Complex Trigger Unit for Nexus MPC5xxx

MANUAL

Complex Trigger Unit for Nexus MPC5xxx

TRACE32 Online Help

TRACE32 Directory

TRACE32 Index

TRACE32 Documents ....

ICD In-Circuit Debugger ....

Processor Architecture Manuals ....

Qorivva MPC5xxx/SPC5xx ....

Application Note for Nexus MPC5xxx ....

Complex Trigger Unit for Nexus MPC5xxx **1**

History **4**

Introduction **5**

Program Structure **7**

Conditions **8**

Declaration Reference **10**

ADDRESS — Address selectors — 10

EVENTCOUNTER — Event counter — 11

Event TRUE after n Clocks — 12

Event TRUE till n Clocks — 12

Event Windows — 13

FLAGS — Flags — 13

HWME — Hardware message events — 14

OTME — Ownership trace message events — 14

TIMECOUNTER — Time counter — 15

Timer Running till Overflow — 16

Timer TRUE after Time — 16

Timer TRUE till Time — 16

Time Windows — 17

TRIG — External triggers — 17

Instruction Reference **19**

BREAK — Analyzer stop — 19

Bus — Bus trigger — 19

CONTinue — Sequential level switching — 20

Counter — Counter control — 20

Flag — Flag control — 23

GOTO — Level switching — 24

Mark Recording markers 24
Out Output control 25
Sample Recording control 26
Trigger Trigger control 28

CTU Programming Examples 30
Data Trace Message based events 30
Example: Trace trigger on data value 31
Example: Program break on data value 31
Watchpoint hit message based events 32
Example: Runtime measurement with markers 34
Example: Program break based function runtime 37
Using external signals with the CTU 38
Example: Record single message on rising edge of trigger input 39
Example: Program break based on pulse interval of IN input 40

Appendix: Complex Trigger Unit Keyword Reference 41

Version 06-Jun-2024

History

18-Jun-2022 Initial version.

Introduction

This application note describes the features and programming of the Complex Trigger Unit for MPC5XXX processors with a parallel Nexus trace port.

The PowerTrace module contains the Complex Trigger Unit, short CTU. The CTU is a trigger sequencer that provides additional trigger and filter possibilities. The usage of the CTU has no impact on the real-time behavior of the processor.

The Complex Trigger Unit (CTU) is supported with following trace modules:

- PowerTrace Ethernet (LA-7707, LA-7690)
- PowerTrace II (LA-7692, LA-7693, LA-7694)
- PowerTrace PX (LA-3510)

Supported for port widths: MDO8, MDO12 and MDO16.

On-chip traces and Aurora NEXUS trace ports are not supported.

The CTU is programmed by a special trigger language.

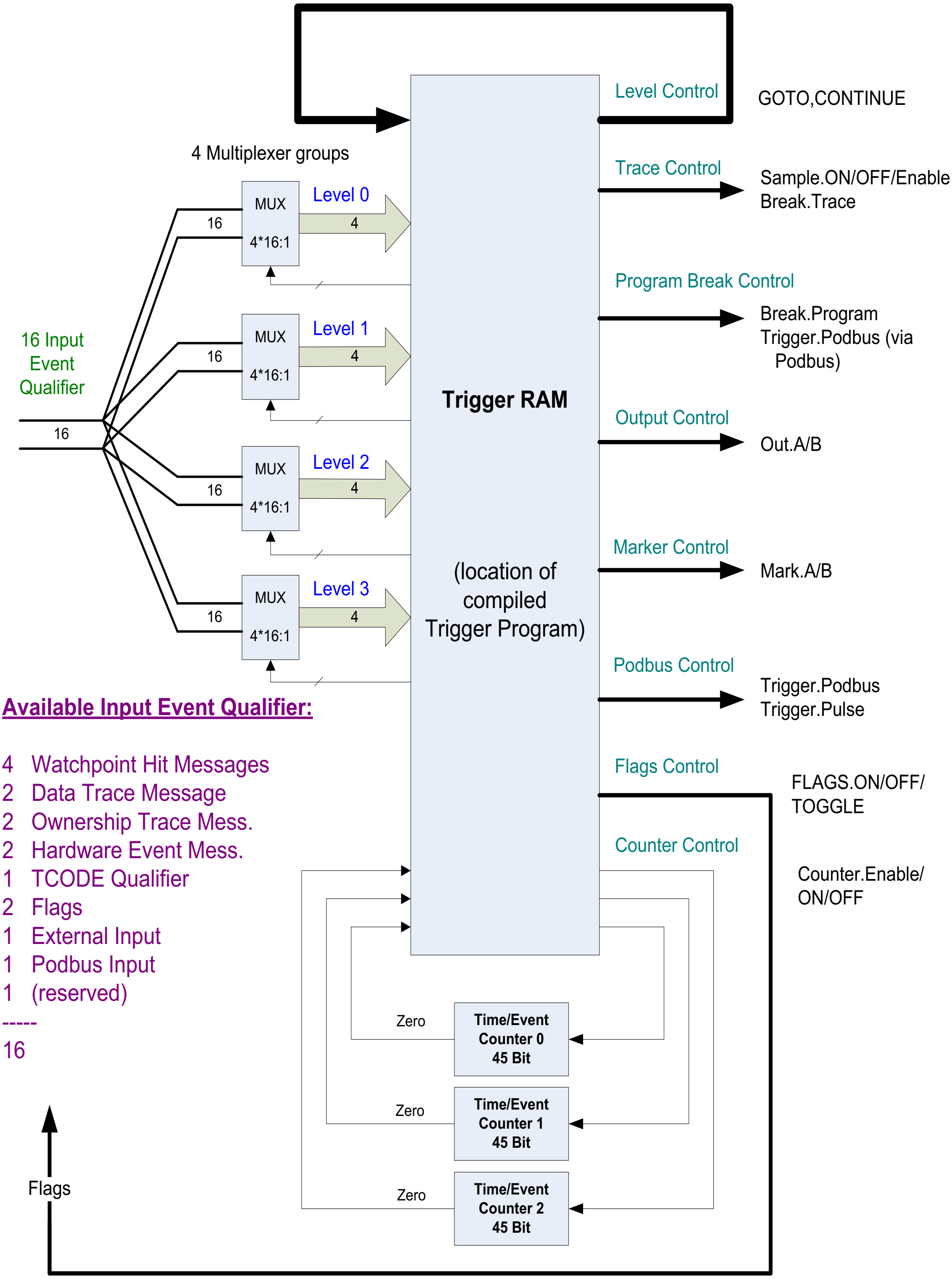
The main input events for the CTU together with a NEXUS adapter are:

- Watchpoint hit messages
- Data trace messages
- Ownership trace messages
- External trigger input pin

Based in this input events, the CTU can provide the following features:

- Trigger output signals
- Halt program execution
- Halt trace recording (trace trigger)
- Trace filtering
- Set marks in trace recording

Complex Trigger Unit (CTU) Diagramm



Program Structure

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

Comments	Comments are allowed anywhere in the trigger program. They begin with a ";" or with “//“.
Declarations	Declarations define input events which need to be declared. Such events are address selectors, data selectors or counters.
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 bus cycle, an access to an address or a predefined data pattern.

Levels	The beginning of a level is defined by the name of the level followed by a colon ":". The end of a level is the beginning 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 a trigger program starts within the first written level or the level with the name "START:".
Global instructions	Global instructions 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 consists of global instructions only.
Local instructions	Local instructions are valid within one trigger level only. All local instructions defined within a level and all global instructions are checked simultaneously.

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, it will be assumed 'TRUE'. The program

```
Sample.enable
```

will produce the same results as

```
Sample.enable IF TRUE
```

Input events can be combined by standard logical operators:

(...)

! or **N:** for NOT

&& or **:A:** for AND

^^ or **:X:** for XOR

|| or **:O:** for OR

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

The following two conditions will produce the same results:

```
(BetaBreak&&User)||!(UserData&&!AlphaBreak)
 BetaBreak&&User||!UserData||AlphaBreak
```

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  AlphaBreak
        GOTO Count_Level
        GOTO Error_Level                       IF  Write&&BetaBreak
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 when an AlphaBreak occurs.

Instructions are executed top to down

In the example above the instruction "GOTO Count_Level", 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 when the condition "Write&&BetaBreak" 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 "Write&&BetaBreak".

If the order of the "GOTO" statements is changed:

```
GOTO Error_Level IF Write&&BetaBreak
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.

ADDRESS Address selectors

Format:	**ADDRESS** <*breakpoint*> <*address*> …
<*breakpoint*>:	**AlphaBreak** **BetaBreak** **CharlyBreak** **DeltaBreak** **EchoBreak**

The names of the **address selectors** are predefined and assigned to the breakpoints. Individual names cannot be assigned.

These functions must be disabled, before using the breakpoints as address selectors. The breakpoints **AlphaBreak** and **BetaBreak** have no fixed functions, they are the first choice for analyzer address selectors. The **CharlyBreak** selector can be used as a background spot in multitasking environments. **DeltaBreak** and **EchoBreak** have the same function as **AlphaBreak** and **BetaBreak**.

Address selectors can be used with previous declaration. In this case all breakpoints of that type are defined in the analyzer program. Without declaration it is possible to use breakpoints, which were set by other commands. This gives more flexibility in the assignment of breakpoints. Useful commands to set breakpoints for the analyzer are:

Break.Set	Set breakpoints
Break.SetFunc	Set breakpoints on functions entries
Break.SetLine	Set breakpoints on HLL lines
Var.Break.Set	Set breakpoints on HLL structures
sYmbol.ForEach	Set breakpoints on a symbol pattern

An address selector declaration in the analyzer programming can define multiple addresses or address ranges. One declaration line can define multiple addresses by using multiple segment ranges:

```
ADDRESS AlphaBreak main||sieve||inchr||outchr
```

Multiple declaration lines can be used to define a more complex breakpoint definition:

```
ADDRESS AlphaBreak main--sieve
ADDRESS AlphaBreak SD:0x0f2--0x0f7
ADDRESS AlphaBreak SP:0x10000..0x0ffffff
```

The following declaration sets the selector at two consecutive bytes:

```
ADDRESS BetaBreak \\MOTSTEU\MOTOR1\Speed\value1++1
```

The size of HLL structures can be accessed by special functions. The declaration

```
ADDRESS AlphaBreak V.RANGE(function3)
```

marks the whole code of 'function3' with breakpoints. Using HLL expressions for the address is also possible:

```
ADDRESS AlphaBreak V.RANGE("stra[2].count") V.RANGE("stra[1]")
```

The following example makes a selective trace on all accesses to a variable:

```
; Declaration
ADDRESS AlphaBreak V.RANGE(flags)
; Global instruction
Sample.enable IF AlphaBreak
```

EVENTCOUNTER — Event counter

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. 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 equal to the maximum.

Each counter is **released selectively** and the state of the counters can be used as an input event. Event ranges will occupy two universal counters.

This event delay counter will be re-loaded automatically during entering the delay level. The counter could be released only general (Counter.Increment DELAY without any condition) in the delay level and could be used as an input event for other commands like Sample.Enable, BREAK …

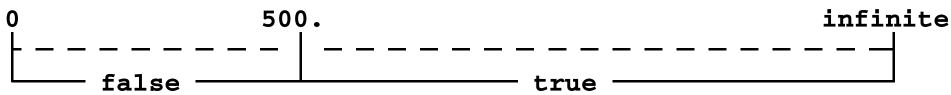
Analyzer Type	Counters	Max. Value
ICD	3	3.5e13

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

Event TRUE after n Clocks

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

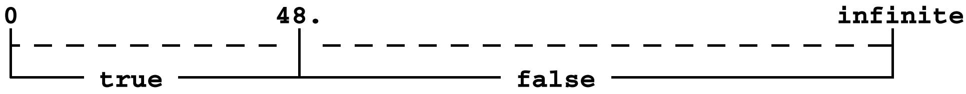
```
EVENTCOUNTER CYCLE_CNT 500.
Counter.Increment CYCLE_CNT IF TRUE
Sample.enable               IF CYCLE_CNT
```



Event TRUE till n Clocks

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

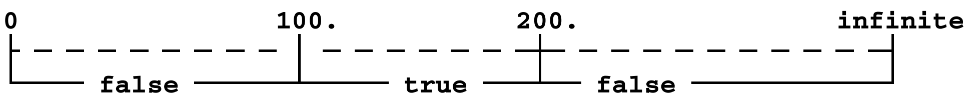
```
EVENTCOUNTER NR_cnt 0--48.
Counter.Increment NR_cnt
Sample.enable  IF NR_cnt
```



Event Windows

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

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



FLAGS Flags

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.

The complex trigger unit of **ICD** analyzer has 2 flags.

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

The following program samples only, when the variable 'var2' has the value zero:

```
FLAGS      VAR2_IS_ZERO
ADDRESS    AlphaBreak    var2
DATA.W     ZERO          0x0

Flag.TRUE  VAR2_IS_ZERO IF AlphaBreak&&WRITE&&ZERO
Flag.FALSE VAR2_IS_ZERO IF AlphaBreak&&WRITE&&!ZERO

Sample.enable           IF VAR2_IS_ZERO
```

HWME — Hardware message events

Format:	**HWME** <*name*> <*mask*>

Any name can be assigned to the hardware message event (HWME), as long as it doesn't conflict with the reserved names of other events. 2 physical events are available and are selected automatically by the system, depending on their usage as input events in conditions. HWME could be used to react on signals set from the customer specific chip units in the hardware event register.

The following program samples only, when a UART access sets bit12 in the hardware event register:

```
HWME UART1 0x1000

Sample.enable IF UART1
```

OTME — Ownership trace message events

Format:	**OTME** <*name*> <*value*> [**/** <*unitname*>]
<*unitname*>:	**DMA** | **ETPU1** | **ETPU2** | **ETPCDC** | **PPCCORE** | **UNIT0** | … | **UNIT15**

Any name can be assigned to the ownership trace message event (OTME), as long as it doesn't conflict with the reserved names of other events. 2 physical events are available and are selected automatically by the system, depending on their usage as input events in conditions. OTME could be used to react on ownership trace messages with a certain value.

The following program samples only, when a certain task is running. The hardware must be configured in the way that OTMEs contain the actual task number.

```
OTME task3 0x1234           ; OTME with PID 0x1234 for task3

Sample.ON  IF task3         ; start recording cycles if a OTM with
Sample.OFF IF TCODE_OTM     ; PID 0x1234 occurs and stop recording
                            ; after the next OTM
```

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. If a time counter reaches its declared value it will stop automatically. The timers can be **re-loaded** in real-time. However, program dependent dead times can result. The default value is equal to the maximum time. Each timer is **released selectively** and the state of the counters can be used as an input event.

Analyzer Type	Counters	Max. Time	Resolution
ICD	3	8 days	20 ns

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

Time values can be entered in the following units:

Nanoseconds (ns)

Microseconds (us)

Milliseconds (ms)

Seconds (s)

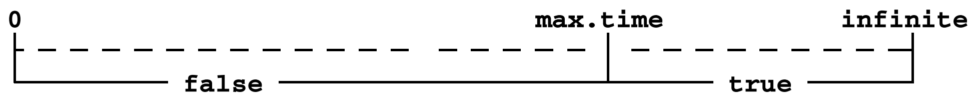
Kiloseconds (ks)

Timer Running till Overflow

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 analyzer begins sampling.

```
TIMECOUNTER Timer_1

Counter.Increment Timer_1
Sample.enable  IF Timer_1
```

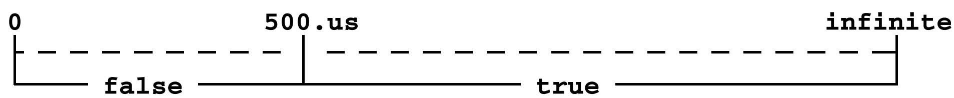


Timer TRUE after Time

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

```
TIMECOUNTER Timer_A 500us

Counter.Increment Timer_A
Sample.enable  IF Timer_A
```

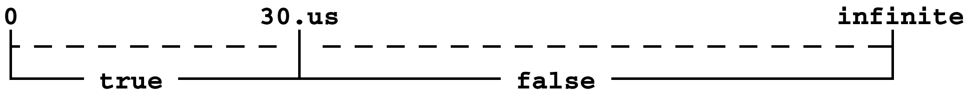


Timer TRUE till Time

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

```
TIMECOUNTER Timer_B 0.us--30.us

Counter.Increment Timer_B
Sample.enable  IF Timer_B
```

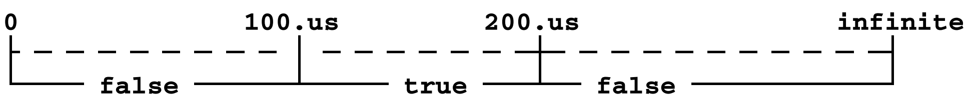


Time Windows

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 analyzer begins sampling after 100 us and stops recording 100 us later. Two physical counters are used by the trigger unit.

```
TIMECOUNTER Timer_C 100.us--200.us

Counter.Increment Timer_C
Sample.enable  IF Timer_C
```



TRIG — External triggers

Format:	**TRIG.**<*channel*> <*name*> <*data*> …
<*channel*>:	**A** **B**

External triggers allow the analyzer to react on external signals.

A trigger event is true, when the declared value matches the levels at the input probe. Bit masks and hex masks are allowed to ignore input pins. The name for the trigger selector can be chosen freely.

Declaration of 3 trigger events called BANK0 with value 00, BANK1 with value 01 and BANK2 with value 02 on trigger input A.

```
TRIG.A BANK0 00
TRIG.A BANK1 01
TRIG.A BANK2 02
```

The trigger selector named CS_PERF becomes true if value of the bit mask 01010xx appears on trigger input B.

```
TRIG.B CS_PERF 0y01010xx
```

The **analyzer breaks** when a write access and a low signal on line 0 of trigger input A is performed.

```
TRIG.A EXT_CS 0y0xxxxxxx0    ;declaration

BREAK IF EXT_CS&&Write       ;global instruction
```

Several trigger event declarations.

```
; declarations

TRIG.A T_sel0 0x55                     ; trigger selector on the input
                                       ; TRIGGER A with the value 0x55

TRIG.B T_sel1 0x0x5                    ; trigger selector on the input
                                       ; TRIGGER B with a hex mask, all
                                       ; values with ;the low nibble 5

TRIG.B SEL_B  0y0xxxxxxx0              ; trigger selector with a bit mask
                                       ; bit number 0 low

TRIG.A T_RANGE 0x10--0x20              ; trigger selector range, values
                                       ; between 0x10 to 0x20

TRIG.A TEV_VAL 33.||55.||0xfe          ; trigger selector with 3 different
                                       ; values

TRIG.B TEV_NEX 77. 88. 99.             ; identical as above without
                                       ; logical OR

;global or local instruction

Counter.Increment CNT_1 IF SEL_B       ; the counter counts if the trigger
                                       ; input TRIGGER A bit 0 is low
```

BREAK — Analyzer stop

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

<*mode*>: **PROGRAM**
TRACE

Mode description.

PROGRAM	This event is usually used to stop the user program (asynchronous breakpoint).
TRACE	Stops only the recording of the analyzer.

When the analyzer BREAKs, it stops recording and the trigger unit is switched off. The analyzer can be read out while in break state, similar to the OFF state.

The analyzer stops, whenever the address "Subr_end" appears on the address bus.

```
ADDRESS  BetaBreak  Subr_end
...
BREAK IF BetaBreak

...
```

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	Activates bus trigger lines A.

CONTinue

Sequential level switching

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 analyzer is stopped.

In the example the analyzer will change to level "infunc" after an access to an Alpha breakpoint and stop the analyzer after the next access to an Beta breakpoint:

```
start:  CONTinue IF AlphaBreak

infunc: CONTinue IF BetaBreak
        Sample.enable
```

Counter

Counter control

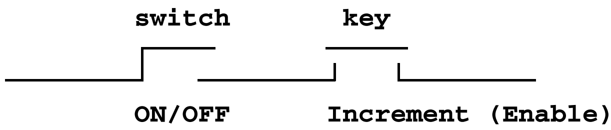
Format: **Counter**[**.**<*mode*>] <*counter_name*> [**IF** <*condition*>]

<*mode*>: **Enable (obsolete)**
Increment
OFF
ON
Restart

This instruction controls the trigger units counters. The instructions **Counter.ON** and **Counter.Increment** will be programmed automatically, if they are not used in the trigger program. The counters have to be declared according to their functions (see also declaration **EVENTCOUNTER**, and **TIMECOUNTER**).

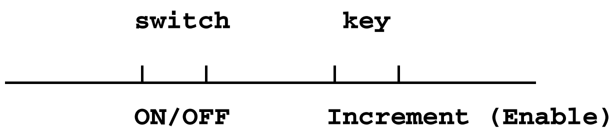
Enable (obsolete)	Releases counters when the specified condition is true.
Increment	Releases counters when the specified condition is true.
OFF	Switches the enable Flip-flop OFF.
ON	Switches the enable Flip-flop ON.
Restart	The counter is reset to zero.

The instructions **ON**, **OFF** and **Increment (Enable)** function as controlled as a switch and a key in series. If the switch is closed (**Counter.ON**) it remains closed until 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 a step.

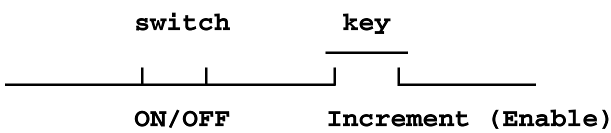


The counter is incremented whenever the switch and the key are closed.

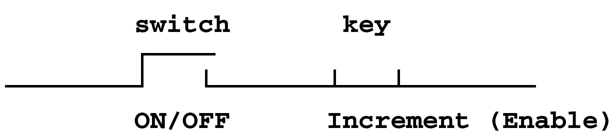
If neither ON/OFF nor Increment (Enable) are used in the complete trigger program, the switch and the key are closed, therefore the counter counts time or events (cycles) depending on its declaration.



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 by ON/OFF only.



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

Counter CYCLE_CNT is counting every CPU cycle.

```
EVENTCOUNTER CYCLE_CNT                    ; declaration

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

Counter CCC0 is incremented by 1 every time, when addr1 has been reached.

```
EVENTCOUNTER CCC0                          ; declarations
ADDRESS      AlphaBreak addr1

Counter.Increment CCC0 IF AlphaBreak       ; global or local instruction
```

Counter EV_LIMIT is counting occurrence of Func1. After 100 counts it stops recording via the **analyzer break command**.

```
EVENTCOUNTER EV_LIMIT   100.               ; declarations
ADDRESS      AlphaBreak Func1

Counter.Increment EV_LIMIT IF AlphaBreak   ; global or local
BREAK                      IF EV_LIMIT     ; instructions
```

The trigger program is sampling all cycles and waiting in level "Level0" to access address "TRAP_S". After this event the level changes to level "Level1" and the counter "T_LIMIT" is released to the count time. After 50 us the sampling is stopped by the **analyzer break command**.

```
TIMECOUNTERR T_LIMIT   50.us                ; declarations
ADDRESS      BetaBreak TRAP_S

Sample.enable                               ; global instruction

Level0: CONTinue             IF BetaBreak   ; local instructions
Level1: Counter.Increment T_LIMIT
        BREAK               IF T_LIMIT
```

The time counter "MESURE_T" starts counting at the entry of Level5. It stops time measurement when reaching the Level7. The counter contains the time between entrance in Level5 and entrance in Level7.

```
TIMECOUNTER MEASURE_T                       ; declarations
ADDRESS     AlphaBreak sp:0x1000
ADDRESS     BetaBreak  sp:0x1025

Sample.enable                               ; global instruction

start:  Counter.OFF MEASURE_T               ; switch off counter
...                                         ; (counter is on when the
                                            ; trigger program starts)

Level5: Counter.ON MEASURE_T                ; Switch on counter
        CONTinue            IF AlphaBreak
Level6: CONTinue            IF BetaBreak
Level7: Counter.OFF MEASURE_T               ; Switch off counter
...
```

Retrigger Timer1 when it has expired.

```
Counter.Restart Timer1 IF Timer1
```

The execution will stop if the interrupt service routine INT_Service needs more than 225 us or less than 200 us.

```
TIMECOUNTER OVERUNDERFLOW 200us--225us          ; declaration
ADDRESS     AlphaBreak    INT_Service_Start
ADDRESS     BetaBreak     INT_Service_End

...                                             ; local instructions
LLL1: Counter.Restart OVERUNDERFLOW
      CONTinue  IF AlphaBreak

LLL2: Counter.Increment OVERUNDERFLOW
      Trigger.A IF BetaBreak
      CONTinue  IF OVERUNDERFLOW

LLL3: Counter.Increment OVERUNDERFLOW
      Trigger.A IF !OVERUNDERFLOW
      GOTO LLL1 IF BetaBreak
```

Flag — Flag control

Format:	**Flag.**<*mode*> <*name*> [**IF** <*condition*>]
<*mode*>:	**FALSE** **OFF (obsolete)** **ON (obsolete)** **Toggle** **TRUE**

Flags are used to mark event occurrences. Flags have to be declared at the beginning of a trigger program. The default state at the beginning is OFF. The current state of the used flags is visible in real time in the **analyzer state window**. Flags are also sampled in the trace buffer.

FALSE, OFF	Resets the flag.
TRUE, ON	Sets the flag.
Toggle	Reverses the current state.

Set Flag1 if timer_1 has not expired.

```
FLAGS Flag1                    ; declaration

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

Toggle Flag4 if data_event occurs.

```
Flag.Toggle Flag4 IF 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 following table shows the number of trigger levels available on each analyzer hardware:

Analyzer Type	Trigger Levels
ICD	4

Mark — Recording markers

Format:	**Mark.** *<name>* [**IF** *<condition>*]
<name>:	**A** **B**

Four markers can be used to mark specific events in trace memory. They make it easier to find and display special events, allow time displays between the markers and detailed statistic analysis. The markers are set when the specified **condition** is true. They cannot be used as input events to the trigger unit, like **Flags**.

The following program is used for detailed performance analysis of functions. It samples the entry and exit points of each function and marks the entries with 'A' markers and the exits with 'B' markers:

```
Sample.enable IF AlphaBreak
Sample.enable IF BetaBreak
Mark.A        IF AlphaBreak
Mark.B        IF BetaBreak
```

Out

Output control

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

Six signals can be generated to trigger other devices (e.g. analyzers or oscilloscopes) or to stimulate the target hardware. Two of these signals are accessible via coaxial socket connectors at the back of the analyzer chassis, the others can be accessed via an output probe at the front of the analyzer chassis.

A	Activates the universal bidirectional coaxial output A at the back of the analyzer chassis.
B	Activates the universal coaxial output B at the back of the analyzer chassis.
C	Activates the universal trigger outputs located at the front socket connector "OUT".

Release trigger line A if **address selector** CharlyBreak is active.

```
Out.A IF CharlyBreak
```

The trigger output lines for the socket “OUT” at the front of the chassis can be accessed via an output probe. The **output probe's** pin assignment is described in document trace_user.pdf.

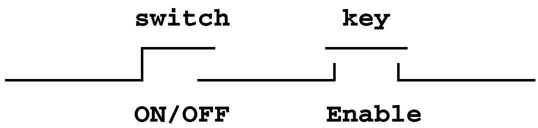
Format:	**Sample**[**.**<*mode*>] [**IF** <*condition*>]
<*mode*>:	**Enable** **OFF** **ON**

Control trace memory recording. The instructions **Sample.ON** and **Sample.Enable** will be programmed automatically, if they aren't used in the trigger program.

These instructions do not effect the recording of the trigger event (marked with T), the first cycle (marked with Go) and last cycle before the user program will stop (marked with BRK).

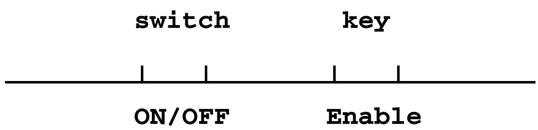
enable	Releases trace memory for recording when the specified condition is true.
OFF	Disables the Flip-flop for sampling.
ON	Enables the Flip-flop for sampling.

The instructions **ON**, **OFF** and **Enable** function 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.

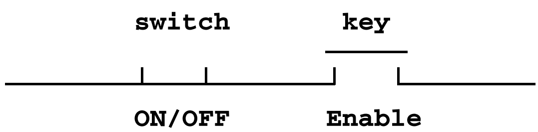


Only if the switch and the key are closed sampling is done.

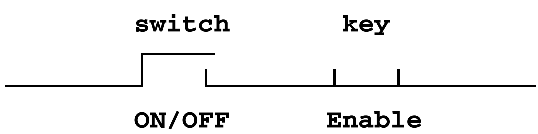
If neither ON/OFF nor Enable are used in the complete trigger program, the switch and the key are closed, that means all cycles are recorded (Implicit global **Sample.ON IF TRUE** and **Sample.enable IF TRUE**).



If only Enable is used in the trigger program, the switch ON/OFF is closed automatically, then 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, then 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 equally and will sample all bus cycles:

```
Sample.Enable IF TRUE
Sample.enable
S.e
s
```

Sample only write cycle:

```
Sample.enable IF Write
```

The analyzer starts and waits in Level0 without recording till the appearance of the declared address selector AlphaBreak (INT3_Service). That cycle (memory access to AlphaBreak) causes one sample and change to the level "Level1". In this level all cycles are recorded.

```
; declaration area
ADDRESS AlphaBreak INT3_Service
...

; local area
Level0: Sample.enable IF AlphaBreak
        CONTinue      IF AlphaBreak
Level1: Sample.enable
...
```

Trigger — Trigger control

Format:	**Trigger.**<*mode*> [**IF** <*condition*>]
<*mode*>:	**PODBUS** (only ICD) **Pulse**

Trigger other systems.

PODBUS	Activates bus trigger line A from PODBUS
Pulse	Releases a pulse of the pulse generator (**PULSE**).

Whenever a write access with data word 0x0EE55 is performed on the data bus, the **spot system** executes a spotpoint.

```
;declaration
DATA.W DAT_1 0x0EE55

;global or local instruction
Trigger.Spot IF DAT_1&&Write
```

Whenever a write access to the memory address func_9 is executed a pulse is generated on the STROBE probe. The pulse width and polarity is controlled by the **PULSE** command.

```
; declaration
ADDRESS BetaBreak func_9

; global or local instruction
Trigger.Pulse IF BetaBreak&&Write
```

The **execution will stop** at write access to address SD:0x1488 with the data 0x30.

```
; declaration
ADDRESS AlphaBreak sd:0x1488
DATA.B  Stop_value 0x30

; global or local instruction
Trigger.A IF AlphaBreak&&Stop_value&&Write
```

If a memory access to address "TRAP" is executed the **exception unit** will issue the selected exception.

```
; declaration
ADDRESS BetaBreak TRAP

; global or local instruction
Trigger.eXception IF BetaBreak
```

Data Trace Message based events

Stop the sampling to the trace buffer if a 1 as a byte is written to the variable flags[3].

To declare the input event - write of byte 1 to the address flags[3] - a **Data Trace Message Qualifier** has to be used. The CTU provides 2 Data Trace Message Qualifiers. The 2 Data Trace Message Qualifiers can be used to qualify 2 single Data Trace Message Qualifiers or to qualify 1 Data Trace Message Qualifier for an address range.

The special feature of a Data Trace Message Qualifier is that the full data address and the full data value is reconstructed by the PowerTrace hardware in real-time out of the compressed information provided by the Data Trace Messages.

Data Trace Message Qualifiers have to be declared before they can be used in a trigger program.

Format:	**ADDRESS** *<selector>* *<address>*| *<range>* ***/HARD*** [*<option>* …]
<selector>:	**AlphaBreak** **BetaBreak** **CharlyBreak** **DeltaBreak** **EchoBreak**
<option>:	**Read | Write | ReadWrite** **Data.auto** *<value>* **| Data.Byte** *<value>* **| Data.Word** *<value>* **|** **Data.Long** *<value>*

```
ADDRESS AlphaBreak 0x7403             /HARD /Write /DATA.Byte 0x01

ADDRESS AlphaBreak V.RANGE(flags[3]) /HARD /Write /DATA.Byte 0

ADDRESS AlphaBreak V.RANGE(flags)    /HARD /Write /DATA.Byte 1
```

The CTU can either stop the sampling to the trace buffer or the program execution when the trigger event defined by the Data Trace Message Qualifier occurs.

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

Example: Trace trigger on data value

Full example for stopping the sampling to the trace buffer:

```
NEXUS.DTM Write          ;enable data trace messaging for write accesses

Analyzer.ReProgram       ;set trigger program
(
  ADDRESS AlphaBreak V.RANGE(flags[3]) /HARD /WRITE /DATA.Byte 1
  BREAK.TRACE IF AlphaBreak
)

Go                       ;run application
```

Example: Program break on data value

Full example for a data value breakpoint on MPC55XX (e200z1, z3, z6 cores do not provide DVC). In order to prevent FIFO overflows, data trace is limited to the variable of interest. The stop of the program execution is delayed (asynchronous stop). For this reason the defined event is marked with an “A” marker in the trace listing.

```
Var.Break.Set flags[3] /Write /TraceData ;data trace only for flags[3]
TrOnchip.EVTI ON         ;use EVTI to halt the core as fast as possible

Analyzer.ReProgram       ;set trigger program
(
  ADDRESS AlphaBreak V.RANGE(flags[3]) /HARD /WRITE /DATA.Byte 1
  BREAK.Program IF AlphaBreak
  MARK.A        IF AlphaBreak
)

Go                       ;run application

WAIT !STATE.RUN()        ;wait until core halted
Analyzer.List MARK DEFault ;show program flow with markers
```

The reconstruction of the full data address and the full data value by the PowerTrace hardware can fail if too many data trace messages are generated in quick succession.

Watchpoint hit message based events

Analyze the run-time behavior of the function sieve.

The basic ideas for this analysis are:

- generate a watchpoint hit message (WHM) at the entry and at the exit of the function sieve.
- sample only these watchpoint hit messages to the trace buffer.
- to differentiate between the watchpoint hit messages generated for the function entry and those generated for the function exit, mark the Watchpoint Trace Messages generated for the function entry with an A marker and the Watchpoint Trace Messages generated for the function exit with a B marker

Watchpoint hit messages have to be declared before they can be used in a trigger program.

Format:	**ADDRESS** *<selector>* *<address>*| *<range>* **/Onchip /Program**
<selector>	**AlphaBreak** **BetaBreak** **CharlyBreak** **DeltaBreak** **EchoBreak**

```
ADDRESS AlphaBreak sieve              /Program /Onchip

ADDRESS BetaBreak sYmbol.EXIT(sieve)  /Program /Onchip
```

The following trigger instructions are available to control the sampling to the trace buffer:

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

Enable	Releases trace memory for recording when the specified condition is true.
OFF	Switch the sampling to the trace buffer to OFF
ON	Switch the sampling to the trace buffer to ON.

The following trigger instructions are available to mark a record in the trace buffer:

Format:	**Mark**[**.**<*marker*>] [**IF** <*condition*>]
<*marker*>:	**A** **B**

Example: Runtime measurement with markers

Now the full example:

```
NEXUS.BTM OFF           ;optional: disable program/data trace for maximum
NEXUS.DTM OFF           ;accuracy (only watchpoint messages are used)

Analyzer.ReProgram      ;set trigger program
(
  ADDRESS AlphaBreak sYmbol.BEGIN(sieve) /Program /Onchip
  ADDRESS BetaBreak  sYmbol.EXIT(sieve)  /Program /Onchip

  Mark.A IF AlphaBreak
  Mark.B IF BetaBreak
)

Go                                 ;run application
WAIT 2.s
Break

Analyzer.STATistic.DURation        ;display a function run-time statistic
Analyzer.PROfileChart.DURation     ;display a function run-time chart
```

If the run-time analysis of the function sieve showed, that the function sieve took several times much longer then you expected, a new trigger program can help you to find the reason for this behavior.

The basis idea of this new trigger program is:

- stop the program execution if the function sieve takes longer the 80 µs.

To write this trigger program 2 new concepts of the trigger programming language are required:

- Time Counters
- Trigger Levels

Time Counters: The CTU provides 3 45-bit time counter with a resolution of 20 ns. Before a Time Counter can be used in a trigger program it has to be declared.

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

```
TImeCouNTer sievec             80.us

TImeCouNTer interrupt_response 10.ms
```

The following trigger instructions can be used to control the Time Counters:

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

Increment	Increment the counter if the specified condition is matched.
OFF	Switch the counter to ON if the specified condition is matched.
ON	Switch the counter to OFF if the specified condition is matched.
Restart	The counter is reset to zero if the specified condition is matched.

If a Time Counter is used in a condition, the Time Counter is a true event when it has reached the declared time value. The current contents of a Time Counter can be see in the **Trace.state** window.

```
Counter.Increment sievec             IF AlphaBreak

Counter.Restart   interrupt_response IF BetaBreak

Break.Program                        IF sievec
```

Trigger Level: The CTU provides 4 trigger levels.

- A trigger level starts at its label.
- A trigger level ends at the following label or at the end of the trigger program.
- The levels determine which trigger instructions are active at the same time.

Changing a trigger level is done by the following trigger instruction:

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

Example: Program break based function runtime

Here the full example:

```
Analyzer.ReProgram
(
  ADDRESS AlphaBreak sYmbol.BEGIN(sieve) /Program /Onchip
  ADDRESS BetaBreak  sYmbol.EXIT(sieve)  /Program /Onchip

  TImeCouNTer sievec 80.us

  start:
    GOTO insieve IF AlphaBreak

  insieve:
    Counter.Increment sievec
    Counter.Restart   sievec, GOTO start IF BetaBreak
    BREAK.PROGRAM IF sievec&&!BetaBreak
)

Go
WAIT !STATE.RUN()
PRINT "Function sieve exceeded maximum runtime!"
```

The complex trigger unit also provides flags to store an internal state. This state can be used as element of conditions.

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

Using external signals with the CTU

The CTU supports two input signals. The IN input of the NEXUS adapter, and the PodBus trigger signal.

The IN input is labeled “IX0” on the NEXUS AutoFocus adapter LA-7630 and “IN0” on LA-7610.

The PodBus trigger signal can be either an internal source (Debug module, Power Probe or Power Integrator logic analyzers) or it can stem from an external source through the “Trigger” connector of the debug module.

Format:	[<*action*>] [**IF** <*condition*>]	
<*condition*>:	**BUSA | !BUSA**	(PodBus trigger signal)
	IN | !IN	(IN connector of NEXUS adapter)

There are also output signals available. The OUT output of the NEXUS adapter, and the PodBus trigger signal.

The OUT output is labeled “OX0” on the NEXUS AutoFocus adapter LA-7630 and “OUT0” on LA-7610.

The PodBus trigger signal can trigger any device on the PodBus (Debug module, Power Probe or Power Integrator logic analyzers). It can also be used to trigger external devices using the “Trigger” connector of the debug module.

Format:	[<*action*>] [**IF** <*condition*>]	
<*action*>:	**TRIGGER.PODBUS**	(PodBus trigger signal)
	OUT.A	(OUT connector of NEXUS adapter)

Example: Record single message on rising edge of trigger input

The next example demonstrates how to control trace recording based on an external signal connected to the Trigger input of the debug module.

```
Analyzer.ReProgram
(
waitrisingedge:
  Sample.Enable         IF BUSA
  GOTO waitfallingedge IF BUSA

waitfallingedge:
  GOTO waitrisingedge  IF !BUSA
  )
)

;configure Trigger connector as high-active input
TrBus.Connect In
TrBus.Mode HIGH
```

Example: Program break based on pulse interval of IN input

The next example demonstrates how to use a flag to monitor a state change. Mark B is set on every rising edge of the IN signal. The program execution is halted if the time interval of two rising edges is shorter than 10ms.

```
Analyzer.ReProgram
(
  TImeCouNTer interval 10.ms
  FLAGS       lastin

  ;FLAG lastin is value of IN, delayed by one CTU cycle
  FLAG.TRUE   lastin       if IN
  FLAG.FALSE  lastin       if !IN
  ;generate MARK.B on rising edge of IN
  MARK.B                   if IN&&!lastin

waitnext:
  Counter.ON      interval
  Counter.Restart interval if IN&&!lastin
  GOTO criticaltime        if IN&&!lastin

criticaltime:
  GOTO waitnext if interval               ;interval elapsed -> OK
  GOTO timeviol if IN&&!lastin&&!interval ;pulse within interval -> fail

timeviol:
  Counter.OFF interval
  BREAK.PROGRAM
)

Go
WAIT !STATE.RUN()
PRINT "Found two pulses within one 10ms interval"
Analyzer.List Trigger.0 MARK.B TIME.MARKBB DEFault
```

Input Event	Meaning
IN	external input event IN0 or IN1 occurred
CM, TCODE_2, TCODE_CM	correlation message
DBM, TCODE_3, TCODE_DBM	direct branch message
DBSM, TCODE_B, TCODE_DBSM	direct branch sync message
DRM, TCODE_6, TCODE_DRM	data read message
DRSM, TCODE_E, TCODE_DRSM	data read sync message
DSM, TCODE_0, TCODE_DSM	debug status message
DWM, TCODE_5, TCODE_DWM	data write message
DWSM, TCODE_D, TCODE_DWSM	data write sync message
EM, TCODE_8, TCODE_EM	error message
EM_0, TCODE_8_0	error message 0 - OTM loss
EM_1, TCODE_8_1	error message 1 - BTM loss
EM_2, TCODE_8_2	error message 2 - DTM loss
EM_3, TCODE_8_3	error message 3 - r/w access error
EM_5, TCODE_8_5	error message 2 - invalid access opcode
EM_6, TCODE_8_6	error message 6 - WHM loss
EM_7, TCODE_8_7	error message 7 - BTM/DTM/OTM loss
EM_8, TCODE_8_8	error message 8 - BTM/DTM/OTM/WHM loss
EM_24, TCODE_8_24	error message 24
EM_31, TCODE_8_31	error message 31
HBM, TCODE_1C, TCODE_HBM	hardware event message
HWM, TCODE_38, TCODE_HWM	hardware event message
HWSM, TCODE_1D, TCODE_HWSM	hardware event sync message
IBM, TCODE_4, TCODE_IBM	indirect branch message
IBSM, TCODE_C, TCODE_IBSM	indirect branch sync message
OTM, TCODE_2, TCODE_OTM	ownership trace message

RBM, TCODE_1E, TCODE_RBM	repeat branch message
RFM, TCODE_1B, TCODE_RFM	resource full message
WHM, TCODE_F, TCODE_WHM	watchpoint hit message