EDOC081

EUROGAM PROJECT

NSF DATA ACQUISITION SYSTEM

EUROGAM Event Builder Language

Edition 2.1
July 1994

Nuclear Structure Software Support Group
Liverpool University

Contents

EUROGAM Event Builder Language

John Cresswell and Linda Pratt

Edition 2.1
July 1994

1 Note on Revision 2.1

The first release of this document was originally produced as the specification for the event builder language before the software was written. Hence much of it was in the future tense and it discussed matters from a design point of view. Subsequent releases reflected new commands and features in the language, but the discussion parts of the document were not updated. Hence the early sections have now been substantially rewritten, but may still prove helpful to the user in explaining concepts behind the language.

New language features have been necessary for phase II of EUROGAM to handle the clover detectors: the main one is the **groupcopy** command. Some additional changes have been needed to make sense of this command.

Finally, some more examples have been added to the manual.

2 Introduction

The prime function of the event builder is to build events containing the defined set of parameters in the defined format.

There is also some consistency checking done on the quality of events processed on input. For example, notification of parameter words that are not expected. In this case the user has wrongly identified the set of parameters to be collected, or the hardware has generated corrupt information. These checks are incorporated automatically into the event builder software and are not under the control of the user. (This document deals only with user-specific requirements. It is not intended as a complete technical manual.)

The EUROGAM project defined a list of general actions for the event builder, together with suggestions of what it should not do. The event builder language provides support for Compton suppression, new word formation, event filtering and simple arithmetic. It does not support operations such as gain matching and ballistic deficit correction.

Another criterion for event builder processing could be to reduce the data volume for onward shipment. Since the event builder has to process 100% of a high data volume (by definition) then any processing has to be simple and effective. This should be remembered when the user defines the commands to be executed.

The efficiency of computation required to cope with the level of processing necessary means that code has to be specific to the EUROGAM array. Then the known structure of the array and the input format of the data can be used to maximise the efficiency of the software. However, the provision of the event builder compiler allows the user some flexibility over what processing is done to the data without having to understand and write code at a low level.

3 Data considerations

The document "EUROGAM Hardware Event Format Usage" (J. Cresswell, EDOC064) describes the structure of the data stream on input to the event builder. The document describes two types of parameter – system control words and user data words. All parameters have the same general format (i.e. a 32 bit word divided into address and data fields). The system control words are removed by the event builder software, leaving only the data words for the user to deal with, i.e. the user is not concerned with which ROCO produced what data words.

3.1 Groups and Items

Each data word (or *item*) has a unique 14 bit EUROGAM *address* which is written into the data acquisition hardware during the set-up phase. The address is divided logically into two parts: an 8 bit *group number* and a 6 bit *item number*. The repeated structure of the array apparatus is such that most parameters expected by the EUROGAM acquisition system naturally fall into *groups*. In particular, the set of data from a single Ge and associated BGO detectors can be considered as a *group* of parameters to be treated together. For each Ge detector the associated group will has a maximum of 32 parameters or *items* with item numbers running from 0–31. Each group has a group number. Each event consists of several such groups preceded by a trigger or system group (number 255). Extra data acquisition devices (e.g. the recoil mass separator, FIFI, inner balls, ...) each provide a group of parameters with its own unique group number.

Many of the groups described above are identical because they all have the same type of data that is to be treated similarly. This suggests a further conceptual organisation of the data into sets of groups that are to be treated similarly. Each such *groupset* can be given a *group name* in the language via a **group** format specification (§6). Each groupset covers a *range* of group numbers, thus allowing, say, 45 identical Ge groups to be specified in one statement. In the same statement, *item names* are associated with the item numbers in a particular groupset, so avoiding the need for the user to refer to addresses.

3.2 Trigger Group

The trigger group is the only fixed group to appear in every event. The group number is always 255. The parameters available from the trigger module are more fully described in "EUROGAM Trigger System" and "EUROGAM Master Trigger Module" (I. Lazarus, EDOC022 and EDOC055).

The items for the trigger group are numbered starting from 48 (@30 in hex), rather than from 0 as for other groups on input to the event builder. When the trigger group is written to output, then its items are numbered implicitly from 0, as for all the other groups.

3.3 Single Item Names vs Group Item Names

The item names introduced in §3.1 may actually describe several items of data. These are termed *group item names*. It is possible to associate an item name with an address specified explicitly and uniquely (see §6). This is a *single item name*. An item is an item, no matter how it is specified, but the way the names are used is slightly different. Most of the parameters from an experiment will fall into groups. Groupsets with only one group and groups with only one item are permitted.

3.4 Local Variables and Constants

A *local variable* is a variable that the user may define to perform calculations. It is local in the sense that it is entirely confined to the event builder: it is not expected as input and is never needed as output, unlike the items, etc described above. A local variable does not have a EUROGAM address associated with it.

There is the facility in the language to initialise the value of a variable at the start of each event (§8). Generally, however, a local variable is introduced by using it in a statement and is automatically initialised to zero at the start of processing each event. Note: it is not possible for the user to save or accumulate information from one event and access it in subsequent events.

Constants may also be declared. The value of a constant is set once during the set up phase and is local in the same way that a variable is.

4 Organisation

In order to define the operation of the event builder the user has to create a text file using any favourite editor. The suffix of the filename must be `.ebl`. The text file must be read by a programme (compiler) called ebc to generate the software required. The specification of the language can be found in the following pages. A quick reference card for the syntax of the language in the document entitled "EUROGAM EBC Reference Card" (L.C. Pratt and J.R. Cresswell, EDOC082) version 2.01 and a supplementary card "EUROGAM SS Reference Card" (L.C. Pratt) version 2.01 are available.

The language file is essentially free format, subject to starting each separate command on a new line. The file is divided into 4 sections with the following titles, and in the following order:

input formats
: Describes what items are expected as input to the event builder.

output formats
: Describes how the items are to be output.

values
: To initialise the user's local variables and constants and
to define templates for copying from group to group.
This section is optional.

commands
: Any processing required.

The file is terminated with the statement:

finish

The file specifies the expected input, what may be required as output from the event builder and what to do the data to turn the one into the other. However, the formats sections describe the maximum set of data to be handled; only a subset of this data will be present in a given event. To handle this situation, the **whenever** command should be used (§9.6). It defines sets of commands to be executed for each group present that belongs to a specified groupset. It tests the address of items. An **if** statement (§9.5) can be used in one of 2 ways, either to test the data value of items, variables and constants or to test whether an item is present in the the event.

The output of a group is performed by associating common names and then ordering the items according to the output format. Output is performed by the **stream** command or by default on reaching the end of the event processing (see §9.9).

As processing is carried out, parts of the event may be discarded using the **reject** command (§9.7) according to whatever criteria the user sets. (The event may also be rejected altogether). This prevents output of the rejected data when it comes to using a **stream** command. The data rejected is no longer accessible via subsequent commands and has essentially been discarded. The **suppress** command (§9.8) is an exception to this statement. It accesses hit pattern words from groups that have been rejected earlier on in the processing. The data is, therefore, still present in the memory, even when it has been rejected.

5 Command Syntax

Within the syntax definitions, the following conventions are followed: All substitutable components are specified as $< \cdots >$, for example **<name>**, **<address>**, etc... Where more than one occurrence of a component is possible, the repeat specifier is denoted as a lower case "r" following the component, for example **<name>r** . Syntax that is only optional is denoted using sloped font ***like this*** .

In the examples, keywords from the event builder language appear in `UPPER CASE` and names chosen by the user appear in `lower case`.

Statements

All statements must begin on a new line. Most statements are short enough to occupy only one line and it is recommended that they do so if possible. The syntax will allow longer commands to be continued on successive lines, but line breaks are only allowed at points between specified command components. Blank lines between (but NOT in the middle of) statements are simply ignored. Unnecessary white space between command components is also ignored.

Names

All names consist of between 1 and 8 alphanumeric characters from the set A–Z, a–z, 0–9 and underscore(_). The first character must be alphabetic.

Case

The compiler is not case-sensitive. For example, the forms `MRAW`, `Mraw` and `mraw` are all treated as the same name.

Numbers

Only integer arithmetic is handled by the event builder. Numeric values may take any of the following forms:

Decimal integer	`0 25 -33`
Binary integer	`%00001110001`
Hexadecimal integer	`@ffff @00001234`

For all three integer formats the number of characters is variable, and assumes the least significant digit is to the right. The maximum number of digits accepted is 32 for the binary representation, 10 for the decimal representation and 8 for the hexadecimal representation. These limits are all subject to generating a legal 32 bit integer value.

Comments

Comments may be incorporated into the text subject to the following rules. Any text following an exclamation mark (!) up to the end of the line is ignored by the compiler. Hence whole lines or part of lines after commands may be used for commenting the text.

6 Input Definitions

The user must specify the set of parameters expected to be received from the data acquisition hardware. This specification relates a user-defined name to a particular data acquisition channel address. The address is the 14 bit value as defined in other documentation.

If any expected parameters are missed out of the definition, the event builder will not know what to do with the data from the missing parameters and an error statistic will be incremented. If extra parameters are specified which will never be received, this results in wasted resources such as memory, executing unnecessary instructions and in extreme cases (e.g. if every numerically possible item 0–32 and group 1–255 were defined as expected), may result in the event builder accepting corrupt data as valid.

Input Section Syntax

input formats

The input section consists of the above statement, which should be the first statement in the EUROGAM event builder language source file (excluding comments), followed by one or more format specification statements. The order in which things are specified in the input section is not significant.

Format Specification Syntax

<item-name> <address>

or

group <group-name> [<range>] (<item-list>)

where:

<item-name>	is a user-provided name for a parameter,
<address>	is the 14 bit hardware address of the parameter,
<group-name>	is a user-provided name for a group of parameters,
<range>	is the set of group numbers and
<item-list>	is a list of <item-name>s separated by commas.

The set of group numbers may be specified as a single number, as an inclusive range of consecutive numbers $i : j$ (where $i < j$) or a mixture of both, for example `[5,7:15,31:46]`. The group ranges should not overlap, neither should the same group number be declared in different groupsets, i.e. each group number should be unique within a given groupset, and within the **input formats** section.

The item names specified in the <item-list> should not clash with each other, item names specified in other groupsets, item names declared using the single item format (the first format statement type given above) or group names, i.e. item names and group names should be unique within the **input formats** section.

The **group** statement implicitly declares consecutive item numbers for the group items. Sometimes, there may be missing item numbers from the expected list, for example, the trigger words can be activated or inactivated independently of one another. To facilitate this, a keyword **blank** may appear in <item-list>. It is used as a spacer instead of an <item-name>. The <item-list> may contain the keyword several times, but it is an error for it to consist entirely of spacers. The keyword should not be used as the last element in <item-list>. The keyword **blank** is thus a reserved word and may not be used for any other purpose. It is not an <item-name>: for example, references to `blank` in an expression (see §9.1) will cause a compiler error.

Example:

```
tac @0103
GROUP trg[255](param1, BLANK, param3, param4)
GROUP germ[1:70](e1, e2, t1, ebgo, tbgo, hit)
```

The first line declares a single item, `tac`, is expected as input with the address shown. In effect it has group number 3 and item number 1.

The second line defines a groupset, `trg`, which contains one group, with group number `255` containing items `param1, param3` and `param4`. Their item numbers are implicitly defined as 0, 2 and 3 respectively. The `BLANK` is present no item with item number 1 is expected from hardware.

The last line declares data from a groupset of 70 groups with group numbers from `1` to `70` is expected as input. Each group contains items `e1` to `hit` with item numbers 0 to 5.

Valid line break points:

- after the right square bracket (]) following <range>,
- after one of the commas (,) in <item-list>.

7 Output Definitions

The user must specify the set of parameters to be written to the output event buffer. The list of parameters to be output may include all or a subset of those input together with any new parameters created during event building. If an item in the input parameter list is also specified in the output list then it must remain in the same group and have the same group number. Names used as item or group names in the input definitions may not be used as group or item names respectively in the output definitions. The syntax is similar to that for the **input formats** section.

In the absence of any **stream** commands later on in the language file (see §9.9), the order in which things are specified in the output section is the order in which they will be written to the output buffer (if the data is present in the event).

Output Section Syntax

output formats

The **output formats** section follows on from the **input section** and consists of the above statement, followed by one or more format specification statements.

Format Specification Syntax

<item-name> <address>

or

group <group-name> [<range>] (<item-list>)

where:

<item-name>	is a user-provided name for a parameter,
<address>	is the 14 bit hardware address of the parameter,
<group-name>	is a user-provided name for a groupset,
<range>	is the set of group numbers and
<item-list>	is a list of <item-name>s separated by commas.

The set of group numbers may be specified as a single number, as an inclusive range of consecutive numbers $i : j$ (where $i < j$), or a mixture of both, for example `[5,7:15,31:46]`. The group ranges should not overlap, neither should the same group be declared in different named groupsets, i.e. each group number should be unique within a given groupset, and within the output groupsets. It is further necessary that the addresses specified in the single item format do not refer to combinations of group number and item number implicitly specified in a group format statement. A groupset that appears in both input and output sections must have identically matching group numbers.

The item names specified in the <item-list> should not clash with item names specified in other groupsets or item names declared using the single item format (the first format given above) or group names, i.e. item and group names should be unique within the output definitions section.

Example:

```
tac @0103
GROUP germ[1:70](e1,t1)
```

The example is similar to that for the input formats section. However, the definition here specifies how the data will be output. In the groupset `germ` fewer items have been specified than in the input section, so the user has reduced the number of words to be output.

Valid line break points:

- after the right square bracket (]) following <range>,
- after one of the commas (,) in <item-list>.

8 Variable and Constant Initialisation

Any names that have not been defined in the input or output definitions section are treated as local to the event builder. By default the names are assumed to be local variables and are set to zero at the start of processing each event. The **values** section allows the user to specify an alternative value to this default or to declare a constant that does not have to be set every event.

Items that are defined only in the **output formats** section may also be initialised in this section. They are NOT initialised to zero by default, but are considered to be absent unless they get set during processing.

This section is also used to define *template*s for the **groupcopy** command. The source and destination groupsets and how to transfer the items belonging to the group are specified and the definition is associated with a template name, which can be referred to subsequently in **groupcopy** commands in the **commands** section. The detailed description of this is deferred until §9.10 and §9.11.

Values Section Syntax

values

The **values** section is optional. It follows on from the **output formats** section and consists of the above statement, followed by one or more value specification statements.

Value Specification Syntax

set <variable-name> = <integer>

or

set <item-name> = <integer>

or

constant <constant-name> = <integer>

or

groupcopy <template-name> to <group-name>
from <group-name> <template-body>

where:

<variable-name>	is a user-provided name for a local variable,
<item-name>	is the name of an item defined only in the output section,
<integer>	is the value specified,
<constant-name>	is a user-provided name for a local constant,
<template-name>	is a user-provided name for the template,
<group-name>	is the name of a groupset defined in one of the formats sections and
<template-body>	is explained in §9.11.

The <variable-name> or <constant-name> should be unique within the **values** section and must not be in use as an item name in the **input formats** section or as a group name in either format section.

The **set** statement ensures that the local variable or item is reset to the given value at the start of every event. The **constant** statement causes the value of the constant to be defined at set-up time. A constant can not subsequently be redefined with an assignment statement, say, whereas a variable can be.

Example:

```
CONSTANT emin = 50
CONSTANT emax = 10000
SET ntest = 9
```

This defines `emin` and `emax` to have fixed values as shown, and ensures that `ntest` is set to 9 on starting to process each event.

9 Commands

The user's commands appearing in the **commands** section only take effect once the input data has been scanned once and passed the checks for corruption.

Actions that apply to groups of parameters of a similar type can only be applied in **whenever** blocks (§9.6). The group range facility in the input format specification (§6) allows all the groups in an event satisfying the range criteria to be acted upon by the same commands.

Commands Section Syntax

commands

The **commands** section comes after the **values** section if present, or otherwise after the **output formats** section. It consists of the above statement followed by any number of command statements. The **commands** section is compulsory, but it may be empty, ie the user may wish to do no special processing at all. The possible commands are detailed in the following sections.

Qualification Bits

There are two of these, named **q0** (most significant bit) and **q1** (next most significant bit) at the front of every word received by the event builder that are not part of the 14 bit address or the 16 bit data value. They may be accessed in the **commands** section by adding `.q0` or `.q1` to any valid reference to an item name in an expression or assignment. They are discarded on output.

The qualification bits can be copied onto the top of the dataword with the **moveqq** command (see §9.4).

Example:

```
IF e1.Q0 EQ 1 THEN REJECT GROUP
```

See §9.5 for the explanation of the **if** command and §9.7 for the **reject** command.

Special Names

There are 2 special names which are used to access specific information in the commands section. The **eventno** is the EUROGAM event number of the current event and **groupno** is the EUROGAM group number of the group currently being processed inside a **whenever** block (see §9.6).

They are conceptually somewhere between a local variable and a constant. They are like a constant because they can never be set by the user, or appear in one of the formats sections. Other variables or items can be set to the value of one of these special names, or they can be used in expressions in comparisons. The **eventno** is the same for the processing of one event, but the value is not truly constant because it is different from event to event. The value of **groupno** is not constant at all, it is more like a loop counter.

9.1 Arithmetic Expressions

<variable> = <expression>

where

<variable>	can be an <item-name> from either the input or output parameter lists or a local variable and
<expression>	is any arithmetic expression obeying C operator precedence (see table below) involving <item-name>s, local variables and constants.

The operands in <expression> may be <item-name>s, integers or local variables and constants. If not listed in at least one of **input formats**, **output formats** or **values** section then <variable> is assumed to be a local variable, and similarly for any undeclared names in <expression>. An arithmetic operation on a group name is not allowed. Parameter variables which are only defined in the **output formats** section were formerly automatically initialised to zero at the start of processing every event under the EUROGAM phase I compiler. This behaviour has been altered for phase II: they are not set and are considered not to be present in the event until explicitly set by an instruction from the user.

Up to 6 levels of parentheses are allowed in <expression> and the operators given in the following precedence table may be used:

				associativity
not	bitwise NOT (complement)			right
*	multiply	/	divide	left
+	plus	−	minus	left
<	shift bits left	>	shift bits right	left
&	bitwise AND			left
xor	exclusive bitwise OR			left
ior	inclusive bitwise OR			left

Operators on the same row have the same precedence; the first row has the highest precedence. (FORTRAN precedence is, in fact, a subset of C, since the bitwise operators do not exist as such in FORTRAN.)

Example:

```
sumenergy = (sumenergy + ebg0)/2 + e2
```

Valid line break points:

- after the equivalence sign (=),
- after one of the operators.

9.2 Increment Command

increment <variable>

where

<variable> has the same definition as for arithmetic commands.

This is the simple arithmetic operation of adding one to the value of a variable, item or group item. It is provided in this form only as a more efficient alternative to using `a = a + 1`.

Example:

```
INCREMENT a
```

9.3 Decrement Command

decrement <variable>

where

<variable> has the same definition as for arithmetic commands.

This is a simple arithmetic operation of subtracting one from the value of a variable. It is provided in this form only as a more efficient alternative to using `a = a - 1`.

Example:

```
DECREMENT a
```

9.4 Moveqq Command

moveqq <item-name>

where

<item-name> is the name of an item defined in input or output section.

This command causes a copy of the qualifying bits to be taken and put on the top 2 bits of the dataword. The <item-name> must not be qualified with access to specific qualifier bits. If the bits on the input data word are numbered 0–31, with bits 0–15 being the data value, then:
(q0) bit 31 → bit 15
(q1) bit 30 → bit 14
Both bits 31 and 30 are always copied, even if they are not set or are never used and the copy overwrites bits 15 and 14, regardless of their values.

Moving the bits changes the value of the data, both in expressions in subsequent event builder code and further on down the datapath, eg for the sorter.

It should be noted that the tagging mode of shared suppression (see the **suppress** command, §9.8) also allows the top bit (15) of the data value to be set, which could be a source of confusion.

Example:

```
MOVEQQ e1
```

9.5 If ... Else Construct

if <condition> then <statements>

where

<statements> may be a single statement or {<statement>r}

<condition> is one of

<variable> <test> <value>
<variable> passes [<value>,<value>]
<variable> fails [<value>,<value>]

where

<variable>	is an item name or local variable name,
<test>	is one of **eq** (equality), **ne** (not equal to), **lt** (less than) **le** (less than/equal) **gt** (greater than), **ge** (greater than/equal)
<value>	is an integer value, an item name a local variable name or a constant name.

This allows the execution of blocks of code conditional on the values of the data in the event. Up to 6 **if** blocks may be nested inside one another. Note that the condition may not contain an expression. The limits on the gate are inclusive.

Finally, an **if** block may be followed by an **else** block:

else <statements>

The **else** block is optional and is executed if the condition in the **if** is not fulfilled. When using nested **if** ... **else** constructs there can be some ambiguity about which if condition the else matches. This is resolved in the same way as in C and FORTRAN: the condition from the most recent (innermost) **else**less **if** is used. This arises when the single statement form is used, and is eliminated by using braces round the statement.

If an item that is not present in the event is referenced in an **if** condition, it is treated as if it had value zero for the purpose of the test.

Example:

```
IF mraw LT 8 THEN REJECT EVENT
IF sumener PASSES [50,345] THEN
   {
   IF nge LT 5 THEN REJECT EVENT
   sumener = sumener/10
   }
IF tac FAILS[20,403] THEN REJECT EVENT
ELSE INCREMENT tcount
```

If `sumenergy` has value 50, then the second condition is fulfilled and the statements in braces are executed. If `tac` has value 20, then the third condition is not fulfilled and the event is not rejected, but `tcount` is incremented instead.

Valid line break points:

- before the left brace ({),
- after the left brace ({),
- at a valid line break point in <statement>.

NB: the right brace (}) must be on a line on its own (apart from a trailing comment).

9.6 Whenever Command

whenever <condition> then <statements>

where

<statements> may be a single statement or {<statement>r},

<condition> is one of

group = <group-number>,
group = <group-name> or
item = <sitem-name>

where

<group-number>	is an integer group number between 1 and 255,
<group-name>	is the name of a groupset defined in the input or output section and
<sitem-name>	is an item name declared with the single item format in the input or output section.

The **whenever** command executes a block of code (<statement-list>) for the <group-number> or <sitem-name> given if they are present in the event. The block is executed for every group present in the event that fulfills a <group-name> condition, i.e. this type of **whenever** statement implies looping over the block of code. The group number of the group currently being processed can be obtained using the special symbol **groupno** inside the block (see §9).

Whenever blocks may be nested within each other. Also, **if**s inside **whenever**s and vice versa are allowed. A **whenever** command implies looking for the presence of word(s) in the event. The **if** command tests the value of data. (<sitem-name> may not have a qualification bit, that would not make sense.)

Example:

```
WHENEVER GROUP = germ THEN INCREMENT ndet
WHENEVER GROUP = germ THEN
   {
   INCREMENT nge
   sumener = sumener + (ebgo/2) + e2
   IF t1 FAILS [100,250] THEN REJECT GROUP
   }
```

The statement(s) shown are executed for all groups found in the event that belong to the groupset with group name `germ`. Note the use of group item names like `t1` and `ebgo` is only valid in a **whenever** statement or block, as shown in the example. If the tt reject command causes the condition in the `whenever` statement no longer to be true, that block of code is exited immediately. (See §9.7 for the explanation of the **reject** command.)

Valid line break points:

- after the THEN,
- after the left brace ({),
- at a valid line break point in <statement>.

NB: the right brace (}) must be on a line on its own (apart from a trailing comment).

9.7 Reject Command

reject <group-name>

or

reject <sitem-name>

or

reject event

where

<group-name> is the name of a groupset
<sitem-name> is the name of an item defined with a single item format statement.

The first form of the **reject** command may be used with **if** and **whenever** commands to prevent all data from groups present in the event that belong to the specified groupset from being output or used in processing. The second form prevents the specified item from being output if it is present. The 3rd form effectively terminates processing for the current event. Any data already written to the output streams will be passed on. A further form of the **reject** command:

reject group

is only valid within a **whenever** block. It rejects the current group being processed. If there are nested **whenever** statement blocks, any ambiguity is resolved by taking the current group of the innermost bf whenever block where the condition uses the `whenever group =` syntax.

It does not matter whether the item or groupset is defined in the **input** or **output format** sections.

If the rejection command causes any whenever condition in force no longer to be true, that whenever block is exited. The user should be aware that code inside the block is executed up until the rejection command takes effect. Thus a multiplicity counter for a groupset might have the value 3 because 3 groups were processed in a `whenever group =` <group-name> block and on the 4th group, the groupset was rejected, so the number of valid groups in the groupset is 0.

A rejected item or group has not been wiped from memory. It is simply prevented from being output by a **stream** command or in the default output that occurs at the end of event processing. The data is still available and the BGO hit patterns will be used by the **suppress** command but subsequent **whenever** access to other items of rejected data is not available.

Example:

```
IF mraw LT 8 THEN REJECT EVENT
WHENEVER ITEM = param THEN
   {
    IF param FAILS [1000,4000] THEN REJECT param
   }
WHENEVER GROUP = germ THEN
   {
   IF t1 FAILS [100,250] THEN REJECT GROUP
   ELSE INCREMENT count
   }
IF count LT 4 THEN REJECT germ
```

The first line of the example rejects the whole event if `mraw` is less than `8`. The 1st `whenever` block rejects single item `param` if it fails a filter. The 2nd `whenever` block checks item `t1` for every group in groupset `germ`, rejecting the group if `t1` fails a filter and counting the number of groups for which the test is passed. In the last line, if `count` is too low, the whole of groupset `germ` is rejected.

9.8 Suppress Command

suppress scheme <scheme-number> file <data-file> ***tag***

where

<scheme-number>	is an integer code for the suppression scheme,
<data-file>	contains suppression information and
tag	is an optional keyword to select tagging mode.

This command performs the (shared) suppression on the event according to the instructions in <data-file>. The filename should be a Unix filename and should have the extension `.ss`. The <scheme-number> is to allow the user to choose between different possible suppression schemes described in the same file. In general, the higher the code, the more BGO detectors are allowed to suppress a given detector.

The keyword **tag** is optional. It specifies that the effect of the **suppress** command is to tag one of the datawords in each suppressed group by marking the first bit of the data value as a 1. Note that this is one of the 16 bits of the data value and any subsequent calculations using a tagged item will use a value that includes the tag bit. The data is then still available to be output by a **stream** command. Note that the tagged item must be one that gets output in order for the suppression information to be passed on. This is *tagging mode suppression*. Note also that the commands **groupcopy** and **moveqq** also add bits in to the top of item data values.

In the absence of the **tag** keyword, suppressed groups are marked as discarded as if they had been the subject of a **reject** command. The data for the suppressed groups are not then written to the output streams. This is *rejection mode suppression*.

Appendix A contains a description of the format of <data-file> contents accepted. It details what different <scheme-number>s mean and how to specify which item in a group should be tagged and which item is to be taken as the hit pattern.

9.9 Stream Command

stream <stream-number> <output-list>

or

stream <stream-number> event

where

<stream-number>	is an integer stream number between 0 and 4 inclusive,
<output-list>	is a list of <groupname>s and <item-name>s defined in the output section, separated by commas,
event	is a keyword specifying the whole event.

This command allows streamed output from the event builder. (The sort package also has the facility to stream data.) The specified data items are written in the given order to stream <stream-number> if they are present in the event at the current stage of processing. The format in which the data is written is described in "Event Format [Sorting + Storage]" (J. Cresswell, EDOC074).

If the **event** keyword is specified, the data for the entire event is output as specified in the **output formats** section (aside from groups that have been discarded with the **reject** or **suppress** commands). The same data may be written to different streams. There are 4 streams numbered 1–4 which can send data along the data path and ultimately to tape. (The mechanism for specifying which stream goes where is not part of the event builder language. The responsibility for supervising streaming lies with the tape server. The on-line sorter spies on the data path to obtain its input.)

There is a default setting: if no **stream** commands are used in the source file, then the event is written to stream 1 at the end of processing. This guarantees output will be passed on without the user having to say anything. This default does not apply if there are any **stream** commands used, even if none have been executed for the given event. If, because for example, the source file contains a single **stream** command inside an **if** block of statements for which the condition was not fulfilled by the event, then no output would result for that event. A single **stream** statement in the source file means that all further output must be specified explicitly.

Example:

```
STREAM 1 trigger, germ
STREAM 2 recoil
STREAM 3 EVENT
```

This will write the data for any groups present in groupsets `trigger` and `germ` to stream `1`, and that for `recoil` to stream `2`. All the data specified in the **output formats** section that is present will be written to stream `3`.

Valid line break points:

- after <stream-number>,
- after one of the commas (,) in <output-list>.

9.10 Groupcopy Command

The **groupcopy** command allows the user to copy data from a given group to another given group. The event builder language was designed to handle items in the same group together. This new command has been added to allow data from a group to be split up by copying some of its items to another group belonging to a different groupset.

The way it works is that the user includes in the **values** section the specification of a *template*, which defines how to do the copy from any group belonging to a given groupset to any group belonging to another given groupset. The template is given a name, by which it is invoked with a **groupcopy** command in the **commands** section, giving *group numbers* as arguments. The groups must belong to the groupsets defined in the template. The template syntax is given in §9.11 and the command syntax is given below.

groupcopy <template-name> to <group-number> from <group-number>

where:

<template-name>	is the name of a template defined with a **groupcopy** statement in the **values** section and
<group-number>	is an integer number, a constant or a variable giving the number of the group belonging appropriate to the groupset defined in the template.

Any data already in the destination group is overwritten by the **groupcopy** command. Any items in the destination group for which there is no original item defined or present in the event have their values set to zero.

Example:

```
GROUPCOPY templ1 TO 81 FROM 1
GROUPCOPY templ2 TO gn FROM GROUPNO
```

The first example copies the items from group `1` to group `81` according to the instructions in a template which has been given the name `templ1`. Group `81` must belong to the destination groupset as defined in `templ1` and similarly group `1` must belong to the originating groupset. The second example copies the items from the current group being treated by a **whenever** block (by reference to the special symbol `GROUPNO`, see §9) into a group whose number is given by the variable `gn` according to template `templ2`. Similar restrictions on groups belonging to defined groupsets apply as in the first example. The compiler attempts to check these, but in the second example it is not possible as the value of `gn` is not known at compilation time. If the event builder detects such an error during processing, an error statistic is incremented.

9.11 Groupcopy Template Definition

N.B. The template definition is done in the **values** section of the language file. Here is the template definition syntax, some of which has already been presented in §8:

groupcopy <template-name> to <group-name> from <group-name>
from <group-name> <template-body>

where:

<template-name>	is a user-provided name for the template,
<group-name>	is the name of a groupset defined in one of the format sections,
<template-body>	is {<template-line>r},
<template-line>	is given by

<item-list> : <item-list> *: <**tag-word**>*

where:

<tag-word>	is **tag** or **notag**

The <item-list>s are lists of <item-names> separated by commas as defined before in other commands. The first and second (or destination and originating) <item-list>s contain items belonging to the groupsets given by the first and second (or destination and originating) <group-name>s respectively.

When the template is invoked, the event builder looks for the next item in the originating list and if it is present in the event copies the value across to the first item in the destination list not yet set. If the originating item is not present, the event builder carries on down the originating list and looks until it does find an item to copy. This carries on until the destination items have all been set or there are no more items to look for in the originating list. Any destination items that have not then been filled have their values set to zero. Note that it is possible for one or both <item-list>s to contain only one item. Also there is little point in a destination list being longer than the originating list, as the extra items will never be filled.

When the items are copied across, information as to which item originally contained the value is lost. This may not matter: for example, if there is only one item in each list it is still obvious where the value came from. In some cases it could be important to know which item provided the value, in the context of leaves of a clover detector, for instance. The **tag** keyword is used to instruct that the copied item value is tagged with its position in the originating list using the top 3 bits. Since only 3 bits are available for tagging, this unfortunately means this feature is not totally general as only values 1-7 can be used for tagging, and the number of items in the originating list is only limited by the number of items in the originating group, which could be upto 32. The **notag** keyword indicates that tagging should not be done. The <tag-word> and the preceding colon may be omitted, in which case the default is not to tag the value.

Example:

```
GROUPCOPY templ TO opgs FROM ipgs
 {
  t       : t1   : NOTAG
  e       : e1,e2   : NOTAG
  ita,itb : i1,i2,i3,i4 : TAG
 }
```

The example shows the definition of a template called `templ`. It defines how to copy groups belonging to the originating groupset, `ipgs` into groups belonging to the destination groupset, `opgs`. Items `t1, e1, e2, i1, i2, i3` and `i4` must all belong to groupset `ipgs` and items `t, e, ita` and `itb` belong to groupset `opgs`. The first <template-line> will cause the value of item `t1` to be copied to item `t`, if `t1` is present: otherwise `t` is set to zero. The value will not be tagged, as it can only have come from one item. For the second <template-line>, `e` will take the value of `e1` if that is present, otherwise `e2` if that is present, otherwise zero. No tagging is done, so the user will not know if `e` contains the value of `e1, e2` or neither of these, but presumably does not care.

For the third <template-line> there are several possible outcomes as to which items get copied where, depending which of the originating items are present in the event. The event builder first looks to fill `ita` by checking for `i1`, if `i1` is absent, it looks for `i2` and so on. When/if it fills `ita` it then looks to fill `itb` by carrying on with the `i`*n* if there are any left. Any items it fails to fill are set to value zero. Tagging is used to say which items were copied. A tag value of zero means that no original item was copied into that destination item.

Suppose items `i2` and `i4` are present in the event. Then `ita` will have a data value copied from `i2` and be tagged with the value 2 and `itb` will have a data value copied from `i4` and be tagged with the value 4. Notice that the tag values of 2 and 4 are not the same as the item numbers of the item in the groupset `ipgs`. They could be items 16 and 7 respectively (the definition of the groupset is not shown). It is the position in the originating <item-list> that is used for the tag value.

The use of **groupcopy** is further illustrated in §11.2 and §11.3.

10 Reserved Words

The following is a list of reserved words which should not be used as user- defined names of items, variables, groups, etc. The list consists of key words that have special significance in the event builder language and the reserved words of ANSI C.

asm	event	int	suppress
auto	eventno	item	switch
blank	extern	le	tag
break	fails	long	then
case	file	lt	to
char	finish	main	typedef
commands	float	moveqq	union
const	for	ne	unsigned
constant	formats	notag	values
continue	fortran	output	void
debuga	from	passes	volatile
debugl	ge	register	whenever
debugp	goto	reject	while
decrement	group	return	stream
default	groupcopy	scheme	struct
do	groupno	set	
double	gt	short	
else	if	signed	
enum	increment	sizeof	
eq	input	static	

11 Examples

11.1 Example 1: mind.ebl

```
! File mind.ebl
!
! This is taken from the document by Beausang and Fallon entitled ‘A Mind
! Experiment with EUROGAM’.  It is not intended to be a complete example,
! but to give a flavour.
!
INPUT FORMATS

! Note: the case of the characters is not significant to the compiler.
group TRIGGER[255] (TYPE,Mraw)
GROUP germ[1:70] (e1,e2,t,tbd,ebgo,tbgo,hit)

OUTPUT FORMATS

GROUP trigger[255] (type,mraw,nge)
GROUP germ[1:70] (e1,t)

VALUES

! Define some constants to be used in filtering.
CONSTANT a = 2000
CONSTANT b = 15000
CONSTANT c = 300
CONSTANT d = 10000
CONSTANT e = 8000
CONSTANT f = 30000

COMMANDS

! Reject events with too few gamma rays.
WHENEVER GROUP = trigger THEN
   {
   IF mraw LT 8 THEN REJECT EVENT
   }

! Check the germanium groups calculating sum energy and other parameters.
WHENEVER GROUP = germ THEN
   {
   sumener=sumener+(e2+ebg0)
   SUPPRESS SCHEME 1 FILE config.ss
   IF e1 FAILS [a,b] THEN REJECT EVENT
   INCREMENT nge
   IF t1 PASSES [c,d] THEN INCREMENT ft1
   }
```

```
! Final event filtering.
IF sumener FAILS [e,f] THEN REJECT EVENT
IF ft1 LT 1 THEN REJECT EVENT

STREAM 1 EVENT                ! Send all the output data to one stream only.

FINISH
```

Named parameters (e.g. `sumener, ft1`) which are not defined in the **input formats** section are assumed by usage to be zeroed at start of each event and do not require zeroing specifically at the start of the **commands** section.

11.2 Example 2: clover.ebl

```
! File clover.ebl
!
! To illustrate the GROUPCOPY facility.  This file could be used to split
! the clover groups up into sets in which the same number of leaves fired.
! It also has the effect of reducing the volume of data for output by
! getting rid of many input items missing in the clovers groupset, which
! would be output with value zero.
!
INPUT FORMATS

GROUP trigger[255] (type, mraw, msup, mbgo, muser, mtac)
GROUP germ[1:50] (ge1, ge2, get, gebd, bgoe, bgot, bgop)

! The keyword BLANK is used to fill in for items not activated in hardware.
GROUP clovers[51:80](BLANK, BLANK, BLANK, BLANK,
                     clbgoe, clbgot, clbgop, BLANK,
                     geA1, geA2, geAt, geAbd, geB1, geB2, geBt, geBbd,
                     geC1, geC2, geCt, geCbd, geD1, geD2, geDt, geDbd)

OUTPUT FORMATS

GROUP trigger[255] (type, mraw, msup, mbgo, muser, mtac)
GROUP germ[1:50] (ge1, get, bgop)

! Define output groupsets to receive subsets of groupset clovers depending
! how many leaves fired.
GROUP clovers1[81:110]  (cl1bgop, ge1A1, ge1At)
GROUP clovers2[111:140] (cl2bgop, ge2A1, ge2At, ge2B1, ge2Bt)
GROUP clovers3[141:170] (cl3bgop, ge3A1, ge3At, ge3B1, ge3Bt, ge3C1, ge3Ct)
GROUP clovers4[171:200] (cl4bgop, ge4A1, ge4At, ge4B1, ge4Bt, ge4C1, ge4Ct,
                                                             ge4D1, ge4Dt)

VALUES

! Define groupno offsets for clover groupsets with different no of leaves
! firing.
CONSTANT cloff1 = 30
CONSTANT cloff2 = 60
CONSTANT cloff3 = 90
CONSTANT cloff4 = 120

! The VALUES section continues on the next page.
```

```
! The continuation of the VALUES section from the previous page.

! Define group copy templates to instruct how to copy a group into an
! appropriate place depending how many clover leaves have fired.  A
! maximum of 4 items appear in the originating lists, so it is OK to use
! tagging.
GROUPCOPY cp4 TO clovers4 FROM clovers
{
 cl4bgop                    :clbgop                 :NOTAG
 ge4A1, ge4B1, ge4C1, ge4D1 :geA1, geB1, geC1, geD1 :TAG
 ge4At, ge4Bt, ge4Ct, ge4Dt :geAt, geBt, geCt, geDt :TAG
}
GROUPCOPY cp3 TO clovers3 FROM clovers
{
 cl3bgop                    :clbgop                 :NOTAG
 ge3A1, ge3B1, ge3C1        :geA1, geB1, geC1, geD1 :TAG
 ge3At, ge3Bt, ge3Ct        :geAt, geBt, geCt, geDt :TAG
}
GROUPCOPY cp2 TO clovers2 FROM clovers
{
 cl2bgop                    :clbgop                 :NOTAG
 ge2A1, ge2B1               :geA1, geB1, geC1, geD1 :TAG
 ge2At, ge2Bt               :geAt, geBt, geCt, geDt :TAG
}
GROUPCOPY cp1 TO clovers1 FROM clovers
{
 cl1bgop                    :clbgop                 :NOTAG
 ge1A1                      :geA1, geB1, geC1, geD1 :TAG
 ge1At                      :geAt, geBt, geCt, geDt :TAG
}

COMMANDS

WHENEVER GROUP = clovers THEN
{
 ! Work out how many leaves have fired.
 leaves = 0

 IF geA1 GT 0 THEN INCREMENT leaves
 IF geB1 GT 0 THEN INCREMENT leaves
 IF geC1 GT 0 THEN INCREMENT leaves
 IF geD1 GT 0 THEN INCREMENT leaves

 ! The current WHENEVER block is continued on the next page.
```

```
 ! The WHENEVER block continued from the previous page.

 ! Copy the group into the appropriate groupset depending how many leaves
 ! fired.
 IF leaves EQ 1 THEN
  {
    gn = cloff1 + GROUPNO
    GROUPCOPY cp1 TO gn FROM GROUPNO
  }
 IF leaves EQ 2 THEN
  {
    gn = cloff2 + GROUPNO
    GROUPCOPY cp2 TO gn FROM GROUPNO
  }
 IF leaves EQ 3 THEN
  {
    gn = cloff3 + GROUPNO
    GROUPCOPY cp3 TO gn FROM GROUPNO
  }
 IF leaves EQ 4 THEN
  {
    gn = cloff4 + GROUPNO
    GROUPCOPY cp4 TO gn FROM GROUPNO
  }

}                                            ! End of the WHENEVER block.

STREAM 1 EVENT

FINISH
```

11.3 Example 3: shields.ebl

```
! File shields.ebl
!
! This demonstrates how GROUPCOPY could be used to split up groups into
! germanium and BGO words.
!

INPUT FORMATS

GROUP trigger[255]   (type, mraw, msup, mbgo, muser, mtac)
GROUP    germ[1:64]  (neut, ge2, get, gebd, bgoe, bgot, bgop)
GROUP clovers[65:74] (e1, e2, e3, e4, n, t, b, be, bt, hit)

OUTPUT FORMATS

GROUP trigger[255]    (type, mraw)
GROUP germ   [1:64]   (neut, ge2, get, getbd)
GROUP clovers[65:74]  (e1, e2, e3, e4)
GROUP shields[101:174](sbgoe, sbgot, sbgop)

VALUES

! Define how to copy the bgo items from the germ groupset to the shields
! groupset. The tagging field is left out, so the default of NOTAG will
! be adopted.
GROUPCOPY bgocp TO shields FROM germ
{
 sbgoe :bgoe
 sbgot :bgot
 sbgop :bgop
}

! Define how to copy the bgo items from the clover groupset to the shields
! groupset. Note that another template is needed to copy the clover groupset
! but the destination groupset, shields, is the same as for the previous
! template.
GROUPCOPY bgocp2 TO shields FROM clovers
{
 sbgoe :be
 sbgot :bt
 sbgop :hit
}
```

```
COMMANDS

! Copy the bgo words from the germ groupset.
WHENEVER GROUP = germ THEN
{
  gn = GROUPNO + 100
  GROUPCOPY bgocp gn FROM GROUPNO
}

! Copy the bgo words from the clovers groupset.
WHENEVER GROUP = clovers THEN
{
  gn = GROUPNO + 100
  GROUPCOPY bgocp2 gn FROM GROUPNO
}

STREAM 1 EVENT

FINISH
```

11.4 Example 4: streams.ebl

```
! file streams.ebl
!
! To show writing data to different streams.
!

INPUT FORMATS

GROUP trigger[255]   (type, mraw, msup, mbgo, muser, mtac)
GROUP germ   [1:64]  (neut, ge2, get, gebd, bgoe, bgot, bgop)
GROUP other  [81:90] (e1, e2, e3, e4, t1, t2, t3, t4, mode)

OUTPUT FORMATS

GROUP trigger[255]    (type, evn)
GROUP germ   [1:64]   (neut, ge2, get, getbd)
GROUP other  [81:90]  (mode, e3, t2)
GROUP shields[101:164](sbgoe, sbgot, sbgop)

VALUES

! Define template to split off the bgo items from the germ groupset.
GROUPCOPY bgocp TO shields FROM germ
{
 sbgoe :bgoe
 sbgot :bgot
 sbgop :bgop
}

COMMANDS

! Save the current event number as a trigger word.  This would be needed
! should reassembly of all the data on different streams for the same event
! be required.
evn = EVENTNO

! Copy the bgo words.
WHENEVER GROUP = germ THEN
{
  gn = GROUPNO + 100
  GROUPCOPY USING bgocp TO gn FROM GROUPNO
}
```

```
! Write the germanium and bgo data to one stream and the data from the other
! detector to another.
STREAM 1 trigger, germ, shields
STREAM 2 trigger, other

FINISH
```

A Appendix on Suppression File Format

The suppression file has to convey a large volume of information, i.e. denote which BGO detector suppresses which Ge detector for every Ge detector present in the EUROGAM array. Additional information needed is which word contains the BGO hit pattern, which word should be tagged with a bit in the event of the **tag** keyword being used with the **suppress** command and which word(s) should be used to decide whether a Ge detector has fired. By way of an illustration, a sample suppression file is shown on in §A.2 on page 41

The defaults line is optional. It specifies default values for which item in the input format of the group contains the hit pattern and which item in the output format of the group is to be tagged if the suppression is done in tagging mode. Optionally, it can also specify which item(s) in the i/p group denote(s) that the Ge detector has fired. This information is only needed if the **butnot** facility is used in the file (see below). The values defined in the defaults line will be applied to every group if no other values are specified for that group. In the absence of the defaults line, the item numbers must be specified for every group. The defaults line finishes with a backslash (\) or end row character.

The remainder of the suppression file format uses the numbering scheme for array positions (as distinct from the names of the detectors occupying those positions) as given in the document "Detector Position Numbering Scheme" (J. Simpson, EDOC046). Each Ge position has a unique number in the range 1–72. Each BGO position around the Ge position has a unique letter in the range a–j. So specifying 27 refers to the Ge data from position 27 and 34e refers to the BGO data from position e round Ge position 34.

It should be noted that the Ge array position numbers are the same as the group numbers described in the main body of this document.

A facility has been added (since version 1.1 of this manual) to allow a more complex type of shared suppression to be specified than that described above. This is used if a given BGO element is to be allowed to suppress the current group, but only if another Ge detector has not fired. This uses the keyword **butnot**. Supposing on the row for group 11, we find a term specifying 27e butnot 28. If BGO element 27e fires but Ge 28 does not, then the data for group 11 is suppressed. However, if BGO 27e fires and Ge 28 also fires then group 11 is not suppressed by the example term. The issue of whether a given Ge has fired is resolved by examining the fire item(s) given in the defaults line or column 1 of the row for that detector. So for our example, if the row for group 28 has a fire item specified, that is the one that will be used, otherwise the default value will be used.

The description of the specification below has been altered since version 1.1 of the manual, but **butnot** is an additional facility, so the basic defined syntax has not been altered, since the new instructions are all optional. This means that existing suppression files written using the description in version 1.1 need no changes.

A.1 Suppression Specification Syntax

defaults hititem = <***item-number***> ***tagitem*** = <***item-number***>
⟦ ***fireitem*** = <***fire-term***> ⟧ \
<**row**> r

where

<row> is given by:

<**Ge-position**> <***item-number***> <***item-number***>
⟦<***fire-term***>⟧ | <<BGO-term> r | > r \

and where

<item-number>	is an integer item number in the range 0–32,
<fire-term>	is either <**item-number**> or (<**item-number**> **r**)
<Ge-position>	is an integer value in the range 1–72,
<BGO-term>	is either <**BGO-range**> or (<**BGO-formula**>)
<BGO-range>	is either <**Ge-position**><**BGO-position**> or <**Ge-position**><**BGO-position**> – <**BGO-position**>
<BGO-position>	is a letter in the range a–j,
<BGO-formula>	is given by: <**BGO-list**> **butnot** <**Ge-list**>
<BGO-list>	is a list of <BGO-range>s separated by spaces, and
<Ge-list>	is a list of <Ge-position>s separated by spaces.

Please note: the ⟦ ⟧ round the <fire-term> parts of command are not part of the definition: they indicate that the fire item information is an optional part of an optional piece of syntax.

The **defaults** line is optional. It establishes default values for the hit item, tag item and fire items described below for every row in the table. It saves them being specified repeatedly as in its absence, they must be specified on each row of the table. The default values can be overridden by specifying different values on a row.

The suppression specification is a table of rows. With the exception of the **defaults** line, each row in the table has 2 or more columns. The file is essentially free format with unnecessary white space between components being ignored but it makes things easier to follow to line the data up in some way as has been done in the example. Blank lines and line breaks are ignored and comments are allowed as specified for the event builder language. This is also illustrated in the example: a blank line has been left between EUROGAM faces. Columns are separated by a column separator character, | , referred to as a *bar*. The first column does not start with a bar, but the last column must finish with a bar. A row of data (as opposed to a line in the file) is finished with an *end row* character, \.

Each row of the table pertains to the suppression of a particular Ge group. The first column in a row contains the Ge group number, followed by 2 numbers and optionally another item number or list of item numbers in parentheses. If there is a **defaults** line at the start of the file, then the item numbers are optional: otherwise the first 2 item numbers are compulsory. The first item number is that of the hit pattern word in the group as specified in the **input formats** section. The second is that of the item as specified in the **output formats** section that should be tagged with a suppression bit if the **suppress** command is invoked with the optional **tag** keyword. Note that if tagging mode suppression is desired, then the **tag** keyword must be specified in the suppress command: specification of the tagged item in the suppression file is not sufficient. (Remember also that items are numbered consecutively from zero). The third term gives item numbers that are used to decide if a Ge detector has fired as specified in the **input formats** section. For a phase II clover detector, for example, the event builder might have to test 4 items to decide whether the clover has fired, whereas for an ordinary phase I detector, a single item might suffice. The fire term is only needed if the **butnot** keyword is used in the suppression file.

The data for each Ge position must start on a new line. There must be a row in the table present for every group from which data is expected. There may be data for additional groups present, but no group may have 2 rows of data.

The second and subsequent columns specify sets of BGO positions that suppress the Ge group. Each column is associated with a scheme (column 2 for scheme 1, column 3 for scheme 3, etc.). The BGO positions specified in later columns are *in addition* to those in earlier columns, so no BGO position should be specified twice in the same row. A column may be blank, which says that no additional BGOs suppress the Ge on the current row for the next scheme. The letter ranges should be sensible, i.e. the limits should be in alphabetical order and ranges for the same Ge position should not overlap. The compiler is case-insensitive, so upper or lower case letters may be used to specify BGO positions indiscriminately.

Example

```
69 6 0 | 69a-j | 64A-B 45i 45k 48f 48g |       | 70d 68A |\
```

This example just illustrates the syntax, rather than being realistic. This is a line from a table describing 4 suppression schemes for Ge group 69. The hit pattern is to be found in item 6 and any tagging will be on item 0. In scheme 1 the Ge detector is only suppressed by its own BGO's. In scheme 2, it is also suppressed by some of the BGO's from neighbouring groups. No additional BGO's can do the suppression in scheme 3, but another 2 may do so in scheme 4.

The **butnot** keyword is intended to allow the specification of a list of BGO elements that should be included in the suppression only if a given list of Ge detectors has not fired. Nested **butnot** operations are not allowed. An example of the syntax is shown in the following example.

Example

```
10 6 0   | 10a-j | 9a (11a BUTNOT 12) (11b 12c 12d 14a-d BUTNOT 12 13)|\
11 6 0   | 11a-j |                                                    |\
12 6 0 0 | 12a-j |                                                    |\
13 6 0 (0 1 2 3)
         | 13a-j |                                                    |\
14 6 0   | 14a-j |                                                    |\
```

In scheme 2, Ge detector `10` is suppressed by the BGOs from its own shield and element `a` from shield `9`. Element `a` from shield `11` suppresses group `10` in the absence of a hit in Ge detector `12`. Element `b` of shield `11`, elements `c` and `d` from shield `12` and elements `a`, b, c and `d` from shield `14` suppress group `10` only in the absence of a hit in Ges `12` and `13`. A single fire item has been specified in column 1 of the rows for detector `12` as it is needed by the `butnot` formulas in scheme 2 for group `10`. A list of fire items has been specified for group `13` for the same reason. Note that the specification for group `13` has been split over 2 lines.

A substantial example of suppression file format is given in §A.2. This file starts with a defaults line, so for most groups the hit pattern is received as item `6` and tagging will be on output item `0`. Detector `4` is different to the others for some reason: its suppression bit will be on item `1` instead.

A.2 Suppression Example: js.ss

```
! File js.ss
!
! This example shows how to specify the schemes shown in the document
! ``EUROGAM'' Suppression (J. Simpson, EDOC045).  Scheme 1 is what is
! referred to as ``individual suppression'' and scheme 2 is the
! ``shared suppression''.

DEFAULTS HITITEM = 6 TAGITEM = 0 \

  1 |  1a-j |  2f  2g  2h  6a  6e  5c  5d  5e 35a 35b  7i  7j |\
  2 |  2a-j |  3f  3g  3h  6a  6b  1c  1d  1e 11a 11b 13i 13j |\
  3 |  3a-j |  4f  4g  4h  6b  6c  2c  2d  2e 17a 17b 19i 19j |\
  4 6 1 |  4a-j |  5f  5g  5h  6c  6d  3c  3d  3e 23a 23b 25i 25j |\
  5 |  5a-j |  1f  1g  1h  6d  6e  4c  4d  4e 29a 29b 31i 31j |\

  7 |  7a-j |  8f  8g  8h 12e 12a 11c 11d 11e  1a  1b 35i 35j |\
  8 |  8a-j |  9f  9g  9h 12a 12b  7c  7d  7e 34a 34b 37i 37j |\
  9 |  9a-j | 10f 10g 10h 12b 12c  8c  8d  8e 39a 39b 42i 42j |\
 10 | 10a-j | 11f 11g 11h 12c 12d  9c  9d  9e 41a 41b 14i 14j |\
 11 | 11a-j |  7f  7g  7h 12d 12e 10c 10d 10e 13a 13b  2i  2j |\
 12 | 12a-e  7e  7f  8e  8f  9e  9f 10e 10f 11e 11f |         |\

 13 | 13a-j | 14f 14g 14h 18e 18a 17c 17d 17e  2a  2b 11i 11j |\
 14 | 14a-j | 15f 15g 15h 18a 18b 13c 13d 13e 10a 10b 41i 41j |\
 15 | 15a-j | 16f 16g 16h 18b 18c 14c 14d 14e 43a 43b 46i 46j |\
 16 | 16a-j | 17f 17g 17h 18c 18d 15c 15d 15e 45a 45b 20i 20j |\
 17 | 17a-j | 13f 13g 13h 18d 18e 16c 16d 16e 19a 19b  3i  3j |\
 18 | 18a-e 13e 13f 14e 14f 15e 15f 16e 16f 17e 17f |         |\
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
 37 | 37a-j | 38f 38g 38h 41c 41d 41e 8a 8b 34i 34j          |\
 38 | 38a-j | 37c 37d 37e 33a 33b 65i 65j                    |\
 41 | 41a-j | 37f 37g 37h 44a 44b  6i  6j                    |\

 43 | 43a-j | 44f 44g 44h 47c 47d 47e 14a 14b 10i 10j        |\
 44 | 44a-j | 43c 43d 43e  9a  9b 41i 41j                    |\
 47 | 47a-j | 43f 43g 43h 50a 50b 15i 15j                    |\
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
                        etc
```