LAUTERBACH
DEVELOPMENT TOOLS

Application Note for Complex Trigger Language

Release 09.2023

MANUAL

Application Note for Complex Trigger Language

Application Note for Complex Trigger Language

Version 09-Oct-2023

History

15-Feb-2023	Added onchip CTL support for miniMCDS.
29-Jun-2022	Initial version.

NOTE:	**This manual is still under construction.**

Introduction

Complex Trigger Language (CTL) is a high-level parallel programming language. The main idea behind CTL is to offer TRACE32 users a simple and powerful interface to debug and trace complex scenarios without any specific knowledge about the low-level onchip triggers logic. The language is defined to grant a fine control of the debug logic and trace sources. CTL enables the user to fully benefit from the debug and trace capabilities offered by the target while keeping the entire focus on debugging and testing.

Additionally to onchip triggers logic, CTL supports **Trace.Find** as a target. This empowers TRACE32 with an advanced trace find feature. When operating in SPY mode, the trace find results could be used as a test vehicle for onchip triggers. This enables CTL for targets that do not provide any hardware support to implement complex triggers.

This document is divided into the following sections:

1. **Basic Structure of CTL Programs**
2. **TRACE32 Commands Using CTL Programs**
3. **CTL for Onchip Triggers Logic**

   Separate sub-sections discuss peculiarities of each implementation for onchip CTL and present selected use cases with example CTL programs:

   - **CTL for TriCore MCDS**
   - **CTL for Arm ETM**
4. **Examples for CTL Trace Find**
5. **Keyword Reference: CTL Conditions/Triggers**
6. **Keyword Reference: CTL Actions**
7. **CTL programming errors**

NOTE:	In this document, simple triggers refer to breakpoints that are enabled via **Break.Set** commands. CTL is not intended to replace simple triggers, although most breakpoints could be easily written in CTL as well. The reason is that the onchip trigger unit programmed by CTL might behave differently from the trigger logic programmed by simple triggers. E.g. for TriCore the stopping breakpoints set via simple triggers are programmed to OCDS (break-before-make breakpoints). While the CTL **Break** actions are programmed to MCDS (the cores are stopped a few instructions after the trigger event).

Basic Structure of CTL Programs

CTL makes an abstraction of the target architecture whenever possible. Apart from a few exceptions, e.g. special bus agents, the syntax is architecture-independent and valid for all CTL targets.

Following is the list of elements composing CTL programs:

- Complex statements
- Agents (optional)
- Levels (optional)
- Comments (start with `//` or `;` and end with the next line break).

```
; this is a comment
// this is also a comment
[<agent>::]
[<level>:]
    IF <condition>   ; this is another comment
        <action>
```

CTL is **not** white space sensitive, but it is recommended to use indentations for better readability of the program.

CTL keywords are **not** case-sensitive. The following examples of CTL programs are similar:

```
if var.program(sieve)
    traceenable program
```

```
IF Var.Program(sieve)
    TraceEnable Program
```

Upper case letters indicate the short forms of CTL keywords and must not be omitted. All lower case letters can be omitted. Following is a short form of the above example program:

```
if v.p(sieve)
    te p
```

Complex Statements

Complex statements are the basic elements of any CTL program.

Each complex statement is composed of:

- One condition
- One or multiple action(s) to be performed when the condition is satisfied

A CTL condition starts with the keyword `IF` followed by a logical combination of one or more sub-expressions. The condition's sub-expressions could be issued from different or similar qualifier types (program comparators, memory address comparators, access types,...).

A line break separates the condition from its associated action(s).

Multiple actions of a complex statement must be separated by line breaks.

Example:

```
// Enable program trace for the first instruction of the function sieve
IF Program(ENTRY:sieve)
    TraceEnable Program
```

Details about CTL conditions and actions are provided in the following sections:

- **Keyword Reference: CTL Conditions/Triggers**
- **Keyword Reference: CTL Actions**

NOTE:	Given that CTL is a parallel programming language, the order in which the complex statements appear in a CTL program is not important. All the complex statements are evaluated in parallel. If the CTL program is implementing a state machine, all the complex statements belonging to the active state are evaluated in parallel. Complex statements belonging to the inactive levels of the state machine are not evaluated.

Agents

Each CTL condition is evaluated for a specified agent, and likewise, each CTL action is to be performed by a specified agent. The syntax to specify an agent is as follows:

```
<agent_name>::
```

The CTL syntax allows using agents with global scopes or local scopes.

Local scope agents are to be specified as prefixes to the associated actions and/or sub-expressions of the CTL conditions.

Example:

```
IF CORE0::Program(ENTRY:sieve0)||CORE1::Program(ENTRY:sieve1)
    INCrement mycounter
```

An agent that is not prefixing any action or condition's sub-expression is a global scope agent.

The scope of a global agent starts from the specification of the agent name and ends with the specification of another global scope agent name.

Example:

```
CORE0::
    IF Program(sieve0)
        TraceEnable Program
    IF Var.Write(mstatic1)
        TraceEnable Write Address Data
CORE1::
    IF Var.Program(func2)
        TraceEnable Write Address Data
```

Two types of agents are to be distinguished:

- Core agents
- Bus agents/monitors

Core Agents

The syntax to specify a core agent is as follows:

```
CORE<n>::
```

The index *n* refers to the logical core number controlled by the TRCAE32 PowerView instance.

`SplitCORE::` and `JoinCORE::` are multicore agents. These are used to specify that a complex statement is to be evaluated for all the cores that are assigned to the PowerView instance (except for limitations from the target). The difference between `SplitCORE::` and `JoinCORE::` is as follows:

- `SplitCORE::` specifies that each statement is to be evaluated for each core separately.
- `JoinCORE::` specifies that all cores collaborate to evaluate a statement.

Example 1:

The TRACE32 PowerView instance is controlling 2 cores of the target CPU (**CORE.ASSIGN** 1. 2.).

In the following CTL program `CORE1::` is used as global scope agent:

```
CORE1::
    IF Var.Write(mstatic)
        TraceEnable Program
```

This enables program flow trace of the second core if write access to the variable `mstatic` is performed by the same core.

Example 2:

The TRACE32 PowerView instance is controlling 2 cores of the target CPU (**CORE.ASSIGN** 1. 2.).

In the following CTL program `SplitCORE::` is used as a global scope agent:

```
SplitCORE::
    IF Var.Write(mstatic)
        TraceEnable Program
```

When loading this CTL program, the complex statements is programmed for both cores separately. This means that:

- The program flow trace of the first core is enabled when the latter performs a write access to the variable `mstatic`.
- The program flow trace of the second core is enabled when the latter performs a write access to the variable `mstatic`.

Example 3:

The TRACE32 PowerView instance is controlling 2 cores of the target CPU (**CORE.ASSIGN** 1. 2.).

In the following CTL program `JoinCORE::` is used as global scope agent:

```
JoinCORE::
    IF Var.Write(mstatic)
        TraceEnable Program
```

When loading this CTL program, both cores collaborate to evaluate the complex statement. This means that the program flow trace of both cores is enabled when one of both cores performs a write access to the variable `mstatic`.

Bus Monitors

When using bus agents the trace sources are observed at the level of bus transactions. Thus, no program trace or program triggers are available for bus agents.

The list of bus agents is architecture-dependent. The list of available agents varies also depending on the target CPU.

In some cases, the bus name is used as the CTL agent. In other cases, bus agents refer either to bus masters initiating the transaction (e.g. DMA) or bus slaves incurring the transaction (e.g. memory units).

Examples of bus agents:

- `SPB::` is to be used for observing the transactions on the Shared Peripheral Bus (SPB) of a TriCore AURIX device.
- `SRI-LMU::` is to be used for observing accesses to Local Memory Unit and EMEM of an AURIX TC2x device via the Shared Resource Interface (SRI) fabric.
- `SRI-DMA::` is to be used for observing DMA transactions on AURIX TC3x device via the SRI fabric.

Default Agent

If no agent is specified, `SplitCORE::` is used as the default agent.

In the following example, both CTL programs are equivalent.

Example:

```
IF Var.Program(sieve)
    TraceEnable Program
```

```
SplitCORE::
IF Var.Program(sieve)
    TraceEnable Program
```

Using CTL State Machines

State machines could be used for debugging sequential events. The syntax to specify levels of state machines is as follows:

```
<level_name>:
```

The scope of a state machine level starts from the specification of the level name and ends with the specification of another level name.

The complex statements belonging to the scope of a state machine level are only evaluated when the level is active.

The first level specified in a CTL program is handled as the start level.

Transitions between different levels of a state machine are to be specified using **GOTO** *<target_level>* actions.

Example:

```
CORE0::
start:
    // transition to level1 statement
    IF Program(ENTRY:sieve)
        GOTO level1
level1:
    // transition back to start level
    IF Program(RETURN:sieve)
        GOTO start
    // stopping statement
    IF Var.Write(mstatic1==2)
        Break
```

In this example, the CTL program implements a state machine with 2 levels/states.

The state machine is initially at the `start` state. As soon as `core0` executes the entry point of `sieve()` function, a state transition to `level1` occurs. When executing the return instruction of the function `sieve()`, a state transition back to the level `start` occurs.

This implicates that `level1` is active as long as `core0` is executing the function `sieve()` or one of its nested functions. The level `start` is active otherwise.

Only when `level1` is active the stopping statement is evaluated.

Activating this CTL program will cause the target to **Break** when the following conditions are fulfilled:

- The agent `core0` is executing the function `sieve()` or one of its nested functions.
- The agent `core0` writes the value 2 to the variable `mstatic1`.

If initially, `core0` is already in `sieve()`, the stopping statement would not be evaluated until the next execution of `sieve()`, triggering a state machine transition to `level1`.

Multiple State Machines

CTL allows programming multiple state machines. Levels that belong to a state machine are to be prefixed by the state machine name as follows:

```
<state_machine>.<level_name>:
```

Example:

In the following CTL program `m1` and `m2` are independent state machines:

- `m1` specifies that the target is to be stopped if `func1()` is called by `func9()` or one of its nested functions.
- `m2` specifies that the target is to be stopped if the variable `mstatic1` is written outside `func2()` or one of its nested functions.

```
//-----------------------------------------
// implementation of the state machine m1
//-----------------------------------------
m1.start:
    IF Program(ENTRY:func9)
        GOTO m1.level1
m1.level1:
    IF Program(RETURN:func9)
        GOTO m1.start
    IF Program(ENTRY:func1)
        Break

//-----------------------------------------
// implementation of the state machine m2
//-----------------------------------------
m2.level0:
    IF Program(ENTRY:func2)
        GOTO m2.level1
    IF Var.Write(mstatic1)
        Break
m2.level1:
    IF Program(RETURN:func2)
        GOTO m2.level0
```

TRACE32 Commands Using CTL Programs

CTL could be used with different targets:

- CTL for Onchip Triggers Logic
- CTL for Trace Find
- CTL Streaming Trace Trigger

This section presents different CTL targets and their corresponding TRACE32 commands.

CTL Onchip Triggers Logic

CTL for onchip trigger logic (or Onchip CTL) requires that the target CPU provides the onchip logic to implement complex triggers. While the complexity level is limited by the onchip resources provided by the trigger unit, onchip CTL has the fastest response time compared to other CTL targets.

The following table recapitulates the list of TRACE32 commands that are used for onchip CTL.

Break.Program	Opens interactive softkey-driven editor for CTL programs
Break.ReProgram	Activates existing program file
Break.ViewProgram	Opens a window that shows the state of the CTL trigger unit
Break.CLEAR	Resets onchip trigger logic that is programmed by CTL. This command doesn't reset simple triggers.

More information about CTL onchip triggers can be found in the chapter **CTL for Onchip Triggers Logic.**

CTL for Trace Find

Using CTL for trace find allows searching for the occurrence(s) of complex events in the trace recording, e.g. sequential events happening in a specific or even arbitrary order.

After the CTL program for trace find is activated, the commands **Trace.Find** and **Trace.FindAll** are to be used to find the matching items in the trace recording that are fulfilling the complex search criteria as specified by the CTL program.

CTL for trace find does not require any onchip triggering logic. Thus, CTL for trace find has unlimited complexity and can be used with any target providing trace capabilities.

When using **<trace>.Mode STREAM,** it is possible to analyze trace results while streaming using the option /**SPY**:

```
Trace.FindAll /SPY
```

The search result could be used as a test vehicle for onchip triggers: The trace stream file is processed and analyzed at runtime (while the target is running and the trace is armed) to search for items fulfilling the complex search criteria as specified by the CTL program. The target and/or the trace recording could be stopped (**Break** or **TraceTrigger**) when the scenario of interest is recorded and detected.

Compared to Onchip CTL, CTL for Trace Find has a longer response time. The response time is affected by:

- The processing capacity of the host computer.
- The bandwidth of the whole trace transmission chain (from TRACE32 debug and trace tool to the hard drive of the host computer).

The following table recapitulates the list of the TRACE32 commands that are used for CTL Trace Find.

Trace.FindProgram	Opens interactive softkey-driven editor for trace find CTL programs
Trace.FindReProgram	Activates existing program file for trace find target
Trace.FindViewProgram	Opens a window that shows the state of the CTL trace find program

CTL Streaming Trace Trigger

tbd.

RTS.Program	tbd.
RTS.ReProgram	tbd.
RTS.ViewProgram	tbd.
RTS.CLEAR	tbd.

CTL for Onchip Triggers Logic

To use onchip CTL, the target CPU must provide hardware support to implement complex triggers.

The following subsections are independent. Each is discussing onchip CTL implementation for a specific target architecture. Selected use cases and example CTL programs are presented.

- **CTL for TriCore MCDS**
- **CTL for Arm ETM**

CTL for TriCore MCDS

Supported Targets

Onchip CTL is only supported for AURIX devices with available MCDS modules (MCDS, MCDSlight, or miniMCDS). CTL support for miniMCDS requires TRACE32 release 2023/02 or newer.

The PRACTICE function **MCDS.Module.NAME()** could be used to check the name of the MCDS module for the selected CPU.

Multicore Support

The MCDS module of TriCore devices is restricted to generating trace and trigger information for a limited number of cores. The consequence is that the multicore agents are restricted to the TriCore cores that are assigned to the PowerView instance, and that are selected as core agents via the MCDS window or using the commands **MCDS.ProgramTrace.Agents** and **MCDS.DataTrace.Agents**.

Selective Bus Trace

CTL provides a simple interface for selective bus trace. The complex statements are to be assigned to the appropriate bus agents.

- The agent `SPB::` is to be used for tracing and triggering over the System Peripheral Bus (SPB).
- The TriCore MCDS module is using trace multiplexers to select which trace sources are to be observed on the Shared Resource Interconnect (SRI) fabric. SRI agent names are formed by the SRI- prefix, followed by the name of the trace source as defined by Trace Source Multiplexer setting options in the Infineon documentation. Following are some examples:
  - `SRI-LMU::` AURIX TC2x agent name to observe access to LMU SRAM and EMEM via SRI.
  - `SRI-OLDA::` AURIX TC3x agent name to observe access to Online Data Acquisition via SRI.
  - `SRI-DMA::` AURIX TC3x agent name to observe DMA transactions via SRI.
  - `SRI-CPU1::` AURIX TC2x/TC3x agent name to observe access to TriCore1 local memories via SRI.

Not all trace sources are available for all target CPUs. The exhaustive list of available bus agents for each CPU selection could be displayed by clicking on the advanced button of the MCDS window.

Automatic Configuration of the Trace Source Multiplexers

When activating a CTL program, an automatic configuration of the MCDS trace source multiplexers is performed. TRACE32 combines the list of agents issued from the following configurations and configures the trace source multiplexers accordingly:

- The list of agents that are used by the compiled CTL program.
- The list of core agents that are selected by the commands **MCDS.ProgramTrace.Agents** and **MCDS.DataTrace.Agents**
- The list of bus trace agents that are selected by the command **MCDS.BusTrace.Agents**
- The status of the peripheral trace that is configured by the command **MCDS.PERipheralTrace**

An error is thrown if there is no valid MCDS configuration to observe all the selected agents at the same time. The user must decide which agents are most important to be observed for his use case.

	When a CTL program is activated, the configuration of the trace source multiplexers performed by the same PowerView instance via the commands **MCDS.SOURCE.Set** is discarded.

TriCore Data Trace: COREx Vs. SRI-CPUx

Using the MCDS module of AURIX devices there are 2 options to observe memory accesses relatively to a TriCore core:

- Observe the read/write accesses performed by the core (e.g. when executing a load or store instruction). In this configuration, the TriCore core is observed as a bus master.
- Observe the read/write accesses to the core local memories via the SRI fabric. In this configuration, the TriCore core is observed as a bus slave incurring the access.

Different CTL agents are to be used in both cases.

Example:

`CORE0::` is used to observe the first core assigned to the PowerView instance as a master. In this case, memory accesses performed by the core are to be observed.

`SRI-CPU0::` is used to observe TriCore0 as an SRI slave. In this case, accesses to the core local memories via SRI are to be observed. The agents are distinguished by the physical index of the TriCore cores (`SRI-CPU1::` refers to TriCore1, …, `SRI-CPU5::` refers to TriCore5).

MCDS module of AURIX TC2x allows a selected TriCore to be observed as a core (read/write accesses generated by the core are observed) and SRI slave at the same time. E.g. a CPU source Multiplexer could be configured to observe the core accesses, and a SRI source multiplexer could be configured to observe the same core as SRI slave.

As opposed to AURIX TC2x, MCDS module of AURIX TC3x only allows the core to be observed either as a master or as an SRI slave but not both at the same time. CTL throws an error when a user program causing such a conflict is enabled.

Limitations

- The current implementation of onchip CTL for TriCore doesn't support complex statements that combine conditions and actions issued from different agents.
- `JointCORE::` agent is currently not supported by onchip CTL for TriCore.
- Due to known behavior of the MCDS module, there is a dead time of up to 2 MCDS clock cycles during counters, flags, and state changes. This must be considered by the user when writing CTL programs or analyzing the test results.

In this section, selected CTL use-cases for TriCore MCDS are presented.

Use case 1: Debug Memory Overwrite

User Story - Part 1:

In this example, an AURIX TC3x emulation device is used (e.g. TC397XE)

The user expects the variable `vdouble` to be only changed by the function `func2c`. But this gets overwritten by other values than the function `func2c` is expected to write.

As a first test, the following CTL program is used to check if TriCore0 is performing any write access to `vdouble` from outside the function `func2c`.

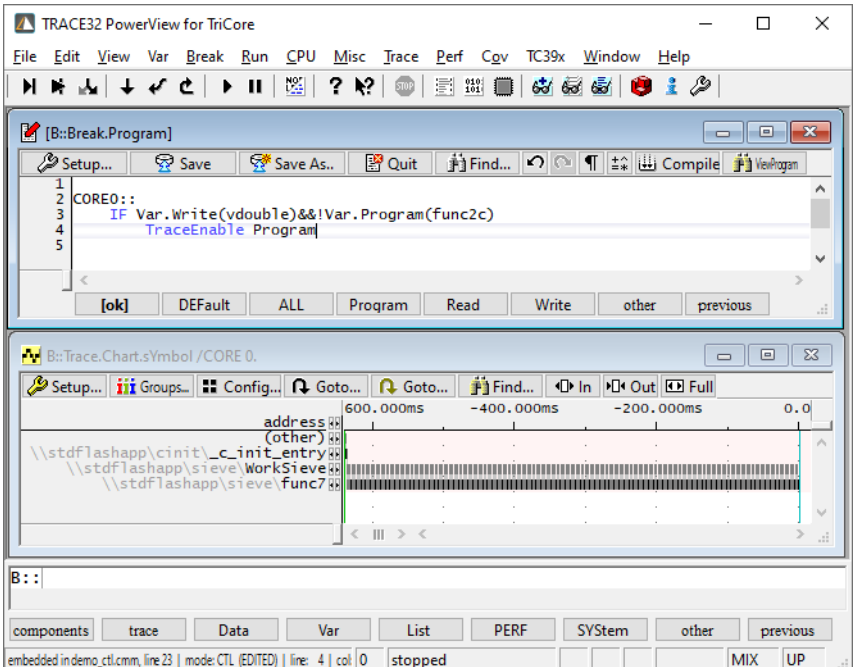
CTL Program:

```
CORE0::
    IF Var.Write(vdouble)&&!Var.Program(func2c)
        TraceEnable Program
```

By prefixing a qualifier with "**Var.**", the address range of the specified HLL expression is used.

Results:

The test shows that the startup code `_c_init_entry` and other functions (`WorkSieve` and `func7`) are also writing to `vdouble` (see the trace chart in the following screenshot).



User Story - Part 2:

The variable `vdouble` is located in the DSPR of TriCore0. The user requirement is to check that no other TriCore cores or bus agents are also writing to this variable. To achieve this, TriCore0 is to be observed as SRI slave using the agent `SRI-CPU0::`

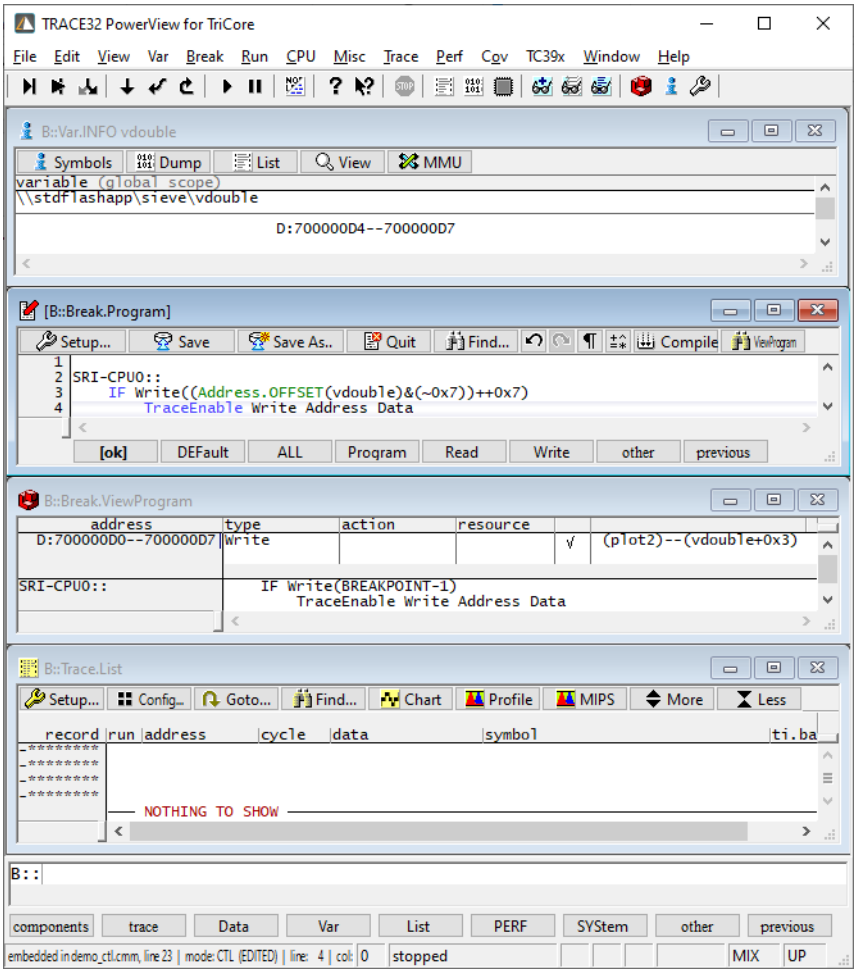
Considering that the write access might be part of a 64-bit burst write, the address range needs to be extended to cover a 64-bit aligned range. PRACTICE functions and other arithmetic calculations could be invoked by the CTL program. In this example the PRACTICE function **ADDRESS.OFFSET()** is used to retrieve the variable address in-order to calculate the 64-bit address range to be observed.

CTL Program:

```
SRI-CPU0::
    IF Write((Address.OFFSET(vdouble)&(~0x7))++0x7)
        TraceEnable Write Address Data
```

Results:

This second test proves that no other write access to `vdouble` is performed by any agent other than TriCore0: The resulting **Trace.List** window in the following screen shot is empty.



In this example, an AURIX TC3x emulation device is used (e.g. TC397XE)

User Story:

In the used test application the variable `mcount` is incremented each iteration of the function `mainloop`. The user needs to measure the runtime of the function `sieve` which is called once by `mainloop` iteration. Runtime measurement is to be started after a specified number of iterations.

The user requirements for this test case are as follows:

- Trace all write accesses to `mcount` (the address and the write values are to be sampled).
- Starting from the iteration number 20000 of the function `mainloop`, the entry and return instructions of the function `sieve` are to be traced.
- The measurements are to be stopped after collecting 1000 runtime samples.

CTL Program:

CTL flags can be used to implement the test requirements:

```
IF Var.Write(mcount)
    TraceEnable Write Address Data

IF Var.Write(mcount==20000.)
    SET myFlag

IF (Program(ENTRY:sieve)||Program(RETURN:sieve))&&FLAG(myFlag)
    TraceEnable Program

IF Var.Write(mcount==21000.)
    Break
```

Results:

The **Trace.List** window shows that program trace (for the entry and return instructions of the function sieve) is started after the value 20000 (0x4E20) is written to `mcount`.

```
B::Trace.List
Setup... Config... Goto... Find... Chart Profile MIPS More Less
  record run address        cycle   data              symbol                          ti.back
-00035538     D:70000030 wr-data         00004E13 ..c39x_sieve_intmem\taskc\mcount   30.300us
-00035523     D:70000030 wr-data         00004E14 ..c39x_sieve_intmem\taskc\mcount   30.380us
-00035509     D:70000030 wr-data         00004E15 ..c39x_sieve_intmem\taskc\mcount   30.410us
-00035494     D:70000030 wr-data         00004E16 ..c39x_sieve_intmem\taskc\mcount   30.380us
-00035479     D:70000030 wr-data         00004E17 ..c39x_sieve_intmem\taskc\mcount   30.320us
-00035465     D:70000030 wr-data         00004E18 ..c39x_sieve_intmem\taskc\mcount   30.300us
-00035451     D:70000030 wr-data         00004E19 ..c39x_sieve_intmem\taskc\mcount   30.380us
-00035437     D:70000030 wr-data         00004E1A ..c39x_sieve_intmem\taskc\mcount   30.410us
-00035422     D:70000030 wr-data         00004E1B ..c39x_sieve_intmem\taskc\mcount   30.380us
-00035407     D:70000030 wr-data         00004E1C ..c39x_sieve_intmem\taskc\mcount   30.320us
-00035392     D:70000030 wr-data         00004E1D ..c39x_sieve_intmem\taskc\mcount   30.300us
-00035378     D:70000030 wr-data         00004E1E ..c39x_sieve_intmem\taskc\mcount   30.380us
-00035364     D:70000030 wr-data         00004E1F ..c39x_sieve_intmem\taskc\mcount   30.410us
-00035349     D:70000030 wr-data         00004E20 ..c39x_sieve_intmem\taskc\mcount   30.380us
-00035338     P:70100BE4 ptrace                   ..tc39x_sieve_intmem\taskc\sieve   25.040us
            ...
            #define MAX_SIZE 128

            char flags[MAX_SIZE+1];

            static int sieve(void)                  /* sieve of erathostenes */
            {
                    register int    i, prime, k;
                    int             count;
       771          int             size = ((mcount)%20000) ? (MAX_SIZE>>3) : MAX_SIZE;
            ld.w      d15,0x70000030   ; d15,mcount
            mov       d0,#0x4E20
            TRACE ENABLE
-00035324     P:70100C54 ptrace                   ..._sieve_intmem\taskc\sieve+0x70   36.120us
       790  }
            ret16
-00035319     P:70100B7A ptrace                   ..ve_intmem\taskc\mainloop+0x23A    0.000us

       728          datas.b[0] = 0x12;
            mov16     d15,#0x12
            st.b      0x70000004,d15   ; datas,d15
-00035311     D:70000030 wr-data         00004E21 ..c39x_sieve_intmem\taskc\mcount    0.590us
            TRACE ENABLE
-00035300     P:70100BE4 ptrace                   ..tc39x_sieve_intmem\taskc\sieve   25.020us
            ...
            #define MAX_SIZE 128

            char flags[MAX_SIZE+1];

            static int sieve(void)                  /* sieve of erathostenes */
            {
                    register int    i, prime, k;
                    int             count;
       771          int             size = ((mcount)%20000) ? (MAX_SIZE>>3) : MAX_SIZE;
            ld.w      d15,0x70000030   ; d15,mcount
            mov       d0,#0x4E20
            TRACE ENABLE
-00035290     P:70100C54 ptrace                   ..._sieve_intmem\taskc\sieve+0x70    4.790us
```

The window **Trace.STATistic.AddressDURation** shows that 1000 runtime measurements of the function `sieve` are recorded (samples: 1000). In a single iteration, the execution time of `sieve` took longer time (max: 36.12µs) than in usual runs (avr:4.741µs)

```
B::Trace.STATistic.AddressDURation sieve sYmbol.EXIT(sieve)
Setup... Chart Zoom Zoom Full
          samples:      1000.  avr:   4.741us  min:   4.710us  max:  36.120us
          total:     638.195ms  in:    4.741ms  out: 633.453ms  ratio:  0.742%

     up to     count      ratio   1%     2%        5%     10%     20%       50%   100
<    4.000us       0.     0.000%
     6.000us     999.    99.900%  ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
     8.000us       0.     0.000%
    10.000us       0.     0.000%
    12.000us       0.     0.000%
    14.000us       0.     0.000%
    16.000us       0.     0.000%
    18.000us       0.     0.000%
    20.000us       0.     0.000%
    22.000us       0.     0.000%
    24.000us       0.     0.000%
    26.000us       0.     0.000%
    28.000us       0.     0.000%
    30.000us       0.     0.000%
    32.000us       0.     0.000%
    34.000us       0.     0.000%
    36.000us       0.     0.000%
>                  1.     0.100%  ◆
```

User Story:

In the used test application the variable `mcount` is incremented each iteration of the `main` loop.

The user needs to start sampling the program flow after a given number of iterations. The user requirements for this test case are as follows:

- Trace all write accesses to `mcount` (the address and the write values are to be sampled).
- Starting from the 5th iteration of the `main` loop, the program flow trace is to be enabled.
- The target is to be stopped after the 10th iteration of the main loop.

CTL Program:

In the following example program, a CTL counter "`mycounter`" is used to count the number of write access to the variable `mcount`.

```
IF Var.Write(mcount)
    TraceEnable Write Address Data
    INCrement mycounter

IF COUNT(mycounter>=5.)
    TraceEnable Program

IF COUNT(mycounter>=10.)
    Break
```

Results:

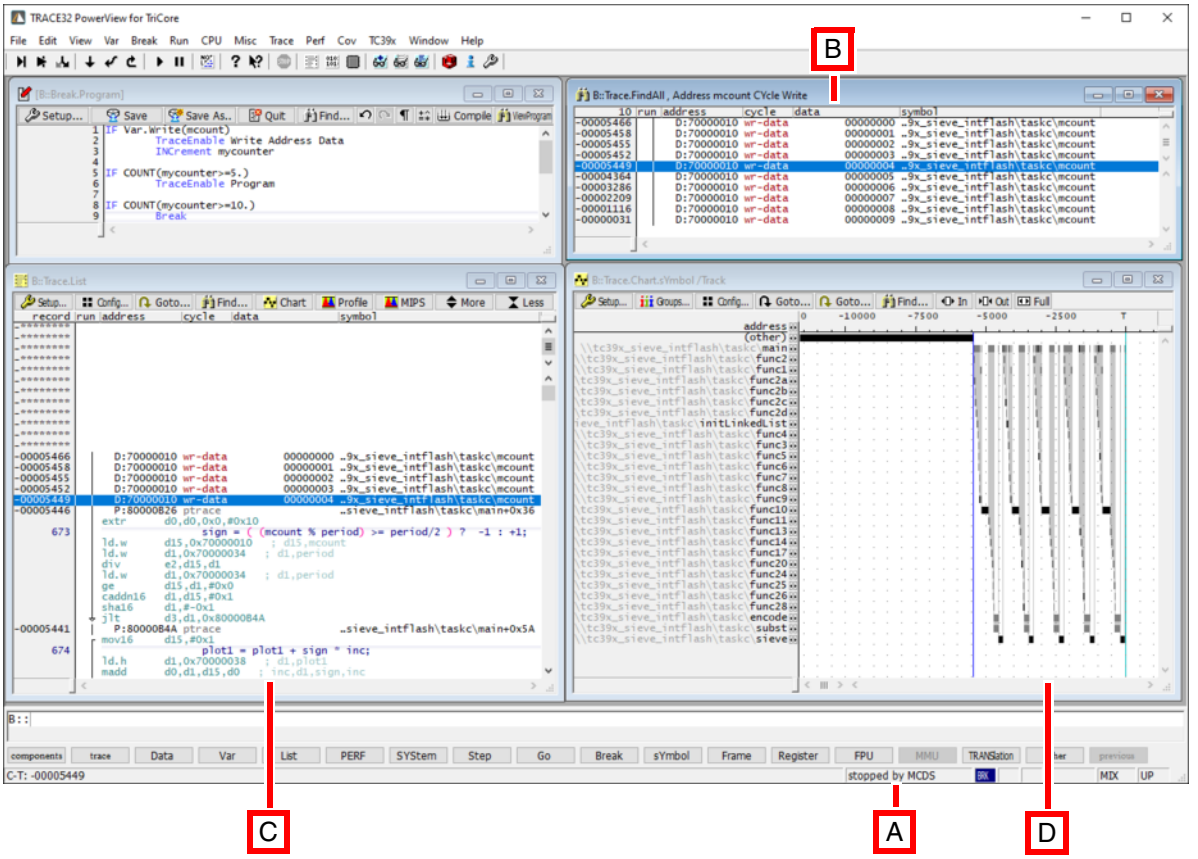
Test results shown in the following screenshot could be interpreted as follows:

A The status bar shows the state "stopped by MCDS". This indicates that an MCDS trigger has stopped the target.

B The trace find window shows that the target executed the main loop for exactly 10 iterations (10 writes to `mcount` are recorded).

C The trace list window shows that the program flow trace was enabled starting from the 5th iteration.

D The trace chart shows that the program trace was enabled during the last 5 iterations: The leaf function `sieve` which is called once per `main` loop was called exactly 5 times.


TRACE32 PowerView for TriCore
B
C
A
D

User Story:

In a real-time context, the execution time of the function `mainloop` must not exceed a maximum specified time of 35 µs. In rare cases, it happens that the execution time exceeds 60 µs.

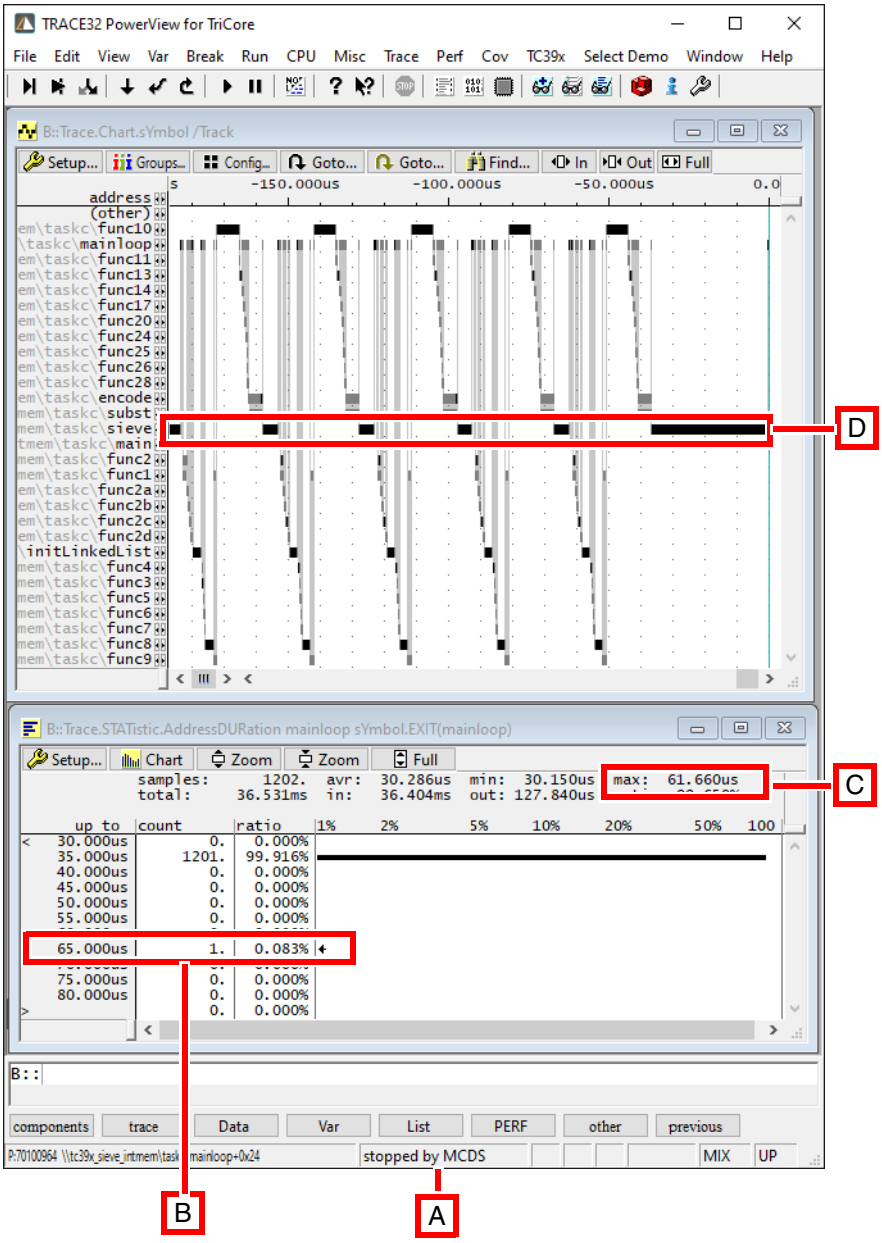
An AURIX TC3x emulation device is used, but the target board doesn't provide an AGBT interface. Only an onchip trace buffer of 2MBytes is available. The user estimates that, when the error case occurs, the program flow history available in the onchip trace buffer should be enough to track down the issue. The requirements are as follows:

- Unconditional program flow trace must be enabled.
- To track the number of execution loops, all write accesses to the variable `mcount` must be recorded.
- When detecting the error case (i.e. the execution time of the function `mainloop` exceeds a maximum specified duration of 35 µs) the trace recording must be kept enabled till the next return of the function `mainloop` is executed.

CTL Program:

```
//-----------------------------------------
// State independent complex statements
//-----------------------------------------

// Enable unconditional program flow trace
IF TRUE()
    TraceEnable Program

// Sample all write access to mcount
IF Var.Write(mcount)
    TraceData

//-----------------------------------------
// State Machine implementation to stop
// tracing and Break if the mainloop
// execution time exceeds 35µs.
//-----------------------------------------
level0:
    IF Program(ENTRY:mainloop)
        GOTO level1

level1:
    // Restart the timer at entry to level1
    IF STATE.ENTER()
        RELOAD task_timer

    // Keep counting as long as level1 is active
    IF TRUE()
        ENABLE task_timer

    // mainloop return with no timeout detected
    // => Go back to level0
    IF Program(RETURN:mainloop)
        GOTO level0

    // A timeout is detected => Go to level3
    IF TIME(task_timer>=35.us)
        GOTO level3

level3:
    // level3 is only reached if a time-out is detected
    // => Stop tracing and break at return of the mainloop
    IF Program(RETURN:mainloop)
        TraceTrigger
        Break
```

Results:



A The status bar shows the state “stopped by MCDS”. This indicates that an MCDS trigger has stopped the target.

B The **Trace.STATistic.AddressDURation** window shows that the execution time of the function `mainloop` has once exceeded the duration of 35 µs.

C The maximum measured duration of `mainloop` is 61.660 µs.

D The **Trace.Chart.sYmbol** window shows that the function `sieve` took a longer execution time than the usual runs. The user must examine the program flow trace and focus his analysis on the function `sieve`.

User Story:

In a real time context, a function `func10` must be called at least once every 35 µs. In rare cases, the time distance between 2 consecutive calls exceeds 60 µs.

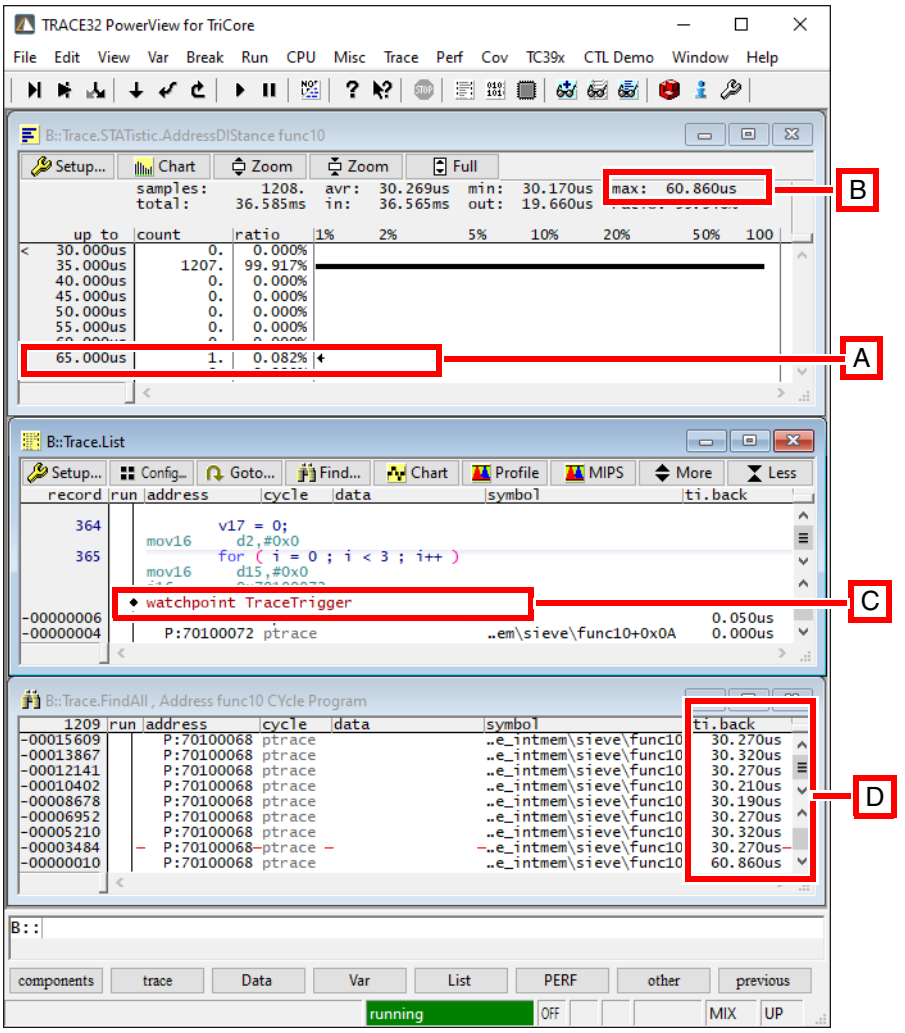
An onchip trace configuration on AURIX TC3x emulation device is used. The user needs to examine the program flow trace when the timing constraint is violated. The requirements are as follows:

- Unconditional program flow must be enabled.
- When the timing constraint is violated, the program flow recording is to be kept enabled until the next call of `func10` is executed.

CTL Program:

```
// Enable unconditional program flow trace
IF TRUE()
    TraceEnable Program

// Set monitoring flag and reload the distance timer at every
// execution of the func10 entry
IF Program(ENTRY:func10)
    SET monitoring_flag
    RELOAD distance_tmr

// Keep incrementing the distance timer as long as the monitoring
// flag is set
IF FLAG(monitoring_flag)
    ENABLE distance_tmr

// Set a time-out flag when the value of the distance monitoring
// timer exceeds 35 µs
IF Time(distance_tmr>35.us)
    SET timeout_flag

// Stop trace recording at the first execution of func10 entry
// following a time-out
IF FLAG(timeout_flag)&&Program(ENTRY:func10)
    TraceTrigger
```

Results:



A The **Trace.STATistic.AddressDIStance** window shows that, over 1208 recorded trace samples, the timing constraint is violated once.

B The maximum recorded time distance between consecutive calls of `func10` is 60.86 µs.

C The watchpoint mark the occurrence of the TraceTrigger action

D The column “ti.back” shows that the distance between the last 2 calls of `func10` exceeds 35 µs for the first time.

Examples for CTL Trace Find

Use case 1: Checking Variable Access

User Story:

The user needs to find all write access to a variable `vlong` from outside the function `main`.

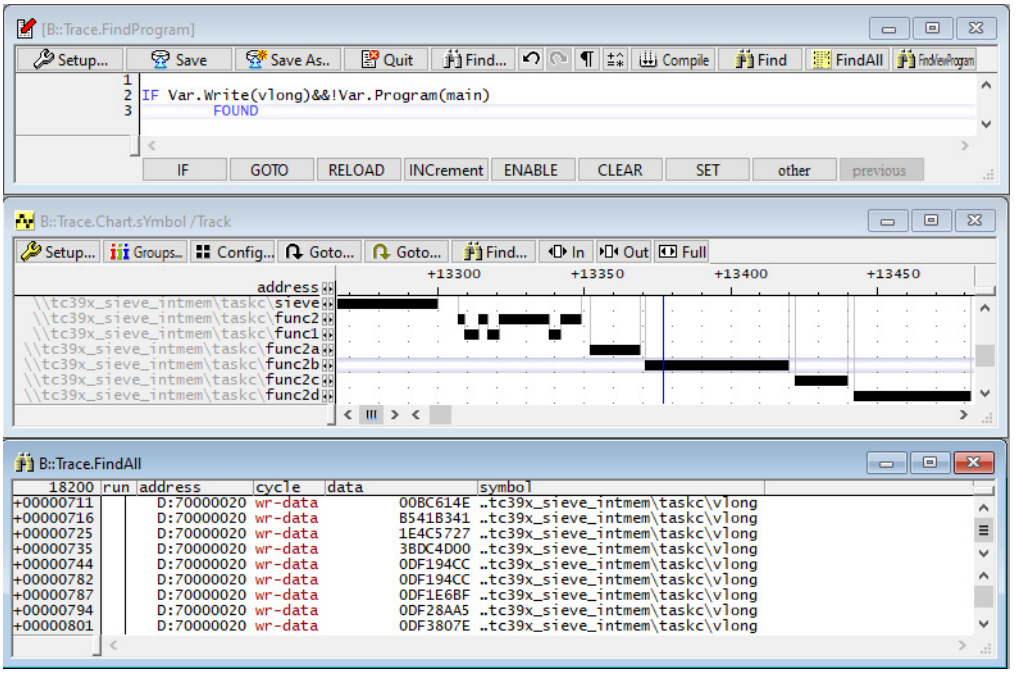
Trace Find Program: vlong_access.ct

```
IF Var.Write(vlong)&&!Var.Program(main)
   FOUND
```

Results:

The results could be explored using the TRACE32 PowerView:

1. Click on the compile button in the CTL Trace Find editor.
2. Open a trace chart window with the **/Track** option by executing the command **Trace.Chart.sYmbol /Track**
3. Navigate through the matching items using the button **Find**. Each time the trace chart window will be updated; the function that is performing the access is highlighted. Alternatively, the **FindAll** button could be used to list all the matching items in the **Trace.FindAll** window.



A scripting approach could also be used. The following PRACTICE script prints the names of matching items' functions to the message area window.

```
// activate the CTL find program from the file vlong_access.ct
Trace.FindReProgram ~~~~/vlong_access.ct

// search for the first trace record fulfilling the CTL program search
// criteria
Trace.Find

WHILE FOUND()
(
   // print the name of the matching item's function to the area window
   PRINT sYmbol.FUNCTION(TRACK.ADDRESS.PROG())

   // search for the next trace record fulfilling the CTL program search
   // criteria
   Trace.Find
)
```

Use case 2: Checking Timing Constraints - Address Duration

User Story:

The user needs to perform post-mortem analysis of a trace recording. The trace is loaded to a TRACE32 Instruction Set Simulator. The user needs to verify that the execution time of the function `mainloop` does not exceed a specified time of 35 µs.

Trace Find Program: check_address_duration.ct

```
start_level:
    // Detected Entry of the main loop
    IF Program(ENTRY:mainloop)
        GOTO check_level

check_level:
    // Reset the task_timer
    IF STATE.ENTER()
        RELOAD task_timer

    // Enable the task timer as long as check_level is active
    IF TRUE()
        ENABLE task_timer

    // Go back to start_level at Return from the function
    // mainloop
    IF Program(RETURN:mainloop)
        GOTO start_level

    // task_timer exceeds 35 µs
    // => a timout is detected
    IF TIME(task_timer>=35.us)
        GOTO timout_level

timout_level:
    // Search for the next return from mainloop and
    // mark it as FOUND and go back to start_level to
    // continue the test
    IF Program(RETURN:mainloop)
        FOUND
        GOTO start_level
```

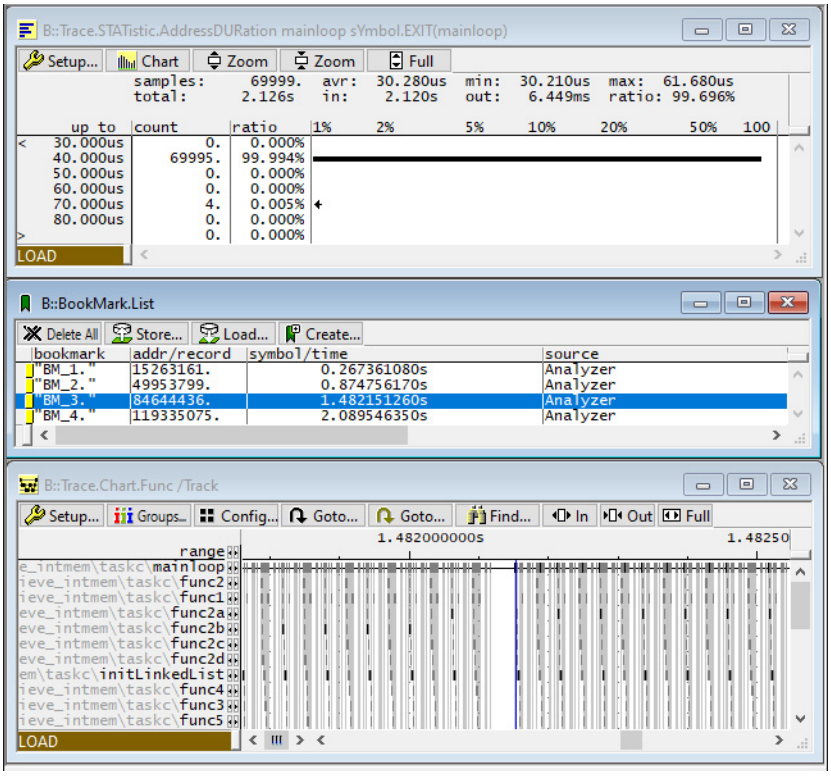
A PRACTICE script could be used to set bookmarks for all the trace records breaking the timing constraint.

Script:

```
PRIVATE &index
Trace.FindReProgram ~~~~/check_address_duration.ct
BookMark.RESet

&index=1.

// Find the first trace record matching the CTL find program criteria
Trace.Find
WHILE FOUND()
(
   // Compose a unique bookmark name
   &bookmark="BM_"+"&index"
   // Set a bookmark for the FOUND trace record
   Trace.Bookmark "&bookmark" TRACK.RECORD()
   &index=&index+1.
   // Find the next trace record matching the CTL find program criteria
   Trace.Find
)
```

Results:

In this example, the timing constraint was broken four times. These are identified by the bookmarks “BM_1”, “BM_2”, “BM_3”, and “BM_4”.



Use case 3: Checking Timing Constraints - Address Distance

User Story:

Runtime analysis of the user application program flow using **Trace.STATistic.AddressDIStance** shows that a function `classifyAbs` is called at least once each 1 ms. In some cases (3 times/4095 samples) the time distance between consecutive calls of `classifyAbs` exceeds 6 ms. A trace find program can be used to locate the trace samples where the timing constraint gets violated.

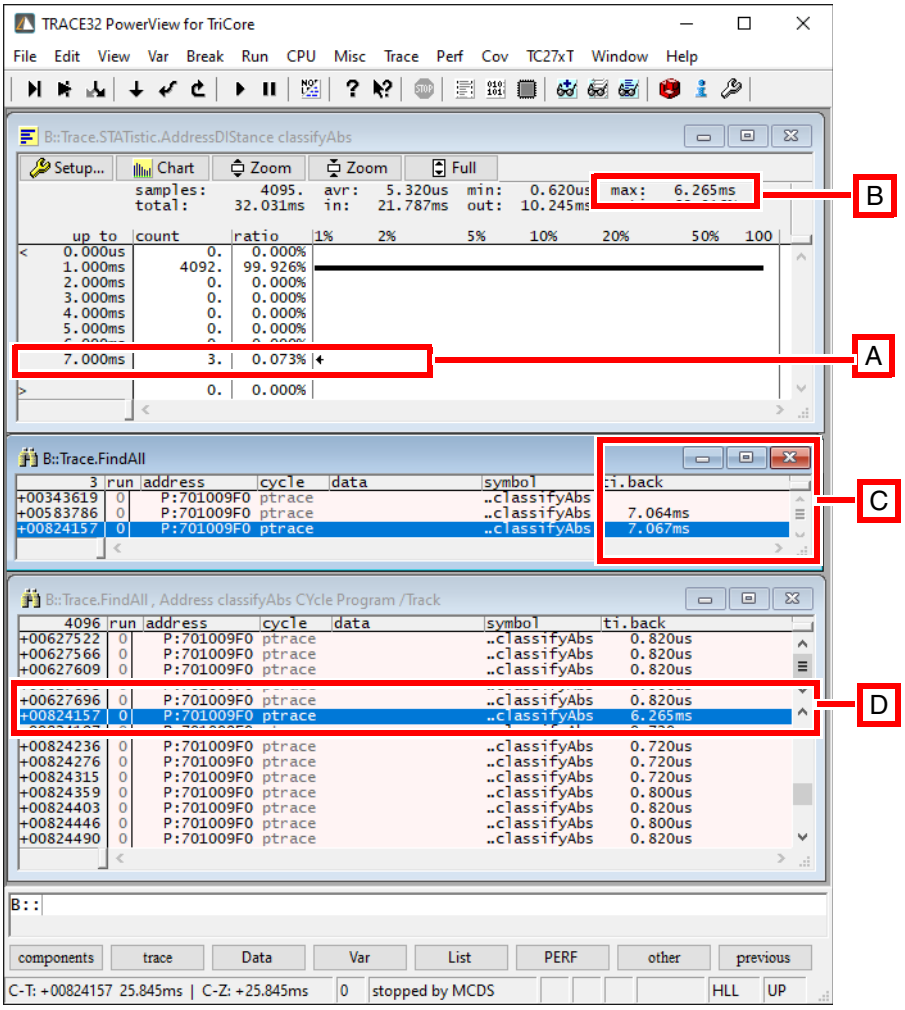
Trace Find Program: check_address_distance.ct

```
// For each trace sample corresponding to the entry of the function
// classifyAbs:
//   + Set a flag "monitoring_flag"
//   + Reload the distance monitoring timer "distance_tmr"
IF Program(ENTRY:classifyAbs)
    SET monitoring_flag
    RELOAD distance_tmr

// As long as the monitoring flag is set, keep incrementing
// the distance monitoring timer
IF FLAG(monitoring_flag)
    ENABLE distance_tmr

// In case the monitoring timer exceeds a limit of 1 ms
//   + The timout flag is set
//   + The monitoring flag is cleared
//   + The distance timer is reset
IF Time(distance_tmr>1.0ms)
    SET timeout_flag
    CLEAR monitoring_flag
    RELOAD distance_tmr

// The next trace sample representing the entry of the function
// classifyAbs with the timeout flag set is added to the trace find
// results of interest (FOUND)
// The timeout flag is cleared for processing the rest of the trace
// samples
IF Program(ENTRY:classifyAbs)&&FLAG(timeout_flag)
    FOUND
    CLEAR timeout_flag
```

The trace find program could compiled/activated via the command **Trace.FindReProgram** . Trace samples of interest can be listed using the command **Trace.FindAll** e.g. as follows:

```
// Activate the trace find program
Trace.FindReProgram ~~~~/check_address_distance.ct
// Display the trace find results
Trace.FindAll
```

Results:



A The **Trace.STATistic.AddressDIStance** window shows that, over 4095 recorded trace samples, the timing constraint is violated 3 times.

B The maximum recorded time distance between consecutive calls of `classifyAbs` is 6.265 ms.

C The time displayed in ti.back column of **Trace.FindAll** window is not corresponding to the time distance between 2 consecutive calls of `classifyAbs`. Actually, this represents the time distance between 2 samples of `classifyAbs` violating the timing constraint.

D The window Trace.FindAll, Address classifyAbs CYcle Program **/Track** is to be used to examine the maximum time distance between 2 consecutive calls of `classifyAbs`

BREAKPOINT — ABCDE breakpoint

Format:	**BREAKPOINT (***<type>***)**
<type>:	**Alpha** | **Beta** | **Charly** | **Echo**

BusTrigger — Incoming trigger signal

Format:	**BusTrigger (***<channel>***)**

BMC — Benchmark counter event

Format:	**BMC (***<event>***)**

This condition is only supported if the target processor provides an Embedded Trace Macrocell (ETM) and BenchMark Counters. Refer to the corresponding **"Processor Architecture Manuals"**.

Example:

```
IF BMC(dcachemiss)
        INCrement counter1
```

COUNT — Trigger on event counter

Format:	**COUNT (***<name/count>***)**

CLOCKS — Trigger on clock cycles counter

Format:	**CLOCKS (*<name/count>*)**

Example:

Enable program trace for 100 clock cycles starting from the execution of the first instruction of the `sieve` function.

```
start:
   IF Program(ENTRY:sieve)
           GOTO level1
level1:
   // Reset the clock counter at state change to level1
   IF STATE.ENTER()
           RELOAD clock_counter

   // Enable Program trace and clock_counter as long as
   // level1 is active
   IF TRUE()
        TraceEnable Program
        ENABLE clock_counter

   // Stop tracing when the clock_counter reach the limit of 100 cycles
   IF CLOCKS(clock_counter>=100.)
        TraceTrigger
```

CTM — Cross trigger

Format:	**CTM (*<channel>*)**

This condition is only supported if the target processor provides an Embedded Trace Macrocell (ETM) and a CoreSight Trigger Matrix (CTM).

EXTIN — External input

Format:	**EXTIN (*<channel>*)**

This condition is only supported if the target processor provides an Embedded Trace Macrocell (ETM).

FALSE — Never condition

Format:	**FALSE ()**

FLAG — Flag status

Format:	**FLAG (***<name/value>***)**

Example:

Enable program trace for the address range of the function `sieve` if the variable `mstatic1` has a value of 2:

```
IF Var.Write(mstatic1==2)
        SET myflag

IF Var.Write(mstatic1!=2)
        CLEAR myflag

IF Var.Program(sieve)&&FLAG(myflag)
        TraceEnable Program
```

MACHINE — Machine comparator

Format:	**MACHINE (***<machine>***)**

This condition is only supported if the target processor provides an Embedded Trace Macrocell (ETM) and hypervisor extensions.

Program — Program access comparator

Format:	**Program (***<item>***)**
<item>:	[*<logical_operator>*] [**ENTRY:** | **RETURN:** | **RANGE:**] *<addr/data>*
<logical_operator>:	**~** | **==** | **!=** | **<** | **<=** | **>=** | **>**

ENTRY:	Address of the function entry point.
RETURN:	Address of the function exit (function epilogue)
RANGE:	Function address range

If nothing is specified in front of a function name, ENTRY is default.

Example:

```
IF Program(ENTRY:sieve)
        TraceON Program

IF Program(RETURN:sieve)
        TraceOFF Program
```

ProgramFail — Conditional instruction execution

Format:	**ProgramFail (***<item>***)**
<item>:	[*<logical_operator>*] [**ENTRY:** | **RETURN:** | **RANGE:**] *<addr/data>*
<logical_operator>:	**~** | **==** | **!=** | **<** | **<=** | **>=** | **>**

ProgramPass — Conditional instruction execution

Format:	**ProgramPass (***<item>***)**
<item>:	[*<logical_operator>*] [**ENTRY:** | **RETURN:** | **RANGE:**] *<addr/data>*
<logical_operator>:	**~** | **==** | **!=** | **<** | **<=** | **>=** | **>**

Read — Read access

Format:	**Read (***<item>***)**
<item>:	[*<logical_operator>*] *<addr/data>*
<logical_operator>:	**~** | **==** | **!=** | **<** | **<=** | **>=** | **>**

Example:

```
IF Read(flags)
        TraceEnable ALL
```

Format:	**ReadWrite (***<item>***)**
<item>:	[*<logical_operator>*] *<addr/data>*
<logical_operator>:	**~ | == | != | < | <= | >= | >**

Example 1:

```
IF ReadWrite(mstatic1)
        TraceData DEFault
```

Example 2:

```
IF ReadWrite(vint==1000)
        SET myFlag

IF FLAG(myFlag)&&Program(ENTRY:sieve)
        Break
```

SingleShot — Single shot comparators

The SingleShot conditions are only supported if the target processor provides an Embedded Trace Macrocell with a single shot comparator (e.g. ETMv4).

SingleShot.Program — Single shot program execution

Format:	**SingleShot.Program (***<item>***)**
<item>:	[*<logical_operator>*] [**ENTRY:** | **RETURN:** | **RANGE:**] *<addr/data>*
<logical_operator>:	**~** | **==** | **!=** | **<** | **<=** | **>=** | **>**

Example:

```
IF SingleShot.Program(ENTRY:sieve)
        TraceEnable Program
```

SingleShot.ProgramFail — Single shot conditional execution

Format:	**SingleShot.ProgramFail (***<item>***)**
<item>:	[*<logical_operator>*] [**ENTRY:** | **RETURN:** | **RANGE:**] *<addr/data>*
<logical_operator>:	**~** | **==** | **!=** | **<** | **<=** | **>=** | **>**

SingleShot.ProgramPass — Single shot conditional execution

Format:	**SingleShot.ProgramPass (***<item>***)**
<item>:	[*<logical_operator>*] [**ENTRY:** | **RETURN:** | **RANGE:**] *<addr/data>*
<logical_operator>:	**~** | **==** | **!=** | **<** | **<=** | **>=** | **>**

SingleShot.Read — Single shot read access

Format:	**SingleShot.Read (***<item>***)**
<item>:	[*<logical_operator>*] *<addr/data>*
<logical_operator>:	**~** | **==** | **!=** | **<** | **<=** | **>=** | **>**

Example:

```
IF SingleShot.Read(mstatic1)
        TraceData Read Address Data
```

SingleShot.ReadWrite — Single shot read or write access

Format:	**SingleShot.ReadWrite (***<item>***)**
<item>:	[*<logical_operator>*] *<addr/data>*
<logical_operator>:	**~ | == | != | < | <= | >= | >**

Example:

```
IF SingleShot.ReadWrite(mstatic1)
        TraceData DEFault
```

SingleShot.Write — Single shot write access

Format:	**SingleShot.Write (***<item>***)**
<item>:	[*<logical_operator>*] *<addr/data>*
<logical_operator>:	**~ | == | != | < | <= | >= | >**

Example:

```
IF SingleShot.Write(flags)
        TraceEnable ALL
```

NoSingleShot — Non single shot comparators

The NoSingleShot conditions are only supported if the target processor provides an Embedded Trace Macrocell with a single shot comparator (e.g. ETMv4).

NoSingleShot.Program — Non single shot program execution

Format:	**NoSingleShot.Program (***<item>***)**
<item>:	[*<logical_operator>*] [**ENTRY:** | **RETURN:** | **RANGE:**] *<addr/data>*
<logical_operator>:	**~** | **==** | **!=** | **<** | **<=** | **>=** | **>**

Example:

```
IF NoSingleShot.Program(ENTRY:sieve)
        TraceEnable Program
```

NoSingleShot.ProgramFail — Non single shot conditional execution

Format:	**NoSingleShot.ProgramFail (***<item>***)**
<item>:	[*<logical_operator>*] [**ENTRY:** | **RETURN:** | **RANGE:**] *<addr/data>*
<logical_operator>:	**~** | **==** | **!=** | **<** | **<=** | **>=** | **>**

NoSingleShot.ProgramPass — Non single shot conditional execution

Format:	**NoSingleShot.ProgramPass (***<item>***)**
<item>:	[*<logical_operator>*] [**ENTRY:** | **RETURN:** | **RANGE:**] *<addr/data>*
<logical_operator>:	**~** | **==** | **!=** | **<** | **<=** | **>=** | **>**

NoSingleShot.Read — Non single shot read access

Format:	**NoSingleShot.Read (***<item>***)**
<item>:	[*<logical_operator>*] *<addr/data>*
<logical_operator>:	**~** | **==** | **!=** | **<** | **<=** | **>=** | **>**

NoSingleShot.ReadWrite — Non single shot read or write access

Format:	**NoSingleShot.ReadWrite (***<item>***)**
<item>:	[*<logical_operator>*] *<addr/data>*
<logical_operator>:	**~** | **==** | **!=** | **<** | **<=** | **>=** | **>**

Format:	**NoSingleShot.Write (***<item>***)**
<item>:	[*<logical_operator>*] *<addr/data>*
<logical_operator>:	**~** | **==** | **!=** | **<** | **<=** | **>=** | **>**

STATE.LEAVE — Leave the state transition (edge sensitive)

Format:	**STATE.LEAVE ()**

Example:

```
start:
   IF Program(ENTRY:sieve)
           GOTO level1

   IF STATE.LEAVE()
           RELOAD blink_counter

level1:
   IF Program(RETURN:sieve)
           GOTO start

   IF Program(ENTRY:Blink)
           INCrement blink_counter

   IF COUNT(blink_counter>2)
           Break
```

STATE.ENTER — Enter the state transition (edge sensitive)

Format:	**STATE.ENTER ()**

Example:

```
start:
   IF Program(ENTRY:sieve)
           GOTO level1

level1:
   IF STATE.ENTER()
           RELOAD blink_counter

   IF Program(RETURN:sieve)
           GOTO start

   IF Program(ENTRY:Blink)
           INCrement blink_counter

   IF COUNT(blink_counter>2)
           Break
```

STATE.TRACEON — Active state of a TraceON action

Format:	**STATE.TRACEON ()**

This condition is only supported if the target processor provides an Embedded Trace Macrocell (ETM) versions older than ETMv4.

TASK — Task comparator

Format:	**TASK (***<task>***)**

This condition is only supported if the target processor provides an Embedded Trace Macrocell (ETM).

TIME — Time counter comparator

Format:	**TIME (***<name/time>***)**

Example:

```
start:
   IF Program(ENTRY:sieve)
           GOTO level1

level1:
   IF ENTRY()
           RELOAD sievetimer
           TraceON Program

   IF TRUE()
           ENABLE sievetimer

   IF Program(RETURN:sieve)
           TraceOFF Program
           GOTO start

   IF TIME(sievetimer>200.us)
           Break
```

Format:	**TRUE ()**

Example:

```
start:
   IF Program(ENTRY:sieve)
           GOTO level1
level1:

   IF TRUE()
        TraceEnable Program


   IF Program(RETURN:sieve)
        GOTO start
```

Var prefix allows to specify the HLL expression in the syntax of the used programming language (e.g. C, C++).

Then condition will consider the full function/variable range.

Var.Program — Flat function execution

Format:	**Var.Program (***<item>***)**
<item>:	[*<logical_operator>*] [**ENTRY:** | **RETURN:** | **RANGE:**] *<var/data>*
<logical_operator>:	**~** | **==** | **!=** | **<** | **<=** | **>=** | **>**

Example:

```
// Trace all write access when executing instructions in the sieve
// function address range
IF Var.Program(sieve)
        TraceData Write Address Data
```

Var.Read — Variable read access

Format:	**Var.Read (***<item>***)**
<item>:	[*<logical_operator>*] *<var/data>*
<logical_operator>:	**~** | **==** | **!=** | **<** | **<=** | **>=** | **>**

Var.ReadWrite — Variable read or write access

Format:	**Var.ReadWrite (***<item>***)**
<item>:	[*<logical_operator>*] *<var/data>*
<logical_operator>:	**~** | **==** | **!=** | **<** | **<=** | **>=** | **>**

Var.status — tbd.

Format:	**Var.status (***<item>***)**
<item>:	[*<logical_operator>*] *<var>*
<logical_operator>:	**~** | **==** | **!=** | **<** | **<=** | **>=** | **>**

Var.Write — Variable write access

Format:	**Var.Write (***<item>***)**
<item>:	[*<logical_operator>*] *<var/data>*
<logical_operator>:	**~** | **==** | **!=** | **<** | **<=** | **>=** | **>**

Example:

```
IF Var.Write(vint==1000)
        SET myflag
```

Write — Write access

Format:	**Write (***<item>***)**
<item>:	[*<logical_operator>*] *<addr/data>*
<logical_operator>:	**~** | **==** | **!=** | **<** | **<=** | **>=** | **>**

Example:

```
IF Write(0x70001000!=0x55)
         TraceEnable ALL
```

ZONE — Zone comparator

Format:	**ZONE (***<zone>***)**

This condition is only supported if target processor provides an Embedded Trace Macrocell (ETM).

tbd.**if** ((EMU_ArmHypervisor || EMU_ArmSecure || EMU_EtmV4)) && ! EMU_EtmWithoutProgramAddressComparators).

Break

Stop the program execution

Format: **Break**

Example:

```
// Stop if any write access to mstatic1 is generated by an instruction
// from outside the sieve function range

IF var.Write(mstatic1)&&!Var.Program(sieve)
        Break
```

BusCLOCKS

tbd.

Format: **BusCLOCKS** *<counter>*

BusCount

tbd.

Format: **BusCount** *<counter>*

BusTIME

tbd.

Format: **BusTIME** *<counter>*

BusTrigger — tbd.

Format:	**BusTrigger** [**DEFault** | **0** | **1**]

This action is only supported if the target processor provides an Embedded Trace Macrocell (ETM) and a CoreSight Trigger Matrix (CTM).

DEFault	tbd.
0	tbd.
1	tbd.

CLEAR — Clear flag

Format:	**CLEAR** *<flag>*

Example:

```
IF Program(ENTRY:sieve)
        SET flag_sieve

IF Program(RETURN:sieve)
        CLEAR flag_sieve

IF Var.Write(mstatic1)&&!FLAG(flag_sieve)
        TRACEENABLE Program Write Address Data
```

CTM — Cross trigger

Format:	**CTM** [**0** | **1** | **2** | **3**]

This condition is only supported if the target processor provides an Embedded Trace Macrocell (ETM) and a CoreSight Trigger Matrix (CTM).

0	tbd.
1	tbd.
2	tbd.
3	tbd.

ENABLE — Enable counter

Format: **ENABLE** *<counter>* | *<timer>*

Enable counting while the condition is verified.

Example:

```
IF Program(ENTRY:sieve)
        SET flag_sieve

IF Program(RETURN:sieve)
        CLEAR flag_sieve

IF FLAG(flag_sieve)
        ENABLE timer_sieve
        TRACEENABLE Program

IF TIME(timer_sieve>10.us)
        TraceTrigger
```

EVENT — Trace event

Format: **EVENT** [**0** | **1** | **2** | **3**]

EXTOUT External output

Format:	**EXTOUT** [**0** | **1** | **2** | **3**]

0	tbd.
1	tbd.
2	tbd.
3	tbd.

FOUND Add the trace sample to the search items result

Format:	**FOUND**

Example:

```
IF Program(ENTRY:sieve)
       FOUND
```

GOTO Change active state

Format:	**GOTO** *<state>*

Example:

```
start:
       IF Program(ENTRY:sieve)
               GOTO level1

level1:
       IF Program(RETURN:sieve)
               GOTO start
       IF Var.Write(mstatic1)
               TraceData Write Address Data
```

INCrement — Increment counter

Format	**INCrement** *<counter>*

Example:

```
IF Program(ENTRY:sieve)
        INCrement sieve_cnt
        TraceEnable Program

IF COUNT(sieve_cnt==10.)
        Break
```

RELOAD — Reload counter

Format	**RELOAD** *<counter>*

Example:

```
start:
       IF Program(ENTRY:sieve)
               GOTO level1

level1:
       IF Program(RETURN:sieve)
               GOTO start
       IF Program(ENTRY:Blink)
               INCrement blink_counter
       IF STATE.ENTER()
               RELOAD blink_counter
       IF COUNT(blink_counter>2)
               Break
```

SET — Set flag

Format	**SET** *<flag>*

Example:

```
IF Var.Write(mstatic1)
        Set myflag

IF Program(ENTRY:sieve)&&FLAG(myflag)
        Break
```

Spot — Shortly stop the program execution

Format	**Spot**

Example:

```
IF Program(ENTRY:sieve)
        Spot
```

Format	**TraceData** [**DEFault** | **Read** | **Write** | **ReadWrite** | **Address** | **Data**]

DEFault	Is equivalent to ReadWrite Address Data.
Read	Sample read cycles.
Write	Sample write cycles.
ReadWrite	Sample read and write cycles.
Address	Sample cycle address.
Data	Sample cycle data.

NOTE:	**TraceData** [**Read** | **Write** | **ReadWrite**] without specifying Address or Data will sample cycle address and data.

Example:

```
// Trace all the write cycles that are performed by the instructions
// in the address range of the function sieve
IF Var.Program(sieve)
        TraceData Write Address Data
```

By using the action TraceData, the status of unconditional program trace is not changed. E.g. if unconditional program trace is enabled, the resulting trace recording will contain unconditional program trace additionally to the selective data trace for the specified events.

Format	**TraceEnable** *<parameter>*
<parameter>:	[**DEFault** | **ALL** | **Program** | **Read** | **Write** | **ReadWrite** | **Address** | **Data**]

DEFault	Sample the event depending on the condition. e.g. if the condition is a program condition the program cycles are sampled.
ALL	Sample program, read, and write cycles. For the read and write cycles, the sample address and data are included.
Program	Sample program cycles.
Read	Sample read cycles.
Write	Sample write cycles.
ReadWrite	Sample read and write cycles.
Address	Sample cycle address.
Data	Sample cycle data.

NOTE:	**TraceEnable** [**Read** | **Write** | **ReadWrite**] without specifying Address or Data will sample cycle address and data.

Example 1:

```
// Trace program and all the write cycles that are performed by the
// instructions in the address range of the function sieve
IF Var.Program(sieve)
        TraceEnable Program Write Address Data
```

Example 2:

```
// Trace all the write cycles to mstatic1

IF Var.Write(mstatic1)
        TraceEnable DEFault
```

Format	**TraceOFF** *<parameter>*
<parameter>:	[**DEFault** | **ALL** | **Program** | **Read** | **Write** | **ReadWrite** | **Address** | **Data**]

DEFault	tbd.
ALL	Switch off sampling program, read, and write cycles.
Program	Switch off sampling program cycles.
Read	Switch off sampling read cycles.
Write	Switch off sampling write cycles.
ReadWrite	Switch off sampling read and write cycles.
Address	Switch off sampling cycle address.
Data	Switch off sampling cycle data.

TraceON — Switch ON the trace sampling

Format	**TraceON** *<parameter>*
<parameter>:	[**DEFault** | **ALL** | **Program** | **Read** | **Write** | **ReadWrite** | **Address** | **Data**]

DEFault	tbd.
ALL	Switch on sampling program, read, and write cycles.
Program	Switch on sampling program cycles.
Read	Switch on sampling read cycles.
Write	Switch on sampling write cycles.
ReadWrite	Switch on sampling read and write cycles.
Address	Switch on sampling cycle address.
Data	Switch on sampling cycle data.

TraceTIME — tbd.

Format	**TraceTIME**

TraceTrigger — Stop sampling to the trace buffer on specified event

Format	**TraceTrigger** *<cycles>* | *<percent>*

A trigger delay could be specified in number of cycles or percentage of the trace buffer size.

CTL Programming Errors