LAUTERBACH
DEVELOPMENT TOOLS

Mico32 Debugger

MANUAL

Mico32 Debugger

Version 06-Jun-2024

History

07-Mar-2019 Initial version of the manual.

Introduction

This manual serves as a guideline for debugging Mico32 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 Mico32-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/mico32/ 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 at the Mico32-EBA (exception base address). 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 MICO32            ; Set target CPU, here the generic
                             ; MICO32 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.

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**.
- The debug monitor running on the target is not supported by the debugger. Make sure a supported **debug monitor** was programmed onto the target.
- 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.

Mico32 specific Implementations

Mico32 Configuration

The hardware feature set of the Mico32 architecture is highly configurable and customizable. Configuration possibilities range from memories and computational features to the debug infrastructure. That means, some of the debug features discussed here might not be available for every Mico32 implementation.

Mico32 Debug Monitor

In order to debug a Mico32 target, a debug monitor is required. The debug monitor is a software program which executes on the target whenever the target receives a halt request, e.g. by a breakpoint or a user initiated break. The debug monitor then communicates with the debugger, which allows access to the target system. Therefore, the debug monitor capabilities have a direct influence on the debugger capabilities.

Lauterbach provides a debug monitor which is compatible with the debug driver of the generic **MICO32** selection of the **SYStem.CPU** command. The debug monitor is designed to support all basic and advanced debug features offered by the Mico32 architecture. The debug monitor can be found in the demo folder `~~/demo/mico32/monitor/debug_monitor`.

The Lauterbach debug monitor requires 2KiB of memory and must be loaded to the Mico32 *Debug exception base address* (DEBA). In general, the debug monitor code must be present in the target memory before the debugger can be used. However, if the Mico32 hardware-based debugging support is available for the specific target, then it is also possible to hot-load the debug monitor while the target is running. Limitations due to the used memory type for the DEBA memory region may still apply.

Debug monitor hot-loading:

```
; Configuration, e.g. CPU selection and JTAG settings
; ...

; Attach to the target
SYStem.Attach

; Load a debug monitor
; Run-time access class E: must be used as an offset parameter
Data.LOAD.Elf <debug_monitor_file> E:

; ... your code
```

Access Classes

Access classes are used to specify which memory to access. For background information about the term access class, see **"TRACE32 Concepts"** (trace32_concepts.pdf).

The following common access classes have the same meaning for all CPUs of the Mico32 architecture.

Access Class	Description
P	Program or data memory access. Target implementation defined.
D	Data memory access
CSR	Control and status register access.
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(CSR:0x6)
```

P Access Class

The Mico32 architecture is a Harvard-type processor architecture. It has two separate buses, an instruction WHISBONE bus for program memory accesses and a data WHISBONE bus for data memory accesses. Whether the debugger's P access class is able to provide access to the program memory or not, is defined by the target implementation. The ability to provide program memory access depends on the memory mapping of the targeted Mico32 and if the Mico32 hardware-based debugging support is available.

Hardware-based debugging support available

If hardware-based debugging support is available, the instruction WHISBONE bus of the Mico32 architecture is exposed and can be accessed by the debugger. This allows the debugger to access all connected program memories.

Run-time memory access via the EP access class is possible without stopping the CPU. Although the access via the instruction WHISBONE bus is non-intrusive at run-time, the access can still have an effect on the CPU and the debugging behavior. This is due to the following limitations:

- The access may slow down the target execution.
- The access will generate instruction bus errors when unmapped memory regions are accessed.
- Write accesses might not take immediate effect because of caching effects.

If required, the direct instruction WHISBONE access can be disabled via the command **SYStem.Option.AllowDirectIWAccess**. The system then behaves as if the Mico32 hardware-based debugging support was not available. This is helpful to detect errors, as it allows target debugging when only the instruction WHISBONE access of the debugger fails.

Hardware-based debugging support not available

If the hardware-based debugging support is not available, memory access for the debugger is only available via the data WHISBONE bus of the Mico32. In this case, the P access class is mapped to the D access class and both will produce the same results. A full access to all program memories might still be possible if the instruction and data WHISBONE buses have an identical memory mapping. Non-intrusive run-time memory access via the EP access class is no longer possible.

CSR Access Class

The CSR access class allows access to the *control and status* registers of the Mico32 core. Limitations regarding reading and writing access are the same as specified in the official processor reference manual.

The address specified for the access corresponds to the official index of the respective CSR. The access width of a CSR is always 32 bits.

Examples:

```
Data.Set CSR:0x0 0x0              ; Writing access to the IE register

Data.Set CSR:0x1 0x18             ; Writing access to the IM register

PRINT Data.Long(CSR:0x6)          ; Reading access to the CFG register
```

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 (Mico32 *break* 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 Mico32 supports up to 4 program on-chip breakpoints. These breakpoints are single address only. The debugger is able to detect the number of available on-chip breakpoints by analyzing the contents of the *CFG* (Configuration) control and status 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 Mico32 core into debug mode. On-chip breakpoints are break before make.

Examples:

```
Break.Set 0x08000024 /Program          ; Configures an on-chip breakpoint
/Onchip                                ; which activates when the program
                                       ; counter matches 0x08000024

Break.Set 0x08000024 /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 Mico32 supports up to 4 data on-chip watchpoints. These watchpoints are single address only. The debugger is able to detect the number of available on-chip watchpoints by analyzing the contents of the *CFG* (Configuration) control and status 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 Mico32 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 0x08001000 /Read /Onchip    ; Configures an on-chip read
                                      ; watchpoint which activates when
                                      ; a load instruction accesses
                                      ; address 0x08001000

Break.Set 0x08001000 /Read            ; Same as above, since read
                                      ; breakpoints are always on-chip

Break.Set 0x08001000 /Write           ; Write watchpoint for store
                                      ; instructions

Break.Set 0x08001000 /ReadWrite       ; ReadWrite watchpoint for load and
                                      ; store instructions
```

Mico32 specific Event for the ON and GLOBALON Command

TRACE32 can be programmed to detect CPU specific events and execute a user-defined *<action>* in response to the detected *<event>*. The user-defined action is a PRACTICE script (*.cmm).

The following commands and CPU specific events are available:

GLOBALON *<event>* [*<action>*]	Global event-controlled PRACTICE script execution. The event is detectable during an entire TRACE32 session.
ON *<event>* [*<action>*]	Event-controlled PRACTICE script execution. The event is detectable only by a particular PRACTICE script.

CPU specific <event>	Description
PDRESETOFF	The target performs a transition from the *power down reset* state into any other target state. The availability of the event is target CPU dependent and can only be used if **SYStem.Option.CorePowerDetection** is enabled.

SETUP.DIS Disassembler configuration

Format:	**SETUP.DIS** [*<fields>*] [*<bar>*] [*<constants>*]
<constants>:	[**RegNames** | **AliasNames**] [*<other_constants>*]

Sets **default values** for configuring the disassembler output of **newly opened windows**. Affected windows and commands are **List.Asm** and **Register.view**.

The command does **not affect existing windows** containing disassembler output.

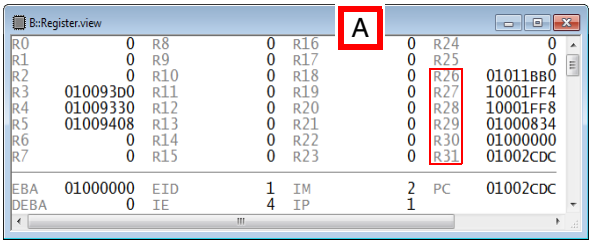
<fields>, *<bar>*, *<constants>*	For a description of the generic arguments, see **SETUP.DIS** in **general_ref_s.pdf**.
AliasNames	Use the *alias* naming scheme for the names of the Mico32 general purpose registers.
RegNames (default naming scheme)	Use the generic *register number* (r0, r1, …, r31) naming scheme for the names of the Mico32 general purpose registers.

Example 1: The changed naming scheme takes immediate effect in the **Register.view** window.

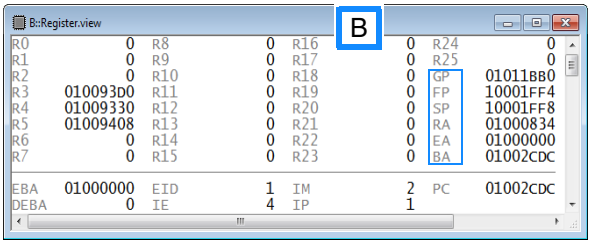
```
SETUP.DIS RegNames      ;by default, the register number naming scheme is
                        ;used for the general purpose registers
Register.view           ;let's open a register window

;... your code

SETUP.DIS AliasNames    ;let's now switch the naming scheme of the general
                        ;purpose registers to the alias naming scheme
```



A Register number naming scheme (default naming scheme).



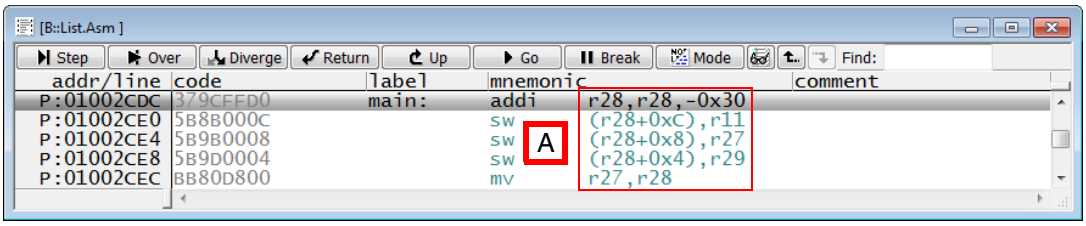
B Alias naming scheme. The alias names are also available in **Register.Set**.

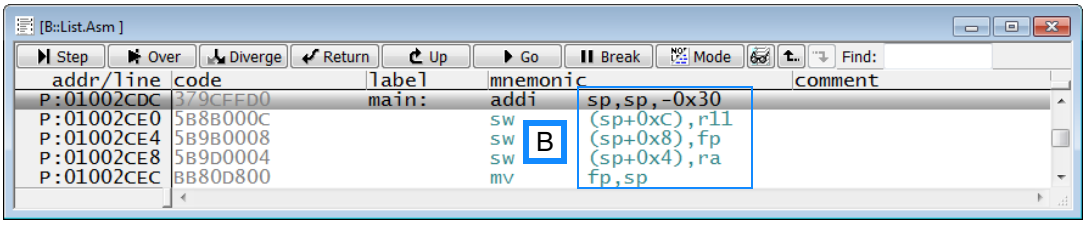
Example 2: The changed naming scheme does **not affect an existing List.Asm** window. You need to open another **List.Asm** window to view the changed naming scheme.

```
SETUP.DIS RegNames
List.Asm

;... your code

SETUP.DIS AliasNames
List.Asm                    ;open another disassembler output window
```





A Register number naming scheme (default naming scheme).

B Alias naming scheme. The alias names are also available in **Register.Set**.

NOTE:	The command **Register.Set** and the function **Register()** always accept the generic name and the alias name as a register parameter. They are not affected by this **SETUP** command.

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>:	**CORE** *<core> <chip>*
(DebugPort)	**DEBUGPORT** [**DebugCable0**]
	DEBUGPORTTYPE [**JTAG**]
	Slave [**ON** | **OFF**]
	TriState [**ON** | **OFF**]
<parameter>:	**DRPOST** *<bits>*
(JTAG)	**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.

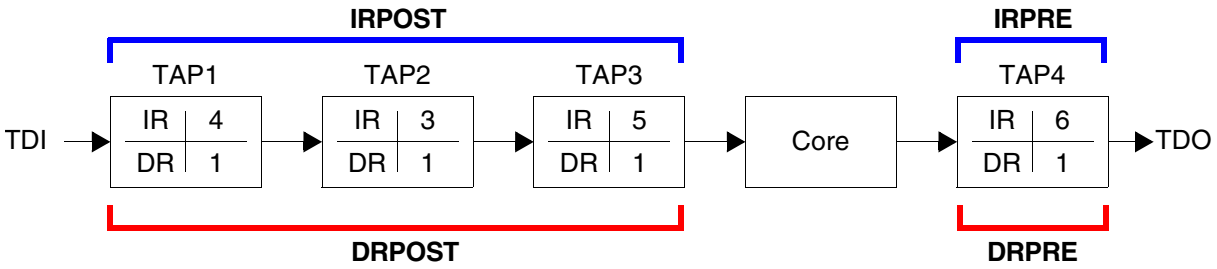
TriState [**ON** | **OFF**]

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.

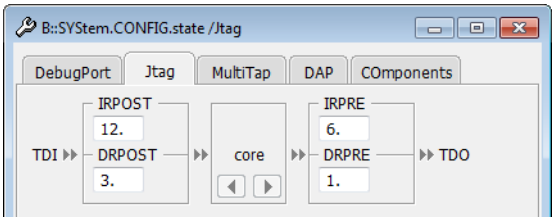
Default: OFF.

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.



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.

Format:	**SYStem.CPU** *<cpu>*
<cpu>:	**LATTICE | MICO32**

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

LATTICE	Generic CPU for LATTICE Mico32 targets. Supports the LATTICE debug monitor.
MICO32	Generic CPU for Mico32 targets. Supports the debug monitor provided by Lauterbach. For more information, see **Mico32 Debug Monitor**.

SYStem.JtagClock Define JTAG frequency

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

Default frequency: 1 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.

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.

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

Default: CPU.

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.

NOTE:	Non-intrusive run-time memory access for the Mico32 is only possible for the P access class and only if Mico32 hardware-based debugging support is available. For more information, see chapter **P Access Class**.

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
```

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

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
StandBy Not available.

SYStem.Option.AllowDirectIWAccess Allow direct instruction bus access

Only available for some CPUs.

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

Default: ON.

Enables or disables the debugger's ability to directly access the instruction WHISBONE bus of the target. Consequently, the command has a direct influence on the P access class. For more information, see chapter **P Access Class**. This option only has an effect if the Mico32 hardware-based debugging support is available.

SYStem.Option.CacheCoherentACCESS Second level cache settings

Only available for some CPUs.

Format: **SYStem.Option.CacheCoherentACCESS** *<parameter>*

<parameter>: **ENABLE** [**ON** | **OFF**]
CFGREGADDRESS *<address>*

Default **ENABLE**: OFF.
Default **CFGREGADDRESS**: 0x800.

Enables or disables cache-coherent memory access in target systems with second-level instruction and data caches. If enabled, the P and D access classes will yield the same results when shared memory locations are accessed. If disabled, memory access results may differ depending on the cache status.

ENABLE [**ON** | **OFF**] Enables or disables cache-coherent memory access.

CFGREGADDRESS *<address>* Specifies the address of the target register that is responsible for enabling and disabling the settings for cache-coherent memory access. If not specified correctly, **SYStem.Option.CacheCoherentACCESS ENABLE** will fail or will have no effect.

SYStem.Option.CorePowerDetection Special core power detection

Only available for some CPUs.

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

Default: ON.

Enables or disables the core power detection. If enabled, special target registers will be polled to detect the power status of the target core.

When this option is enabled, the special PRACTICE event **PDRESETOFF** becomes available.

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

Default: ON.

All TRACE32 windows that display memory are updated while the processor is executing code (e.g. **Data.dump**, **List.Mix**, **PER.view**, **Var.View**). This setting has no effect if **SYStem.Option.MemAccess** is disabled.

If only selected memory windows should update their content during runtime, leave **SYStem.Option.DUALPORT OFF** and use the access class prefix **E** or the format option **%E** for the specific windows.

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. If you want to halt in the interrupt routine, use **TrOnchip.StepVector** **ON**.

SYStem.Option.STEPSOFT Use alternative method for ASM single step

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

Default: ON.

Normally, the default value does not need to be changed. Turning this option **OFF** might yield in strange stepping-behavior if not fully supported by the target.

ON Regular assembler single-stepping on the Mico32 is achieved via software breakpoints.

OFF Allows to make use of Mico32's hardware-based single step functionality if hardware-based debugging support is available for the target implementation.

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.

TrOnchip.StepVector — Halt on exception entry when single-stepping

Format: **TrOnchip.StepVector** [**ON** | **OFF**]

Default: OFF.

Determines the behavior of a single step when an exception or an interrupt occurs.

ON	A breakpoint will be set on the interrupt exception vector when *single-stepping* through code. This is helpful to check if interrupts occur while single-stepping.
OFF	Halts on the next instruction.

Target Adaption

Probe Cables

For debugging, the following kinds of probe cables can be used to connect the debugger to the target:

- Debug Cable
- n/a

Interface Standards JTAG

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 adaptions also cover the Mico32 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).