LAUTERBACH
DEVELOPMENT TOOLS

PowerIntegrator Programming Guide

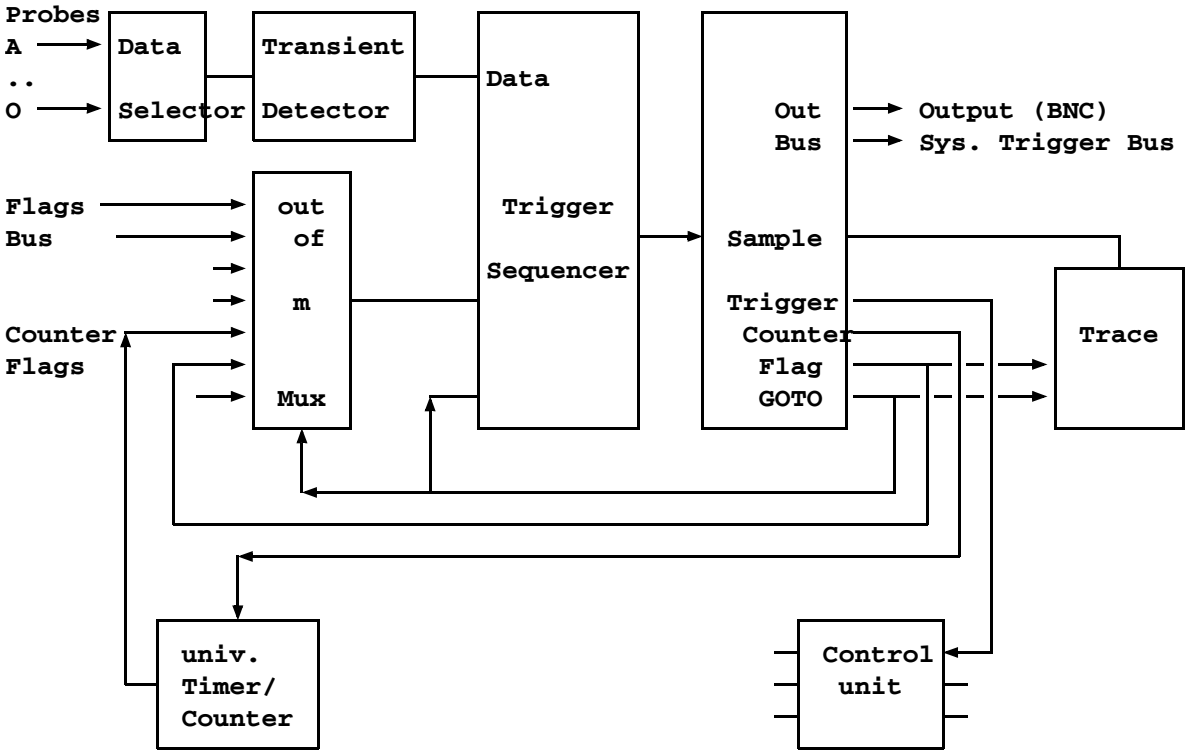
MANUAL

PowerIntegrator Programming Guide

Version 06-Jun-2024

PowerIntegrator

The trigger unit of the timing analyzer is a powerful tool to find complex errors or to sample selective data for advanced measurements. The trigger unit is programmed by an ASCII definition file. The command **Integrator.Program** is used to create a new trigger program. Writing the program is supported by softkeys and online help. The command **Integrator.ReProgram** can be used to load ready-to-run programs in the trigger unit. The commands in this manual refer to the trigger program, unless otherwise mentioned.



Hardware structure of trigger unit

Program Structure

A trigger program for the PowerIntegrator consists of the following parts:

Comments	Are allowed anywhere in the trigger program. They begin with a "//" or ";".
Declarations	Define input events which need to be declared. Such events are flags, data patterns or counters (see also **declarations**).
Instructions	**Instructions** control the action taken by the trigger unit. Usually they are only executed when a defined condition becomes true. A **condition** is the combination of **internal** or external events of the analyzer. An event is the occurrence of a specific internal trigger bus signal or a predefined data pattern.
Levels	The begin of a **level** is defined by the name of the level followed by a colon ":". The end of a level is the begin of the next level or the end of the trigger program. All commands within a level and the global commands are valid while the level is active. Commands outside the level are not active. Only one level can be active at any time. Usually the begin of a trigger program is the first written level or the level with the name "START:".
Global instructions	They are located between declarations and the **first** label, i.e. the first local instruction. They are valid in all used levels. A trigger program may only consists of **global instructions**.
Local instructions	Valid within one trigger level. All **local instructions** defined within a level and all global instructions are checked simultaneously.

Sample Trigger Program

The following sample trigger programs gives an overview about the capabilities of the trigger unit. The program is entered in a window generated by the **Integrator.Program** command.

```
;----- declarations -----

                                          <-
EVENTCOUNTER  max      20.                <-  event counter declaration

;----- global statements -----

Sample.Enable                             <-  sample everything

;----- local statements -----

start:                                    <-  label
           Counter.I   max   if x.high.gt <-  counter enable
           GOTO end          if max       <-  level control

end:
           Trigger

;----- end of trigger program -----
```

Declarations

Declarations are used to assign events to independently selected names (flags, counter or time events). In addition, the event value is specified in the declaration (eg. counter value range, etc.).

Each declaration starts with one of the following keywords: **EVENTCOUNTER**, **FLAGS**, **TIMECOUNTER**. After the keyword the name for the event is defined.

Data Selectors

Not applicable with this device, please use **NAME.Group** or **NAME.Set** or **NAME.Word** instead.

Data selectors are used to trigger on the occurrence of a specific data pattern on the input probes. A declaration consists of a free definable name for this data pattern and the data values. The name is used in the condition for the **data event**. The declaration

```
DATA  high  A:0Yxxxxxxx1
```

defines a data event named "high", which is true if the input signal on probe A Pin 0 has high level. Data declarations may be on a byte, word or long word base. Byte declarations can be defined by ranges:

```
DATA  ascii  A:'A'--'Z'||'a'--'z'
```

If more than one value is one line is used, this operation between the values is an logical AND. The definition

```
DATA  write_sio_control  A:4  cs write
```

is only TRUE, is the WRITE signal, the CHIPSELECT and the address sampled with probe A is true.

Data events can be used in trigger programs with postfix symbols:

Mode	Synonyms	
.HH		true on high level
.HL	**.FALLING**	true on falling edge
.HX		true when coming from high level
.LH	**.RISING**	true on rising edge
.LL		true on low level
.LX		true when coming from low level
.XH	**.1 .HIGH**	true when ending with high level
.XL	**.0 .LOW**	true when ending with low level
.XX	**.X**	always true
.DOUBLE		true on static level
.EDGE		true on falling or rising edge

```
NAME.Set i.f0 write
SELECTOR write_fifo i.write.HIGH
```

Event Counters

Counters can be used to monitor the n.th occurrence of an event. A counter is allocated by a **counter event** declaration. The declaration

```
EVENTCOUNTER  minmax  10.--20.
```

allocates a logical counter event named "minmax", which is true (as an input event) between 10 and 20 occurences of the selected events.

Flags

Flags are free usable flip-flops to store one bit of information. To allocate a **flags** only the keyword and the name of the flag is required. The declaration

```
FLAGS  reset_state, initialized
```

defines two flags having the names "reset_state" and "initialized".

Counters and flags may be displayed while the trace is armed.

Time Counters

To monitor time relations, it is possible to declare **Time events**. The resolution of the timer is 4 ns.. The declaration

```
TIMECOUNTER  after_5ms  5ms
```

allocates one counter named "after_5ms", which is true after 5 ms.

Global Instructions

Global statements are commands, which are not related to a trigger level. The shortest possible trigger program can be made up of one single global statement. For example, the statement

```
S.e
```

(short form for **Sample.enable**) is a valid trigger program.

Statements that are placed **before the first label** are global statements. If declarations are present in a trigger program, global statements must be written after them.

The goal of global statements is to make programming easier. Statements common to all levels need to be entered only once. Each global statement is valid in all levels of the trigger program. The commands for global statements are the same as those for local statements.

Local Instructions

As opposed to global statements, local statements are valid only in one level. Levels begin after the definition of their label and end at the next label or when the trigger program ends. Thus, a label indicates when a new level is started. A level can contain any number of statements.
A statement consists of two parts, the instruction and the condition. The instruction defines what action should be taken, e.g. switch-on trace sampling (**Sample.Enable**), set a flag (**Flag.TRUE**), reset a counter (**Counter.Restart**), or go to the next level (**CONTinue**). The condition defines under which conditions the action is to be taken. For example, the command

```
Sample.Enable if i.high
```

records only, while the data event 'high' is TRUE.
The condition, if defined, must be separated from the instruction by the keyword **IF**. If **no condition** is defined, then the instruction is always executed. Local statements however, are executed only if the level is active. The program

```
start:
     GOTO end if i.high
end:
     TRIGGER
```

will change to level "end" after the data event high is true.

Output Command table

The following instructions control the outputs of the trigger unit:

Instruction	Action	Description
BREAK	**.TRACE**	Stop the recording immediately without delay.
Bus	**.A**	Release trigger bus line A (old command syntax for **Trigger.PODBUS**)
CONTinue		Sequential level switching
Counter	**.Enable** **.Increment** **.OFF** **.ON** **.Restart**	Count cycle (old-fashioned for **.Increment**) Count cycle Counter clock disable Counter clock enable Reload counter
Flag	**.FALSE** **.OFF** **.ON** **.Toggle** **.TRUE**	Reset flag Reset flag (old-fashioned for **.FALSE**) Set flag (old-fashioned for **.TRUE**) Toggle flag Set flag
GOTO		Level switching
Out	**.A** **.B** **.C** **.D**	Set trigger outline TOUT0 Set trigger outline TOUT1 Set trigger outline TOUT2 Set trigger outline TOUT3
Sample	**.Enable** **.OFF** **.ON**	Sample cycle Sample clock disable Sample clock enable
Trigger	**PODBUS** **TRACE** **TRCNT**	Release trigger bus line A (same as old-fashioned **BUS.A**) Start trigger delay for breaking PowerIntegrator (same as old-fashioned **Trigger.A**) Start Counter from Simple Trigger

Events

The actions taken by the trigger unit are controlled by events. An event can be a special trigger bus signal from other devices, e.g. pattern generator or an internal state of the PowerIntegrator. Events can also be the result of a declaration, like counter events or time events. For each instruction in a trigger program (e.g. start trace recording, set flag), conditions can be specified. These conditions are logical combinations of the individual events. The program

```
Sample.Enable if i.a0
```

will make a selective trace as long as the input line i.a0 is high.

Counter Events

The counter counts up if the specified condition is true. A counter event is true, when the counter reaches the declared value. An event range needs two counters. The example samples databytes, which are read from an FIFO in the target. It ignores always the first 1000 bytes after the last write to the FIFO:

```
; Name setting
NAME.Set i.a0 write_fifo
NAME.Set i.a3 reset_fifo
```

```
; Selector declaration
Selector write i.write_fifo.XH
Selector reset i.reset_fifo.XH

; Action
Counter.Increment delay IF i.write
Counter.RESTART         IF i.reset
Sample.Enable           IF delay
```

Data Selectors

All not reserved names are allowed as data selector names.

```
SELECTOR low     I.A0.LOW
SELECTOR high    W.adr 0x55
SELECTOR active  Integrator.F0.XH  Integrator.F1.XL  Integrator.F2.XH
SELECTOR select  I.cs.XL I.astrobe.XH I.write.XL
```

Flags

Flags are flip-flops which can be set or reset, depending on input events. The state of the flip-flops can be used as an input event in the program. The following program will sample data only between the reset and the first write to the device fifo.

```
FLAGS fifo_empty

FLAG.TRUE  fifo_empty IF i.reset_fifo
FLAG.FALSE fifo_empty IF i.write_fifo
Sample.Enable         IF fifo_empty
```

Time Events

The resolution is 4 ns. An event is true, when the counter reaches the declared value. A time range needs two counters. The following program stops sampling 50 µs after the trigger point.

```
TIMECOUNTER delay 50us

start:
     GOTO next IF i.a0.DT
next:
     Counter.Increment delay
     BREAK.TRACE    if delay
```

Other Events

It is not necessary to declare implicit events of the analyzer. The following input events are available:

Event	Description
BUSA	Trigger bus A
FALSE	Always false
SYNC	
TRUE	Always true

Conditions

Conditions are combinations of events, which define when an instruction of the trigger program is executed. Multiple instructions can be linked together in one line to share the same condition. If the condition is missing for an instruction, the condition is always assumed to be 'TRUE'. The program

```
Sample.Enable
```

will produce the same results as

```
Sample.Enable IF TRUE
```

Input events can be combined by standard logical operators:

(…)	
!	for NOT
&&	for AND
^^	for XOR
||	for OR

The brackets have the highest priority, the OR operator has the lowest.

The following two conditions will produce the same results:

```
(v1&&v2)||!(v3&&!v4)
v1&&v2||!v3||v4
```

As instructions can be used more than once in a level or in a statement line, it is possible to have conflicting instructions or conditions. The following trigger program has two such conflicts:

```
START: Counter.ON count1, Counter.OFF count1  IF  i.fifo_write
     GOTO Count_Level
     GOTO Error_Level  if reset_state
Level2:
…
```

Instructions are executed from left to right

In the above example the flip-flop used for controlling the counter will be switched to OFF if the fifo_write condition is true.

Instructions are executed top to down

In the example above this means that the "GOTO" with the condition, which is "always valid", i.e. the jump to "Count_Level", is programmed first. This programming is overwritten by the second "GOTO" with a jump to "Error_Level" only in the case that the condition "reset_state" is true.

The trigger unit remains in the "START" level for of one cycle and will then switch either to the trigger level "Error_Level", or to "Count_Level" depending on the condition "reset_state".
If the order of the "GOTO" statements is changed:

```
GOTO Error_Level  IF  i.fifo_write
GOTO Count_Level
```

then the first statement is completely overwritten.

Global statements have a low priority

Global statements are used, as they would have been typed before any other statement in a trigger level.

Levels

Trigger levels can be used to realize a sequential or non-sequential trigger function. This means, that after one trigger condition has arrived, another condition is checked. The beginning of a level is defined by its label. The end of the level is the label of the following level, or the end of the program. All statements located between these boundaries are part of that level.

The levels determine which statements are active at the same time. All conditions for instructions in a level are checked in parallel during each cycle and all instructions whose condition is TRUE are executed. Only one level can be active at any time. The current level is recorded in the trace and can be viewed in real-time in the integrator state window.

The instructions **CONTinue** and **GOTO** will change the level.

CONTinue

The **CONTinue instruction** can switch to the next program level following the current one. If no level follows, then "CONTinue" is the same as the **Trigger.TRACE instruction**, i.e. the trace is stopped. In the example the analyzer will change to level "infunc" after "fifo_reset" and stop the analyzer after "fifo_write".

```
start:
     CONTINUE IF i.fifo_reset

infunc:
     CONTINUE if i.fifo_write
     SAMPLE
```

GOTO

The **GOTO instruction** can switch to any level. The following program will change to the level "init" when the "fifo_reset" is true is, and change back to "start" on "fifo_write". The probe data is sampled only when the trigger unit is in the "init" level.

```
start:
     GOTO init IF i.fifo_reset

init:
     GOTO start if i.fifo_write
     SAMPLE
```

The **TRIGGER.TRACE** respectively **BREAK.TRACE** instruction causes the trace to stop. Breaking the analyzer means stopping recording and deactivating the trigger unit.

```
start:
     TRIGGER IF fifo_write
```

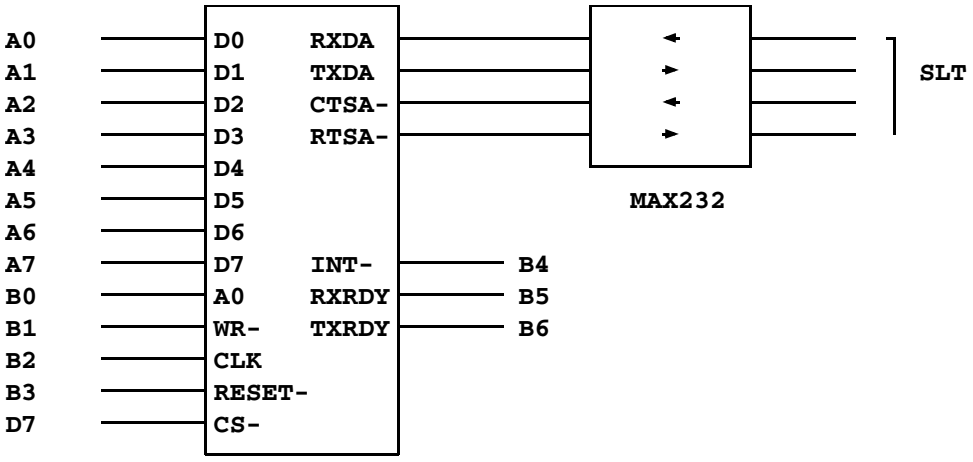
When implementing multiple level change instructions in one trigger level, the instruction priority must be observed. Instruction priority rises for instructions written later in the level, global instructions have a low priority. The following example shows this relation:

```
; Declarations
selector fifo_reset  i.reset.XH
selector fifo_write  cs write a:4
selector dma         b:0Yxxxxxxx0
selector nmi         d:0Yxxxxxx0x
…

; global statements

sample
…

; local statements

level0:
     continue    if fifo_reset

level1:
     goto level3 if nmi
     goto level3 if dma
     continue    if dma&&nmi

level2: …

level3: …
     trigger.trace
```

When the analyzer is in level "level1" and assuming that during "dma" active "nmi" also gets active, then the program branches to "level2" and not to the trigger level.

Programming Examples

All programming examples are explained by a basic SIO circuit:



First the probes are connected and the names of the input signals are defined:

```
name.group  BUS_DATA  i.a0 i.a1 i.a2 i.a3 i.a4 i.a5 i.a6 i.a7
name.set i.a8  i.BUS_A0
name.set i.a9  i.BUS_WR -
name.set i.a10 i.CLK
name.set i.a11 i.RESET -
name.set i.a12 i.INT -
name.set i.a13 i.RXRDY
name.set i.a14 i.TXRDY
name.set i.a31 i.CS -
```

Selective recording is done with the instruction **Sample.Enable**. Normally do not use **Sample.ON** and **Sample.OFF**, because the state of this line cannot be monitored in the trace. Using two levels instead is usually a better solution.

```
sample.enable if true
```

or

```
s.e
```

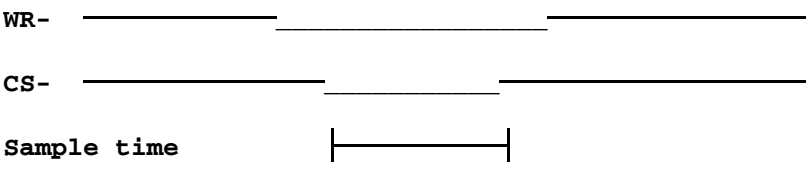
All input data is sampled.

Samples if the SIO is selected and the write line is active. The same trace function can be defined by input

```
; Selector declaration
Selector ValidRecord i.BUS_WR.XL i.CS.XL

; Action
Sample.enable if ValidRecord
```

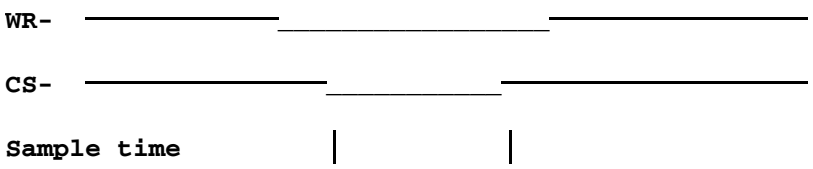
masks:

```
DATA SIO_WRITE  B:0Yxxxxxx0x D:0Y0xxxxxxx

Sample.enable if Group.SIO_WRITE
```



In synchronous mode the data will be sampled on the clock of D7 only. The channel A must be switched to synchronous mode:

```
A.SYnch  A  ON

DATA SIO_WRITE  B:0Yxxxxxx0x

Sample.enable if SIO_WRITE.S
```



Stopping the PowerIntegrator

There are some reasons which can stop the analyzer recording:
Trace Full

Stopping the recording when the analyzer is full can be achieved by selecting **Stack** operation mode in the integrator control window (command **Integrator.Mode Stack**).

```
intergrator.mode stack
```

The analyzer stops recording independent of the current logical level.

Trigger.TRACE or **BREAK.TRACE**

Recording stops at a specified condition defined by the **Trigger** statement. The trigger delay may be defined between 0 and 100% of the trace storage.

The statement **BREAK.TRACE** is a synonym for the combination of a Trigger statement with a trigger delay of 0.

```
trigger.trace if SIO_WRITE
```

Stimulating Output Lines

The **instruction OUT** can control the external trigger outputs of the PowerIntegrator. These lines can be used to trigger external timing analyzers or oscilloscopes or generate stimuli signals for the target hardware. The example generates an output signal at the connector TOUT0 on the top of the analyzer unit every time the interrupt line becomes active:

```
OUT.A if i.INT.gt
```

Using the Internal Trigger Bus

The **intertrigger BUS** can trigger other systems of the TRACE32 system. The inter-trigger bus of the system can be used to trigger the pattern or pluse generator by the timing analyzer. The following example triggers the pattern generator when an access to a specific address is made. In the pattern generator the **BusA** line must be selected as trigger source (command **Pattern.TSELect BusA**).

```
NAME.Set i.a0 i.SIO_WRITE

BUS.A  IF  i.SIO_WRITE                ; the bus line A is activated on
                                      ; write cycles to the SIO
```

The PowerIntegrator can also be triggered by other analyzers. The event **BUSA** can be used for this purpose. When the line BUSA is released from the trace analyzer, the PowerIntegrator can be controlled by the trace analyzer. In the following example the PowerIntergrator records for 1 ms after the trigger event from the trace analyzer.

```
; declaration
TIMECOUNTER delay  1ms              ; delay definition

; local instruction
L00: CONTinue     IF BUSA           ; wait until the line BUSA is active

L01:
     Counter.Increment delay        ; activate delay counter
     Trigger.TRACE  IF delay        ; break after 1 ms
```

The keyword are **TIMECOUNTER** or **EVENTCOUNTER** for the declaration and the **instruction COUNTER** controls the counter. Time and event counters need a declaration for assigning names and initial values. Counters are always assigned for the whole trigger program, even the control of the counter is level specific. The following program defines a counter, which will display the time a program is running in a function. The total time can be viewed in the integrator state window.

The following example triggers on a write cycle longer than 10.us:

```
NAME i.a0 i.SIO_WRITE

; declaration
TIMECOUNTER timeout 10.us

Counter.Restart   timeout  IF  !i.SIO_WRITE
Counter.Increment timeout  IF   i.SIO_WRITE

Trigger.TRACE              IF  timeout
```

In the next example counts the number of write accesses to the SIO:

```
; declaration
SELECTOR     SIO_WRITE   i.a0.LH
EVENTCOUNTER write_cycle

; counter operation
Counter.Increment write_cycle  IF  i.SIO_WRITE
```

This example stops the trace after 1000 write accesses to the SIO:

```
; declaration
SELECTOR     SIO_WRITE   i.A0.LH
EVENTCOUNTER write_cycle 1000.

; counter operation
Counter.Increment write_cycle  IF  i.SIO_WRITE
Trigger.TRACE                  IF  write_cycle
```

Using Flags

The keyword **FLAGS is used in the declaration** and the **instruction FLAG** can modify the value of a flag. Flags are useful for remembering the occurrence of a certain state. In some cases they can replace the use of multiple trigger levels in an integrator trigger program. The following example will monitor the state of an I/O port in real time. The state of the port can be viewed in the integrator state window:

```
NAME.Group Group.WRITE_TO_CONTROL i.a0 i.a1+ i.a8
NAME.Set    i.63+ i.TX_EN

; declaration
FLAGS TX_ENABLE                 ; declaration of 1 flag

FLAG.TRUE  TX_ENABLE  IF   i.TX_EN&&Group.WRITE_TO_CONTROL
FLAG.FALSE TX_ENABLE  IF  !i.TX_EN&&Group.WRITE_TO_CONTROL
```

Switching Trigger Levels

The instructions **CONTinue** or **GOTO** can be used to change the level of the trigger unit. The instruction **Trigger.TRACE** will disable the trace and the trigger unit.

```
; declaration
…

; global instructions
…
```

```
; local instructions
LEVEL0: …
     CONTinue  if i.SIO_WRITE     ; change sequential to the next logical
                                  ; level

     …
LEVEL1: …
     CONTinue  if CNT_Limit       ; change to the next logical level if
                                  ; the counter CNT_Limit is ready

      GOTO LEVEL0 if i.BB         ; otherwise jump return to the level
                                  ; LEVEL0 if the data event BB is true

      …
 LEVEL2: TRIGGER.TRACE if         ; stop recording (user program remains
 DELAY_CNT                        ; running) if the counter DELAY_CNT is
                                  ; ready

START: GOTO LEVEL0 if i.INT       ; this is the start level after analyzer
                                  ; init
                                  ; Wait for an interrupt acknowledge,
                                  ; then jump to the level LEVEL0
```

SELECTOR — Data selectors

Format:	**SELECTOR** *<name>* *<pin>* **.** *<mode>* | *<w.name>* *<value>* …
<pin>:	**i.a0** | … | **i.f15** | **i.j0** | … | **i.o15** | **clcka** | … | **clckf** | **clckj** | … | **clcko** | *<name declared with name.set>*
<mode>:	**0** | **1** | **DOUBLE** | **EDGE** | **FALLING** | **HH** | **HIGH** | **HL** | **HX** | **LH** | **LL** | **LOW** | **LX** | **RISING** | **XH** | **XL**
<w.name>:	Word declared with “name.word”
<value>:	0 | 1 for pins, integer value or bit mask for words

A **data selector** named Port_1 with the value 0x55 on the pins i.a0 (LSB) to i.a7 (MSB).

```
NAME.WORD    adr i.a0 i.a1 i.a2 i.a3 i.a4 i.a5 i.a6 i.a7
SELECTOR Port_1  w.adr 0x55
```

A data selector named Port_2 with the value 0x55 on the pins i.a0 (LSB) to i.a7(MSB) and with the bit mask 1010xx1x on pins i.a8 (LSB) to i.a15 (MSB):

```
NAME.WORD    adr  i.a0 i.a1 i.a2 i.a3 i.a4 i.a5 i.a6 i.a7
NAME.WORD    data i.a8 i.a9 i.a10 i.a11 i.a12 i.a13 i.a14 i.a15
SELECTOR Port_2  w.adr 0x55 w.data 0y1010xx1x
```

All alphabetic characters of the ASCII character set (lower case and upper case) are assigned to a data selector called "ascii":

```
SELECTOR ascii w.data ('a'--'z')||('A'--'Z')
```

Logical pin names can be used like this

```
NAME.Set i.f0 write
SELECTOR write_fifo i.write.HIGH
```

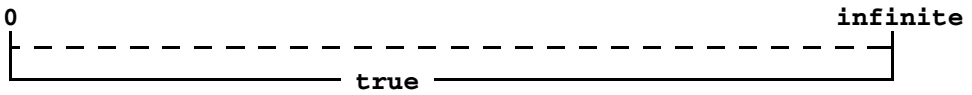
Format: **EVENTCOUNTER** <*name*> [<*event*>]

Any name can be assigned to the counter, as long as it doesn't conflict with the reserved names of other events. The physical counters are selected automatically by the system, depending on their usage. Three universal counters for event counting are available on the PowerIntegrator. They have a width of 45 bits. If a event counter reaches its declared value it will stop automatically. The event counters can be reloaded in real-time. However, program-dependent dead times can result. The default value is the maximum time.

The current value of the counters are visible in real-time in the integrator state window.

Endless Counter

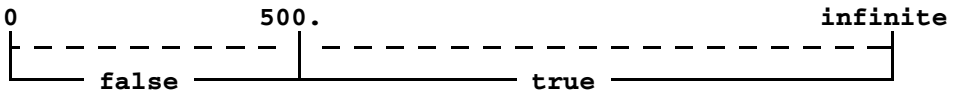
```
EVENTCOUNTER Evcntr_2 0
Counter.increment Evcntr_2 IF true
Sample.enable              IF Evcntr_2
```



Declaration of an event counter called "Evcntr_2", count argument 0. The counter is always enabled but it never counts because it immediately reaches the declared value. In this example the integrator begins sampling immediately .

Event TRUE after n Clocks

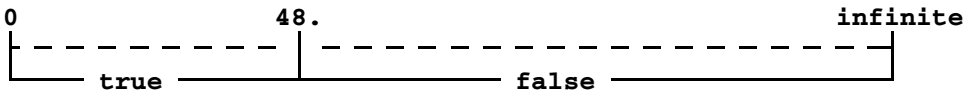
```
EVENTCOUNTER CYCLE_CNT 500.
Counter.increment CYCLE_CNT  IF  true
Sample.enable                IF  CYCLE_CNT
```



Declaration of an event counter called "CYCLE_CNT". The counter is always enabled and counts all cycles. The integrator begins sampling after a delay of 500 cycles.

Event TRUE till n Clocks

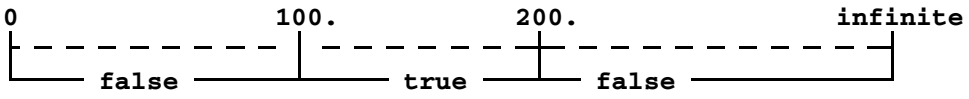
```
EVENTCOUNTER NR_cnt 0x0--0x30
Counter.increment NR_cnt IF true
Sample.enable            IF NR_cnt
```



Declaration of an event counter called "NR_cnt", event argument is 30 (hex). The counter is always enabled. The integrator begins sampling immediately and stops recording after 48 sampled cycles.

Event Windows

```
EVENTCOUNTER EV_Range 100.--200.
Counter.increment EV_range IF true
Sample.enable             IF EV_range
```



Declaration of an event counter called "EV_range" with an event range from 100 to 200. The counter is always enabled and counts all cycle. The integrator begins sampling after 100 cycles and stops recording 100 cycles later. Two physical counters are used by the trigger unit.

Format: **EXTERNSYNCCOUNTER** <*name*> [<*event*>]

Any name can be assigned to the counter, as long as it doesn't conflict with the reserved names of other events. The physical counters are selected automatically by the system, depending on their usage. Three universal counters for counting external clock cycles are available on the PowerIntegrator. They have a width of 45 bits. A synchronous counter is synchronized to an external clock, which means that the counter will count only at the start of an external clock cycle. Because of this the events used in the conditions for the controlling instructions for the counter should use signals, which are also synchronized to the external clock:

```
name.set i.a0 d0 sync
name.set i.a1 d1 sync
name.word  data i.b0 i.b1

SELECTOR           datahigh  w.data 0x3
EXTERNSYNCCOUNTER  threehigh 3.
Counter.increment threehigh IF datahigh
Sample.enable               IF threehigh
```

The counter will become true, after three clock cycles in which the i.b0 and i.b1 pins are high.

If events used in a condition for an instruction for a synchronous counter, are not synchronized to the external clock.

Format: **FLAGS** <*name*> …

Flags are Flip-flops which can be controlled and read by the trigger unit. The hardware for the flags is assigned automatically by the system, depending on their usage. There are a maximum of 2 flags available.

After programming the trigger unit, or after the command **Integrator.Init** all flags are set to off. Flags can be set, reset or toggled.

The following program samples only, when the flag 'init_state' has the value TRUE:

```
FLAGS init_state
SELECTOR  reset_fifo  i.c0.high
SELECTOR  write_fifo  i.c1.low i.c2.high

FLAG.TRUE  init_state if reset_fifo
FLAG.FALSE init_state if write_fifo

SAMPLE.ENABLE         if init_state
```

TIMECOUNTER Time counter

Format: **TIMECOUNTER** <*name*> [<*time*>]

Any name can be assigned to the counter, as long as it doesn't conflict with the reserved names of other events. The physical counters are selected automatically by the system, depending on their usage. Three universal counters for timing measurements are available on the PowerIntegrator. They have a resolution of 10 ns (>50 MHz) respectively 20 ns (50 MHz) and a width of 45 bits. If a time counter reaches its declared value, it will be stopped automatically. The timers can be reloaded in real-time. However, program-dependent dead times can result. The default value is the maximum time.

The current value of the counters can be viewed in real-time in the integrator state window.

Time values can be entered in the following units:

Nanoseconds (ns)

Microseconds (µs)

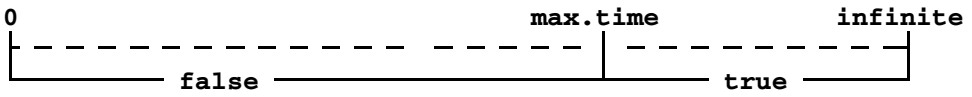
Milliseconds (ms)

Seconds (s)

Kiloseconds (ks)

Timer running till Overflow

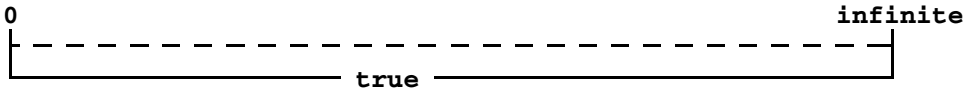
```
TIMECOUNTER Timer_1
Counter.increment Timer_1 IF true
Sample.enable             IF Timer_1
```



Declaration of a time counter called Timer_1 without time argument. The counter is always enabled and counts every time. After the maximum time the integrator starts sampling input data.

Always running Timer

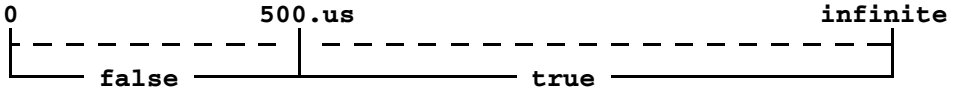
```
TIMECOUNTER Timer_2 0ms
Counter.increment Timer_2 IF true
Sample.enable             IF Timer_2
```



Declaration of a time counter called Timer_2, time argument 0ms. The counter is always enabled but it never counts because it immediately reaches the declared value.

Timer TRUE after Time

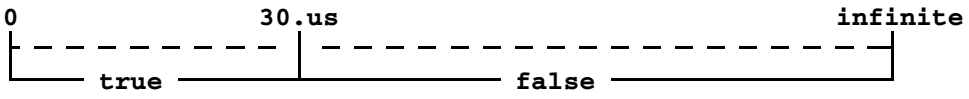
```
TIMECOUNTER Timer_A 500.us
Counter.increment Timer_A IF true
Sample.enable             IF Timer_A
```



Declaration of a time counter "Timer_A", time argument is 500us. The counter is always enabled. The integrator begins sampling after a time delay of 500us.

Timer TRUE till Time

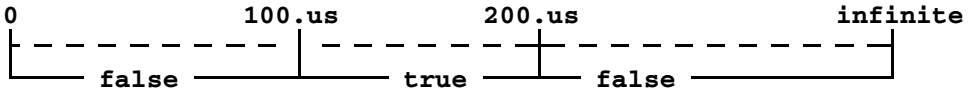
```
TIMECOUNTER Timer_B 0.us--30.us
Counter.increment Timer_B IF true
Sample.enable             IF Timer_B
```



Declaration of a time counter called "Timer_B". The counter is always enabled. The integrator begins sampling immediately and stops recording after a time of 30us.

Time Windows

```
TIMECOUNTER Timer_C 100.us--200.us
Counter.increment Timer_C if true
Sample.enable             if Timer_C
```



Declaration of a timer called "Timer_C" with a time range of 100 … 200 µs. The counter is always enabled and counts every time. The integrator begins sampling after 100 µs and stops recording 100 µs later. Two physical counters are used by the trigger unit.

BREAK — Trace stop

Format: **BREAK**[**.TRACE**] [**IF** <*condition*>]

The analyzer breaks and stops recording immediately, independently from the before defined trigger delay. The value from a before used **Integrator.TDelay** command will be ignored. The use of a BREAK command inside a trigger program resets automatically the trigger delay to 0. The analyzer can be read out when in break state, similar to the OFF state. The break level is reset by the command **Integrator.Init**. See also the command **Trigger.TRACE**.

```
…
BREAK.TRACE if fifo_reset
…
```

The integrator breaks, whenever the state "fifo_reset" is true.

Bus — Bus trigger

Format: **Bus.**<*mode*> [**IF** <*condition*>]

<*mode*>: **A**

In order to be able to trigger more than one TRACE32 system, several trigger lines are available on the inter-trigger bus. A synonym for this command will be **Trigger.PODBUS**.

A Activates podbus trigger line A.

Format: **CONTinue** [**IF** <*condition*>]

A sequential **level** switch (to the next written level) will be done, when the specified condition is true. If no further written level is present, the trace is stopped.

In the example the analyzer will change to level "init" if the data pattern "fifo_reset" is true.

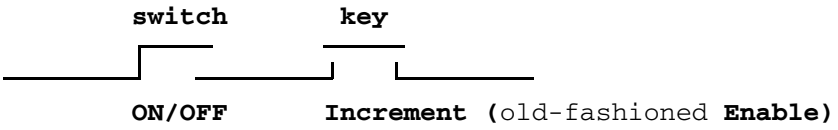
```
SELECTOR  fifo_reset  I.A0.XL

start:  CONTINUE IF fifo_reset

init:
```

Format:	**Counter**[.<*mode*>] <*counter_name*> [**IF** <*condition*>]
<*mode*>:	**Increment** **OFF** **ON** **Restart**

Control the trigger units counters. The instructions Counter.ON and Counter.Increment are programmed automatically, if they are not used in the trigger program. The counters have to be declared according to their function (see also declaration EVENTCOUNTER, EXTERNSYNCCOUNTER or TIMECOUNTER and chapters Counter Events or Time Events).

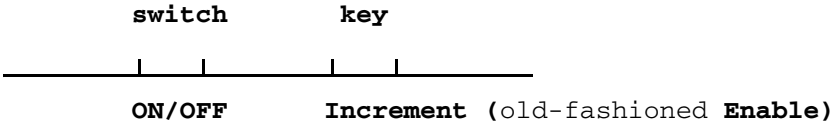
Increment	Releases counters when the specified condition is matched.
OFF	Switches the enable Flip-flop OFF.
ON	Switches the enable Flip-flop ON.
Restart	The counter is reset to zero.

The instructions **ON**, **OFF** and **Increment (**`old-fashioned` **Enable**) can be seen as a controlled switch and a key in series. If the switch is closed (**Counter.ON**) it remains closed till it is opened by **Counter.OFF**. The key is closed only for the cycle which meets the specified condition, i.e. an event counter will make one step.

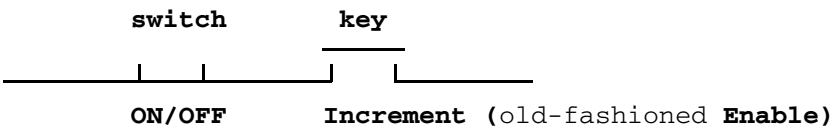


Only if the switch and the key are closed the counter advances.

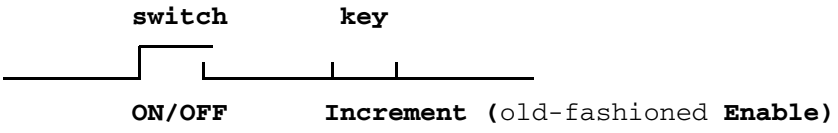
If neither ON/OFF nor Enable are used in the complete trigger program, the switch and the key are closed, that means the counter counts time or events (cycles) depending on its declaration.



If only Enable is used in the trigger program, the switch ON/OFF is closed automatically, that means counting is controlled only by Increment (old-fashioned Enable).



If only ON/OFF is used in the trigger program, the key Increment (old-fashioned Enable) is closed automatically, that means counting is controlled only by ON/OFF.



NOTE: In all cases during the first cycle the switch ON/OFF is closed!

Counter CYCLE_CNT is running all the time

```
; declaration
EVENTCOUNTER CYCLE_CNT

; global or local instruction
Counter.Increment CYCLE_CNT
```

Counter "count_reset" is incremented by 1 every time, the input line A0 is pulled to low.

```
; declaration
SELECTOR     reset_fifo   i.a0.XL
EVENTCOUNTER count_reset

; global or local instruction
Counter.Increment count_reset if i.reset_fifo
```

Counter "reset_puls" is measuring the pulse width of the reset signal.

```
; declaration
SELECTOR   reset_fifo i.a0.XL
TIMECOUNTER reset_puls

start:
     GOTO res_valid IF reset_fifo
res_valid:
     Counter.increment reset_puls
     GOTO next      IF !reset_fifo
next:
     TRIGGER.TRACE
```

The counter "ascii_count" is incremented on synchronous clocks with a valid upper-case ASCII character on probe A. The counter stops at 100. and the trace stops.

```
; declaration
SELECTOR     upper_ascii w.databusB0 'A'--'Z'
EVENTCOUNTER ascii_count 100.

; global or local instruction
Counter.increment ascii_count.s
TRIGGER.TRACE  IF ascii_count
```

If the pulse width of the "cs_fifo" signal is more than 500 ns, the timing trace will stop

```
; declaration
SELECTOR    cs_fifo  I.A0.XH
TIMECOUNTER time_out 500ns

; global instruction
Sample.enable
Counter.Restart    time_out IF !cs_fifo
Counter.increment  time_out IF cs_fifo
TRIGGER.TRACE               IF time_out
```

The first write_fifo event must be within 100 us after the fifo_reset state. Otherwise the integrator is stopped. The counter max_time measures the real time between 'reset' and 'write' on the break condition.

```
; declaration
TIMECOUNTER first_write 100us
TIMECOUNTER max_time
SELECTOR reset_fifo  i.a0.XL
SELECTOR write_fifo  i.a1.Xl i.a2.XH

start:
     GOTO reset_state IF  reset_fifo
reset_state:
     Counter.Restart first_write
     Counter.Restart max_time
     GOTO no_reset    IF  !reset_fifo
no_reset:
     Counter.increment first_write
     Counter.increment max_time
     GOTO start       IF  write_fifo
     Trigger.TRACE    IF  write_fifo&&!first_write
```

Format:	**Flag.**<*mode*> <*name*> [**IF** <*condition*>]
<*mode*>:	**FALSE** **Toggle** **TRUE**

Flags are used to mark event occurrences. Flags have to be declared at the beginning of a trigger program (see chapter FLAGS). The default state at the beginning is OFF. The current state of the used flags is visible in real time in the integrator state window.

FALSE	Resets the flag.
TRUE	Sets the flag.
Toggle	Reverses the current state.

Set Flag1 if timer_1 has not expired.

```
; declaration
FLAGS Flag1

; global or local instruction
Flag.TRUE Flag1 if !timer_1
```

Toggle Flag4 if data_event occurs.

```
Flag.Toggle Flag4 if i.data_event
```

GOTO Level switching

Format:	**GOTO** <*level*> [**IF** <*condition*>]
<*level*>:	**name** **START**

Change the current level of the trigger unit. GOTO may be used more than once in a level.

The first level which is active after the trigger unit has been programmed is the start level. It is defined by the label "START:". If no level has been defined this way, then the first level in the program is the start level. The level marked with "START" has to be the first level written in the program.

On the PowerIntegrator there are 4 levels available.

```
Start:counter.restart  int_count
     goto ll8

      …

LL5:  sample.enable              if  dma
     GOTO ll8                   if  dma&&last_transfer

      …

LL8:  sample.enable
     counter.increment  int_count if  int_adr

      …
```

Out — Output control

Format:	**Out.** <*mode*> [**IF** <*condition*>]
<*mode*>:	**A | B | C | D**

Four signals can be generated to trigger other devices (e.g. analyzers or oscilloscopes) or to stimulate the target hardware. These signals are accessible via socket connectors at the integrator box.

A, B, C, D — Activates the universal output TOUT0..TOUT3 at the top of the integrator box.

```
Out.A if i.time_out.gt
```

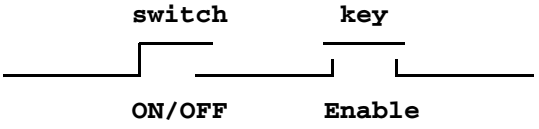
Release trigger line TOUT0 if the event time_out becomes true.

Format:	**Sample**[.<*mode*>] [**IF** <*condition*>]
<*mode*>:	**Enable** **OFF** **ON**

Controls trace memory recording. The instructions **Sample.ON** and **Sample.Enable** are programmed automatically, if they aren't used in the trigger program. This instruction does not effect the recording of the emulator trigger event and the first and last cycle before the user program stopped.

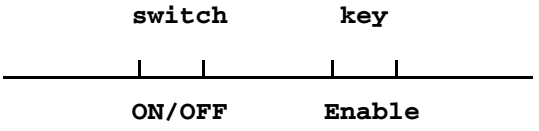
Enable	Releases trace memory for recording when the specified condition is true.
OFF	Disables the Flip-flop for sampling.
ON	Enables the Flip-flop for sampling.

The instructions **ON**, **OFF** and **Enable** can be seen as a controlled switch and a key in series. If the switch is closed (**Sample.ON**) it remains closed till it is opened by **Sample.OFF**. The key is closed only for the cycle which meets the specified condition, i.e. one bus cycle is stored in the trace buffer.

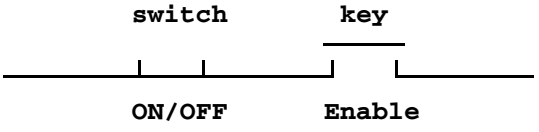


Only if the switch and the key are closed sampling is done.

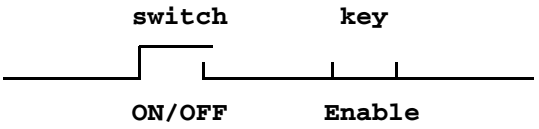
If neither ON/OFF nor Enable are used in the complete trigger program, the switch and the key are closed, that means all cycles are recorded (Implicit global "Sample.ON if true" and "Sample.Enable if true").



If only Enable is used in the trigger program, the switch ON/OFF is closed automatically, that means sampling is controlled only via the Enable (Implicit global "Sample.ON if true").



If only ON/OFF is used in the trigger program, the key Enable is closed automatically, that means sampling is controlled only via ON/OFF (Implicit global "Sample.Enable if true").



NOTE: In all cases during the first cycle the switch ON/OFF is closed!

The following statements are equal and will sample all cycles:

```
Sample.Enable if true
Sample.Enable
S.E
s
```

Sample only if line i.a0 is high:

The trace starts and waits in Level0 without recording till the appearance of the int1 line.

```
NAME.Set i.a0 i.high

Sample.Enable if i.high
```

```
; declaration area
SELECTOR int1_low  I.C3.0
…

; local area
Level0: s.e      if i.int1_low
     continue if i.int1_low
Level1: S.e
     …
```

Format:	**Trigger.** <*mode*> [**IF** <*condition*>]
<*mode*>:	**A** | **P** | **PODBUS** | **TRACE** | **TRCNT**

Trigger other systems of the PowerIntegrator.

PODBUS	Releases a signal on the inter-trigger bus (**BUS.A**).
TRACE, **A (obsolete)**	Starts the trigger delay counter defined by the command **Integrator.TDelay**
TRCNT	Release pulse on universal counter input line.

Trigger.PODBUS:

In order to be able to trigger more than one TRACE32 system, a special trigger line is available on the inter-trigger bus. A synonym for this command will be **BUS.A**.

Trigger.TRACE:

When the trace is triggered, it stops recording after the, with the command **Integrator.TDelay** defined trigger delay. The analyzer can be read out when in break state, similar to the OFF state. The break level is reset by the command **Integrator.Init**. See also the command **BREAK.TRACE**.

```
…
Trigger.TRACE if fifo_reset
…
```

The trace stops recording, whenever the state "fifo_reset" is true.

PowerIntegrator Programming Language Syntax

NOTE: The following symbols are meta-symbols belonging to the formalism and not symbols of the trigger programming language.

[ ]	0 up to 1 iteration of the expression included (the expression can be omitted)
{ }	1 up to infinite iteration of the expression inside (the expression must be written at least once)
( )	summary (summarize alternatives)
|	separates alternatives
:	the *name* (nonterminal symbol) on the left can be substituted with the expression on the right
text/ **text**	the characters written in bold letters are terminal symbols which cannot be substituted any more (the characters have to be typed in this way)

The meta symbols mustn't written in the trigger program.
The timing analyzer programming language starts with the nonterminal symbol *ppta_prog*.

```
ppta_prog:      [{EOL}] [decls]  [globals] [{locals}] {EOL} EOF


decls:          [(eve_dec | flg_dec | sel_dec | tim_dec)] [comment]
                {EOL} [decls]
eve_dec:        EVENTCOUNTER       name1 [ int ]
ext_dec:        EXTERNSYNCCOUNTER  name1 [ int ]
flg_dec:        FLAGS              name2 [{[,] name2}]
sel_dec:        SELECTOR name3 {{dataname_prefix1 . dataname_postfix|
                                dataname_prefix2 . dataname_postfix}
                               [. mode]   [{int | range | bitmask}] }
tim_dec:        TIMECOUNTER        name1 [ time ]


globals:                instr
locals:         label [instr]
label:          (name | START) :


instr:          [comlist] [comment] {EOL} [instr]
comment:        (// | ;) text
comlist:        command [{[,] command}] [ IF condition]
command:        c_break | c_bus | c_continue | c_counter | c_flag |
                c_goto | c_out | c_sample | c_trigger
c_break:        BREAK [.TRACE]
```

c_bus: (**B** | **BUS**) [**.A**]

c_continue: (**CONT** | **CONTINUE**)

c_counter: (**C** | **COUNTER**) [**.** (**I**|**INCREMENT**|**OFF**|**ON**|**R**|**RESTART**)] {*name1*}

c_flag: (**F** | **FLAG**) **.** (**FALSE**|**T**|**TOGGLE**|**TRUE**) {*name2*}

c_goto: **GOTO** (*name* | **START**)

c_out: (**O** | **OUT**) **.** (**A** | **B** | **C** | **D**)

c_sample: (**S** | **SAMPLE**) [**.** (**E**|**ENABLE**|**OFF**|**ON**)]

c_trigger: (**T** | **TRIGGER**) [**.** (**A**|**P**|**PATTERN**|**PODUS**|**PULSE**|**TRACE**|**TRCNT**)]

condition: *t1* {[**||** *t1*]}

t1: *t2* {[**^^** *t2*]}

t2: *t3* {[**&&** *t3*]}

t3: (**!** *t3*) | **(** *condition* **)** | *name1* | *name2* | *dataname_prefix1* **.** *dataname_postfix* **.***mode* | *name3* [**.***mode*] | *inline_name2*

dataname_prefix1: **I** | **INTEGRATOR**

dataname_prefix2: **W** | **WORD**

dataname_postfix: **A0..F15** | **J0..O15** | **CLCKA..CLCKF** | **CLCKJ..CLCKO** | from user with **NAME.Group** or **NAME.Set** or **NAME.Word** defined names

mode: **0** | **1** | **DOUBLE** | **EDGE** | **FALLING** | **HH** | **HIGH** | **HL** | **HX** | **LH** | **LL** | **LOW** | **LX** | **RISING** | **XH** | **XL** | **X** | **XX**

inline_name2: **BUSA** | **FALSE** | **SYNC** | **TRUE**

name1,name2,name3: is chosen from the user and must correspond with the 'C'-name conventions

int: syntax described in the **Operation System User's Guide**

time: dto.

text: all characters excepted EOL and EOF