MTsort Language - EDOC033

John Cresswell

Janet Sampson

MTsort Language - EDOC033

John Cresswell
Janet Sampson

Publication date 21 Oct 2010

Abstract

This manual describes the sort language. It can currently be used to sort a wide selection of event formats, including Eurogam, Euroball, GammaSphere, IN2P3, Goosy, Oak Ridge, Exogam and GREAT format data. More features and data formats are being added according to users' requirements.

Table of Contents

Introduction

In order to run a sort the user supplies a file of sorting instructions written in the sort language. This data file should contain a description of the experimental setup, the number of spectra (histograms) required and a set of commands to be applied to each event being sorted. The sort package checks the syntax of these instructions and translates them into a more low level language, i.e. C. This translation is then compiled to produce a sort program based on the user's original sort instructions.

If there are mistakes in your sortfile the sort package reports them as warnings or errors when you try to set up a sort. Warning and error messages are given with the line number and a copy of the erroneous line.

Feedback

Feedback is most certainly welcome for this document. Send your additions, comments and criticisms to the following email address : <support@ns.ph.liv.ac.uk>.

Data File Format

General Structure

The user-supplied data file is divided into several sections. Each section is identified by an associated **starword**.

The recognised starwords are:

- *FORMATS
- *TRIGGERS (optional)
- *DATA (optional)
- *SPECTRA (optional)
- *AUTOGAIN (optional)
- *COMMANDS (optional)
- *FINISH (denotes end of sortfile)
- *TRIGGERS (optional)

and should appear in the data file in the above order. Items in the data file are in free format separated by spaces. Each starword section is described in a following chapter.

All **starwords**, **commands** and other sort language statements are expected on a new line. Any statement which exceeds a single line may just be continued on the following line. There are no line continuation symbols.

Names used may contain one or two components depending on whether the quantity is a group name or not. Each component must commence with an alphabetic character and may be up to 16 characters in length. Only alphanumeric and underscore "_" characters are allowed.

All names used for single and group parameters, sortwords, items, arrays, maps, parameter lists and routine arguments must be unique, e.g. a parameter list may not have the same name as a sortword.

Numerical values should be specified as appropriate to the command:

- integer (in the range -32768 to 32767)
- real (in the range 10^{-38} to 10^{38})

  e.g. 2.143 2376. 936.52E5 1.509E-23
- binary (up to 16 bits) e.g. %1011101101
- hexadecimal (in the range FFFF to 7FFF) e.g. @0065

Reserved words consist of all keywords and maths function names and cannot be used as other sort program names. See Appendix A for a list of all reserved words.

Comments may be placed anywhere in the text. Any text following an exclamation mark ! or double forward slash // up to end-of-line is ignored by the setup procedure.

Notation

The convention in this manual is to show command names and keywords in upper-case, and substitutable values in lower-case italic.

Note, however, that command entry in the sort-file is not case-sensitive.

Optional quantities are enclosed in square brackets, e.g. # optvar #.

The letter "r"' following a quantity indicates that the item may be repeated.

Alternative quantities are denoted by |, so a|b indicates **either** a **or** b.

In the commands section wherever <statements> is used it refers to either a simple statement (single command) or a complex statement (group of commands enclosed within curly braces), i.e.

```
statements  ->  single-command
                |
                {
                  single-command
                  single-command
                 ...
                }
```

File Inclusion

```
INCLUDE <filename>
```

This statement allows other text files to be included in any section of the data file. Only one level of inclusion is allowed, i.e. included files may not contain any **INCLUDE** statements.

*FORMATS

This section is used to specify all experimental parameters and any other parameters required during event processing. Parameters can be specified in two different formats depending on how they are to be accessed: either as group or single parameter format.

Single Parameter Format

```
<name> <address>
```

In single parameter format the 14-bit address of the parameter needs to be specified.

Example

```
*FORMATS
GE13_E2  @010D
silena1  513
```

This address must be unique and not lie within the address range of any group format names [See the Eurogam document EDOC014 (Event Builder + Sorter Control) for further information].

This format is useful for sorting non-Eurogam format data.

Group Parameter Format

<*name* > <[*number*]> <(*item-list*)>
<*name* > <[*number-range*]> <(*item-list*)>

where <*number-range*> is a subset of the allowed group numbers (0 to 1023) *enclosed in* [] *brackets*, and takes the form:
<*lower-limit*>
<*lower-limit*> : <*upper-limit*>
<*lower-limit*> , <*next-lower-limit*> : <*next-upper-limit*> , ...
<*lower-limit*> : <*upper-limit*> , <*next-lower-limit*> : <*next-upper-limit*> , ...

and <*item-list*> is a list of the items contained within a group separated by commas. Each item consists of a name followed by an optional bit field, or an array:

<*item-name*> [: <*number-of-bits*>]
<*item-name*> (<*array-length*>)

Group numbers less than 256 correspond to standard group format; group numbers of 256 upwards correspond to extended group format. Within any one group the format must be the same, i.e. group numbers must be in one format range only (0--255 or 256--1023). A group consists of all the parameters associated with one device, e.g. a germanium detector, would have an associated energy word, ballistic deficit correction words, etc. The structure of all devices having the same sets of associated parameters can be specified concisely using group format,

Example

```
*FORMATS
GE[2,4:10,19,23:26] (E1,E2,TAC,TACBD)
CLOVER[51:74] (BGOE,BGOT,BGOP,
                A1,A2,A3,A4,
                B1,B2,B3,B4,
                C1,C2,C3,C4,
                D1,D2,D3,D4)
CLOVER1[101:124] (E20, E4TAG1:3, E4DAT1:13)
CLOVER2[151:174] (E20, E4TAG1:3, E4DAT1:13, E4TAG2:3,  E4DAT2:13)
TRACES[256:300]  ( pulse(128) )
```

where the group name **GE** represents a group type consisting of 4 items: **E1**, **E2**, **TAC** and **TACBD**. defined for group numbers 2,4,5,6,7,8,9,10,19,23,24,25,26.

CLOVER has 19 items defined for group numbers 51 to 74 inclusive, whereas **CLOVER1** and **CLOVER2** are examples of groups which use bit fields to specify sections of the item data words for ease of access in the ***COMMANDS** section.

Within the commands section the syntax used to refer to a single item of a particular group would be:

<group-name> # <group-number> # . <item-name>

where **<group-number>** need only be specified if a range of group numbers have been defined for <group-name>.

Example

```
GE[13].E1
```

would refer to item **E1** of group 13.

Example

```
CLOVER[153].E4TAG2
```

would refer to the item **E4TAG2**, i.e. to the top 3 bits of the third data word, of group 153.

If only one group number is defined for a single group name, in ***FORMATS**, then it may be referenced in the commands section without specifying the group number, e.g.

Example

```
TRIG[255] (S1,S2)
```

would be referenced as **TRIG.S2** to access the second item of group **TRIG**.

Example

```
traces[256:300] ( pulse(128) )
```

would be referenced as **TRACES.PULSE(I)** to access the i'th **PULSE** item of group **TRACES**.

```
e.g.
i = 17
sample = traces[256].pulse(i)
```

*TRIGGERS

This optional section is provided for compatibility with non-Eurogam format data.

<trigger-number> [< adc-name >] r

where <trigger-number> is in the range 0 to 64.

For each trigger used the list of associated adcs should be specified. e.g.

Example

```
*FORMATS
GE1 1
GE2 2
GE3 3
GE4 4
GE5 5
*TRIGGERS
24 GE1 GE2 GE3 GE4 GE5
...
```

specifies that the event data words **GE1**, ..., **GE5** are declared as single parameters and are associated with trigger number 24.

*DATA

Sort variables and other program data are defined in this section.

Sortwords

Sortwords are variables used within the commands section to pass values between commands. They may be of type word, long, longlong or float. Long longlong and float types must be explicitly declared in this section. Any undeclared variables in the commands section are assumed to be of type word. Sortwords are not limited in scope i.e. they are recognised in the main commands section and all routines.

If a sortword is defined in this section and initialised with a starting value, then the sortword is considered global. This has the effect of keeping its value across events. Sortwords are normally undefined until first use in an event.

```
WORD <name> # = <integer-value> # ...

LONG <name> # = <integer-value> # ...

LONGLONG <name> # = <integer-value> # ...

FLOAT <name> # = <floating-point-value> # ...
```

```
where
WORD declares a 16-bit integer,
LONG a 32-bit integer
LONGLONG a 64-bit integer
and FLOAT a 32-bit real.
An optional initialisation value may be specified; if omitted it
will default to zero.
```

Example

```
WORD COUNTER1=1 COUNTER2=1
FLOAT PI=3.14159
```

declares two 16-bit integer variables **COUNTER1** and **COUNTER2** both initialised to 1 and one 32-bit floating point variable **PI** initialised to 3.14159.

Initialisation occurs once at the start of each sort program run.

If a **WORD** variable is to be output from the commands section using the **OUTPUT** command then it must be defined with an associated address:

```
WORD <name> # = <integer value> # AT <14-bit-address>
```

The address is neccessary for word variables to be output in Eurogam format, i.e. a data word with a 14-bit address, so that they can be re-sorted later as pseudo-adc words. The address must lie in the range 0 to 16383 (2^{14}-1) and not coincide with any addresses assigned in the ***FORMATS** section.

Example

```
WORD GAMA AT @A
```

would define the word **GAMA** with hexidecimal address **A**.

Pre-defined Sortwords

The following sortwords have predetermined usage and value:

RANDOM	floating point sortword, random value between 0.0 and 1.0
IRANDOM	integer sortword, random value between 0 and 32767
GATE	see IF... and LOOPIF...MASKEDBY|GATEDBY commands
WORDX	see LOOPIF... command
WORDY	see LOOPIF... command
STREAM	Usually set =1
RUNFILE_NUMBER	runfile number of currently sorted tape (1 for first file, etc.)
BLOCK_NUMBER	current block number in currently sorted runfile
LOOP	no longer used, see doloop command

Gates

Sets of gates may be defined here for later use in the commands section through which to filter the event-by-event data. If a data word being tested matches a particular gate condition it is said to pass that particular gate.

Bitmask, 1D and 2D gates are stored as 8-bit lookup maps. Elliptical gates are stored as lists of coordinates and axes. When a sortword value is tested against a gate in the commands section it will pass either zero or one of the gates in the map. The gate number passed will be stored in the reserved variable GATE

Bitmask gates

A set of bitmask gates consists of one bit pattern per gate. Within a set of gates earlier gate definitions have precedence over later ones. This means that in the commands section if the same value would pass more than one gate out of a set then the earliest gate defined would be the one passed.

```
GATES MASK <bitmask-gate-set-name>
<bitmask1> <bitmask2> … <bitmaskngates>
```

Example

```
GATES  MASK  BITMAP1
%10000 %01000 %00100 %00010 %00001
```

Each data item consists of a 16-bit mask and represents one gate. A value will pass a gate if all the bits set in the 16-bit value are also set in the 16-bit mask of that gate.

Within the commands section a value will pass a gate if it falls in between the lower and upper limits (inclusive) of that gate.

1D gates

A 1D gate-map consists of one or more pairs of values. The range of values in between each pair (inclusive) defines a single gate. Within a set of gates successive gates in a 1D set have precedence over earlier ones. This means that in the commands section if a value would pass more than one gate out of a set then the latest such gate defined would be the one passed.

```
GATES 1D <1d-gate-map-name> [ < x-range > ]
( <low-limit> <high-limit> )1
( <low-limit> <high-limit> )2
...
( <low-limit> <high-limit> )ngates
```

where <x-range> is specified as:
<lower-limit> : <upper-limit>
or
<range>
where <lower-limit> would be set to zero and <upper-limit> would be equal to <range> minus 1.

Example

```
GATES  1D  BAND1[0:511]
(123 126)  (245 259)  (257 270)
```

defines a set of 1D gates **BAND1** within the limits 0 to 511 inclusive which contains 3 gate definitions:

```
gate 1 is defined as channels 123, 124, 125, 126;
gate 2 as channels 245, 246, 247,..., 254, 255, 256;
and gate 3 as channels 257, 258, 259, 260,..., 268, 269, 270
because gate 3 overlaps gate 2.
```

2D gates

A 2D gate-map consists of one or more sets of x--y coordinate pairs. Each set defines a polygonal-shaped gate in two dimensions against which pairs of values may be tested in the commands section. If any polygons overlap within a set successive gates have precedence over earlier ones.

GATES 2D <*2d-gate-map-name*> [< *x-range , y-range* >]
(<*gate-of-1D-coordinate-pairs*>)$_1$
(<*gate-of-1D-coordinate-pairs*>)$_2$
...
(<*gate-of-1D-coordinate-pairs*>)$_{ngates}$

Example

```
GATES  2D  MASSMAP[64,64]
(11  44  13  36  18  30  25  29  28  35  30  49  26  60 20  58)
(31  62  29  1  52  1  51  62)
```

Defines the map **MASSMAP** with limits 0 to 63 in both the x- and y- directions. A coordinate pair will pass a polygonal gate if the point it defines falls within the polygonal shape defined by that gate.

Note

1. The coordinate pairs are not individually separated to simplify the syntax, hence care must be taken when inputting the data.

2. 2D gatemaps become large for large values of <x-range> and <y-range> .

Elliptical gates

Elliptical gates may be specified in 2 or 3 dimensions. They are defined by specifying the coordinates and axes of each ellipse or ellipsoid making up the list of gates.

GATES ELLIPSE2D <*2D-elliptical-gate-name*>
(<*x-coordinate*> <*y-coordinate*> <*x-radius*> <*y-radius*>)$_1$

(<*x-coordinate*> <*y-coordinate*> <*x-radius*> <*y-radius*>)$_2$
...
(<*x-coordinate*> <*y-coordinate*> <*x-radius*> <*y-radius*>)*ngates*

where <*x-radius*> and <*y-radius*> define the radii for each axis of the ellipse.

Each set of coordinates and radii defines an elliptical gate against which pairs of values may be tested in the commands section. If any gates overlap within a set earlier gates have precedence over later ones. See IF...GATEDBY and LOOPIF...GATEDBY commands.

GATES ELLIPSE3D <*3D-elliptical-gate-name*>
(<*x-coordinate*> <*y-coordinate*> <*z-coordinate*> <*x-radius*> <*y-radius*> <*z-radius*>)$_1$
(<*x-coordinate*> <*y-coordinate*> <*z-coordinate*> <*x-radius*> <*y-radius*> <*z-radius*>)$_2$
...
(<*x-coordinate*> <*y-coordinate*> <*z-coordinate*> <*x-radius*> <*y-radius*> <*z-radius*>)$_{ngates}$

where <x-radius> <y-radius> and <z-radius> define the radii for each axis of the ellipsoid.

Each set of coordinates and radii defines an ellipsoidal gate against which 3 values may be tested at a time in the commands section.

Data arrays

Three types of arrays may be defined to store data for subsequent access in the commands section. Value arrays may be used to store integer or real data. Gate arrays store pairs of integer values to define arrays of gates. Gain arrays store the gain parameters associated with particular sortwords.

Value and 1D gate arrays both allow a lookup facility dependent on another parameter, e.g. group number.

Value arrays

VALUEARRAY defines a 1D, 2D or 3D array of 32-bit integer or real values that can be accessed in the commands section.

VALUEARRAY <*array-name*>
<*x-range*> #, <*y-range*> #, <*z-range*> # # SAVE [< *data-list* >]

where <*x-range*> is the channel range in the first dimension, and the y- and z- quantities the corresponding values in higher dimensions if applicable, specified in the same way as for gate-maps.

If no starting channel is given it is assumed to be zero and the maximum channel will be (<*x-range*> - 1) as before. If the **SAVE** keyword is specified, then the array is written back to disc at the end of the sort, allowing modified arrays to be preserved. This discfile will normally be in the sort setup directory created when a sortfile is compiled. The <*data-list*> is allowed in free format.

Note

The array data type (integer or float) is determined from the type of the first data element specified in <*data-list*> .

The values specified in <*data-list*> are given in C-style ordering: the z-parameter changes more quickly than y- which changes more quickly than x-. This is the opposite way round to the convention used in FORTRAN.

Example

```
VALUEARRAY   ANGLES   [1:20]
157.60 157.60 157.60 157.60 157.60
133.57   0    107.94   0    107.94
133.57  94.16 133.57 107.94  94.16
107.94 133.57   0    133.57 107.94
```

defines a real 1D array **ANGLES** containing 20 elements.

Example

```
VALUEARRAY   ARRAY2  [2:6,3]  1 11 21  2 12 22  3 13 23  4 14 24  5 15 25
```

defines an integer 2D array **ARRAY2** spanning from channels 2 to 6 in the first dimension (5 channels) and from channels 0 to 2 in the second. The values will be assigned as follows:

(2,0),(2,1),(2,2),(3,0),(3,1),(3,2),(4,0),(4,1),(4,2)... etc.

An example of their use in the commands section would be:

Example

```
A = B / ANGLES( <argument> )
C = ARRAY2(  <argument> 1,  <argument> 2)
```

where <argument> is an integer expression. See Arithmetic Expressions.

Any array elements not initialised by **VALUEARRAY** are set to zero.

Gate arrays

A gate-array contains the definition of a 1D array of pairs of channel numbers and the corresponding array element number. Data is allowed in free format specified in order of increasing array element number in the range 0 to 255 where array elements may be omitted from the sequence. Each pair of channel numbers defines a gate.

GATEARRAY <*1D-gate-array-name*>
<*array-index*$_1$> (<*low-limit*> <*high-limit*>)$_1$
<*array-index*$_2$> (<*low-limit*> <*high-limit*>)$_2$
...
<*array-index*$_{ngates}$> (<*low-limit*> <*high-limit*>)$_{ngates}$

Gate-arrays may be defined here and used in the commands section to filter all group format data of the same type through different 1D gates. See commands IF...PASSES and LOOPIF...PASSES.

Example

```
GATEARRAY TACGATES
1 (100 4000)   2 (95 4000)  6 (100 3950) ...  40 (85 4000)  45 (100 4000)
```

would define the gate array **TACGATES**. This array could then be accessed in the commands section to filter each germanium TAC word with parameters dependent on the group number.

Gain arrays

Sets of gain matching parameters may be specified in this section by means of a **GAINWORD** or **GAINARRAY** statement and referenced in the commands section via the GAIN command.

GAINWORD <*parameter-set-name*> <a> <b> <c>

GAINARRAY <*gain-array-name*> <a> <b> <c>
<*array-index*$_1$> <a_1> <b_1> <c_1>
<*array-index*$_2$> <a_2> <b_2> <c_2>
...
<*array-index*$_n$> <a_n> <b_n> <c_n>

where <*parameter-set-name*> contains the single set of parameters <a_n> <b_n> <c_n> and

<gain-array-name> contains <n> sets of gain matching parameters.

GAINWORD is designed for use with single variables and **GAINARRAY** is used with group-format variables, allowing the same item name associated with all groups of the same type to be gain-matched by a single command line in the commands section,

GAINARRAY data is allowed in free format with array element number in the range 0 to 255.

Example

```
*FORMATS
GAINWORD GE_19  0.3 -0.05  0.00     // single variable parameters
GAINWORD GE_29  0.6  0.02  0.00

GAINARRAY E2GAINS                    // group format
1  (-0.2  0.10  0.00)
3  (0.7 -0.03  0.00)
...
70 (-0.1  0.05  0.00)
```

Each statement stores a set of gain-matching parameters associated with a particular sortword.

The value of <sortword> may then be modified in the commands section using the **GAIN** command according to the equation:

<sortword> = a + b* <sortword> + c* <sortword> ^{2}

Arrays of arrays

A set of arrays of the same type (valuearray, gatearray or gainarray) may be defined. This is currently implemented for gainarrays and gatemaps in commands if...gatedby and loopif...gatedby.

ARRAYLIST <*arraylist-name*> [< *array-name* >] r

The **arraylist** defines an array starting at element zero.

*SPECTRA

<*spectrum-name*> # <[*index-range*]> # <*number-of-channels*> [< *type* >] # DISC #
where <*index-range*> is expressed as
<*lower-limit*> : <*upper-limit*>
and <*lower-limit*> and <*upper-limit*> are optional integer values which allow more than one spectrum to be declared by a single statement.
See Indexed Spectrum Updates section for an example application.
and <*number-of-channels*> is one of:

<*integer*>	1D spectrum
<*integer*> * <*integer*>	Rectangular 2D spectrum
2D	Square 2D spectrum
<*integer*> * <*integer*> * <*integer*>	Cuboid
3D	Symmetrised 1/6 cube

and <type> (optional) is:

8	Signed byte precision, 8-bits per channel
16	Signed single precision, 16-bits per channel
32	Signed double precision, 32-bits per channel

The optional keyword `DISC` makes the spectrum disc-based during sorting.

If <type> is omitted then the default of 32 is assumed for 1D, 16 for 2D and 3D.

By default, spectra are sorted into shared memory. It is the user's responsibility to ensure that there is sufficient memory available. Any combination of memory and disc-based spectra may be specified but as the sort package is essentially memory-based and is not fully optimised for disc-based sorting, the use of disc-updated spectra will degrade the performance.

For the maximum number of spectra allowed, see Appendix A.

Example

Some typical spectrum declarations might be:

```
*SPECTRA
TIME 1024                  // 1D 16-bit, 1024 channels
GEL1 4096 32               // 1D 32-bit, 4096 channels
GEL3 4096*1024             // 2D 16-bit, 4096 by 1024 channels
GEL4 1024 2D               // 2D 16-bit, 1024 channels square
GSPEC 1024*1024*8          // 3D 16-bit, 1024 by 1024 by 8 channels
SM[4:10] 4000              // 7 1D 16-bit spectra, 4000 channels each
CUBE[1:5] 16 3D 8          // 5 3D 8-bit 16 channel cubes
```

*AUTOGAIN

Gain drifts can be monitored via the ***AUTOGAIN** section.

The gain matched values of two well-defined peaks must be supplied. Initial gain coefficients for a quadratic fit may be supplied in the ***DATA** section. Alternatively, two peak positions for each data value to be monitored may be supplied in this section and initial linear coefficients will be derived.

The ***AUTOGAIN** section adjusts the gain coefficients by measuring the shift in two peak positions for each spectrum. For an initial calibration E for data value x:

$$E = a + bx + cx^2$$

and for the shifted energy E_{new} given by:

$$E_{new} = A + BE$$

then the shifted coefficients are derived as:

$$a_{new} = Ba + A$$

$$b_{new} = Bb$$

$$c_{new} = Bc$$

The gain coefficients are applied to the data by means of the GAIN command in the ***COMMANDS** section.

The gain coefficients are calculated in the autogain section and updated into a gain array (defined in the ***DATA** section). This gain array needs to be associated with a set of data words by an **INIT** statement in the autogain section.

The user can set the number of blocks (autogain period) over which the gain coefficients are initially calculated and subsequently monitored during the sort. A minimum acceptable peak area and maximum deviation may also be specified.

After the sample number of data blocks have been read, the peak centroids in the autogain spectra are determined and matched to the control values. This enables the gain shift to be calculated. At the start of an autogain sort, only the commands in the autogain section are executed until all the initial gain coefficients have been determined. If they have not all been determined after three times the autogain period then the autogain phase will stop. Any unresolved coefficients will take the initial values supplied at the start of the autogain phase.

After this initial autogain phase offline, the first file will be rewound and the sort will restart, now executing the statements in the ***COMMANDS** section and checking for gain drifts each autogain period.

If a peak centroid in a gain spectrum shifts by more than the specified deviation then the gain coefficients for the corresponding data word will be recalculated. All autogain spectra are zeroed after the gains are monitored each time.

Declarations

SAMPLE <*nblocks*>

where <*nblocks*> is the number of blocks after which the gain coefficients are recalculated.

Up to four times the sample number of blocks may be used to obtain the initial gain coefficients. A default value of 50000 blocks is assumed if none is specified.

PEAKAREA <*minimum-acceptable-peak-area*>

where <*minimum-acceptable-peak-area*> is the minimum integration area under a peak required for the gain spectrum to be used to evaluate gain coefficients.

An estimate is made of the background under the peak in order to calculate the peak area. A default value of 50 is assumed.

DEVIATION <*maximum-acceptable-centroid-shift*>

where <*maximum-acceptable-centroid-shift*> is the maximum centroid shift of a peak in a gain spectrum to avoid calculation of new gain coefficients.

A default value of 1.0 is assumed.

INIT <*gain-array*> FROM <*1D-spectrum*> CENTROIDS <$centroid_1$> <$width_1$> <$centroid_2$> <$width_2$>
PEAKS
<*group-number*> <$centroid_1$> <$width_1$> <$centroid_2$> <$width_2$> #r
#

where <*gain-array*> is the name of a gain array already declared in the *DATA section,
<*1D-spectrum*> is the name of an array of 1D 32-bit precision spectra declared in the *SPECTRA section
and the centroids and widths are floating point numbers denoting the gain matched positions and widths of two control peaks to gain match to.

The **INIT** line is optionally followed by a **PEAKS** statement in which estimates of the actual peak parameters are specified for each group number.

VOVERC <*real number*>

COPYGAIN FROM <*gain-array*> [< *group-range* >] TO <*gain-array*> [< *group-range* >] ANGLES <*value-array*> DELTAS <*value-array*>

These statements allow the gain coefficients to be adjusted for detectors where multiple leaves fire and the midpoint angle is used to correct the autogained coefficients for such data. See example in Appendix B for use.

Example

```
*DATA
GAINARRAY GAINS1
*SPECTRA
GSPEC[2:5] 4096 32
*AUTOGAIN
SAMPLE 10000
PEAKAREA 40
DEVIATION 0.80
INIT GAINS1 FROM GSPEC CENTROIDS 550 5.0 1204 7.3
PEAKS
2 545 5.0 1200 7.2
3 546 5.0 1202 7.2
5 551 5.0 1207 7.3
  ...
```

Commands

Only a subset of the full ***COMMANDS** section is available here to allow the update of gain spectra.

```
CREATELIST <group-parameter-list-name> FROM <group-name>
```

CREATELIST defines an internal list of data words from <*group-parameter-list-name*> which consists of the variables specified in an item list that are found in the current event,

Example

```
CREATELIST  GELIST  FROM  GE
```

would create the group-parameter-list **GELIST** consisting of all the germanium groups.

```
INC <autogain-spectrum-name> ( <x-channel> ) INDEXED <index>
```

Spectra may be indexed my means of the **INDEXED** keyword used with the **INC** command where <*index*> may be an integer expression, or dollar word used to specify a group number. The value of <*index*> determines which spectrum will be incremented: a value of 1 indicates the first spectrum in the array; 2 indicates the second, and so on. Spectra indexed in this way must all have the same dimensions and precision and be defined consecutively in the spectra section.

See first example in Appendix B for how to use autogain to derive gain coefficients and then use them in the main commands section.

*COMMANDS

This starword recognises as **keywords** all sort command names. The sort commands are executed for each event in the order in which they appear in the setup file. A **sort command** is a built-in routine which performs a function on the sortwords.

List of Commands

INC	increments a spectrum
DEC	decrements a spectrum
SET	assigns a value to a spectrum channel
INCBITS	increments the bit pattern of an expression into a 1D-spectrum
CREATELIST	defines a parameter-list of parameters from the event
EXTRACT	obtains subsets of valid parameters from a defined parameter-list
LOOPEXTRACT	obtains subsets of valid parameters from a defined parameter-list
IF	conditional execution of sort commands
LOOPIF	conditional execution of sort commands in parameter-list environment
SELECT	allows correlation of sort commands with parameter values
GOTO	jump forward to a specified label
LABEL	define a label to jump to
INVALIDATE	allows a group to be removed from the event
GROUPFILTER	allows groups to be removed from the event
ORDER	orders a list of sortwords according to their value
GAIN	adjusts the gain of a sortword using a quadratic
ROUTINE	starts a subroutine-like section
CALL	execute the set of commands in a ROUTINE
EXEC	execute an external sort function
DOLOOP	allows looping over several commands
OUTPUT	outputs a list of sortwords or a complete event
ENDEVENT	terminates event processing
END	ends event processing, or returns from called ROUTINE
PAUSE	pauses event processing
x=expr	arithmetic operations, assign expression to sortword/spectrum x

A particular sort command may be used as many times as necessary subject to any system-dependent limit on resultant program size. Any sortword generated by a command may be used as input to any succeeding command.

The **IF**...**ELSE**, **LOOPIF**...**LOOPFAIL** and **IF**...**GOTO label** block structures should normally be used to define the processing flow.

Parameter Lists

High fold data can usually be sorted more easily by means of parameter lists. There are three types of list:

word-parameter-list	consisting of individual data words
group-parameter-list	consisting of members of a group
item-parameter-list	consisting of lists of items from one or more groups

Once a list has been created it can be operated on by other sort commands to allow the same command to loop over every item in the list. Commands which operate directly on parameter lists are:

CREATELIST
EXTRACT
LOOPEXTRACT
INC
DEC
INCBITS
LOOPIF

Simple Spectrum update commands

INC|DEC <*spectrum-name*> (<*x-channel*> # <*y-channel*> # <*z-channel*> # #)

SET <*spectrum-name*> (<*x-channel*> # <*y-channel*> # <*z-channel*> # #) = <expression>

where a channel is defined as one of the following ...
arithmetic expression
parameter-list
$word = group-parameter-list

INC | DEC increment/decrement the spectrum channel specified. **SET** sets the spectrum channel to the value given by <expression> . If a parameter-list is specified for a channel then the command will be applied to all members of the list present in the event. If the same list is used for different channels of a spectrum then the i^{th} element of a list will not be incremented with the i^{th} element.

Example

```
INC GMAT(GELIST.E2,GELIST.E2)
```

... increment the channels given by each **E2** word for parameter-list **GELIST**. Channels given by the i^{th} element of the first list and the i^{th} element of the second list are not incremented, i.e. the same gamma-ray is not incremented with itself.

Example

```
INC GS1(GE[1].E1)
```

... increment the channel **GE[1].E1** of spectrum **GS1**, where **E1** is an item associated with the group **GE[1]**.

Example

```
DEC GAMSPC((GAM1+100)/2)
```

... decrement channel **(GAM1+100)/2** of spectrum **GAMSPC**.

Example

```
SET TCHECK(10) = clock.w1
```

... assign the data word clock.w1 to channel 10 of spectrum **TCHECK**.

Example

```
INC MAT2D(WORDX,LISTX)
```

... increment channel given by the word **WORDX** (x-coordinate) and all valid words in word-parameter-list **LISTX** (y-coordinate) of 2D spectrum **MAT2D**.

Example

```
INC ALL_GES(GELIST.E2)
```

... increment channel given by all valid **E2** words in group-parameter-list **GELIST** of spectrum **ALL_GES**.

Example

```
INC GFEX3(GAMA,GAMB,GAMC)
```

... increment the symmetrised cube **GFEX3** at the location given by **GAMA**, **GAMB** and **GAMC**.

Any spectrum update attempted by a command in this sort package which falls outside the defined spectrum dimensions will be safely ignored.

Indexed Spectrum update commands

```
INC|DEC <spectrum-name> ( <x-channel> # <y-channel> # <z-channel> # #) INDEXED
<index>

SET <spectrum-name> ( <x-channel> # <y-channel> # <z-channel> # # ) INDEXED
<index> = <expression>
```

Spectra may be indexed my means of the **INDEXED** keyword used with the **INC** or **DEC** commands where index may be an integer expression, or dollar word used to specify a group number. The value of index determines which spectrum will be incremented, decremented or set: a value of 1 indicates the actual spectrum specified; 2 indicates the subsequent spectrum defined in memory, and so on. Spectra indexed in this way must all have the same dimensions and precision and be defined consecutively in the spectra section.

For example, it is sometimes useful to be able to update a different spectrum for each gate number passed by a gate-map testing command. e.g.

Example

```
*SPECTRA
    ...
CE132[3:20] 4096
    ...
*COMMANDS
    ...
IF GAM1 GATEDBY GLREC {
    ...
INC CE132(GAM2) INDEXED GATE
    ...
```

where **GATE** denotes the gate number passed in the gate-map **GLREC** by the **IF...GATEDBY** command.

If **GATE** is 4 then the 3rd spectrum defined after **CE132[3]** in the ***SPECTRA** section, i.e. **CE132[6]**, will be incremented with the value of sortword **GAM2**.

It is also possible to use this feature to increment a spectrum according to the group number of a word, e.g.

Example

```
INC CE132(GROUPA=GELIST.E2) INDEXED $GROUPA
```

where **$GROUPA** denotes the group number passed in the parameter list **GELIST**.

Incbits command

```
INCBITS <1D-spectrum-name> ( <bit-pattern> ) OFFSET <integer-offset> # IN-
DEXED <index> #
```

This command increments a bit-pattern into 16 channels of a 1D spectrum commencing at the offset specified, where **bit-pattern** is an expression. It may be optionally indexed. The least significant bit will be incremented at the channel given by **integer-offset** and successive bits in subsequent channels.

E.g. to obtain a spectrum of 4 bit-pattern sortwords, the command would be used as follows:

Example

```
INCBITS MULT(MB1) OFFSET 0
INCBITS MULT(MB2) OFFSET 16
INCBITS MULT(MB3) OFFSET 32
INCBITS MULT(MB4) OFFSET 48
```

where **MB1**, **MB2**, **MB3** and **MB4** are 4 adc pattern words and **MULT** is a 1D spectrum 64 channels long. The bit-pattern of **MB1** will be incremented into spectrum **MULT** starting at the first channel (offset 0), that of **MB2** incremented starting at the 17^{th} channel, etc.

Createlist command

<CREATELIST> <*word-parameter-list-name*> FROM [< *word-name* >] r

<CREATELIST> <*group-parameter-list-name*> FROM <*group-name*> # [<*group-number-range*>] #

where the optional <group-number-range> is used to specify a subset of group numbers from those defined for <group-name>
and takes the form of one or more of the following separated by , (comma) ...

<*group-number*>
<*group-number*$_1$> , <*group-number*$_2$> ...
or
<*lower-limit:upper-limit*>
<*lower-limit*$_1$*:upper-limit*$_1$> , <*lower-limit*$_2$*:upper-limit*$_2$> ...

CREATELIST <*item-parameter-list-name*> FROM [< *group.item-name* >] r

CREATELIST defines a list of data words present in the current event. A word parameter list is constructed from a list of individual words,

Example

```
CREATELIST  VARS1  FROM  TAC1 TAC2 TAC3 TAC4
```

A group parameter list is specified using a single group name,

Example

```
CREATELIST  GELIST  FROM  GE
```

would create the group-parameter-list **GELIST** consisting of all the germanium groups.

A group parameter list may also be specified from a subset of a group,

Example

```
CREATELIST  GELISTA  FROM  GE[1:6]
CREATELIST  GELISTB  FROM  GE[7,9:12]
```

would create the group-parameter-lists **GELISTA** consisting of germanium groups 1 to 6 and **GELISTB** consisting of germanium groups 7 to 12 omitting 8.

An item parameter list is specified using one or more group.item combinations,

Example

```
CREATELIST  GAMLIST  FROM  GE.E4  CLOVERS1.A2DAT
```

would create the item-parameter-list **GAMLIST** consisting of all the **E4** items in group **GE** and all the **A2DAT** words in group **CLOVERS1**.

Lists may be used with other commands so that a single command may be applied to all the members of the list in turn, .e.g.

Example

```
INC GAMTOT(GELIST.E2)
```

would increment all the **E2** words in list **GELIST** which were present in an event into spectrum **GAMTOT**.

Copylist command

<COPYLIST> <*word-parameter-list-name*$_1$> TO <*word-parameter-list-name*$_2$>

<COPYLIST> <*group-parameter-list-name*$_1$> TO <*group-parameter-list-name*$_2$>

COPYLIST copies the list of data words present in the input list to the output list.

Example

```
createlist gelist from ge
  ...
copylist  gelist to newgelist
  ...
```

Extract command

EXTRACT <*word-parameter-list-name*> INTO <*parameter-list*> ORDERED | REVERSED

EXTRACT <*group-parameter-list-name*> . <*item*> INTO <*parameter-list*> ORDERED | REVERSED

EXTRACT <*group-parameter-list-name*> INTO <*parameter-list*>

where <*parameter-list*> is:
<*word*> [< *word* >] r

EXTRACT places valid words from a parameter list into output words to be accessed individually.

In the first 2 forms of the command **EXTRACT** scans the items in <**-parameter-list-name*> and copies only those which are valid in the current event to <*parameter-list*> . In this case the values of the parameters in the input list are copied to the specified words,

Example

```
EXTRACT  LIST1.E2  INTO  GAMA GAMB GAMC GAMD
```

copies valid parameter values from the group parameter list **LIST1.E2** into the words **GAMA**, **GAMB**, etc.

In the third form of the command the group information is retained in the output parameters. This is through use of the $ symbol which indicates that the variable references a group-format parameter and does not contain the actual value itself. This means that the group number information, i.e. address, is retained in $-parameters.

The output data must be accessed by specifying the group and an associated item ...

Example

```
EXTRACT LIST1  INTO  $GA $GB
IF $GA.E2 PASSES GARRAY($GA)  {
   IF $GB.E2 PASSES GARRAY($GB)  {
       ...
   }
}
```

copies the group numbers of valid parameters in group parameter list **LIST1** into the group-words **$GA** and **$GB**. This group information is then used by the **IF...PASSES** commands to index into the gate array **GARRAY**.

If there are fewer valid words than output words they will be placed in the first output words specified in the command. If there are more valid words than available output words then the output words are chosen randomly from the valid words in the input list.

EXTRACT can be combined with the arithmetic **NUMBER** function (See Arithmetic Functions). to obtain the number of words in the parameter list which are present in the event,

Example

```
EXTRACT  GELIST.E2  INTO  GAMA GAMB GAMC GAMD GAME GAMF
IF NUMBER(GELIST) GT 4 {
    ...
```

The optional keywords **ORDERED** (**REVERSED**) enable the user to specify whether the values of words in the input list are to be output in order of increasing (decreasing) numerical value. If no keyword is specified then the values are output in the order in which they appear in the input list.

Loopextract command

LOOPEXTRACT <*word-parameter-list-name*> INTO <*parameter-list*> ORDERED | REVERSED {
<*statements*>
}

LOOPEXTRACT <*group-parameter-list-name*> . <*item*> INTO <*parameter-list*> ORDERED | REVERSED {
<*statements*>
}

LOOPEXTRACT <*group-parameter-list-name*> INTO <*parameter-list*> {
<*statements*>
}

where <*parameter-list*> is:
<*word*> [< *word* >] r

LOOPEXTRACT works in the same way as the **EXTRACT** command except that if there are more valid words in the input parameter-list than available output words then event processing will loop over all combinations of input words for the associated <*statements*> .

Example

```
CREATELIST  LIST1  FROM  GE
   ...
LOOPEXTRACT  LIST1.E2  INTO  A B
    {
    INC MAT2(A,B)
    INC MAT2(B,A)
    }
```

So if there were 3 valid words in list **LIST1**, say *x*, *y* and *z*, then **LOOPEXTRACT** would be executed for the following parameter combinations (*x*,*y*), (*x*,*z*) and (*y*,*z*).

If...else... command (single sortword environment)}

IF <*test*> <*statements*> # ELSE <*statements*> #

where <*test*> is one of the following:
<*sortword* > VALID
<*sortword*$_1$> | <*bracketed_expr*$_1$> EQ|NE|GE|LE|GT|LT <*sortword*$_2$> | <*bracketed_expr2*>
<*sortword*> | <*bracketed_expr*> PASSES|FAILS <*1D-gate*>
<*sortword*> | <*bracketed_expr*> MASKEDBY <*16-bit mask*>
<*sortword*$_x$> | <*bracketed_expr*$_x$> # <*sortword*$_y$> | <*bracketed_expr*$_y$> # GATEDBY <*gate-expression*>
<*bracketed_expr*> is a normal <*expression*> surrounded in brackets e.g. (a+b)
and <*gate-expression*> is one of:
<*gate-name*>
<*arraylist-of-gates*> [< *expression* >]
where the gate dimension must match the number of arguments tested. The <*expression*> specified with the arraylist argument determines the offset into the arraylist, starting from zero, and hence the particular gate map referenced.

The **IF** command allows conditional execution of <*statements*$_1$> or <*statements*$_2$> depending on the result of a test. If the result is true, then <*statements*$_1$> are executed, otherwise <*statements*$_2$> are executed (if specified). The various forms of the **IF** command are described below in more detail.

Validation test operator (VALID)

IF <*sortword*> VALID ...

where **VALID** means `present in the event' or `set true by some previous statement in the sortfile', i.e. by an arithmetic assignment, **EXTRACT**, **LOOPEXTRACT** or **CALL** statement. For each such word, <*statements*$_1$> are executed.

Comparison operators (EQ,NE,GE,LE,GT,LT)

IF <*sortword*$_1$> | <*bracketed_expr*$_1$> EQ|NE|GE|LE|GT|LT <*sortword*$_2$> | <*constant*> | <*bracketed_expr*$_2$> ...

where
EQ denotes ``is equal to''
NE denotes ``is not equal to''
GE denotes ``is greater than or equal to''
LE denotes ``is less than or equal to''
GT denotes ``is greater than''
LT denotes ``is less than''
and <*constant*> may be an integer or real constant.

The first <*sortword*> or <*expression*> is compared with the second according to the operator specified,

Example

```
IF GAMA GT THRESHOLD {
    INC SPEC1(GAMA)
    INC SPEC2(TAC)
    }
ELSE  EVENTEND
```

In the above example, the spectrum increments are performed if the value of **GAMA** is greater than that of **THRESHOLD**. Otherwise event processing is terminated for that event (**EVENTEND** command).

Filtering operators (PASSES,FAILS)

IF <*sortword*> | <*bracketed_expr*> PASSES (<*lower-limit*> , <*upper-limit*>) ...

where expressions may be used to define <*lower-limit*> and <*upper-limit*> . **IF...PASSES** is true if <*sortword*> or <*expression*> falls inside the gate defined by <*lower-limit*> and <*upper-limit*> inclusive,

Example

```
IF GAMA PASSES (100 , HLIMIT)    INC SPEC1(GAMB)
```

causes spectrum **SPEC1** to be incremented if the value of sortword **GAMA** lies between 100 and the value of **HLIMIT** (inclusive).

If <*upper-limit*> is less than <*lower-limit*> then the **IF** test will always give the result FALSE.

IF <*sortword*> | <*bracketed_expr*> PASSES <*gate-array-name*> (<*index*>) ...

where <*index*> is an integer expression which gives the array element number used to obtain a gate defined in <*gate-array-name*> . See gate-array command.

Example

```
EXTRACT GELIST INTO $G1
IF $G1.TAC PASSES TACLIST($G1) {
   ...
```

IF <*sortword*> | <*bracketed_expr*> FAILS <*gate-array-name*> (<*index*>) ...

Conversely **IF...FAILS** is true only if <*sortword*> or <*expression*> is present in the event *and* falls outside the limits of the gate,

Example

```
IF GAMA FAILS (100 , HLIMIT) {
    DEC SPEC2(GAMB)
    DEC SPEC3(GAMC)
    }
```

causes spectra **SPEC2** and **SPEC3** to be decremented if sortword **GAMA** is outside the range defined by 100 and **HLIMIT** (inclusive).

Masking operator (MASKEDBY)

```
IF <sortword> | <bracketed_expr> MASKEDBY <16-bit mask-value> ...
```

IF...MASKEDBY is true only if all the bits set in the 16-bit mask are present in the sortword or expression being tested,

Example

```
IF GAMA MASKEDBY %00110101    INC SPEC1(MB1)
```

causes **SPEC1** to be incremented if all the bits set in **GAMA** are also set in the bit pattern **%00110101**.

```
IF <sortword> | <bracketed_expr> MASKEDBY <bitmask-set-name> ...
```

This form of the command is true if there is a bit pattern in <*bitmask-set-name*> for which all the bits set are present in <*sortword*> or <*expression*> . The gate number of the bit pattern which satisfies this condition is placed in the reserved word **GATE**.

Gate-testing operator (GATEDBY)

IF...GATEDBY is true if the value or <*sortword*> or <*expression*> falls within the set of gates associated with <*gate-expression*> . The gate number passed is placed in the reserved variable **GATE**. This variable is only recognised within <*statements*> . See section group name for definition of gate limits.

```
IF <sortword> | <bracketed_expr> GATEDBY <1D-gate-expr> ...
```

Example

```
IF GAMTOT GATEDBY  GLIST1 {
    INC SPEC1(GAMTOT)
    INC SUMSPEC(SUMEN)
    }
ELSE  EVENTEND
```

where the spectra **SPEC1** and **SUMSPEC** are incremented if sortword **GAMTOT** passes any of the 1D gates defined in the gate-map **GLIST1**. If no gates are passed then command processing is terminated for that event (**EVENTEND** command).

IF <*sortword*$_x$> | <*bracketed_expr*$_x$> <*sortword*$_y$> | <*bracketed_expr*$_y$> GATEDBY <*2d-gate-expr*> ...

In the 2D gate test command the gate is passed if the coordinates given by <*expr*$_x$> and <*expr*$_y$> fall within the set of 2D polygonal or elliptical gates specified in <*2d-gate-expr*> . The intersection of two polygons is given the gate number of the later defined gate.

Example

```
IF EDELTE SUMEN GATEDBY  GREC1 {
    INC MASS1(GAMTOT)
    SELECT(GATE)
       ... commands dependent upon which gate passed ...
    }
ELSE {
    IF EDELTE SUMEN GATEDBY GREC2
        INC MASS2(GAMTOT)
    }
```

If **EDELTE** (x-coordinate) and **SUMEN** (y-coordinate) pass any of the gates defined in the gate-map **GREC1** then the commands within the first set of braces will be executed, otherwise if they pass any of the gates defined in **GREC2** then the commands within the second set of braces will be obeyed:

Loopif...loopfail... command (parameter-list environment)}

LOOPIF <*test*> # NEWLISTX= <*list-name*> # # NEWLISTY= <*list-name*> # <*statements*$_1$> # LOOPFAIL <*statements*$_2$> #

where <*test*> is one of the following:
<*word-param-list*> VALID
<*$-word*> = <*group-param-list*> VALID
<*list-expression*> EQ|NE|GE|LE|GT|LT <*sortword*> | (<*expression*>)
<*list-expression*> PASSES|FAILS <*1D-gate*>
<*list-expression*> MASKEDBY <*16-bit mask*>
<*list-expression*$_x$> <*list-expression*$_y$> GATEDBY <*2d-gate-expression*>
<*list-expression*$_x$> <*sortword*$_y$> | (<*expression*$_y$>) GATEDBY <*2d-gate-expression*>
<*sortword*$_x$> | (<*expression*$_x$>) <*list-expression*$_y$> GATEDBY <*2d-gate-expression*>
and <*list-expression*> is either a word- or a group- parameter list:
<*word-parameter-list-name*>
<*$-word*> = # <*group-parameter-list-name*> . <*item-name*>
where for a group parameter list the particular group passed may be stored in the group variable $-word if specified.
<*gate-expression*> is one of:
<*gate-map-name*>
<*arraylist-of-gatemaps*> [< *expression* >]
where the gate dimension must match the number of arguments tested. The <*expression*> specified with the arraylist argument determines the offset into the arraylist, starting from zero, and hence the particular gate map referenced.

The **LOOPIF** command is a parameter-list form of the **IF** command, i.e. it executes an **IF** test for all words in an input parameter-list. Each time an item from the list (or item pair from the lists) is found that satisfies the test <*statements*$_1$> are executed.

If the input list is a word parameter list (See Createlist command) then for each test the current parameter value being tested is placed in the reserved variable **WORDX** (or **WORDY** for <*parameter-list-name*$_y$>).

For the **LOOPIF...MASKEDBY** <*bitmask-set-name*> and **LOOPIF...GATEDBY** commands, the gate number passed each time is placed in the reserved variable **GATE**.

If the **NEWLISTX** and/or **NEWLISTY** (for the second list in 2D gate-map case) keywords are used to specify output parameter lists then for each parameter which satisifes the test all the other parameters in the input list are copied to the specified output lists.

If no parameter-list items satisfy the test and **LOOPFAIL** has been specified then <*statements*$_2$> are executed.

Example

Consider the case where 2 group parameter lists are being tested together against a 2D gate-map. Each valid word in the first list is tested with all valid words in the second list:

```
CREATELIST LIST1 GE30
CREATELIST LIST2 GE150
LOOPIF LIST1.E2 LIST2.E2 GATEDBY GATES1  NEWLISTX=LIST1A  NEWLISTY=LIST2A
   INC MAT6(LIST1A.E2,LIST2A.E2)
```

i.e. all the combinations of **E2** words in lists **LIST1** and **LIST2** will be tested. For those combinations which pass any of the gates in the gate-map **GATES1** then the **INC** command will cause all permutations of the remaining **E2** words in the lists (**LIST1A** and **LIST2A**) to be incremented into the 2D spectrum **MAT6**.

Validation test operator (VALID)

LOOPIF <*word-param-list*> VALID ...

LOOPIF # <*$-word*> = # <*group-param-list*> VALID ...

For a word parameter list, <$statements_1$> are executed for each word in the list which is present in the event. For a group parameter list, <$statements_1$> are executed for each group in the list which is present in the event. If the optional " <*$-word*> =" is specified in the command then the group identifier is assigned to the **$-word** variable for each iteration of the loop.

Comparison operators (EQ,NE,GE,LE,GT,LT)

LOOPIF <*list-expression*> EQ|NE|GE|LE|GT|LT <*sortword*> | <*constant*> | <*expression*> ...

where
EQ denotes ``is equal to''
NE denotes ``is not equal to''
GE denotes ``is greater than or equal to''
LE denotes ``is less than or equal to''
GT denotes ``is greater than''
LT denotes ``is less than''
and <*constant*> may be an integer or real constant.

In this format of the **LOOPIF** command all words from <*parameter-list*> found in the event are compared with the value of <*sortword*> or <*expression*> according to the operator specified. For each case which gives the result true then <$statements_1$> are executed.

Example

```
CREATELIST  GELIST  FROM  GE
LOOPIF GELIST.E1 GT THRESHOLD {
    INC SPEC1(GAMA)
    INC SPEC2(TAC)
    }
LOOPFAIL  EVENTEND
```

In the above example, the spectrum increments are performed if the value of **GELIST.E1** is greater than that of sortword **THRESHOLD**. Otherwise event processing is terminated for that event (**EVENTEND** command).

Filtering operators (PASSES,FAILS)

LOOPIF <*list-expression*> PASSES|FAILS <*1D-gate*> (<*lower-limit*> , <*upper-lim-*

```
it> ) ...

LOOPIF <list-expression> PASSES|FAILS <gate-array-name> ( <index> ) ...
```

The first form of the command allows valid member of <*parameter-list*> to tested against the gate defined by the expressions <*lower-limit*> and <*upper-limit*> . In the second form of the command each valid member of <*parameter-list*> to tested against a gate in the array <*gate-array-name*> (See Gate-Array command) where the array element number used is given by the expression <*index*> .

Example

```
*DATA
GATEARRAY TACGATES
1 (100 4000)  2 (90 4000) ...
 ...
*COMMANDS
CREATELIST  GELIST  FROM  GE
LOOPIF $GROUPX=GELIST.TAC PASSES TACGATES($GROUPX) {
    ...
   }
```

LOOPIF...PASSES is true for each case where a member of <*parameter-list*> falls inside the gate defined by <*lower-limit*> and <*upper-limit*> inclusive. Conversely **LOOPIF...FAILS** is true for each case where a member of <*parameter-list*> is present in the event *and* falls outside the limits of the gate.

Masking operator (MASKEDBY)

```
LOOPIF <list-expression> MASKEDBY <16-bit mask-value or bitmask-
set-name> ...
```

LOOPIF...MASKEDBY is true for words in the list where all the bits set in the 16-bit mask or a gate in <*bit-mask-set-name*> are present in the word being tested. For the latter case the gate number passed is placed in the reserved word **GATE**.

Gate-testing operator (GATEDBY)

LOOPIF...GATEDBY is true for each case where a member of <*parameter-list*> falls within the set of gates associated with <*gate-expr*> . Whenever a gate is passed the gate number will be placed in the reserved variable **GATE**. The data defining the gate limits within the gate-map is specified in the ***DATA** section.

```
LOOPIF <list-expression> GATEDBY <1D-gate-expr> ...
```

In the above case each member of the parameter-list found in the event is tested against the gate-map specified,

Example

```
 ...
*DATA
GATEMAP 1D MASS130A[0:600]
```

```
(100 180) (160 270) (250 340) (330
 460) (440 560)
  ...
*COMMANDS
  ...
CREATELIST RLIST1 FROM GE
LOOPIF RLIST1.E2 GATEDBY MASS130A  NEWLISTX=RLIST2 {
      ...
    LOOPIF RLIST2.E2 GATEDBY ... {
           ...
        }
    }
```

LOOPIF <*list-expression*$_x$> <*list-expression*$_y$> GATEDBY <*2d-gate-expression*> …

LOOPIF <*list-expression*$_x$> <*sortword*$_y$> | (<*expression*$_y$>) GATEDBY <*2d-gate-expression*> …

LOOPIF <*sortword*$_x$> | (<*expression*$_x$>) <*list-expression*$_y$> GATEDBY <*2d-gate-expression*> …

For the 2D gate-map-test format one or both of the parameters to be tested must be a list-expression,

Example

```
  ...
*DATA
GATES 2D MAP1[64,64]
(12 20  20 22  23 26  19 31  17 27  16 23)   (19 24  25 21  30 27  31 38
29 40  26 38  23 26  21 24)   (35 28  38 32  40 33  42 41  40 45  38 41
37 39  36 32)
  ...
*COMMANDS
  ...
SUMEN = ...
CREATELIST GELIST FROM GE
LOOPIF GELIST.E2 SUMEN GATEDBY MAP1 NEWLISTX=OUTLIST1 {
    LOOPIF OUTLIST1.E2 GATEDBY ...
        ...
```

Select command

```
SELECT ( <expression> )
# ( <value> ) <statements> #r
(NOMATCH) <statements>

SELECT ( <expression1> , <expression2> )
# ( <value1> , <value2> ) <statements> #r
(NOMATCH) <statements>
```

The **SELECT** command is used for matching specific parameter values to specific sort commands. This allows gate numbers passed by any of the gate-map-test commands to be matched to specific update commands.

Each combination of values specified within the **SELECT** command must be specified only once but any number of sort commands can be associated with it.

The **SELECT** command evaluates the expression (or expressions) to obtain a value (or values) which is then compared with the sets of values following. If one set matches then any associated commands are executed. If no values match then the commands associated with **(NOMATCH)** are executed, if it has been specified.

After the sort commands for a specific combination have been executed command execution passes to the command following the **SELECT** command.

Example

Consider updates after a gate-map-test command. See if...gatedby and loopif...gatedby commands.

```
IF GAMA GATEDBY BAND1 {
      ...
    SELECT (GATE)
        (1) INC SPEC1(GAMA)
        (4) INC SPEC4(GAMA)
        (5) {
            INC SPEC5(GAMA)
            INC SPEC9(GAMA)
            }
       ...
    }
```

Example

or correlated updates after two gate-map-test commands:

```
IF GAMB GATEDBY BAND2 {
      ...
    SELECT(GATE1,GATE)
        (1,1) {
              INC MAT1(GAMA,GAMB)
              INC SPEC12(GAMB)
              }
        (1,2) INC MAT2(GAMA,GAMB)
        (NOMATCH) ENDEVENT
    }
```

In the second example, if no gate combination matches those provided by the **SELECT** command then ENDEVENT is executed so no more commands are processing for that event.

Goto command

```
GOTO <label-name>

LABEL <label-name>
```

The **GOTO** command allows event processing to be passed *forwards* only in the ***COMMANDS** section to a point specified using **LABEL**.

GOTO may not be used to jump into a **DOLOOP**, **IF** or **SELECT** command.

Example

```
IF GAMA GATEDBY GREC2
    GOTO BAND1
    ...
LABEL BAND1
    ...
```

Arithmetic operations

<*sortword*> = <*expression*>

<*array-name*> (<*x-index*>) = <expression>
<*array-name*> (<*x-index*> , <*y-index*>) = <*expression*>
<*array-name*> (<*x-index*> , <*y-index*> , <*z-index*>) = <*expression*>

where <*expression*> is
<*operand*> # <*operator*> <*operand*> #r
<*operand*> is either a <*sortword*> or a <*constant*> .
and <*x-index*> , <*y-index*> and <*z-index*>
may each be one of <*sortword*> or <*integer constant*>

Evaluated expressions may be assigned to a sortword variable, i.e. a word, long or float type variable.

C precedence determines the order in which operations are performed in the absence of parentheses. Up to 6 nested pairs of parentheses are allowed.

A floating point randomised value in the range 0.0 to 1.0 may be obtained by specifying the reserved float variable **RANDOM**. A random integer value in the range 0 to 32767 may be accessed using the reserved word **IRANDOM**.

Arithmetic Operators

+ - * / mod < > & ior xor
where
mod is the integer modulus operator (a mod b)
& ior xor are logical bitwise operators
< > are left and right arithmetic shift operators respectively which need to be followed by an integer value between 0 and 15
(to specify the number of places to the left or right by which the bits are to be shifted).

Maths functions

SQRT EXP ABS NOT LOG LOG10 SIN COS TAN ASIN ACOS ATAN

The argument should be specified in parentheses following the function name.

Example

```
... + EXP( A + ( B * X )) ...
```

Command Functions

```
GROUP <$-word>
```

returns the value of the group number of group identifier $-word e.g. **group($x)**.

```
NWORDS <$-word>
```

returns the number of items in the group identifier $-word.

```
POW ( <expression1> , <expression2> )
```

returns the value of <expression$_1$> to the power <expression$_2$> .

```
NUMBER ( <parameter-list-name> )
```

returns the length of the named list present in the current event.

```
<array-name> ( <x-index> )
<array-name> ( <x-index> , <y-index> )
<array-name> ( <x-index> , <y-index> , <z-index> )
```

returns the contents of an array location where the array must be initialised using **VALUEARRAY** in the ***DATA** section. Each channel can be specified by a sortword or integer constant.

```
NBIT ( <integer-expression> )
```

evaluates the number of bits set in the 16-bit expression.

```
<variable1> : <variable2>
<variable1> : <variable2> : <variable3>
<variable1> : <variable2> : <variable3> : <variable4>
```

evaluates 2 16-bit group-item parameters or sortwords as a 32-bit integer, and 3 or 4 16-bit words as a 64-bit integer (longlong).

```
TIMESTAMPOF ( <variable> )
```

evaluates the 64-bit absolute timestamp associated with <variable> , which has to be a raw data group-item parameter. The value is returned as a **LONGLONG** data type. Currently only supported for GREAT TDR format input data.

Gain command

GAIN <*sortword*> # FACTOR <*shift-factor*> #

GAIN <*sortword*> <*gain-array-name*> INDEXED <*index*> # FACTOR <*shift-factor*> #

GAIN <*sortword*> <*array-list-name*> [<*array-index*>] INDEXED <*index*> # FACTOR <*shift-factor*> #

GAIN <*group-param-list-name*> . <*item-name*> <*gain-array-name*> # FACTOR <*shift-factor*> #

GAIN <*group-param-list-name*> . <*item-name*> <*array-list-name*> [<*array-index*>] # FACTOR <*shift-factor*> #

where <*array-index*> and <*index*> are expressions.

Gain coefficients must be specified in the *DATA section.

A sortword can be gain matched using coefficients stored in a **GAINWORD** element (first format); the <*index*> th element of <*gain-array-name*> (second and third formats). For the third format, the gain array must be a member of an **ARRAYLIST** in the ***DATA** section, where <array-index> specifies the gain array used via it's index in the arraylist. Indices start from zero.

In the fourth format of the command the words in <group-param-list-name> are all gain-matched with the corresponding parameters in <gain-array-name> indexed by absolute group number, i.e. if the particular item in group *n* is present in the event it is gain matched with the n^{th} set of gain matching parameters defined in the gain array. The fifth format is similar to this but allows the particular gain array used to be selected via an index into an arraylist where <array-index> can be an expression.

An expression may also be supplied as an optional <shift-factor> argument to be applied to each of the gain coefficients. This is useful for making a Doppler shift correction when the original gain coefficients have been derived from source measurements.

Example

The value of a word is modified according to:

```
<word>  = a + b*  <word>  + c*  <word> 2
```

The calculation includes a randomisation process which adjusts the result by at most one channel in order to produce a smooth function.

Gain drifts may be automatically adjusted by adding an *AUTOGAIN section.

Example

```
*FORMATS
ge[1:35] (e20,e4,ft,co,bgoe,bgot,hitpat)
*DATA
GAINARRAY gegains
1  (-.3 0.09 0.004)
2  (0.6 0.10 0.002)
  ...
```

```
*COMMANDS
CREATELIST gelist FROM ge
GAIN gelist.e4 gegains
```

will gain match all the **E4** adcs associated with each **GE** group number.

Example

```
*FORMATS
clovers1[81:110]  (bgop, A1, A2tag:3,A2dat:13, A3)
*DATA
GAINARRAY segA                                        !! segment A
    81     (-6.4294434      0.9705133    0.0000000)
    82     (-1.8133545      0.9134621    0.0000000)
    83     ( 2.0946655      1.0252383    0.0000000)
  ...
*COMMANDS
CREATELIST clist1 FROM clovers1
loopif $c1=clist1 valid
   {
   groupno = group($c1)
   inc clgroups(groupno)
   select ($c1.a2tag)                  !! check tag to see which seg fired
   (1)  {                                    !! segment A
        gain $c1.a2dat segA indexed $c1
        }
   (2)  {                                    !! segment B
        gain $c1.a2dat segB indexed $c1
        }
   (3)  {                                    !! segment C
        gain $c1.a2dat segC indexed $c1
        }
   (4)  {                                    !! segment D
        gain $c1.a2dat segD indexed $c1
        }
   }
```

will gain match all the **a2dat** adcs associated with each **clovers1** group number, where the gainarray used is found by checking **a2tag** to determine which element of the detector has fired.

Example

```
CREATELIST qlist FROM clover
LOOPIF $q=qlist VALID
 {
 detid = GROUP($q) - 30
 capmult = NWORDS($q)/3
 SELECT (capmult)
 (1) {                                      ! Single hit
     capid = $q.Atag
     theta = qthetas(detid,capid)
     gfac  = (2.0)/((1.0)+(cos(theta)*beta))
     GAIN $q.Agehigh qgains[detid] INDEXED capid FACTOR gfac ! GM ener
     GAIN $q.Agetime qgainst[detid] INDEXED capid      !gain match TAC
     INC esum($q.Agehigh)
     }
 (2) {                                      ! Double hit
     capid0 = $q.Atag
     capid1 = $q.Btag
     theta0 = qthetas(detid,capid0)
     theta1 = qthetas(detid,capid1)
     theta2 = (theta0 + theta1)/2.0
     costh  = cos(theta2)
     gfac = 2.0/(1.0 + costh*beta)
```

```
    GAIN $q.Agehigh qgains[detid] INDEXED capid0 FACTOR gfac
    GAIN $q.Bgehigh qgains[detid] INDEXED capid1 FACTOR gfac
    GAIN $q.Agetime qgainst[detid] INDEXED capid0     !gain match TAC
    GAIN $q.Bgetime qgainst[detid] INDEXED capid1     !gain match TAC
    INC esum($q.Agehigh + $q.Bgehigh)
    }
}
```

will gain match and Doppler correct all single and double hit energies and times for the group **clover**, updating the shifted energies into the spectrum **esum**.

Invalidate command

```
INVALIDATE <group-identifier>

where <group-identifier> is:
<group-name> or <$-group-variable>
```

The **INVALIDATE** command removes all the items associated with the <*group-identifier*> specified from the current event.

Example

```
INVALIDATE   GE[12]
```

would remove group 12

Example

```
INVALIDATE   RMS
```

would remove the **RMS** group

Example

```
INVALIDATE   $GROUPX
```

would remove the group referenced by **$GROUPX**.

Groupfilter command

GROUPFILTER <*group-name*> # FIXEDLEN= <*length*> # # VARLEN= <*length*> # # ITEM= <*item-offset*> # <*filter*>

where <*group-name*> is defined here.
FIXEDLEN and VARLEN require integer values to define the fixed and variable number of words in the group definition. If omitted they default to zero.
<*item-offset*> is the index of the item in the group, starting from zero, that is being tested and <*filter*> is:
KEEP | REJECT (<*expr*> , <*expr*>)

The **GROUPFILTER** command removes all the items associated with each member of <*group-name*> from the current event that do not satisfy the condition specified by <*filter*>

Example

```
GROUPFILTER   CLUST  VARLEN 3 ITEM 1 REJECT (0,0)
```

would remove all members of group **CLUST** where the 2nd item for each subset of 3 items was zero.

Order command

ORDER $word_1$ $word_2$ ... *UP*|*DOWN*

where UP outputs the values in increasing numerical order
and DOWN outputs them in decreasing numerical order.

The **ORDER** command orders sortwords according to their value.

Example

```
ORDER  GAM1 GAM2 GAM3  DOWN
```

will reassign the highest value from **GAM1**, **GAM2** and **GAM3** to **GAM1**, the next highest to **GAM2** and the lowest to **GAM3**.

Routines

CALL <*routine-name*> # (<*argument*> # <*argument*> #r) #

ROUTINE <*routine-name*> # (<*argument*> # <*argument*> #r) #

where <*argument*> is a sortword, i.e. a
WORD , LONG , LONGLONG or FLOAT type.

A **ROUTINE** is a set of sort commands which are physically placed after the main command section and accessed by a **CALL** command within the main section. The last command in a **ROUTINE** should be **END** or **ENDEVENT**. **END** returns event processing to the command following the **CALL** command. when the routine has been executed whereas **ENDEVENT** terminates processing of the current event at that point in the sortfile.

A routine must be called at least once before it is defined in the sortfile so that the argument types can be determined, i.e. word, long or float, before the routine is specified. The current maximum number of arguments that may be passed to a routine may be found in Appendix A. If a sortword argument is specified any operation performed on that parameter within the routine will result in a corresponding change in value of the sortword upon returning from the routine,

Example

```
IF GELI3 GATEDBY GLIST2 {
    SELECT(GATE) {
        (1) CALL ABC(SUMEN,INDEX)
        (2) CALL XXXXYYYY
            ...
    }
}
END
ROUTINE ABC(ENERGY,OFFSET)
    ...
CALL XXXXYYYY
    ...
END
ROUTINE XXXXYYYY
    ...
END
```

where **SUMEN** and **INDEX** are sortwords, **ENERGY** and **OFFSET** are dummy sortword arguments local to the routine **ABC**.

Calls may be nested up to a maximum level of 8, but routines must not be called recursively, i.e. a routine may not call itself or one that directly or indirectly calls it. A routine must be called at least once before it is specified, i.e. at least one **CALL** statement for a given routine must occur before the **ROUTINE** statement in the sortfile.

Exec Command

EXEC <*function-name*> [< *sortarg* >] r # INIT [< *initarg* >] r #

where
<*function-name*> is the name of the external command,
<*sortarg*> r is a list of runtime arguments from the following list, which are passed as the variable address...
<*sortword*> , <*spectrum*> , <*indexed-spectrum*> ,map, array, <*group-name[number]*> ,
<*initarg*> r is a list of initialisation arguments to the function, e.g. filenames, that are passed asis without change.
Note: Newlines are not allowed within this command.

Deprecated command ...
USER <*function-name*> ([< *sortarg* >] r)

where <*function-name*> is the name of the external command, which will be forced to be UPPER-case.
<*sortarg*> r is a list of runtime arguments, e.g. sortwords, spectrum names, which are passed as the variable address.

The **EXEC** command allows externally defined subroutines to be accessed from the sort. It allows different sorting and storage algorithms to be used via the sort language.

<*function-name*> is a string by which the new external routine is known. The arguments associated with <*function-name*> are specific to the corresponding routine. <*function-name*> will be forced to be LOWER-case, see example below.

The return code of the routine is tested for success. A non-zero return code will stop processing the current event at that point.

There are two sets of arguments, associated with the two routines defined below in the C language ...

<*function-name*> (<$sortarg_1$> , ...)

<*function-name*> _init (<$initarg_1$> , ...)

The <*function-name*> _init routine is optional, and will only be executed if the **init** section of the **exec** command is specified. If present, the init routine will be executed once only before any events are processed. The current maximum number of arguments that may be passed to an **exec** routine may be found in Appendix A.

Example

```
...
exec printit sortword
...
```

will require the following C code ...

```
int printit(short *sortword)
{
   printf("Sortword value = %d\n",*sortword);
   return 0;
}
```

Example

To output some information to a file, then the following code can be used to place the file in the sort directory. Note: Be aware that outputting every event may produce a large file.

```
   ...
   exec fileit sortword
   ...
```

will require the following C code ...

```
extern char basedir[];
static char filename[128];
static FILE *fp;

int fileit_init()
{
   strcpy(filename,basedir);
   strcat(filename,"/myfile");
   fp = fopen (filename, "w"));

   return 0;
}

int fileit(short *sortword)
{
   fprintf(fp, "%d\n", *sortword);
   return 0;
}
```

Synchronization

If it is necessary to execute a routine to tidy up after the sort has finished, but not yet exited, a hook mechanism is provided for that purpose. This allows a user-provided routine to be executed at the sort program flushing stage immediately before exit. The mechanism to connect the routine into the sort program is to execute a routine called **flush_hook_add()** in the <function-name> _init routine.

Such a mechanism could be useful if the current state or a set of variables needs to be saved.

<*flush_hook_add*> (<function-name> _tidy)

Example

```
   ...
   exec calculate sortword init
```

```
   ...
```

will require the following C code ...

```
calculate_tidy()
   {
      /* Tidy up calculate before exit */
      ...
      return 0;
   }
calculate_init()
   {
      flush_hook_add( calculate_tidy() );
      return 0;
   }
```

Doloop command

DOLOOP <*loop-count*> FROM <*initial-loop-value*> TO <*final-loop-value*> STEP <*loop-step-size*>

where <*loop-count*> is a sortword, and integer or sortword values may be used to specify the loop initial, final and step values.
DOLOOP commands may be nested.

The **DOLOOP** command allows <*statements*> to be executed a defined number of times with an incrementing variable. The loop will always be executed at least once since the loop-count variable will be incremented at the end of each loop. This variable, if omitted, is an automatically created word named **LOOP**. The variable may be used freely within the loop,

Example

```
DOLOOP LOOP1  FROM 1  TO 8  STEP 2
    {
      ...
    NEWWORD = POSITION * LOOP1
    INC NEWWORD POSSPEC
      ...
    }
```

will execute the commands within curly braces for values of the word **LOOP1** of 1,3,5 and 7.

Example

The loop values may be negative ...

```
DOLOOP  INDEX FROM 7  TO -2  STEP -3
```

will execute the contained commands for values of the variable **INDEX** of 7,4,1 and -2.

Example

To exit from a loop before the loop variable has reached the final loop value the **IF...GOTO** command should be used ...

```
DOLOOP  FROM X1  TO X2  STEP I  {
      ...
    IF ...
        GOTO ABCD
          ...
   }
LABEL ABCD:
     ...
```

Output command

The **OUTPUT** command allows whole events or data words, 16-bit sortwords and parameter lists to be output on up to 4 different streams. The output <*stream number*> must be an integer value between 1 and 4 (inclusive). The data will automatically be output in Eurogam-style format with an event-header, etc. per event.

OUTPUT <*stream number*> EVENT

This form of the command will output all elements of the event as defined in the ***FORMATS** section. Note that any data items in the event, but not defined in the ***FORMATS** section, will not be output.

Although several such statements can be included (e.g. within **if** clauses), only the first statement reached will be executed.

Example

```
SELECT (I)
(1) OUTPUT  1  EVENT
(2) OUTPUT  2  EVENT
(3) OUTPUT  1  EVENT
```

OUTPUT <*stream number*> <*output-parameter*>

where <*output-parameter*> may be one of the following:
<*single-parameter-word*>
<*group-parameter-word*>
<*group-parameter-list*>
<*group-parameter-word*> (<*item-list*>)
<*group-parameter-list*> (<*item-list*>)
where <*item-list*> is a subset of the original item list associated with the group that was declared in the *FORMATS section.

This command can be used to output partial event components and generated simple variables.

Example

```
*FORMATS
TRIG[255] (MUSER,MTAC)
GE[1:54] (E1,E2)
*COMMANDS
OUTPUT  1   TRIG(MTAC) GE(E2)
```

If **WORD** type variables are output in the format specified above they must also be defined with an associated address to make them simulate real ADCs. See Sortwords section.

Note

If any commands have been used to alter any event parameters, e.g. **GAIN**, **INVALIDATE** or arith-

metic operations, prior to **OUTPUT EVENT** then the altered values will be output.

OUTPUT <*stream number*> GROUP <*group number*> ([< *item-list* >] r)

where <*group number*> is an integer or sortword.
where <*item-list*> is a list of sortwords.

Example

```
OUTPUT  1 GROUP 234  ( A, B)
```

This command can be used either to output an existing group with a modified item list, or a new group.

Care must be taken that group numbers are not duplicated within an event. The following example is illegal if group 15 is present in the raw data, but would not be checked for by the compiler.

Example

```
OUTPUT  1  EVENT
...
OUTPUT  1  GROUP 15 (A, B, C)
```

Endevent command

```
ENDEVENT
```

ENDEVENT terminates event processing at this point in the sortfile. It may be used anywhere in the commands section. See also the Routine command.

End command

```
END
```

END is used to specify the end of the main command section prior to any routines or the end of a routine. If used at the end of a routine then during execution event processing will pass to the command following the point from where the routine was called after the routine has been executed.

Example

```
*COMMANDS
    ...
END
    ...
```

Example

```
*COMMANDS
    ...
END
ROUTINE
    ...
END
ROUTINE
    ...
END
    ...
```

Pause command

```
PAUSE
```

The **Pause before each event** feature in the Run window causes a temporary halt in event processing at a point just before executing the first event command. Pressing the **resume** button allows event processing to occur upto the next pause point. If the **Pause before each event** button is unchecked, then **resume** will allow event processing to continue without further pauses.

The **pause** command works in the same way as the **Pause before each event** feature in the Run window, except that processing is paused at the point the **pause** command is executed. It may be used anywhere in the commands section.

If the **pause** command is placed inside a conditional statement, then the pause may be used to inspect spectra and variable values after some defined set of circumstances. Hence it can be a useful diagnostic aid.

The pause can be turned on and off by setting a global sortword in the *DATA section. See the example below which uses the sortword **pauseflag**.

Example

```
*COMMANDS
    ...
if pauseflag gt 0 pause
    ...
END
    ...
```

*RUNFILES (offline analysis only)

Statements in this section allow input tape or disc files to be specified.

<*tape-volume-name*> <*file-name*> # <*start-block*> # <*finish-block*> # #

<*tape-volume-name*> <*file-specifier*> # , <*file-specifier*> #

where
<*tape-volume-name*> is the tape name or label
<*file-name*> is the name of a file to be sorted on the tape
and <*file-specifier*> can be:
<*file-name*>
<*file-pattern*>
or a range:
<$file\text{-}name_1$> - <$file\text{-}name_2$>
which will sort all files from <$file\text{-}name_1$> to <$file\text{-}name_2$> inclusive.
If <*start-block*> is greater than zero, that number of blocks will be skipped at the start of the file. If <*finish-block*> is greater than <*start-block*> , processing will stop at that point, else continue to end of file.

<*file-pattern*> may consist of the wildcard characters ``*" to match any character combination and ``?" to match a single wildcard character,

e.g. CAL* to sort any filename beginning with the letters ``CAL", RUN2? to sort any filename beginning with the characters ``RUN2" and followed by one further character.

A large subset of the characters defined in the ANSI tape standard X3.27-1987 are recognised:

alphanumeric (A to Z, 0 to 9)

and the following non-alphanumeric characters:

" % & ' ( ) + - . / : ; < = > \ _

Names must start with an alphanumeric character.

The volume name is contained in a field 6 characters long and the file name contained in 17 characters. For non-ANSI format tapes, e.g. ones with no file headers, filenames of RUNxx should be used where xx denotes the file number on tape. For unlabelled tapes, the same convention using TAPExx should be used to distinguish between different volumes.

Example

```
*RUNFILES
SN001 RUN01
SN002 RUN02-RUN04, RUN06-RUN15, RUN17, *
SN003 RUN21 5000 9999999
SN003 RUN22
SN003 RUN23
SN004 *
   ...
```

In this example the whole of file **RUN01** on tape **SN001** would be sorted, followed by files **RUN02** up to **RUN04**, **RUN06** up to **RUN15**, and **RUN17** onwards on tape **SN002**. Any files not included in the specified

ranges, e.g. **RUN05**, are omitted. File **RUN21** on tape **SN003** is sorted from block 5000 onwards, followed by files **RUN22** and **RUN23** and the whole of tape **SN004**.

Disc files may be specified as follows:

DISC <*file-name*> # <*start-block*> # <*finish-block*> # #
where
<*file-specifier*> can be:
<*file-name*>
<*file-pattern*>
If <*start-block*> is greater than zero, that number of blocks will be skipped at the start of the file. If <*finish-block*> is greater than <*start-block*> , processing will stop at that point, else continue to end of file.

<*file-pattern*> may consist of the wildcard characters ``*" to match any character combination and ``?" to match a single wildcard character, in the same way as for tapes above. If a full disc-file pathname is given, then wild-card characters may only appear in the file-name, and not in the directory name(s).

Example

```
*RUNFILES
DISC /disc1/calib/eu152/run1
DISC /disc1/calib/eu152/run3 1 100
DISC /disc1/calib/eu152/run2*
```

Constraints

This section contains lists of reserverd words and maximum array values.

Reserved words

This is the list of names that should not be used for user-defined arrays, spectra or sortwords.

ABS
ACOS
ANGLES
ARRAYLIST
ASIN
AT
ATAN
CALL
CENTROIDS
COPYGAIN
COS
CREATELIST
DEC
DELTAS
DEVIATION
DISC
DOLOOP
DOWN
ELLIPSE2D
ELLIPSE3D
ELSE
END
ENDEVENT
EQ
EVENT
EXEC
EXP
EXTRACT
FAILS
FACTOR
FIXEDLEN
FLOAT
FROM
GAIN
GAINARRAY
GAINWORD
GATEARRAY
GATEDBY
GATEMAP
GATES
GOTO
GROUP
GROUPFILTER
GT
IF
INC
INCBITS
INDEXED
INDEXED
INDEXED
INIT
INTO
INVALID
INVALIDATE
IOR
IRANDOM
ITEM
KEEP
LABEL
LE
LOG
LOG10
LONG
LOOPEXTRACT
LOOPFAIL
LOOPIF
LT
MASK
MASKEDBY
NBIT
NE
NEWLISTX
NEWLISTY
NOT
NOMATCH
NUMBER
NWORDS
OFFSET
ORDER
ORDERED
OUTPUT
PASSES
PEAKAREA
PEAKS
POW
RANDOM
REJECT
REVERSED
ROUTINE
SAMPLE
SAVE
SELECT
SET
SIN
SQRT
STEP
TAN
TO
UP
USER
VALUEARRAY
VALID
VALUE
VARLEN
VOVERC
WORD
XOR

Predefined sortwords

These are sortwords that are automatically defined for every sort. They may be used in the same way as normal sortwords.

STREAM
GATE
WORDX
WORDY
BLOCK_NUMBER
RUNFILE_NUMBER

Maximum values

Name length	30
Number of Spectra	32768
Disc-based update Spectra	4096
Spectrum length per dimension	65536
Number of gain names	512
Number of gate names	512
Number of arrays	512
Number of arraylists	512
Number of single adcwords	200
Number of groups	2048
Number of sortwords	512
Number of lists	64
Number of routines	64
Number of routine args	8
Number of exec/user args	64
Number of runfiles	255
Files per wildcard expansion	4096
Spectrum memory space 32bit	~1.5GB
Spectrum memory space 64bit	at least 4GB

Data file examples

Eurogam phase 2 autogain sort

A typical autogain sort for phase 2...

```
*FORMATS
ge[1:35] (e20,e4,ft,co,bgoe,bgot,hitpat)
clovers1[81:104]  (bgop, A1, A2tag:3,A2dat:13, A3)
clovers2[111:134] (bgop, A1, A2tag:3,A2dat:13, A3,
                         B1, B2tag:3,B2dat:13, B3)
*DATA
!! gain coefficients calculated
!! from Eu152 source spectra for v/c=0.0105
GAINARRAY  gphase1       !! phase1 detectors
  1 (3.054180 0.985850 0.000000)
  2 (4.106800 1.053940 0.000001)
  3 (3.101520 0.888870 0.000000)
     ...
GAINARRAY  segA       !! segment A
 81 (-5.956440 0.968458 0.000000)
 82 (-2.043540 0.912272 0.000000)
 83 (1.866570 1.023430 0.000000)
     ...
GAINARRAY  segB       !! segment B
 81 (-4.002160 0.922770 0.000000)
 82 (-4.108340 0.908958 0.000000)
 83 (-3.820860 1.028980 0.000000)
     ...
GAINARRAY  segC       !! segment C
 81 (-0.804466 0.927298 0.000000)
 82 (-2.431520 0.922218 0.000000)
 83 (-1.492300 1.027120 0.000000)
     ...
GAINARRAY  segD       !! segment D
 81 (-8.195100 0.950084 0.000000)
 82 (0.832342 0.884106 0.000000)
 83 (-1.531620 1.039610 0.000001)
     ...
GAINARRAY segAmod
GAINARRAY segBmod
GAINARRAY segCmod
GAINARRAY segDmod

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!
!! EUROGAM PHASE2 ARRAY CLOVER ANGLES
!! USED BY AUTOGAIN ROUTINE TO CORRECT GAIN COEFFICIENTS TO ADDBACK
!! CLOVER DATA
!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
VALUEARRAY eurogam[81:104]    !! clover midpoint angles
 104.5  75.5 104.5  75.5 104.5  75.5 104.5  75.5 104.5  75.5 104.5  75.5
  75.5 104.5  75.5 104.5  75.5 104.5  75.5 104.5  75.5 104.5  75.5 104.5
VALUEARRAY deltAB[81:104]     !! delta angles for segments A and B
   4.5  -4.5   4.5  -4.5   4.5  -4.5   4.5  -4.5   4.5  -4.5   4.5  -4.5
  -4.5   4.5  -4.5   4.5  -4.5   4.5  -4.5   4.5  -4.5   4.5  -4.5   4.5
VALUEARRAY deltCD[81:104]     !! delta angles for segments C and D
  -4.5   4.5  -4.5   4.5  -4.5   4.5  -4.5   4.5  -4.5   4.5  -4.5   4.5
   4.5  -4.5   4.5  -4.5   4.5  -4.5   4.5  -4.5   4.5  -4.5   4.5  -4.5

*SPECTRA
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!
!! DECLARE SPECTRA FOR USE IN *AUTOGAIN SECTION
!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
xautog[1:35] 4096 32
xautoa[81:104] 4096 32
xautob[81:104] 4096 32
xautoc[81:104] 4096 32
xautod[81:104] 4096 32

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!
!! DECLARE SPECTRA FOR USE IN *COMMANDS SECTION
```

```
!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
gamtot1 4096 32
gamtot2 4096 32
cl2ab 4096 32
cl2ac 4096 32
cl2ad 4096 32
cl2bc 4096 32
cl2bd 4096 32
cl2cd 4096 32

*AUTOGAIN
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!
!! NUMBER OF BLOCKS AFTER WHICH TO CALCULATE GAINS OR CHECK DRIFTS
!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
sample 15000

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!
!! CHANGE PEAK VALUES TO CORRESPOND TO 2 PEAK POSITIONS IN YOUR DATA
!! ...FOR 4MeV GAIN COEFFICIENTS
!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
init gphase1 from xautog centroids  393.0 8.0  735 10.0
init segA from xautoa centroids     393.0 8.0  735 10.0
init segB from xautob centroids     393.0 8.0  735 10.0
init segC from xautoc centroids     393.0 8.0  735 10.0
init segD from xautod centroids     393.0 8.0  735 10.0

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!
!! VOVERC FOR EXPERIMENT (0 < voverc < 1.0)
!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
voverc 0.0105

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!
!! CORRECT GAINS TO USE FOR ADDBACK WHENEVER AUTOGAINED COEFFICIENTS
!! ARE SHIFTED
!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
copygain from segA[81:104] to segAmod[81:104] angles eurogam deltas deltAB
copygain from segB[81:104] to segBmod[81:104] angles eurogam deltas deltAB
copygain from segC[81:104] to segCmod[81:104] angles eurogam deltas deltCD
copygain from segD[81:104] to segDmod[81:104] angles eurogam deltas deltCD

createlist gelist from ge
inc xautog($auto=gelist.e4) indexed $auto

createlist clist1 from clovers1
loopif $c1=clist1.a2dat valid
   {
   select ($c1.a2tag)
   (1)  {                                    !! segment A
        inc xautoa($c1.a2dat) indexed $c1
        }
   (2)  {                                    !! segment B
        inc xautob($c1.a2dat) indexed $c1
        }
   (3)  {                                    !! segment C
        inc xautoc($c1.a2dat) indexed $c1
        }
   (4)  {                                    !! segment D
        inc xautod($c1.a2dat) indexed $c1
        }
   }

!!-------------------------------------------------------------------
!!
!! THESE COMMANDS ARE EXECUTED FOR EVERY EVENT
!!
!!-------------------------------------------------------------------
*COMMANDS
```

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!
!! GAIN MATCH PHASE 1 DETECTORS
!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
gain gelist.e4 gphase1

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!
!! GAIN MATCH CLOVER DETECTORS WITH SINGLE HITS
!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
loopif $c1=clist1.a2dat gt 0
   {
   groupno = group($c1)
   select ($c1.a2tag)
   (1)  {                                    !! segment A
        gain $c1.a2dat segA indexed $c1
        }
   (2)  {                                    !! segment B
        gain $c1.a2dat segB indexed $c1
        }
   (3)  {                                    !! segment C
        gain $c1.a2dat segC indexed $c1
        }
   (4)  {                                    !! segment D
        gain $c1.a2dat segD indexed $c1
        }
   }

loopif $c1=clist1.a2dat gt 0 newlistx=clist1a
   {
   inc gamtot1($c1.a2dat)
   }

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!
!! GAIN MATCH CLOVER DETECTORS WITH DOUBLE HITS
!!
!!      simple coeffs    : A-B, C-D
!!      corrected coeffs : A-C, A-D, B-C, B-D
!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
createlist clist2 from clovers2
loopif $c2=clist2.a2dat gt 0
   {
   if ($c2.b2dat) gt 0
      {
      gaingrp = group($c2) - 30
      sum = 0
      select ($c2.a2tag,$c2.b2tag)
      (1,2)  {                                    !! A-B
             gain $c2.a2dat segA indexed gaingrp
             gain $c2.b2dat segB indexed gaingrp
             sum = $c2.a2dat + $c2.b2dat
             inc cl2ab(sum)
             }
      (1,3)  {                                    !! A-C
             gain $c2.a2dat segAmod indexed gaingrp
             gain $c2.b2dat segCmod indexed gaingrp
             sum = $c2.a2dat + $c2.b2dat
             inc cl2ac(sum)
             }
      (1,4)  {                                    !! A-D
             gain $c2.a2dat segAmod indexed gaingrp
             gain $c2.b2dat segDmod indexed gaingrp
             sum = $c2.a2dat + $c2.b2dat
             inc cl2ad(sum)
             }
      (2,3)  {                                    !! B-C
             gain $c2.a2dat segBmod indexed gaingrp
             gain $c2.b2dat segCmod indexed gaingrp
             sum = $c2.a2dat + $c2.b2dat
             inc cl2bc(sum)
             }
      (2,4)  {                                    !! B-D
             gain $c2.a2dat segBmod indexed gaingrp
```

```
            gain $c2.b2dat segDmod indexed gaingrp
            sum = $c2.a2dat + $c2.b2dat
            inc cl2bd(sum)
            }
     (3,4)  {                                    !! C-D
            gain $c2.a2dat segC indexed gaingrp
            gain $c2.b2dat segD indexed gaingrp
            sum = $c2.a2dat + $c2.b2dat
            inc cl2cd(sum)
            }
     inc gamtot2(sum)
     }
  }
...
```

Auto-gained correlation sort

This example illustrates an offline angular correlation sort where the germaniums have been divided into separate groups dependent on the angle of the detector in the array. The germanium data is auto-gain matched.

```
*FORMATS
TRIGGER[255] (TYPE,MRAW,MSUP)
GERM158[1:5] (GE20,GE4,GET,GETBD)
GERM134[6,11,13,17,19,23,25,29,35,36] (GE20,GE4,GET,GETBD)
GERM108[8,10,14,16,20,22,26,28,32,34] (GE20,GE4,GET,GETBD)
GERM90[12,15,21,27,33,37,40,46,49,52] (GE20,GE4,GET,GETBD)
GERM72[38,39,41,42,44,45,47,48,50,51] (GE20,GE4,GET,GETBD)

*DATA
GAINARRAY E4GAINS

*SPECTRA
GAINSP[1:54] 4096 32
GG158158 3000 2D
GG90158 3000 2D
GG134134 3000 2D
GG90134 3000 2D

*AUTOGAIN
SAMPLE 20000
PEAKAREA 50
DEVIATION 1.0
INIT E4GAINS FROM GAINSP CENTROIDS 331.4 3 891.0 4
PEAKS
1 654 10 1758 15
2 634 10  1723 15
3 532 10  1440 15
4 607 10  1637 15
   ...
48 596 10  1585 15
49 671 10  1811 15
50 605 10  1627 15
51 929 10  2257 15
52 654 10  1765 15
CREATELIST GE158 FROM GERM158
INC GAINSP($A=GE158.GE4) INDEXED $A
CREATELIST GE134 FROM GERM134
INC GAINSP($B=GE134.GE4) INDEXED $B
! ignore this angle
!CREATELIST GE108 FROM GERM108
!INC GAINSP($C=GE108.GE4) INDEXED $C
CREATELIST GE90 FROM GERM90
INC GAINSP($D=GE90.GE4) INDEXED $D
! ignore this angle
! CREATELIST GE72 FROM GERM72
! INC GAINSP($E=GE158.GE4) INDEXED $E

*COMMANDS
GAIN GE158.GE4 E4GAINS
GAIN GE134.GE4 E4GAINS
! GAIN GE108.GE4 E4GAINS
GAIN GE90.GE4  E4GAINS
! GAIN GE72.GE4  E4GAINS

LOOPEXTRACT GE90.GE4 INTO RIGHT
{
 LOOPEXTRACT GE158.GE4 INTO BACK
 {
  INC GG90158(BACK,RIGHT)
 }
}
LOOPEXTRACT GE158.GE4 INTO BACK1 BACK2
{
 INC GG158158(BACK1,BACK2)
 INC GG158158(BACK2,BACK1)      ! make symmetric
 LOOPEXTRACT GE90.GE4 INTO RIGHT
 {
  INC GG90158(BACK1,RIGHT)
  INC GG90158(BACK2,RIGHT)
```

```
 }
}

LOOPEXTRACT GE90.GE4 INTO RIGHT
{
 LOOPEXTRACT GE134.GE4 INTO BACK
  {
   INC GG90134(BACK,RIGHT)
  }
}
LOOPEXTRACT GE134.GE4 INTO BACK1 BACK2
{
 INC GG134134(BACK1,BACK2)
 INC GG134134(BACK2,BACK1)      ! make symmetric
 LOOPEXTRACT GE90.GE4 INTO RIGHT
 {
  INC GG90134(BACK1,RIGHT)
  INC GG90134(BACK2,RIGHT)
 }
}


END

*RUNFILES
XE122 RUN1
XE122 RUN2
*FINISH
```

Quadsort

This example illustrates an offline double gated sort updating a 2D matrix. The update algorithm is designed to produce spikeless spectra when slices and projections of the matrix are made. The data has been compressed so that all groups have a single item represented the gain matched energy value. The tag bits have been preserved in the first 3 bits of each data word.

```
! compressed data sortfile
! double gates updating 2D matrix
!
*FORMATS
gam[1:134] (tag:3, e4:13)

*DATA

!! gates for energy=e4/2
GATEMAP 1D gates1 [2050]
( 672  680)  (1165 1175)  (1346 1358)
(1474 1486)  (1544 1556)  (1686 1698)
(1566 1578)  (2036 2048)  (998 1006)
(1239 1249)  (1433 1443)

*SPECTRA
!== sort spectra ======
mat2d   4096 2D

*COMMANDS

CREATELIST gamlist FROM gam

LOOPIF $c=gamlist.e4 VALID
   $c.e4=$c.e4/2

if (NUMBER(gamlist)) LT 4
   ENDEVENT

!! double gated 2D update
!!
!! Use LOOPIF to decide whether the event satisfies 2, 3 or 4 gates
!! Then loop over the appropriate words in event to update matrix
!!  and exit loop
!!
!! Update Algorithm:
!! For m-dim update and p gates, words which satisfy gates are g parameters,
!! all others are x parameters.  Have 3 possible cases:
!!  1. satisfy exactly p gates -- update m-tuples from x params
!!  2. satisfy at least p+m gates -- update m-tuples from g+x params
!!  3. satisfy p+k gates, k<= m -- update m-tuples which involve <= k g params
}
!!
!! for this case:
!!  1. satisfy exactly 2 gates -- update doubles from listp2
!!  2. satisfy at least 4 gates -- update doubles from gamlist
!!  3. satisfy exactly 3 gates -- update g params with singles from listp3
!!
LOOPIF gamlist.e4 GATEDBY gates1 NEWLISTX=listp1
   {
                                                          !! >=1 gate
   LOOPIF listp1.e4 GATEDBY gates1 NEWLISTX=listp2
      {
                                                          !! >=2 gates
      LOOPIF listp2.e4 GATEDBY gates1 NEWLISTX=listp3
         {
                                                          !! >=3 gates
         LOOPIF listp3.e4 GATEDBY gates1
            {
                                                          !! >=4 gates
!! ...at least 4 gates satisfied in gamlist -- update all parameters
            INC mat2d(gamlist.e4,gamlist.e4)
            GOTO endloop1
            }
         LOOPFAIL
            {
```

```
                                                                  !! =3 gates
!! ...no gates satisfied in listp3
            LOOPIF $a=gamlist.e4 GATEDBY gates1
               {
               INC mat2d($a.e4,listp3.e4)
               INC mat2d(listp3.e4,$a.e4)
               }
            INC mat2d(listp3.e4,listp3.e4)
            GOTO endloop1
            }
         }
      LOOPFAIL
         {
                                                                  !! =2 gates
!! ...no gates satisfied in listp2 so listp2 contains just the x parameters
         INC mat2d(listp2.e4,listp2.e4)
         GOTO endloop1
         }
      }
   }

LABEL endloop1

*RUNFILES
COMP1 RUN1
COMP1 RUN2
COMP2 RUN3

*FINISH
```

Quinsort

This example is similar to the previous one except that it contains a triple gate instead of a double one.

```
! compressed data sortfile
! triple gates updating 2D matrix
!
*FORMATS
gam[1:134] (tag:3, e4:13)

*DATA

!! gates for energy=e4/2
GATEMAP 1D gates1 [2050]
( 672  680)  (1165 1175)  (1346 1358)
(1474 1486)  (1544 1556)  (1686 1698)
(1566 1578)  (2036 2048)  (998 1006)
(1239 1249)  (1433 1443)

*SPECTRA
!== sort spectra ======
triple2d   4096 2D

*COMMANDS

CREATELIST gamlist FROM gam

LOOPIF $c=gamlist.e4 VALID
   $c.e4=$c.e4/2

if (NUMBER(gamlist)) LT 5
   ENDEVENT

!! triple gated 2D update
!!
!! Use LOOPIF to decide whether the event satisfies 3, 4 or 5 gates
!! Then loop over the appropriate words in event to update matrix
!!  and exit loop
!!
!! Update Algorithm:
!! For m-dim update and p gates, words which satisfy gates are g parameters,
!! all others are x parameters.  Have 3 possible cases:
!!  1. satisfy exactly p gates -- update m-tuples from x params
!!  2. satisfy at least p+m gates -- update m-tuples from g+x params
!!  3. satisfy p+k gates, k<= m -- update m-tuples which involve <= k g params
}
!!
!! for this case:
!!  1. satisfy exactly 3 gates -- update doubles from listp3
!!  2. satisfy at least 5 gates -- update doubles from gamlist
!!  3. satisfy exactly 4 gates -- update g params with singles from listp4
!!
LOOPIF gamlist.e4 GATEDBY gates1 NEWLISTX=listp1
   {
                                                          !! >=1 gate
   LOOPIF listp1.e4 GATEDBY gates1 NEWLISTX=listp2
      {
                                                          !! >=2 gates
      LOOPIF listp2.e4 GATEDBY gates1 NEWLISTX=listp3
         {
                                                          !! >=3 gates
         LOOPIF listp3.e4 GATEDBY gates1 NEWLISTX=listp4
            {
                                                          !! >=4 gates
            LOOPIF listp4.e4 GATEDBY gates1
               {
!! ...at least 5 gates satisfied in gamlist -- update all parameters
               INC triple2d(gamlist.e4,gamlist.e4)
               GOTO endloop1
               }
            LOOPFAIL
               {
                                                          !! =4 gates
!! ...no gates satisfied in listp4
```

```
               LOOPIF $a=gamlist.e4 GATEDBY gates1
                  {
                  INC triple2d($a.e4,listp4.e4)
                  INC triple2d(listp4.e4,$a.e4)
                  }
               INC triple2d(listp4.e4,listp4.e4)
               GOTO endloop1
               }
            }
         LOOPFAIL
            {
                                                                  !! =3 gates
!! ...no gates satisfied in listp3 so listp3 contains just the x parameters
            INC triple2d(listp3.e4,listp3.e4)
            GOTO endloop1
            }
         }
      }
   }

LABEL endloop1

*RUNFILES
COMP1 RUN1
COMP1 RUN2
COMP2 RUN3

*FINISH
```

Pulse Processing

This example shows a simple pulse processing code example.

```
*FORMATS !#####################################################################

grt[1:63] (e1,e2)          ! Read GRT4 energy value from MWD
posxy[255] (x,y,nstep)           ! Position table
fadcs[260:350] (pulse(250))      ! GRT4 (14 bits 80Mhz)

*DATA  !######################################################################

gatemap  2D  coincidence[512,512]
(181 155 186 160 198 148 203 132 189 125 182 136)

valuearray pulse[512]         ! Initialise pulseshape array

valuearray grtenergies[1:37]     ! Initialise energy array
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1

valuearray grtbases[1:37]     ! Initialise base array
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1

*SPECTRA !####################################################################

fadchiti          128  32  ! FADC hit pattern
fadchito          128  32  ! FADC hit pattern
grtid             256  32  ! GRT ID
foldf             128  32  ! FADC fold (should be 1)
foldgrt           128  32  ! Fold grt
multge            128  32  ! Segment multiplicity
gehighi[1:40]     8192  32  ! Energy
grtenergyi[1:40] 8192  32  ! Energy from grt4
grttot           8192  32

! ============ Pulse Shapes ================
pulseshape[1:40]  512 2D 32     ! Shapes
pulses[1:37]      256 32
! ============ Position Table ==============

posx                        4096 32   ! x position (ungated)
posy                       4096 32   ! y position (ungated)
position         128 2D 32   ! Raw position matrix

*COMMANDS !###################################################################

!-----------------------------------------------------------------------
! Reinitialise the arrays each event
!-----------------------------------------------------------------------

doloop loop1 from 1 to 37 step 1
 {
 grtenergies(loop1) = -1
 }

!-----------------------------------------------------------------------
! Increment the position matrix: density of photon interaction
!-----------------------------------------------------------------------

inc posx(posxy.x)             ! Position x (vertical)
inc posy(posxy.y)             ! Position y (horizontal)
posxx = (posxy.x/20)       ! Step corresponds to the scan step
posyy = (posxy.y/20)       !
inc position(posxx,posyy)

!-----------------------------------------------------------------------
! Now let us look at the data ....
!-----------------------------------------------------------------------

createlist fadclist from fadcs
mult = number(fadclist)
inc foldf(mult)

loopif $f = fadclist valid
```

```
 {
 mult = 0
 energy = 0
 adcvalue = 0

 numfadc = group($f)
 numfadc = numfadc - 259      ! Renumber 1->37
 numbase = numfadc
 inc fadchiti(numfadc)

 ! User routine to calculate energy ...
 !-------------------------------------------------------------------
 exec fadc_12bit $f numfadc energy init 0
 !-------------------------------------------------------------------

 inc fadchito(numfadc)                ! Output hit pattern

 if numbase lt 37
 {
  doloop loop1 from 1 to 250 step 1    ! Generate pulse histogram
  {
   a = $f.pulse[loop1]
   a = ((a/15)+50)
   inc pulseshape(loop1,a) indexed numbase
   set pulses(loop1) indexed numbase = a
  }
 }

 grtbases(numbase) = adcvalue

 if numfadc le 37
 {
    inc gehighi(energy) indexed numfadc
 }

} ! Close main loop

multgrt = 0
createlist grtlist from grt
multgrt = number(grtlist)
inc foldgrt(multgrt)

loopif $g = grtlist valid
 {
 groupno = group($g)
 groupno = groupno - 3
 inc grtid(groupno)

 temp = $g.e2 / 4
 if temp gt 5              ! Arbitrary 0 cutoff
  {
    inc grtenergyi(temp) indexed groupno  ! Raw energu
    grtenergies(groupno) = temp           ! Store the energies
  }
 else
  {
    grtenergies(groupno) = 0           ! Store the energies
  }
 }

! ========================================================
! ============ Now lets Analyse the Data =================
! ========================================================

mult = 0
doloop loop1 from 1 to 36 step 1
{
 temp = grtenergies(loop1)
 if temp ge 5
 {
  mult = mult + 1
  hitsegment = loop1
  inc grttot(temp)
 }
}
inc multge(mult)     ! Increment segment multiplicity
```

```
end
*RUNFILES !#################################################################
DISC /net/npr3/d1/CALIB/Run6_0
*FINISH !###################################################################
```