History

06-Aug-19  Added an option for the command line option --t32-screen=.

05-Oct-18  Added an example for the command line option --t32-logautostart.

17-Apr-18  The sub-chapter “SNOOPer” was removed from the “Trace Methods” chapter. The SNOOPer trace is now described in “Application Note for the SNOOPer Trace” (app_snooper.pdf).

12-Jan-18  Updated descriptions, examples, and screenshots in section “Command Line Arguments for Starting TRACE32” to explain the new command line syntax.
Warning

**WARNING:** To prevent debugger and target from damage it is recommended to connect or disconnect the debug cable only while the target power is OFF.

Recommendation for the software start:

1. Disconnect the debug cable from the target while the target power is off.
2. Connect the host system, the TRACE32 hardware and the debug cable.
3. Power ON the TRACE32 hardware.
4. Start the TRACE32 software to load the debugger firmware.
5. Connect the debug cable to the target.
6. Switch the target power ON.
7. Configure your debugger e.g. via a start-up script.

Power down:

1. Switch off the target power.
2. Disconnect the debug cable from the target.
3. Close the TRACE32 software.
4. Power OFF the TRACE32 hardware.

**Important Information Concerning the Use of the TRACE32 Development System**

Due to the special nature of the TRACE32 development system, the user is advised that it can generate higher than normal levels of electromagnetic radiation which can interfere with the operation of all kinds of radio and other equipment.

To comply with the European Approval Regulations therefore, the following restrictions must be observed:

1. The development system must be used only in an industrial (or comparable) area.
2. The system must not be operated within 20 metres of any equipment which may be affected by such emissions (radio receivers, TVs etc).
Basics

ROM Monitor and JTAG/BDM Debuggers are low cost microprocessor development systems working for numerous CPUs. A number of devices is available which can be combined very flexible to the developer's needs.

- In Circuit Debugger Module (supports BDM, ONCE, JTAG, COP … ports)
- ROM Monitor Interface with EPROM Simulator
- Serial Linked ROM Monitor
- Custom Linked ROM Monitor
- Special Debugger Variants (e.g. public domain hardware or EVA boards)
- ICD State Analyzer Module with Performance and Code Coverage functions
- Serial Line Remote Debugger
- Stimuli-Generator
- Clock-Generator
- Evaluation Boards

These devices are connected to the PC via the PODBUS Interface Card, USB2.0/1.1 or Ethernet. The corresponding software on PC runs with Windows-95/98/ME/NT/2000/XP or Linux and is compatible to the TRACE32-ICE in-circuit emulator system.

In addition to the high speed interface the PODBUS Interface Card contains a universal counter system and a trigger input/output.

You can also use the ICD Debugger as an extension for the TRACE32-ICE system e.g. to control additional CPUs on the target system. The devices are connected to the PODBUS connector of the emulation controller unit. In this configuration the software is available for the same operating systems as the TRACE32-ICE system.

Restrictions

- If the stimuli generator is used, it must be the first device connected to the PODBUS interface.
- There is a maximum of five devices which can be connected to the PODBUS.
Debugger Licenses

For debugging it is necessary to obtain the licenses for the particular processors. The license is saved within debug cable. For some processors, it is possible to save more than one license into the same cable. Those licenses are available as so-called extension licenses (X-licenses, in case of processors belonging to the same family) or additional licenses (A-licenses).

The licenses are dropped within debug cable into a so-called license matrix, consisting of 3 rows and 4 columns (example):

<table>
<thead>
<tr>
<th>ARM Family</th>
<th>ARM7</th>
<th>ARM9</th>
<th>ARM11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LA-7746</td>
<td>LA-7742X</td>
<td>LA-7765X</td>
</tr>
<tr>
<td>C55x Family</td>
<td>TMS320C55x</td>
<td>TMS320C54x</td>
<td>TI DSP Trace</td>
</tr>
<tr>
<td></td>
<td>LA-7830A</td>
<td>LA-7831X</td>
<td>Support</td>
</tr>
<tr>
<td>C6400 Family</td>
<td>TMS320C6400</td>
<td></td>
<td>LA-3802X</td>
</tr>
<tr>
<td></td>
<td>LA-7838A</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The entries in each row within the table shown above are associated to

- Same TRACE32 executable
- Same serial number
- Same maintenance contract
The licenses located in the currently used debug cable can be viewed by typing the `VERSION` command at the TRACE32 command line or by choosing Help menu > About TRACE32:

X-Licenses are possible for processors of the same family as the base license or associated A-License. To learn which license combinations are possible please contact LAUTERBACH sales department or distributor.
Configuration Overview

Regarding the PODBUS interface there are three possible configurations:

- PODBUS Interface Card (PODPC)
- Parallel - PODBUS Converter (PODPAR)
- SCU - PODBUS Converter (PODSCU)

PODBUS Interface Card

The PODBUS Interface Card requires no extra power supply. The counter system on the PODPC card provides measurement functions as described in the 'Count' command group. The BNC connector is used to in- or output the triggers from the debug module.

Basic configuration

The PODBUS Interface Card requires no extra power supply. The counter system on the PODPC card provides measurement functions as described in the 'Count' command group. The BNC connector is used to in- or output the triggers from the debug module.
Parallel - PODBUS Converter

Please use only the standard power supply unit, which you will receive together with the PODPAR interface box. The trigger connector is used to in- or output the triggers from the debug module.

SCU - PODBUS Converter

The PODSCU is used together with the SCU of the standard TRACE32 emulator. The intention of this solution is to provide an Ethernet interface for the PODBUS based debug system.

No extra power supply is required. The trigger connector is used to in- or output the triggers from the debug module. The PODSCU contains the same counter system as the PODPC card.
Together with the debug module you receive a single cable to connect the CPU clock to the target cable plug. This way you can use the divided CPU clock as BDM clock. This option is not available at all ICD Debuggers.

Optionally an EPROM simulator can be used to load and debug programs in the ROM area. One EPROM simulator is used to simulate two 8 bit or one 16 bit EPROM. For that you get 3 adapters together with the EPROM simulator, one for 16 bit EPROMs, one for 8 bit high and one for 8 bit low. To simulate EPROMs with a wider bus size, two or more EPROM simulators can be put together.
ICD Configuration for ROM Monitor

Basic configuration for the ROM Monitor Interface

One EPROM simulator is used to simulate two 8 bit or one 16 bit EPROM. To simulate EPROMs with a wider bus size, two or more EPROM simulators can be chained together. NOTE: The address lines of both adapters of an EPROM simulator are directly connected. If the address lines on the target are not the same (e.g. write enable lines for FLASH), they must be isolated from the target on one adapter.

Together with the EPROM simulator you get 3 adapters. One for 16 Bit EPROMs, one for 8 Bit High and one for 8 Bit Low.
On each adapter there are three extra pins for GND, NMI and RES. NMI and RES are outputs from a HTC Schmitt trigger.

The RESet pin of one adapter can be connected to the reset pin of the CPU to give the ROM Monitor program the possibility to control the RESET of the CPU.

The NMI pin of one adapter can be connected to the NMI pin of the CPU to manually stop a program running on the target system.

ROM Monitors are provided for CPUs without a debugging interface. To run the ROM Monitor a 8KB monitor program ((ROM<cpu_type>.HEX) has to be loaded to the target system together with the application program. This monitor program provides the debugging interface.
Expanded Configurations

The base systems can be expanded very flexible. The following examples should give you an impression of possible configurations.

**32 Bit ROM Monitor with Evaluation Board**

![Diagram of 32 Bit ROM Monitor with Evaluation Board]

**BDM Debugger with I/O Test System**

![Diagram of BDM Debugger with I/O Test System]

**BDM Debugger with 32 Bit EPROM Simulator and Evaluation Board**

![Diagram of BDM Debugger with 32 Bit EPROM Simulator and Evaluation Board]
Installation

In this section the installation for the ICD Debugger connected to the PC is described. The configuration of the ICD debug system is done in the file CONFIG.T32 which must reside in the TRACE32 system directory.

Software Installation

TRACE32 can be started with and without command line arguments. If you do not pass any command line arguments, the following default will be used:

- For the configuration file, the first available file from the following locations will be used:
  - The file specified by the environment variable `T32CONFIG`
  - The file `config.t32` from the working directory (from where TRACE32 was started).
  - The file `config.t32` from the TRACE32 system directory (usually C:\t32)

- After the start of a TRACE32 instance, the PRACTICE script `autostart.cmm` from the TRACE32 system directory will be executed, which then calls the following scripts:
  - `system-settings.cmm` (from the TRACE32 system directory, usually C:\t32)
  - `user-settings.cmm` (from the user settings directory, on Windows `%APPDATA%\TRACE32` or `~/.trace32` otherwise)
  - `work-settings.cmm` (from the current working directory)

If you pass command line arguments, you can adapt the start and the environment of TRACE32 to your project-specific needs. You can define a user-specific name and location for the configuration file and your own PRACTICE start-up script. In addition, you can pass user-defined parameters to your configuration file and to your start-up script.

The command line syntax for Windows, Linux, and Unix is as follows:

```
Format: t32m<arch>[-<x>] [<options>] [-c <config_file> [<c_args>]] [-s <startup_script> [<s_args>]]
```

- `c <config_file>`
  By default, TRACE32 expects to find the configuration file `config.t32` in the same folder as the TRACE32 executable. An error message is displayed if the configuration file `config.t32` does not exist.

  - `c <config_file>` allows you to define a user-specific name and location for the TRACE32 configuration file.

For information about the configuration options in the configuration file, type at the TRACE32 command line: `HELP.Index "config.t32"

**NOTE:**
- `-c` is case sensitive, i.e. `-C` results in an error.
- The name of the `<config_file>` must not start with a hyphen and must not contain any commas.
-s <startup_script>  
When a TRACE32 instance starts, the PRACTICE script `autostart.cmm` is executed, which then calls the following scripts:

- **system-settings.cmm** (from the TRACE32 system directory, usually C:\t32)
- **user-settings.cmm** (from the user settings directory: on Windows %APPDATA%\TRACE32 or ~/.trace32 otherwise)
- **work-settings.cmm** (from the current working directory)

With the command line option `-s <startup_script>` you can specify an additional PRACTICE script (*.cmm) which is automatically started afterwards.

**NOTE**: If you use command line option `-s <startup_script>` but don't have the file `autostart.cmm` in your TRACE32 system directory, only the file specified by the command line option `-s` will be executed.

If you don’t use the command line option `-s <startup_script>` and don’t have the file `autostart.cmm` either, TRACE32 will fall back to a legacy mode and execute the script `t32.cmm` from the working directory or from the TRACE32 system directory if the `t32.cmm` does not exist in the working directory.

| <arch> | Architecture, e.g. ARM in t32marm.exe stands for the ARM architecture. |
| <c_args> | Sequence of white-space separated arguments passed to the configuration file. |

The individual command line arguments are assigned to the configuration options in the configuration file using `${n}` where `n` is the number of the command line argument. The numbering starts at 1.

**Example**: `HEADER=${2}` means that the second `<c_arg>` is assigned to the configuration option `HEADER=`

**NOTE**: The `<c_args>` must not start with a hyphen and must not contain any commas.
<table>
<thead>
<tr>
<th><strong>&lt;options&gt;</strong></th>
<th>The following command line options are available for all architectures:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• <strong>--t32-help</strong> Lists all available command line options and suppresses the automatic execution of any PRACTICE script after starting TRACE32.</td>
</tr>
<tr>
<td></td>
<td>• <strong>--t32-help-diag</strong> Prints all found unknown command line options into the message <strong>AREA</strong> window.</td>
</tr>
<tr>
<td></td>
<td>• <strong>--t32-cmdline-quote-all</strong> Passes all parameters to the start-up script enclosed in double quotes.</td>
</tr>
<tr>
<td></td>
<td>• <strong>--t32-cmdline-quote-esc</strong> Passes all parameters to start-up script enclosed in double quotes. Additionally the command line option escapes already included double quotes in parameters, so that you get the same character sequence when accessed with <strong>PRACTICE.ARG()</strong>.</td>
</tr>
<tr>
<td></td>
<td>• <strong>--t32-area-size-lines=&lt;lines&gt;</strong> Sets the initial number of lines in the message <strong>AREA</strong> window.</td>
</tr>
<tr>
<td></td>
<td>• <strong>--t32-safestart</strong> Suppresses the automatic execution of any PRACTICE script after starting TRACE32.</td>
</tr>
<tr>
<td></td>
<td>• <strong>--t32-logautostart</strong> Internally executes the <strong>LOG.DO</strong> command to generate an autostart log in the temporary directory of TRACE32. The resulting autostart log file lists all PRACTICE scripts which were called during start-up of TRACE32. For an example and a description of the file name convention, see below.</td>
</tr>
<tr>
<td></td>
<td>• <strong>--t32-screen=&lt;screendriver&gt;</strong> Set the screen driver used to display all PowerView windows. Possible values area: auto, cde, qt, qt4, qt5</td>
</tr>
</tbody>
</table>

| **<s_args>** | Sequence of white-space separated arguments passed to the PRACTICE start-up script (*.cmm). All characters are permissible. However, if an **<s_arg>** is enclosed in quotation marks (e.g. "Hello"), then the **<s_arg>** must not be interrupted by another quotation mark. An **<s_arg>** that contains one or more blanks must be enclosed in quotation marks (e.g. "Hello World"). |

| **<x>** | The file name extension of the executable is optional on Windows. (no extension for Linux and Unix) |
Examples

- Example 1: Logging PRACTICE script calls during start-up of TRACE32 with --t32-logautostart
- Example 2: Passing command line arguments via a Windows shortcut to TRACE32
- Example 3: Passing command line arguments to a TRACE32 configuration file (*.t32)
- Example 4: Passing command line arguments to a PRACTICE start-up script (*.cmm)
- Example 5: Returning environment variables in a PRACTICE start-up script (*.cmm)

Example 1: Logging PRACTICE script calls during start-up of TRACE32 with --t32-logautostart

In this example, a Windows batch file (*.bat) sets the folder `C:\T32\project_c` as the working directory for TRACE32. The next script line starts the TRACE32 executable for ARMv8 by using the configuration file `config.t32` file and the PRACTICE start-up script `sieve.cmm` from the working directory.

The command line option `--t32-logautostart` causes the autostart log file to be generated.

```
C:
cd C:\T32\project_c
start C:\T32\bin\windows64\t32marm64.exe --t32-logautostart ^
-c config.t32 ^
-s sieve.cmm
```

The caret sign `^` serves as a line continuation character in Windows batch files (*.bat). White space characters after `^` are NOT permissible.

To access the autostart log file in TRACE32:
1. Execute your *.bat file to start TRACE32.
   The autostart log file is generated automatically.

2. Choose File menu > Automatic Scripts on Start > View Autostart log.
   The file opens in the TYPE window. The screenshot shows an example of an autostart log file:

   A The file name convention of the autostart log is described below.

   B The log file header tells you how the autostart log was generated. For alternatives, see “Logging the Call Hierarchy of PRACTICE Scripts” (practice_user.pdf).

   **File name convention of the autostart log file:** ~~~/<id>_t32m<arch>_<xx>_autostart.log

<table>
<thead>
<tr>
<th>~~~</th>
<th>Path prefix of the temporary directory of TRACE32. See also OS.PresentTemporaryDirectory().</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;id&gt;</td>
<td>ID of the PowerView GUI that was started. See also OS.ID().</td>
</tr>
<tr>
<td>t32m&lt;arch&gt;</td>
<td>Name of the PowerView executable (without file extension), e.g. “t32marm64”</td>
</tr>
<tr>
<td>&lt;xx&gt;</td>
<td>The instance number of the PowerView executable.</td>
</tr>
</tbody>
</table>
Example 2: Passing command line arguments via a Windows shortcut to TRACE32

This example shows how to pass TRACE32 command line arguments via a Windows shortcut to TRACE32. The command line arguments are:

- A user-defined configuration file called `config_usb.t32`
- A user-defined PRACTICE start-up script called `start.cmm`

**Operating System side**

```
Target: vs64\t32\arm\arm.exe config_usb.t32 -c start.cmm
Start in: C:\t32\project_x
```

**TRACE32 side**

```
TO: [vs64\t32\arm\arm.exe config_usb.t32 -c start.cmm]
```

A Path and name of a TRACE32 executable. In a default installation, all TRACE32 executables are located in `C:\t32\bin\<os>\`

B The option `-c <config_file>` allows you to define a user-specific name and location for the TRACE32 configuration file (`*.t32`).

C The option `-s <startup_script>` allows you to define a user-specific name and location for your PRACTICE start-up script (`*.cmm`).

D User-defined working directory. In the above `Properties` dialog box, the `Start in` text box sets the path for the `Target` text box - unless different paths are specified in the `Target` text box. This means for our example that the `t32\arm.exe` searches for the files `config_usb.t32` and `start.cmm` in `C:\t32\project_x`.

E TRACE32 system directory (by default `c:\t32`). It is specified during the installation of TRACE32. Normally, you do not need to change anything here.
Example 3: Passing command line arguments to a TRACE32 configuration file (*.t32)

The following example shows how to pass TRACE32 command line arguments from a batch file (*.bat) to the configuration file (*.t32). The command line arguments are:

- `<c_arg1>`: A user-defined window title for TRACE32
- `<c_arg2>`: A network folder path containing the pdf files of the TRACE32 online help

**Batch file start2_usb.bat:**

```bash
rem <c_arg1> <c_arg2>
start C:\T32\t32marm.exe -c config2_usb.t32 "Project X" "g:\TRACE32_pdf"
```

**TRACE32 configuration file config2_usb.t32:**

```plaintext
// Example of a TRACE32 configuration file
OS=
ID=T32002
TMP=C:\temp ; temporary directory for TRACE32
SYS=C:\T32 ; system directory for TRACE32
HELP=${2} ; help directory for TRACE32

PBI=USB

PRINTER=WINDOWS

SCREEN=
HEADER=TRACE32 ${1}
```

The values passed as command line arguments to the user-defined configuration file are now used by TRACE32:

- **Value of argument ${1}**
- **Value of argument ${2}**

TRACE32 now retrieves the pdfs of the help system from this network folder.

**NOTE:** The `help.t32` file of the online help must reside in the TRACE32 system directory (by default `C:\t32`).
Example 4: Passing command line arguments to a PRACTICE start-up script (*.cmm)

The following example shows how to pass TRACE32 command line arguments from an MS batch file (*.bat) to a PRACTICE start-up script (*.cmm). The command line arguments are:

- `<startup_script>`: A user-defined PRACTICE start-up script (*.cmm)
- `<s_args>`: The arguments passed to the PRACTICE start-up script are 0x1 and 0x2 and a string with a blank "Hello World!"

Batch file start3.bat:

```batch
rem                                           <startup_script> <s_args>
start C:\T32\t32marm.exe -c config3_usb.t32 -s start3.cmm 0x1 ^
                          0x2 ^
                          "Hello World!"
```

The caret sign ^ serves as a line continuation character in Windows batch files (*.bat). White space characters after ^ are NOT permissible.

PRACTICE start-up script start3.cmm:

```c
LOCAL &arg1 &arg2 &arg3 ;Declare local PRACTICE macros
&arg1=PRACTICE.ARG(0) ;Assign the values of the command line
&arg2=PRACTICE.ARG(1) ;arguments to the PRACTICE macros
&arg3=PRACTICE.ARG(2)
AREA.view                   ;Open an AREA.view window
PRINT "Arguments passed to the start-up script: " OS.PPF()
PRINT "arg1: " %COLOR.RED  "&arg1"
PRINT "arg2: " %COLOR.GREEN "&arg2"
PRINT "arg3: " %COLOR.NAVY  "&arg3"
```

The values passed as command line arguments to the PRACTICE start-up script are printed to the AREA.view window.
Example 5: Returning environment variables in a PRACTICE start-up script (*.cmm)

The first two tables show how to create the user-defined environment variable $T32P1$ and start TRACE32 under Windows and Linux. After starting TRACE32, the parameter of $T32P1$ is returned in the PRACTICE start-up script start4.cmm with the `OS.ENV()` function and printed to the TRACE32 message line, see 3rd table.

**Windows:** start4.bat

```
SET T32P1=C:\ain't this\a complicated\path\myprog.elf
START t32marm.exe -c config4_sim.t32 -s start4.cmm
```

**Linux:** dash/bash script

```
T32P1="/ain't this/a complicated/path/myprog.elf"
START t32marm.exe -c config4_sim.t32 -s start4.cmm
```

**TRACE32:** PRACTICE start-up script start4.cmm

```
;<your_code>

PRINT OS.ENV(T32P1) ;prints C:\ain't this\a complicated\path\myprog.elf
;...
```
PODBUS Interface Card

The default address of the PODBUS Interface Card is set to 350h. To change this address the jumpers and
the file CONFIG.T32 have to be modified.

Address Select Jumpers:

<table>
<thead>
<tr>
<th>JP0</th>
<th>JP1</th>
<th>JP2</th>
<th>Address (hex)</th>
<th>Address (dec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>on</td>
<td>on</td>
<td>on</td>
<td>350</td>
<td>848</td>
</tr>
<tr>
<td>off</td>
<td>on</td>
<td>on</td>
<td>250</td>
<td>592</td>
</tr>
<tr>
<td>on</td>
<td>off</td>
<td>on</td>
<td>260</td>
<td>608</td>
</tr>
<tr>
<td>off</td>
<td>off</td>
<td>on</td>
<td>280</td>
<td>640</td>
</tr>
<tr>
<td>on</td>
<td>on</td>
<td>off</td>
<td>300</td>
<td>768</td>
</tr>
<tr>
<td>off</td>
<td>on</td>
<td>off</td>
<td>330</td>
<td>816</td>
</tr>
<tr>
<td>on</td>
<td>off</td>
<td>off</td>
<td>340</td>
<td>832</td>
</tr>
<tr>
<td>off</td>
<td>off</td>
<td>off</td>
<td>390</td>
<td>912</td>
</tr>
</tbody>
</table>

JP3 must always be on.

The file CONFIG.T32 must have the following content:

```
PBI=
ADDRESS=816 ; the address must be entered in decimal format
```
Parallel PODBUS Converter

The Parallel PODBUS converter supports standard and EPP parallel ports. For standard configuration the file CONFIG.T32 must have the following content:

```
PBI= LPT1 ; or optional LPT2
```

NOTE: The relation between the LPTx names in the config.t32 file and port addresses is fixed. LPT1 is at 378, LPT2 at 278 and LPT3 at 3BC. This may not be the same as the names used in Windows. If you have a parallel port on your motherboard or graphics card it may be address 3BC, Windows may call it LPT1, but in the config file you must specify LPT3.

If the PC supports EPP mode the following CONFIG.T32 file is required:

```
PBI= LPT1 EPP ; or optional LPT2
```

EPP mode is the fastest operation mode for the PODBUS interface. It should be used wherever possible.

At common PC’s the parallel port mode can be selected in the BIOS. The BIOS must be switched to EPP, EPP V1.7 or EPP V1.9. Some PC’s require special tools to modify the parallel port mode. Please ask your PC manufacturer.
Serial Line Configuration

Modifications to the CONFIG.T32 configuration file:

```
PBI=COM2 baudrate=38400
```

Using Multiple Devices on one PODBUS

One PODBUS can control multiple independent devices. Each device or group can be controlled by a different user interface. The definition is made in the config.t32 file:

```
PBI=
USE=1001
```

The bitmask defines which devices (PowerDebug, PowerTrace, PowerProbe, ...) are controlled by the host application program. A ‘1’ means control the device, a ‘0’ means skip the device. The leftmost bit controls the first device on the bus i.e. the device with the host connection (USB, ETH).

Normally a use mask will contain a single set bit e.g. “1000”, indicating that only one of the devices in the PODBUS is used by the respective program (e.g. PowerView ARM frontend software uses a PowerTrace device to access the target).

When using PowerView to control two devices the use mask will contain two ‘1’ bits. An example for this is a PowerDebug device and a PowerProbe device which are used to record signals corresponding to a debug session of PowerDebug. Using this mechanism it is possible to combine a debugger and a PowerProbe device, independently of their position in the PODBUS chain.

**NOTE:** PowerDebug modules without USB or with USB 1.1 only are special in the sense that it must be considered as TWO devices (internally it combines multiple functions that were historically located in separate boxes). Therefore the PowerDebug device is represented by either ‘00’ or ‘11’ in the use mask.
Multicore debugging in general means that there are multiple cores on one chip which use the same JTAG interface. In contrary, if there is more than one chip with different JTAG interfaces to be debugged simultaneously, we use the term multiprocessor debugging.

The following chapters describe which hardware and licenses are needed and how to set up the TRACE32 software for multicore debugging. For detailed core specific settings, please refer to the particular Processor Architecture Manuals. In addition, there is a number of PRACTICE sample scripts (*.cmm) for different multicore systems available in the `~/demo/` folder or can be received from LAUTERBACH support (for contact possibilities, see www.lauterbach.com).

There are two general ways for multicore debugging:

- single device solution: one debug module (debug box) for several cores
- multi device solution: several debug modules (debug boxes), one for each core

In any case, each core has its own application software on the host PC, but those applications use different ways to access the LAUTERBACH Debug Hardware, and, in the end, their particular core.

### Tool Configuration for Single Device Solution

When using single device solution, the applications do not care about availability of the joint JTAG interface, the different accesses are arranged by a JTAG handler. The application executables (t32marm and t32m320 in the diagram above) register themselves at the JTAG handler of the common Power Debug Module. Therefore, the configuration files (default file name: config.t32) of the applications have to contain `core=` settings (as specified below).

The master application (those belonging to the base license of the debug cable) has to be started first to install the proper JTAG handler. Nevertheless, the setting to define which core is addressed actually is done later on.
The multi device alternative demands self-administration of the access to the joint JTAG interface for each application. The application reserves the JTAG port for usage, and releases it afterwards transferring its Debug Module into tristate mode. The applications (t32marm and t32m320 in diagram above) have to be configured to use their own separate debug box. Therefore, the configuration files (default file name: config.t32) of the applications have to contain `use=` settings (as specified below).

In this case it does not matter which application is started first.

As well as in single device solution, the setting to define which core is addressed at the end is done later on.
The following configuration examples provide an overview on usable LAUTERBACH debugging hardware for multicore debugging. To see which multicore processor ICs are supported by specific configurations, please contact LAUTERBACH sales department or local distributor.

**Configuration 1: Single PowerDebug-Module**

Description: The TRACE32 debugging software running on the Host-PC uses a joint JTAG interface to communicate with the cores on the multicore-IC. With Power Debug Ethernet Module the configuration looks likewise, except of interposed hub and ethernet connection at the host PC.
Description: The TRACE32 debugging software running on the Host-PC uses a joint JTAG interface to communicate with the cores on the multicore-IC. If several cores on the target provide trace information it is possible to select which core is accessed by the Trace Preprocessor. If there are several trace ports on the target, refer to configuration 5.

Using the USB connector of the Power Trace Ethernet module, the configuration looks likewise (without the Ethenet hub of course).
Configuration 3: Two (or more) Power Debug Modules

Description: The TRACE32 debugging software running on the host-PC uses separate debug boxes to communicate with the cores on the multicore-IC. To do this, one debug cable for each core is necessary. Some target boards already provide several JTAG connectors that represent a single one actually, in that case the y-adapter is not needed.

The number of Power Debug Modules needed depends on the number of cores to be debugged.
Configuration 4: One PowerTrace- and one or more Power Debug Modules

Description: The TRACE32 debugging software running on the Host-PC uses separate JTAG interfaces to communicate with the cores on the multicore-IC. To do this, one debug cable for each core is necessary. If several cores on the target provide trace information it is possible to select which core is accessed by the Trace Preprocessor. If there are several trace ports on the target, refer to configuration 5.

Some target boards already provide several JTAG connectors that represent a single one actually, in that case the y-adapter is not needed.

The number of Power Debug Modules needed depends on the number of cores to be debugged.
Description: The TRACE32 debugging software running on the Host-PC uses separate debug boxes to communicate with the cores on the multicore-IC. To do this, one debug cable for each core is necessary. Some target boards already provide several JTAG connectors that represent a single one actually, in that case the y-adapter is not necessary.

The number of additional Power Debug Modules needed depends on the number of cores to be debugged.
For multicore debugging, it is necessary to obtain the licenses for the particular cores. In case of configurations 3, 4 and 5 (see chapter Hardware Configuration for Multicore Debugging), the licenses are present in the respective debug cables (e.g. base licenses for each core). In contrast, the configurations 1 and 2 require all necessary licenses to be present in one debug cable. Those multicore debug cables always consist of the so-called base license and additional A- or X-license(s) according to the further core(s). In case there are only several equal cores of the same type on the chip, the so-called multicore license has to be added to the base license. If there are already several different core license within one cable, this namely permits multicore debugging (if supported by software).

The on-chip-trace licenses like ARM-ETB license do not count as a core license, to debug several same ARM cores the multicore license LA-7960X is required.

To learn which license combinations are possible please contact LAUTERBACH sales department or distributor.
Install Debugger Software for Multicore Debugging

The basic way to install the software on the different operating systems is described in “TRACE32 Installation Guide” (installation.pdf).

For multicore debugging, all the debuggers for the different cores have to be installed.

![CPU selection](image.png)

selection of several debuggers by left-click
General

At multicore debugging, each core belongs to one instance of the TRACE32 software, i.e., there are as many TRACE32 instances resp. executables running as cores are to be debugged simultaneously. The appropriate executable to be started is titled according to the name of the core family, and can be checked in the properties of the installed debugger shortcuts.

```
t32m<corefamily>.exe ; on Windows
t32m<corefamily> ; on Linux or workstations
```

After installation, the particular predefined config files (config.t32) are for single core debugging only. Therefore it is necessary to add certain settings as described in the following. It is recommended to copy the existing config.t32 into a new file, the filename is freely chooseable, and can be provided when starting the executable.

**Config file settings for single device solution**

At single device solution, all TRACE32 applications use the same debug box:

![Diagram of Host PC, Power Debug Module, and Target](image)

To specify this, we use `core=` setting within PBI section:

```
PBI=
CORE=1 ; within config file of first core
...

PBI=
CORE=2 ; within config file of 2nd core
...

... ; possibly more config files
```
At multi device solution, each TRACE32 application uses its own debug box:

To specify this, we use the `use=` setting within PBI section:

```
PBI= ; within config file of first core
USE=10

PBI= ; within config file of 2nd core
USE=01
```

(see also chapter Using Multiple Devices on one PODBUS). In addition, it is necessary to specify the `ID=` setting within OS section (the ID is used to store PRACTICE command history and associated source code files within TMP directory, therefore it is alternatively possible to use same ID and assign separate TMP directories to each TRACE32 instance):

```
OS= ; within config file of first core
ID=T32_CORE1
...

OS= ; within config file of 2nd core
ID=T32_CORE2
...
```

... ; possibly more config files
To use the start/stop synchronization between the different core debuggers, the InterCom port settings are necessary (PORT= setting within the IC= section, see also chapter Start Stop Synchronisation):

```plaintext
IC=NETASSIST
PORT=20001 ; within config file of first core

IC=NETASSIST
PORT=20002 ; within config file of 2nd core

...

; possibly more config files
```

**Example: Static config files for each TRACE32 instance**

First TRACE32 instance:

```plaintext
; The configuration file for the ARM9 core
; =================================================

; 1. Specify the ARM9 core as the first core.

PBI=
CORE=1
USB

; 2. Specify an ID for the ARM9 debugger.

OS=
SYS=..
TMP=C:\temp
ID=T32ARM9
HELP=H:\T32NEW\pdf

; 3. Specify a better core identification for the TRACE32 window.

SCREEN=
HEADER=TRACE32-ARM Multicore ; Header for the
T32 InterCom for Start/Stop synchronisation

IC=NETASSIST
PORT=20002
```

In the example above the call syntax would be (assuming the config filename is “config_core1.t32”):

```plaintext
t32marm -c config_core1.t32
```
Second TRACE32 instance:

; The configuration file for the TMS320C55X core
;===============================================================================

; 1. Specify the TMS320C55X core as the second core.

PBI=
CORE=2
USB

; 2. Specify an ID for the TMS320C55X debugger.

OS=
SYS=.
TMP=C:\temp
HELP=H:\T32NEW\pdf
ID=T32C55x
; Each debugger creates copies of source file into the TMP directory. The
; specified ID allows each
; debugger to identify its source files.

; 3. Specify a better core identification for the TRACE32 window.

SCREEN=
HEADER=TRACE32-C55x Multicore ; Header for the TRACE32 window
PALETTE 1 = 255 255 223 ; Yellow background color

; T32 InterCom for Start/Stop synchronisation
IC=NETASSIST
PORT=20001

t32m320 -c config_core2.t32
Example: Generic config file for each TRACE32 instance

Besides, there is the possibility to use just one generic template config file for all cores. The particular settings are passed as parameter. Example for generic config file:

```plaintext
IC=NETASSIST
PORT=${1}

; Environment variables
OS=
ID=T32${1}
TMP=C:\temp
SYS=\.
HELP=h:\t32new\pdf

PBI=
${3}
${4}
${5}
${6}

; Printer settings
PRINTER=WINDOWS

; Screen fonts
SCREEN=
FONT=SMALL
HEADER=TRACE32 ${2} Multicore 0${3} ${4} ${5} ${6}
```

The template variables ${1} to ${6} are replaced textually by corresponding calling parameters. In that way, it is possible to select even the connection method (Parallel, USB or Ethernet) by parameters.

In this generic example the call syntax could be (assuming the config filename is “config_mc.t32”):

```plaintext
; Start ARM executable as 1st core via USB and InterCom port 20001
 t32marm -c config_mc.t32 20001 ARM USB CORE=1

; Start TMS320 executable as 2nd core via USB and InterCom port 20002
t32m320 -c config_mc.t32 20002 C55x USB CORE=2
```
Those calls are also possible through shortcut:

```
; Start ARM executable as 1st core via ETH and InterCom port 20001
 t32marm -c config_mc.t32 20001 ARM NET NODE=10.2.2.234 CORE=1

; Start TMS320 executable as 2nd core via ETH and InterCom port 20002
 t32m320 -c config_mc.t32 20002 C55x NET NODE=10.2.2.234 CORE=2
```
Example: start of further TRACE32 instances by first instance (with generic config file)

Using the **OS** command of TRACE32 PRACTICE language, it is possible to start further instances of TRACE32 (see also IDE Reference Guide). This offers the possibility to start multiple core debuggers from one PRACTICE script file (*.cmm). In combination with a generic config file, this is a very smart way to set up multicore debugging.

```plaintext
; start tpu executable as 2nd core via USB and InterCom port 20002
OS t32mtpu -c config_mc.t32 20002 eTPU_A USB CORE=2
```

For a description of the parameters refer to chapter **Generic config file for each TRACE32 instance**, see above.

The very first instance, however, has to be started in common way (by batch file or shortcut), as described in chapter **Generic config file for each TRACE32 instance**, see above:

```plaintext
; start PPC executable as 1st core via USB and InterCom port 20001
t32mppc -c config_mc.t32 20001 PowerPC USB CORE=1

; start PPC executable as 1st core via ETH and InterCom port 20001
t32mppc -c config_mc.t32 20001 PowerPC NET NODE=10.2.2.234 CORE=1
```
Complete example PRACTICE script for generic starting of multiple instances on Windows, using MPC5554 (with E200 and two eTPU cores), in which it is dynamically selectable to use USB or Ethernet connection:

```
; Sample eTPU setup script

; start eTPU debugger software and check InterCom
=============================
&nodename=NODENAME()

&etpu1="InterCom localhost:20002"
&etpu2="InterCom localhost:20003"

; Test is etpu software is already running or not
IF !InterCom.PING(localhost:20003)
{
  IF "&nodename"=="
  {
    print "assuming connection is USB"
    OS t32mtpu -c config_mc.t32 20002 eTPU_A USB CORE=2
    ; wait until software is started
    InterCom.WAIT localhost:20002
    OS t32mtpu -c config_mc.t32 20003 eTPU_B USB CORE=3
  }
  ELSE
  {
    PRINT "connection is ETH"
    OS t32mtpu -c config_mc.t32 20002 eTPU_A NET NODE=&nodename
    PACKLEN=1024 CORE=2
    ; wait until software is started
    InterCom.WAIT localhost:20002
    OS t32mtpu -c config_mc.t32 20003 eTPU_B NET NODE=&nodename
    PACKLEN=1024 CORE=3
  }
}; wait some time until the software finished booting
; if you get and InterCom timeout error on the following framepos
; commands
; increase time
WAIT 3.s

; multicore settings
=================================================================
InterCom.WAIT localhost:20002
InterCom.WAIT localhost:20003

&etpu1 SYStem.CONFIG.CORE 2.
&etpu2 SYStem.CONFIG.CORE 3.

)```
In the example above, the other TRACE32 instances are accessed from first instance by

```
InterCom localhost:<port_number>
```

command, which is stored into local PRACTICE variables (`&etpu1` and `&etpu2`).

**Automatic setup of config files**

(for Windows only)

The usage of the LAUTERBACH add-on **T32Start** is described in “**T32Start**” (app_t32start.pdf).
The installation of the TRACE32 software and the config files is finished now. All TRACE32 applications access the same JTAG port, either by single device or by multi device solution. Now we have to tell the TRACE32 applications which of the cores they should debug.

The communication to the specified core is done not until the `SYStem.Up` command, and for preparation (besides the proper processor selection) there might be some additional multicore settings to be done.

There is a number of PRACTICE sample scripts for different processors available in the `~/.demo/…` folder or directly from LAUTERBACH support (for contact possibilities, see www.lauterbach.com), which may assist you in setting up multicore debugging.

Basically, there are two different ways to address the particular core:

- daisy chaining
- multiplexing
Daisy Chain Settings

For those processors that have multiple so-called TAP (=Test Access Port) controllers, eg. one for each core, it is necessary to provide the TRACE32 applications with daisy chain settings.

The JTAG port of the individual cores are arranged serially (daisy chained). The TAP controller controls the scanning of data in the instruction register and the data register of the JTAG architecture.

- The instruction register has a specific width for each architecture
- The width of the data register depends on the instruction used (always 1 bit if the instruction register contains the BYPASS instruction)

According to the JTAG convention there is an option to generate sequences for the scan chain that ensure that individual TAP controllers behave neutrally (BYPASS). To address a specific core the debugger must therefore be configured in such a way that, when generating the scan chain, BYPASS sequences are created for all TAP controllers before and after this core. The associated settings are either determined by selection of the processor (if fixed) and/or can be modified in the SYStem.CONFIG.state window.

To access the SYStem.CONFIG.state window, do one of the following:

- Choose CPU menu > System Settings, and then click the CONFIG button.
- At the TRACE32 command line, type:

  SYStem.CONFIG.state

  Shows the state of the MULTICORE setting.
IRPRE

<number> of instruction register bits of all cores in the JTAG chain between the current core and the TDO signal.

IRPOST

<number> of instruction register bits of all cores in the JTAG chain between TDI signal and the current core.

DRPRE

<number> of cores resp. TAP controllers in the JTAG chain between the current core and the TDO signal (one data register bit per TAP controller which is in BYPASS mode).

DRPOST

<number> of cores resp. TAP controllers in the JTAG chain between the TDI signal and the current core (one data register bit per TAP controller which is in BYPASS mode).

Slave

All debuggers except one must have this option active. Only one debugger - the master - is allowed to control the signals nTRST and nSRST (nRESET).
In case of multi device solution, the following additional settings have to be made:

**TriState**
If more than one debug box shares the same JTAG port, this option is required. The debugger switches after each JTAG access to tristate mode. Then other debuggers can access the port.

**TAPState**
(default: 7 = Select-DR-Scan). This is the state of the TAP controller when the debugger switches to tristate mode. All states of the JTAG TAP controller are selectable. The default value is recommended.

**TCKLevel**
Level of TCK signal when all debuggers are tristated (0 in case of pull-down on target, 1 if pull-up).

In case the daisy chain settings are preconfigured by processor selection, they are displayed within `SYStem.CONFIG.state` window:

The example above shows both core and ARM-ETB daisy chain settings, at which they both are located within same daisy chain sequentially. If there are additional TAP controllers beyond the selected multicore processor located on the target board (e.g. several daisy-chained multicore chips together on one target board), the necessary values have to be entered into edit controls.
For some processors (like STARCORE or NIOS-II) it is necessary to specify individually which core the current TRACE32 instance should debug.

| SYStem.CPU < multicore_processor > | ; select multicore processor type |
| SYStem.CONFIG.CORE 1 | ; specify current TRACE32 instance |
| SYStem.CONFIG.CORE 1 | ; as core 1 debugger |

| SYStem.CPU < multicore_processor > | ; select multicore processor type |
| SYStem.CONFIG.CORE 2 | ; specify current TRACE32 instance |
| SYStem.CONFIG.CORE 2 | ; as core 2 debugger |

The **SYStem.CONFIG.CORE** setting does technically NOT refer to **CORE=** setting within config file. Of course, it is recommended to use same core number.
Start Stop Synchronisation

**Settings**

To use the Start/Stop Synchronisation, the InterCom port settings in the IC= section of the config file(s) are required (refer to chapter Setup and Start of TRACE32 Software.

The synchronisation itself is done by **SYnch** commands, according to the following steps:

1. Set up the **SYnch** command for first core as master core

   ```
   SYnch.state ;Display the current settings of the SYnch command
   ```

   ```
   SYnch.Connect Localhost:20001 [<further_ports>]
   ```

<table>
<thead>
<tr>
<th><strong>SYnch Settings</strong></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Connect</strong></td>
<td>Establish a connection to the debugger attached to the defined communication port(s). Several debuggers resp. ports can be specified, separated by space.</td>
</tr>
<tr>
<td><strong>MasterGo ON</strong></td>
<td>If the program execution is started, the program execution for all other processors which have SlaveGo ON is also started.</td>
</tr>
<tr>
<td><strong>MasterBrk ON</strong></td>
<td>If the program execution is stopped, the program execution for all other debuggers which have SlaveBrk ON is also stopped.</td>
</tr>
<tr>
<td><strong>MasterStep ON</strong></td>
<td>If an asm single step is executed, all processors which have SlaveStep ON will also asm single step.</td>
</tr>
</tbody>
</table>
It is optionally possible to set both master and slave settings ON, to let this core react on other core events, too.

| ![Warning Icon] | The PORT= value within IC section of the config file of the current core has to match the value(s) of the SYNch.Connect command(s) of the corresponding debugger(s), while the SYNch.Connect value(s) of the current core has(have) to match those values from the config files of the other core debugger(s). In other words, the PORT= value(s) within config file(s) are server port numbers, while the SYNch.Connect values are client port numbers (see also chapter Setup and Start of TRACE32 Software). |

2. Set up the SYNch command for the further cores.

SYnch.Connect Localhost:20000 [<further_ports>]

### SYNch Settings

<table>
<thead>
<tr>
<th>Connect</th>
<th>Establish a connection to the debugger attached to the defined communication port</th>
</tr>
</thead>
<tbody>
<tr>
<td>SlaveGo ON</td>
<td>The program execution is started, if a processor with MasterGo ON starts its program execution.</td>
</tr>
<tr>
<td>SlaveBrk ON</td>
<td>The program execution is stopped, if a processor with MasterBrk ON stops its program execution.</td>
</tr>
<tr>
<td>SlaveStep ON</td>
<td>A asm single step is performed, if a processor with MasterStep On performs an asm single step.</td>
</tr>
</tbody>
</table>

It is optionally possible to set both master and slave settings ON, to let other cores react on current core events, too.
Summary (Example with one master and two slave core debuggers):

; Complete script has to be run on TRACE32 instance of first core

; SYNCH commands for 1st core
SYNCH.RESet
SYNCH.Connect Localhost:20001 Localhost:20002
SYNCH.MasterGo ON
SYNCH.MasterBRK ON
SYNCH.MasterStep ON

; SYNCH commands for 2nd core
InterCom.execute Localhost:20001 SYNCH.RESet
InterCom.execute Localhost:20001 SYNCH.Connect Localhost:20000 Localhost:20001
InterCom.execute Localhost:20001 SYNCH.SlaveGo ON
InterCom.execute Localhost:20001 SYNCH.SlaveBRK ON
InterCom.execute Localhost:20001 SYNCH.SlaveStep ON

; SYNCH commands for 3rd core
InterCom.execute Localhost:20002 SYNCH.RESet
InterCom.execute Localhost:20002 SYNCH.Connect Localhost:20000 Localhost:20001
InterCom.execute Localhost:20002 SYNCH.SlaveGo ON
InterCom.execute Localhost:20002 SYNCH.SlaveBRK ON
InterCom.execute Localhost:20002 SYNCH.SlaveStep ON

ENDDO
Result of start/stop synchronization

The master starts the program execution

Both slaves are started
The master stops the program execution.

Both slaves are stopped.
There is a time delay between reaction of different cores. The reaction time of the appended slave core(s) depends on the technical realization of the synchronization. In worst case, this is done by software, i.e. the master debugger informs the slave debugger(s) via socket communication on the host about specified events. In best case, the synchronization takes place directly on the processor, and the in between solutions are PODBUS trigger signal or to connect certain PINs, if available, on the JTAG connector or the board. Which solution is offered by the particular multicore debugging environment is depending on the chip and/or board hardware. The LAUTERBACH Debuggers always use the fastest available possibility. The software synch mechanism starts all cores simultaneously by TRACE32 software, either by socked communication (slow) or internal synchronized within TRACE32 (fast). The fast method is only implemented for some processors, while the software method by socket is always available.

Special on STEP-Signal: By selecting MasterStep/SlaveStep ON, only the asm single steps are synchronized. The HLL steps are published by master debugger as a sequence of (asm)step-go-break (or just go-break), which means the slave debugger(s) do(es) not really perform its own HLL single step.

<table>
<thead>
<tr>
<th>Solution:</th>
<th>Software (slow)</th>
<th>Software (fast)</th>
<th>PODBUS Trigger</th>
<th>Hardware (JTAG or Board)</th>
<th>Chip Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delay time:</td>
<td>1-5 ms</td>
<td>~10 µs</td>
<td>100-300 ns</td>
<td>~10 ns</td>
<td>0-10 ns</td>
</tr>
</tbody>
</table>
Here is a list of steps to set up and start the LAUTERBACH debugger for multicore debugging:

- Choose debug hardware (see *Hardware Configuration for Multicore Debugging*).
- Install suitable debugger software (see *Install Debugger Software for Multicore Debugging*).
- Setup config files (see *Setup and Start of TRACE32 Software*).
- Start executables with particular config files (see *Setup and Start of TRACE32 Software*).
- Run core specific script (see *TRACE32 Multicore Configuration*).
Multiple ICD Debuggers can be started and stopped synchronously. The time delay depends on the implementation of the debug port. On a 68332 the start time delay is about 10us and the stop delay about 5us. The coupling of the different systems is made by the **SYnch** command.

**NOTE:** This command requires that the InterCom system is configured in the host. The synchronous break can be made with the **Trigger** command. The following example shows the configuration files and commands required for a system where one master controls two slaves:

Configuration file for master:

```
PBI=
USE=100
IC=NETASSIST
PORT=20000
```

Configuration file for slave 1:

```
PBI=
USE=010
IC=NETASSIST
PORT=20001
```

Configuration file for slave 2:

```
PBI=
USE=001
IC=NETASSIST
PORT=20002
```

Startup commands for master:

```
sy.connect localhost:20001 localhost:20002
sy.mastergo on
sy.on
trigger.out busa on
```

Startup commands for slaves:

```
sy.slavego on
sy.on
trigger.set busa on
```
To start TRACE32, do one of the following:

- Click the Windows Start button, and then select TRACE32 ...

- Navigate to c:\t32\bin\<os_name>\<32_or_64_bit_version>, and then double-click the required executable file.

  The executable file which has to be started is named T32M<cpu_type>.exe (e.g. T32M68.EXE, T32MPPC.EXE, T32MARM.EXE…).

To exit from TRACE32, do one of the following:

- At the TRACE32 command line, type: QUIT

- Choose File menu > exit.
Installation as TRACE32-ICE Extension

Software Installation

To install the ICD Debugger together with the TRACE32-ICE see the Installation Guide of the TRACE32 System. In addition to the normal TRACE32 emulator software the following files may be required:

- `mccm<cpu_type>.t32`; Driver for JTAG/BDM Debugger or ROM Monitor, CPU specific (for old and new controller)
- `mcpm<cpu_type>.t32`; CPU specific (for old and new controller)
- `mccesi.t32`; Driver for the EPROM Simulator
- `mcpesi.t32`;
- `mccitim.t32`; Driver for the PowerProbe
- `mcpitim.t32`;
- `mccstg.t32`; Driver for the Stimuli Generator
- `mcpstg.t32`;
- `mcctim.t32`; Driver for the Timing Analyzer (TA32)
- `mcptim.t32`;

A configuration file “software.t32” may be created for special configurations. This file is not required for standard configurations.

<table>
<thead>
<tr>
<th>Format:</th>
<th><code>&lt;driver&gt;&lt;cpu_type&gt;[&lt;usemask&gt;]</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;driver&gt;</code>:</td>
<td>ICD SIM</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>&lt;cpu_type&gt;</code></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>08</td>
<td>for 68HC0x</td>
</tr>
<tr>
<td>11</td>
<td>for M68HC11</td>
</tr>
<tr>
<td>12</td>
<td>for M68HC12</td>
</tr>
<tr>
<td>166</td>
<td>for C166</td>
</tr>
<tr>
<td>16</td>
<td>for M68HC16</td>
</tr>
<tr>
<td>186</td>
<td>for i86</td>
</tr>
<tr>
<td>M32R</td>
<td>for M32R</td>
</tr>
<tr>
<td>51</td>
<td>for i8051</td>
</tr>
<tr>
<td><code>&lt;cpu_type&gt;</code></td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
</tr>
<tr>
<td>56</td>
<td>for DSP56k</td>
</tr>
<tr>
<td>68</td>
<td>for M68k / ColdFire</td>
</tr>
<tr>
<td>78K</td>
<td>for 78K</td>
</tr>
<tr>
<td>96</td>
<td>for i196</td>
</tr>
<tr>
<td>2000</td>
<td>for TMS320C2xxx</td>
</tr>
<tr>
<td>5000</td>
<td>for TMS320C5xxx</td>
</tr>
<tr>
<td>6000</td>
<td>for TMS320C6xxx</td>
</tr>
<tr>
<td>ANDE</td>
<td>for AndeStar</td>
</tr>
<tr>
<td>APEX</td>
<td>for APEX</td>
</tr>
<tr>
<td>APS</td>
<td>for APS3</td>
</tr>
<tr>
<td>ARC</td>
<td>for ARC</td>
</tr>
<tr>
<td>ARM64</td>
<td>for ARM (64-bit)</td>
</tr>
<tr>
<td>ARM</td>
<td>for ARM (32-bit)</td>
</tr>
<tr>
<td>AVR</td>
<td>for AVR32</td>
</tr>
<tr>
<td>BA</td>
<td>for Beyond</td>
</tr>
<tr>
<td>BF</td>
<td>for Blackfin</td>
</tr>
<tr>
<td>CEVA</td>
<td>for CEVA-X</td>
</tr>
<tr>
<td>CORE</td>
<td>for M.CORE</td>
</tr>
<tr>
<td>GTM</td>
<td>for GTM</td>
</tr>
<tr>
<td>H83</td>
<td>for Renesas-H8</td>
</tr>
<tr>
<td>MIPS64</td>
<td>for MIPS (64-bit)</td>
</tr>
<tr>
<td>MIPS</td>
<td>for MIPS (32-bit)</td>
</tr>
<tr>
<td>MB</td>
<td>for MicroBlaze</td>
</tr>
<tr>
<td>MDSP</td>
<td>for MMDSP+</td>
</tr>
<tr>
<td>NIOS</td>
<td>for Nios-II</td>
</tr>
<tr>
<td>PPC64</td>
<td>for PowerPC (64-bit)</td>
</tr>
<tr>
<td>PPC</td>
<td>for PowerPC (32-bit)</td>
</tr>
<tr>
<td>QDSP6</td>
<td>for Hexagon</td>
</tr>
<tr>
<td>RX</td>
<td>for Renesas-RX</td>
</tr>
<tr>
<td>SC</td>
<td>for StarCore</td>
</tr>
<tr>
<td>SH</td>
<td>for SH</td>
</tr>
<tr>
<td>&lt;cpu_type&gt;</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>M430</td>
<td>for MSP430</td>
</tr>
<tr>
<td>SPT</td>
<td>for SPT</td>
</tr>
<tr>
<td>TC</td>
<td>for TriCore</td>
</tr>
<tr>
<td>TEAK</td>
<td>for CEVA-Teak</td>
</tr>
<tr>
<td>TPU</td>
<td>for eTPU</td>
</tr>
<tr>
<td>UBI</td>
<td>for Ubicom32</td>
</tr>
<tr>
<td>V800</td>
<td>for V850/RH850</td>
</tr>
<tr>
<td>X86</td>
<td>for Intel Arch. (32-bit)</td>
</tr>
<tr>
<td>X64</td>
<td>for Intel Arch. (64-bit)</td>
</tr>
<tr>
<td>XA</td>
<td>for XA51</td>
</tr>
<tr>
<td>XTEN</td>
<td>for Xtensa</td>
</tr>
<tr>
<td>Z80</td>
<td>for Z80</td>
</tr>
<tr>
<td>ZSP</td>
<td>for ZSP</td>
</tr>
</tbody>
</table>
Available Device Prompts

The TRACE32 system includes a number of devices, e.g. in circuit emulator, timing analyzer, ICD Debugger. Each device has its own command set.

To work with the device-specific command set, the device must be selected by entering the device identifier ending with a double colon in the command line. The currently selected device is displayed by the prompt of the command line. At the ICD debug system the following prompts are available.

**B:** JTAG/BDM Debugger or ROM Monitor. This is the prompt which will typically be used for debugging. Please note that if you use the ESI as ROM Monitor or as emulation memory together with an ICD module, all mapping must be done on the B: prompt.

**ESI:** This prompt is only required if you want to use the ESI as pure EPROM simulator, without any debug capabilities.

**PP:** Used to control the PowerProbe

**STG:** Used to control the Stimuli Generator

**T:** Used to control the Timing Analyzer (TA32)
ICD Commands and Procedures

The start-up procedures and CPU specific commands for the ICD Debugger are described in the CPU-specific documentation section.

You can look up the function and syntax for all general commands in the Emulator Reference for the TRACE32 system.

Specific commands and procedures for the ICD debug system are described in the following chapters.
Mapping the EPROM Simulator

One EPROM simulator is used to simulate two 8 bit or one 16 bit EPROM. To simulate EPROMs with a bigger bus size, two or more EPROM simulator can be put together.

Whenever the EPROM simulator is used the configuration of the target EPROM's must be specified in the ICD Debugger software with the MAP command.

The reproduction of the EPROM configuration is done as follows:

1. Reset the mapping system (MAP.RESet command).
2. Map the EPROM simulator within the specified range (MAP.ROM command).
3. Set the EPROM bus size (MAP.BUSXX command). The default bus size is 8 bit.
4. Set the EPROM width (MAP.BYTE or MAP.WORD command).
5. Verify your configuration with the MAP.List command.

```plaintext
; maps one 8K x 8 EPROM
; 8 bit adapter low
b:
  map.res
  map.rom 0x0--0x01fff

; maps two 8K x 8 EPROMS in parallel
; 8 bit adapter low and high
b:
  map.res
  map.rom 0x0--0x03fff
  map.bus16 0x0--0x03fff

; maps one 4K x 16 EPROM
; 16 bit adapter
b:
  map.res
  map.rom 0x0--0x01fff
  map.bus16 0x0--0x01fff
  map.word 0x0--0x01fff
```
; maps one paged addressed EPROM with 4 pages (4 x 16K x 8)
; 8 bit adapter low
b:
map.res

map.rom 0x00000--0x03fff
map.rom 0x04000--0x07fff
map.rom 0x08000--0x0bfff
map.rom 0x0c000--0x0ffff

map.page 0 0x00000--0x03fff
map.page 1 0x04000--0x07fff
map.page 2 0x08000--0x0bfff
map.page 3 0x0c000--0x0ffff

; maps two fragments in one 8 bit EPROM
; 8 bit adapter low
b:
map.rom 0x0--0x7fff
map.rom 0x10000--0x17fff

map.frag 1 0 0x0--0x7fff
map.frag 1 0x8000 0x10000--0x17fff

; relocates one 128K x 8 EPROM mapped from 0--1fffff to 40000
; while the system is up
b:
map.relocate 0x40000 0x0--0x1fffff

; maps four 64K x 8 EPROMs for a bus size of 32 bit
; two EPROM simulators
; for each 8 bit adapter high and low
map.rom 0x0--0x3fff
map.bus32 0x0--0x3fff

; maps two 64K x 16 EPROMs for a bus size of 32 bit
; two EPROM simulators
; for each 16 bit adapter
map.rom 0x0--0x3fff
map.bus32 0x0--0x3fff
map.word
The ICD Debugger uses software breakpoints. For that reason only Program, HLL and Spot Breakpoints are available.

If the CPU provides hardware breakpoints, read and write breakpoints can be used. For more information see the CPU specific section.

If the CPU provides hardware execution breakpoints, they will be used when a breakpoint is set in an area marked with `MAP.ReadOnly`.

Manual break for ROM Monitors can be accomplished in different ways:

- Use of the NMI line
- Use of the Serial Line Interrupt (Serial Monitors)
- Polling of serial line or ESI by target
- NIL character send by virtual terminal

See the Processor Architecture Manual and monitor source code for your target for more details.
The eXeption commands are used to adapt and connect the RES and NMI signals generated by the ROM Monitor software to the target CPU.

On each adapter for the EPROM simulator there are three extra pins for GND, NMI and RES. NMI and RES are outputs from a HTC Schmitt trigger.

The RESet pin of one adapter can be connected to the reset pin of the CPU to give the ROM Monitor program the possibility to control the CPU reset. The polarity of the reset can be adapted to the CPU type by the eXeption.RESetPOL command.

If the RESet pin is connected to the CPU, the CPU held in reset, while the system is down.

The NMI pin of one adapter can be connected to the NMI pin of the CPU to manually break a program running on the target system. The polarity of the NMI can be adapted to the CPU type by the eXeption.NMIPOL command. The connection is enabled by eXeption.NMIBREAK ON.

No connection for the RES and NMI pins of the EPROM simulator are needed, if a LAUTERBACH evaluation board is used. On these boards the target CPU gets this signals from the PODBUS interface.
**eXception.NMIBREAK**

Format:  
```
   eXception.NMIBREAK ON | OFF
```

The connection of the NMI signal to the target CPU is enabled or disabled.

**eXception.NMIPOL**

Format:  
```
   eXception.NMIPOL <polarity>
```

<polarity>:  
```
   + | -
```

Sets the polarity for the NMI.

**eXception.RESetPOL**

Format:  
```
   eXception.RESetPOL <polarity>
```

<polarity>:  
```
   + | -
```

Sets the polarity for the Reset.
RESet

Format:  RESet

(ROM Monitor) Switches off the EPROM simulator and resets the mapping system to its default state.

The target CPU is reset only if the RES pin of one adapter is connected and adapted to the target CPU.
Either the divided CPU frequency is used as the BDM clock or a fixed clock rate. The fixed clock rate must be used when the operation frequency is very slow clock line is not available.

The default is a fixed rate of 1 MHz.
### SYStem.Mode

**Format:** `SYStem.Mode <mode>`

**<mode>:**
- **Down**
- **NoDebug**
- **Go**
- **Up**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Down</strong> (BDM)</td>
<td>Disables the debug mode. The state of the CPU remains unchanged.</td>
</tr>
<tr>
<td><strong>Down</strong> (ROM Monitor)</td>
<td>Switches of the EPROM Simulator. Asserts the Reset Port.</td>
</tr>
<tr>
<td><strong>Go</strong> (BDM)</td>
<td>Resets the system with debug mode enabled. The system runs until any break condition occurs.</td>
</tr>
<tr>
<td><strong>Go</strong> (Serial)</td>
<td>Attaches the debugger to a running target.</td>
</tr>
<tr>
<td><strong>ATTACH</strong> (BDM)</td>
<td>Attaches the debugger to a running target without reset.</td>
</tr>
<tr>
<td><strong>ATTACH</strong> (Serial)</td>
<td>Attaches the debugger to a stopped target.</td>
</tr>
<tr>
<td><strong>NoDebug</strong> (BDM only)</td>
<td>Resets the system with debug mode disabled.</td>
</tr>
<tr>
<td><strong>Up</strong> (BDM)</td>
<td>Resets the system with debug mode enabled and breaks before the first Op-Fetch.</td>
</tr>
<tr>
<td><strong>Up</strong> (ROM Monitor)</td>
<td>Activates the EPROM Simulator, then the Reset Port is negated.</td>
</tr>
<tr>
<td><strong>Up</strong> (Serial)</td>
<td>Removes the RESET lines (if attached) and waits for monitor entry.</td>
</tr>
</tbody>
</table>
SYStem.Option

<table>
<thead>
<tr>
<th>Format:</th>
<th>SYStem.Option &lt;option&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;option&gt;:</td>
<td>BASE</td>
</tr>
</tbody>
</table>

**BASE**

Defines the base address of the internal peripherals. This value is used by the `PER` command to display the internal registers.
Trigger

Trigger.Set (BDM only)

Format: `Trigger.Set <source> [ON | OFF]`

`<source>`: BUSA

Enables the external Trigger input. The emulation is breaked when a high pulse is active for more than 300 ns.

```plaintext
b:
  Trigger.Set BUSA ON
```

The TRACE32-ICE system can generate the trigger pulse to break the emulation using the trigger bus line A. The trigger pulse is controlled by an analyzer program.

```plaintext
bus.a if ab ; the trigger pulse to break the emulation is
  ; generated at an alpha breakpoint
```

Trigger.Out (BDM only)

Format: `Trigger.Out`

Outputs a high pulse of 200 ns.

An analyzer trigger program of the TRACE32-ICE system controls the reaction to a trigger pulse of the BDM Debugger.

```plaintext
trigger.a if busa ; asserts trigger A of the TRACE32-ICE system,
  ; when the BDM Debugger outputs a trigger pulse
```
Trace Methods

This section describes which methods are provided by TRACE32 to record run-time information even if only a debugger is used.

Debugging is handled mostly via on-chip debugging interfaces e.g. JTAG, BDM, OCDS. The main tasks of a debugger are: to start the program and to stop it at a more or less complex breakpoint. With a halted program the current state of the CPU/target can be read out and modified respectively.

Debugging comes to its limits when run-time errors occur. Run-time errors are errors whose causes, either time-related or in the program hierarchy, are so far apart that the correlation between cause and effect can no longer be detected by a debugger. Moreover no analysis of the runtime behavior is possible by using debugger.

In the following, ancillary trace methods are presented. These trace methods allow to record run-time information even with a debugger.

ART

LOGGER

SNOOPer: Snooping allows a periodic sampling of up to 16 data items or the program counter.

FDX: FDX (Fast Data eXchange) allows to transfer arbitrary data between the target and the host. It is implemented by using a specialized communication protocol, supplied by LAUTERBACH.

OnChip
ART - Advanced Register Trace

Debugging on Mixed Level

ART logs all CPU registers when single stepping on assembler level. This information is used to provide a single step trace.

![ART log example](image)

Assembler steps that only changed the contents of CPU registers can be undone by pushing the **Undo** button.

| Mode.Mix | Select mixed mode debugging |
| ART.List | List the contents of the ART trace |
| Frame.UNDO | Undo the register changes performed by the last assembler step |
| Frame.REDO | Reverse Frame.UNDO |

If complex single step commands are performed e.g.

```c
Var.Step.Till flags[3]==0&&i>8
```

ART shows which program steps were performed by the CPU to get into the specified state of the system.
The default size of the ART is 256 single steps (records). The max. size is 65536.

**ART.state**
Display ART configuration window

**ART.SIZE <records>**
Specify size for ART trace
In order to realize a fast debugging a `hllstep` is realized by starting the program execution and stop it at the next HLL line. Due to this proceeding ART can’t log the CPU registers for each instruction executed. As a result part of the assembler program flow is lost for ART.

In many cases TRACE32 can reconstruct the assembler program flow (`hllseq`).
Problem description: A TRACE32-ICD debugger is used to test and integrate an application program on the target. Now a problem occurs, that could easily be solved if more information about the program history would be available.

Usually a TRACE32-ICD hardware trace extension can be used to get more information about the program history. But not all targets allow the operation of such a hardware trace.

For these targets Lauterbach is offering a software trace. The software trace however needs RAM from the target and is influencing the real-time behavior.

To operate a software trace, Lauterbach provides:

- A general trace format for a software trace located in the target RAM
- Configuration and display commands for the software trace in the TRACE32 software (command: LOGGER)
- Predefined algorithms to operate the software trace from the target program

To use the software trace basic knowledge of the target hardware and the processor architecture is required.
<table>
<thead>
<tr>
<th>Implementation of the trace memory</th>
<th>The user reserves a part of the target RAM, that can be used by TRACE32 as trace memory.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max. trace size</td>
<td>Any desired.</td>
</tr>
<tr>
<td>Sampling</td>
<td>The trace memory is filled either by an algorithm predefined by Lauterbach or by a user-defined algorithm. The algorithm can either be called by an interrupt or the code has to be instrumented.</td>
</tr>
<tr>
<td>Influence on the real-time behavior</td>
<td>Yes, depends on the implementation of the sampling algorithm.</td>
</tr>
<tr>
<td>Selective tracing</td>
<td>Possible by the sampling algorithm.</td>
</tr>
<tr>
<td>Fastest sampling rate</td>
<td>Depends on the sampling algorithm.</td>
</tr>
</tbody>
</table>
## The Trace Format

<table>
<thead>
<tr>
<th>LOGGER Description Block</th>
<th>16-bit flags</th>
<th>48-bit timestamp</th>
<th>32-bit address</th>
<th>32-bit data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start address (32-bit)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Size (32-bit)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Index (32-bit)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trigger counter (32-bit)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flags from the host (32-bit)</td>
<td>Bit 0: ARM (high active) &lt;br&gt; Bit 8: FIFO mode (low active), Stack mode (high active)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flags to the host (32-bit)</td>
<td>Bit 0: Overrun (high active) &lt;br&gt; Bit 8: Trigger (high active) &lt;br&gt; Bit 9: Break (high active)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(currently not used)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(currently not used)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### LOGGER Description Block

- **Start address (32-bit)**: Start address of the software trace in the target RAM.
- **Size (32-bit)**: Number of trace records.
- **Index (32-bit)**: Record number for the next entry.
- **Trigger counter (32-bit)**
- **Flags from the host (32-bit)**: Bit 0: ARM (high active) <br> Bit 8: FIFO mode (low active), Stack mode (high active)
- **Flags to the host (32-bit)**: Bit 0: Overrun (high active) <br> Bit 8: Trigger (high active) <br> Bit 9: Break (high active)
- **(currently not used)**
Operation Modes

The software trace can work in 2 operation modes:

- **Address/data trace**
  
  Address and data information is sampled.

- **Flow trace**
  
  A flow trace is available on architectures which provide a ‘branch trace’ capability like all PowerPC families and the SuperH SH4. For a flow trace all changes in the program flow are sampled. The TRACE32 software reconstructs and displays the complete program flow out of this information.

### Address and Data Trace

#### Software Trace Record Description

<table>
<thead>
<tr>
<th>Flags (16-bit)</th>
<th>Fetch (0x10)</th>
<th>Data Read (0x20)</th>
<th>Data Write (0x30)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timestamp (48-bit)</td>
<td>Timestamp from a timer, counter etc. from the target</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address (32-bit)</td>
<td>32-bit address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data (32-bit)</td>
<td>32-bit data</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If a normal software trace is used, the software trace listing displays the address and data, the cycle type and a timestamp.
If the software trace is used to sample the program flow, the software trace listing reconstructs the program flow based on the sampled information about the changes in the program flow.
Software Trace Configuration

TRACE32 Software
LOGGER configuration commands

LOGGER description block

Predefined algorithm
in debug monitor (not visible) e.g. SH4
in code linked to the application e.g. PPC

User-defined algorithm

Software trace in target RAM
1. Definition of the LOGGER description block and the software trace

Example:

```c
typedef struct
{
    unsigned long tshigh;
    /* high part of timestamp and cycle info */
    unsigned long tsslow;
    /* low part of timestamp */
    unsigned long address;
    unsigned long data;
} T32_loggerData;
#define T32_LOGGER_SIZE 1024
int T32_LoggerFast = TRUE;
extern T32_loggerData T32_LoggerBuffer[];

struct
{
    T32_loggerData * ptr;
    /* pointer to trace data */
    long size;    /* size of trace buffer */
    volatile long index;
    /* current write pointer */
    long tindex;  /* index of trigger record */
    long iflags;  /* incoming flags, Bit 0: ARM, */
    /* Bit 8: Stack Mode */
    long oflags;  /* outgoing flags, Bit 0: Overflow, */
    /* Bit 8: Trigger, Bit 9: */

    Break
    long reserved1;
    long reserved2;
    T32_loggerData buffer[T32_LOGGER_SIZE];
} T32_LoggerStruct;
```
2. Instrument your program to fill the software trace.

Add commands to fill the software trace at the appropriate location in your application.

Example:

```
T32_LoggerData(<cycle_type>, <address>, <data>)
```

Use an interrupt to fill the software trace.

Example:

```
__interrupt__ void T32_LoggerInterrupt()
{
    T32_LoggerData(<cycle_type>, <address>, <data>);
}
```

The main tasks of the T32_LoggerData algorithm are:

- To handle the flags in the software trace
- To enter the timestamp (optional)
- To enter the trace data
- To maintain the software trace by the logger description block

An example for a T32_LoggerData algorithm can be found in the TRACE32 demo folder, e.g. in

```
~/demo/powerpc/etc/logger/mpc500/logdemo.c
```
3. Configure the software trace in the TRACE32 software.

**LOGGER.state**  
Display the LOGGER configuration window

<table>
<thead>
<tr>
<th><strong>LOGGER address</strong></th>
<th>Enter the start address of the logger description block</th>
</tr>
</thead>
</table>
| **Timestamp configuration** | OFF: no timestamp is used  
Up: timestamp counts up  
Down: timestamp counts down  
Rate: Frequency of the timestamp in Hz |
| **Usage of the software trace** | Flow Trace OFF: software trace is used to sample address and data information  
Flow Trace ON: software trace is used to sample program flow |
4. Initialize the software trace.

Push the **Init** button in the **commands** field to initialize the software trace. Prior to the initialization the chip selects have to be configured in order to get access to the target RAM.

After the initialization the **SIZE** field contains the size of the software trace.

5. Switch on the software trace in the **TRace** configuration window.
1. **Definition of the LOGGER description block**

Command: **LOGGER.state**

**LOGGER.state** Display the LOGGER configuration window

![Logger Configuration Window](image)

- **Size**
  - **Define the size of the software trace**

- **Timestamp configuration**
  - **OFF**: no timestamp is used
  - **Up**: timestamp counts up
  - **Down**: timestamp counts down
  - **Rate**: Frequency of the timestamp in Hz

- **Usage of the software trace**
  - **Flow Trace OFF**: software trace is used to sample address and data information
  - **Flow Trace ON**: software trace is used to sample program flow
The required target memory size can be calculated as follows:

- Memory required for the LOGGER Description block: 32 byte
- Memory required for the Software Trace: Number of records x 16 byte

Logger_Memory_Size = 32 byte + (Number of records x 16 byte)

2. Instrument your program to fill the software trace.

Add commands to fill the software trace at the appropriate location in your application

**Example:**

```c
T32_LoggerData(<cycle_type>, <address>, <data>)
```

Use an interrupt to fill the software trace.

**Example:**

```c
__interrupt__ void T32_LoggerInterrupt()
{
    T32_LoggerData(<cycle_type>, <address>, <data>);
}
```

The main tasks of the T32_LoggerData algorithm are:

- to handle the flags in the software trace
- to enter the timestamp (optional)
- to enter the trace data
- to maintain the software trace by the logger description block

An example for a T32_LoggerData algorithm can be found in the TRACE32 demo folder, e.g. in `~/demo/powerpc/etc/logger/mpc500/logdemo.c`

For the SH4 family a hidden monitor is provided by the TRACE32 software to fill the software trace. So if a software trace is used to sample the program flow for the SH4, no instrumentation of the code is necessary.
3. Initialize the software trace.

Push the **Init** button in the **commands** field to initialize the software trace. Prior to the initialization the chip selects have to be configured in order to get access to the target RAM.

After the initialization the hidden monitor for the SH4 family is automatically activated.

4. Switch on the software trace in the TRace configuration window.
Display the Software Trace

TRACE32 Software
LOGGER display commands

LOGGER description block

Software trace in target RAM

Click here to get a trace listing
Software Trace as a Flow Trace for the SH4

This section shows you, how to set up a flow trace for the SH4 by using the TRACE32-ICD software trace.

Background

The SH4 core includes a 8-stage on-chip branch trace. This trace holds the source and destination address of the last eight changes in the program flow. Sampling to the on-chip branch trace in non-intrusive.

3 classes of branch operations can be traced by the on-chip branch trace of the SH4:

- Exception branches
  All exceptions other than ASE exception, as well as all interrupt operation and RTE instruction
- Subroutine branches
  BSR, BSRF, JSR and RTS instructions
- General branches
  BF, BT, BF/S, BT/S, BRA, BRAF, and JMP instructions

The user program can be stopped by a TRACE exception, when the buffer contains 1 or 6 records.

This on-chip trace buffer is used as the basis for the software trace. At the moment only **General branches** are supported for the use in the software trace.
Select what should be sampled into on-chip branch trace (FIFO)

Command: FIFO

<table>
<thead>
<tr>
<th>Address</th>
<th>Mnemonic</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>-00015</td>
<td>BCO000FE</td>
<td>bf</td>
</tr>
<tr>
<td>-00014</td>
<td>BCO000FSF</td>
<td>mov,1</td>
</tr>
<tr>
<td>-00013</td>
<td>BCO000FSF</td>
<td>bf</td>
</tr>
<tr>
<td>-00012</td>
<td>BCO000FSF</td>
<td>mov,1</td>
</tr>
<tr>
<td>-00011</td>
<td>BCO000FSF</td>
<td>bf</td>
</tr>
<tr>
<td>-00010</td>
<td>BCO000FSF</td>
<td>mov,1</td>
</tr>
<tr>
<td>-00009</td>
<td>BCO000FE</td>
<td>bf</td>
</tr>
<tr>
<td>-00008</td>
<td>BCO000FE</td>
<td>bf</td>
</tr>
<tr>
<td>-00007</td>
<td>BCO000FSF</td>
<td>mov,1</td>
</tr>
<tr>
<td>-00006</td>
<td>BCO000FSF</td>
<td>mov,1</td>
</tr>
<tr>
<td>-00005</td>
<td>BCO000FSF</td>
<td>mov,1</td>
</tr>
<tr>
<td>-00004</td>
<td>BCO000FSF</td>
<td>mov,1</td>
</tr>
<tr>
<td>-00003</td>
<td>BCO000FSF</td>
<td>mov,1</td>
</tr>
</tbody>
</table>

FIFO Modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFF</td>
<td>The on-chip branch trace is disabled.</td>
</tr>
<tr>
<td>eXception</td>
<td>The on-chip branch trace samples exceptions, interrupts and RTE instructions.</td>
</tr>
<tr>
<td>Subroutine</td>
<td>The on-chip branch trace samples exceptions, interrupts and RTE, BSR, BSRF, JSR and RTS instructions.</td>
</tr>
<tr>
<td>ALL</td>
<td>The on-chip branch trace samples all changes in the program flow.</td>
</tr>
<tr>
<td><strong>SoftwAre Trace Record Description FlowTrace for SH4</strong></td>
<td></td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td></td>
</tr>
<tr>
<td><strong>Flags (16-bit)</strong></td>
<td>0xF00n: 1st flow trace record that was copied from the on-chip branch trace, n records with the same timestamp will follow 0xF00E: following flow trace record 0xF00F: last flow trace record in memory, all records until record SIZE-1 are empty, next valid entry starts at record number 1. This flag is used, if there are not enough empty records left in the software trace to store the next 6 entries from the on-chip branch trace.</td>
</tr>
<tr>
<td><strong>Timestamp (48-bit)</strong></td>
<td>Timestamp from performance counter (elapsed time)</td>
</tr>
<tr>
<td><strong>Address (32-bit)</strong></td>
<td>Branch source address</td>
</tr>
<tr>
<td><strong>Address (32-bit)</strong></td>
<td>Branch destination address</td>
</tr>
</tbody>
</table>
1. Allocate memory space for the logger description block and the software trace.

**Command: LOGGER**

Allocate memory space for the logger description block and the software trace. The required target memory size can be calculated as follows:

Memory required for the LOGGER Description block: 32 byte

Memory required for the Software Trace: Number of records x 16 byte

Logger_Memory_Size = 32 byte + (Number of records x 16 byte)

Enter the start address of the available target memory block into the **ADDRESS** field of the LOGGER Window and switch **ON** the **Create** check box in the **Mode** field to inform the TRACE32 software that you want to create the LOGGER Description block and the software trace from the TRACE32 software.

Enter the number of records used for the Software Trace to the **SIZE** field.

Select **ALL** in the Flow Trace field. Then all changes in the program flow are sampled into the software trace.

TRACE32 now allocates the memory for the software trace subsequently to the logger description block.
2. Define the method in which the Software Trace is filled.

By default a TRACE exception is generated after 6 valid entries to the on-chip branch trace. The user program is stopped by this TRACE exception and a hidden monitor program is started. This monitor program copies the contents of the on-chip branch trace into the software trace and then restarts the program execution.

The influence on the run-time depends on the target program. With the estimation that the program flow changes every 5 cycles, the program runtime will be 5 times slower.

If the check box AllCycles is switched ON, a TRACE exception is generated after each valid entry to the on-chip branch trace.
3. Define the usage of the TimeStamp.

TRACE32 provides the possibility to mark the entries to the software trace with a timestamp. One of the Performance counters of the SH4 is used as timer here. The used performance counter is counting upwards.

When the hidden monitor program copies the entries from the on-chip branch trace to the software trace, the entries are marked with the timestamp. That means when 6 entries are copied at a time, all 6 entries have the same timestamp.

The Performance Counter is stopped, when the hidden monitor is active.
4. Initialize the software trace

Push the **Init** button in the **commands** field to initialize the software trace. Prior to the initialization the chip selects have to be configured in order to get access to the target RAM.

After the initialization the hidden monitor for the SH4 family is automatically activated.

5. Start the user program by Go and stop it again. Display the software trace.
This section shows you, how to set up a flow trace for the MPC860 by using the TRACE32-ICD software trace.

**Background**

The MPC860 has a Trace Exception. The Trace Exception occurs:

- if MSR[SE]=1 and any instruction other then rfi is successfully completed
- if MSR[BE]=1 and a branch is completed

If the Trace Exception causes the processor to enter into the debug mode or if the Trace Exception is handled by an interrupt service routine, can be decided by setting the TRE (Trace interrupt enable bit) bit in the DER (Debug Enable Register).

To configure the MPC860 to support a software trace as flow trace:

- MSR[BE]=1 and MSR[SE]=0 has to be set
- DER[TRE]=0 has to be set

In consequence of this single stepping is not possible while a software trace as flow trace is used.

The time base facility (TB) of the MPC860 can be used as source for the timestamp unit.

**Software Trace Format**

<table>
<thead>
<tr>
<th>SoftwAre Trace Record Description FlowTrace for MPC860</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flags (16-bit)</td>
</tr>
<tr>
<td>Timestamp (48-bit)</td>
</tr>
<tr>
<td>Address (32-bit)</td>
</tr>
<tr>
<td>Address (32-bit)</td>
</tr>
</tbody>
</table>
How to use the Software Trace

1. Add a definition for the LOGGER Description Block and the Software Trace to your application.
2. Add a interrupt service routine for the Trace Exception to your application.
   The main tasks of the interrupt service routine are:
   - handle the flags in the software trace
   - read the Time Base and enter it as timestamp (optional)
   - enter the branch destination address
   - maintain the software trace by the logger description block
   An example can be found on the TRACE32 software DVD:

   ```
   ~/demo/powerpc/etc/logger/mpc500/logdemo.c
   ```

3. Set the MSR register.

   ```
   Register.Set MSR 0x0202
   ```

4. Disable the Trace Exception in DER.

   ```
   TrOnchip.Set TRE OFF
   ```
5. Configure the software trace in the TRACE32 software.

Command: **LOGGER.state**

**LOGGER.state** Display LOGGER configuration window

![Image of LOGGER configuration window]

- **Timestamp configuration**
- **LOGGER address**
- Define the usage of the software trace

<table>
<thead>
<tr>
<th>LOGGER address</th>
<th>Enter the start address of the logger description block</th>
</tr>
</thead>
</table>
| **Timestamp configuration** | OFF: no timestamp is used  
Up: Time Base counts upwards  
Rate: Frequency of the Time Base in Hz |
| **Usage of the software trace** | Flow Trace ON: software trace is used to sample program flow |

6. Initialize the software trace.

![Image of initialization process]

Push **Init** to initialize the software trace

Push the **Init** button in the **commands** field to initialize the software trace. Prior to the initialization the chip selects have to be configured in order to get access to the target RAM.
7. Switch on the software trace in the Trace configuration window.

8. Start the user program by Go and stop it again. Display the software trace.