LAUTERBACH
DEVELOPMENT TOOLS

Debugger for eSi-RISC

MANUAL

Debugger for eSi-RISC

Version 06-Jun-2024

Introduction

This manual serves as a guideline for debugging one or multiple eSi-RISC cores via TRACE32.

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.

PRACTICE Script Language:

- **“Training Script Language PRACTICE”** (training_practice.pdf)
- **“PRACTICE Script Language Reference Guide”** (practice_ref.pdf)

Demo and Start-up Scripts

Lauterbach provides ready-to-run start-up scripts for known eSi-RISC 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/esirisc/` 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 JTAG Debugger

Starting up the debugger is done as follows:

1. Select the device prompt for the ICD Debugger and reset the system.

   ```
   B::
   
   RESet
   ```

   The device prompt `B::` is normally already selected in the TRACE32 command line. If this is not the case, enter `B::` to set the correct device prompt. The **RESet** command is only necessary if you do not start directly after booting the TRACE32 development tool.

2. Specify the CPU specific settings.

   ```
   SYStem.CPU ESI3200
   ```

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

3. Enter debug mode.

   ```
   SYStem.Up
   ```

   This command resets the CPU and enters debug mode. After this command is executed, it is possible to access memory and registers.

4. Load the program.

   ```
   Data.LOAD <file>                    ;load the compiler output.
   ```

   The format of the **Data.LOAD** command depends on the file format generated by the compiler.

   A detailed description of the **Data.LOAD** command and all available options is given in the **"General Commands Reference"**.

A simple start sequence without EPROM simulator is shown below. This sequence can be written to a PRACTICE script file (*.cmm, ASCII format) and executed with the command **DO** *<file>*.

```
B::                                  ; Select the ICD device prompt

WinCLEAR                             ; Clear all windows

SYStem.Up                            ; Reset the target and enter debug
                                     ; mode

Data.LOAD.elf my_project             ; Load the application
```

```
Register.Set PC _ResetVector        ; Set the program counter (PC) to
                                    ; start point

List.Mix                            ; Open source code window *)

Register.view /SpotLight            ; Open register window *)

Frame.view /Locals /Caller          ; Open the stack frame with
                                    ; local variables *)

Var.Watch %SpotLight flags ast      ; Open watch window for variables *)
```

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

Troubleshooting

SYStem.Up Errors

The **SYStem.Up** command is the first command of a debug session where communication with the target is required. If you receive error messages while executing this command, this may have the following reasons:

- The target has no power.
- The target is in reset.
- The core is not enabled.
- There is logic added to the JTAG state machine.
- There are additional loads or capacities or serial resistors on the JTAG lines.
- There is a short circuit on at least one of the output lines of the core.
- There are stubs on the signal line.

FAQ

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

eSi-RISC Specific Implementations

Access Classes

For background information about the term *access class*, see **"TRACE32 Glossary"** (glossary.pdf).

The following eSi-RISC specific access classes are available.

Access Class	Description
P	Program memory access
D	Data memory access
E	Run-time memory access (see **SYStem.CpuAccess** and **SYStem.MemAccess**)
CSR	CSR (Control and Status Register) access

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

Example:

```
Data.dump D:0x0--0x3
```

Breakpoints

There are two types of breakpoints available: Software breakpoints and on-chip breakpoints.

Software Breakpoints

Software breakpoints are the default breakpoints for program breakpoints. A software breakpoint is implemented by patching a break code into the memory.

There is no restriction in the number of software breakpoints.

On-chip Breakpoints

The resources for the on-chip breakpoints are provided by the CPU.

The following list gives an overview of the on-chip breakpoints for the eSi-RISC:

- **On-chip breakpoints:** Total amount of available on-chip breakpoints.
- **Instruction breakpoints:** Number of on-chip breakpoints that can be used to set Program breakpoints into ROM/FLASH/EEPROM.
- **Read/Write breakpoints:** Number of on-chip breakpoints that can be used as Read or Write breakpoints.
- **Data breakpoint:** Number of on-chip data breakpoints that can be used to stop the program when a specific data value is written to an address or when a specific data value is read from an address.

	On-chip Breakpoints	**Instruction Breakpoints**	**Read/Write Breakpoints**	**Data Breakpoint**
eSi-RISC	up to 8 instruction up to 8 read/write	up to 8 single address	up to 8 single address or up to 4 ranges	—

SYStem.CONFIG.state

Display target configuration

Format:	**SYStem.CONFIG.state** [**/**<*tab*>]
<*tab*>:	**DebugPort** | **Jtag**

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.

Alternatively, you can modify the target configuration settings via the TRACE32 command line with the **SYStem.CONFIG** commands. Note that the command line provides *additional* **SYStem.CONFIG** commands for settings that are *not* included in the **SYStem.CONFIG.state** window.

<*tab*>	Opens the **SYStem.CONFIG.state** window on the specified tab. For tab descriptions, see below.
DebugPort (default)	The **DebugPort** tab informs the debugger about the debug connector type and the communication protocol it shall use. For descriptions of the commands on the **DebugPort** tab, see **DebugPort**.
Jtag	The **Jtag** tab informs the debugger about the position of the Test Access Ports (TAP) in the JTAG chain which the debugger needs to talk to in order to access the debug and trace facilities on the chip. For descriptions of the commands on the **Jtag** tab, see **Jtag**.

Format:	**SYStem.CONFIG** *<parameter>*
<parameter>: **(DebugPort)**	**CORE** *<core> <chip>* **CoreNumber** *<number>* **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 and trace 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 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,...).
CoreNumber *<number>*	Number of cores to be considered in an SMP (symmetric multiprocessing) debug session. There are eSi-RISC core types which can be used as a single core processor or as a scalable multicore processor of the same type. If you intend to debug more than one such core in an SMP debug session you need to specify the number of cores you intend to debug. Default: 1.

DEBUGPORT [**DebugCable0**]	It specifies which probe cable shall be used e.g. “DebugCable0”. At the moment only the CombiProbe allows to connect more than one probe cable. 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. 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. 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. 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. Default: 0.
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.

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.

SYStem.CPU   Select the used CPU

Format:	**SYStem.CPU** *<cpu>*
<cpu>:	**ESI3200**

ESI3200 is the *default* entry for eSi-RISC. If you want to extend this about your CPU or SoC, please contact technical support.

SYStem.JtagClock   Define JTAG frequency

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

Default frequency: 10 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   Tristate the JTAG port

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

Default: OFF.

If the system is locked, no access to the JTAG port will be performed by the debugger. While locked the JTAG connector of the debugger is tristated. The intention of the **SYStem.LOCK** command is, for example, to give JTAG access to another tool. The process can also be automated, see **SYStem.CONFIG TriState**.

It must be ensured that the state of the RISC-V DTM JTAG state machine remains unchanged while the system is locked. To ensure correct hand-over, the options **SYStem.CONFIG TAPState** and **SYStem.CONFIG TCKLevel** must be set properly. They define the TAP state and TCK level which is selected when the debugger switches to tristate mode.

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

Default: Denied.

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)	Memory access during program execution to target is enabled.
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.

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

```
Data.dump E:0x100

Var.View %E first
```

SYStem.Mode — Establish the communication with the 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** **Go** **Attach** **Up** **NoDebug**

Down (default)	Disables the debugger. The state of the CPU remains unchanged. The JTAG port is tristated.
Go	Initializes a debug connection, resets the target and lets the CPU run from its reset vector.
Attach	Initializes a debug connection. The target is *not* reset, i.e. the state of the target is *not* changed. Consequently the user program stays running if it was running, or stays stopped if it was stopped.
Up	Initializes a debug connection, resets the target, sets the CPU to debug mode and stops the CPU at its reset vector.
StandBy	Not supported.
NoDebug	Disables the debugger. The state of the CPU remains unchanged. The JTAG port is tristated.

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.GPREG Configure number of GP registers

Format: **SYStem.Option.GPREG** *<number_of_registers>*

Default: 16.

Defines the number of GP registers in current core. Allowed values are 8, 16 and 32.

SYStem.state Display SYStem.state window

Format: **SYStem.state**

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

CPU specific TrOnchip Commands

The **TrOnchip** command group is not available for the eSi-RISC debugger.

Connector Type and Pinout

It is recommended to connect all N/C pins to GND (if you work with LAUTERBACH tools only).

Signal	Pin	Pin	Signal
TDO	1	2	N/C
TDI	3	4	TRST- (*)
N/C	5	6	VCCS
TCK	7	8	N/C
TMS	9	10	N/C
SRST-	11	12	N/C
N/C	13	-	KEY
N/C	15	16	GND

This is a standard 16-pin double row (two rows of eight pins) connector (pin-to-pin spacing: 0.100 in.).