Global Event Header Type Definitions

Revised October 18, 2015 by H. L. Crawford (hlcrawford@lbl.gov)

This document provides a brief description of the various types of global event data.

Global event header format:

```
struct GEBHeader {
  int32_t type;
  int32_t length; /* length of payload following the header, in bytes */
  int64_t timestamp;
}
```

Type 1: Decomposed GRETINA Data

This data type is the decomposed GRETINA data (mode 2) format. Mode 2 data is output from the signal decomposition process, and gives interaction points and energies from a single crystal. The format is defined as a fixed-length C struct (can be found in GRETINA DAQ software in `/global/devel/XX-X/decompLib/decompLibApp/gdecomp.h`). Two versions of this data type have existed, with two different identifiers, as outline below. The two mode2 data formats cover eras from June 2012 - present, while the previous version existed from GRETINA Commissioning Run (∼September 2011) - June 2012. Prior to the commissioning run phase (and prior to the use of global event headers), mode2 data had a format without the 'type' identifier, which is also shown below, for completeness. Definition of Mode2 data formats controlled by M. Cromaz (LBNL).

Decomposed GRETINA Data Format, June 2012 - now:

```
#define MAX_INTPTS 16

struct crys_intpts {
  int type; /* as of June2012:  abcd5678 */
  int crystal_id;
  int num; /*# of interaction points from decomp, or # of nets on decomp error */
  float tot_e; /* CC energy for the central contact selected for use in decomp
                  (calibrated, and for 10MeV channels, includes DNL correction) */
  int core_e[4]; /* 4 raw core energies from FPGA filter (uncalibrated) */
  long long int timestamp;
  long long trig_time; /* not yet implemented */
  float t0;
  float cfd;
  float chisq;
  float norm_chisq;
  float baseline;
  float prestep; /* avg trace value before step (baseline) */
  float poststep; /* avg traces value after step (flat-top) */
  int pad; /* non-0 with a decomp error, value gives error type */
  struct {
    float x, y, z, e; /* here e refers to the fraction */
    int seg; /* segment number hit */
    float seg_ener; /* energy of the hit segment */
  }intpts[MAX_INTPTS];
};
```

Decomposed GRETINA Data Format, Start of GRETINA Commissioning Runs - June 2012:

```
#define MAX_INTPTS 16

struct crys_intpts {
  int type; /* up until June2012:  abcd1234 */
  int crystal_id;
  int num; /*# of interaction points from decomp, or # of nets on decomp error */
  float tot_e; /* CC energy for the central contact (calibrated) */
  long long int timestamp;
  long long trig_time; /* not yet implemented */
  float t0;
  float cfd;
  float chisq;
  float norm_chisq;
  float baseline;
  int pad; /* non-0 with a decomp error, value gives error type */
  struct {
    float x, y, z, e; /* here e refers to the fraction */
    int seg; /* segment number hit */
    float seg_ener; /* energy of the hit segment */
  }intpts[MAX_INTPTS];
};
```

Decomposed GRETINA Data Format, Start of GRETINA Project - ∼September 2011:

```
#define MAX_INTPTS 16

struct crys_intpts {
  int num; /*# of interaction points from decomp, or # of nets on decomp error */
  int crystal_id;
  float tot_e; /* CC energy for the central contact selected for use in decomp */
  float t0;
  float chisq;
  float norm_chisq;
  long long int timestamp;
  struct {
    float x, y, z, e; /* here e refers to the fraction */
  }intpts[MAX_INTPTS];
};
```

Type 2: Raw GRETINA Data

This data type contains raw (Mode 3) data output directly by the GRETINA digitizers. Each event corresponds to a single digitizer channel, and includes a header with timestamp and energy information, as well as a section of trace. The detailed format is presented as a block diagram below:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Board ID[1]																Packet Length (header included)											GA				
LED/external timestamp bits 0-15*																LED/external timestamp bits 16-31*															
LED/external timestamp bits 32-48*																Energy bits 0 -15															
Energy bits 16-24								X	X	T	S	E	C	P		CFD timestamp bits 0-15															
CFD timestamp bits 16-31																CFD timestamp bits 32-47															
CFD point 1 bits 0-15																CFD point 1 bits 16-31															
CFD point 2 bits 0-15																CFD point 2 bits 16-31															
Raw data point 0 (sign extended)																Raw data point 1 (sign extended)															
Raw data point 2 (sign extended)																Raw data point 3 (sign extended)															
...																...															
...																...															
...																...															

The energy is a 24-bit 2's complement signed integer, and must be converted to a 32-bit signed integer before use. The detector ID is a compound word consisting of 4 bit-fields as follows:

```
<5-bit detector #><2-bit crystal #><2-bit VME slot #><4-bit channel #>,
```

where the channel numbers range from 0-9, and the VME slot # from 0-3 (corresponding to the 4 digitizers within one bank of electronics). Currently the CFD fields are invalid, as the constant fraction discriminator is not fully implemented.

As Mode 3 (raw) data is the most difficult to interpret, due to the required reconstruction of energies, etc. from the data, a basic sample unpacker is included as an appendix to this document.

Type 3: Tracked GRETINA Data

This data type contains tracked GRETINA (mode 1) data. Mode 1 data is a superset of mode 2 (type 1) data, in that all interaction point information, etc. is maintained, and the results of tracking, namely a grouping of interaction points into a gamma-ray event, and a preferred ordering (first-hit identification) of interaction points is included. The structure of this data is similar to mode2, using a fixed length C struct, as defined below (can also be found in GRETINA DAQ software in `/global/devel/XX-X/trackLib/trackLibApp/ctk.h`). Definition of Mode2 data formats controlled by T. Lauritsen (ANL).

```
#define MAX_NDET 30

struct CLUSTER_INTPTS {
  int valid;
  int ndet;
  int tracked;
  float fom;
  float esum;
  int trackno;
  int bestPermutation;
  int processed;
  struct {
    float xx, yy, zz;
    float edet;
    int order;
    long long int timestamp;
    int shellHitPos;
    int detno;
  }intpts[MAX_NDET];
```


```
};
```

Type 4: BGS Raw Data

This data type contains raw BGS data, which is a simple array of short integers, which is mapped simply to values read from the data acquisition. A parameter list, which simply provides a label to each short integer in the data array provides the framework to unpack the BGS data. The parameter list for the BGS data in the commissioning runs of GRETINA is presented below. Additional information regarding the BGS data structure can be obtained from H. Crawford (hlcrawford@lbl.gov).

```
/* Parameter list for the C3 BGS FP Detector.  */

/* From the CAEN V830 scaler */
#define iusec  1  /* bits 00-15 10MHz pulse train */
#define iUSEC  2  /* bits 16-31 10MHz pulse train */
#define ireqt  3  /* bits 00-15 of requested triggers */
#define iREQT  4  /* bits 16-31 of requested triggers */
#define iacct  5  /* bits 00-15 of accepted triggers */
#define iACCT  6  /* bits 16-31 of accepted triggers */
#define istmp  7  /* bits 00-15 of GRETINA timestamp */
#define iSTMP  8  /* bits 16-31 of GRETINA timestamp */

/* From the SIS3806 scaler */
#define iuswh  24  /* bits 00-15 of time since wheel */
#define iUSWH  25  /* bits 16-31 of time since wheel */
#define iusbp  26  /* bits 00-15 of time since pause */
#define iUSBP  27  /* bits 16-31 of time since pause */
#define ibit6  28  /* user bits from SIS3806 scaler */

/* From the CAEN V262 I/O module */
#define iV262  29  /* CAEN V262 I/O module status */
#define ierr1  30  /* readout error location */
#define ierr2  31  /* readout error type */

#define nch  3
#define nfp  32  /* # of signals in FP strip groups */
#define npt  32  /* # of punchthru signals */
#define nus  24  /* # of upstream signals */
#define nge  12  /* # of gamma detector signals */
#define nws  8  /* # of wing detector signals */

/* V785 ADC #0, 1 N568 AMP #0, 1 */
#define iftl  32  /* index of 0th lo E signal from front of top chip */
#define ifth  64  /* index of 0th hi E signal from front of top chip */

/* V785 ADC #2, 3 N568 AMP #2, 3 */
#define ifwl  96  /* index of 0th lo E signal from front of west chip */
#define ifwh  128  /* index of 0th hi E signal from front of west chip */

/* V785 ADC #4, 5 N568 AMP #4, 5 */
#define ifel  160  /* index of 0th lo E signal from front of east chip */
```


```
#define ifeh  192  /* index of 0th hi E signal from front of east chip */

/* V785 ADC #6, 7 N568 AMP #6, 7 */
#define ifbl  224  /* index of 0th lo E signal from back of chips */
#define ifbh  256  /* index of 0th hi E signal from back of chips */

/* V785 ADC #8 N568 AMP #8, 9 */
#define iush  288  /* index of 0th hi E upstream signal */
#define iwdh  312  /* index of 0th hi E east wing signal */

/* V785 ADC #9 N568 AMP #10, 11 */
#define iptl  320  /* index of 0th lo E punchthru signal */

/* V785 ADC #10 */
#define igee  352  /* index of 0th Ge energy (E) signal */
#define iree  380  /* index of Rutherford east signal */
#define irwe  381  /* index of Rutherford west signal */
#define imae  382  /* index of MWAC signal */

/* V785 TDC */
#define iget  400  /* index of 0th Ge TDC signal */

#define iftt  384  /* TDC for front of FP top chip */
#define ifet  385  /* TDC for front of FP east chip */
#define ifwt  386  /* TDC for front of FP west chip */
#define ifbt  387  /* TDC for back of FP detector chips */
#define iust  388  /* TDC for upstream */
#define imat  389  /* TDC for MWPC anode */
#define iptt  390  /* TDC for punchthru detector */

/* These are four-letter mnemonics to identify the words in
  the data.  For the first group of 32, CAPITAL LETTERS denote the
  most significant sixteen bits of the 32-bit scaler, lower case
   is for the least significant 16 bits.  Numbers (XXX:) denote
  unused words.  */


#define taglist
  "000:", "usec", "USEC", "reqt", "REQT", "acct", "ACCT", "stmp",
  "STMP", "009:", "010:", "011:", "012:", "013:", "014:", "015:",
  "016:", "017:", "018:", "019:", "020:", "021:", "022:", "023:",
  "uswh", "USWH", "usbp", "USBP", "bit6", "V262", "err1", "err2",

  "tl00", "tl01", "tl02", "tl03", "tl04", "tl05", "tl06", "tl06",
  "tl08", "tl09", "tl10", "tl11", "tl12", "tl13", "tl14", "tl15",
  "tl16", "tl17", "tl18", "tl19", "tl20", "tl21", "tl22", "tl23",
  "tl24", "tl25", "tl26", "tl27", "tl28", "tl29", "tl30", "tl31",

  "th00", "th01", "th02", "th03", "th04", "th05", "th06", "th06",
  "th08", "th09", "th10", "th11", "th12", "th13", "th14", "th15",
  "th16", "th17", "th18", "th19", "th20", "th21", "th22", "th23",
```


```
"th24", "th25", "th26", "th27", "th28", "th29", "th30", "th31",

"wl00", "wl01", "wl02", "wl03", "wl04", "wl05", "wl06", "wl06",
"wl08", "wl09", "wl10", "wl11", "wl12", "wl13", "wl14", "wl15",
"wl16", "wl17", "wl18", "wl19", "wl20", "wl21", "wl22", "wl23",
"wl24", "wl25", "wl26", "wl27", "wl28", "wl29", "wl30", "wl31",

"wh00", "wh01", "wh02", "wh03", "wh04", "wh05", "wh06", "wh06",
"wh08", "wh09", "wh10", "wh11", "wh12", "wh13", "wh14", "wh15",
"wh16", "wh17", "wh18", "wh19", "wh20", "wh21", "wh22", "wh23",
"wh24", "wh25", "wh26", "wh27", "wh28", "wh29", "wh30", "wh31",

"el00", "el01", "el02", "el03", "el04", "el05", "el06", "el06",
"el08", "el09", "el10", "el11", "el12", "el13", "el14", "el15",
"el16", "el17", "el18", "el19", "el20", "el21", "el22", "el23",
"el24", "el25", "el26", "el27", "el28", "el29", "el30", "el31",

"eh00", "eh01", "eh02", "eh03", "eh04", "eh05", "eh06", "eh06",
"eh08", "eh09", "eh10", "eh11", "eh12", "eh13", "eh14", "eh15",
"eh16", "eh17", "eh18", "eh19", "eh20", "eh21", "eh22", "eh23",
"eh24", "eh25", "eh26", "eh27", "eh28", "eh29", "eh30", "eh31",

"bl00", "bl01", "bl02", "bl03", "bl04", "bl05", "bl06", "bl06",
"bl08", "bl09", "bl10", "bl11", "bl12", "bl13", "bl14", "bl15",
"bl16", "bl17", "bl18", "bl19", "bl20", "bl21", "bl22", "bl23",
"bl24", "bl25", "bl26", "bl27", "bl28", "bl29", "bl30", "bl31",

"bh00", "bh01", "bh02", "bh03", "bh04", "bh05", "bh06", "bh06",
"bh08", "bh09", "bh10", "bh11", "bh12", "bh13", "bh14", "bh15",
"bh16", "bh17", "bh18", "bh19", "bh20", "bh21", "bh22", "bh23",
"bh24", "bh25", "bh26", "bh27", "bh28", "bh29", "bh30", "bh31",

"uh00", "uh01", "uh02", "uh03", "uh04", "uh05", "uh06", "uh06",
"uh08", "uh09", "uh10", "uh11", "uh12", "uh13", "uh14", "uh15",
"uh16", "uh17", "uh18", "uh19", "uh20", "uh21", "uh22", "uh23",
"EW00", "EW01", "EW02", "EW03", "WW00", "WW01", "WW02", "WW03",

"pt00", "pt01", "pt02", "pt03", "pt04", "pt05", "pt06", "pt06",
"pt08", "pt09", "pt10", "pt11", "pt12", "pt13", "pt14", "pt15",
"pt16", "pt17", "pt18", "pt19", "pt20", "pt21", "pt22", "pt23",
"pt24", "pt25", "pt26", "pt27", "pt28", "pt29", "pt30", "pt31",

"ge00", "ge01", "ge02", "ge03", "ge04", "ge05", "ge06", "ge07",
"ge08", "ge09", "ge10", "ge11", "364:", "365:", "366:", "367:",
"368:", "369:", "370:", "371:", "372:", "373:", "374:", "375:",
"376:", "377:", "378:", "379:", "rute", "rutw", "mwae", "383:",

"fptt", "fpwt", "fpet", "fpbt", "_ust", "mwat", "390:", "391:",
"392:", "393:", "394:", "395:", "396:", "397:", "398:", "399:",
"gt00", "gt01", "gt02", "gt03", "gt04", "gt05", "gt06", "gt07",
"gt08", "gt09", "gt10", "gt11", "412:", "413:", "414:", "415:",
```

```
"rloe", "rhie", "ploe", "phie", "dloe", "dhie", "floe", "fhie",
"radt", "rfdt", "tf1l", "tf1h", "tf2l", "tf2h", "430:", "431:",
"pl00", "pl01", "pl02", "pl03", "pl04", "pl05", "pl06", "pl07",
"pl08", "pl09", "pl10", "pl11", "pl12", "pl13", "pl14", "pl15",
"pl16", "pl17", "pl18", "pl19", "pl20", "pl21", "pl22", "pl23",
"pl24", "pl25", "pl26", "pl27", "pl28", "pl29", "pl30", "pl31",
"shut", "upst", "anti", "mwpc", "fisc", "mfpe", "mupe", "bsiz",
"rell", "reul", "rwll", "rwul", "tsig", "psig", "pmul", "adrc"
```

Type 5: S800 Raw Data

This data type contains the raw S800 data buffers. Description of the complete S800 raw data structure is available on the S800 wiki at https://groups.nscl.msu.edu/opdevtech/wiki/index.php/S800_data_format.

Type 6: NSCL Non-Event Data

This data type contains NSCL non-event data structures, including beginning of run, end of run and scaler NSCL buffers, which are generated by the NSCL readout software. In general, the first two 32-bit integers of the payload for a type 6 event will describe the buffer length in bytes, and the buffer packet type, in a RingItemHeader, defined as:

```
typedef struct _RingItemHeader {
  uint32_t s_size;
  uint32_t s_type;
}
```

Details regarding the definition of non-event data structures can be found in the NSCL documentation available at: http://docs.nscl.msu.edu/daq/ringbuffer/r17361.html, while the definition of the possible non-event data types is available at: http://docs.nscl.msu.edu/daq/ringtutorial/a1889.html.

Type 7: GRETINA Scaler Data

This data type contains GRETINA digitizer-level scaler information. These scalers are currently in beta-testing, and will be implemented once testing is complete. The scaler data can be described by the following table of byte offsets and values:

Byte Offset	Size (in 4-byte words)	Description	Scaler Scope
0	1	Separator (0xbbbb1234)	Packet
4	1	Code version + board bits (0123)	Packet
8	1	Detector hole ID (user set)	Board3
12	2	Latched timestamp	Board3
20	1	Board serial number	Board3
24	1	Global DAQ board number	Board3
28	1	Main FPGA firmware revision	Board3
32	1	Main FPGA firmware subrevision	Board3
36	1	Main FPGA firmware build	Board3
40	10	Channel LED counters 0-9	Board3

continued on next page...

continued from previous page...

Byte Offset	Size (in 4-byte words)	Description	Scaler Scope
80	10	Channel packate counters 0-9	Board3
120	7	Front bus counters	Board3
148	10	Channel events (printCounters)	Board3
188	1	SERDES packet errors	Master
192	7	Master received counters	Master
220	7	Master issued counters	Master
248	40	Event size counters	IOC
408	1	Bad event size counter	IOC
412	6	Readout FIFO state counters	IOC
436	1	Detector hole ID (user set)	Board2
440	2	Latched timestamp	Board2
448	1	Board serial number	Board2
452	1	Global DAQ board number	Board2
456	1	Main FPGA firmware revision	Board2
460	1	Main FPGA firmware subrevision	Board2
464	1	Main FPGA firmware build	Board2
468	10	Channel LED counters 0-9	Board2
508	10	Channel packate counters 0-9	Board2
548	7	Front bus counters	Board2
576	10	Channel events (printCounters)	Board2
616	1	Detector hole ID (user set)	Board1
620	2	Latched timestamp	Board1
628	1	Board serial number	Board1
632	1	Global DAQ board number	Board1
636	1	Main FPGA firmware revision	Board1
640	1	Main FPGA firmware subrevision	Board1
644	1	Main FPGA firmware build	Board1
648	10	Channel LED counters 0-9	Board1
688	10	Channel packate counters 0-9	Board1
728	7	Front bus counters	Board1
756	10	Channel events (printCounters)	Board1
796	1	Detector hole ID (user set)	Board0
800	2	Latched timestamp	Board0
808	1	Board serial number	Board0
812	1	Global DAQ board number	Board0
816	1	Main FPGA firmware revision	Board0
820	1	Main FPGA firmware subrevision	Board0
824	1	Main FPGA firmware build	Board0
828	10	Channel LED counters 0-9	Board0
868	10	Channel packate counters 0-9	Board0
908	7	Front bus counters	Board0
936	10	Channel events (printCounters)	Board0
976	76	extra space at end of packet	-

Type 8: Bank 29 Raw GRETINA Data

This data type is structurally identical to type 2 (raw GRETINA data), but has been designated to identify exclusively data originating from Bank 29 in the GRETINA DAQ. This is a single digitizer which is used to digitize

signals from auxiliary systems which may be useful to improve the resolution of particle-gamma timing, for instance. A new type number was assigned to mark these data as being inappropriate for decomposition, and to funnel these data into the appropriate GRETINA output file.

Type 9: S800 Processed (Physics) Data

This data type contains post-processed S800 data, which contains the S800 parameters relevant to and required for particle identification (energy loss and time of flight information), and Doppler correction of GRETINA spectra (S800 positions and angles). This is the structure that can be used if GRETINA analysis and S800 analysis are treated somewhat independently, to be able to make the appropriate Doppler corrections and obtain the best possible GRETINA spectra. Data type was defined by D. Weisshaar (NSCL).

```
struct s800_physicsdata {
  int32_t type; /* defined as abcd1234 for indicating this (first) version */
  float crdc1_x; /* crdc x/y positions, in mm */
  float crdc1_y;
  float crdc2_x;
  float crdc2_y;
  float ic_sum; /* ion chamber energy loss */
  float tof_xfp; /* tof scintillator after the A1900 */
  float tof_obj; /* tof scintillator in object box */
  float rf; /* cyclotron rf for tof */
  int32_t trigger; /* trigger register bit pattern */
  /*-----------------------------------------------------*/
  /* from here corrected values, extracted from data above */
  /*-----------------------------------------------------*/
  float ic_de;
  /* tof values with corrections applied (from afp/crdc x) */
  float tof_xfpe1;
  float tof_obje1;
  float tof_rfe1;
  /* Trajectory information at target position calculated from
  a map and afp/bfp/xfp/yfp.  New map and you need to re-calc */
  float ata; /* dispersive angle */
  float bta; /* non-dispersive angle */
  float dta; /* dT/T T = kinetic energy */
  float yta; /* non-dispersive position */
};
```

Type 10: Timestamped NSCL Non-Event Data

This data type is identical in format to Type 6 (NSCL Non-Event Data), but was created to reflect a change in the way that NSCL non-event data is sent to the global event builder. Type 6 non-event data was assigned a recent timestamp once it arrived to the global event builder, which could result in trouble with respect to the ordering of scaler packets, etc. In response to this, within the NSCL data acquisition system a change was made to provide non-event data with a valid timestamp before being sent to the global event builder. This data type reflects this change, and should signify that the timestamp of these packets are generated by the NSCL DAQ, and not assigned in the GEB.

Type 11: GRETINA Geant4 Simulation Data

This data type is reserved for GRETINA simulation data as generated from the Geant4 simulation of the array. Description of this data format is pending. This data type is defined by Lew Riley (Ursinus College).

```
#define MAX_SIM_GAMMAS 10
struct g4Sim_emittedGamma {
  float e;
  float x, y, z;
  float phi, theta;
  float beta;
};
struct g4Sim_abcd1234 {
  int type /* Defined as 0×abcd1234 for this version */
  int num; /* Number of gammas */
  int full; /* Full energy deposited in detectors */
  g4Sim_emittedGamma[MAX_SIM_GAMMAS];
};
```

Type 12: CHICO Raw Data

This data type is reserved for raw data from the CHICO auxiliary detector. Details on the data format are pending. This data type is controlled by T. Lauritsen (ANL).

Type 13: Superstitious Mystery Data

As we don't want to press our luck, no real data will be assigned as type 13.

Type 14: Digital Gammasphere Data

This data type is reserved for data related to digital Gammasphere (DGS) and the GTReceiver3 at ANL. Details for this data type are pending. This data type is controlled by T. Lauritsen (ANL).

Type 15: Digital Gammasphere Trigger Data

This data type is reserved for data related to digital Gammasphere (DGS) trigger information. Details for this data type are pending. This data type is controlled by T. Lauritsen (ANL).

Type 16: Digital FMA (Fragment Mass Analyzer)

This data type is reserved for data related to the digital FMA. Details for the data type are pending. This data type is controlled by Darek Seweryniak (ANL).

Type 17: Phoswich Wall

This data type is reserved for data related to the Washington University Phoswich wall. Details for the data type are pending; this data type is controlled by the WashU group (D. Sarantites *et* al.).

Type 18: Phoswich Wall Auxiliary

This data type is reserved for auxiliary data related to the Washington University Phoswich wall. Details for the data type are pending; this data type is controlled by the WashU group (D. Sarantites *et* al.).

Type 19: GODDESS

This data type is reserved for the GODDESS (ARUBA) Silicon array. Details for this data type are pending; the definition for this type is controlled by the Rutgers/ORNL GODDESS collaboration.

Appendix: Sample GRETINA Mode3 (Raw Data) Unpacking Code

```
#define MAX_TRACE_LENGTH 2048

struct mode3DataPacket {
  unsigned short aahdr[2];
  unsigned short hdr[14];
  unsigned short waveform[MAX_TRACE_LENGTH];
};

struct GEBHeader {
  int32_t type;
  int32_t length; /* length of payload following the header, in bytes */
  int 64_t timestamp;
}

GEBHeader gHeader;
mode3DataPacket *dp;
unsigned char gBuf[32*32*1024] = 0;
unsigned char *tmp;
unsigned int mode3i = 0;
```

Included below is the code snippet using the structures/variables defined above to unpack GRETINA Mode3 data. This assumes a global header has been read into the `gHeader` structure, and will result in unpacking all GRETINA channel events following a single global header.

```
int siz = fread(gBuf, gHeader.length, 1, inputFile);
if (siz != 1) {
  cerr « "Error reading GRETINA data.  Aborting now...  " « endl;
  exit;
}
for (int j=0; j<gHeader.length; j++) { swap(*(gBuf + j), *(gBuf + j + 1)); }
mode3i = 0;
while( (int)(mode3i*2) < gHeader.length ) {
  tmp = (gBuf + mode3i*2);
  if !(dp = (mode3DataPacket*)malloc(sizeof(dp->aahdr) + sizeof(dp->hdr) +
               sizeof(dp->waveform))) ) { exit(-1); }
  memset(dp->waveform, 1, gMode3.tracelength*sizeof(unsigned short));
  memmove(&dp->aahdr[0], tmp, (sizeof(dp->hdr) + sizeof(dp->aahdr)));
  if ((dp->aahdr[0] != 43690) || (dp->aahdr[1] != 43690)) {
    cerr « "Didn't find 'AAAA' header as expected!" « endl;
  }
  mode3i += (sizeof(dp->aahdr) + sizeof(dp->hdr))/2;
  gMode3->tracelength = (((dp->hdr[0] & 0x7ff))*2) - 14;
  tmp = (gBuf + mode3i*2);
  memmove(&dp->waveform[0], tmp, gMode3->tracelength*sizeof(unsigned short));
  mode3i += (gMode3->tracelength*sizeof(unsigned short))/2;
  tmp = (gBuf + mode3i*2);
  /* Interpret the header information...  */
  gMode3->board_id = (dp->hdr[0] » 11);
  gMode3->chan_id = (dp->hdr[1] & 0xf);
  gMode3->module = (dp->hdr[1] » 4);
  if (gMode3->module/16 == Q1) {
    gMode3->id = (gMode3->module - (Q1-Q1E)*4*4)*10 + gMode3->chan_id;
```

```
} else if (gMode3->module/16 == Q2) {
  gMode3->id = (gMode3->module - (Q2-Q2E)*4*4)*10 + gMode3->chan_id;
} else if (gMode3->module/16 == Q3) {
  gMode3->id = (gMode3->module - (Q3-Q3E)*4*4)*10 + gMode3->chan_id;
} else if (gMode3->module/16 == Q4) {
  gMode3->id = (gMode3->module - (Q4-Q4E)*4*4)*10 + gMode3->chan_id;
} else if (gMode3->module/16 == Q5) {
  gMode3->id = (gMode3->module - (Q5-Q5E)*4*4)*10 + gMode3->chan_id;
} else if (gMode3->module/16 == Q6) {
  gMode3->id = (gMode3->module - (Q6-Q6E)*4*4)*10 + gMode3->chan_id;
} else if (gMode3->module/16 == Q7) {
  gMode3->id = (gMode3->module - (Q7-Q7E)*4*4)*10 + gMode3->chan_id;
} else {
  gMode3->id = (gMode3->module)*10 + gMode3->chan_id;
}
int hienergy = 0;
hienergy = (dp->hdr[7] & 0x00ff);
bool sign = 0;
sign = (dp->hdr[7] & 0x0100);
assert( (((hienergy) « 16) & (uint)(0xff000000)) == 0);
uint tmp_energy = 0;
int tmp_int_energy = 0;
tmp_energy = ((uint)(hienergy) « 16);
tmp_energy += dp->hdr[4];
assert((tmp_energy & 0xff000000) == 0);
tmp_int_energy = (int)(tmp_energy);
assert((tmp_int_energy & 0xff000000) == 0);
if (sign) {
  if ( (((int)(gMode3->id)%10) == 9) ) { /* CC */
    tmp_int_energy = (int)(tmp_int_energy - (int)0x01000000);
  } else {
    tmp_int_energy = (int)(tmp_int_energy - (int)0x01000000);
    tmp_int_energy = -(int)(tmp_int_energy);
  }
} else {
  if ( (((int)(gMode3->id)%10) == 9) ) { /* CC */
    tmp_int_energy = (int)(tmp_int_energy);
  } else {
    tmp_int_energy = -(int)(tmp_int_energy);
  }
}
gMode3->mask |= 0x0001;
gMode3->mask |= 0x0002;
gMode3->energy = (tmp_int_energy/32);
gMode3->mask |= 0x0004;
gMode3->cfd_timestamp = 0;
gMode3->led_timestamp = 0;
gMode3->cfd_timestamp = (ULong64_t)((ULong64_t)(dp->hdr[6]) +
              ((ULong64_t)(dp->hdr[9]) « 16) +
              ((ULong64_t)(dp->hdr[8]) « 32));
gMode3->mask |= 0x0008;
```


```
    gMode3->led_timestamp = (ULong64_t)((ULong64_t)(dp->hdr[3]) +
                  ((ULong64_t)(dp->hdr[2]) « 16) +
                  ((ULong64_t)(dp->hdr[5]) « 32));
    gMode3->waveform.clear();
    for (int j=0; j<gMode3->tracelength; j++) {
      if (dp->waveform[j] & 0x8000) {
        if (gMode3->id%10 == 9) {
          gMode3->waveform.push_back(dp->waveform[j] -
                  std::numeric_limits<unsigned int>::max());
        } else {
          gMode3->waveform.push_back((-(dp->waveform[j] -
                  std::numeric_limits<unsigned int>::max())));
        }
      } else {
        if (gMode3->id%10 == 9) {
          gMode3->waveform.push_back(dp->waveform[j]);
        } else {
          gMode3->waveform.push_back((-(dp->waveform[j])));
        }
      }
    }
    for (int j=0; j<(gMode3->tracelength+1); j+=2) {
      swap(gMode3->waveform[j], gMode3->waveform[j+1]);
    }
    free(dp);
  } /* End of while( (int)(mode3i*2) < gHeader.length ) */
```