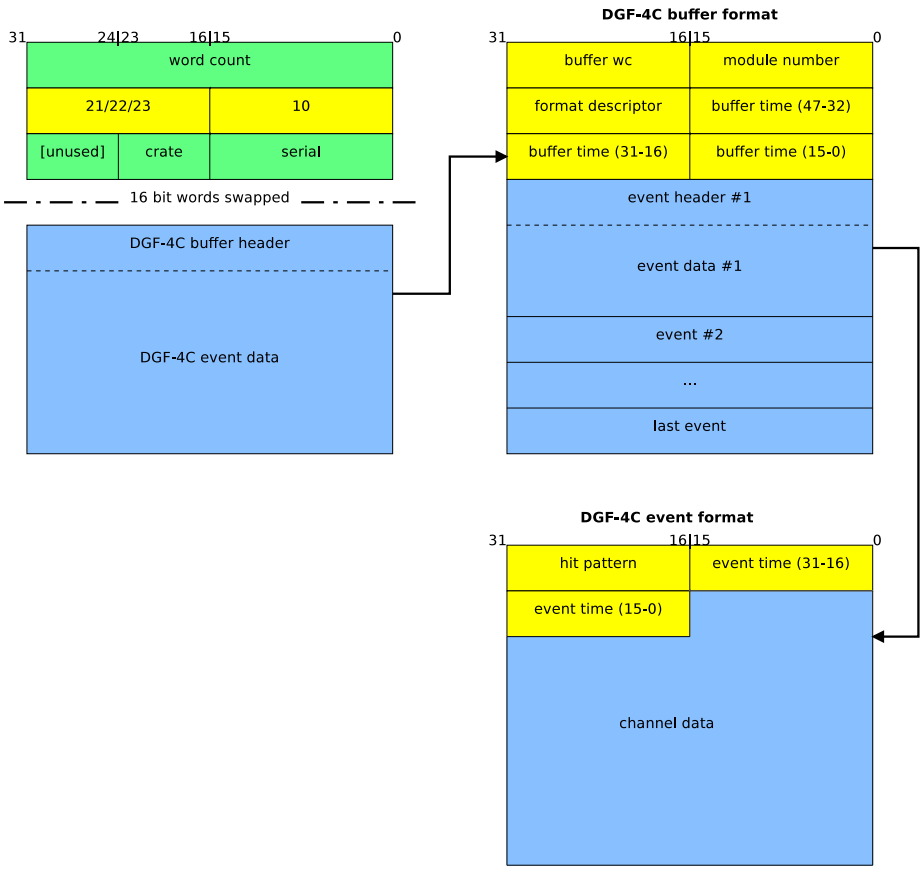

Figure 5: Subevent format [10,2X]: DGF-4C buffer data

	buffer header
`buffer wc`	number of 16 bit words in this buffer
`module number`	module serial number [a]
`format descriptor`	data format used for channel data
`buffer time`	48 bit buffer starting time
	event header
`hit pattern`	one bit per active channel
`event time`	32 bit event starting time

[a] assigned sequentially during `Config.C` step

Several list mode formats are available to control the DGF-4C data flow. Depending on the value of the format descriptor in the buffer header (fig. 5) long or short channel headers with or without trace data will be written. Fig. 6 shows different channel layouts. For a detailed description see [3].

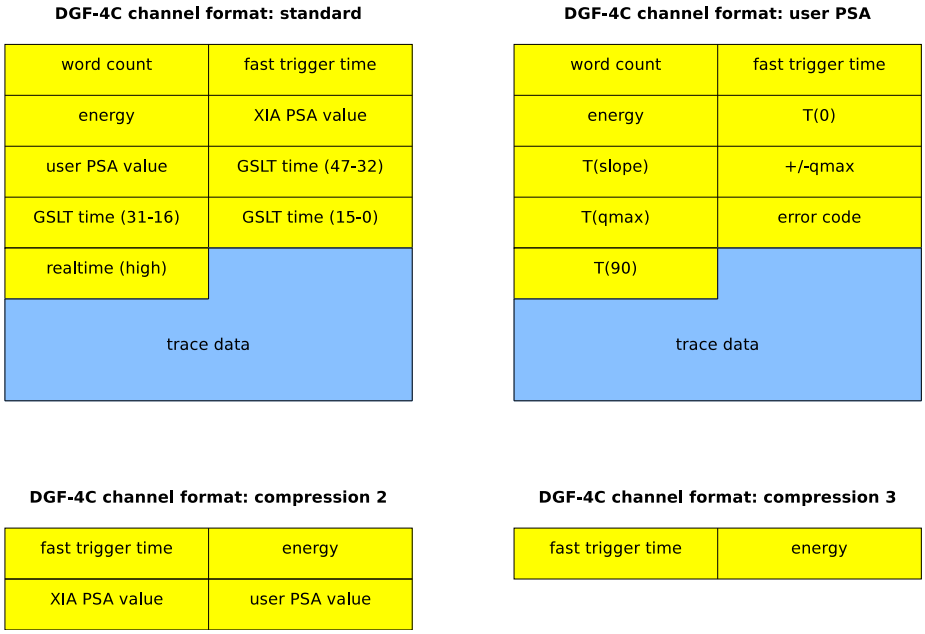

Figure 6: Subevent format [10,2X]: DGF-4C list mode formats

channel header	
`word count`	number of 16 bit words written for this channel
`fast trigger time`	time of arrival
`energy`	converted energy value
`PSA value`	result of pulse shape analysis (XIA and user)
`GSLT time`	48 bit arrival time of global second level trigger
`realtime`	time since last reboot or reset (high word: bits 47-32)
`trace data`	array containing trace data depending on format descriptor

2.4 Silena 4418V/T data: subevent formats [10,31] and [10,32]

Formats [10,31] and [10,32] are used to store zero-compressed data from Silena 4418V/T modules. Several modules may be stored in one subevent. In case of uncompressed Silena data subevent type [10,11] has to be used instead (one module per subevent only).

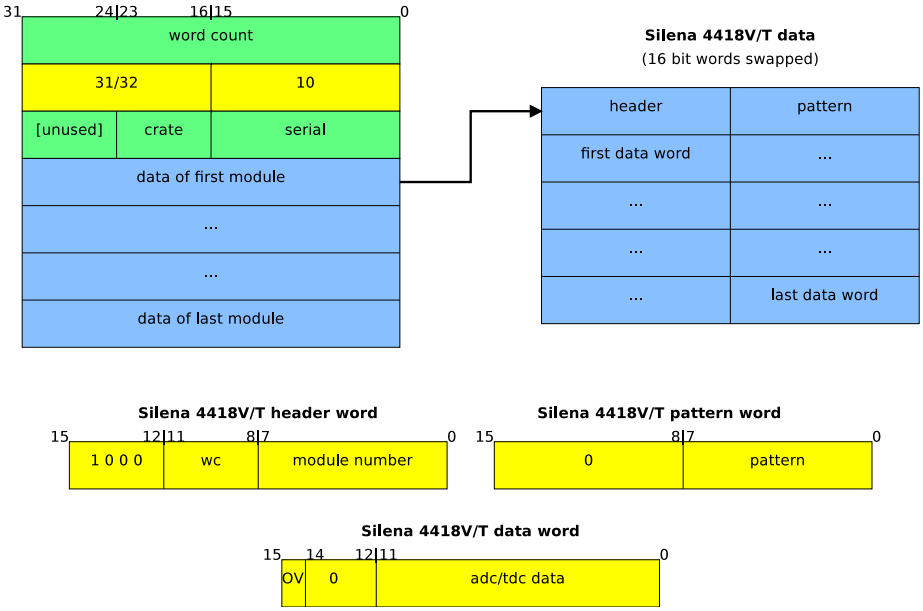

Figure 7: Subevent format [10,3X]: Silena 4418V/T data

`wc`	number of words (including header and pattern)
`module number`	module number [a]
`pattern`	8 bit pattern word: active channels have bit=1
`data`	12 bit adc/tdc data

[a] assigned sequentially during `Config.C` step

2.5 CAEN V7X5 data: subevent formats [10,41], [10,42], and [10,43]

Formats [10,4X] provide containers for original CAEN list mode data produced by modules CAEN V785 and CAEN V775, respectively. Each CAEN buffer may contain up to 32 events. In addition, as each event is tagged with module number one may store data from several CAEN modules in one subevent. To be able to correlate time stamps in DGF and CAEN branches in MINIBALL experiments data have to be stored one module per subevent, however. A detailed description of this format may be found in [4]

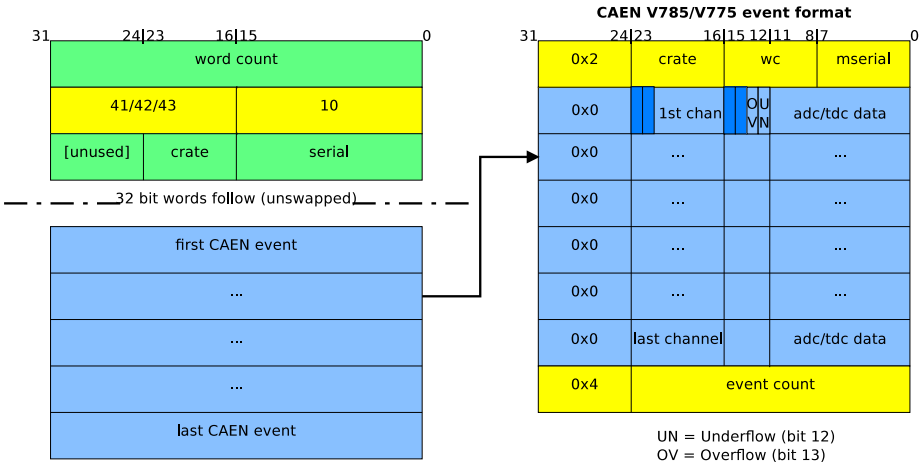

Figure 8: Subevent format [10,4X]: CAEN V7X5 ADC/TDC data

`0x2`	header word including word count and module number [a]
`0x0`	data word: channel number and converted data[a]
`0x4`	trailer word: event count[a]
`crate`	crate number [b]
`wc`	number of channel data (32 bit, excluding header & trailer)
`mserial`	module serial number [c]
`channel`	channel number (0 . . . 31)
`data`	12 bit adc/tdc data + overflow (OV, bit 13) + underflow (UN, bit 12)
`event count`	number of events since last reset

[a] GEO address not used
[b] currently unused in MARaBOU
[c] assigned sequentially during `Config.C` step

2.6 CAEN V965 data: subevent format [10,44]

Format [10,44] provides a container for data produced by a CAEN V965 QDC. It is very similar to formats [10,4X] described in 2.5. As the qdc module integrates twice for each input there may be two data words per channel. The range bit then denotes whether it is the low (fine gain) or high (coarse gain) integration.

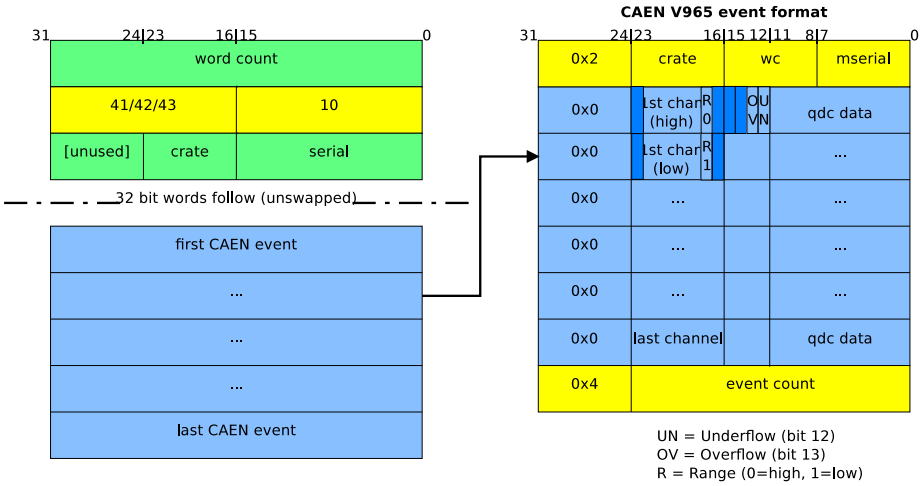

Figure 9: Subevent format [10,44]: CAEN V965 QDC data

`0x2`	header word including word count and module number [a]
`0x0`	data word: channel number and converted data[a]
`0x4`	trailer word: event count[a]
`crate`	crate number [b]
`wc`	number of channel data (32 bit, excluding header & trailer)
`mserial`	module serial number [c]
`channel`	channel number (V965A: 0 ... 7, V965: 0 ... 15) + range bit (bit 17)
`data`	12 bit adc/tdc data + overflow (OV, bit 13) + underflow (UN, bit 12)
`event count`	number of events since last reset

[a] GEO address not used
[b] currently unused in MARaBOU
[c] assigned sequentially during `Config.C` step

2.7 SIS 3XXX data: subevent formats [10,51], [10,52], and [10,53]

Formats [10,5X] are designed to store data produced by SIS 3600 or SIS 3801 modules. This format is identical to format [10,12].

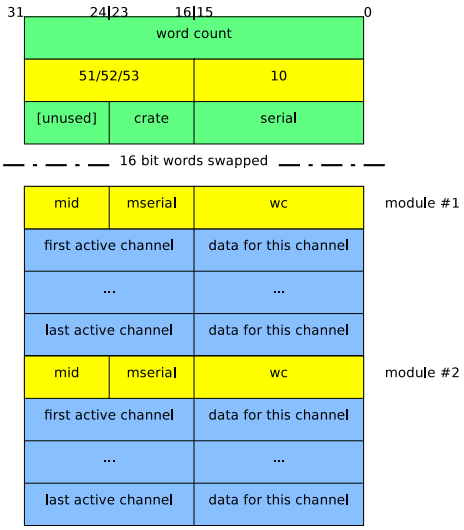

Figure 10: Subevent format [10,5X]: SIS 3XXX data

`mid`	module id
`mserial`	module serial number [a]
`wc`	word count, including header words

[a] assigned sequentially during `Config.C` step

Note: In current MINIBALL experiments data from SIS 3801 scalers will be written using format [10,11] rather than this one.

2.8 SIS3302 data: subevent format [10,54]

Format [10,54] is used to store data from module SIS3302.

2.9 MESYTEC data: subevent formats [10,81], [10,82], and [10,83]

Formats [10,8X] are designed to store data produced by Mesytec modules of type MADC32, MTDC32, MQDC32, and MDPP16.

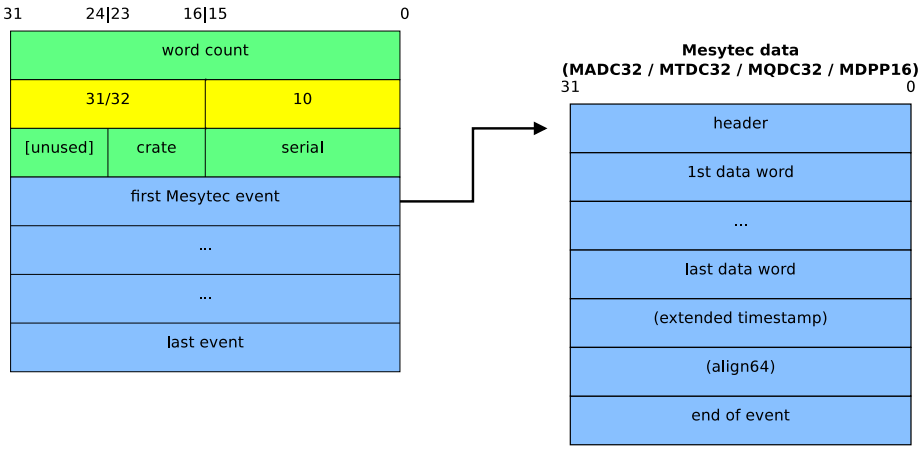

Figure 11: Subevent format [10,8X]: Mesytec data structure

`header`	header word (0b01 / 0x4...): module id, adc/tdc resolution, wc
`data word`	data (0b00 / 0x04...): channel, adc amplitude, tdc time diff
`end of event`	trailer (0b11): event counter / timestamp low bits 0-29[a]
`extended timestamp`	timestamp high bits 30-45 (0b00 / 0x048...)[a]
`align64`	zero data word to align to 64 bit boundaries[b]

[a] depending on register 0x6038
[b] depending on word count

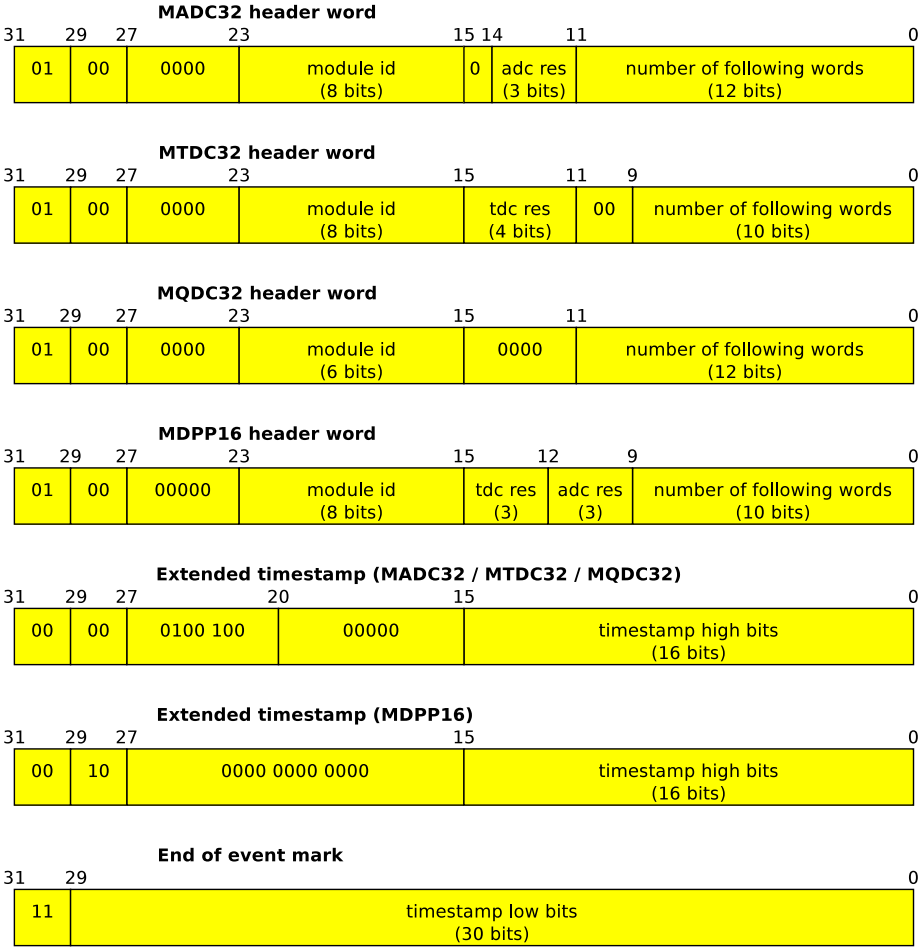

Figure 12: Subevent format [10,8X]: Mesytec: header & trailer words

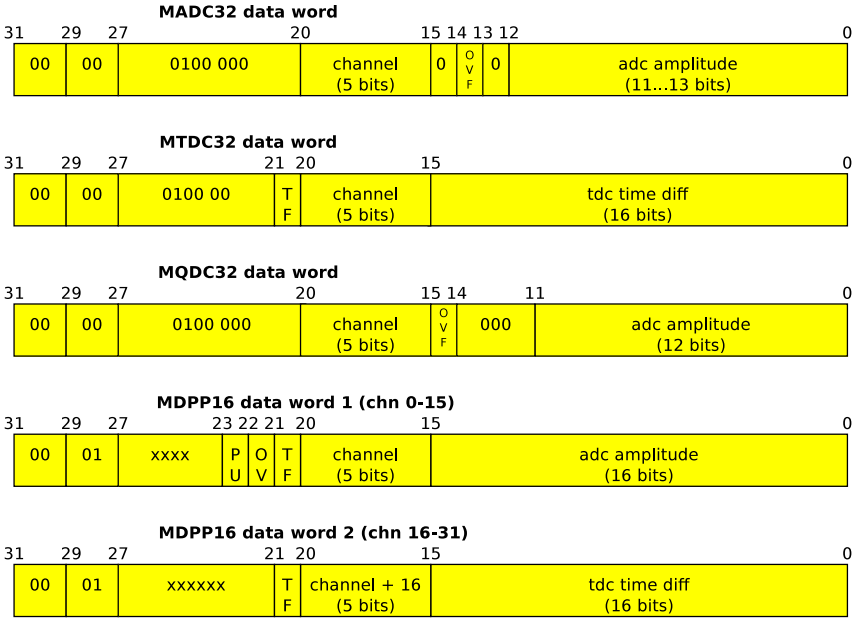

Figure 13: Subevent format [10,8X]: Mesytec: data words

2.10 Plain data containers: subevent formats [10,91], 10,92], and [10,93]

Formats [10,9X] provide containers to store data that are not directly related to a hardware module (e.g. internal DGF scalers). There are containers for short [10,91], long [10,92], and float items [10,93], respectively.

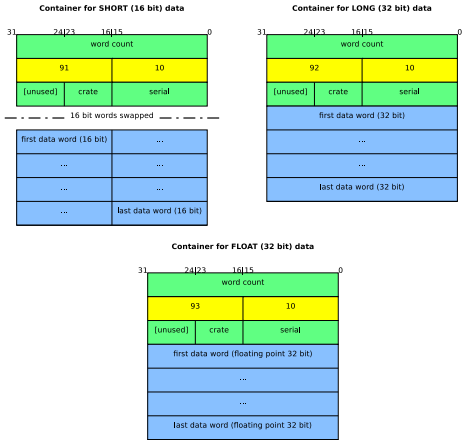

Figure 14: Subevent format [10,9X]: Plain data containers

3 User Interface to MED data (C API)

This section describes the C user interface which may be used to access MED data without running ROOT. It includes function calls to

- open med files
- read event by event and dispatch over event trigger
- decode subevent header
- extract subevent data and dispatch over subevent type and/or serial number

Any prototype for this user interface is defined in file `mbsio.h`. So you have to include this file in front of your code:

```
#include <stdio.h>
#include "mbsio.h"
```

To include this package in a `C++` environment, add prototype definitions from file `mbsio_protos.h`:

```
#include "mbsio_protos.h"
```

3.1 Open a `.med` file

```
MBSDataIO * mbs_open_file(const char * FileName, const char * Connect,
                                                 int BufSize, FILE * HdrOut);
int mbs_close_file(MBSDataIO * MbsHandle);
```

`const char * FileName`	file name, extension has to be `.med`
`const char * Connect`	how to connect to med data stream, has to be `F` (”File”)
`int BufSize`	buffer size, normally `0x4000` (16k)
`FILE * HdrOut`	where to output header info, set to `NULL` for ”no output”
`MBSDataIO * MbsHandle`	file handle to access med data

`mbs_open_file` opens a med raw data file for reading and returns a file handle to refer to it.
`mbs_close_file` closes the file pointed to by the file handle.

Example:

```
MBSDataIO * mbsHandle;
mbsHandle = mbs_open_file("run123.med", "F", 0x4000, NULL);
if (mbsHandle = NULL) { printf("Error\n"); exit(1); }
```

3.2 Read event by event and dispatch over event trigger

```
unsigned int mbs_next_event(MBSDataIO * MbsHandle);
int mbs_get_event_trigger(MBSDataIO * MbsHandle);
```

`MBSDataIO * MbsHandle` file handle as returned by `mbs_open_file`

`mbs_next_event` moves on to next event, adjusts internal pointers. Returns event type `[subtype,type]` which is always `[1,10]`. User has to check for special return values `MBS_ETYPE_EOF` (end of file), `MBS_ETYPE_ERROR` (error), and `MBS_ETYPE_ABORT` (abort).
`mbs_get_event_trigger` returns trigger number of current event.

Example:

```
unsigned int evtType;
int evtTrigger;
while (1) {
    evtType = mbs_next_event(mbsHandle);
    if (evtType == MBS_ETYPE_EOF) {
          printf("End of file\n");
          mbs_close_file(mbsHandle);
    } else if (evtType == MBS_ETYPE_ERROR) {
          printf("Illegal event - skipped\n");
          continue;
    } else if (evtType == MBS_ETYPE_ABORT) {
          printf("Illegal event - aborting\n");
          break;
    } else {
          evtTrigger = mbs_get_event_trigger(mbsHandle);
          switch (evtTrigger) {
              case kMrbTriggerStartAcquisition:
                   printf("Trigger \"start acquisition\"\n");
                   break;
              case kMrbTriggerStopAcquisition:
                   printf("Trigger \"stop acquisition\"\n");
                   break;
              case kMrbTriggerReadout:
                   process_event(mbsHandle, evtTrigger);
                   break;
              case .....
                   break;
              default:
                   printf("Illegal trigger %d\n", evtTrigger);
                   break;
          }                    }
 }
```

3.3 Decode subevent header

```
unsigned int mbs_next_sheader(MBSDataIO * MbsHandle);
unsigned int mbs_get_sevent_subtype(MBSDataIO * MbsHandle);
int mbs_get_sevent_serial(MBSDataIO * MbsHandle);
```

`MBSDataIO * MbsHandle` file handle as returned by `mbs_open_file`

`mbs_next_sheader` moves on to next subevent of current event. Decodes header information and returns subevent type `[subtype,type]`. User has to check for special return values `MBS_STYPE_EOE` (end of event), `MBS_STYPE_ERROR` (error), and `MBS_STYPE_ABORT` (abort).
`mbs_get_sevent_subtype` returns subtype portion of subevent type (LH word of `[subtype,type]`, right-shifted).
`mbs_get_sevent_serial` returns serial number of current subevent.
Subevent type and/or serial number may then be used to dispatch to different decoding routines.

Example:

```
void process_event(MBSDataIO * mbsHandle, int evtTrigger) {
    unsigned int sevtType;
    int sevtSerial;
    while (1) {
        sevtType = mbs_next_sheader(mbsHandle);
        if (sevtType == MBS_STYPE_EOE) {
            return;
        } else if (sevtType == MBS_STYPE_ERROR) {
            printf("Illegal subevent - skipped\n");
            continue;
        } else if (sevtType == MBS_STYPE_ABORT) {
            printf("Illegal subevent - aborting\n");
            break;
        } else {
            sevtSerial = mbs_get_sevent_serial(mbsHandle);
            process_subevent(mbsHandle, sevtSerial);
        }
    }
}
```

3.4 Extract subevent data, dispatch over subevent serial and/or type

```
unsigned int mbs_next_sdata(MBSDataIO * MbsHandle);
int mbs_get_sevent_wc(MBSDataIO * MbsHandle);
unsigned short * mbs_get_sevent_dataptr(MBSDataIO * MbsHandle);
```

`MBSDataIO * MbsHandle` file handle as returned by `mbs_open_file`

`mbs_next_sdata` moves on to data section of current subevent, adjusts pointers. Returns subevent type `[subtype,type]`. User has to check for special return values `MBS_STYPE_ERROR` (error) and `MBS_STYPE_ABORT` (abort).
`mbs_get_sevent_wc` returns word count of current subevent (16 bit words).
`mbs_get_sevent_dataptr` returns pointer to first data word.

Example:

```
void process_subevent(MBSDataIO * mbsHandle, int sevtSerial) {
    unsigned int sevtType;
    unsigned short * dataPtr;
    int wc, clusterNo, caenNo;
    sevtType = mbs_next_sdata(mbsHandle);
    if (sevtType == MBS_STYPE_ERROR) {
          printf("Illegal subevent - skipped\n");
          return;
    } else if (sevtType == MBS_STYPE_ABORT) {
          printf("Illegal subevent - aborting\n");
          exit(1);
    } else {
          wc = mbs_get_sevent_wc(mbsHandle);
          dataPtr = mbs_get_sevent_dataptr(mbsHandle);
          switch (sevtType) {
              case MBS_STYPE_CAMAC_DGF_3:
                   clusterNo = sevtSerial - kMrbSevtClu1 + 1;
                   process_dgf_data(clusterNo, dataPtr, wc);
                   break;
              case MBS_STYPE_VME_CAEN_3:
                   caenNo = sevtSerial - kMrbSevtCaen1 + 1;
                   process_caen_data(caenNo, dataPtr, wc);
                   break;
              case .....
          }
     }
}
```

User should refer to `mbsio.h` for possible subevent types `MBS_STYPE_<XXX>` and to `DgfCommonIndices.h` for valid serial numbers `kMrbSevt<xxx>` defined for his experiment.

4 Appendix

4.1 C structure MBSDataIO

C structure `MBSDataIO` holds all information needed to describe an open connection to a `.med` data file. In addition to the methods described so far user may access all of its elements by addressing

```
mbsHandle->element_name
```

A description of all data members of structure `MBSDataIO`:

`char id[16];`	internal struct id: `%MBS_RAW_DATA%`
`FILE *input;`	input stream descr (fopen/fread)
`int fileno;`	channel number (open/read)
`char device[MBS_L_STR];`	name of input dev
`char host[MBS_L_STR];`	host name
`unsigned int connection;`	device type, `MBS_DTYPE_xxxx`
`MBSBufferElem *buftype;`	buffer type
`int byte_order;`	byte ordering
`MBSShowElem show_elems[MBS_N_BELEMS];`	buffer elements to be shown automatically
`int bufsiz;`	buffer size
`MBSServerInfo *server_info;`	info block for server access
`int max_streams;`	max number of streams to process
`int slow_down;`	number of secs to wait after each stream
`int nof_streams;`	number of streams processed so far
`int nof_buffers;`	number of buffers
`int nof_events;`	number of events
`int cur_bufno;`	buffer number
`int cur_bufno_stream;`	... within current stream
`int bufno_mbs;`	buffer number as given by MBS
`int buf_to_be_dumped;`	if `n>0` every n^{th} buffer will be dumped
`char *hdr_data;`	file header data
`MBSBufferPool buf_pool[MBS_N_BUFFERS];`	buffer pool
`MBSBufferPool * poolpt;`	... pointer to current buffer in pool
`char *bufpt;`	pointer to current data
`int buf_valid;`	`TRUE` if buffer data valid
`int buf_oo_phase;`	buffer out of phase
`MBSBufferElem *evttype;`	event type
`int evtsiz;`	event size (bytes)
`char *evtpt;`	ptr to current event in buffer
`int evtno;`	current event number within buffer
`int evtno_mbs;`	event number as given by MBS
`char *evt_data;`	copy of event data (original, byte-swapped if necessary)
`MBSBufferElem *sevttype;`	subevent type
`int sevtsiz;`	subevent size (bytes)
`char *sevtpt;`	ptr to original subevent in `evt_data`
`int sevtno;`	current subevent number within event
`int nof_sevents;`	number of subevents
`int sevt_id;`	current subevent id
`unsigned int sevt_otype;`	original subevent type `[subtype,type]`
`int sevt_minwc;`	min number of data words expected
`int sevt_wc;`	number of data words
`char *sevt_data;`	ptr to subevent data (unpacked)

References

[1] H. Essel et al.: GOOSY Buffer Structure.
See `http://wwwgsivms.gsi.de/goodoc/GM_BUFFER.ps`

[2] R. Lutter, O. Schaile et al.:
MARaBOU - MBS and ROOT Based Online/Offline Utility.
See `http://www.bl.physik.unimuenchen.de/marabou/html`

[3] X-Ray Instrumentation Associates: DGF-4C User's Manual, DGF-4C Programmer's Manual

[4] CAEN S.p.A: V785/V775 Technical Information Manual.
See `http://www.caen.it/nuclear/product.php?mod=V785`