LAUTERBACH
DEVELOPMENT TOOLS

Beyond Debugger and Trace

MANUAL

Beyond Debugger and Trace

Beyond Debugger and Trace

Version 06-Jun-2024

History

20-Jul-22	For the MMU.SCAN ALL command, CLEAR is now possible as an optional second parameter.

Introduction

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.

Demo and Start-up Scripts

Lauterbach provides ready-to-run start-up scripts for known Beyond 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/beyond/` subfolder of the system directory of TRACE32.

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.

Limitations

- Beyond processors of the BA1x family are not supported.

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 <cpu_type>
   ```

   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. Inform the debugger about read-only address ranges (ROM, FLASH).

   ```
   MAP.BOnchip 0xF0000000++0x01ffffff
   ```

   The B(reak)Onchip information is necessary to decide where on-chip breakpoints must be used. On-chip breakpoints are necessary to set program breakpoints to FLASH/ROM.

4. Select JTAG interface (ORIGINAL or FAST interface is available for some devices like BA22)

   ```
   SYStem.CONFIG.DebugProtocol (ORIGINAL|FAST)
   ```

   Note: This settings preselects also the most suitable Memory Access Module. For more information please check **SYStem.CONFIG.MemoryAccessModule**.

5. Set endianness to match your targets endianness

   ```
   SYStem.Option.LittleEnd (ON|OFF)
   ```

6. 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.

7. Load the program.

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

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 Reference Guide"**.

In case your program should be loaded to flash please read section about flash programming first.

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>*. Addresses and address ranges are only examples and not guaranteed to work. See also `~~/demo/beyond/hardware/ba22/custom/` for a template script.

```
WinCLEAR                          ; Clear all windows

SYStem.CPU BA22                   ; Select the core type

SYStem.CONFIG.DebugProtocol       ; Select fast JTAG interface (the
FAST                              ; target must support this!)

SYStem.Option.LittleEnd OFF       ; Select big endian mode for target

MAP.BOnchip                       ; Specify where FLASH/ROM is
0xf0000000++0x01ffffff

SYStem.Up                         ; Reset the target and enter debug mode

Data.LOAD.ELF sieve.elf           ; Load the application

Register.Set pc main              ; Set the PC to function main

List.Mix                          ; Open source code window          *)

Register.view /SpotLight          ; Open register window             *)

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

Var.Watch ast                     ; Add variable ast to watch window *)

Break.Set 0x00001000 /p           ; Set software breakpoint to address
                                  ; 00001000 (address outside of BOnchip
                                  ; range)

Break.Set 0xf0040000 /p           ; Set on-chip breakpoint to address
                                  ; f0040000 (Within BOnchip range)
```

*) 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 below. “target processor in reset” is just a follow-up error message. Open the **AREA.view** window to see all error messages.

- The target has no power or the debug cable is not connected to the target. This results in the error message “target power fail”.
- You did not select the correct core type **SYStem.CPU** *<type>* or a wrong system option (endianness, jtag interface type etc.).
- There is an issue with the JTAG interface. See 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.
- There is the need to enable (jumper) the debug features on the target. It will for example not work if nTRST signal is directly connected to ground on target side.
- The target is in an unrecoverable state. Re-power your target and try again.
- The target can not communicate with the debugger while in reset. Try **SYStem.Mode Attach** followed by **Break** instead of **SYStem.Up**.
- 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.
- The core is used in a multicore system and the appropriate settings for the debugger are missing. See for example **SYStem.CONFIG IRPRE**. This is the case if you get a value IR_Width > 4 when you enter “DIAG 16001” and “AREA”. If you get IR_Width = 4, then you have just your core and you do not need to set these options. If the value can not be detected, then you might have a JTAG interface issue.
- The core has no clock.
- The core is kept in reset.
- Your target needs special debugger settings. Check the directory ~~/demo/beyond/ if there is an suitable script file *.cmm for your target.

FAQ

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

Beyond Specific Implementations

Breakpoints

Software Breakpoints

If a software breakpoint is used, the original code at the breakpoint location is patched by a breakpoint code.

There is no restriction in the number of software breakpoints. Simple Breakpoints can be software breakpoint. Complex breakpoints (e.g. range, read/write etc.) are realized as onchip breakpoints.

On-chip Breakpoints for Instructions

If on-chip breakpoints are used, the resources to set the breakpoints are provided by the CPU. On-chip breakpoints are usually needed for instructions in FLASH/ROM.

With the command **MAP.BOnchip** *<range>* it is possible to tell the debugger where you have ROM / FLASH on the target. If a breakpoint is set into a location mapped as BOnchip one on-chip breakpoint will be used.

On-chip Breakpoints for Data

To stop the CPU after a read or write access to a memory location on-chip breakpoints are required. Overview:

- **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/EPROM.
- **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

Core	On-chip Breakpoints	Instruction Breakpoints	Read/Write Breakpoints	Data Breakpoint
BA22	up to 8	up to 8	up to 8	up to 4
JN5148	up to 4	up to 4	up to 4	up to 2
JN5168	up to 4	up to 4	up to 4	up to 2

- The number of available Breakpoints depends on the CPU configuration, i.e. available resources, and on the complexity of a set breakpoint (ranges etc.).

Example for Standard Breakpoints

Assume you have a target with

- FLASH from `0xf0000000--0xf1ffffff`
- SDRAM from `0xc0000000--0xc00fffff`

The command to configure TRACE32 correctly for this configuration is:

```
  Map.BOnchip 0xf0000000--0xf1ffffff
```

The following standard breakpoint combinations are possible.

1. Unlimited breakpoints in RAM and up to eight breakpoints in ROM/FLASH

```
Break.Set 0xc0000000 /Program          ; Software breakpoint 1
Break.Set 0xc0001000 /Program          ; Software breakpoint 2
Break.Set sram_addr /Program           ; Software breakpoint 3
Break.Set 0xf0000100 /Program          ; On-chip breakpoint
```

2. Unlimited breakpoints in RAM and one breakpoint in RAM on a read or write access

```
Break.Set 0xc0000000 /Program          ; Software breakpoint 1
Break.Set 0xc0001000 /Program          ; Software breakpoint 2
Break.Set sram_addr /Program           ; Software breakpoint 3
Break.Set 0xc0002000 /Write            ; On-chip breakpoint
```

3. Two breakpoints in ROM/FLASH

```
Break.Set 0xf0000100 /Program          ; On-chip breakpoint 1
Break.Set 0xf0000200 /Program          ; On-chip breakpoint 2
```

4. Two breakpoints on a read or write access

```
Break.Set 0xc0008000 /Write            ; On-chip breakpoint 1
Break.Set 0xc0008010 /Read             ; On-chip breakpoint 2
```

5. One breakpoint in ROM/FLASH and one breakpoint on a read or write access

```
Break.Set 0xf0000100 /Program          ; On-chip breakpoint 1
Break.Set 0xc0008010 /Read             ; On-chip breakpoint 2
```

Runtime Measurement

The command **RunTime** allows run time measurement based on polling the CPU run status by software. Therefore the result will be about few milliseconds higher than the real value.

Memory Classes

The following Beyond specific memory classes are available.

Memory Class	Description
P	Program Memory
D	Data Memory
S	Supervisor Memory (privileged access)
U	User Memory (non-privileged access) If supported by core.
J	Java code, currently not applicable for beyond
A	Absolute addressing (physical address) without MMU
ANC	Physical access without cache and MMU
DC	Data Memory as seen through Data Cache
NC	Memory seen with cache switched off
SPR	System Registers (addressing see below)
VM	Virtual Memory (memory on the debug system)
E	Run-time memory access (see **SYStem.CpuAccess** and **SYStem.MemAccess**)

SYStem.CONFIG Configure debugger according to target topology

Format:	**SYStem.CONFIG** *<parameter>*
<parameter>: **(General)**	**state** **IRPRE** *<bits>* **IRPOST***<bits>* **DRPRE** *<bits>* **DRPOST** *<bits>* **Slave** [**ON** | **OFF**] **TAPState** *<state>* **TCKLevel** *<level>* **TriState** [**ON** | **OFF**]
<parameter>: **(vectors)**	**ReSeTException** *<address>* **BUSErrorException** *<address>* **DataPageFaultException** *<address>* **InstrPageFaultException** *<address>* **TickTimerException** *<address>* **AlignmentException** *<address>* **IllegalInstrException** *<address>* **INTerruptException** *<address>* **DtlbMissException** *<address>* **ItlbMissException** *<address>* **RangeException** *<address>* **SystemCallException** *<address>* **FloatingPointException** *<address>* **TrapException** *<address>*

If there is more than one TAP controller in the JTAG chain, the chain must be defined to be able to access the correct TAP controller.

The four parameters IRPRE, IRPOST, DRPRE, DRPOST are required to inform the debugger of the TAP controller position in the JTAG chain if there is more than one core in the JTAG chain. The information is required before the debugger can be activated, e.g. via **SYStem.Mode.Attach**.

state	Show **SYStem.CONFIG** settings window.
DRPRE *<bits>*	(default: 0) *<number>* of TAPs in the JTAG chain between the core of interest and the TDO signal of the debugger. If each core in the system contributes only one TAP to the JTAG chain, DRPRE is the number of cores between the core of interest and the TDO signal of the debugger.
DRPOST *<bits>*	(default: 0) *<number>* of TAPs in the JTAG chain between the TDI signal of the debugger and the core of interest. If each core in the system contributes only one TAP to the JTAG chain, DRPOST is the number of cores between the TDI signal of the debugger and the core of interest.
IRPRE *<bits>*	(default: 0) *<number>* of instruction register bits in the JTAG chain between the core of interest and the TDO signal of the debugger. This is the sum of the instruction register length of all TAPs between the core of interest and the TDO signal of the debugger.
IRPOST *<bits>*	(default: 0) *<number>* of instruction register bits in the JTAG chain between the TDI signal and the core of interest. This is the sum of the instruction register lengths of all TAPs between the TDI signal of the debugger and the core of interest.
Slave [**ON** | **OFF**]	(default: OFF) If more than one debugger share the same JTAG port, all except one must have this option active. Only one debugger - the “master” - is allowed to control the signals nTRST and nSRST (nRESET).
TAPState *<state>*	(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.
TCKLevel [**0** | **1**]	(default: 0) Level of TCK signal when all debuggers are tristated (e.g. pull-down => TCKLevel=0.).
TriState [**ON** | **OFF**]	(default: OFF) If more than one debugger share the same JTAG port, this option is required. The debugger switches to tristate mode after each JTAG access. Then other debuggers can access the port.

Vectors - Core-specific Sub-Commands

The BA2x Architecture features configurable exception vectors while synthesis time. The following list of SYStem.CONFIG commands allows to configure the exception vector base addresses for your system.

The short form of the subcommands is equal to the **TrOnchip.view** window and commands

ReSeTException *<address>*	Exception Vector for Reset Exception.
BUSErrorException *<address>*	Exception Vector for Bus Error Exception.
DataPageFaultException *<address>*	Exception Vector for Data Page Fault Exception.
InstrPageFaultException *<address>*	Exception Vector for Instruction Page Fault Exception.
TickTimerException *<address>*	Exception Vector for Tick Timer Exception.
AlignmentException *<address>*	Exception Vector for Alignment Exception.
IllegalInstrException *<address>*	Exception Vector for Illegal Instruction Exception.
INTerruptException *<address>*	Exception Vector for external Interrupt.
DtlbMissException *<address>*	Exception Vector for Data TLB Miss Exception.
ItlbMissException *<address>*	Exception Vector for Instruction TLB Miss Exception.
RangeException *<address>*	Exception Vector for Range Exception.
SystemCallException *<address>*	Exception Vector for System Call Exception.
FloatingPointException *<address>*	Exception Vector for Floating Point Exception.
TrapException *<address>*	Exception Vector for Trap Event Exception.

SYStem.CONFIG.DebugProtocol — Implemented debug protocol of the CPU

Format:	**SYStem.CONFIG.DebugProtocol** [**ORIGINAL** | **FAST**] **SYStem.Option.FASTJTAG** [**ON** | **OFF**] (obsolete)

This option allows to select the implemented Debug Protocol of the CPU. If option FAST is selected the 34 bit interface is used. If option ORIGINAL is selected the 80 bit interface is used. Which protocol is supported by the target is core implementation specific.

For supported derivatives of Beyond BA2x architecture which are selectable in the cpu list the suitable protocol is preselected as e.g. for JN5148 and JN5168.

ORIGINAL	JTAG Protocol "ORIGINAL" (80bit) is used
FAST	JTAG Protocol "FAST" (34bit) is used

SYStem.CONFIG.MemAccessModule — Select memory access module

Format:	**SYStem.CONFIG.MemAccessModule** [**WISHBONE** | **CPU**]

This option allows to select the between the implemented Memory Access Modules of the Beyond BA2x Architecture.

Which Memory Access Module is supported by the target is core implementation specific. For supported derivatives of the Beyond BA2x architecture which are selectable in the cpu list the suitable module is preselected.

WISHBONE	Use the Wishbone (Module 0) module to access physical memory. Standard for **SYStem.CONFIG.DebugProtocol ORIGINAL**.
CPU	Use the CPU (SPR Method) to access memory via Cache & MMU. Standard for **SYStem.CONFIG.DebugProtocol FAST**.

Format:	**SYStem.CPU** *<cpu>*
<cpu>:	**BA22** | **BA25** | **JN5148** | **JN5168**

Selects the processor type. If your chip is not listed, contact technical support.

Default selection: BA22

Format:	**SYStem.JtagClock** [<*frequency*> | **CTCK** <*frequency*>] **SYStem.BdmClock** <*frequency*> (obsolete)
<*frequency*>:	**4 kHz**…**100 MHz**

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. Therefore we recommend to use the default setting if possible.

<*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 “100kHz” or “15MHz” can also be used. The short forms imply a decimal value, although no “.” is used.
CTCK	With this option higher JTAG speeds can be reached. The TDO signal will be sampled by a signal which derives from TCK, but which is timely compensated regarding the debugger-internal driver propagation delays (**C**ompensation by **TCK**). The debugger sets CTCK by default.

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 Beyond 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.

Please note: nTRST must have a pull-up resistor on the target.

SYStem.MemAccess — Select run-time memory access method

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

Default: Denied.

There's no possibility to access memory while CPU is running. So only **Denied** or **StopAndGo** can be used. The debugger can access memory only if the CPU is stopped.

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**
	NoDebug
	Go
	Attach
	Up

Down	Disables the debugger (default). The state of the CPU remains unchanged. The JTAG port is tristated.
NoDebug	Resets the CPU and disables the debugger. The CPU can start without any interference of the debug interface. Debug Mode can be enabled with the use of an "Attach". The JTAG port is tristated.
Go	Resets the target, enables the debugger and starts the program execution. Program execution can be stopped by the break command or external trigger.
Attach	User program remains running (no reset) and the debug mode is activated. After this command the user program can be stopped with the break command or if any break condition occurs.

StandBy	You need to be in DOWN state when switching to this mode. It resets and starts the program when power is detected. Halt the program execution and set all the breakpoints and trace conditions you need, then re-start the program. Now you can even debug a power cycle, because debug register (breakpoints and trace control) will be restored on power up. This mode is not yet available.
Up	Resets the target, sets the CPU to debug mode and stops the CPU. After the execution of this command the CPU is stopped and all register are set to the default level.

SYStem.Option.DBGRQ Assert DBGRQ line while reset

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

This option allows to assert the DBGRQ line on the **JTAG connector** while **SYStem.Up** is executed. This is useful to signal e.g. an ROM Bootloader that an external Debug Request is pending (e.g. JN516x).

SYStem.Option.FLOWTRACE Debug support while FLOWTRACE

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

This option enables/disables support for debugging while a Off-Chip-Trace/Flowtrace is used. It changes the debuggers behavior in such a way that the Single-Step/Go/Breakpoint/User-Break information is added into the trace buffer. This option automatically enables **SETUP.StepBeforeGo**.

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.

If enabled, the Global Interrupt Enable Bit s(SR.IEE and SR.TEE) 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.

If disabled, a pending interrupt will be executed on a single-step, but it does not halt there i.e. the interrupt handler is always over stepped. If you want to halt in the interrupt routine, use **TrOnchip.StepVector** **ON**.

SYStem.Option.LittleEnd CPU endianness for memory access

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

Default: OFF.

If enabled, the accesses to memory are performed in Little-Endian (low-byte first) mode. Which mode is used by the CPU is Beyond Ba2x implementation specific.

Please note that the SPR: (Special Purpose Register) access-class is not affected by this setting as SPRs are always in Big-Endian notation.

SYStem.Option.LPMDebug Polling for low-power-mode

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

Default: OFF.

If enabled the debugger tries to continuously poll the device whether it is responding or not. A device which is in Low-Power-Mode/Sleep mode does not respond and is shown as "running (not responding)" in the status bar. Debugging is not possible in this status.

Further this option causes an continuous initialization of the debug registers which will improve the software breakpoint behavior after wake-up. Unfortunately some core types clear their ONCHIP-Breakpoint registers while Low-Power-Mode. So the target software must ensure that the ONCHIP-Breakpoints are reconstructed after wake-up.

SYStem.Option.MMUSPACES Separate address spaces by space IDs

Format: **SYStem.Option.MMUSPACES** [**ON** | **OFF**]
SYStem.Option.MMUspaces [**ON** | **OFF**] (deprecated)
SYStem.Option.MMU [**ON** | **OFF**] (deprecated)

Default: OFF.

Enables the use of space IDs for logical addresses to support **multiple** address spaces.

For an explanation of the TRACE32 concept of address spaces (zone spaces, MMU spaces, and machine spaces), see **“TRACE32 Concepts”** (trace32_concepts.pdf).

NOTE:	**SYStem.Option.MMUSPACES** should not be set to **ON** if only one translation table is used on the target. If a debug session requires space IDs, you must observe the following sequence of steps: 1. Activate **SYStem.Option.MMUSPACES**. 2. Load the symbols with **Data.LOAD**. Otherwise, the internal symbol database of TRACE32 may become inconsistent.

Examples:

```
;Dump logical address 0xC00208A belonging to memory space with
;space ID 0x012A:
Data.dump D:0x012A:0xC00208A

;Dump logical address 0xC00208A belonging to memory space with
;space ID 0x0203:
Data.dump D:0x0203:0xC00208A
```

SYStem.Option.ResetDURation — Reset assertion time

Format: **SYStem.Option.ResetDURation** [**AUTO** | <*time*>]

Default: AUTO.

Specifies the nRESET assertion time. Use this option if a reset pulse of more than 1ms is required.

SYStem.Option.TURBO — Speed up memory access

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

Default: OFF.

If TURBO is disabled the CPU checks after each system memory access in debug mode if the CPU has finished the corresponding cycle. This check will significantly reduce the down- and upload speed (30-40%).

If TURBO is enabled the CPU will make no checks. This may result in unpredictable errors if the memory interface is slow. Therefore it is recommended to use this option only for a program download and in case you know that the memory interface is fast enough to take the data with the speed it is provided by the debugger.

SYStem.Option.WaitReset — Wait with JTAG activities after deasserting reset

Format:	**SYStem.Option.WaitReset** [**AUTO** | *<time>*] [**/Poll**]

Default: AUTO.

AUTO	The debugger will try to autodetect if the JTAG daisy-chain works while nRESET is asserted. If yes, it will try to STOP the CPU while nRESET is asserted and then release nRESET. As a fallback, it will simply deassert nRESET, wait and then connect.
<time>	The debugger will NOT try to autodetect the most appropriate mode and simply use a deassert nRESET, wait <*time*> and then connect sequence.
<time> **/Poll**	The connection sequence will try to deassert nRESET and then to continuously poll/stop the target for *<time>* seconds. The option **Poll** might decrease the time between CPU start and debugger stop on some platforms.

SYStem.state — Display SYStem.state window

Format:	**SYStem.state**

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

MMU.DUMP

Page wise display of MMU translation table

Format:	**MMU.DUMP** *<table>* [*<range>* | *<address>* | *<range>* *<root>* | *<address>* *<root>*] **MMU.***<table>***.dump** (deprecated)
<table>:	**PageTable** **KernelPageTable** **TaskPageTable** *<task_magic>* | *<task_id>* | *<task_name>* | *<space_id>***:0x0** *<cpu_specific_tables>*

Displays the contents of the CPU specific MMU translation table.

- If called without parameters, the complete table will be displayed.
- If the command is called with either an address range or an explicit address, table entries will only be displayed if their **logical** address matches with the given parameter.

<root>	The *<root>* argument can be used to specify a page table base address deviating from the default page table base address. This allows to display a page table located anywhere in memory.
<range> *<address>*	Limit the address range displayed to either an address range or to addresses larger or equal to *<address>*. For most table types, the arguments *<range>* or *<address>* can also be used to select the translation table of a specific process if a space ID is given.
PageTable	Displays the entries of an MMU translation table. • if *<range>* or *<address>* have a space ID: displays the translation table of the specified process • else, this command displays the table the CPU currently uses for MMU translation.

KernelPageTable	Displays the MMU translation table of the kernel. If specified with the **MMU.FORMAT** command, this command reads the MMU translation table of the kernel and displays its table entries.
TaskPageTable *<task_magic>* | *<task_id>* | *<task_name>* | *<space_id>***:0x0**	Displays the MMU translation table entries of the given process. Specify one of the **TaskPageTable** arguments to choose the process you want. In MMU-based operating systems, each process uses its own MMU translation table. This command reads the table of the specified process, and displays its table entries. • For information about the first three parameters, see **"What to know about the Task Parameters"** (general_ref_t.pdf). • See also the appropriate **OS Awareness Manuals**.

CPU specific Tables in MMU.DUMP <table>

ITLB	Displays the contents of the Instruction Translation Lookaside Buffer.
DTLB	Displays the contents of the Data Translation Lookaside Buffer.

Format:	**MMU.List** *<table>* [*<range>* | *<address>* | *<range>* *<root>* | *<address>* *<root>*] **MMU.***<table>***.List** (deprecated)
<table>:	**PageTable** **KernelPageTable** **TaskPageTable** *<task_magic>* | *<task_id>* | *<task_name>* | *<space_id>***:0x0**

Lists the address translation of the CPU-specific MMU table.

- If called without address or range parameters, the complete table will be displayed.
- If called without a table specifier, this command shows the debugger-internal translation table. See **TRANSlation.List**.
- If the command is called with either an address range or an explicit address, table entries will only be displayed if their **logical** address matches with the given parameter.

<root>	The *<root>* argument can be used to specify a page table base address deviating from the default page table base address. This allows to display a page table located anywhere in memory.
<range> *<address>*	Limit the address range displayed to either an address range or to addresses larger or equal to *<address>*. For most table types, the arguments *<range>* or *<address>* can also be used to select the translation table of a specific process if a space ID is given.
PageTable	Lists the entries of an MMU translation table. • if *<range>* or *<address>* have a space ID: list the translation table of the specified process • else, this command lists the table the CPU currently uses for MMU translation.
KernelPageTable	Lists the MMU translation table of the kernel. If specified with the **MMU.FORMAT** command, this command reads the MMU translation table of the kernel and lists its address translation.
TaskPageTable *<task_magic>* | *<task_id>* | *<task_name>* | *<space_id>***:0x0**	Lists the MMU translation of the given process. Specify one of the **TaskPageTable** arguments to choose the process you want. In MMU-based operating systems, each process uses its own MMU translation table. This command reads the table of the specified process, and lists its address translation. • For information about the first three parameters, see **“What to know about the Task Parameters”** (general_ref_t.pdf). • See also the appropriate **OS Awareness Manuals**.

Format:	**MMU.SCAN** *<table>* [*<range>* *<address>*] **MMU.***<table>***.SCAN** (deprecated)
<table>:	**PageTable** **KernelPageTable** **TaskPageTable** *<task_magic>* | *<task_id>* | *<task_name>* | *<space_id>***:0x0** **ALL** [**Clear**] *<cpu_specific_tables>*

Loads the CPU-specific MMU translation table from the CPU to the debugger-internal static translation table.

- If called without parameters, the complete page table will be loaded. The list of static address translations can be viewed with **TRANSlation.List**.
- If the command is called with either an address range or an explicit address, page table entries will only be loaded if their **logical** address matches with the given parameter.

Use this command to make the translation information available for the debugger even when the program execution is running and the debugger has no access to the page tables and TLBs. This is required for the real-time memory access. Use the command **TRANSlation.ON** to enable the debugger-internal MMU table.

PageTable	Loads the entries of an MMU translation table and copies the address translation into the debugger-internal static translation table. • if *<range>* or *<address>* have a space ID: loads the translation table of the specified process • else, this command loads the table the CPU currently uses for MMU translation.

KernelPageTable	Loads the MMU translation table of the kernel. If specified with the **MMU.FORMAT** command, this command reads the table of the kernel and copies its address translation into the debugger-internal static translation table.
TaskPageTable *<task_magic>* | *<task_id>* | *<task_name>* | *<space_id>***:0x0**	Loads the MMU address translation of the given process. Specify one of the **TaskPageTable** arguments to choose the process you want. In MMU-based operating systems, each process uses its own MMU translation table. This command reads the table of the specified process, and copies its address translation into the debugger-internal static translation table. • For information about the first three parameters, see **"What to know about the Task Parameters"** (general_ref_t.pdf). • See also the appropriate **OS Awareness Manual**.
ALL [**Clear**]	Loads all known MMU address translations. This command reads the OS kernel MMU table and the MMU tables of all processes and copies the complete address translation into the debugger-internal static translation table. See also the appropriate **OS Awareness Manual**. **Clear:** This option allows to clear the static translations list before reading it from all page translation tables.

CPU specific Table in MMU.SCAN <table>

ITLB	Loads the current contents of the Instruction Translation Lookaside Buffer.
DTLB	Loads the current contents of the Data Translation Lookaside Buffer.

Beyond Specific TrOnchip Commands

The **TrOnchip** command provides low-level access to the on-chip debug register.

TrOnchip.RESet — Reset on-chip trigger settings

```
Format:          TrOnchip.RESet
```

Reset all TrOnchip settings to their default values.

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

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

Default: OFF.

If StepVector is activated the debugger handles the entry into the exception handler. Therefore the debugger manipulates the exception registers (SPR: EPCR, ESR, EEAR) as well as the supervision register (SR). Finally the exception entry is forced by modifying the program counter (PC) to the respective exception vector.

This is useful for designs which deactivate the debug features while exceptions and allows to single step these handlers. Please select the relevant exception with the **TrOnchip.Set** feature.

TrOnchip.state — Display on-chip trigger window

```
Format:          TrOnchip.state
```

Opens the **TrOnchip.state** window.

Format:	**TrOnchip.Set** *<event>* [**ON** | **OFF**]
<event>:	**RE** **BUSEE** **DPFE** **IPFE** **TTE** **AE** **IIE** **INTE** **DME** **IME** **RE** **SCE** **FPE**

Default: RSTE ON and IIE ON.

The TrOnchip.Set feature allows to trigger on specific exception events. The activation of a trigger causes the CPU to stop when the trigger event occurs (for example illegal instruction exception (IIE)). If activated the trigger will act similar to a breakpoint. In conjunction with the **TrOnchip.StepVector** feature it is possible to force single-stepping into exceptions.

RSTE	Stop on reset exception.
BUSEE	Stop on bus error exception. e.g. caused attempt to access an invalid physical address
DPFE	Stop on data page fault exception. e.g. caused by page protection violation for load/store
IPFE	Stop on instruction page fault exception. e.g. caused by page protection violation for instruction fetch
TTE	Stop on tick timer exception.
AE	Stop on alignment exception. e.g. load/store to wrong aligned address
IIE	Stop on illegal instruction exception. e.g. instruction could not be decoded
INTE	Stop on external interrupt exception.
DME	Stop on D-TLB miss exception. e.g. no valid TLB entry found for access

IME	Stop on I-TLB miss exception. e.g. no valid TLB entry found for access
RE	Stop on range exception.
SCE	Stop on system call exception. e.g. caused by invocation of `b.sys` instruction
FPE	Stop on floating point exception.

Beyond Specific TERM Commands

The **TERM** command provides various methods for ASCII character based communication with the target, see **TERM**.

TERM.METHOD.BufferQUICK — Intrusive buffer based virtual terminal

Format:	**TERM.METHOD BufferQUICK** <*trap_address*> <*buffer_out*> <*buffer_in*> <*cap_out*> <*cap_in*>

The BufferQUICK terminal is similar to the **TERM.METHOD BufferS** but shifts the functionality from the GUI to the debug box. It allows much lower latencies compared to BufferS. A demo how to use the BufferQUICK terminal is included in ~~/demo/beyond/etc/virtual_terminal

<*trap_address*>	Address of the b.trap instruction the CPU stops at in case communication is requested.
<*buffer_out*>	Address of the Data-Buffer used for TARGET to GUI communication.
<*buffer_in*>	Address of the Data-Buffer used for GUI to TARGET communication.
<*cap_out*>	Capacity of the <*buffer_out*> buffer. Default: 256 bytes
<*cap_in*>	Capacity of the <*buffer_in*> buffer. Default: 256 bytes

Syntax examples:

```
; The following examples are highly code dependent.
; In this example we use
; bufferout   = 0x1000 , length 0x100 bytes
; bufferin    = 0x1200 , length 0x100 bytes
; trapaddress = 0x4002

; Printf & Scanf or Out/In Terminal
TERM.RESet
TERM.Method BufferQUICK 0x4002 0x1000 0x1200

; Printf or Out Terminal only
TERM.RESet
TERM.Method BufferQUICK 0x4002 0x1000

; Printf or Out Terminal only, bufferout is 0x20 bytes long
TERM.RESet
TERM.Method BufferQUICK 0x4002 0x1000, , 0x20
```

JTAG Connection

Pinout 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	7	8	GND
TCK	9	10	GND
N/C	11	12	GND
TDO	13	14	GND
RESET-	15	16	GND
DBGRQ	17	18	GND
N/C	19	20	GND

For details on logical functionality, physical connector, alternative connectors, electrical characteristics, timing behavior and printing circuit design hints refer to the application note **“Arm Debug and Trace Interface Specification”** (app_arm_target_interface.pdf). It describes a debug cable which is technically the same as the Beyond debug cable.

Trace Connection

Pinout of the Mictor Trace connector:

Signal	Pin	Pin	Signal
N/C	1	2	N/C
N/C	3	4	N/C
N/C	5	6	CLK (TRACE)
N/C	7	8	N/C
SRST-	9	10	N/C
TDO	11	12	VREF-TRACE
N/C	13	14	VREF-DEBUG
TCK	15	16	DATA0
TMS	17	18	DATA1
TDI	19	20	DATA2
TRST-	21	22	DATA3
N/C	23	24	DATA4
N/C	25	26	DATA5
N/C	27	28	DATA6
N/C	29	30	DATA7-CHUNK
N/C	31	32	N/C
N/C	33	34	N/C
N/C	35	36	N/C
N/C	37	38	VALID