DUMAND Archived Data Format Description DIR-10-92

Dennis J. Nicklaus
University of Wisconsin

Last Updated June 15, 1993

Contents

Change Log:

Changes since Aug. 31, 1992 version (please at least read these parts):

1. *Bit order in Interesting Interrupt word is changed.*
2. *Bit order in hit data word is changed (time and om number swapped).*
3. *Definition of wordcount field in interesting interrupt is changed.*
4. *Added ID field in Interesting Interrupt word.*
5. *The Unix tar command will be used to write and read the archive tapes.*
6. *In Section 5.2.6, the number of scalers is changed to reflect the fact that scalers are also kept for the CMs.*
7. *Addition of Starter and Terminator record types in Section 5.*
8. *The Starter Record contains a version number. All software creating event archive files should write a starter record containing the version number used.*
9. *The Starter Record also contains a run number. Efforts will be made to make run numbers continually increasing (until the 4byte number rolls over). Also, the log file names will be include the run number as part of the filename.*
10. *The chisq (chi squared) value in the Fitting Results Record will be recorded as chi squared times 100 in order to store it as an integer, not a floating point number.*
11. *Defined standard of Record Type constants – integers made from 4-letter sequences using 2-letter institution abbreviation.*
12. *Noted that provisions are made in the software to allow the event number to continually increase, even when the system is stopped and re-started.*
13. *Added an event type classification of 64=other (interesting but don't know what).*
14. *There are now 9bits for the partition address in the Microsecond Marker Word and only three for a "string number" identifier. The user is cautioned against using that 3bit identifier in general.*
15. *Added Appendix 1 describing the Raw Digitizer Output Format.*

16. *Note that a separate document, titled* <u>*Format of Data Words Passed From CCB to FLP*</u> *dated 11/30/1992 has been created. This format is of interest because that data may be saved into an archive file when a part or all of the trigger processor cache is saved.*

17. *The energy stored in the standard fit results structures now is now stored in GeV, not MeV to give it a more useful range.*

Changes between May 6 and Aug. 31 version:

1. *Added ASCII standard format (last section of the document).*
2. *Files will not be compressed before writing to tape.*
3. *Left open the exact format of how files are written to tape. This will be specified in a later version.*
4. *The "Trigger Reason" field of the event record is changed to be a bitmask of all applicable triggers and is only one unsigned integer now.*
5. *The "Total Hits" field of the event record is changed to indicate the total number of hits in the whole event, not just the triggered microsecond.*
6. *For "TOY and Scaler" save records (saved by the trigger processor once per second), the "Total Hits" field will contain the total number of long-ons in the last second (and contained in this record), and the "Total Energy" field of the event record will contain the total number of errors in the last second.*
7. *Added ability to add additonal data in the form of "tail structures" after the hit data in Event Data Records. Section 5.2.13 defines the formats of these tail structures.*
8. *Proposed more details for the record format of the fitting record (Section 5.8).*
9. *Note that the first implementation will only use 8 bits for the energy of a hit, and two more bits are reserved in the hit word for later extension to 10 bits.*

Changes between March 5 and May 6 version:

1. *Included the omonword in the event data stream — this word indicates which OMs were on (Long-on's) at the end of the microsecond.*
2. *Bit 0 (zero) of the interesting interrupt word is now unused. (It was formerly used to indicate a "high hit count First Level Trigger" which is no longer a trigger.)*
3. *The "energy" stored in each hit word is now ten bits wide.*

4. Added explanatory section at the end of the document.

5. added an event number in the event data structure. This event number is increased by one with every event saved.

1 Purpose

This document describes in detail the format of all the data files which the DUMAND software creates. There are several goals of this document and the formats contained herein:

1. To provide one common format for data exchange. Future Monte Carlo, track-fitting, or event display programs for DUMAND data should use this format for reading or writing data.
2. To document the format in case someone wants to write a translator to copy DUMAND data into another database format such as Zebra.
3. To solicit ideas on how the data storage can be done more efficiently and correctly.
4. To provide a robust system which will be able to efficiently handle all the types of DUMAND data which will be saved.

This archived data includes the following:

- The event data stream, including event data, noise hits, scalers, and GPS Time-of-Year (TOY) one second ticks.
- Positional Information computed by the Environmental Computer
- Other environmental conditions reported
- Operator and Error Log
- Trigger parameters
- Configuration and data format parameters (e.g. how many strings, how many trigger parameters in the trigger parameter section,...)
- Fitting Results
- Calibrations Applied
- Log of all communications from SCC serial lines
- Cross-reference "bookmarks"
- Operator-inserted data (the kitchen sink)

It was deemed necessary to create a new format for DUMAND data in order to efficiently store the type of data available, keeping in mind the nature of the event data stream from the trigger processor.

2 Applicable Documents

- Dumand System Description, J. Learned, Feb.,1992
- DUMAND Shore Host Computer Software Requirements, D. Nicklaus, Nov. 1991
- DUMAND Trigger Software Design, D. Nicklaus, Updated May, 1992

3 Definitions

In this document, several C language or C-like definitions will be used to document certain structure formats. For these purposes, a *short* integer is 16 bits (2 bytes). A character takes one byte. All other words, integers and long integers are 4 bytes. A *float* (floating point number) is 4 bytes and a *double* (double precision floating point) is 8 bytes.

All numbers stored in standard Sun format. For floating point numbers, this means IEEE std. floating point (although I do not yet store any floating point numbers). For integers, it means you get the most significant bit (and byte) first, and the least significant last. In the value 12345678 hex, the first byte is the most significant and has value 12 hex, the last, least significant byte has value 78 hex, and this is the way it is stored in the file. None of this wacky 78563412 kind of thing which VAXes like to do. Translators from Sun to Vax format are easily available. I note this format here because this document describes several data words which are made of bit patterns which I always list in the Sun standard format.

Where structures or constant value definitions are included in this document, the user is reminded that the most recent definitions should always be taken from the appropriate source code files before implementing in a program.

4 Tape Format

In this and the following sections, I will attempt to describe the archive formats in a top-down manner: first indicating what is on a tape, then indicating what kinds of records each tape file can contain, then finally the format of the records and the words within the records.

There are three files of interest which will be routinely written to the archive tape: the Event Data file, the Operator and Error Log, and the SCC Log. The SCC Log may eventually be left out of the archive tape, but for now, I'll plan on it being included. Other files may also be included (e.g. files from the Environmental Computer).

These 3 files are all written independently to disk. Periodically during DUMAND operation (e.g. once per hour and at a shutdown point), all the files will be closed and copied to tape. At this point, new files will be opened on disk for new data or log messages.

The archive tapes will be written with a 5GB 8mm tape drive. In the end, if we decide we want the tapes written at 2.3GB density, the 5GB drives can be made to write 2.3G density, but by that time, 5GB will probably be the standard density used.

The tapes will be written using the unix *tar* (*T*ape *AR*chive) command to put a set of files onto the tape as a group. Tar is a nice interface which would use the tape fairly efficiently. You will have to use tar to extract onto disk any files you want to examine before you can use them. There are VMS versions of tar available in the public domain and Bob Svoboda has agreed to write a program to read the archive tapes on a VMS system.

The filenames will contain the date and time of creation, so it will be straightforward to scan the tape for archive files from a specific day or hour.

The rest of this document will discuss the format of disk files which are written to the tapes with tar. These files will be used to interchange DUMAND data.

5 Collection File Format

The Collection File is the primary DUMAND archive file. It will contain multiple variable length records and is designed to be accessed sequentially.

The record types it may contain include:

1. Starter Record: Indicates start and stop date/time of opening of file, a run number, and a software version number.

2. Terminator Record: Indicates start and stop date/time of closing of file. The presence of this record indicates that the archive file was closed cleanly and normally.

3. File Header: May be used to record items such as version number, size of header in the event records, and other descriptions of the file.

4. Event Data: Hit data saved by trigger software, may include results of the built-in DUMAND fitting program

5. Scaler Data: TOY synchronization and scaler hit counts saved by trigger software

6. Trigger Parameters: This may include array configuration data (e.g. number of active strings) as well trigger software parameters such as energy thresholds, etc.

7. Monte Carlo Data: Simulated hit data, in the same format as Event Data

8. Position Information: Describes positions of the strings as calculated by the Environmental Computer
9. Environmental Information: Environmental data collected from the array such as NBU, compass, accelerometer, and tiltmeter readings
10. Fitting Results: Results of any analysis program in a standard format
11. "Bookmarks": A bookmark enables some type of cross-referencing to be accomplished between the various files
12. Calibration Record: Contains results of the analysis of the data obtained after firing the calibration modules. There may be more than one type of calibration record required to account for different types of calibrations
13. User Record: anything the user desires may be placed in this record, but it must adhere to the standard record format as described below
14. User Text: similar to a user record but specifically designed to hold only ascii text

The format of every record is as follows:

```
int recordType;     /* indicates what type of record this is */
int HowManyBytes;  /* how many bytes follow (not incl. this word) */
char data[HowManyBytes]; /* the actual data itself */
```

The actual data will contain varying types of information, depending on the record type. In general, the first word of the data should contain the time in the format returned by the Unix gettimeofday() call. This function returns the number of seconds since Jan. 1, 1970, GMT. An exception to this rule is any of the event or scaler data. In these cases, a more complete time from the GPS clock is recorded. (See the description of the Event Data Record format for more details.)

A Collection File is meant to be processed sequentially. This means, once a Trigger Parameters record is read, those trigger parameters apply to all the event data records which follow until a new Trigger Parameters record is read. Similarly, when a Positions record is encountered, those positions listed in it will be the most recent positions computed (for use in fitting subsequent data) until a new Positions record is seen.

The DUMAND host software will start each new Collection File with a Starter record, a File Header record, a Trigger Parameters record, a Environment record, and a Positions record. It may be that the environment or positions data is not available (hasn't been computed yet) at start time. In this case a record filled with default values or flags which indicate the values are not known will be written.

When the host Data Harvest program starts a new collection file, it will include a Starter record and a version number in the Starter record. Using this version number, software which reads the archive file can make sure it was built using the same versions of the relevant include files. The version number is called VERSION_RECTYPES and is in the file rectypes.h. Likewise, any other software which creates an archive file must include a starter record with this version number filled in. (This again ensures that any software reading the file can know the writer and the reader used the same include file versions.)

5.1 Record Type Constants

The recordType of each archive record is an integer flag selected from the file rectypes.h. The contents of rectypes.h are listed below:

```
/* DR stands for DUMAND Record */
#define DR_EVENT   'UEVT'
#define DR_SCALERS 'USCA'
#define DR_PARAMS  'UPRM'
#define DR_HEADER  'UHDR'
#define DR_MONTECARLO 'UMCO'
#define DR_POSITIONS 'UPOS'
#define DR_ENVIRONMENT 'UENV'
#define DR_FITRESULTS 'UFIT'
#define DR_BOOKMARK    'UBMK'
#define DR_CALIBRATE  'UCAL'
#define DR_USERTEXT    'UUTX'
#define DR_USERDATA    'UUDA'
#define DR_TERMINATOR 'UTRM'
#define DR_STARTER     'USTA'
```

Note that each of these integer constants is actually created by using the number created by 4 ASCII characters. For instance, the number represented by 'UEVT' is hex 55455654, or decimal 143060744. This is done with the hope of making it easier to visually scan parts of a binary archive file.

In processing a DUMAND archive file, the experimentor may want to create his own unique records and add them to the file. When doing so, the experimentor must create their own constant by taking their institution's 2-letter initial, adding a third character which should be reserved for that individual by agreement within his institution (for instance his initial), and then adding a fourth character. That user is solely responsible for unique allocation of the fourth character and documenting any meaning behind that letter.

This assumes we're all relatively intelligent on name collision avoidance – if there is a John, a Joan, and a Jill at your institution, one of them takes the J, the other takes K and the other takes L to use as their 3rd-letter initial.

The letter 'U' (as in Universal) is always chosen as the first letter of all the standard constants (those I have defined in rectypes.h). Thus, no institute can start their 2-letter id sequence with that letter. 'U' is chosen because using 'U' as in University for your abbreviation doesn't add any information anyway. Below is a *suggested* list of 2-letter institution abbreviations:

Aachen	AA
Bern	BE
Boston	BO
Hawaii	HI
KEK	KE
Kiel	KI
Iowa State	IS
LSU	LS
Scripps	SC
Tokyo	TY
Tohuku	TH
Vanderbilt	VB
Washington	WA
Wisconsin	WI

This type of constant should be used for the RecordType word in the file records and for value of the tail_marker word in tail structures (discussed in the next section).

5.2 Event Data Records

Event Data Records store all the PMT hit data from all the tubes (OMs and CMs) for a small time window. For triggered events, this time window will be 5us, centered on the microsecond which triggered the save (i.e. E-2 through E+2, where E is the microsecond containing the event). The length (in microseconds) of this window will be one of the parameters in the Trigger Parameters record, so if a MC program wishes to record a shorter time window, it may write the time window length with a Trigger Parameters Record, then write the data records.

An Event Data Record may contain:

```
int recordType;          /* indicates what type of record this is */
int HowManyBytes;        /* how many bytes follow, incl. end_marker */
int DataBytes;           /* How many bytes of event data */
data2host_struct event;  /* structure containing hit data*/
int end_marker;          /* indicates the end of this record */
```

Alternately, various "tail structures" may be appended at the end of the event data but before the end_marker, e.g.:

```
int recordType;          /* indicates what type of record this is */
int HowManyBytes;        /* how many bytes follow,
                            including tail structures and markers */
int DataBytes;           /* points to the end of the data -- the start
```

```
                               of the first tail_marker in this case. */
data2host_struct event; /* structure containing hit data*/
int tail_marker         /* tells the type of this tail structure  */
struct whatever         /* any data structure the user inserts here */
int tail_marker2;
struct whatever2;
int end_marker;         /* indicates the end of this record */
```

With this format, we might have an event record as follows: (showing the values of each word in the structure where possible)
1 – Constant indicating this is an event record
200 – 200 more bytes follow, so the record is 208 bytes long
192 – how many data bytes to skip to get to end_marker
(here would be 192 bytes of event header and data)
1999 – Special constant defined for end_marker.

Alternately, if there were a tail structure appended:
1 – Constant indicating this is an event record
256 – 256 more bytes follow, so the record is 264 bytes long
192 – how many data bytes to skip to get to the first tail_marker
(here would be 192 bytes of event header and data)
427 – my special tail_marker define.
(here would be 52 bytes of this tail structure)
0x5545454d – Special constant defined for end_marker.

Note that HowManyBytes is the total bytes in the record minus 8. Minus 8 because it doesn't count the 4 bytes in the recordType and it doesn't count itself. (This is designed to be useful in using the 'C' language functions fseek() or lseek().)

The format of the data2host_struct structure (containing the event hit data) is found in the C include file passhost.h, which is shown below. This structure follows the `RecordType` and `HowManyBytes` words in the event record.

```
/* File: passhost.h           */

typedef struct {
  unsigned long toy_marker[4]; /* two words for GPS, 2 for DUMAND */
  long eventnumber; /* increases one with every event stored */
  unsigned int trigger_reason; /*trigger reason or transfer type*/
  int total_hits; /* total hits in the triggered us, not the neighborhood*/
  int total_en; /* total energy in the triggered us, not the neighborhood*/
  int microsec_time; /* 20bit time (precise usec up to 1sec)
                      of  the triggered us */
  unsigned long string_data[(PARTITION_WORDSIZE+1)*WINDOW_SAVED*MAXSTRINGS
                            + WINDOW_SAVED + 1];
  /* contains up to 5 usec's for up to 9 strings */
  /* in this goes a string number and a string partition for
     each string for each partition.  This is densely packed,
     so we don't copy the unfilled part of any string's partition.
     The final +WINDOW_SAVED is so we know we have an
     end marker each usec.
     The final +1 is for an "End of Event" marker flag.*/
} data2host_struct;
/* the same structure is used for the TOY data, but less of the
   string_data memory is needed because we only do 1 usec, but
   note that for each string, we use:
   24 words for scalers
   MAX_LONGONS_PER_SEC+1 for longons
   MAX_ERRORS_PER_SEC+1 for errors.
   Which is currently 154 which is < than 5 (Window_saved) * 65 (NP+1). */
```

5.2.1 TOY Marker

The first two words of this field contain the GPS time-of-year and the last two words contain the DUMAND TOY. Both these TOY's are the TOY which apply to the central microsecond of the triggered event (the 3rd of the 5 microseconds). Therefore, it is possible that an event record will indicate, for example, 2000002.000000 as the TOY, but (from examining the event data usechdr words), the microseconds stored are actually 2000001.999998, 2000001.999999 2000002.000000, 2000002.000001, and 2000002.000002 if the event is triggered by the zero-th microsecond. While slightly confusing at first glance, all the necessary information is stored in the event record.

The TOY values contain the full precision of the two clocks so that analysis programs may accurately correct DUMAND event times to GPS time.

5.2.2 Trigger Reason

This field consists of one unsigned integer. This acts as a bitmask to indicate which triggers were used to save this data event. Various triggers are defined in the file *trigmask.h*. Besides trigger processor triggers, constants are also defined for events saved and created by MC programs or randomly generated/saved events.

5.2.3 Total Hits

This indicates the total number of hits across all strings in the event record.

For "TOY and Scaler" save records (saved by the trigger processor once per second), the "Total Hits" field will contain the total number of long-ons in the last second (and contained in this record)

5.2.4 Total Energy

This indicates the total energy across all strings in the triggered microsecond (the 3rd of the 5). This may not always be available (some trigger conditions won't bother to compute it). In those cases, it will be set to 0.

For "TOY and Scaler" save records (saved by the trigger processor once per second), the "Total Energy" field will contain the total number of errors in the last second (and contained in this record)

5.2.5 String Data

The string_data part of the structure contains the actual event hits, along with headers for the microsecond. These are taken directly from the event data stored in VME partitioned memory by the Trigger First Level Processor.

String_data contains data in the following format for each microsecond saved (typically 5us, indicated by WINDOW_SAVED):

```
int stringnum;          /* string geometry number of string1 */
unsigned int intint;    /* interesting interrupt string1, this us */
unsigned int usechdr;  /* microsecond header for string1, this us */
unsigned int hitdata[N];/* up to a max N of 61 words -- the number of
                           hits here is told in the intint */
unsigned int omonword  /* OM-on word -- indicates which OM's were on
                            (long-on) at the end of the microsecond */
```

The above four items (which can be up to a maximum of 65 words) are then repeated for each string which had any hits during that microsecond. Note that omonword follows immediately after the hitdata, dependent on the number of hits there are. (There are no blank hit words filling in to 61 words to make the omonword fall in the 62nd position.)

For one microsecond, after all the strings' hit data is included, the next stringnum will be a -1 (negative one) which indicates that there are no more strings with any data for that microsecond.

The above pattern is repeated for each of the 5 microseconds saved. A few examples may help. These examples will show the contents of one microsecond, one string on a line of text. Values of the form intintN will be used for interesting interrupts to show they contain the number N indicating how many hits that string had that microsecond.

Here is the string_data section for a 5 microsecond window which had no hits at all. The total size of the string_data section of this would only be five words.

```
-1
-1
-1
-1
-1
```

This example below shows an event where 3 strings recorded hits in the first microsecond, 4 strings in the second microsecond, 5 strings in the third, 5 strings in the 4th, and 4 strings in the 5th.

```
1 intint3 usechdr hit1 hit2 hit3 omonword
2 intint2 usechdr hit1 hit2 omonword
9 intint5 usechdr hit1 hit2 hit3 hit4 hit5 omonword
-1
2 intint2 usechdr hit1 hit2 omonword
4 intint2 usechdr hit1 hit2 omonword
5 intint1 usechdr hit1 hit2 omonword
7 intint3 usechdr hit1 hit2 hit3 omonword
-1
1 intint4 usechdr hit1 hit2 hit3 hit4 omonword
3 intint5 usechdr hit1 hit2 hit3 hit4 hit5 omonword
4 intint3 usechdr hit1 hit2 hit3 omonword
5 intint4 usechdr hit1 hit2 hit3 hit4 omonword
```

```
9 intint5 usechdr hit1 hit2 hit3 hit4 hit5 omonword
-1
1 intint2 usechdr hit1 hit2 omonword
3 intint1 usechdr hit1 omonword
5 intint2 usechdr hit1 hit2 omonword
9 intint3 usechdr hit1 hit2 hit3 omonword
7 intint1 usechdr hit1 hit2 omonword
-1
2 intint2 usechdr hit1 hit2 omonword
3 intint1 usechdr hit1 omonword
4 intint2 usechdr hit1 hit2 omonword
8 intint1 usechdr hit1 hit2 omonword
-1
```

5.2.6 String Data in Scaler/TOY Records

When the first short int of the collect reason indicates this was a TOY collection, the format of the string_data is substantially different. For TOY collections, string_data contains data in the following format:

```
int stringnum; /* string geometry number of the string */
short highpe_scalers[26]; /* one for each OM and two CMs */
short lowpe_scalers[26];  /* one for each OM and two CMs */

int numlongons;
one unsigned int for each longon in the past second on this
string up to a maximum of 64.

int numerrors;
one unsigned int for each error in the past second on this
string up to a maximum of 64.
```

This pattern is repeated for each string. A -1 is written after the last string's data to indicate that there are no more strings. For a three string example, the TOY string_data might look like this:

```
1
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
0
0
2
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
3 longon1 longon2 longon3
```

```
4 error1 error1 error3 error4
2
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
3 longon1 longon2 longon3
4 error1 error2 error3 error4
3
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
0
2 error1 error2
-1
```

5.2.7 Interesting Interrupt

The Interesting Interrupt is the first of two header words written For each microsecond of data separately on each string. The Interesting Interrupt is used primarily as an input to the Trigger Refinement software (as opposed to being useful to any post-acquisition analysis programs).

The bits indicating the wordcount (number of hits+2 this microsecond) are the primary piece of this word which will be useful for post-processing In the case of simulated data, as much as possible of this word should be filled in order that the simulated data can be used as test inputs to the trigger refinement software.

The format of this word is as follows:

```
Bit     Contains
31      First Bit of 3bit ID field
30      First Bit of 3bit ID field
29      First Bit of 3bit ID field

28      TOY Set Indicator
27  Scaler Update Indicator Flag (1=this interrupt means
        process the new TOY and scaler values written in the
reserved partition)

26 Interesting because of Error indicator
25 Interesting because of Calibrate indicator
24 Interesting because of T3 indicator
23 Interesting because of T2 indicator
22  Interesting because of LongOn indicator

21 wordcount this string, this us MSB
20 wordcount
19 wordcount
```

```
18 wordcount
17 wordcount
16 wordcount LSB

15  Interesting Processing Done Fla g

14 This us, s9 interesting OR
13 This us, s8 interesting OR
12 This us, s7 interesting OR
11 This us, s6 interesting OR
10 This us, s5 interesting OR
9 This us, s4 interesting OR
8  This us, s3 interesting OR
7 This us, s2 interesting OR
6 This us, s1 interesting OR

5 Number of strings with any interesting flag last us MSB
4 Number of strings with any interesting flag last us
3 Number of strings with any interesting flag last us
2 Number of strings with any interesting flag last us LSB

1 OR flag set if any string had a T3 in the PREVIOUS us
0 OR flag set if any OTHER string has a T3 THIS us
```

The 3bit ID field will always have all three bits set to 1 for an interesting interrupt.

There are 6 bits for the wordcount, allowing a wordcount between 0 and 63. This indicates how many hit words were collected by this string (FLP) for the microsecond in question. This field contains the actual number of hits plus 2, to count the header words also. Thus if there are 8 hit words, then the wordcount field will have a value of 10 (8 hits and 2 header words). Similarly a wordcount field value of 11 means 9 hits, 12 means 10 hits, etc.

The "interesting because of" bits get set when the FLP identifies the corresponding interesting condition anywhere in the usec for that string.

The FLP always sets the Interesting Processing Done Flag to zero when it writes the usec header word. This flag is not used in the current scheme, but is reserved in case it is needed at a later stage in the design. (In a previous version of the design, Trigger Refinement set the flag to one when Trigger Refinement is finished with its characterization of that usec for that string to indicate that Trigger Refinement finished.)

The Interesting Interrupt includes a 9-bit bitmask indicating which of the strings were interesting for any reason at all. This is useful because it can be passed on to the Gate Keeper to tell the Gate Keeper which strings were interesting for a usec and so which strings it must to look at.

There are two (one each for the current and preceding usecs) 1-bit indicators of whether any string in the array had a T3. This helps Trigger Refinement decide whether to start the Gate Keeper or not. This T3 bit for the current microsecond (bit 24) is only set if a different string (other than the string sending the interesting interrupt to the DSP) had a T3.

The field containing the 4bit count of "Number of strings with any interesting flag last us" is a count of how many bits were set in the 9bit bitmask for the previous usec. The FLP provides Trigger Refinement with information about the previous microsecond to help detect events which fall across two microseconds. A T3 by itself may not be sufficient, but a T3 with a T3 in the previous microsecond may be. Thus when the DSP sees the T3 with a previous T3, it may notify the Gate Keeper. In this way, the Gate Keeper only has to look into the past usecs, not also the next future usec to look for cross-usec events. It doesn't have to look into the future because it knows the future usec will always wake it up if it needs to.

The Scaler Update Indicator Flag bit is set when the FLP is telling Trigger Refinement to harvest the new scaler values. Otherwise it is 0.

The TOY Set Indicator tells Trigger Refinement which of the TOY word sets was updated by the most recent one-second GPS time strobe (so it indicates which TOY set is valid for this microsecond).

5.2.8 Microsecond Header

The microsecond header contains fields which are mostly only relevant to the trigger processing software or post-processing software which cares about which microsecond within a second is indicated. Any post-acquisition analysis software should not use the string number in this header word — use the string number in the event data which separates one string from the next.

The 3bit string number in the Microsecond Header is just an extra identifier which may be used for lower level testing. It's purpose may change in future versions of the software system. In particular, the value of 1,1,1 for this string number field is not allowed since that would conflict with the interesting interrupt ID field.

If simulated data is intended as test inputs to the trigger software, all these fields must be filled in coherently.

```
Bit     Contains

31      String number 3bit number MSB
30      String number 3bit number
29      String number 3bit number LSB

28      Address MSB -- 9 bits go from 0 to 511.
27 Address
26 Address
25 Address
24 Address
```

23	Address
22	Address
21	Address
20	Address LSB
19	MSB of 20 bit slow time (accurate to the microsecond)
18	20 bit slow time
17	20 bit slow time
16	20 bit slow time
15	20 bit slow time
14	20 bit slow time
13	20 bit slow time
12	20 bit slow time
11	20 bit slow time
10	20 bit slow time
9	20 bit slow time
8	20 bit slow time
7	20 bit slow time
6	20 bit slow time
5	20 bit slow time
4	20 bit slow time
3	20 bit slow time
2	20 bit slow time
1	20 bit slow time
0	LSB of 20 bit slow time

There are 9 bits for the address, allowing an address between 0 and 511. This specifies which partition contains the microsecond which generated this trigger.

The 20bit time is accurate to the microsecond, so it counts up to approximately one second.

5.2.9 Hit Data Word

Each hit in the Sorted Hit Data has the following format:

Bit	Contains
31	OM or CM 5bit number
30	OM or CM 5bit number
29	OM or CM 5bit number
28	OM or CM 5bit number
27	OM or CM 5bit number
26	MSB of 10 bit fast time (accurate to the nanosecond)
25	10 bit fast time
24	10 bit fast time

23	10 bit fast time	
22	10 bit fast time	
21	10 bit fast time	
20	10 bit fast time	
19	10 bit fast time	
18	10 bit fast time	
17	LSB of 10 bit fast time	
16	Error flag	
15	Error flag	
14	Error flag	
13	T3 indicator	(set if this hit is end of a T3)
12	T2 indicator	(set if this hit is end of a T2)
11	Skip indicator	(set if the T2 is T2SKIP, or if T3 is T3SKIP)
10	Long On indicator	(set if this hit is a Long-On)
9	Reserved for future extension to 10bit energy	
8	Reserved for future extension to 10bit energy	
7	MSB of 8bit Energy	
6	Part of 8bit Energy	
5	Part of 8bit Energy	
4	Part of 8bit Energy	
3	Part of 8bit Energy	
2	Part of 8bit Energy	
1	Part of 8bit Energy	
0	LSB of 8bit Energy	

The T3s recognized by the FLP are simple time coincidences. They do not use a "difference of differences" scheme.

The errors here are used to flag digitization errors such as queue overflows.

The 10bit fast time in the hit word begins where the microsecond header word time leaves off. Thus, since the microsecond header word is accurate to a microsecond, the most significant bit of this 10bit time indicates 512 nanoseconds. The least significant time bit indicates the time to the nanosecond.

See Section 8.5 for a discussion of the units of energy stored for each hit. Remember that this word defined here is the word stored in the binary data archive file to contain information about the hit.

5.2.10 OM-on Word

Following the hitdata for each string is one 32bit word that indicates which OM's were still on at the end of the microsecond. This indicates which OMs were in a Long-On state. This least significant 24 bits of this word each correspond to one OM. If the OM was in a Long-On state, then the corresponding bit is set to 1.

5.2.11 Error Word

Whenever the trigger software sees a hit word flagged as an error, it is collected into an error buffer which is collected and stored with the scalers upon the GPS once-per-second pulse. Each error word saved in a second has the following format:

Bit	Contains
31	OM number 5bit number MSB
30	OM number 5bit number
29	OM number 5bit number
28	OM number 5bit number
27	OM number 5bit number LSB
26	Hit Error Bits
25	Hit Error Bits
24	Hit Error Bits
23	unused
22	unused
21	unused
20	unused
19	MSB of 20 bit slow time (accurate to the microsecond)
18	20 bit slow time
17	20 bit slow time
16	20 bit slow time
15	20 bit slow time
14	20 bit slow time
13	20 bit slow time
12	20 bit slow time
11	20 bit slow time
10	20 bit slow time
9	20 bit slow time
8	20 bit slow time
7	20 bit slow time
6	20 bit slow time
5	20 bit slow time
4	20 bit slow time
3	20 bit slow time
2	20 bit slow time
1	20 bit slow time
0	LSB of 20 bit slow time

Note that the time here is only accurate to the microsecond (within one second).

5.2.12 Longon Word

Whenever the trigger software sees a hit word flagged as a longon, it is collected into a longon buffer which is collected and stored with the scalers upon the GPS once-per-second pulse. Each longon word saved in a second has the following format:

Bit	Contains
31	`OM number 5bit number MSB`
30	`OM number 5bit number`
29	`OM number 5bit number`
28	`OM number 5bit number`
27	`OM number 5bit number LSB`
26	`MSB of 20 bit slow time  (accurate to the microsecond)`
25	`20 bit slow time`
24	`20 bit slow time`
23	`20 bit slow time`
22	`20 bit slow time`
21	`20 bit slow time`
20	`20 bit slow time`
19	`20 bit slow time`
18	`20 bit slow time`
17	`20 bit slow time`
16	`20 bit slow time`
15	`20 bit slow time`
14	`20 bit slow time`
13	`20 bit slow time`
12	`20 bit slow time`
11	`20 bit slow time`
10	`20 bit slow time`
9	`20 bit slow time`
8	`20 bit slow time`
7	`LSB of 20 bit slow time`
6	`MSB of 7 bit fast time`
5	`7 bit fast time`
4	`7 bit fast time`
3	`7 bit fast time`
2	`7 bit fast time`
1	`7 bit fast time`
0	`LSB of 7 bit fast time`

A 27bit time is recorded using the 20bit time from the usec header and the 7 most significant bits of the hit time. This gives time accurate to 8 ns within one second. (Recall that these are always stored in a TOY record to give a complete time.)

5.2.13 Tail Structures

This section explains the format of the tail structures which may be appended to the end of the event data in an Event Data Record. When a tail structure is tucked onto the end of the event data, a tail_marker word must be written. After that, the software should write an integer indicating how many more bytes remain in that tail structure before the next tail_marker or end_marker will be found. The tail_marker word should indicate to the reader what type of data is contained in the tail structure. For instance, we would assign value of 200 to mean Dick's fit data, 201 to indicate Jane's fitter output, and 202 to indicate Spot's comments follow, etc.

One special value of a tail_marker word, defined as TAILSTRUCT_STD_ONLINE_FIT in *rectypes.h* is used to indicate that the structure contains the fitting parameters arrived at by the standard on-line fitting routine filled in these values. Since this is a standard structure defined (in *rectypes.h*), it does not contain a bytecount word. (Other structures which might be defined and added by any user anywhere must contain the bytecount because someone else who didn't know what structure they used would be unable to know how much data was contained.)

The standard on-line completed fit structure is as follows:

```
typedef struct {
  int type;     /* muon track, cascade, SN, noise ... */
  int x,y,z;            /* distance from origin in mm */
  int xdir,ydir,zdir; /* direction of muon track
                         in direction cosines times 1000000 */
  int energy;           /* Event energy in GeV */
  int time;             /* time (difference) before event in ns  */
  int chisq;            /* chi squared indicator of fit success */
} std_event_fit_tailrecord;
```

Note that the directions are in 10^6 times direction cosines (unit vectors). The value 1000000 means cosine of 1.0 or 0 (zero) degrees, 500000 is 60 degrees, and 707107 is 45 degrees. This allows integers to be stored instead of floating point numbers (as in cosines). I could easily store floating point numbers there, but individual users would have to be responsible for the conversion from Sun floating point to the floating point representation of (insert your least favorite computer company here).

5.3 TOY Scaler Records

The format of the TOY Scaler Records will be similar to that of the Event Data records except that the string_data field will be different, as noted in Section 5.2.6.

5.4 Trigger Parameter Records

Format is to be determined at a later date.

5.5 Monte Carlo Data Records

The data format should be identical to that of Event Data Records.

5.6 Position Information Records

Format is to be determined at a later date, but probably will be similar to the format in which the Environmental Computer passes back the position information.

5.7 Environmental Information Records

Format is to be determined at a later date, but probably will be similar to the format in which the environmental data is passed to the Environmental Computer.

5.8 Fitting Results Records

After the recortType and number_of_bytes indicators, a structure to contain fitting results might look something like this.

```
typedef struct {
  int fitter_id;      /* Identify which fitter made these results */
  int event_number;   /* event number this fit corresponds to */
  int time_of_year;   /* time when this fit was completed (seconds) */
  int type;     /* muon track, cascade, SN, noise ... */
  int x,y,z;          /* distance from origin in mm */
  int xdir,ydir,zdir; /* muon track direction (10**6 times cosine) */
  int energy;         /* total event energy in GeV */
  int time;           /* time (difference) before event in ns  */
  int chisq;          /* chi squared indicator of fit success (times 100)
This is in units of chisq*100 to make it an integer.*/
} gen_purpose_fit_record;
```

This record may be slightly redundant with the fit tail structures in an event record. However, one may envision a situation where the on-line fitter takes a relatively long time to complete. In this case, we may write out the event record and start the fit. While the fit is running, several more events may arrive to be written out. then when the fit completes, a fit record is then added at the end of the archive file.

5.9 Bookmarks Records

The purpose of these bookmark records is to provide a convenient cross-reference to entries in the Error Log or SCC Log which were recorded at the same time. This is accomplished by recording the byte offsets into the other files which are required to view entries from the same approximate time. For instance, if the operator sees some unusual condition in an event and the error log from the same time, the operator might place a bookmark at that time. Then, in data analysis, when the analysis program encounters the bookmark in the Collection data stream, it can look-up the byte offset for the Error Log, open the corresponding error log file, skip to the required byte offset, and display the text therein. The text at the bookmark will be the message the operator inserted as part of the bookmark.

After the record-type indicator, the following is the format of the Bookmark record:

```
int bytes_to_follow; /* standard, must be here */
int time_of_year;
int errlog_offset;
int scclog_offset;
int reserved4future1;
int reserved4future2;
```

Using this standard format, the value in bytes_to_follow would be 20 (four bytes times five words).

5.10 Calibration Records

The first word must be the length in bytes of the rest of the record. After that, there would be one or more calibration values for each OM of each string in the array. The format here is not defined yet because very little related to the calibration processed has been defined at this point.

5.11 User Records

The second word must be the length in bytes of the rest of the record. The third word should contain the time/date stamp. Other than that, this may contain anything the user desires. It is suggested that the fourth word of the record (after the record type, length, and time indicators) be another integer key which can be assigned different values so that the user can differentiate between his own various user records.

5.12 User Text Records

The second word must be the length in bytes of the rest of the record. The third word should contain the time/date stamp. The rest of the record will be ascii text characters.

6 Error and Operator Log File

This file is completely ascii text. The format of each line is:

mmm dd hh:mm:ss yyyy ID messageString

where mmm is the month, dd is the day of the month, hh is hours, mm is minutes, ss is seconds, yyyy is year, ID is a program identifier, and messageString is any sequence of characters terminated by a carriage return. ID is always exactly two characters, and ID identifies the program which posted the message. Some example IDs are:

- EL = Error Logger
- BM = Bookmark
- OP = Operator Comment
- OM = Optical Module (and other modules, too) Status Monitor
- DH = Data Harvest
- EC = Environmental Computer
- TP = Trigger Processor (68040)

7 Usage Explanation

This section contains an oversimplified example of the organization of a data file collected by the DUMAND software. Keep in mind that this is a simple example and it is not to be taken literally. For instance, there may be many more than three header records as shown in this example.

A sample archive data file collected by the DUMAND software might have records like these:

- record1: basic description of array: how many strings, which OMs have shorted out, etc.
- record2: triggering parameters used
- record3: calibration results from most recent CM firing sequence
- record4: Unprocessed Environmental information #1
- record5: Array positions #1, calculated by EC from env. info. #1
- record6: event1
- record7: event2

- record8: TOY stamp, including scalers, longons, etc.
  ⋮
- recordA: event n
- recordB: TOY stamp, including scalers, longons, etc.
- recordC: event o
- recordD: event p
  ⋮
- recordE: event q
  ⋮
- hundreds and hundreds more event records and TOY stamp records
  ⋮
- recordF: Environmental Info #2
- recordG: Position data #2
- recordH: event r
- recordI: event s
- recordJ: TOY stamp, including scalers, longons, etc.
  ⋮
- recordK: event t
- recordL: event u
  ⋮
- hundreds and hundreds more event records and TOY stamp records

When an investigator runs his program which searches for a certain event type on this file, then a similarly formatted file, but only containing the matched events should be written out by this program. For instance, suppose the only two events in this file which meet his criteria are Events "o" and "t". The new file output by this search program might contain the following records:

- record1,2, and 3
- records 4 & 5
- recordB (time-of-year stamp for event "o")
- recordC event "o"

- records F & G
- recordJ (time-of-year stamp for record "t")
- recordK event "t"

This new file should be pretty compact because it has left out all the hundreds or thousands of events not matching the search criteria. However, it has copied some of the other information that any down-the-line analysis program will need: Records 1-3 are needed to describe the state of the array to that down-the-line analysis program. Records 4 & 5 are saved to know the positions of the OMs for event "o" (presumably these exact positions would be wanted by a detailed fitting program), and so on.

Since the shorter output file is in the same format as the original DUMAND archive file, either file could be used as an input file to another event-matching program.

8 Text Event Representation

This section specifies a standard format for an ascii text file describing an event. The format is line oriented, with the first character of a line determining the type of information line. The standard only defines three line types. Lines beginning with an 'E' indicate the beginning of a new event. Lines beginning with an 'H' contain information about one PMT hit in that event. Lines beginning with an 'F' contain information related to the fitting of the event (either by a reconstruction program or from the MC program generating the event). (Section 8.5 also defines a fourth 'R' line which is not discussed here yet to avoid too much confusion.) The formats of the lines are:

E evnum nhits GPStMSW GPStLSW DUMtMSW DUMtLSW μstime trigger
H string om HitEnergy timehit coincidence
F type x y z xdir ydir zdir EventEnergy time chisq

The F line is optional. If there has been no fit performed, or if the data is simply random noise generated by a MC program, no F line need be included for an event. (Similarly, if there are zero hits in an event, there would be no H lines).

Definitions of each of the items on these lines is as follows:

1. <u>evnum</u> This is an arbitrary number, generally monotonically increasing over the course of a run, used to help identify an event. Provisions are made in the software to allow this number to continually increase, even when the system is stopped and re-started.

2. <u>nhits</u> Number of hits to follow in this event.

3. <u>GPStMSW</u> The Most Significant Word of the latest GPS time reading reported (within the last second). This gives time resolution from seconds up through years.

4. GPStLSW The LeasT Significant Word of the latest GPS time reading reported (within the last second). This gives time resolution from nanoseconds up to one second.

5. DUMtMSW The Most Significant Word of the DUMAND time reading at the time of the last GPS time reporting.

6. DUMtLSW The Least Significant Word of the DUMAND time reading at the time of the last GPS time reporting.

7. μstime The corrected DUMAND time to the microsecond up to one second (20 bits) of the start of this event's triggered (center, or third) microsecond. This (the façt that μstime is really the time of the third microsecond saved) will undoubtedly be the source of much confusion. See also timehit below.

8. trigger This is the hexadecimal value of the trigger_reason field of the event. MC programs should simply set this to COLLECT_REASON_MC. trigger is used as a bitmask and so will be printed in hexadecimal.

9. string The string number reporting this hit.

10. om The OM number recording this hit.

11. HitEnergy The HitEnergy recorded by this hit in PEs. This is a floating point number (e.g. 1.5), however, you should keep in mind that the binary archive record format stores very limited precision values. See the Section 8.5 for a discussion of this.

12. timehit The time of this hit, in nanoseconds from a start point defined as μstime - 2 microseconds.

13. coincidence This is an optional field, which can indicate whether this hit made a T2 or a T3. This field is a string. If it is "T1", it means it did not form any coincidence. If it is "T2", that means this hit made a T2 coincidence. If it is "T2s", that means this hit made a T2skip coincidence. If it is "T3", that means this hit made a T3 coincidence. If it is "T3s", that means this hit made a T3skip coincidence. This field is optional. If you leave it off, that means you either didn't compute it or don't care or both.

14. type This indicates what type of event the fitter (or MC program) determined this event to be. A possible set of values might be: 1=don't know, 2=contains noise, 4=muon, 8=cascade, 16=SN, 32=fuzzball/quarkbag/monopole, 64=other (interesting but don't know what). A value of zero means no classification was even attempted. type is used as a bitmask and so will be printed in hexadecimal.

    This is a bitmask because someone's MC program may generate events like "muon tracks during a SN with background noise", then they set type to $2 + 4 + 16$ and code in all the information about what was being generated.

15. x, y, and z These are the origination/intersection point of the event, in mm from the center of the DUMAND coordinate system.

16. xdir, ydir, and zdir These indicate the direction of the muon track. These are printed as floating point cosines (unit vectors) (not times one million here, so these printed ASCII values are one-millionth of the values stored in the std. binary data block).

17. EventEnergy The fitted (or generated) energy of an event in GeV.

18. time The fitted (or generated) time in nanoseconds from the time of the event trigger. This will generally be a positive number. This number is subtracted from the microsecond time of triggered microsecond to give the event time.

19. chisq The chi-squared value from the fitting of this event. MC program output would just fill in a value of zero for this. This is printed as a floating point number, and is not multiplied by 100 as it is when stored in the standard fit structure.

In general, all the above values are integers unless otherwise specified. If a MC program or fitter is not using certain types of information, it should just fill in 0 (zero) for that value (for instance, if a fitter isn't fitting the energy).

8.1 Extending the Text Representation

Note that the standard only includes those lines whose first character is 'E', 'F', or 'H'. Users may insert other lines beginning with any other character to use as comments or to hold other user-defined information (perhaps for interchange within a local group).

8.2 Sorted Order of Hits

The standard way of listing the hits in the text event file is as follows: First, all the hits from one string in the first microsecond of the event are listed in time-sorted order (earliest hits listed first). Then the hits from another string with hits in the first microsecond, and so on until all strings with hits in the first microsecond have had their hits listed for that microsecond. Then the process is repeated for the each subsequent microsecond until each microsecond has been listed. For example, suppose we have three strings, A, B, and C, and an event containing three microseconds. The hits would be listed in the following order:

1. String A, all hits from the first microsecond

2. String B, all hits from the first microsecond

3. String C, all hits from the first microsecond

4. String A, all hits from the second microsecond

5. String B, all hits from the second microsecond

6. String C, all hits from the second microsecond

   ⋮

7. String A, all hits from the fifth microsecond

8. String B, all hits from the fifth microsecond

9. String C, all hits from the fifth microsecond

If a string has no hits for a given microsecond, then simply no hits are listed for that string that microsecond. For instance, if there were no hits in the first microsecond on string B, then Item 2 in the above list would simply not be there. If there are no hits on all strings for a microsecond, then, as you might expect, there are no hits listed for that microsecond. In this example, Items 4 through 6 would be deleted from the above list.

Note that there is nothing which insists that $C > B > A$. (In other words, the strings may be listed in any particular order.)

This format may seem difficult to use at first, but it has several key advantages:

- It is very similar to the order hits are stored within a binary DUMAND archive file.

- Since it is listed microsecond by microsecond and sorted within the microseconds for each string, it simplifies writing programs to feed data from one of these files into the DUMAND data acquisition algorithms for testing.

- The time-sorted order makes it straightforward to write out a very long time period "event", such as an MC program creating several seconds of simulated data might create.

- I proposed it, it must be good.

The next section contains an example which might help you understand the format.

8.3 Sample Text Representation

Below is a sample text file containing 2 events:

```
E 1 18 1002030 12345678 1001029 3452129 3567 1c
C This line is ignored as a comment since it doesn't begin with E, H, or F.
H 8 10 1.0 180 T1
H 1 1 1.5 443 T1
H 3 23 1.0 693 T1
H 3 18 2.5 577 T1
C Here come the hits from the second microsecond
H 8 17 3.0 1578 T2
H 8 19 1.0 .1593 T3
H 8 16 2.0 1604 T3
H 8 15 1.5 1639 T3
C Here come the hits from the third microsecond
H 8 23 1.0 2858 T1
H 1 16 3.0 2192 T1
H 1 14 4.0 2192 T2s
H 3 2 1.0 2428 T1
C Here come the hits from the fourth microsecond
H 3 3 1.5 3093 T1
H 3 8 1.0 3577 T1
H 3 6 1.0 3878 T1
C Here come the hits from the fifth microsecond
H 8 13 2.0 4099 T1
H 1 5 1.0 4277 T1
H 3 20 1.0 4999    T1
E 2 15 1002310 21658733 1001327 4325999 427 aa0
H 8 10 1.0 180 T1
H 1 12 7.0 443 T1
H 3 13 1.5 693 T1
H 8 8 1.0 1492 T1
H 1 7 3.0 1578 T1
H 3 9 1.5 1999 T1
H 8 16 2.0 2525 T1
H 1 15 2.0 2102 T1
H 3 23 1.0 2858 T1
H 8 16 3.0 3019 T1
H 1 14 1.0 3829 T1
H 3 2 1.0 3564 T1
H 8 3 3.0 4823 T1
H 1 8 1.0 4712 T1
H 3 6 1.0 4601 T1
```

```
C Here comes the fit for this most recent event (#2).
F 1 123 29 27 803400 757190 12324 239000 0 12
```

The first event has 18 hits, but no fit line at the end. The second event has 15 hits and a fit line at the end. Only hits for three strings are shown. Notice the way the hits are time-sorted within one microsecond (within each 1000 ns block) for each string.

8.4 Program Available

Dennis Nicklaus at the University of Wisconsin has a program which converts from the binary archive file format to the ASCII text format.

8.5 Hit Energy Discussion

The energy value stored in the binary archive file as defined in Section 5.2.9 is technically in arbitrary units. These units are whatever look-up values are defined for the FLP (UW-Madison designed First Level Processor) for its 10bit to 8bit energy conversion. However, one would expect these lookup values to correspond to something physical, such as PE's or PE's time 10 (to record tenths of PEs). If/When all 10 energy bits are used, the 10bit "energy" stored in that word will actually be a time duration – the number of nanoseconds the tube remained on.

To recap the confusing factors:

- MC programs currently produce data which give hit energies in PE's, perhaps to some fractional precision.
- The first design of the FLP will only provide an 8bit energy value which presumably will correspond to PEs, but may not be linear.
- A later design of the FLP may simply record the 10bit hit time duration (in ns). There would be other calibration data stored in the archive (in a Calibration record) to allow conversion to PE's.

Due to these factors, we also need an 'R' (for Raw) line similar to the 'H' line. The 'H' line remains as above, with the HitEnergy defined in PE's as a floating point number:

H string om HitEnergy timehit coincidence

The 'R' line looks similar:

R string om PulseWidth timehit coincidence

The PulseWidth is the raw value originally stored in the binary data archive record. One could convert an 'R' line to an 'H' line by knowing the correct conversion factors which would be computed during a calibration. Keep in mind that "PulseWidth" is used rather loosely here because for data produced by the currently designed FLP, pulsewidth is actually an 8bit compressed value of the actual pulse width seen.

9 Appendix 1. Raw Digitizer Output Format

During OM testing and perhaps at other times, it will be necessary to store "raw" digitizer output. This may help test whether PMTs and the digitizer and the optical link are all functioning properly. This data contains the 40 bit ocean bottom digitizer word with a 24 bit absolute time appended to it. A series of words like this will be stored as the raw digitizer output:

```
word1
word2
word1
word2
word1
word2
...
```

Where each word is 32bits because the Sun doesn't really like 64bit words (except for double precision floating point).

word1 looks like this:

```
MSBit                                                                  LSBit
-----------------------------------------------------------------------------
|Usec(24) | parity(4) | NullMark(1) | ovfl_b(1) | ovfl_a2(1)| u/d2 (1)|
-----------------------------------------------------------------------------
```

word2 looks like this:

```
MSBit                                                                  LSBit
-----------------------------------------------------------------------------
|time2(10) | OMnum2(5) | ovfl_a1(1) | u/d1(1) |  time1(10) | OMnum1(5)|
-----------------------------------------------------------------------------
```

Numbers in parentheses indicate the field widths in bits.

Within the 64 bit long word, there will be two 17 bit words side by side, the first one on bits 0-16, and the second on bits 17-33, then an overflow flag on bit 34, a null word marker on bit 35, and 4 parity bits on bits 36-39. Then there will be a 24 bit absolute time on bits 40-63. The absolute time will have a least significant bit of exactly 1uS, and every 16 seconds it will go back to 0. The decimal numbers 16,000,001 to 16,777,216 should never occur. This absolute time will mark the time that the data word got strobed out of the hot rod receiver and into the Sparc board (or wherever) FIFO, so it is only an approximation of when the event actually occurred because of the unknown delay in the digitizer FIFO and through the hot rod/optical link. In the two 17 bit words, the bits 0-4 (or bits 17-21 for the higher word) will be the channel number. Bits 5-14 will be the time, bit 15 is the up/down flag, and bit 16 is an overflow flag. Notice this is identical to the digitizer word format as described in the digitizer specification. A rollover has occurred when channel 31 comes around, and for this case, the time data is actually hydrophone data and should be ignored for the histograms. The entire 64 bit word that contains a 1 in the null marker (bit 35) is a null word. Also, one of the OM channels, probably #26, is used for the digitzer local time of the last null word.