Meta Debugger

MANUAL

TRACE32 Online Help

TRACE32 Directory

TRACE32 Index

TRACE32 Documents .... 🗁

ICD In-Circuit Debugger .... 🗁

Processor Architecture Manuals .... 🗁

Meta .... 🗁

Meta Debugger 1

Introduction 5

Brief Overview of Documents for New Users 5

Demo and Start-up Scripts 6

Warning 6

Quick Start of the Debugger 8

Troubleshooting 10

Communication between Debugger and Processor can not be established 10

FAQ 10

Meta specific Implementations 11

Meta Configuration 11

Access Classes 12

Breakpoints 13

Software Breakpoints 13

On-chip Breakpoints 13

On-chip Watchpoints 13

SYStem.CONFIG Configure debugger according to target topology 15

<parameters> describing the “DebugPort” 16

<parameters> describing the “JTAG” scan chain and signal behavior 18

SYStem.CONFIG.state Display target configuration 21

SYStem.CPU Select the used CPU 21

SYStem.JtagClock Define JTAG frequency 21

SYStem.LOCK Lock and tristate the debug port 22

SYStem.MemAccess Select run-time memory access method 23

SYStem.Mode Establish communication with target 24

SYStem.Option.IMASKASM Disable interrupts while single stepping 24

SYStem.Option.IMASKHLL Disable interrupts while HLL single stepping 26

SYStem.Option.MINIM Map execution- to storage address range 26

SYStem.state Display SYStem.state window 26
TrOnchip.RESet Set on-chip trigger to default state 27
TrOnchip.state Display on-chip trigger window 27

Target Adaption 28
Interface Standards JTAG, Serial Wire Debug, cJTAG 28
Connector Type and Pinout 28
Debug Cable 28

Version 06-Jun-2024

Introduction

This manual serves as a guideline for debugging Meta cores and describes all processor-specific TRACE32 settings and features.

Please keep in mind that only the **Processor Architecture Manual** (the document you are reading at the moment) is CPU specific, while all other parts of the online help are generic for all CPUs supported by Lauterbach. So if there are questions related to the CPU, the Processor Architecture Manual should be your first choice.

Brief Overview of Documents for New Users

Architecture-independent information:

- **“Training Basic Debugging”** (training_debugger.pdf): Get familiar with the basic features of a TRACE32 debugger.
- **“T32Start”** (app_t32start.pdf): T32Start assists you in starting TRACE32 PowerView instances for different configurations of the debugger. T32Start is only available for Windows.
- **“General Commands”** (general_ref_*<x>*.pdf): Alphabetic list of debug commands.

Architecture-specific information:

- **“Processor Architecture Manuals”**: These manuals describe commands that are specific for the processor architecture supported by your Debug Cable. To access the manual for your processor architecture, proceed as follows:
  - Choose **Help** menu > **Processor Architecture Manual**.
- **“OS Awareness Manuals”** (rtos_*<os>*.pdf): TRACE32 PowerView can be extended for operating system-aware debugging. The appropriate OS Awareness manual informs you how to enable the OS-aware debugging.

To get started with the most important manuals, use the **Welcome to TRACE32!** dialog (**WELCOME.view**):

Demo and Start-up Scripts

Lauterbach provides ready-to-run start-up scripts for known Meta-based hardware.

To search for PRACTICE scripts, do one of the following in TRACE32 PowerView:

- Type at the command line: **WELCOME.SCRIPTS**
- or choose **File** menu > **Search for Script**.

  You can now search the demo folder and its subdirectories for PRACTICE start-up scripts (*.cmm) and other demo software.

You can also manually navigate in the `~~/demo/meta/` subfolder of the system directory of TRACE32.

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.

Quick Start of the Debugger

Starting up the debugger is done as follows:

1. Reset the debugger.

   ```
   RESet
   ```

   The **RESet** command is only necessary if you do not start directly after booting the TRACE32 development tool.

2. Set the target CPU to configure the debugger.

   ```
   SYStem.CPU <cpu>
   ```

   The default values of all other options are set in such a way that it should be possible to work without modifications. Please consider that this may not be the best configuration for your target.

3. Establish the communication to the target.

   ```
   SYStem.Up
   ```

   This command resets the target and tries to stop it. After this command is executed, it is possible to access memory and registers.

4. Load the program into the memory.

   ```
   Data.LOAD.Elf sieve.elf      ; .ELF specifies the file format
                                ; sieve.elf is the file name
   ```

A typical start sequence is shown below. This sequence can be written to a PRACTICE script file (*.cmm, ASCII format) and executed with the command **DO** *<file>*.

```
RESet                           ; Reset the debugger

System.CPU META-MTP             ; Set target CPU, here the generic
                                ; META-MTP to configure the debugger

SYStem.Up                       ; Establish communication to target

Data.LOAD.Elf sieve.elf         ; Load the application program

WinCLEAR                        ; Remove all windows

List.Mix                        ; Open source window          *)

Register.view                   ; Open register window        *)
```

*) These commands open windows on the screen. The window position can be specified with the **WinPOS** command.

Troubleshooting

Communication between Debugger and Processor can not be established

Typically the **SYStem.Up** command is the first command of a debug session where communication with the target is required. If you receive error messages like “debug port fail” or “debug port time out” while executing this command, this may have the reasons described below. “target processor in reset” is just a follow-up error message. Open the **AREA.view** window to view all error messages.

- The target has no power or the debug cable is not connected to the target or the target reference voltage is not connected to the debug connector. This results in the error message “target power fail”.
- The target is in reset.
- The target is in an unrecoverable state. Re-power your target and try again.
- You have selected the incorrect CPU with **SYStem.CPU**.
- There is an issue with the JTAG interface. See **“Arm JTAG Interface Specifications”** (app_arm_jtag.pdf) and the manuals or schematic of your target to check the physical and electrical interface. Maybe there is the need to set jumpers on the target to connect the correct signals to the JTAG connector.
- The default JTAG clock speed is too fast, especially if you emulate your core or if you use an FPGA-based target. In this case try **SYStem.JtagClock 50kHz** and optimize the speed when you got it working.
- You might have several TAP controllers in a JTAG-chain. Example: You have a multicore system with chained TAPs. In this case you have to check your pre- and post-bit configuration. See **SYStem.CONFIG IRPRE** or **SYStem.CONFIG DRPRE**.

FAQ

Please refer to https://support.lauterbach.com/kb.

Meta specific Implementations

Meta Configuration

For code compression the Meta architecture provides a so-called MiniM instruction set. But unlike to other architectures the HLL debug information are not created for this reduced -, but for the expanded program code. So the Meta tool chain creates an additional output file with the extension *.ldr which includes the Minim code without debug information and a storage address which is different to the linker execution address.

This file must be loaded with the command Data.load.LDR <file_name> which will also enable the MiniM operating mode in the Core, by setting the MinimEnable flag in the TXPRIVEXT register.

To provide HLL debugging capabilities for this concept the T32 Lauterbach SW is reading the MiniM opcodes from the storage- and encodes it to the linker execution address. The HLL debug information are disposed by additionally loading the referring *.elf file with the /NoCODE option. To get this working the storage and execution memory range must be declared by the **System.Option.MINIM** command. HLL debugging is working as usual and breakpoints are automatically mapped to storage memory.

NOTE:	Loading the *.elf file code, compiled for MiniM usage, to the target will fail, because tit is linked in a way that parts of the instruction memory will be overloaded by the .data section!

Access Classes

Access classes are used to specify which memory to access. For background information about the term access class, see **“TRACE32 Glossary”** (glossary.pdf).

The following common access classes have the same meaning for all CPUs of the Meta architecture.

Access Class	Description
P	Program or data memory access. Target implementation defined.
D	Data memory access
R0A	Data Unit D0 DSP RAM Block A.
R0B	Data Unit D0 DSP RAM Block B.
R1A	Data Unit D1 DSP RAM Block A.
R1B	Data Unit D1 DSP RAM Block B.
DBG	Special, virtual memory. Target implementation defined.
E	Prefix: Run-time access specifier.
VM	Virtual Memory. Memory on the debug host system.

To perform an access with a certain access class, write the class in front of the address.

Examples:

```
List P:0x80000
List EP:0x120000
Data.dump D:0x4--0x7
PRINT Data.Long(R0B:0x6)
```

Breakpoints

For general information about setting breakpoints, refer to the **Break.Set** command.

Software Breakpoints

If a software breakpoint is used, the original code at the breakpoint location is temporarily patched by a breakpoint code (Meta *switch* instruction). There is no restriction in the number of software breakpoints. Software breakpoints are break before make.

On-chip Breakpoints

If on-chip breakpoints are used, the resources to set the breakpoints are provided by the hardware of the core itself. The Meta supports up to 8 program on-chip breakpoints. The debugger is able to detect the number of available on-chip breakpoints by analyzing the contents of the Meta core revision register.

If programmed, the breakpoint hardware compares its breakpoint address and the current program counter. If they are equal, a *breakpoint* exception is raised which in general will set the Meta core into debug mode. On-chip breakpoints are break before make.

Examples:

```
Break.Set 0x80000024 /Program         ; Configures an on-chip breakpoint
/Onchip                               ; which activates when the program
                                      ; counter matches 0x80000024

Break.Set 0x80000024 /Onchip          ; Same as above, since the default
                                      ; for breakpoints is /Program
```

On-chip Watchpoints

If on-chip watchpoints are used, the resources to set the watchpoints are provided by the hardware of the core itself. The Meta supports up to 8 data on-chip watchpoints. The debugger is able to detect the number of available on-chip watchpoints by analyzing the contents of the Meta core revision register.

On-chip watchpoints compare their programmed address and their read, write or read-write access condition with addresses of load and store instructions. If addresses and conditions match, a *watchpoint* exception is raised which in general will set the M core into debug mode. On-chip watchpoints are break before make.

In TRACE32, the on-chip watchpoint functionality is mapped to data address breakpoints. That means to set a watchpoint, the **Break.Set** command is used in conjunction with the **Read**, **Write** or **ReadWrite** options.

Examples:

```
Break.Set 0x80001000 /Read /Onchip    ; Configures an on-chip read
                                      ; watchpoint which activates when
                                      ; a load instruction accesses
                                      ; address 0x80001000

Break.Set 0x80001000 /Read            ; Same as above, since read
                                      ; breakpoints are always on-chip

Break.Set 0x80001000 /Write           ; Write watchpoint for store
                                      ; instructions

Break.Set 0x80001000 /ReadWrite       ; ReadWrite watchpoint for load and
                                      ; store instructions
```

Format:	**SYStem.CONFIG** *<parameter>*
<parameter>: **(DebugPort)**	**CORE** *<core> <chip>* **DEBUGPORT** [**DebugCable0**] **DEBUGPORTTYPE** [**JTAG**] **Slave** [**ON** | **OFF**] **TriState** [**ON** | **OFF**]
<parameter>: **(JTAG)**	**DRPOST** *<bits>* **DRPRE** *<bits>* **IRPOST** *<bits>* **IRPRE** *<bits>* **Slave** [**ON** | **OFF**] **TAPState** *<state>* **TCKLevel** *<level>* **TriState** [**ON** | **OFF**]

The **SYStem.CONFIG** commands inform the debugger about the available on-chip debug and trace components and how to access them.

The **SYStem.CONFIG** command information shall be provided after the **SYStem.CPU** command, which might be a precondition to enter certain **SYStem.CONFIG** commands, and before you start up the debug session, e.g. by **SYStem.Up**.

Syntax Remarks

The commands are not case sensitive. Capital letters show how the command can be shortened.
Example: “SYStem.CONFIG.TriState ON” -> “SYStem.CONFIG.TS ON”

The dots after “SYStem.CONFIG” can alternatively be a blank.
Example:
“SYStem.CONFIG.TriState ON” or “SYStem.CONFIG TriState ON”

CORE *<core>* *<chip>*	The command helps to identify debug and trace resources which are commonly used by different cores. The command might be required in a multicore environment if you use multiple debugger instances (multiple TRACE32 PowerView GUIs) to simultaneously debug different cores on the same target system. Because of the default setting of this command debugger#1: *<core>*=1 *<chip>*=1 debugger#2: *<core>*=1 *<chip>*=2 ... each debugger instance assumes that all notified debug resources can exclusively be used. But some target systems have shared resources for different cores, for example a common trace port. The default setting causes that each debugger instance controls the same trace port. Sometimes it does not hurt if such a module is controlled twice. But sometimes it is a must to tell the debugger that these cores share resources on the same *<chip>*. Whereby the "chip" does not need to be identical with the device on your target board: debugger#1: *<core>*=1 *<chip>*=1 debugger#2: *<core>*=2 *<chip>*=1
CORE *<core>* *<chip>* (cont.)	For cores on the same *<chip>,* the debugger assumes that the cores share the same resource if the control registers of the resource have the same address. Default: *<core>* depends on CPU selection, usually 1. *<chip>* derives from the `CORE=` parameter in the configuration file (config.t32), usually 1. If you start multiple debugger instances with the help of t32start.exe, you will get ascending values (1, 2, 3,...).
DEBUGPORT [**DebugCable0**]	It specifies which probe cable shall be used e.g. "DebugCable0". Default: depends on detection.
DEBUGPORTTYPE [**JTAG**]	It specifies the used debug port type "JTAG". It assumes the selected type is supported by the target. Default: JTAG.

Slave [**ON** | **OFF**]	If several debuggers share the same debug port, all except one must have this option active. JTAG: Only one debugger - the “master” - is allowed to control the signals nTRST and nSRST (nRESET). The other debuggers need to have the setting **Slave OFF**. Default: OFF. Default: ON if `CORE=`... >1 in the configuration file (e.g. config.t32).
TriState [**ON** | **OFF**]	TriState has to be used if several debug cables are connected to a common JTAG port. **TAPState** and **TCKLevel** define the TAP state and TCK level which is selected when the debugger switches to tristate mode. Please note: • nTRST must have a pull-up resistor on the target. • TCK can have a pull-up or pull-down resistor. • Other trigger inputs need to be kept in inactive state. Default: OFF.

<parameters> describing the “JTAG” scan chain and signal behavior

With the JTAG interface you can access a Test Access Port controller (TAP) which has implemented a state machine to provide a mechanism to read and write data to an Instruction Register (IR) and a Data Register (DR) in the TAP. The JTAG interface will be controlled by 5 signals:

- nTRST(reset)
- TCK (clock)
- TMS (state machine control)
- TDI (data input)
- TDO (data output)

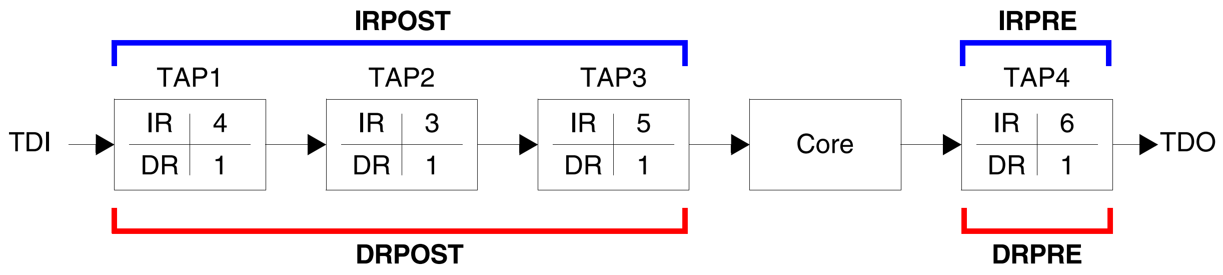
Multiple TAPs can be controlled by one JTAG interface by daisy-chaining the TAPs (serial connection). If you want to talk to one TAP in the chain, you need to send a BYPASS pattern (all ones) to all other TAPs. For this case the debugger needs to know the position of the TAP it wants to talk to. The TAP position can be defined with the first four commands in the table below.

DRPOST *<bits>*	Defines the TAP position in a JTAG scan chain. Number of TAPs in the JTAG chain between the TDI signal and the TAP you are describing. In BYPASS mode, each TAP contributes one data register bit. See example below. Default: 0.
DRPRE *<bits>*	Defines the TAP position in a JTAG scan chain. Number of TAPs in the JTAG chain between the TAP you are describing and the TDO signal. In BYPASS mode, each TAP contributes one data register bit. See example below. Default: 0.
IRPOST *<bits>*	Defines the TAP position in a JTAG scan chain. Number of Instruction Register (IR) bits of all TAPs in the JTAG chain between TDI signal and the TAP you are describing. See example below. Default: 0.
IRPRE *<bits>*	Defines the TAP position in a JTAG scan chain. Number of Instruction Register (IR) bits of all TAPs in the JTAG chain between the TAP you are describing and the TDO signal. See example below. Default: 0.

NOTE:	If you are not sure about your settings concerning **IRPRE**, **IRPOST**, **DRPRE**, and **DRPOST**, you can try to detect the settings automatically with the **SYStem.DETECT.DaisyChain** command.

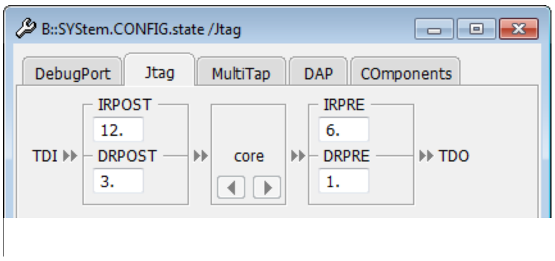
Slave [**ON** | **OFF**]	If several debuggers share the same debug port, all except one must have this option active. JTAG: Only one debugger - the "master" - is allowed to control the signals nTRST and nSRST (nRESET). The other debuggers need to have the setting **Slave OFF**. Default: OFF. Default: ON if `CORE=`... >1 in the configuration file (e.g. config.t32).
TAPState *<state>*	This is the state of the TAP controller when the debugger switches to tristate mode. All states of the JTAG TAP controller are selectable. 0 Exit2-DR 1 Exit1-DR 2 Shift-DR 3 Pause-DR 4 Select-IR-Scan 5 Update-DR 6 Capture-DR 7 Select-DR-Scan 8 Exit2-IR 9 Exit1-IR 10 Shift-IR 11 Pause-IR 12 Run-Test/Idle 13 Update-IR 14 Capture-IR 15 Test-Logic-Reset Default: 7 = Select-DR-Scan.
TCKLevel *<level>*	Level of TCK signal when all debuggers are tristated. Normally defined by a pull-up or pull-down resistor on the target. Default: 0.
TriState [**ON** | **OFF**]	TriState has to be used if several debug cables are connected to a common JTAG port. **TAPState** and **TCKLevel** define the TAP state and TCK level which is selected when the debugger switches to tristate mode. Please note: • nTRST must have a pull-up resistor on the target. • TCK can have a pull-up or pull-down resistor. • Other trigger inputs need to be kept in inactive state. Default: OFF.

Daisy-Chain Example



IR: Instruction register length **DR**: Data register length **Core**: The core you want to debug

Daisy chains can be configured using a PRACTICE script (*.cmm) or the **SYStem.CONFIG.state** window.



IRPOST 12.
DRPOST 3.
IRPRE 6.
DRPRE 1.

Example: This script explains how to obtain the individual IR and DR values for the above daisy chain.

```
SYStem.CONFIG.state /Jtag     ; optional: open the window

SYStem.CONFIG IRPRE   6.      ; IRPRE: There is only one TAP.
                              ; So type just the IR bits of TAP4, i.e. 6.

SYStem.CONFIG IRPOST 12.      ; IRPOST: Add up the IR bits of TAP1, TAP2
                              ; and TAP3, i.e. 4. + 3. + 5. = 12.

SYStem.CONFIG DRPRE   1.      ; DRPRE: There is only one TAP which is
                              ; in BYPASS mode.
                              ; So type just the DR of TAP4, i.e. 1.

SYStem.CONFIG DRPOST  3.      ; DRPOST: Add up one DR bit per TAP which
                              ; is in BYPASS mode, i.e. 1. + 1. + 1. = 3.
                              ; This completes the configuration.
```

NOTE:	In many cases, the number of TAPs equals the number of cores. But in many other cases, additional TAPs have to be taken into account; for example, the TAP of an FPGA or the TAP for boundary scan.

SYStem.CONFIG.state — Display target configuration

Format:	**SYStem.CONFIG.state**

Opens the **SYStem.CONFIG.state** window, where you can view and modify most of the target configuration settings. The configuration settings tell the debugger how to communicate with the chip on the target board and how to access the on-chip debug and trace facilities in order to accomplish the debugger's operations.

SYStem.CPU — Select the used CPU

Format:	**SYStem.CPU** *<cpu>*
<cpu>:	**META-MTP** | **MSR1-DRPU**

The choice of the CPU will determine pre-configurations made by the debugger. It will also determine the supported debug monitor.

META-MTP	Generic CPU for Meta-MTP targets.
MSR1-DRPU	Meta core DRPU on the MSR1 board

SYStem.JtagClock — Define JTAG frequency

Format:	**SYStem.JtagClock** [*<frequency>*]
<frequency>:	**10000.** … **40000000.**

Default frequency: 5 MHz.

Selects the JTAG port frequency (TCK) used by the debugger to communicate with the processor. The frequency affects e.g. the download speed. It could be required to reduce the JTAG frequency if there are buffers, additional loads or high capacities on the JTAG lines or if VTREF is very low. A very high frequency will not work on all systems and will result in an erroneous data transfer.

<frequency>	The debugger cannot select all frequencies accurately. It chooses the next possible frequency and displays the real value in the **SYStem.state** window. Besides a decimal number like “100000.” short forms like “10kHz” or “15MHz” can also be used. The short forms imply a decimal value although no “.” is used.

SYStem.LOCK — Lock and tristate the debug port

Format:	**SYStem.LOCK** [**ON** | **OFF**]

Default: OFF.

If the system is locked, no access to the debug port will be performed by the debugger. While locked, the debug connector of the debugger is tristated. The main intention of the **SYStem.LOCK** command is to give debug access to another tool.

SYStem.MemAccess — Select run-time memory access method

Format:	**SYStem.MemAccess** *<mode>*
<mode>:	**Enable** | **Denied** | **StopAndGo**

Default: Enable.

If **SYStem.MemAccess** is not **Denied**, it is possible to read from memory, to write to memory and to set software breakpoints while the CPU is executing the program.

Enable **CPU** (deprecated)	Run-time memory access is done via the instruction bus of the CPU.
Denied	No memory access is possible while the CPU is executing the program.
StopAndGo	Temporarily halts the core(s) to perform the memory access. Each stop takes some time depending on the speed of the JTAG port, the number of the assigned cores, and the operations that should be performed. For more information, see below.

Example: If specific windows that display memory or variables should be updated while the program is running, select the access class prefix **E** or the format option **%E**.

```
SYStem.MemAccess Enable

Data.dump EP:0x100

List E:

Var.View %E var1
```

SYStem.Mode — Establish communication with target

Format:	**SYStem.Mode** *<mode>*
	SYStem.Attach (alias for SYStem.Mode Attach) **SYStem.Down** (alias for SYStem.Mode Down) **SYStem.Up** (alias for SYStem.Mode Up)
<mode>:	**Down** **Attach** **Up** **Go** **NoDebug**

Attach	No reset happens, the mode of the core (running or halted) does not change. The debug port (JTAG) will be initialized. After this command has been executed, a possible running user program can, for example, be stopped with the **Break** command.
Down	Disables the debugger. The state of the CPU remains unchanged. The JTAG port is tristated.
Up	Resets the target via the reset line, initializes the debug port (JTAG), performs a core (soft) reset and enters debug mode. The core stops at the *exception base address* (EBA). For a reset via the JTAG line, the reset line has to be connected to the debug connector.
Go	Start code execution from reset vector. Actually the debugger performs the same actions than on **SYStem.Mode.Up** followed by **Go.direct**.
NoDebug	The debug adapter gets tristated. The state of the CPU remains unchanged. Debug mode is not active. In this mode the target behaves as if the debugger is not connected.
StandBy	Not available.

SYStem.Option.IMASKASM — Disable interrupts while single stepping

Format:	**SYStem.Option.IMASKASM** [**ON** | **OFF**]

Default: OFF.

ON	The Global Interrupt Enable Bits will be cleared during assembler single-step operations. The interrupt routine is not executed during single-step operations. After single step the Global Interrupt Enable bits will be restored to the value before the step.
OFF	A pending interrupt will be executed on a single-step, but it does not halt there. The specific interrupt handler is completely executed even if single steps are done, i.e. step over is forced per default. If the core should halt in the interrupt routine, use **TrOnchip.StepVector ON**.

SYStem.Option.IMASKHLL Disable interrupts while HLL single stepping

Format:	**SYStem.Option.IMASKHLL** [**ON** | **OFF**]

Default: OFF.

ON	The Global Interrupt Enable Bits will be cleared during high-level-language single-step operations. The interrupt routine is not executed during single-step operations. After single step, the Global Interrupt Enable bit will be restored to the value before the step.
OFF	A pending interrupt will be executed on a single-step, but it does not halt there, i.e. the interrupt handler is always stepped over.

SYStem.Option.MINIM Map execution- to storage address range

Format:	**SYStem.Option.MiniM** {<execution_range><storage_range>}

For expanding the compressed *.ldr file correctly the T32 MiniM SW encoder need the address range where the compressed datas are stored on the target memory <storage_range> and at which address they are linked to for execution. <execution range>.

NOTE 1:	Due to code compression rate 1:2 the execution range must always have twice the size of the storage range!

NOTE 2:	This command is only needed for Large Minim address ranges! Small Minim handling is done automatically in T32 Software without defining Minim address ranges.

SYStem.state Display SYStem.state window

Format:	**SYStem.state**

Displays the **SYStem.state** window for system settings that configure debugger and target behavior.

TrOnchip.RESet — Set on-chip trigger to default state

Format:	**TrOnchip.RESet**

Sets the TrOnchip settings and trigger module to the default settings.

TrOnchip.state — Display on-chip trigger window

Format:	**TrOnchip.state**

Opens the **TrOnchip.state** window.

Target Adaption

Interface Standards JTAG, Serial Wire Debug, cJTAG

The Debug Cable supports the JTAG (IEEE 1149.1) interface standard.

Connector Type and Pinout

Debug Cable

Adaption for ARM Debug Cable: See **https://www.lauterbach.com/adarmdbg.html**. These adaptations also cover the Meta possibilities.

Mechanical description of the 20-pin Debug Cable:

Signal	Pin	Pin	Signal
VREF-DEBUG	1	2	VSUPPLY (not used)
TRST-	3	4	GND
TDI	5	6	GND
TMS|TMSC|SWDIO	7	8	GND
TCK|TCKC|SWCLK	9	10	GND
RTCK	11	12	GND
TDO|-|SWO	13	14	GND
RESET-	15	16	GND
DBGRQ	17	18	GND
DBGACK	19	20	GND

For details on logical functionality, physical connector, alternative connectors, electrical characteristics, timing behavior and printing circuit design hints, refer to **“ARM JTAG Interface Specifications”** (app_arm_jtag.pdf).