LAUTERBACH
DEVELOPMENT TOOLS

PowerProbe Trigger Unit Programming Guide

MANUAL

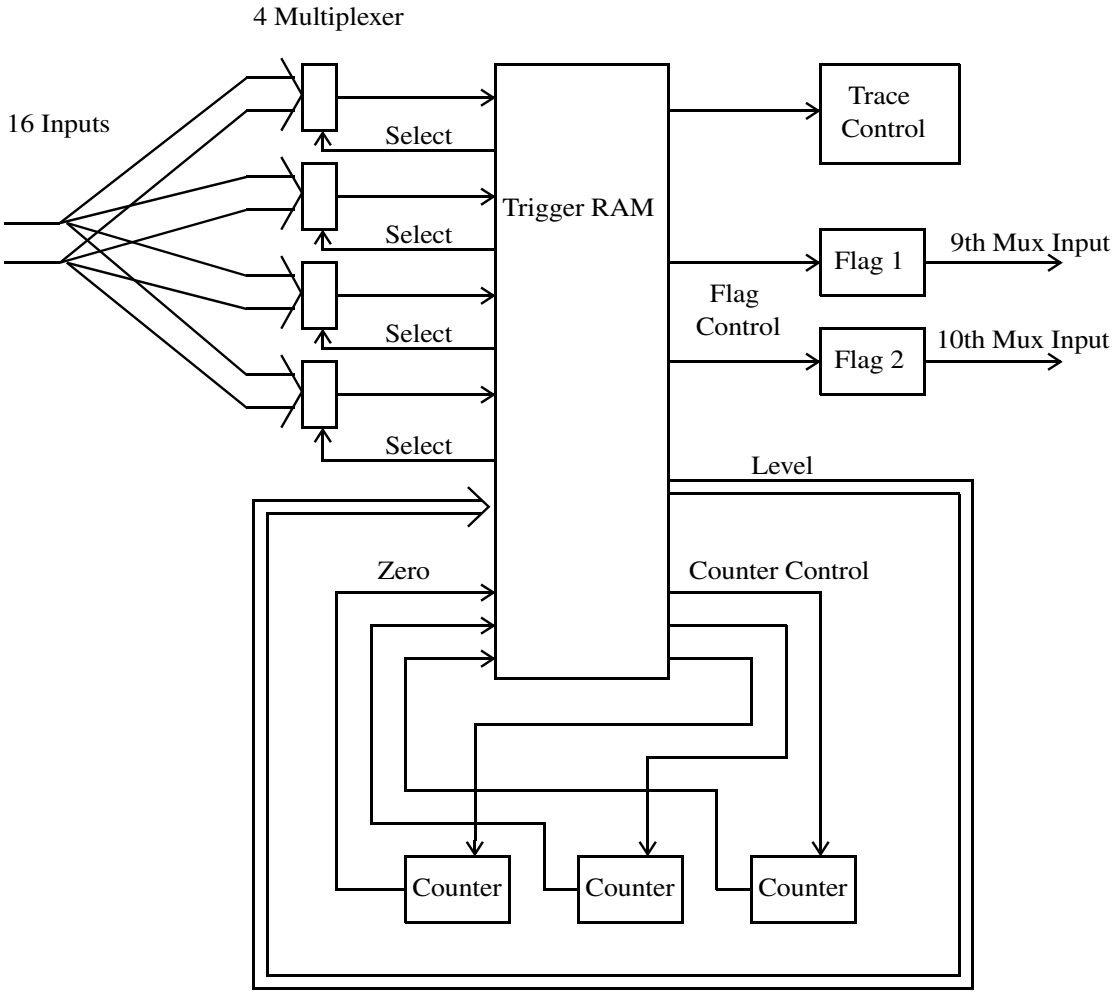
PowerProbe Trigger Unit Programming Guide

PowerProbe Trigger Unit Programming Guide

Version 06-Jun-2024

PowerProbe Programming

The trigger unit of the PowerProbe 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 **Probe.Program** is used to create a new trigger program. Writing the program is supported by softkeys and online help. The command **Probe.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 PowerProbe 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 PowerProbe. 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 **Probe.Program** command.

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

SELECTOR      strobe   x.0 1              <------ data selector declaration
EVENTCOUNTER  max      20.                <------ event counter declaration

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

Sample.Enable                            <------ sample everything

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

start:                                   <------ label
      Counter.Increment max  IF strobe   <------ counter increment
      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 (e.g. counter value range, etc.).

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

Data Selectors

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 the data selector and a pattern definition. The name is used in conditions for the **data event**. The declaration

```
SELECTOR high X.0 1
```

defines a data event named "high", which is true if the input signal on Pin 0 has a high level. Pattern definitions may also refer to words, which were defined with the “name.word” command:

```
SELECTOR ascii Word.data 'A'--'Z'||'a'--'z'
```

If there a several pattern definitions for different signals, the operation between the pattern definitions is a logical AND. The definition

```
SELECTOR write_sio_control Word.adr 0x4 eXt.ds 1 eXt.write 1
```

is only TRUE, if the “X.write” signal and the “X.cs” are high and the address sampled is “0x4”.

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

.df	double false	true on falling edge, suppresses low glitches
.ds	double static	true on high level, suppresses high glitches
.dt	double true	true on rising edge, suppresses high glitches
.fg	false glitch	true when low glitch is detected
.gf	going false	true on falling edge
.gt	going true	true on rising edge
.s	state	true on high level
.tf	true false	true on falling and rising edge
.tg	true glitch	true when high glitch is detected

An Example:

```
SELECTOR write_sio   eXt.write 1   eXt.cssio 1
Trigger.TRACE IF write_sio.gf   ; triggers on falling edge of write_sio
```

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) when the counter has a value between 10 and 20. An event counter which is controlled by the “counter.increment” instruction, counts the false-to-true transitions of the defined condition. Because of this it isn’t necessary to use the “.gt” suffix for **data events** in the condition.

Flags

Flags are free usable flip-flops to store one bit of information. To allocate a **flag** 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 PowerProbe is armed.

Time Counters

To monitor time relations, it is possible to declare **Time events**. The resolution of the timer is 10 ns (> 50 MHz) respectively 20 ns (50 MHz). The declaration

```
TIMECOUNTER after_5ms 5ms
```

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

Synchronous Counters

If an external clock is applied to the PowerProbe, it is possible to synchronize the clock of a counter to this external clock. With such a counter it is possible to count external clock cycles. The declaration

```
EXTERNSYNCCOUNTER clock_cycles 10.
```

allocates one counter named “clock_cycles”, which is true after 10 clock cycles of the external clock.

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 instructions which can be used in 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. enable 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 condition the action is to be taken. For example, the command

```
Sample.Enable IF strobe
```

records only, while the data event “strobe” 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 strobe
end:
     TRIGGER
```

will change to level "end" as soon as the data event “strobe” becomes true. When the level “end” is active a break is triggered.

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	.PATTERN .PODBUS .Pulse .TRACE .TRCNT	Start Pattern generator Release trigger bus line A (same as **BUS.A**) Start Pulse generator Start trigger delay for breaking PowerProbe (same as **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. from the pattern generator or an internal state of the PowerProbe. 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 eXt.0
```

will make a selective trace as long as the input line eXt.0 is high.

Counter Events

The counter counts up when the specified condition *becomes* 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 a FIFO. It ignores always the first 1000 bytes after the last write to the FIFO:

```
SELECTOR write_fifo eXt.0 1          ; declare Selector for write signal
SELECTOR reset_fifo eXt.3 1          ; declare Selector for reset signal

EVENTCOUNTER delay 1000.

Counter.Increment delay IF write_fifo   ; .gt isn't needed
Counter.Restart         IF reset_fifo
Sample.Enable           IF delay        ; enable sampling after
                                        ; 1000 writes
```

Data Selectors

All not reserved names are allowed as data selector names.

```
SELECTOR low     X.0   0
SELECTOR high    W.adr 0x55
SELECTOR active  eXt.0 1    eXt.1     0 eXt.2   1
SELECTOR select  X.cs  0    X.astrobe 1 X.write 0
```

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 only sample data between the reset and the first write to a fifo device.

```
FLAGS fifo_empty

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

Time Events

The resolution is 10 ns (> 50 MHz) or 20 ns (= 50 MHz). A time event is true, when the time counter reaches the declared value. A time range needs two counters. The following program stops sampling 50 µs after input pin 0 becomes high.

```
TIMECOUNTER delay 50us
SELECTOR    dsel0 X.0 1

start:
     GOTO next      IF dsel0.gt
next:
     Counter.Increment delay
     BREAK.TRACE    IF delay
```

Other Events

The following predefined input events are also available:

Event	Description	
BUSA	Trigger bus A	True when a Podbus trigger signal is detected
FALSE	Always false	
SYNC	SyncClock	True at the start of an external synchronous clock cycle
TRUE	Always true	

Conditions

Conditions are logical 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. The following trigger program has two such conflicts:

```
START: Counter.ON count1, Counter.OFF count1 IF 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; the previous “Counter.ON” instruction is overwritten.

Instructions are executed top to down

In the example above this means that the "GOTO Count_Level" with the condition, is overwritten by the second "GOTO Error_Level" when 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 eXt.fifo_write
GOTO Count_Level
```

then the first statement is completely overwritten.

Global statements have a low priority

Global statements are used, as if 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 can be 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.

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 PowerProbe configuration 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 PowerProbe stops recording after the specified trigger delay. In the example the PowerProbe will change to level "infunc" after "fifo_reset" and stop the PowerProbe after "fifo_write".

```
start:
     CONTinue IF fifo_reset

infunc:
     CONTinue IF 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" event is true, 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 fifo_reset

init:
     GOTO start IF fifo_write
     Sample
```

The **TRIGGER.TRACE** respectively **BREAK.TRACE** instruction causes the PowerProbe to break. Breaking the PowerProbe means stopping recording and deactivating the trigger unit.

```
start:
     BREAK.TRACE IF fifo_write
```

The difference between both instructions is that **TRIGGER** will stop the recording after the defined trigger delay and **BREAK** will stop the recording immediately.

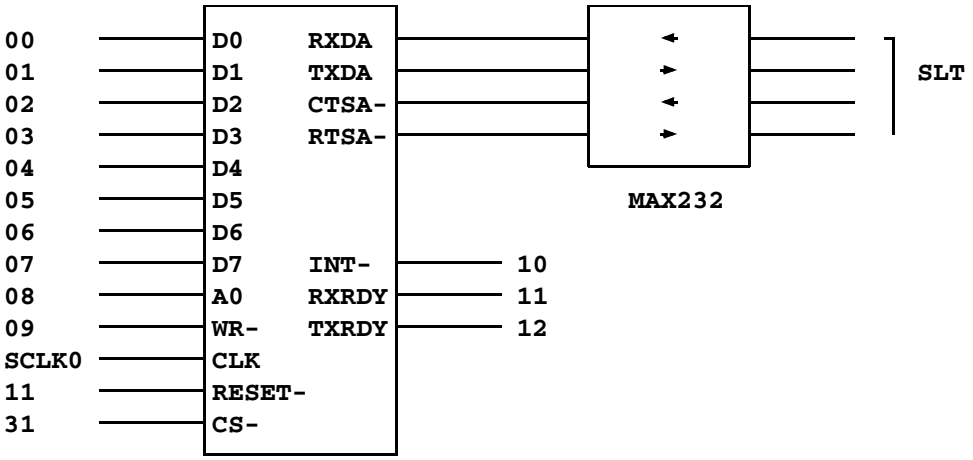
When implementing multiple level change instructions in one trigger level, the instruction order must be observed. Later instructions overwrite conflicting instructions which appear earlier in the trigger program. The following example shows this relation:

```
; Declarations
SELECTOR fifo_reset x.reset 1
SELECTOR fifo_write x.cs 0 x.write 0 w.adr 0x4
SELECTOR dma        x.dma 0
SELECTOR nmi        x.nmi 0
…

; global statements

Sample
…

; local statements

level0:
     CONTinue    IF fifo_reset

level1:
     GOTO level3 IF nmi
     GOTO level3 IF dma.gf
     CONTinue    IF dma&&nmi

level2: …

level3: …
     BREAK.TRACE
```

When the PowerProbe is in level "level1" and assuming that during an active "dma", "nmi" also gets active, the program branches to "level2" and not to “level 3”. When “dma” and “nmi” are both active, the “CONTINUE” statement overwrites the first “GOTO” statement in “level1”.

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.Word BUS_DATA eXt.0 eXt.1 eXt.2 eXt.3 eXt.4 eXt.5 eXt.6 eXt.7
NAME.Set eXt.8  eXt.BUS_A0
NAME.Set eXt.9  eXt.BUS_WR -
Probe.SyncClock sclk0 Rising
NAME.Set eXt.11 eXt.RESET -
NAME.Set eXt.12 eXt.INT -
NAME.Set eXt.13 eXt.RXRDY
NAME.Set eXt.14 eXt.TXRDY
NAME.Set eXt.31 eXt.CS -
```

Selective Recording

Selective recording is done with the instructions **Sample.Enable**.

```
Sample.Enable IF TRUE
```

or

```
s.e
```

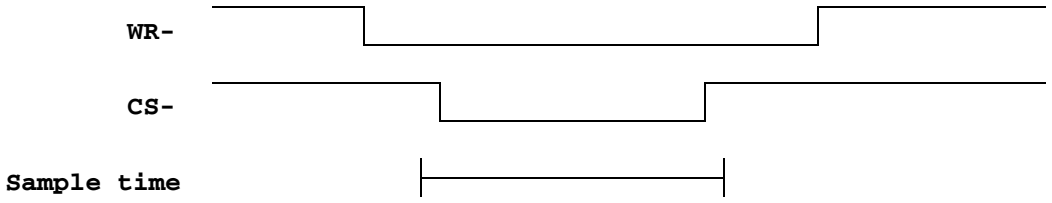
All input data is sampled.

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

```
Sample.Enable IF !eXt.BUS_WR&&!eXt.CS
```

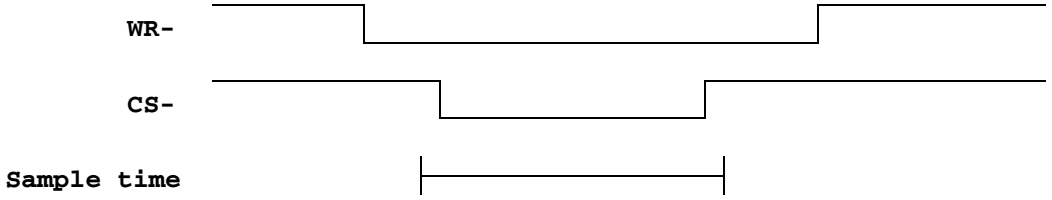
masks:

```
SELECTOR SIO_WRITE eXt.BUS_WR 0 eXt.CS 0

Sample.Enable IF SIO_WRITE
```



In synchronous mode the data will be sampled on the clock SCLK0 only. The pins 0-7 must be switched to synchronous mode:

```
NAME.Set eXt.0 d0 + sync
NAME.Set eXt.1 d1 + sync
…

SELECTOR SIO_WRITE eXt.9 0 eXt.31 0

Sample.Enable IF SIO_WRITE
```



The Sample time is a elongated a little bit, so that it is possible to see the signal change, which was responsible for the activation and deactivation of the “Sample.Enable” instruction.

Stopping the PowerProbe

There are some reasons which can stop the PowerProbe recording:

Trace Full

Stopping the recording when the PowerProbe is full can be achieved by selecting **Stack** operation mode in the PowerProbe configuration window (command **Probe.Mode Stack**).

```
Probe.Mode Stack
```

The PowerProbe stops recording independent of the current logical level.

Trigger or BREAK

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** 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 PowerProbe. These lines can be used to trigger external timing PowerProbes 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 PowerProbe unit every time the interrupt line becomes active low:

```
SELECTOR int_active eXt.INT 0
OUT.A IF int_active.gt
```

Using the Internal Trigger Bus

The **instruction BUS** can trigger other systems of the TRACE32 system. The inter-trigger bus of the system can be used to trigger the pattern or pulse generator by the timing PowerProbe. 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**).

```
SELECTOR SIO_WRITE eXt.bus_wr 0

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

The timing PowerProbe can also be triggered by other PowerProbes. The event **BUSA** can be used for this purpose. When the line BUSA is released from the state PowerProbe, the timing PowerProbe can be controlled by the state PowerProbe. In the following example the timing PowerProbe records for 1 ms after the trigger event from the state PowerProbe.

```
; 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 is **TIMECOUNTER** for the declaration and the instruction **COUNTER** controls the counter. Time and **event counters** need a declaration to give the counters a name and an initial value. Counters are always assigned for the whole trigger program, only the control of the counters is level specific.

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

```
SELECTOR    SIO_WRITE eXt.BUS_WR 0
TIMECOUNTER timeout   10.us

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

Trigger.TRACE              IF timeout
```

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

```
; declaration
SELECTOR     SIO_WRITE    X.BUS_WR 0
EVENTCOUNTER write_cycle

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

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

```
; declaration
SELECTOR SIO_WRITE X.BUS_WR 0
EVENTCOUNTER write_cycle 1000.

; counter operation
Counter.Increment write_cycle IF 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 PowerProbe 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 PowerProbe state window:

```
SELECTOR WRITE_TO_CONTROL Word.BUS_DATA 0x12 eXt.CS 0 eXt.BUS_WR 0

; declaration
FLAGS TX_ENABLE                 ; declaration of 1 flag

FLAG.TRUE  TX_ENABLE  IF   WRITE_TO_CONTROL
FLAG.FALSE TX_ENABLE  IF  !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** and **Break.TRACE** will disable the PowerProbe and the trigger unit.

```
; declaration
…

; global instructions
…
```

```
; local instructions
level0: …
        CONTinue    IF SIO_WRITE      ; change sequential to the next
        …                             ; logical level


level1: …
        CONTinue    IF CNT_Limit
                                      ; change to the next logical level
                                      ; if the counter event CNT_Limit is
                                      ; true

        GOTO level0 IF BB             ; otherwise jump return to the
        …                             ; level level0 if the data event BB
                                      ; is true


level2: BREAK.TRACE IF DELAY_CNT      ; stop recording if the counter
                                      ; event DELAY_CNT is true

START:  GOTO level0 IF INT            ; this is the start level after
                                      ; PowerProbe init jump to level0
                                      ; when the data event INT is true
```

SELECTOR — Data selectors

Format:	**SELECTOR** *<name>* *<pin> <value>|<w.name> <value>* …
<pin>:	x.0 | x.1 | x.2 | x.3 | … | *<name declared with name.set>*
<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 x.0 (LSB) to x.7 (MSB).

```
NAME.WORD    adr  x.0 x.1 x.2 x.3 x.4 x.5 x.6 x.7
SELECTOR Port_1  w.adr 0x55
```

A data selector named Port_2 with the value 0x55 on the pins x.0 (LSB) to x.7(MSB) and with the bit mask 1010xx1x on pins x.8 (LSB) to x.15 (MSB):

```
NAME.WORD    addr  x.0 x.1 x.2  x.3  x.4  x.5  x.6  x.7
NAME.WORD    data  x.8 x.9 x.10 x.11 x.12 x.13 x.14 x.15
SELECTOR Port_2  w.addr 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 x.0 write
SELECTOR write_fifo x.write 1
```

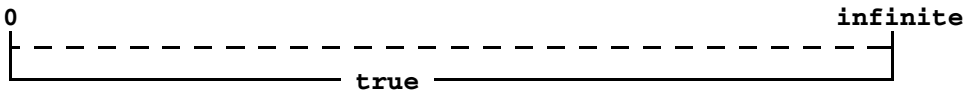
Format: **EVENTCOUNTER** <*name*> [<*event*>]

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 PowerProbe. 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 value.

The current value of the counters are visible in real-time in the PowerProbe configuration window.

Endless Counter

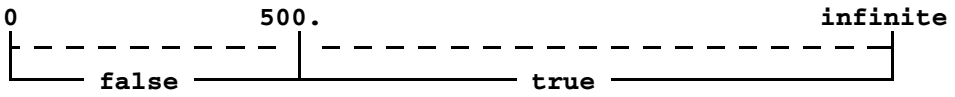
```
EVENTCOUNTER Evcntr_2 0
Counter.increment Evcntr_2 IF true
Sample.enable              IF Evcntr_2
```



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 PowerProbe begins sampling immediately.

Event TRUE after n Clocks

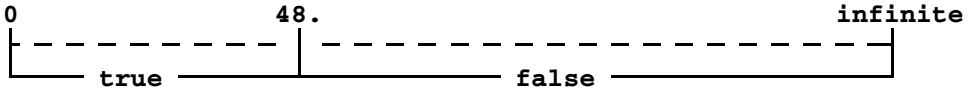
```
EVENTCOUNTER CYCLE_CNT 500.
Counter.increment CYCLE_CNT  IF  SIO_WRITE
Sample.enable                IF  CYCLE_CNT
```



Declaration of an event counter called "CYCLE_CNT". The counter counts rising edges of the SIO_WRITE data event. The PowerProbe begins sampling after a delay of 500 rising edges of SIO_WRITE.

Event TRUE till n Clocks

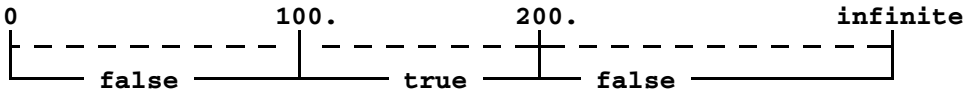
```
EVENTCOUNTER NR_cnt 0x0--0x30
Counter.Increment NR_cnt IF SIO_WRITE
Sample.Enable            IF NR_cnt
```



Declaration of an event counter called "NR_cnt", event argument is 0x30. The counter counts rising edges of the SIO_WRITE data event. The PowerProbe begins sampling immediately and stops recording after 48 rising edges of SIO_WRITE.

Event Windows

```
EVENTCOUNTER EV_Range 100.--200.
Counter.Increment EV_range IF SIO_WRITE
Sample.Enable              IF EV_range
```



Declaration of an event counter called "EV_range" with an event range from 100 to 200. The counter counts rising edges of the SIO_WRITE data event. The PowerProbe begins sampling after 100 rising edges and stops recording 100 rising edges 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 PowerProbe. 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 x.0 d0 sync
name.set x.1 d1 sync
name.word data x.0 x.1

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 x.0 and x.1 pins are high.

If events used in a condition for an instruction for a synchronous counter, are not synchronized to the external clock, it must be ensured, that the events have a setup and hold time of at least 5ns. Otherwise correct operation can’t be guaranteed.

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 **Probe.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  x.0 1
SELECTOR write_fifo  x.1 1

FLAG.TRUE  init_state IF eXt.reset_fifo
FLAG.FALSE init_state IF eXt.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 PowerProbe. 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 PowerProbe configuration window.

Time values can be entered in the following units:

Nanoseconds (ns)

Microseconds (us)

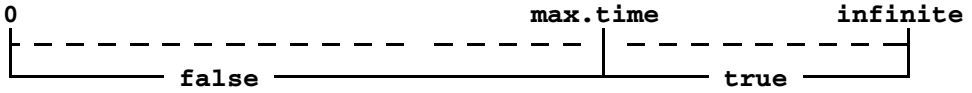
Milliseconds (ms)

Seconds (s)

Kiloseconds (ks)

Timer running till Overflow

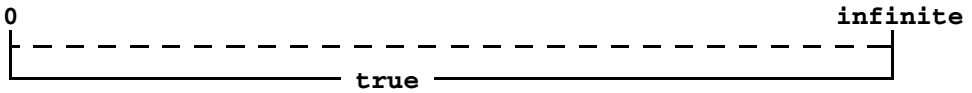
```
TIMECOUNTER Timer_1
Counter.increment Timer_1 IF true
Sample.enable             IF Timer_1
```



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 PowerProbe starts sampling input data.

Always running Timer

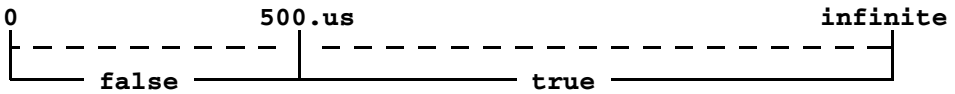
```
TIMECOUNTER Timer_2 0.ms
Counter.Increment Timer_2 IF TRUE
Sample.Enable             IF Timer_2
```



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

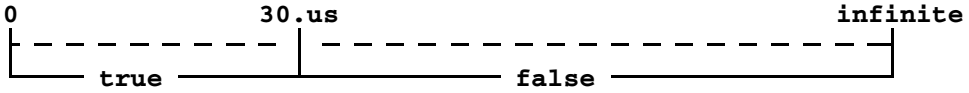
```
TIMECOUNTER Timer_A 500.us
Counter.Increment Timer_A IF TRUE
Sample.Enable             IF Timer_A
```



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

Timer TRUE till time

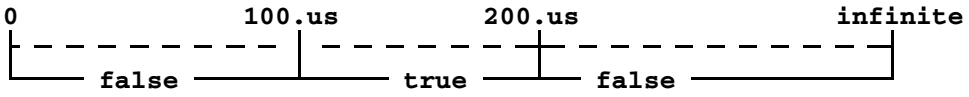
```
TIMECOUNTER Timer_B 0.us--30.us
Counter.Increment Timer_B IF TRUE
Sample.Enable             IF Timer_B
```



Declaration of a time counter called "Timer_B". The counter is always enabled. The PowerProbe 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 from 100 to 200 microseconds. The counter is always enabled and counts every time. The PowerProbe begins sampling after 100us and stops recording 100us later. Two physical counters are used by the trigger unit.

BREAK — PowerProbe stop

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

The PowerProbe breaks and stops recording immediately, independently from the before defined trigger delay. The value from a before used **Probe.TDelay** command will be ignored. The PowerProbe can be read out when in break state, similar to the OFF state. The break level is reset by the command **Probe.Init**. See also the command **Trigger.TRACE**.

```
…
BREAK.TRACE IF fifo_reset
…
```

The PowerProbe breaks, whenever the data event "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 PowerProbe is breaked.

In the example the PowerProbe will change to level "init" if the data event "fifo_reset" is true.

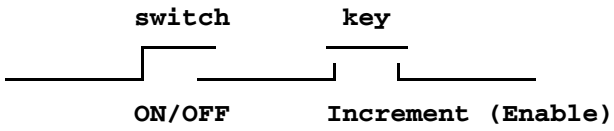
```
SELECTOR    fifo_reset   x.0 0

start: CONTinue IF fifo_reset

init:
```

Format:	**Counter**[.<*mode*>] <*counter_name*> [**IF** <*condition*>]
<*mode*>:	**Enable** (old-fashioned) **Increment** **OFF** **ON** **Restart**

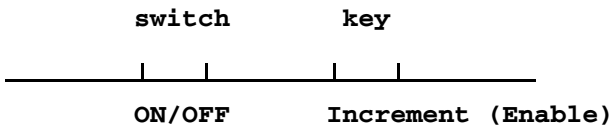
Control the trigger units counters. The instructions Counter.ON and Counter.Increment are programmed automatically, if they are not used in the trigger program. The counters have to be declared according to their function (see also declaration EVENTCOUNTER, EXTERNSYNCCOUNTER or TIMECOUNTER and chapters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

Enable (old-fashioned), **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.

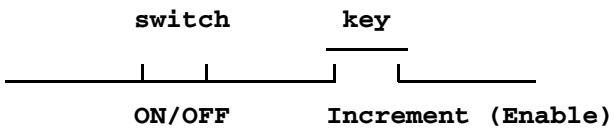
The instructions **ON**, **OFF** and **Increment (Enable)** can be seen as a controlled switch and a key in series. If the switch is closed (**Counter.ON**) it remains closed till it is opened by **Counter.OFF**. The key is closed for the cycles which meet the specified condition. An event counter will only advance once when the key is closed. That means an event counter counts how often the key was closed. A Time counter will count as long as the key is closed. That means a time counter counts how long the key was closed.



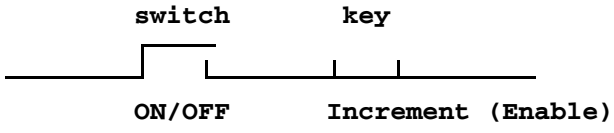
If neither ON/OFF nor Increment (Enable) are used in the complete trigger program, the switch and the key are closed, that means the counter counts time or exactly one event (key is closed when the recording starts) depending on its declaration.



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



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



Counter CYCLE_CNT counts exactly one event:

```
; declaration
EVENTCOUNTER CYCLE_CNT

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

Counter "count_reset" is incremented by 1 every time, the input pin 0 is pulled to low.

```
; declaration
SELECTOR     reset_fifo eXt.0 0
EVENTCOUNTER count_reset

; global or local instruction
Counter.Increment count_reset IF reset_fifo
```

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

```
; declaration
SELECTOR    reset_fifo eXt.0 0
TIMECOUNTER reset_puls

Counter.Increment reset_puls IF fifo_reset

; Wait for start of fifo_reset
level0: GOTO level1, Counter.ON reset_puls   IF fifo_reset.gt

; Wait for end of fifo_reset
level1: BREAK.TRACE, Counter.OFF reset_puls  IF !fifo_reset
```

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 PowerProbe breaks.

```
; declaration
SELECTOR     upper_ascii Word.databus 'A'--'Z'
EVENTCOUNTER ascii_count 100.

; global or local instruction
Counter.Increment ascii_count.s
BREAK.TRACE                    IF ascii_count
```

If the pulse width of the "cs_fifo" signal is more than 500 ns, the timing PowerProbe will break.

```
; declaration
SELECTOR    cs_fifo  Word.cs_fifo_register 0Yxxxxx00x
TIMECOUNTER time_out 500ns

; global instruction
Sample.Enable
Counter.Restart   time_out IF cs_fifo.gt
Counter.Increment time_out IF cs_fifo
BREAK.TRACE                IF time_out
```

The first write_fifo event must be within 100 µs after the fifo_reset state. Otherwise the PowerProbe is breaked. 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  eXt.0 0
SELECTOR    write_fifo  eXt.1 0

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
          BREAK.TRACE                   IF  write_fifo&&!first_write
```

Format:	**Flag.**<*mode*> <*name*> [**IF** <*condition*>]
<*mode*>:	**FALSE** **OFF** **ON** **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 PowerProbe configuration window.

FALSE, OFF	Resets the flag.
TRUE, ON	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 data_event
```

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 PowerProbe there are 4 levels available.

```
Start:
     Counter.Restart int_count
     GOTO LL8

      …

LL5:
     Sample.Enable               IF dma
     GOTO l18                    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. PowerProbes or oscilloscopes) or to stimulate the target hardware. These signals are accessible via socket connectors at the PowerProbe chassis.

A, B, C, D	Activates the universal output TOUT0..TOUT3 at the top of the PowerProbe chassis.

```
Out.A IF time_out.gt
```

Release trigger line TOUT0 for 10 ns when event time_out becomes true.

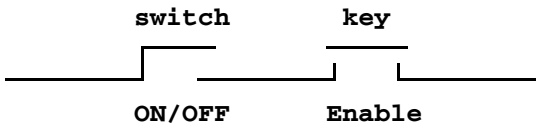
Sample — Recording control

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 trigger event and the first and last cycle before the user program stopped.

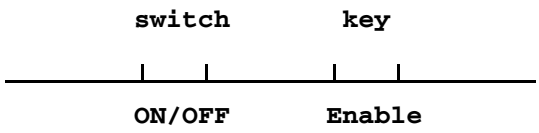
Enable	Releases trace memory for recording when the specified condition is true.
OFF	Disables the Flip-flop for sampling.
ON	Enables the Flip-flop for sampling.

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 until 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.

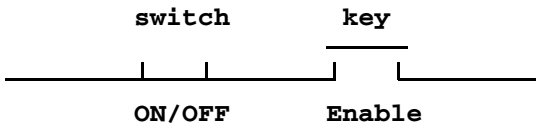


Sampling is only enabled if the switch and the key are closed.

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").

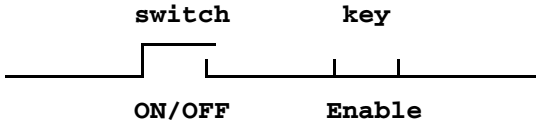


If only Enable is used in the trigger program, the switch ON/OFF is closed automatically, that means sampling is controlled only via “Sample.Enable” (Implicit global "Sample.ON IF true").



If only “Sample.OFF” is used in the trigger program, the key Enable is closed automatically, and the ON/OFF switch is closed initially. The sampling will stop as soon as the “Sample.OFF” instruction is executed.

If only “Sample.ON” and “Sample.OFF” are used, the key Enable is closed, and the ON/OFF switch is open initially. Sampling will only be controlled by the ON/OFF switch.



The following statements are equal and will sample all cycles:

```
Sample.Enable IF TRUE
Sample.Enable
S.E
s
```

Sample only if line eXt.0 is high:

```
SELECTOR x0high eXt.0 1

Sample.Enable IF x0high
```

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

```
; declaration area
SELECTOR int1_low eXt.4 0
…

; local area
Level0: Sample.Enable IF int1_low
        CONTinue      IF int1_low
Level1: Sample.Enable
     …
```

Trigger — Trigger control

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

Trigger other systems of the PowerProbe.

Pattern	Releases a trigger signal to the **pattern generator**.
PODBUS	Releases a signal on the inter-trigger bus (**BUS.A**).
PULSE	Releases a pulse of the pulse generator (**PULSe**).
TRACE, A	Starts the trigger delay counter defined by the command **Probe.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 PowerProbe breaks, it stops recording after the, with the command **Probe.TDelay** defined trigger delay. The PowerProbe can be read out when in break state, similar to the OFF state. The break level is reset by the command **Probe.Init**. See also the command **BREAK.TRACE**.

```
…
Trigger.TRACE IF fifo_reset
…
```

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

PowerProbe 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 PowerProbe programming language starts with the nonterminal symbol *ppta_prog*.

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


decls:              [(eve_dec | ext_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}
                                     [{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_name*

dataname_prefix1: **EXT** | **S** | **SOC** | **X**

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

dataname_postfix: **0..1023** | from user with **NAME.Group** or **NAME.Set** or **NAME.Word** defined names

mode: **DF** | **DS** | **DT** | **FG** | **GF** | **GT** | **S** | **TF** | **TG**

inline_name: **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