## TRACE32 Online Help

## TRACE32 Directory

## TRACE32 Index

### TRACE32 Training

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training Real-time Trace</td>
<td>1</td>
</tr>
<tr>
<td>Trace Methods</td>
<td>5</td>
</tr>
<tr>
<td>Trace Method Analyzer</td>
<td>5</td>
</tr>
<tr>
<td>Other Trace Methods</td>
<td>6</td>
</tr>
<tr>
<td>Configuration (Analyzer)</td>
<td>8</td>
</tr>
<tr>
<td>Configuration TRACE32-ICD Standard</td>
<td>8</td>
</tr>
<tr>
<td>Configuration TRACE32-ICD PowerTrace</td>
<td>9</td>
</tr>
<tr>
<td>Trace Technologies</td>
<td>10</td>
</tr>
<tr>
<td>Bus Trace</td>
<td>10</td>
</tr>
<tr>
<td>Flow Trace</td>
<td>11</td>
</tr>
<tr>
<td>Program Flow Trace</td>
<td>11</td>
</tr>
<tr>
<td>Data Flow Trace</td>
<td>16</td>
</tr>
<tr>
<td>Multiplexed Recording</td>
<td>17</td>
</tr>
<tr>
<td>NEXUS Trace</td>
<td>18</td>
</tr>
<tr>
<td>Branch Trace Messages (BTM)</td>
<td>18</td>
</tr>
<tr>
<td>Data Trace Messages (DTM)</td>
<td>20</td>
</tr>
<tr>
<td>Display the Trace Contents</td>
<td>22</td>
</tr>
<tr>
<td>States of the Trace</td>
<td>24</td>
</tr>
<tr>
<td>Sampling Modes of the Trace</td>
<td>25</td>
</tr>
<tr>
<td>Trace Contents</td>
<td>27</td>
</tr>
<tr>
<td>Trace Configuration Window</td>
<td>31</td>
</tr>
<tr>
<td>Format the Trace Buffer</td>
<td>34</td>
</tr>
<tr>
<td>Correlate the Trace Listing with the Source</td>
<td>35</td>
</tr>
<tr>
<td>Browse through the Trace Buffer</td>
<td>36</td>
</tr>
<tr>
<td>Find a Specific Record</td>
<td>37</td>
</tr>
<tr>
<td>Context Tracking System (CTS)</td>
<td>38</td>
</tr>
<tr>
<td>Trace Based Debugging</td>
<td>39</td>
</tr>
<tr>
<td>HLL Analysis of the Trace Contents</td>
<td>42</td>
</tr>
<tr>
<td>Background</td>
<td>45</td>
</tr>
<tr>
<td>Cache Analysis (PowerTrace)</td>
<td>47</td>
</tr>
</tbody>
</table>

©1989-2013 Lauterbach GmbH
Trace Methods

Trace Method Analyzer

Trace method Analyzer means, that a physical trace memory is provided by the development tool. The following TRACE32 development tools provide a physical trace memory:

- RISC TRACE module
  - or
- POWERTRACE / ETHERNET

<table>
<thead>
<tr>
<th>Implementation of the trace memory</th>
<th>64 K up to 16 M frames physical trace memory is available, 92 bits for each frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max. trace size</td>
<td>16M frames (with POWERTRACE / ETHERNET).</td>
</tr>
<tr>
<td>Sampling</td>
<td>While the user program is running the program (and data) flow is sampled into the physical trace memory.</td>
</tr>
<tr>
<td>Influence on the real-time behavior</td>
<td>No influence on the real-time behavior for nearly all supported CPUs.</td>
</tr>
<tr>
<td>Selective tracing</td>
<td>Possible for most CPUs.</td>
</tr>
<tr>
<td>Frequency</td>
<td>Depends on the CPU and the trace technology.</td>
</tr>
</tbody>
</table>

©1989-2013 Lauterbach GmbH
## Other Trace Methods

<table>
<thead>
<tr>
<th>Trace Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ART</td>
<td>Provides a history for single stepping. For each step the contents of the general purpose registers is recorded. No physical trace memory required! TRACE32 reserve memory on the host for ART. Available for all CPUs.</td>
</tr>
<tr>
<td>LOGGER</td>
<td>A part of the target RAM is used as trace memory. The user has to take care, that his application writes the trace information in the correct format into the target RAM. TRACE32 provides all commands for trace display and evaluation. No physical trace memory required! Available for all CPUs. Examples for the usage of the trace method Logger can be found in the demo directory.</td>
</tr>
<tr>
<td>SNOOPer</td>
<td>The CPU has to support memory read while the program execution is running or the on-chip debugging interface has to provide a communication channel that allows the application to communicate with the debugger while the program execution is running (e.g. Debug Communication Channel for the ARM architecture) TRACE32 can read up to 16 data addresses in a fixed sampling rate and record their contents to the trace. No physical trace memory required! TRACE32 reserve memory on the host for the SNOOPer.</td>
</tr>
<tr>
<td>FDX</td>
<td>The CPU has to support memory read while the program execution is running or the on-chip debugging interface has to provide a communication channel that allows the application to communicate with the debugger while the program execution is running (e.g. Debug Communication Channel for the ARM architecture) A small part of the target RAM is used as a ring buffer. TRACE32 collects the contents of the ring buffer permanently while the program is running and builds the trace contents based on this information. No physical trace memory required! TRACE32 reserve memory on the host for FDX. Examples for the usage of the trace method FDX can be found in the demo directory.</td>
</tr>
<tr>
<td>Onchip</td>
<td>The CPU provides an on-chip trace e.g. XScale.</td>
</tr>
<tr>
<td>Integrator</td>
<td>A TRACE32-PowerIntegrator is set-up to work as a program or/and data flow trace. For more information refer to the powerintegrator_app*.pdf documentation in the PDF directory.</td>
</tr>
</tbody>
</table>

©1989-2013 Lauterbach GmbH
<table>
<thead>
<tr>
<th><strong>Probe</strong></th>
<th>A TRACE32-PowerProbe is set-up to work as a program or/and data flow trace.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LA</strong></td>
<td>The contents of an external trace device (e.g. from a logic analyzer) was imported to TRACE32.</td>
</tr>
</tbody>
</table>
The trace extension for TRACE32-ICD is realized as a universal RISC TRACE module plus a CPU specific preprocessor. It supports:

- 92 trace channels up to 100 MHz
- 64 K/ 128 K/ 256 K/ 512 K frames
- 36 bit time stamp with 25 ns resolution

Depending on the CPU the trace extension is realized either as a

- Bus trace
- Clock trace/program and data flow trace
The trace part of the POWERTRACE / ETHERNET is universal and it is connected to the target via a CPU specific preprocessor. It supports:

- 92 trace channels up to 200 MHz
- 16 MFrames trace depth
- 32 bit time stamp with 20 ns resolution

Depending on the CPU the trace extension is realized either as a

- Bus trace
- Clock trace/program and data flow trace
- NEXUS trace
Bus Trace

For a bus trace the address and data bus and certain state lines are recorded for every CPU cycle. As a result the complete information about the program and data flow is available.

Available Bus Traces

<table>
<thead>
<tr>
<th>Trace Control Features</th>
<th>Context Tracking System</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM7 ARM9 with AMBA</td>
<td>1 address selector implemeneted as a comparator (bit mask). Instruction Set Simulator (full CTS)</td>
</tr>
</tbody>
</table>
| C167 Family           | Several address selectors within a 1 MB range
                        | Several start trace points and several stop trace points within a 1 MB range. Instruction Set Simulator (full CTS) |
| C166CBC               | Several address selectors within a 1 MB range
                        | Several start trace points and several stop trace points within a 1MB range. Instruction Set Simulator (full CTS) |
| MPC8260/MPC8240 PPC603| 1 start trace point and 1stop trace point. Instruction Set Simulator (full CTS) |
In order to achieve higher speeds the programs run from cache or the internal RAM/FLASH. External visibility would reduce the processor performance.

Program Flow Tracking allows to track the program instruction flow with (almost) no performance degradation.

In order to reconstruct the program flow, the code image from memory and the following additional information is required:

- Description about the program flow (pipeline information)
- Branch destination address of indirect branches
How does a program flow work?

1. **Stall or sequential**

   - **start address**
   - One pin of the processor indicates per clock cycle:
     - next instruction executed or not
   - **end address**

   An intelligent software is able to reconstruct the exact program flow by using the start address, the program flow information (here next instruction taken or not), the end address and the code image from memory.

2. **Direct branches**

   - **start address**
   - Two pins of the processor indicate per clock cycle:
     - next instruction executed or not
     - direct branch taken/direct branch not taken
   - **end address**

   An intelligent software is able to reconstruct the exact program flow by using the start address, the program flow information (here next instruction taken or not, direct branch taken/branch not taken), the end address and the code image from memory.

3. **Indirect branches**

   - **start address**
   - Three pins of the processor indicate per clock cycle:
     - next instruction executed or not
     - direct branch taken/branch not taken
     - indirect branch taken + branch destination address
   - **end address**

To reconstruct the program flow when an indirect branch is taken (branch using a register, all interrupts, all return from interrupt etc.) the software needs information about the branch destination address of the indirect branch.
How is the branch destination address shown?

1. **Show cycles (e.g. PowerPC500/800)**

The CPU generates show cycles. That means, the fetch cycles that results from the indirect change of flow are visible on the external bus. The generation of show cycle reduces the performance of the CPU about 1%.

2. **Sequential address output (e.g. TRICORE)**

The branch destination address from an indirect branch is visible in the next 1-4 clock cycles on special pins of the CPU (Trace port). The number of cycles depends if it is a short jump or a long jump.
In order to reconstruct the complete program flow out of the information sampled to the trace buffer, it is necessary to read the code image from memory.

Not all CPUs support memory read while the program is executed. That means the program execution has to be stopped in order to display the trace information or the code has to be loaded to the virtual memory.

The virtual memory is located on the host and TRACE32-ICD is using the virtual memory for the source image information, when no access to the on-chip/target memory is possible while the processor is executing the program.

If the sampling to the trace buffer is stopped (OFF state) while the CPU is executing the program NOACCESS indicates that TRACE32 has no access to the on-chip/target memory. Due to this the program flow can’t be reconstructed and displayed.
If the sampling to the trace buffer is stopped (OFF state) while the CPU is executing the program TRACE32 reads the code image from the virtual memory in order to reconstruct the program flow, if no access to the on-chip/target memory is possible.
Some architecture provide also additional information on the data flow:

- ARM-ETM
- ColdFire (no data address information)
- PowerPC500/800
- SH4 (SH 7751 only)
Most CPUs provide the program and data flow information via dedicated CPU pins (trace port). The only exception is the MPC500/800 family. Since each frame in the trace buffer is 92 bit wide and since most trace ports are small (8 to 16 pins), TRACE32 records the information of up to 4 clock cycles into 1 trace frame.

Depending on the number of recorded cycles per trace frame the size of the trace buffer increases.

<table>
<thead>
<tr>
<th>Method for Branch Destination Address</th>
<th>Number of CPU Cycles per Trace Frame</th>
<th>Maximum Trace Port Speed</th>
<th>Usable Trace Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM-ETM Sequential Address Output</td>
<td>4 clock cycles</td>
<td>270 MHz (full rate)</td>
<td>4 * physical trace size</td>
</tr>
<tr>
<td>TRICORE Sequential Address Output</td>
<td>4 clock cycles</td>
<td>180 MHz</td>
<td>4 * physical trace size</td>
</tr>
<tr>
<td>ColdFire Sequential Address Output</td>
<td>4 clock cycles</td>
<td>180 MHz</td>
<td>4 * physical trace size</td>
</tr>
<tr>
<td>PowerPC4xx Sequential Address Output</td>
<td>4 clock cycles</td>
<td>180 MHz</td>
<td>4 * physical trace size</td>
</tr>
<tr>
<td>PowerPC5xx/8xx Show Cycles</td>
<td>1 clock cycles</td>
<td>66 MHz bus clock</td>
<td>1 * physical trace size</td>
</tr>
<tr>
<td>PowerPC88x Show Cycles</td>
<td>1 clock cycles</td>
<td>80 MHz bus clock</td>
<td>1 * physical trace size</td>
</tr>
<tr>
<td>MIPS4k Sequential Address Output</td>
<td>4 clock cycles</td>
<td>180 MHz</td>
<td>4 * physical trace size</td>
</tr>
<tr>
<td>SH4 Sequential Address Output</td>
<td>4 clock cycles</td>
<td>200 MHz</td>
<td>4 * physical trace size</td>
</tr>
</tbody>
</table>
NEXUS was designed to provide an embedded development pin interface standard for:

- high performance microprocessors and
- highly integrated microprocessors (on-chip cache, on-chip FLASH and on-chip RAM)

Full internal visibility requires new methods to sample the program and data flow.

NEXUS provides the visibility of the program and data flow by outputting so-called trace messages.

**Branch Trace Messages (BTM)**

Branch trace messages provide a standard protocol for program flow visibility.

A branch trace message is issued whenever the program flow changes:

- Messages for **taken direct branches** include how many sequential instructions were executed since the last taken branch or exception. Not taken direct branches count as sequential instructions.
- Messages for **taken indirect branches** include the branch target (destination) address plus how many sequential instructions were executed since the last taken branch or exception. Not taken indirect branches count as sequential instructions. The branch target address is relative to prior address transmissions.
- Messages for **exceptions** include the exception vector address plus how many sequential instructions were executed since the last taken branch or exception.
- Under specific conditions **synchronization messages** with the full address information are issued (e.g. all 256 BTMs).
The complete program flow is reconstructed by software with reference to the code information.

Important information in regard to the time stamp:

1. Each NEXUS message consists of one or more records. A NEXUS message is time stamped when it is entered into the trace buffer after all records were received by TRACE32.

2. Beside the trace messages also other messages are output via the NEXUS Auxiliary Output:
   - Watchpoint messages
   - Real-time memory read messages
   - Error messages
   - The trace messages have the lowest priority.

Result: The time stamp within the trace buffer doesn't reflect exactly the time flow within the microprocessor.
Data Trace Messages provide visibility of read/write accesses to peripherals and memory locations.

It is recommended to limit the trace data to avoid Data Trace Overrun Errors.

The trace data can be limited to 2 address ranges.

**TraceEnable**: trace only the specified data accesses

**TraceData**: trace the specified data accesses plus the complete program flow
A Data Trace Message contains:

- The data address relative to prior address transmissions
- The data value

Under specific conditions **synchronization messages** with the full address information are issued (e.g. all 256 DTMs).
Display the Trace Contents

Trace.List

Display a trace listing

All features described in this section refer to Trace METHOD ANALYZER.
The recording to the trace buffer is controlled:

- On the tool side by the settings of the Trace Configuration Window.

- On the CPU side by the filter and trigger features provided by the on-chip trace hardware.
The trace buffer can either sample or read out for information display.

**States of the Trace**

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DISABLE</strong></td>
<td>The trace is disabled.</td>
</tr>
<tr>
<td><strong>OFF</strong></td>
<td>The trace is not sampling. The trace contents can be displayed.</td>
</tr>
<tr>
<td><strong>Arm</strong></td>
<td>The trace is sampling. The trace contents can not be displayed.</td>
</tr>
<tr>
<td><strong>trigger</strong></td>
<td>(not available for all CPUs)&lt;br&gt;The trace is sampling. The trace contents can not be displayed.&lt;br&gt;&lt;br&gt;A trigger was generated to stop the trace sampling, but the trace is still sampling because a trigger delay was defined (TDelay).</td>
</tr>
<tr>
<td><strong>break</strong></td>
<td>(not available for all CPUs)&lt;br&gt;The trace is not sampling. The trace contents can be displayed.&lt;br&gt;&lt;br&gt;- The trace sampling was stop by a trigger and no TDelay was defined.&lt;br&gt;- A trigger was generated to stop the trace sampling and the number of records defined by TDelay was sampled.</td>
</tr>
</tbody>
</table>
### Trace Modes

#### Fifo
The trace is working in FIFO mode. When the trace buffer is full, the new records will overwrite the older ones.

The trace contains always the last cycles before the program stopped.

#### Stack
The trace is working in STACK mode. As soon as the trace buffer is full, trace capturing will be stopped.

The trace contains always the first cycles after the program start.

#### Leash
The trace is working in LEASH Mode. As soon as the trace is nearly full, the program execution is stopped.
In most cases the trace mode can be derived from the record numbering scheme:

Trace is working in **FIFO** mode, negative record numbers are used.

Trace is working in **STACK/LEASH** mode, positive record numbers are used.
### Trace Contents

#### Record Number

<table>
<thead>
<tr>
<th>address</th>
<th>Address information</th>
</tr>
</thead>
<tbody>
<tr>
<td>cycle</td>
<td>Cycle type information</td>
</tr>
<tr>
<td>data</td>
<td>Data information</td>
</tr>
<tr>
<td>symbol</td>
<td>Symbolic address with path and offset</td>
</tr>
</tbody>
</table>

©1989-2013 Lauterbach GmbH

Training Real-time Trace 27 Display the Trace Contents
Training Real-time Trace

Time Information

<table>
<thead>
<tr>
<th>Trace.List DEFault TIme.Zero</th>
</tr>
</thead>
</table>

| TIme.Back             | Time relative to the previous record. |
| TIme.Zero             | Time relative to the global zero point. |

Set the Global Zero Point

Establish the selected record as global zero point
If your trace port interface supports a cycle accurate mode, timing information can be displayed more precisely.

In order to use cycle accurate mode some presettings are required.

Example for ARM11:

```
ETM.DataTrace OFF ; the cycle accurate mode generates
               ; more load on the trace port
ETM.CycleAccurate ON ; if possible switch data tracing
Trace.CLOCK 176.MHZ ; off
             ; enable cycle accurate mode
; inform TRACE32 about your core
Trace.List DEFault List.TIme TIme.Fore TIme.Back.OFF
```

In order to display the timing information use the following command:

```
Trace.List DEFault List.TIme Fore TIme Back OFF
```

Timing information can now be display for every instruction. (Small rounding errors are possible.)
In addition to the timing information the number of clocks needed by an instruction can be displayed.

```
Trace.List DEFault List.TIme TIme.Fore CLOCKS.Fore TIme.Back.OFF
```
Trace.state
Display trace configuration window
### Trace Configuration Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESet</td>
<td>Reset the Trace configuration window to its default settings.</td>
</tr>
<tr>
<td>Init</td>
<td>Clears the trace buffer and re-arm the triggers.</td>
</tr>
<tr>
<td>SnapShot</td>
<td>Uncommonly used.</td>
</tr>
<tr>
<td>List</td>
<td>Display a trace listing.</td>
</tr>
<tr>
<td>AutoArm</td>
<td>The trace is automatically armed, when the program execution is started.</td>
</tr>
<tr>
<td></td>
<td>The trace is automatically switched off, when the program execution stops.</td>
</tr>
<tr>
<td>AutolInit</td>
<td>By default, the trace buffer is not cleared when the program execution is restarted by Go or Step.</td>
</tr>
<tr>
<td></td>
<td>AutolInit automatically clears the trace buffer and re-arms the triggers at every start of the program execution.</td>
</tr>
<tr>
<td>SelfTest</td>
<td>Uncommonly used.</td>
</tr>
</tbody>
</table>
**Size of the Trace Buffer**

<table>
<thead>
<tr>
<th><strong>used</strong></th>
<th>Number of sampled records in the trace buffer.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SIZE</strong></td>
<td>Size of the trace buffer. The size can be decreased for faster search or save operations.</td>
</tr>
</tbody>
</table>
Format the Trace Buffer

1. time Less  Suppress the display of the branch trace package information.
2. time Less  Suppress the display of the assembly code.
3. time Less  Suppress the data trace information.

The **More** button works vice versa.
Correlate the Trace Listing with the Source

All windows opened with the /Track option follow the cursor movements in the active window.
### Browse through the Trace Buffer

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Pg ↑ | Scroll page up. |
| 
Pg ↓ | Scroll page down. |
| Ctrl - Pg ↑ | Go to the first record sampled in the trace buffer. |
| Ctrl - Pg ↓ | Go to the last record sampled in the trace buffer. |

©1989-2013 Lauterbach GmbH
Example: Find a specific symbol address.
Context tracking is a technique that allows the context of the CPU/target to be reconstructed for a selected trace record based on the information sampled in the trace buffer. Context in this case means the contents of memory and registers.

The basic features of the Context Tracking System as described in this section require that:
- full program and data information (all read accesses) is available
- all CPU cycles until the program execution stopped are sampled to the trace buffer

More features and settings for CTS are described with the CTS command group.
Trace based debugging allows to re-run the program and data flow sampled into the trace buffer on the TRACE32 screen. Trace based debugging is set-up as follows:

1.) Select the recording point for which the state of the CPU/target should be reconstructed.

2.) The PC will be set automatically to this recording point.

3.) During the trace-based debugging the changes of memories, variables and registers can be watched.

With active CTS the windows in the TRACE32 screen don’t show the current state of the CPU and the current contents of the memory and registers. The windows display the memory, registers etc. as they were at the moment, when the selected record was sampled.
To avoid irritations there are a number of indicators that show you that CTS is active.

Yellow buttons in the Data.List window indicate that CTS is active.

A yellow field in the state line shows you, that you see the memories and registers as they were when record number -xxxx. was sampled.

A (CTS) after the logical and symbolical address in the state line shows you, that CTS is active.


CTS also supports a **Step.Back** command.
With **OFF** CTS is switched off. The current state of the CPU and the current values of registers and variables are displayed on the TRACE32 screen.

**Step Back**

**OFF** (Switch CTS OFF)
HLL Analysis of the Trace Contents

CTS provides also a number of features for hll trace display.

Analyse each HLL Step

For each hll step the following information is displayed:

- The values of the local and global variables used in the hll step
- The result of the hll step
- The time needed for the hll step

```
CTS.List [<record|range>] [<item> …] [/option]
```

List pure hll trace.
### Analyse the Function Nesting

For each function you get the following information:

- Function parameters (and return value - not available on all CPUs)
- Time spent in the function

---

**Example Function Nesting Analysis**

```c
static int func3()
{
    return 5;
}
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Parameters</th>
<th>Time Spent</th>
</tr>
</thead>
<tbody>
<tr>
<td>func3()</td>
<td></td>
<td>0.500 ms</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.100 ms</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.600 ms</td>
</tr>
</tbody>
</table>

**Push the Less button to get a function nesting analysis**
Tree Display

Click here to get a tree analysis of the function nesting.

Timing Display

Click here to get a graphical analysis of the function runtime.
Background

CTS read and evaluated the current state of the target.

1. CTS can only perform a correct context reconstruction, if solely the processor core, for which the data flow is sampled into the trace buffer, writes to memory. If there are memory addresses, e.g. dual-ported memories or peripherals, that are change otherwise, these addresses have to be excluded by the command `MAP.VOLATILE` from the CTS context reconstruction. These memory addresses are then displayed as unknown if CTS is used.

2. CTS performs memory reads while performing a context reconstruction. If read accesses to specific memory-mapped peripherals should be prevented, the addresses have to be excluded by the command `MAP.VOLATILE` from the CTS context reconstruction. These memory addresses are then displayed as unknown if CTS is used.

3. Under certain circumstances the reconstruction of the program flow can cause BUSERRORS on the target system. If this is the case, it is recommended to load the code to virtual memory.
4. CTS has to be re-configured if:
   - the program execution is still running while CTS is used.
   - not all CPU cycles until the stop of the program execution are sampled to the trace.
   - the trace contents is reprocessed with a TRACE32 instruction set simulator.
   - only the program flow is sampled to the trace buffer.

In all these cases the current state of the target can not be used by CTS. For more information refer to the command **CTS.state**.

**MAP.VOLATILE <range>**
Exclude addresses from CTS

**CTS.state**
Reconfigure CTS.
Cache Analysis (PowerTrace)

Using the PowerTrace tool

- an analysis of the instruction cache operation is possible, if the information about the program flow is sampled into the trace buffer.
- an analysis of the data cache operation is possible if at least the addresses for the read and write accesses are sampled to the trace buffer.

It is recommended to use the trace in FIFO or LEASH mode in order to perform a cache analysis. No trace filters should be used.

The cache analysis is part of the CTS functionality.
Background

The objective of the Cache Analysis is to verify the cache efficiency.

Effective Cache Usage

Base for an effective cache usage is the mapping of memory addresses to cache lines.

Mapping Example

Instruction cache:
- 16 bytes per cache line
- 512 cache lines
- 2-way set associative cache

A 32-bit memory address is mapped as follows to a cache line of this instruction cache:
- Bit [3..0] are used to address a word/byte within a cache line (4 bits for 16 bytes).
- Bit [12..4] are used to select the cache line (9 bits for 512 cache lines).
- Bit [31..13] are entered as tag to the cache line.

<table>
<thead>
<tr>
<th>Tag entry</th>
<th>Cache line selection</th>
<th>Byte within the cache line</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 13 12 4</td>
<td>3 0</td>
<td></td>
</tr>
</tbody>
</table>

All program parts that match the address in bit [12..4] are stored in the same cache line.
Effective cache usage: The mapping of memory addresses to cache lines results in an evenly usage of all cache line.

Ineffective cache usage: The mapping of memory addresses to cache lines results in an unbalanced usage of cache lines. Many memory addresses are mapped to a few cache lines while other cache lines are hardly used. Ineffective cache usage generates a needless high number of cache victims and this slows down the program execution.

After ineffective cache usage is detected, the basic idea of the cache analysis is:

- to move code/data that is mapped to heavily used cache lines to suitable addresses
- so that code/data is mapped to hardly used cache lines afterwards.

This address relocation can be performed via the relocation information of the linker command file.

It is recommended to optimize the cache efficiency for single time critical functions. An optimization of the complete application is to complex in most cases.
Important Terms

**Cache hit**: An instruction or data required by the program is already located in the cache.

**Cache miss**: An instruction or data required by the program is not located in the cache and therefore has to be loaded from the external memory.

**Cache victim**: An instruction or data that has been evicted from the cache in order to create space (after a cache miss) for the program part currently required. Evicted cache contents has to be reloaded from the slow external memory when the program needs them again.

**Write-through cache**: All write operation to cached addresses are performed in parallel to the cache and the memory.

**Copy-back cache**: Write operations are only performed to cache. The change cache location is marked as dirty and is copied back to memory when evicted.

**Read-allocated cache**: At cache misses on write accesses the write operation is performed on memory and the data is not loaded to cache.
Definition of the Cache Structure

For most CPUs the definition of the cache structure is performed by TRACE32 when the CPU is selected in the SYSTEM window.
Display the definition of the cache structure
The definition of the MMU Architecture is fundamental for the Cache Analysis.

If the MMU Architecture is set, the cache analysis takes all manipulations on the cache control registers into account for the cache analysis:

- Cache flushes
- Switch-on and switch-off of the caches
- Cache locks (not implemented yet)

MMU Architecture is automatically set to NONE if:

- the cache analysis is not fully implemented for the selected CPU.
- for the selected CPU a program and data flow can only be sampled to the trace after the cache was switched off (e.g. PowerQUICC II and PowerQUICC III).
Since the cache analysis is based on CTS
- the program and data flow sampled to the trace
- the current state of the target

is used to perform the cache analysis.

The use of the current state of the target requires, that all CPU cycles until the stop of the program execution are sampled to the trace buffer (Trace.Mode Fifo or Trace.Mode Leash). If this is not the case CTS has to be reconfigured by the CTS.state command.

If the cachability is controlled by the MMU (CTS.CACHE.Mode MMU) then the MMU reconstruction for the cache analysis is always performed via
- the current state of the MMU control registers
- the MMU translation tables in the target memory

No reconfiguration of CTS is possible in this case. This means the current state of the target has to be accurate before the cache analysis is started.
To analyse the cache efficiency of a single function the following steps are required:

1. **Select the Mode CACHE in the CTS window.**

![Select CTS Mode CACHE](image)

If CTS Mode CACHE is selected, the contents of caches and TBLs is reconstructed by TRACE32 in order to perform a cache analysis.

Please activate the CTS Mode CACHE only if you plan to perform a cache analysis. The CTS Mode CACHE consumes a lot of time and allocated a lot of memory on the host system.

2. **Push the View button in the CTS.CACHE window**

![Push View button in CTS.CACHE window](image)
3. Push the Process button.

4. Open a trace listing and set a reference point to the function entry.
5. Move the cursor to the end of the function and use the command Set CTS.
The cache analysis displays the following results:

The cache analysis is performed from the trace record at the reference point (18603.) to the trace record at the current cursor position (34650.)

C-R displays the number of trace records between the current cursor position and the reference point as well as the time spent by the program between those records.
Interpretation of the result:

All memory accesses

- known
  - hits
  - miss
  - victims
  - unknown

- cached
  - trashes
  - nwrites
  - flushes
  - writethrus
  - copybacks
  - writes
  - reads

unknown

hits

miss

victims

unknown

cached

trashes

nwrites

flushes

writethrus

copybacks

writes

reads

©1989-2013 Lauterbach GmbH
<table>
<thead>
<tr>
<th><strong>unknown</strong></th>
<th>all accesses for which TRACE32 has no information</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The cache analysis is based on the memory addresses recorded in the trace buffer. Before the first memory address is mapped to a specific cache line the contents of this cache line is unknown. Other reasons for unknown are: gaps in the trace recording, missing address information etc. (percentage is based on all memory accesses)</td>
</tr>
<tr>
<td><strong>cached</strong></td>
<td>number of accesses to cached addresses (percentage is based on all memory accesses)</td>
</tr>
<tr>
<td><strong>hits</strong></td>
<td>number of cache hits (percentage is based on all cached accesses)</td>
</tr>
<tr>
<td><strong>miss</strong></td>
<td>number of cache misses (percentage is based on all cached accesses)</td>
</tr>
<tr>
<td><strong>victims</strong></td>
<td>number of cache victims (percentage is based on all cached accesses)</td>
</tr>
<tr>
<td><strong>flushes</strong></td>
<td>number of cache lines that were flushed (percentage is based on all memory accesses)</td>
</tr>
<tr>
<td><strong>copybacks</strong></td>
<td>number of cache lines that were copied back to memory (percentage is based on all memory accesses)</td>
</tr>
<tr>
<td><strong>writethrus</strong></td>
<td>number of cache lines that were written through to memory (percentage is based on all memory accesses)</td>
</tr>
<tr>
<td><strong>nawrites</strong></td>
<td>writes in a read-allocated cache (percentage is based on all memory accesses)</td>
</tr>
<tr>
<td><strong>reads</strong></td>
<td>number of not-cached reads (percentage is based on all memory accesses)</td>
</tr>
<tr>
<td><strong>writes</strong></td>
<td>number of not-cached writes (percentage is based on all memory accesses)</td>
</tr>
<tr>
<td><strong>trashes</strong></td>
<td>discarded accesses (ARM11 only) (percentage is based on all memory accesses)</td>
</tr>
</tbody>
</table>
Analysis for Single Cache Set

Which cache sets have a particular high cache victim rate? (to many memory addresses are mapped to these sets)

Are there any cache sets not being or hardly being used? (sets that are suitable for address relocations)

Break down the result to the single cache set.
Which memory addresses compete for a heavily used cache set? (code/data that has to be relocated)

Breakdown the result of a single cache set to the addresses competing for this cache set by a double click.

View the memory contents for a selected address by a double click.

©1989-2013 Lauterbach GmbH
Calculation Details

If you want to see how the number of cache hits / cache misses / unknown is calculated for each assembler line proceed as follows:

1. Select CTS mode CACHE in the CTS window.

2. Open a CTS.List Window.
3. Click to the small dot beside a hll line.

<table>
<thead>
<tr>
<th>Mode</th>
<th>WT</th>
<th>CB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Write Through</td>
<td>Copy Back</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I: &lt;mode&gt; &lt;hits&gt;/&lt;misses&gt;</th>
<th>Instruction cache hits and misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>I: &lt;mode&gt; &lt;number&gt;?</td>
<td>Number of unknown accesses to the instruction cache</td>
</tr>
<tr>
<td>I: NC &lt;number&gt;</td>
<td>Number of accesses to not cached instruction addresses</td>
</tr>
<tr>
<td>D: &lt;mode&gt; &lt;hits&gt;/&lt;misses&gt;</td>
<td>Data cache hits and misses</td>
</tr>
<tr>
<td>D: &lt;mode&gt; &lt;number&gt;?</td>
<td>Number of unknown accesses to the data cache</td>
</tr>
<tr>
<td>D: NC &lt;number&gt;</td>
<td>Number of accesses to not cached data addresses</td>
</tr>
</tbody>
</table>
If you perform trace-based debugging you can also watch the changes in the caches.

1. Select CTS mode CACHE in the CTS window.

2. Select the start point for trace-based debugging.
3. Open a CACHE.Dump window to watch the changes in the cache while re-debugging the trace contents.
Export Required Data from PowerTrace

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trace.SAVE cache</td>
<td>; save the contents of the trace buffer to a file</td>
</tr>
<tr>
<td>Data.SAVE.S3record mmu A:0x1000++0xfff</td>
<td>; Save MMU translation table to file</td>
</tr>
</tbody>
</table>

Check the value of the MMU control registers. They need to be set to the same values in the TRACE32 instruction set simulator.
Set-up the TRACE32 Instruction Set Simulator for the Cache Analysis

1. Select the CPU and activate the simulator.

   SYStem.Down
   SYStem.CPU ARM925T
   SYStem.Up

2. Load the MMU translation table.

   Data.LOAD.S3record mmu

3. Configure the MMU control register.

   Data.Set C15:1 0x127f ; example for ARM9 here
   Data.Set C15:2 0xa000000
   Data.Set C15:3 0x55555555

4. Load trace contents from file.

   Trace.LOAD cache
   Trace.List /FILE ; display trace contents loaded from
                    ; the file

5. Configure CTS.

   CTS.Mode CACHE ; switch on the cache analysis within
                    ; CTS
   CTS.UseMemory OFF ; current contents of the target memory
                      ; can not be used for cache analysis
   CTS.UseRegister OFF ; current contents of the CPU registers
                         ; can not be used for cache analysis
6. Configure the cache structure, if this is not automatically done by selecting the CPU.

```plaintext
CTS.CACHE.WAYS 4.
CTS.CACHE.SETS IC 512.
CTS.CACHE.SETS DC 128.
...
```

7. Process the cache analysis.

```plaintext
CTS.PROCESS /FILE
```
### Actions for the Trace (not available for all architecture)

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TraceEnable</strong></td>
<td>Enable the trace on the specified event.</td>
</tr>
<tr>
<td><strong>TraceON</strong></td>
<td>Switch the sampling to the trace ON on the specified event.</td>
</tr>
<tr>
<td><strong>TraceOFF</strong></td>
<td>Switch the sampling to the trace OFF on the specified event.</td>
</tr>
<tr>
<td><strong>TraceTrigger</strong></td>
<td>Stop the sampling to the trace on the specified event. A trigger delay is possible.</td>
</tr>
</tbody>
</table>
Examples for TraceEnable

**Example:** Sample only the entry to the function sieve into the trace.

1. Set a Program breakpoint to sieve and select the action TraceEnable.

   ![TraceEnable example](image1)

2. Start the program execution and stop it.

3. Display the result.

   ![TraceEnable result](image2)

   ```c
   char flags[SIZE+1];
   int sieve(void) /* sieve of erathostenes */
   {
      register int i, primz, k;
      int anzahl;
      anzahl = 0;
      for (i = 0; i < SIZE+1; i++)
      {
         primz = 1;
         for (k = 0; k < anzahl && primz; k++)
         {
            if (i % primes[k] == 0)
            {
               primz = 0;
               break;
            }
         }
         if (primz)
         {
            flags[i] = 1;
            anzahl += 1;
         }
      }
   }
   ```

   On some architectures the selected cycle plus a few more cycles are sampled.
**Example:** Sample only the function `func2`.

1. Set a Program breakpoint to the entry of the function `func2` and select the action TraceON.
2. Set a Program breakpoint to the exit of the function func2 and select the action TraceOFF.

3. Start the program execution and stop it.

4. Display the result.
Example: Stop the sampling to the trace buffer when the function sieve is entered.

1. Set a Program breakpoint to sieve and select the action TraceTrigger.

2. Start the program execution.

*running* is displayed in green, when the program execution is running and sampling to the trace buffer is active.

*running* is displayed in blue when the program execution is running and sampling to the trace buffer is stopped (TraceTrigger occurred).
Displaying the result requires access to the on-chip/target memory in order to read the code information. If the on-chip debug support doesn't support this, the trace contents can only be displayed if:

- the program execution is stopped
- the code was loaded to the virtual memory.

(Data.LOAD <file> /VM)
Example for TraceTrigger with a Trigger Delay

**Example**: Stop the sampling to the trace buffer when the function sieve is entered and another 10% of the trace buffer is filled.

1. Set a Program breakpoint to sieve and select the action TraceTrigger.

2. Define the trigger delay in the *Trace Configuration* Window.
3. Start the program execution.

- **Running** is displayed in green, when the program execution is running and sampling to the trace buffer is active.

4. Display the result.

- **Running** is displayed in turquoise when the program execution is running and **TraceTrigger** occurred/**TDelay** started.

- **Running** is displayed in blue when the program execution is running and sampling to the trace buffer is stopped (**TraceTrigger** occurred and **TDelay** is up).

Push the **Trigger** button in the **Trace Goto** window to find the record, where **TraceTrigger** was accepted by the trace. Here the sign of the record numbers has changed.

The **TraceTrigger** event is usually shortly before/after this point.
Function Run-Times Analysis

All commands for the function run-time analysis introduced in this chapter use the contents of the trace buffer as base for their analysis.

Software under Analysis (no OS, OS or OS+MMU)

For the use of the function run-time analysis it is helpful to differentiate between three types of application software:

1. Software without operating system (abbreviation: no OS)
2. Software with an operating system without dynamic memory management (abbreviation: OS)
3. Software with an operating system that uses dynamic memory management to handle processes/tasks (abbreviation: OS+MMU). If an OS+MMU is used several processes/tasks can run at the same virtual addresses.

Flat vs. Nesting Analysis

TRACE32 provides two methods to analyze function run-times:

- Flat analysis
- Nesting analysis
Basic Knowledge about the Flat Analysis

The flat function run-time analysis bases on the symbolic instruction addresses of the trace entries. The time spent by an instruction is assigned to the corresponding function/symbol region.

<table>
<thead>
<tr>
<th>min</th>
<th>shortest time continuously in the address range of the function/symbol region</th>
</tr>
</thead>
<tbody>
<tr>
<td>max</td>
<td>longest time continuously in the address range of the function/symbol region</td>
</tr>
</tbody>
</table>

©1989-2013 Lauterbach GmbH
The function nesting analysis analyses only high-level language functions.
In order to display a nested function run-time analysis TRACE32 analyzes the structure of the program execution by processing the trace information to find:

1. **Function entries**

2. **Function exits**

3. **Entries to interrupt service routines (asynchronous)**
   - Interrupts are identified as follows:
     - The trace port broadcasts the occurrence of an interrupt (e.g. PPC4xx).
     - An entry to the vector table is detected and the vector address indicates an asynchronous/hardware interrupt (e.g. ARM9).
     - If the vector table base address is configurable the usage of the command `SYStem.OPTION VECTORS` might be necessary (e.g. MPC55xx).

4. **Exits of interrupt service routines**

5. **Entries to TRAP handlers (synchronous)**

6. **Exits of TRAP handlers**
Summary

The nesting analysis provides more details about the program run, but it is much more sensitive than the flat analysis. Missing or tricky function exits for example result in a worthless nesting analysis.
Flat Analysis

Standard Function Run-time Analysis

No OS or OS

<table>
<thead>
<tr>
<th>Trace.STATistic.sYmbol</th>
<th>Flat function run-time analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>- numerical display</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Trace.Chart.sYmbol</th>
<th>Flat function run-time analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>- graphical display</td>
</tr>
</tbody>
</table>

Preparation:

The flat function run-time analysis does **not** require any **data information** if no OS or an OS is used. That’s why it is recommended to switch the broadcasting of data information off. This avoids FIFOFULLs and allows a better utilization of the trace memory. Examples:

```
ETM.DataTrace OFF ; ARM-ETM
NEXUS.DTM OFF ; PowerPC NEXUS
```
### survey

<table>
<thead>
<tr>
<th>Samples</th>
<th>number of recorded program sections (instruction flow continuously in the address range of a function/symbol region)</th>
</tr>
</thead>
</table>

| Total   | time period recorded by the trace |

### function details

<table>
<thead>
<tr>
<th>Address</th>
<th>function/symbol region name (other) program sections that can not be assigned to a function/symbol region</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Total</th>
<th>time period in the function/symbol region during the recorded time period</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Min</th>
<th>shortest time continuously in the address range of the function/symbol region</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Max</th>
<th>longest time continuously in the address range of the function/symbol region</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Avr</th>
<th>average time continuously in the address range of the function/symbol region</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>number of entries into the address range of the function/symbol region</td>
</tr>
<tr>
<td>---------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ratio</td>
<td>ratio of time in the function/symbol region with regards to the total time period recorded</td>
</tr>
</tbody>
</table>

Pushing the **Config** button provides the possibility to specify a different sorting criterion or a different column layout.

By default the functions/symbol regions are sorted by their recording order.
Pushing the **Chart** button in the **Trace.List** window opens a **Trace.Chart.Symbol** window.
If **Window** is switched ON in the **Chart Config** window, the functions that are active at the selected point of time are visualized in the scope of the **Trace.Chart.Symbol** window. This is helpful especially if you scroll horizontally.

**OS+MMU**

tbd.
Detailed Function Run-time Analysis

If a function seems to be very time consuming details on the run-time of single instructions can be displayed with the help of the **Trace.ISTAT** command group.

**Preparation**

It is recommended to switch the broadcasting of data trace information off. This avoids FIFOFULLs and allows a better utilization of the trace memory.

The run-time results on single instructions are more accurate if cycle-accurate tracing is used.

```
ETM.CycleAccurate ON ; switch cycle accurate tracing on
Trace.CLOCK 600.MHz ; inform TRACE32 about your
                      ; CPU/core frequency
```

A high number of local FIFOFULLs might affect the result of the instruction statistic.

**Processing**

The command group **Trace.ISTAT** works with a data base. The measurement includes the following steps:

1. Specify the core/CPU clock.
2. Clear the data base.
3. Fill the trace memory.
4. Transfer the contents of the trace memory to the data base.
5. Display the result.
6. (Repeat step 3-5 if required)
The following commands are available:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Trace.CLOCK</strong> <code>&lt;clock&gt;</code></td>
<td>Specify the core/CPU clock for the trace evaluation.</td>
</tr>
<tr>
<td><strong>ISTATistic.RESet</strong></td>
<td>Clear the instruction statistic data base.</td>
</tr>
<tr>
<td><strong>ISTATistic.add</strong></td>
<td>Add the contents of the trace memory to the instruction statistic data base.</td>
</tr>
<tr>
<td><strong>ISTATistic.ListFunc</strong></td>
<td>List function run-time analysis based on the contents of the instruction statistic data base.</td>
</tr>
<tr>
<td><strong>Data.List</strong> <code>&lt;address&gt; /ISTAT</code></td>
<td>List run-time analysis for the single instructions.</td>
</tr>
</tbody>
</table>

A detailed function run-time analysis can be performed as follows (ARM11 with ETMv3 as example):

```plaintext
; core specific set-up
ETM.DataTrace OFF ; switch broadcasting of data trace
                   ; information off
ETM.CycleAccurate ON ; switch cycle accurate tracing on
...
Trace.CLOCK 176.MHz ; inform TRACE32 about your CPU
                     ; frequency
ISTATistic.RESet  ; reset instruction statistic data
                    ; base
Trace.Mode Leash ; switch trace to Leash mode
Go ; start program execution
;WAIT !RUN() ; wait until program stops
Trace.FlowProcess ; upload the trace information to
                    ; the host and merge source code
IF A.FLOW.FIFOFULL>6000.
    PRINT “Warning: Please control the FIFOFULLS”
    ISTATistic.add ; add trace information to
                    ; instruction statistic data
                    ; base
    ISTATistic.ListFunc ; list function run-time statistic
```
| **address** | address range of the module, function or hll line |
| **tree**   | flat module/function/hll line tree |
| **coverage** | code coverage of the module, function or hll line |
| **count** | number of function/hll line executions |
| **time** | total time spent by the module, function or hll line |
| **clocks** | total number of clocks spent by the module, function or hll line |
| **ratio** | Percentage of the total measurement time spent in the module, function or hll line |
| **cpi** | average clocks per instruction for the function or the hll line |
Data.ListAsm /ISTAT ; list instruction run-time ; statistic

<table>
<thead>
<tr>
<th>count</th>
<th>total number of instruction executions</th>
</tr>
</thead>
<tbody>
<tr>
<td>clocks</td>
<td>total number of clocks for the instruction</td>
</tr>
<tr>
<td>cpi</td>
<td>average clocks per instruction</td>
</tr>
</tbody>
</table>
If `exec` or `notexec` is 0 for an instruction with condition, the instruction is bold-printed on a yellow background. All other instructions are bold-printed on a yellow background if they were not executed.

<table>
<thead>
<tr>
<th>exec</th>
<th>conditional instructions: number of times the instruction was executed because the condition was true.</th>
</tr>
</thead>
<tbody>
<tr>
<td>notexec</td>
<td>conditional instructions: number of times the instruction wasn’t executed because the condition was false.</td>
</tr>
<tr>
<td>coverage</td>
<td>instruction coverage</td>
</tr>
</tbody>
</table>

If exec or/and notexec is 0 for an instruction with condition, the instruction is bold-printed on a yellow background. All other instruction are bold-printed on a yellow background if they were not executed.

**OS+MMU**

tbd.
Nestng Analysis

Rules

1. The nesting analysis analyses only high-level language functions.
2. The result of the nesting analysis is only reliable if there are no trace gaps (e.g. FIFOFULLs) in the trace.

Analysis

No OS

Preparation:

The nesting function run-time analysis doesn’t require any data information if no os is used. That’s why it is recommended to switch the broadcasting of data information off. This avoids FIFOFULLs and allows a better utilization of the trace memory. Examples:

<table>
<thead>
<tr>
<th>ETM.DataTrace OFF</th>
<th>; ARM-ETM</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEXUS.DTM OFF</td>
<td>; PowerPC NEXUS</td>
</tr>
</tbody>
</table>
In order to prepare the results for the nesting analysis TRACE32 postprocesses the program flow to find:

- **Function entries**

  The execution of the first instruction of a high-level function is regarded as function entry.

  Additional identifications for function entries are implemented depending on the processor architecture and the used compiler.

```
Trace.Chart.Func ; function func10 as example
Trace.List /Track
```
- **Function exits**
  A RETURN instruction within a hll function is regarded as function exit.
  Additional identifications for function exits are implemented depending on the processor architecture and the used compiler.

- **Entries to interrupt service routines (asynchronous)**
  Interrupts are identified if an entry to the vector table is detected and the vector address indicates an asynchronous/hardware interrupt.
  If an interrupt was identified, the following entry to a hll function is regarded as entry to the interrupt service routine.

- **Exits of interrupt service routines**
  A RETURN / RETURN FROM INTERRUPT within the hll interrupt service routine is regarded as exit of the interrupt service routine.

- **Entries to TRAP handlers (synchronous)**
  If an entry to the vector table was identified and if the vector address indicates a synchronous interrupt/trap the following entry to a hll function is regarded as entry to the trap handler.

- **Exits of TRAP handlers**
  A RETURN / RETURN FROM INTERRUPT within the hll TRAP handler is regarded as exit of the TRAP handler.
**Trace.STATistic.Func**  
Nested function run-time analysis  
- numeric display

<table>
<thead>
<tr>
<th>survey</th>
<th>number of functions in the trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>func</td>
<td>total measurement time</td>
</tr>
<tr>
<td>total</td>
<td>total time in interrupt service routines</td>
</tr>
</tbody>
</table>

**funcs**: 92.  
**total**: 4.203ms  
**intr**: 20.665ms
The function nesting is regarded as tree, root is the root of the function nesting.

- Hll function
  \arm\a\_li\_aif\func7

- (root)
  \root

- Hll interrupt service routine
  \umts\bute\build\intr\_os\_wrapper\_intr\_os\_prologue60

- Hll trap handler
  \_\_\_ArmVectorSwi
### columns (cont.)

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>total</td>
<td>total time within the function</td>
</tr>
<tr>
<td>min</td>
<td>shortest time between function entry and exit, time spent in interrupt service routines is excluded</td>
</tr>
<tr>
<td>max</td>
<td>longest time between function entry and exit, time spent in interrupt service routines is excluded</td>
</tr>
<tr>
<td>avr</td>
<td>average time between function entry and exit, time spent in interrupt service routines is excluded</td>
</tr>
</tbody>
</table>

No min time is displayed if a function exit was never executed.
If function entries or exits are missing, this is displayed in the following format:

<times within the function> (<number of missing function entries>/<number of missing function exits>).

Interpretation examples:
1. 2. (2/0): 2 times within the function, 2 function entries missing
2. 4. (0/3): 4 times within the function, 3 function exits missing
3. 11. (1/1): 11 times within the function, 1 function entry and 1 function exit is missing.

If the number of missing function entries or exits is higher than 1 the analysis performed by the command `Trace.STATistic.Func` might fail due to nesting problems. A detailed view to the trace contents is recommended.
Pushing the **Config...** button allows to display additional columns

<table>
<thead>
<tr>
<th>columns (cont.) - times only in function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Internal</strong></td>
</tr>
<tr>
<td><strong>IAVeRage</strong></td>
</tr>
<tr>
<td><strong>IMIN</strong></td>
</tr>
<tr>
<td><strong>IMAX</strong></td>
</tr>
<tr>
<td><strong>InternalRatio</strong></td>
</tr>
<tr>
<td><strong>InternalBAR</strong></td>
</tr>
</tbody>
</table>
### columns (cont.) - times in sub-functions and TRAP handlers

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>External</td>
<td>total time spent within called sub-functions/TRAP handlers</td>
</tr>
<tr>
<td>EAVeRage</td>
<td>average time spent within called sub-functions/TRAP handlers</td>
</tr>
<tr>
<td>EMIN</td>
<td>shortest time spent within called sub-functions/TRAP handlers</td>
</tr>
<tr>
<td>EMAX</td>
<td>longest time spent within called sub-functions/TRAP handlers</td>
</tr>
</tbody>
</table>

### columns (cont.) - interrupt times

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTR</td>
<td>total time the function was interrupted</td>
</tr>
<tr>
<td>INTRMAX</td>
<td>max. time one function pass was interrupted</td>
</tr>
<tr>
<td>INTRCount</td>
<td>number of interrupts that occurred during the function run-time</td>
</tr>
</tbody>
</table>
The following graphic gives an overview how times are calculated:
Further Evaluations

Trace.Chart.Func

Nested function run-time analysis
- graphical display
Trace.STATistic.TREE

Nested function run-time analysis
- tree display

©1989-2013 Lauterbach GmbH
**Trace.STATistic.LINKage**

Nested function run-time analysis

- linkage analysis

---

**Table:**

<table>
<thead>
<tr>
<th>Function</th>
<th>Total</th>
<th>Min</th>
<th>Max</th>
<th>Avg</th>
<th>Count</th>
<th>% Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>1.058</td>
<td>-</td>
<td>1.058</td>
<td>1.057</td>
<td>1</td>
<td>1.058%</td>
</tr>
<tr>
<td>func2</td>
<td>31.870</td>
<td>18.048</td>
<td>18.048</td>
<td>18.048</td>
<td></td>
<td>31.870%</td>
</tr>
<tr>
<td>func3</td>
<td>15.560</td>
<td>18.048</td>
<td>18.048</td>
<td>18.048</td>
<td></td>
<td>15.560%</td>
</tr>
<tr>
<td>func4</td>
<td>18.048</td>
<td>18.048</td>
<td>18.048</td>
<td>18.048</td>
<td></td>
<td>18.048%</td>
</tr>
<tr>
<td>func5</td>
<td>18.048</td>
<td>18.048</td>
<td>18.048</td>
<td>18.048</td>
<td></td>
<td>18.048%</td>
</tr>
</tbody>
</table>

**Diagram:**

- Linkage Analysis

---

©1989-2013 Lauterbach GmbH
OS and OS+MMU

The nesting function run-time analysis requires the complete program flow and information on the process/task switches if an OS or OS+MMU is used.

- **Address and data information for write accesses are broadcasted via the trace port**

The most convenient way to get information about process/task switches is a trace port that can show address and data information for write accesses.

**Preparation:**

It is recommended to set-up the broadcasting of trace information as follows:

- **Break.Set TASK.CONFIG(magic) /TraceData /Write**

This restriction of the information broadcasting avoids FIFOFULLs and allows a better utilization of the trace memory.

- **The trace port interface provides special trace messages to indicate process/task switches and the generation of these message is supported by the OS or OS+MMU.**

**Examples:**

- **ETM.ContextID 32** ; Context-ID messages for some versions of the ARM-ETM
- **NEXUS.OTM ON** ; Ownership trace messages for NEXUS PowerPC
In order to prepare the results for the nesting analysis TRACE32 postprocesses the trace contents to find:

- **Function entries**
  
The execution of the first instruction of a hll function is regarded as function entry.
  
  Additional identifications for function entries are implemented depending on the processor architecture and the used compiler.

```
Trace.Chart.Func                      ; function hal_lsbindex as
Trace.List /Track
```

![Image of Trace.Charting and Trace.List/Track windows](image-url)
• Function exits

A RETURN instruction within a hll function is regarded as function exit.

Additional identifications for function exits are implemented depending on the processor architecture and the used compiler.

• Entries to interrupt service routines (asynchronous)

Interrupts are identified if an entry to the vector table is detected and the vector address indicates an asynchronous/hardware interrupt.

If an interrupt was identified, the following entry to a hll function is regarded as entry to the interrupt service routine.

• Exits of interrupt service routines

A RETURN / RETURN FROM INTERRUPT within the hll interrupt service routine is regarded as exit of the interrupt service routine.
• Entries to TRAP handlers (synchronous)
  If an entry to the vector table was identified and if the vector address indicates a synchronous interrupt/trap the following entry to a hll function is regarded as entry to the trap handler.

• Exits of TRAP handlers
  A RETURN / RETURN FROM INTERRUPT within the hll TRAP handler is regarded as exit of the TRAP handler.

• Task Switches
**Trace.STATistic.TASKFunc**  
RTOS-aware nested function run-time analysis  
- numeric display

---

**Survey**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><em>func</em></td>
<td>number of functions in the trace</td>
</tr>
<tr>
<td><em>total</em></td>
<td>total measurement time</td>
</tr>
<tr>
<td><em>intr</em></td>
<td>total time in interrupt service routines</td>
</tr>
</tbody>
</table>
Hll function assigned to a specific task/thread

```
\demo\sched\Cyg_Scheduler::unlock_inner@Thread_0
```

root function assigned to a specific task/thread

```
(root)@Thread_0
```

The function nesting is regarded as tree, (root) is the root of the function nesting for a specific task/thread.

Hll function assigned to the root task/thread

```
\demo\mutex\Cyg_Mutex::lock@(root)
```

All hll functions that can not be assigned to a task/thread are assigned to the (root) task. This affects most commonly the function, that are recorded in the trace before the first task switch was recorded.

Hll interrupt service routine

```
→\demo\Global\HAL_ARM_IRQ Vector@(root)
```

Hll trap handler

```
→__ArmVectorSwi
```
## columns (cont.)

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>total</td>
<td>total time within the function</td>
</tr>
<tr>
<td>min</td>
<td>shortest time between function entry and exit, time spent in interrupt service routines is excluded. No <strong>min</strong> time is displayed if a function exit was never executed.</td>
</tr>
<tr>
<td>max</td>
<td>longest time between function entry and exit, time spent in interrupt service routines is excluded</td>
</tr>
<tr>
<td>avr</td>
<td>average time between function entry and exit, time spent in interrupt service routines is excluded</td>
</tr>
</tbody>
</table>
If function entries or exits are missing, this is displayed in the following format:

\(<times\ within\ the\ function\>.\ (<number\ of\ missing\ function\ entries>/<number\ of\ missing\ function\ exits>).\)

### Interpretation examples:

1. 2. (2/0): 2 times within the function, 2 function entries missing
2. 4. (0/3): 4 times within the function, 3 function exits missing
3. 11. (1/1): 11 times within the function, 1 function entry and 1 function exit is missing.

If the number of missing function entries or exits is higher than 1, the analysis performed by the command `Trace.STATistic.Func` might fail due to nesting problems. A detailed view to the trace contents is recommended.

### Columns (cont.)

<table>
<thead>
<tr>
<th>count</th>
<th>number of times within the function</th>
</tr>
</thead>
</table>

In the table above, `count` represents the number of times within the function.
columns (cont.) - times only in function

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal</td>
<td>total time between function entry and exit without called sub-functions,TRAP handlers, interrupt service routines</td>
</tr>
<tr>
<td>IAVeRage</td>
<td>average time between function entry and exit without called sub-functions,TRAP handlers, interrupt service routines</td>
</tr>
<tr>
<td>IMIN</td>
<td>shortest time between function entry and exit without called sub-functions,TRAP handlers, interrupt service routines</td>
</tr>
<tr>
<td>IMAX</td>
<td>longest time spent in the function between function entry and exit without called sub-functions,TRAP handlers, interrupt service routines</td>
</tr>
<tr>
<td>InternalRatio</td>
<td>(&lt;Internal time of function&gt;/&lt;Total measurement time&gt;) as a numeric value.</td>
</tr>
<tr>
<td>InternalBAR</td>
<td>(&lt;Internal time of function&gt;/&lt;Total measurement time&gt;) graphically.</td>
</tr>
</tbody>
</table>

Pushing the **Config**… button allows to display additional columns.
### columns (cont.) - times in sub-functions and TRAP handlers

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>External</td>
<td>total time spent within called sub-functions/TRAP handlers</td>
</tr>
<tr>
<td>EAVeRage</td>
<td>average time spent within called sub-functions/TRAP handlers</td>
</tr>
<tr>
<td>EMIN</td>
<td>shortest time spent within called sub-functions/TRAP handlers</td>
</tr>
<tr>
<td>EMAX</td>
<td>longest time spent within called sub-functions/TRAP handlers</td>
</tr>
</tbody>
</table>

### columns (cont.) - interrupt times

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTR</td>
<td>total time the function was interrupted</td>
</tr>
<tr>
<td>INTRMAX</td>
<td>max. time one function pass was interrupted</td>
</tr>
<tr>
<td>INTRCount</td>
<td>number of interrupts that occurred during the function run-time</td>
</tr>
</tbody>
</table>

### columns (cont.) - task/thread related information

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TASK</td>
<td>task/thread name</td>
</tr>
<tr>
<td>ExternalTASK</td>
<td>total time in other tasks</td>
</tr>
<tr>
<td>ExternalTASKMAX</td>
<td>max. time 1 function pass was interrupted by another task</td>
</tr>
<tr>
<td>TASKCount</td>
<td>number of other tasks that interrupt the function</td>
</tr>
</tbody>
</table>

©1989-2013 Lauterbach GmbH
The following graph gives an overview how times are calculated:

- **Start of measurement**
- **First task switch recorded to trace**
- **Entry to func1 in TASK1**
- **func2 in TASK1**
- **func3 in TASK1**
- **TRAP1 in TASK1**
- **func4 in TASK1**
- **TASK3**
- **func4 in TASK1**
- **interrupt1 in TASK1**
- **Exit of func1 in TASK1**
- **Entry to func1 in TASK1**
- **Exit of func1 in TASK1**

- **Total of (root)@root**
- **Total of (root)@TASK1**
- **Total of (func1)@TASK1**
- **Total of (intr)@TASK1**
- **Total of (internal)@TASK1**
- **Total of (external)@TASK1**

- **First entry to TASK1**
- **Last exit of TASK1**

©1989-2013 Lauterbach GmbH
Further Evaluations

Trace.Chart.TASKFunc

RTOS-aware nested function run-time analysis
- graphical display
Trace.STATistic.TASKTREE

- Nested function run-time analysis
- tree display
The command group GROUP allows you to structure application programs consisting of a huge number of functions/modules to ease the debugging process and the evaluation of the trace contents.

Create GROUPS

1. Define a group name.
2. Define the address range(s) for the group.
3. Define a color for the group.
4. Create the group by pushing the Set or Ok button.
5. Verify the settings by using the command GROUP.List.

GROUP.Create <name> <address_range> [/<option>] Create a new group
GROUP.List List group settings

©1989-2013 Lauterbach GmbH
The group members are marked with the defined color in various windows (e.g. in the Source List or the Trace List window).
Hidden Groups

- The display of the code for the group members can be suppressed in the Trace List window.

Activate **hide** in the GROUP List window

- Hidden groups are stepped over during hll debugging.
TRACE32 provides a run-time statistic analysis based on the defined groups.

**Trace.Chart.GROUP**

*Graphical display of group-based run-time analysis*

**Trace.STATistic.GROUP**

*Numerical display of group-based run-time analysis*
Introduction

Task: get the percentage of time used by a high-level language function.

Measurement procedure: The trace recording is stopped periodically in order to sample the current Program Counter.

TRACE32 needs to decompress the recorded program flow information in order to get the Program Counter. This requires that the code can be read from the target memory while the program execution is running. If this is not possible for the processor in use, a copy of the code has to be loaded into TRACE32’s virtual memory:

```
Data.LOAD <file> /VM ; load code from <file> to TRACE32 virtual memory
Data.LOAD <file> /PlusVM ; load code from <file> to target and to TRACE32 virtual memory
```

The command group for sample-based profiling is **PERF.<subcommand>**.
Standard Approach

Steps to be taken:

1. **Switch the broadcasting of data flow information OFF**
   
   Profiling the Program Counter does not require data flow information. That's why it is recommended to switch the broadcasting of data information off. This avoids FIFOFULLs. Examples:
   
   ```
   ETM.DataTrace OFF  ; ARM-ETM
   NEXUS.DTM OFF       ; PowerPC NEXUS
   ```

2. **Load a copy of the code to the TRACE32 virtual memory if your processor does not support reading target memory while the program execution is running.**
   
   ```
   Data.LOAD <file> /VM ; load code from <file> to TRACE32 virtual memory
   ```

3. **Open the PERF configuration window and select the measurement METHOD Trace.**
   
   ![PERF Configuration Window]
   
   **PERF.state**  Display PERF configuration window
   
   **PERF.METHOD Trace**  Select the measurement method trace for the sample-based profiling
4. Enable the sample-based profiling by selecting the OFF state.

**PERF.OFF**

Enable the sample-based profiling

5. Open a result window by pushing the ListFunc button.

**PERF.ListFunc**

Open a HLL function profiling window
6. Start the program execution and the sampling.

A blinking (halt) in the system state field of the TRACE32 state line indicated, that the trace recording is stopped shortly.

The display of perf in blue in any Trace display window indicates, that the trace recording was periodically interrupted by the sample-based profiling. In this case the trace information is inappropriate for any trace analysis.
Push the Detailed button, to get more detailed information on the result.

<table>
<thead>
<tr>
<th>Name</th>
<th>Function name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>Time spent by the function</td>
</tr>
<tr>
<td>WatchTIme</td>
<td>Time the function is observed</td>
</tr>
<tr>
<td>Ratio</td>
<td>Ratio of time spent by the function in percent</td>
</tr>
<tr>
<td>DRatio</td>
<td>Similar to Ratio, but only for the last second</td>
</tr>
<tr>
<td>Address</td>
<td>Function’s address range</td>
</tr>
<tr>
<td>Hits</td>
<td>Number of samples taken for the function</td>
</tr>
</tbody>
</table>
TRACE32 assigns all samples that can not be assigned to a high-level language function to \textit{(other)}. Especially if the ratio for \textit{(other)} is quite high, it might be interesting what code is running there. In this case pushing the button \textbf{ListLABEL} is recommended.

\textbf{PERF.ListLABEL} \hspace{1cm} Open a window for label-based profiling
The program flow sampled into trace buffer can be used to make a detailed code coverage.

A complete code coverage analysis requires the following steps:

1. Clear the code coverage database.
2. Set the trace to Leash mode.
3. Run the program and sample the program flow into the trace buffer.
4. Add the trace contents to the code coverage database.
5. Display the results.

Clear the Code Coverage Database

**COVerage.RESet**
Reset code coverage data base
Set the Trace to Leash Mode

If the trace is working in Leash mode, the program execution is stopped as soon as the trace buffer is full.

Run the Program and Sample the Program Flow

Add the Trace Contents to the Code Coverage Database

```
COVerage.add [/<option>]  Add the trace contents to the coverage data base
```
Display the Results

**COVerage.ListFunc** ` [<address>|<range>]`  
Display coverage for hll functions

**COVerage.SAVE** `<file>`  
Save coverage database to file

**COVerage.LOAD** `<file>`  
Load coverage database from file

---

©1989-2013 Lauterbach GmbH
A TRIGGER connector for the TRACE32-ICD trigger bus is available on:

- Printer Port Interface
- PODBUS Ethernet Controller
- POWER DEBUG / USB
- POWER DEBUG / ETHERNET
- POWERTRACE / ETHERNET
A trigger pulse of at least 100 ns is generated on:

<table>
<thead>
<tr>
<th>Out (to bus)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Break</strong></td>
<td>When the program execution is stopped.</td>
</tr>
</tbody>
</table>
| **ABreak**   | When the state of the trace changed to break (see the **Trace Configuration** window).  
  (for all architecture where the trace port provides trigger features and for all bus traces) |
| **ATrigger** | When the state of the trace changed to trigger (see the **Trace Configuration** window).  
  (for all architecture where the trace port provides trigger features and for all bus traces) |

The polarity of the trigger output can be selected:

<table>
<thead>
<tr>
<th>Mode</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Low</strong></td>
<td>A low active pulse is generated.</td>
</tr>
<tr>
<td><strong>High</strong></td>
<td>A high active pulse is generated.</td>
</tr>
</tbody>
</table>
The trigger pulse is monitored by TRACE32.

**TrBus.Out** Break | ABreak | ATrigger
Define source for external trigger pulse.

**TrBus.Mode** Low | High
Define polarity for external trigger pulse.

**TrBus.view**
Display TrBus settings window.
TRACE32-ICD can react on an external trigger signal:

<table>
<thead>
<tr>
<th>Set (from BUS)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Break</strong></td>
<td>Stop the program execution on an external trigger signal.</td>
</tr>
<tr>
<td><strong>ATrigger</strong></td>
<td>Generate a trigger for the trace. The trigger signal will either stop the trace immediately or it will stop the trace after the trigger delay ran down.</td>
</tr>
</tbody>
</table>

The mode of the trigger input can be selected:

<table>
<thead>
<tr>
<th>Mode</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Low</strong></td>
<td>React on a low active pulse.</td>
</tr>
<tr>
<td><strong>High</strong></td>
<td>React on a high active pulse.</td>
</tr>
<tr>
<td><strong>Falling</strong></td>
<td>React on a falling edge of the external signal.</td>
</tr>
<tr>
<td><strong>Rising</strong></td>
<td>React on a rising edge of the external signal.</td>
</tr>
</tbody>
</table>

**TrBus.Set** Break | ATrigger Define destination for external trigger pulse.  
**TrBus.Mode** Low | High | Falling | Rising Define polarity/edge for external trigger pulse.  
**TrBus.view** Display TrBus settings window.