LAUTERBACH
DEVELOPMENT TOOLS

General Commands Reference Guide J

MANUAL

General Commands Reference Guide J

General Commands Reference Guide J

Version 06-Jun-2024

History

Dec-2021	New feature JTAG.SEQuence.MemAccess.ADD to access memory via JTAG sequences.

Java

Java debugging subsystem

Java code that is compiled into a native program with debug symbols, e.g. an ELF file with DWARF2 records, can be debugged with TRACE32 just like any other HLL (C/C++) code. You don't need to use the **Java** command group to debug such an application.

TRACE32 has built-in debugging support for several Java Virtual Machine byte code interpreters. Just-In-Time compilation is not supported and must be disabled in the Java VM. These Java Virtual Machines are supported:

- **J2ME CDC/RI** (SUN/Oracle Java 2 Micro Edition, Connected Device Configuration / Reference Implementation)
- **J2ME CLDC/RI "KVM"** (SUN/Oracle Java 2 Micro Edition, Connected Limited Device Configuration / Reference Implementation, the "K Virtual Machine", K as in Kilobyte)
- **KAFFE** (a Java-compatible open-source VM that was developed independently from SUN)

The **Java** command group enables you to set the necessary TRACE32 configuration parameters to debug with your Java VM implementation.

TRACE32 has also built-in support for Android Dalvik Virtual Machine's "Dex" file pre-loading and for parsing the symbols contained in the symbol database. Pre-loading and parsing of Dex files is also set up with the **Java** command group. Some part of the VM Awareness Support functionality for Dalvik additionally requires loading and using a dalvik.t32 EXTension module. Please see **"VM Debugger Dalvik"** (vmdalvik.pdf).

NOTE:	If you need support for a different type of virtual machine, or for a proprietary or "optimizing" implementation of a Java virtual machine, please refer to the **EXTension** command group for creating a custom VM Debugging Awareness.

See also

- Java.CONFIG
- Java.LOAD
- Java.LOADCLASS
- Java.MAP
- Java.OFF
- Java.ON
- Java.state

Format:	**Java.CONFIG** *<vmtype>* [*<params>*] **Java.VMTYPE** (deprecated)
<type> and *<params>*:	**J2ME_CDC** *<name_table_address> <method_type_table_address> <field_type_table_address> <class_table_address> <class_table_count_address> <globals_address>* **J2ME_CLDC** *<name_table_address> <class_table_address>* **KAFFE** *<class_table_address> <class_table_size>* **PEARL90** **DALVIK**

Selects the Virtual Machine type for debugging. These are the currently available types:

J2ME_CDC	SUN/Oracle Java 2 Micro Edition, Connected Device Configuration / Reference Implementation
J2ME_CLDC	SUN/Oracle Java 2 Micro Edition, Connected Limited Device Configuration / Reference Implementation, the “K Virtual Machine”, K as in Kilobyte (= Kilobyte Virtual Machine, KVM)
KAFFE	A Java-compatible open-source VM that was developed independently from SUN
PEARL90	PEARL90 interpreter support (non-Java VM, legacy entry)
DALVIK	Android Dalvik VM

Example:

```
; tell the debugger to use KVM specific definitions for VM handling
Java.CONFIG J2ME_CLDC UTFStringTable ClassTable
```

See also

- Java
- Java.state

Format 1:	**Java.LOAD** [**/DIAG**]
Format 2:	**Java.LOAD** *<file>* [**/DIAG**]

Format 1 is used for SUN Java (J2ME_CDC, J2ME_CLDC, KAFFE): Shorthand for **Java.LOADCLASS** for ALL classes found. The standard mapping from class descriptor to source path will be used.

Format 2 is used for Dalvik: Pre-load *.dex/*.odex/*.apk/*.jar file into an internal cache for **Java.LOADCLASS**.

See also

- Java
- Java.state

Java.LOADCLASS Load Java class information

Format 1:	**Java.LOADCLASS** *<address>* [**/DIAG**]
Format 2:	**Java.LOADCLASS** *<dexfile_address>* *<class_descriptor>* [**/DIAG**]

Format 1 is used for SUN Java (J2ME_CDC, J2ME_CLDC, KAFFE): Loads symbolic information for a class residing in memory at *<address>*.

Format 2 is used for Dalvik: Loads the symbols for the named Dalvik/Java class from the cache (filled with **Java.LOAD**).

<dexfile_address>	Is the address of the DeX file in target memory.
<class_descriptor>	Is a Java class descriptor. (Both `Lpackage/class;` and `package.class` are accepted.)

See also

- Java
- Java.state

Format: **Java.MAP**

Displays the mapping of VMSTOP breakpoint, IP, FP, SP, LP, CP for current Java VM configuration.

See also

- Java.MAP.ByteCode
- Java.MAP.CB
- Java.MAP.CP
- Java.MAP.FP
- Java.MAP.IP
- Java.MAP.IPBASE
- Java.MAP.List
- Java.MAP.LOADATTR
- Java.MAP.LP
- Java.MAP.MB
- Java.MAP.NoByteCode
- Java.MAP.NoVM
- Java.MAP.NoVMStop
- Java.MAP.RESet
- Java.MAP.SP
- Java.MAP.VM
- Java.MAP.VMStop
- Java
- Java.state

Java.MAP.ByteCode Define byte code area

Format: **Java.MAP.ByteCode** [*<range>*]

Debugger configuration: Mark specified memory area as containing byte code.

<range> Range in memory where byte code resides.

Example:

```
Java.MAP.ByteCode 8000--87FF    ; mark data area from 8000 to 87FF
                                ; as byte code section
```

See also

- Java.MAP
- Java.MAP.NoByteCode

Format:
Java.MAP.CB NONE
Java.MAP.CB Frame *<range>*
Java.MAP.CB Register *<register_name>*
Java.MAP.CB Static *<name>*
Java.MAP.CB Var *<name>*

Configures the Java VM class block pointer.

See also

- Java.MAP

Java.MAP.CP Configure Java VM class pointer

Format:
Java.MAP.CP NONE
Java.MAP.CP Frame *<range>*
Java.MAP.CP Register *<register_name>*
Java.MAP.CP Static *<name>*
Java.MAP.CP Var *<name>*

Configures the Java VM class pointer.

See also

- Java.MAP

Java.MAP.FP Configure Java VM frame pointer

Format:
Java.MAP.FP NONE
Java.MAP.FP Frame *<range>*
Java.MAP.FP Register *<register_name>*
Java.MAP.FP Static *<name>*
Java.MAP.FP Var *<name>*

Configures the Java VM frame pointer.

See also

- Java.MAP

Format:	**Java.MAP.IP NONE** **Java.MAP.IP Frame** *<range>* **Java.MAP.IP Register** *<register_name>* **Java.MAP.IP Static** *<name>* **Java.MAP.IP Var** *<name>*

Configures the Java VM instruction pointer.

NONE	Delete last assignment.
Frame	Use instruction pointer on stack.
Register	Use named register as VM instruction pointer.
Static	Use named static (global) variable as VM instruction pointer.
Var	Define local variable with address range of validity.

Pointers that must be set for Java VM are:

IP	Java Instruction Pointer (to current byte code)
LP	Java Locals Pointer
SP	Java Stack Pointer
FP	Java Frame Pointer
CP	Java Class Pointer

Example:

```
;set Java VM instruction pointer
Java.MAP.IP Static ip_global

;for routine 'FastInterpret', we need a local variable 'override'
Java.MAP.IP Var IP sYmbol.RANGE(FastInterpret)
```

See also

- Java.MAP

Format: **Java.MAP.IPBASE NONE**
Java.MAP.IPBASE Frame *<range>*
Java.MAP.IPBASE Register *<register_name>*
Java.MAP.IPBASE Static *<name>*
Java.MAP.IPBASE Var *<name>*

Configures the Java VM IPBASE pointer.

See also

- Java.MAP

Java.MAP.List List Java VM specific mappings

Format: **Java.MAP.List**

Displays the mapping of VMSTOP breakpoint, IP, FP, SP, LP, CP for current Java VM configuration.

Example:

```
Java.MAP.List                   ; list VM mappings
```

See also

- Java.MAP

Java.MAP.LOADATTR Load attribute information from Java class files

Format: **Java.MAP.LOADATTR**

Loads the attribute information from Java class files residing in memory.

Example:

```
Java.MAP.LOADATTR
```

See also

- Java.MAP

Java.MAP.LP Configure Java VM LP pointer

Format:
Java.MAP.LP NONE
Java.MAP.LP Frame *<range>*
Java.MAP.LP Register *<register_name>*
Java.MAP.LP Static *<name>*
Java.MAP.LP Var *<name>*

Configures the Java VM LP pointer.

See also

- Java.MAP

Java.MAP.MB Configure Java VM method block pointer

Format:
Java.MAP.MB NONE
Java.MAP.MB Frame *<range>*
Java.MAP.MB Register *<register_name>*
Java.MAP.MB Static *<name>*
Java.MAP.MB Var *<name>*

Configures the Java VM method block pointer.

See also

- Java.MAP

Format: **Java.MAP.NoByteCode** [*<range>*]

Removes the byte code mapping for address range.

Example:

```
Java.MAP.NoByteCode 0x8000--0x80ff       ; remove byte code mapping
```

See also

- Java.MAP
- Java.MAP.ByteCode

Java.MAP.NoVM Remove VM interpreter flag

Format: **Java.MAP.NoVM** [*<range>*]

Removes a VM interpreter flag for address range.

Example:

```
Java.MAP.NoVM 0x1000--0x1080      ; unmap address range as part of VM
```

See also

- Java.MAP
- Java.MAP.VM

Format: **Java.MAP.NoVMStop** *<address>*

Removes a breakpoint in VM interpreter (for single stepping).

Example:

```
; remove debug breakpoint for Java VM (see Java.MAP.VMStop)
Java.MAP.NoVMStop  FastInterpret\next0a
```

See also

- Java.MAP
- Java.MAP.VMStop

Java.MAP.RESet Reset Java VM mappings

Format: **Java.MAP.RESet**

Resets the Java VM mappings.

Example:

```
Java.MAP.RESet                  ; reset all debugger mappings for Java VM
```

See also

- Java.MAP

Format: **Java.MAP.SP NONE**
Java.MAP.SP Frame *<range>*
Java.MAP.SP Register *<register_name>*
Java.MAP.SP Static *<name>*
Java.MAP.SP Var *<name>*

Configures the Java VM stack pointer.

See also

- Java.MAP

Format: **Java.MAP.VM** [*<range>*]

Configures the Java VM interpreter routine area.

Example:

```
; map range of VM interpreter to range
; of interpreter function

Java.MAP.VM sYmbol.RANGE(FastInterpret)
```

See also

- Java.MAP
- Java.MAP.NoVM

Java.MAP.VMStop Configure breakpoint in VM interpreter

Format: **Java.MAP.VMStop** *<address>*

Configures a breakpoint within VM interpreter for single-step.

Example:

```
; set debugger breakpoint for Java VM
; single stepping to label 'next0a' within
; routine 'FastInterpret'

Java.MAP.VMStop FastInterpret\next0a
```

See also

- Java.MAP
- Java.MAP.NoVMStop

Format: **Java.OFF**

Switches off the Java VM debugging support.

Example:

```
Java.OFF                          ; end Java debugging
```

See also

- Java
- Java.state

Java.ON Activate Java debugging subsystem

Format: **Java.ON**

Activates the (Java) VM debugging support. E.g. “step” now does a single step within byte code, **Data.List** window shows Java source mapping to byte code, activate byte code disassembler, etc.

Example:

```
Java.ON                           ; start VM debugging
```

See also

- Java
- Java.state

Format: **Java.state**

Displays the current state of Java VM debug subsystem - VM type, IP, FP, LP, CP, MB, CB and SP.

Example:

```
Java.state                          ; display VM debug subsystem state
```

See also

- Java
- Java.CONFIG
- Java.LOAD
- Java.LOADCLASS
- Java.MAP
- Java.OFF
- Java.ON

JTAG — Low-level JTAG control

General notes:

- The JTAG commands are only available for JTAG debuggers. They are not supported for architectures with other debug interfaces (e.g. BDM).
- Manual JTAG control commands can conflict with the normal debugger operation. A certain course of action is needed, if you want to use the debugger at the same time.
- Examples can be found in the following TRACE32 demo folders:

  ~~/demo/etc/jtag/

  ~~/demo/arc/etc/

  ~~/demo/powerpc/etc/jtag/

Example: This script shows how to retrieve the IDCODE of a single JTAG TAP controller.

```
SYStem.JtagClock 1Mhz      ; configure JTAG clock
JTAG.USECLOCK ON           ; use the JTAG clock instead of pin toggling
                           ; for DEBUG CABLE based solutions
JTAG.PARKSTATE RTI         ; set the hand-over TAP state is RunTestIdle
JTAG.LOCK                  ; prevent debugger from interrupting the
                           ; sequence
JTAG.PIN ENable            ; enable output buffers of DEBUG CABLE
JTAG.SHIFTTMS 1 1 1 1 1 0  ; reset TAP controller and
                           ; go to RUN-TEST-IDLEstate
JTAG.SHIFTTMS 1 0 0        ; go to SHIFT-DRstate
JTAG.SHIFTREG %Long 0x0    ; shift 32-bit data and go to EXIT-1-DRstate
&IDCODE=JTAG.SHIFT()       ; retrieve the shifted data of previous
                           ; SHIFTREG command
JTAG.SHIFTTMS 1 0          ; go to RUN-TEST-IDLEstate
JTAG.UNLOCK                ; allow debugger to use JTAG

IF (&IDCODE&0x1)==0x1      ; check if IDCODE is valid
  PRINT "IDCODE is &IDCODE"
ELSE
  PRINT "Device has no IDCODE"
```

See also

- JTAG.CJTAG
- JTAG.CLIENTINDEX
- JTAG.LOADBIT
- JTAG.LOCK
- JTAG.MIPI34
- JTAG.PARKSTATE
- JTAG.PIN
- JTAG.PROGRAM
- JTAG.RESet
- JTAG.SEQuence
- JTAG.SHIFTREG
- JTAG.SHIFTTDI
- JTAG.SHIFTTMS
- JTAG.UNLOCK
- JTAG.USECLOCK
- JTAG.X7EFUSE
- JTAG.XUSEFUSE

▲ 'CPU specific JTAG.CONFIG Commands' in 'ARC Debugger and Trace'
▲ 'CPU specific JTAG.CONFIG Commands' in 'Intel® x86/x64 Debugger'
▲ 'JTAG Functions' in 'General Function Reference'

Format: **JTAG.CLIENTINDEX** *<value>*

Default: 0.

The **JTAG** command group uses the same underlaying data and algorithms as the Remote API Direct Access functions. Historically, the first Remote API client uses the same data set as the **JTAG** command group. To allow different settings between JTAG and RemoteAPI, use an index other than 0.

<value> Index of the data set. The value must be equal to or greater than 0 and smaller than 256.

Example:

```
;select an index other than any RemoteAPI client can have
JTAG.CLIENTINDEX 16.

;lock the JTAG access against RemoteAPI accesses, too
JTAG.LOCK
```

See also

- JTAG
- ▲ 'Introduction' in 'API for Remote Control and JTAG Access in C'

Format:	**JTAG.LOADBIT** *<file>* (deprecated) **Use JTAG.PROGRAM.Xilinx instead.**

NOTE:	This command is deprecated; prefer **JTAG.PROGRAM.Xilinx**, which operates independently from multicore settings and available on all architectures which support JTAG.

The command **JTAG.LOADBIT** configures a Xilinx FPGA with a bitstream (a .BIT file).

Be sure to make the **correct multicore settings** before invoking the command. The settings are identical to those used for debugging a core.

Before invoking the command the debugger must be in state **SYStem.Mode Down**.

```
SYStem.CPU Virtex5PPC        ; Example for PPC440 in Virtex5
SYStem.Down
JTAG.LOADBIT system.bit
```

PowerPC:	Besides making the correct multicore settings, it is necessary to select the correct CPU **before** configuring the FPGA e.g. SYStem.CPU VIRTEX5PPC.

NOTE:	Configuration using **compressed bitstreams** is supported.

See also

- JTAG

JTAG.LOCK Grab the JTAG port for manual control

Format:	**JTAG.LOCK**

Disables all debugger activity on the JTAG port when the first manual access (**JTAG.PIN**, **JTAG.SHIFTREG**, **JTAG.SHIFTTDI** or, **JTAG.SHIFTTMS**) is performed. This allows to enter manual JTAG control sequences without interfering JTAG accesses by the debugger. For handing back control to the debugger, use **JTAG.UNLOCK** after finishing your manual sequence.

The following steps need to be respected:

1. Lock the JTAG port for the debugger. Now the TAP controller is in the pause parking position. Consult the **Processor Architecture Manual** to find out what is the pause parking position for your core. Normally this is either 12. (Run-Test/Idle) or 7. (Select-DR Scan).
2. Perform your JTAG access.
3. Return to the pause parking position.
4. Unlock the JTAG port for the debugger. The debugger will resume operation immediately.

As long as the JTAG port is locked, the debugger will not access the target.

In case an active debug session is interrupted, never issue a reset or alter the on-chip debug resources unless you know exactly what you do. This may confuse the debugger or leads to a corrupted debug session.

See also

- JTAG
- JTAG.PIN
- JTAG.SHIFTREG
- JTAG.SHIFTTDI
- JTAG.SHIFTTMS
- JTAG.UNLOCK

▲ 'Custom JTAG Access' in 'Application Note JTAG Interface'

Format:	**JTAG.MIPI34** *<pin> <state>*
<pin>:	**PIN12** **PIN14** **PIN16** **PIN18** **PIN20**
<state>:	**NormalOperation** **ForceTristate** **ForceLow** **ForceHigh**

The command **JTAG.MIPI34** makes it possible to override the normal functionality of some pins on the MIPI34 connector.

The command is only available for the MIPI34 CombiProbe/µTrace (MicroTrace) whisker.

NormalOperation	Return to normal operation. This is the default state.
ForceTristate	Never drive the pin. Use the PRACTICE function **JTAG.MIPI34()** to query the current pin level.
ForceLow	Drive a logic ‘0’ to the pin, even if the debug port is in tristate mode.
ForceHigh	Drive a logic ‘1’ to the pin, even if the debug port is in tristate mode.

See also

- JTAG

Format: **JTAG.PARKSTATE** [**RunTestIdle** | **SELectDRscan**]

Default: The default depends on the debug driver of the certain architecture. In case the JTAG commands must cooperate with the debug driver, the park state should be set according to the rest of the JTAG commands.

The park state is the TAP state that is assumed as hand over state between debug driver and JTAG commands when **JTAG.LOCK** and **JTAG.UNLOCK** are used. Before **JTAG.UNLOCK** is executed, the BYPASS instruction needs to be shifted because consecutive TAP state changes use the DR to reach a certain TAP state.

Example:

```
SYStem.JtagClock 1Mhz      ; configure JTAG clock
JTAG.USECLOCK ON           ; use the JTAG clock instead of pin
                           ; toggling for DEBUG CABLE based solutions
JTAG.PARKSTATE RTI         ; the script uses RUN-TEST-IDLE as handover
                           ; state
JTAG.LOCK                  ; prevent debugger from interrupting the
                           ; sequence
JTAG.PIN ENable            ; enable output buffers of DEBUG CABLE
JTAG.SHIFTTMS 1 1 1 1 1 0 ; reset TAP controller and
                           ; go to RUN-TEST-IDLE state
JTAG.SHIFTTMS 1 1 0 0      ; go to SHIFT-IR state
JTAG.SHIFTREG 1 1 1 1 1 1 ; shift BYPASS instruction and go to EXIT-1-IR
JTAG.SHIFTTMS 0 1          ; go to RUN-TEST-IDLE
JTAG.UNLOCK                ; the debugger will now drive from the park
                           ; state to the multi-core tap state
SYStem.CONFIG.SLAVE ON     ; prevent SYStem.Mode command from resetting
                           ; JTAG again
SYStem.Mode.Attach         ; attach to the CPU
```

See also

- JTAG

Format:	**JTAG.PIN** *<signal_name>* [**0** | **1** | **Low** | **High**]
<signal_name>:	**TCK** **TMS** **TDI** **TDO** **NTRST** **NRESET** **VTREF** **RESETLATCH** **VTREFLATCH** **ENable** **DISable**

Sets the level of output signals on JTAG connector to a certain level. It has no effect for input signals. If not already active due to the normal debugger operation, you need to enable the output driver before using the JTAG port. This is done using the pseudo-signal ENable.

For details on the various signals see the table below. Note that the available signals depend on the architecture's JTAG port.

NOTE: There is also a *function* of the same name, **JTAG.PIN()** that can be used to read the level of signals. The result is undefined for output signals.

TCK	output
TMS	output
TDI	output
TDO	input
NTRST	output; JTAG reset; low active
NRESET	input/output; chip reset; low active
VTREF	input; reference voltage; high if applied
RESETLATCH	input; high, if NRESET became low after reading this bit the last time i.e. when there was a pulse on NRESET line.
VTREFLATCH	input; high, if VTREF became low after reading this bit the last time; i.e. when there was a pulse on VTREF line. This may condition may indicate a problem with the target's power supply.

ENable input/output; enables the output driver for all output signals; no further parameter required

DISable input/output; disables the output driver for all output signals; no further parameter required

See also

- JTAG
- JTAG.LOCK
- JTAG.SHIFTREG
- JTAG.SHIFTTDI
- JTAG.SHIFTTMS
- JTAG.UNLOCK
- JTAG.PIN()

Using the commands from the JTAG.PROGRAM group, you can execute JTAG command sequences from files generated by 3rd-party tools. Applications of this include programming FPGA devices that are in the same JTAG chain as a microprocessor or loading softcore designs for debugging.

Before invoking any command from the JTAG.PROGRAM group, the debugger must be in state **SYStem.Mode Down**. Any JTAG commands are executed using the clock frequency specified with **SYStem.JtagClock**.

Use **JTAG.PROGRAM.auto** to automatically infer the file format from the file's extension, or one of the other commands in this group to explicitly specify the file type.

See also

- JTAG.PROGRAM.Altera
- JTAG.PROGRAM.auto
- JTAG.PROGRAM.JAM
- JTAG.PROGRAM.JBC
- JTAG.PROGRAM.SVF
- JTAG.PROGRAM.Xilinx
- JTAG

Format:	**JTAG.PROGRAM.Altera** *<file>* [*/<option>*]
<option>:	**IRPRE** *<value>* **IRPOST** *<value>* **DRPRE** *<value>* **DRPOST** *<value>*

This command programs bitstreams in .rbf format into Altera FPGAs.

NOTE:	To load a bitstream via JTAG, bitstream compression must be disabled in the Quartus II toolchain. For SoC devices (e.g. Cyclone V devices), you must specify the options such that the TAP of the FPGA is accessed, not the one used for the ARM-based HPS.

IRPRE *<value>* **IRPOST** *<value>* **DRPRE** *<value>* **DRPOST** *<value>*	Configures the JTAG chain parameters. These options work like the corresponding commands in **SYStem.Option**: They specify how the FPGA device to be programmed can be reached in the JTAG chain. All values default to 0 if the corresponding option is not specified.

See also

- JTAG.PROGRAM
- JTAG.PROGRAM.auto

Format:	**JTAG.PROGRAM.auto** *<file>.<extension>* [*/<option>*]

Executes a programming file in one of several formats. The format to be used is depends on the file name extension. The supported file extensions are as follows:

File <extension>	Command Used
.rbf	**JTAG.PROGRAM.Altera**
.jam	**JTAG.PROGRAM.JAM**
.jbc	**JTAG.PROGRAM.JBC**
.svf	**JTAG.PROGRAM.SVF**
.bit	**JTAG.PROGRAM.Xilinx**

<options>	See the documentation of the individual subcommands for a description of their options.

See also

- JTAG.PROGRAM
- JTAG.PROGRAM.Altera
- JTAG.PROGRAM.JAM
- JTAG.PROGRAM.JBC
- JTAG.PROGRAM.SVF
- JTAG.PROGRAM.Xilinx

Format:	**JTAG.PROGRAM.JAM** *<file>* **/A "***<action>***"** **JTAG.LOADJAM** (deprecated)

Loads a JAM or STAPL file and executes the *<action>* defined in the file.

The command is intended to configure FPGAs with the debugger. JAM files generated by Alteras Quartus II software usually contain the action CONFIGURE. Executing this action configures the Altera FPGA with the design, which is contained in the JAM file.

IRPRE *<value>* **IRPOST** *<value>* **DRPRE** *<value>* **DRPOST** *<value>*	Configures the JTAG chain parameters. These options work like the corresponding commands in **SYStem.Option**: They specify how the device targeted by the programming file can be reached. All values default to 0 if the corresponding option is not specified.
A "*<action>***"**	Action from the programming file to execute. The names of available actions are defined by the application which generated the file.

NOTE:	JAM/STAPL is a programming language standardized by JEDEC, which allows to control a JTAG port. The terms JAM and STAPL can be used interchangeably and refer to the same programming language and file format. The terms were introduced by Altera (JAM) respectively Xilinx (STAPL).

Example 1: STAPL files generated by Xilinxs iMPACT software usually contain the action *RUN_XILINX_PROC*, which usually configures Xilinx FPGAs.

```
; Execute the action "RUN_XILINX_PROC" in the STAPL file.
; Mind the quotation marks around the <action> parameter!

JTAG.LOADJAM file.stapl /A "RUN_XILINX_PROC"
```

Example 2: Another action which is also often available is *READ_USERCODE*. By executing this action, the user code of the FPGA will be read out.

```
; Execute the action "READ_USERCODE" in the JAM file.

JTAG.PROGRAM.JAM file.jam /A "READ_USERCODE"
```

See also

- JTAG.PROGRAM
- JTAG.PROGRAM.auto

Format: **JTAG.PROGRAM.JBC** *<file>* **/A "***<action>***"**
JTAG.LOADJBC (deprecated)

JBC files are binary encoded JAM files. The command will load a JBC (Jam Byte Code) file and execute the *<action>*, which has to be defined in the JAM file.

The command behaves identically to **JTAG.PROGRAM.JAM**, except for the expected file format.

See also

- JTAG.PROGRAM
- JTAG.PROGRAM.auto

JTAG.PROGRAM.SVF Run programming file in SVF format

Format: **JTAG.PROGRAM.SVF** *<file>* [**/***<option>*]
JTAG.LOADSVF (deprecated)

<option>:
IRPRE *<value>*
IRPOST *<value>*
DRPRE *<value>*
DRPOST *<value>*
InitState *<state>*
IgnoreTDO
Verbose

SVF (Serial Vector Format) is a simple programming language to access JTAG hardware. The command will load a SVF file and execute it, so this command will perform the specified actions on the JTAG port.

This command is intended to configure FPGAs with the debugger. Most FPGA software tools (like Altera's Quartus II or Xilinx's iMPACT) can generate SVF files for this purpose.

IRPRE *<value>* **IRPOST** *<value>* **DRPRE** *<value>* **DRPOST** *<value>*	Configures the JTAG chain parameters. These options work like the corresponding commands in **SYStem.Option**: They specify how the FPGA device to be programmed can be reached in the JTAG chain. All values default to 0 if the corresponding option is not specified. **NOTE**: The SVF file can override these options using the HIR, TIR, HDR and TDR commands.

InitState *<state>*	Selects the actions performed at the beginning and end of the sequence defined by the SVF file. Possible values for *<state>*: • **None**: Default behavior. Navigate to the test logic reset state at the beginning and end of the sequence. • **TestLogicReset**: Assume that the state machine is in the test logic reset state at the beginning of the file and do not change the state after the file has been executed. • **RunTestIdle**: Assume that the state machine is in the run test idle state at the beginning of the file and do not change the state after the file has been executed.
IgnoreTDO	Does not perform the TDO checks specified in the SVF file. For some files, this can improve programming speed.
Verbose	Prints additional progress information to the **AREA.view** window.

Example:

```
JTAG.PROGRAM.SVF <file>                ; this command will load and
                                       ; execute a SVF file.
```

NOTE:	Loading an SVF file is independent of multicore settings made in the debugger, because SVF files are board-specific and thus implicitly reflect the layout of the JTAG scan chain.

See also

- JTAG.PROGRAM
- JTAG.PROGRAM.auto

Format:	**JTAG.PROGRAM.Xilinx** *<file>* [**/***<option>*]
<option>:	**IRPRE** *<value>* **IRPOST** *<value>* **DRPRE** *<value>* **DRPOST** *<value>* **IRWIDTH** *<value>*

This command programs bit streams in .bit format into Xilinx FPGAs.

NOTE: Compressed bitstreams can also be loaded using this command.

For SoC devices (e.g. Zynq-7000 devices), you must specify the options such that the TAP of the Programmable Logic (PL) is accessed, not the one used for the ARM-based PS.

Zynq UltraScale+ devices can alternatively be programmed via the PL using the PRACTICE script **~~/demo/arm/hardware/zynq_ultrascale/scripts/zynq-ultrascale_load_bitstream.cmm**.

IRPRE *<value>* **IRPOST** *<value>* **DRPRE** *<value>* **DRPOST** *<value>*	Configures the JTAG chain parameters. These options work like the corresponding commands in **SYStem.Option**: They specify how the FPGA device to be programmed can be reached in the JTAG chain. All values default to 0 if the corresponding option is not specified.
IRWIDTH *<value>*	Configures the number of bits of the JTAG instruction register. This varies from device to device, but is 6. in most cases.

See also

- JTAG.PROGRAM
- JTAG.PROGRAM.auto

Format: **JTAG.RESet**

Resets **JTAG.CONFIG** settings to default and deletes non-locked JTAG sequences created with the **JTAG.SEQuence** command group.

See also

■ JTAG

▲ 'CPU specific JTAG.CONFIG Commands' in 'ARC Debugger and Trace'
▲ 'CPU specific JTAG.CONFIG Commands' in 'Intel® x86/x64 Debugger'

Some SoCs with a JTAG interface need special JTAG sequences before the core can be accessed. For example, the JTAG-TAP of some ARC cores has to be dynamically added to the JTAG daisy chain of an SoC.

For some CPUs, TRACE32 already provides pre-defined JTAG sequences. For others, you can create user-defined JTAG sequences.

Using the **JTAG.SEQuence** command group, you can create and manage these JTAG sequences. Upon creation with the command **JTAG.SEQuence.Create**, JTAG sequences can be executed as follows:

- By a PowerDebug hardware module if certain events occur.

  A JTAG sequence is linked to an event, e.g. with (a) **SYStem.CONFIG.MULTITAP.JtagSEQuence** or (b) **SYStem.Option.CorePowerDetection.JtagSEQuence**. The linked JTAG sequence is executed automatically when that event occurs.

- By the user at the TRACE32 command line or via PRACTICE scripts (*.cmm)

  Use the command **JTAG.SEQuence.Execute** at the TRACE32 command line or in a PRACTICE script to execute a JTAG sequence on request.

What is the difference between the command groups…?

JTAG.SEQuence	JTAG.SHIFT
• The defined JTAG sequence can be linked to events that occur when, for example, the commands **SYStem.Attach** or **SYStem.Up** are executed. • The defined JTAG sequence can be *executed upon request* with **JTAG.SEQuence.Execute** *<seq_name>*.	• A JTAG sequence defined by a **JTAG.SHIFT** command is *immediately executed* when that **JTAG.SHIFT** command is executed. That is, JTAG sequence creation and execution take place at the same time.

A good way to familiarize yourself with the **JTAG.SEQuence** command group is to start with the examples of **JTAG.SEQuence.ADD**.

See also

- JTAG.SEQuence.ADD
- JTAG.SEQuence.Append
- JTAG.SEQuence.Create
- JTAG.SEQuence.Delete
- JTAG.SEQuence.Execute
- JTAG.SEQuence.List
- JTAG.SEQuence.ReMove
- JTAG.SEQuence.Replace
- JTAG.SEQuence.View
- JTAG.SHIFTREG
- JTAG.SHIFTTDI
- JTAG.SHIFTTMS
- JTAG
- SYStem.CONFIG.MULTITAP.JtagSEQuence

▲ 'CPU specific SYStem Commands' in 'ARC Debugger and Trace'
▲ 'Arm specific SYStem Commands' in 'Arm Debugger'
▲ 'JTAG Functions' in 'General Function Reference'

Format: **JTAG.SEQuence.ADD** *<seq_name> <action>*

<action>: *<basic>* | *<data>* | *<flow>* | *<comfort>*

<basic>:
PrePostRelative *<irpre> <irpost> <drpre> <drpost>*
RawShift *<length> <tms> <tdi>* [*<tdo>*]
ShiftIrAndExit *<length> <tdi>* [*<tdo>*]
ShiftDrAndExit *<length> <tdi>* [*<tdo>*]
TRST HIGH | **LOW**
SELect JTAG
SELect DAP-JTAG-AP *<accessport> <jtagport>*

<data>
ASSIGN *<var>* **=** [**~**] *<src>* [*<operator>* [**~**] *<src>*]
ToByteBuffer *<pos> <acc_width> <src>* [**LittleEndian** | **BigEndian**]
FromByteBuffer *<pos> <acc_width> <dst>* [**LittleEndian** | **BigEndian**]

<flow>
CALL *<seq_name>* [*<param>* [*<param>*]]
Jump *<index>* [*<cmp_src>* [**&** *<mask>*] *<cmp_op> <cmp_val>*]
NOP
WAIT *<time>*
TIMEOUT *<time>*

<comfort>
IRWidth *<width>*
ReadWrite *<ir> <length> <write>* [*<read>*]
ReadWriteIR *<length> <write>* [*<read>*]
ReadWriteDR *< length> <write>* [*<read>*]
RESetIfMaster

<cmp_src>: *<var>* | *<env>*
<mask>: *<hex>* | *<binary>* | *<var>* | *<env>*
<cmp_val>: *<hex>* | *<binary>* | *<integer>* | *<var>* | *<env>*
<length>: *<integer>* | *<var>* | *<env>*
<tms> | *<tdi>*: *<hex>* | *<binary>* | *<integer>* | *<var>* | *<env>*
<tdo> | *<dst>*: *<var>*
<src>: *<hex>* | *<binary>* | *<integer>* | *<var>* | *<env>*
<pos>: *<hex>* | *<binary>* | *<integer>* | *<var>* | *<env>*
<param>: *<hex>* | *<binary>* | *<integer>* | *<var>* | *<env>*
<width>: *<integer>* | *<var>* | *<env>*
<ir> **:** *<hex>* | *<binary>* | *<integer>* | *<var>* | *<env>*
<write>: *<hex>* | *<binary>* | *<integer>* | *<var>* | *<env>*
<read>: *<var>*

<acc_width> **Byte** | **Word** | **Long** | **Quad** | **TByte** | **PByte** | **HByte** | **SByte**

<operator>:	**&** | **<<** | **>>** | **+** | **-** | ***** | **/** | **%** | **^** | **|**
<cmp_op>:	**==** | **!=** | **>** | **<** | **>=** | **<=**
<var>:	**Result0** | **Result1** | **Local0** | **Local1** | **Local2** | **Local3** | **Local4** | **Local5** | **Local6** | **Local7**
<env>:	**SLAVE** | **SYStemMode** | **PortTYPE** | **MCTapState** | **CORE** | **PhysicalCORE** | **ConfigCORE** | **ConfigCHIP**

Adds an action to an existing JTAG sequence. To create a new JTAG sequence, use **JTAG.SEQuence.Create**.

<accessport>	Number of the DAP access port to which a DAP-JTAG-AP is connected.
<binary>	**Parameter Type**: Binary value.
<hex>	**Parameter Type**: Hex value.
<index>	Absolute index of a command within the given JTAG sequence. Index counting starts at 0 (not at 1). See example. **Parameter Type**: Decimal value.
<integer>	**Parameter Type**: Decimal value.
<jtagport>	Number of a JTAG port on a DAP-JTAG-AP.
<seq_name>	Name of the JTAG sequence (without quotes). The name must start with a letter. The following characters can be letters, numbers, underscores, and minus signs, i.e. this regular expression: ^[a-zA-Z] [a-zA-Z0-9_-]* As with TRACE32 commands, you can access a sequence with its full name or with only the capital letters of its name. Regardless of whether you choose the full or the short name, the selection of a sequence is case insensitive.
<time>	**Parameter Type**: Time value.

PrePostRelative	The action **PrePostRelative** adds the offsets given by this action to the pre/post settings of the core or DAP. The resulting pre/post settings are then taken into account within **ShiftIrAndExit** and **ShiftDrAndExit**. You can view the pre/post settings of the core and/or DAP in the dialog window **SYStem.CONFIG.state /Jtag** and modify them during the execution of the JTAG sequence.
RawShift	The action **RawShift** sends out the given TMS/TDI pattern on the TMS/TDI pin. If you are in SWD mode but have not yet used action **SELect**, then no pattern will be send to the pins.
SELect JTAG	Action to switch (back) to the primary JTAG port. This is only useful if you've selected a **DAP-JTAG-AP** before.
SELect DAP-JTAG-AP	Action to switch to a DAP-JTAG-AP port which is selected by DAP-access-port of the JTAG-AP ("accessport") and the port on the JTAG-AP ("jtagport"). After this command all shift actions will be send out to the chosen DAP-JTAG-AP. **NOTE**: Selecting DAP-JTAG-AP is not supported with **JTAG.SEQuence.Execute** when the current **SYStem.Mode** is neither "Up" not "Prepare".
ShiftDrAndExit	The action **ShiftDrAndExit** should be used only if you know that you have entered the shift-dr state already. It will send the given TDI pattern while keeping TMS at 0. For the last TDI bit, TMS will be set to 1, so the JTAG state machine will be in exit-1-dr state after the action. **ShiftIrAndExit** considers the pre/post settings of the core or DAP which might have been modified by the action **PrePostRelative**.
ShiftIrAndExit	The action **ShiftIrAndExit** should be used only if you know that you have entered the shift-ir state already. It will send the given TDI pattern while keeping TMS at 0. For the last TDI bit, TMS will be set to 1, so the JTAG state machine will be in exit-1-ir state after the action. **ShiftIrAndExit** considers the pre/post settings of the core or DAP which might have been modified by action **PrePostRelative**.
TRST HIGH | LOW	Sets the TRST pin to either **HIGH** or **LOW**. Cannot be used when **DAP-JTAG-AP** was selected.

ASSIGN	Assigns a value to the variable selected with *<var>*.
FromByteBuffer (since build 142495)	Loads between 1 and 8 bytes from a global buffer of 512 byte. The buffer is initialized with zeros and is destroyed when no sequence is active any more. By default the byte order is litte-endian but there is also an option for big-endian. When a sequence is registered for memory access with **JTAG.SEQuence.MemAccess.ADD** the you have to load the data for a write-transaction from the global buffer (in little-endian order).
ToByteBuffer (since build 142495)	Save between 1 and 8 bytes to a global buffer of 512 byte. The buffer is initialized with zeros and is destroyed when no sequence is active any more. By default the byte order is litte-endian but there is also an option for big-endian. When a sequence is registered for memory access with **JTAG.SEQuence.MemAccess.ADD** the you have to save the result of a read-transaction to the global buffer (in little-endian order).

Actions <flow>

CALL	Calls another JTAG sequence. The called sequence gets a new set of the local variables **Local0**…**Local7**. The local variables are initialized with the values from the calling sequence. Additionally, you can pass up to two parameters to the called JTAG sequence, which will initialize the local variables **Local0** and **Local1**. This is illustrated in example 2. Please refer to the comment and code lines formatted in **blue**, **red**, and **purple**. The result of a sequence can be passed to the called via the global sequence variables **Result0** and **Result1**.
Jump	Action to jump to another action inside the same sequences. You can define a condition for the jump which has to be met or does not have to be met.
NOP	This action does nothing.
TIMEOUT	Sets a maximum time for the execution of the sequence. This command can be used only once per sequence. The default time-out is 100.ms. The maximum value for **TIMEOUT** is 10.s The time-out of a sequence called by another sequence with **CALL** is truncated to the time-out of the caller during execution. If you specify the time-out via a variable, it's value is considered as a time period in microseconds.

WAIT	Pauses the sequence for the given time period. The maximum waiting time is limited by **TIMEOUT** (see above). If you specify the time to wait via a variable, it's value is considered as a time period in microseconds.

Actions <comfort>>

The comfort actions are only suitable if the JTAG TAPs you dealing with have no side effects on any JTAG state. Especially on entering or staying in state Run-Test/Idle, some JTAG TAPs have a side-effect. In this case the comfort actions are probably not suitable for your TAP.
The functionality of the comfort actions is also possible with the basic actions, but coding is more convenient with them.

All comfort actions (except **IRWidth**) end in the multi-core TAP state configured with **SYStem.CONFIG.TAPState**, which must be either Select-DR-Scan or Run-Test/Idle.

IRWidth (since build 142495)	Sets the width of the IR register used with action **ReadWrite**. The setting is local for every sequence, which means, than a prarent sequence inherits its **IRWidth** to a called child sequence, but if the child sequence changes the value, the **IRWidth** of the parent is not changed. The default is zero, which means you must use **IRWidth** at least once, before using action **ReadWrite.**
ReadWrite (since build 142495)	Enters the Shift-IR state and shifts the value *<ir>* to the IR register. Then the Shift-DR state is entered and *<length>* bits are written/read from the just selected DR register. The active state of the TAP controller must be the multicore TAP state configured with **SYStem.CONFIG.TAPState**. You mus use the action **IRWidth** at least once before using **ReadWrite**.
ReadWriteIR (since build 142495)	Enters the Shift-IR state and writes/reads *<length>* bits to/from the IR register. The active state of the TAP controller must be the multicore TAP state configured with **SYStem.CONFIG.TAPState**.
ReadWriteDR (since build 142495)	Enters the Shift-DR state and writes/reads *<length>* bits to/from the active DR register. The active state of the TAP controller must be the multicore TAP state configured with **SYStem.CONFIG.TAPState**.
RESetIfMaster (since build 142495)	Resets the TAP controller, If **SYStem.CONFIG Slave** is set to OFF and there only one logical core or it is the first core of an SMP system. Use this action with care! Resetting the TAP in **SYStem.Mode** is **Up** might cause the debugger to loose the connection to your target. (Depends on the used chip/architecture) To reset the TAP, the this actions sends several cycles withTMS set to one, to enter state Test Logic Reset. Afterwards the reset the multicore TAP state configured with SYStem.CONFIG.TAPState is entered.

<var>

Local0…**Local7**	8 built-in local sequence variables. They are, for example, used with the action **CALL**. These built-in local sequence variables are **not** user-defined PRACTICE macros of the type **LOCAL**.
Result0 | **Result1**	2 built-in global sequence variables. They are, for example, used with the action **CALL**. These built-in global sequence variables are **not** user-defined PRACTICE macros of the type **GLOBAL**.

<operator>

<< >>	Shift-left and shift-right operators
+ - *	Add, Subtract, Multiply
/ %	Division and Modulo (since build 142495)
&	Binary AND
|	Binary OR
^	Binary XOR
~	Binary INVERT

There must be a blank before and after an operator used with the **ASSIGN** action.

<env>	TRACE32-internal environment variables listed below.
ConfigCHIP	Contains the chip number of the TRACE32 instance, which was set with the second parameter of **SYStem.CONFIG.CORE**.
ConfigCORE	Contains the core number of the TRACE32 instance, which was set with the first parameter of **SYStem.CONFIG.CORE**.
CORE	Contains the logical core (which is used with **CORE.select**)
MCTapState	Contains the setting **SYStem.CONFIG.TAPState** in a numeric value: • **7.** = Select-DR-Scan • **12.** = Run-Test/Idle Any JTAG sequence must start and end in this JTAG TAP state.
PhysicalCORE	Contains the physical core (which is used with **CORE.ASSIGN**)
PortTYPE	Contains setting **SYStem.CONFIG.DEBUGPORTTYPE** in a numeric value: • **1.** = JTAG • **2.** = SWD • **4.** = CJTAG
RUNNING	Contains 1 if the core is running. (since build 142495)
SLAVE	Contains 1 if **SYStem.CONFIG Slave** is set to **ON**. (0 otherwise) In an SMP system it can contain a zero only for the first logical core.
SYStemMode	Contains the **SYStem.Mode** where the following numeric values represent the following states: • **0.** = Down • **2.** = NoDebug • **4.** = Prepare (or during **SYStem.DETECT.DAP**) • **10.** = Attach (Only visible during **SYStem.Mode Attach**) • **11.** = Up When using **SYStem.CONFIG.MULTITAP.JtagSEQuence.Attach**, the assigned sequences will “see” the following numeric values of **SYStemMode** in case of the following events: • SYStem.Mode.Prepare : 4. • SYStem.Mode.Attach : 10. • SYStem.Mode.Up : 11. • SYStem.DETECT.DAP : 4.

Examples

Example 1: Write **0x00004000** to JTAG register **5**:

```
SYStem.CONFIG.TAPState RunTestIdle

;create a JTAG sequence named 'myAttach'
JTAG.SEQuence.Create myAttach
JTAG.SEQuence.Add    myAttach  Jump              2.  SLAVE == 0x0001
JTAG.SEQuence.Add    myAttach  RawShift          7.  0x3F  0x00
JTAG.SEQuence.Add    myAttach  PrePostRelative  +8.  -8.  +1.  -1.
JTAG.SEQuence.Add    myAttach  RawShift          4.  0x03  0x00

;                                                  register 5
JTAG.SEQuence.Add    myAttach  ShiftIrAndExit    4.  0x05
JTAG.SEQuence.Add    myAttach  RawShift          4.  0x03  0x00
JTAG.SEQuence.Add    myAttach  ShiftDrAndExit   32.  0x00004000
JTAG.SEQuence.Add    myAttach  RawShift          2.  0x01  0x00

;link the JTAG sequence 'myAttach' to the Attach event, which occurs
;when the SYStem.Attach command is executed
SYStem.CONFIG.MULTITAP.JtagSEQuence.Attach myAttach
```

Example 2: Get a flag from a JTAG register after setting 3 special function registers via their set-sequence. For a newly-created JTAG sequence, you can replace the sequence name in the **JTAG.SEQuence.Add** commands with a comma, as shown in this example.

```
SYStem.CONFIG.TAPState RunTestIdle

JTAG.SEQuence.Create WriteSPR                                    ;<index>
JTAG.SEQuence.Add , RawShift          4.   0x0003  0                ;0.

;                                  <length> <tdi>
JTAG.SEQuence.Add , ShiftIrAndExit    8.   0x00A0                 ;1.
JTAG.SEQuence.Add , RawShift          4.   0x0003  0                ;2.

;                                  <length> <tdi>   <tdo>
JTAG.SEQuence.Add , ShiftDrAndExit   32.   0x0000  Local2          ;3.

JTAG.SEQuence.Add , RawShift          2.   0x0001  0                ;4.
JTAG.SEQuence.Add , Jump             19.   Local2 & 0x28 != 0x08    ;5.
JTAG.SEQuence.Add , RawShift          4.   0x0003  0                ;6.
JTAG.SEQuence.Add , ShiftIrAndExit    8.   0x00B0                 ;7.
JTAG.SEQuence.Add , RawShift          4.   0x0003  0                ;8.
JTAG.SEQuence.Add , ShiftDrAndExit   32.   Local0                 ;9.
JTAG.SEQuence.Add , RawShift          6.   0x000D  0                ;10.
JTAG.SEQuence.Add , ShiftIrAndExit    8.   0x00C0                 ;11.
JTAG.SEQuence.Add , RawShift          4.   0x0003  0                ;12.
JTAG.SEQuence.Add , ShiftDrAndExit   32.   Local1                 ;13.
JTAG.SEQuence.Add , RawShift          6.   0x000D  0                ;14.
JTAG.SEQuence.Add , ShiftIrAndExit    8.   0x0020                 ;15.
JTAG.SEQuence.Add , RawShift          4.   0x0003  0                ;16.
JTAG.SEQuence.Add , ShiftDrAndExit    4.   0x0001                 ;17.
JTAG.SEQuence.Add , RawShift          2.   0x0001  0                ;18.
JTAG.SEQuence.Add , NOP                                           ;19.

JTAG.SEQuence.Create PowerCheck                                  ;<index>
JTAG.SEQuence.Add , PrePostRelative  +4.  -4.  +1.  -1.           ;0.

;     Call sequence "WriteSPR" with Local0 = 0x80000200, Local1 = 0x0037
JTAG.SEQuence.Add , CALL            WriteSPR 0x80000200  0x0037   ;1.
JTAG.SEQuence.Add , CALL            WriteSPR 0x80000100  0x0000   ;2.
JTAG.SEQuence.Add , CALL            WriteSPR 0x80000100  0x0008   ;3.

JTAG.SEQuence.Add , RawShift          4.  0x0003  0               ;4.
JTAG.SEQuence.Add , ShiftIrAndExit    8.  0x0008                  ;5.
JTAG.SEQuence.Add , RawShift          4.  0x0003  0               ;6.
JTAG.SEQuence.Add , ShiftDrAndExit   32.  0x0000  Result0         ;7.
JTAG.SEQuence.Add , RawShift          2.  0x0001  0               ;8.
JTAG.SEQuence.Add , ASSIGN          Result0 = Result0 & 0x0004    ;9.

JTAG.SEQuence.Execute PowerCheck
```

See also

- JTAG.SEQuence

JTAG.SEQuence.Append — Append one sequence to another sequence

ARC, ARM, Ceva-X, RH850, RISC-V, SDMA, TeakLite, TriCore, Xtensa

Format: **JTAG.SEQuence.Append** *<seq_name_dst> <seq_name_src>*

Appends all actions of one JTAG sequence to another JTAG sequence.

<seq_name_dst> Name of the sequence which is extended by the other sequence.

<seq_name_src> Name of the sequence from which the actions are copied.

See also

- JTAG.SEQuence

JTAG.SEQuence.Create — Create new JTAG sequence

ARC, ARM, Ceva-X, RH850, RISC-V, SDMA, TeakLite, TriCore, Xtensa

Format: **JTAG.SEQuence.Create** *<seq_name>*

Creates a name for a new empty JTAG sequence. To add the actual JTAG sequence, use **JTAG.SEQuence.ADD**.

<seq_name> Name of the JTAG sequence (without quotes). The name must start with a letter. The following characters can be letters, numbers, underscores, and minus signs, i.e. this regular expression: ^[a-zA-Z] [a-zA-Z0-9_-]* If you write the name in lower-case letters only, the name will be converted to upper-case only.

See also

- JTAG.SEQuence

Format: **JTAG.SEQuence.Delete** [*<seq_name>*]

Deletes a JTAG sequence. If the command is used without sequence name, all deletable sequences will be deleted.

Sequences which are locked cannot be deleted. A sequence is locked if it is assigned to an event (e.g. with **SYStem.CONFIG.MULTITAP.JtagSEQuence**) or if the sequence is an internal one which was created by TRACE32 after using **SYStem.CPU**. The names of internal sequences start with an underscore.

<seq_name> For a description of *<seq_name>*, see **JTAG.SEQuence.ADD**.

See also

- JTAG.SEQuence

Format:	**JTAG.SEQuence.Execute** *<seq_name>* [*<value>*…] [**/***<option>*]
<option>:	**CORE current** | **ALL** | *<core>*

Executes the given JTAG sequence on the active core selected with the command **CORE.select** or on the core selected with the option **CORE** of this command.

It is mandatory that the chosen JTAG sequence is written in such a way that it begins and ends in the JTAG TAP state, which is selected by **SYStem.CONFIG.TAPState**.

<value>	Max. 8 *<value>* parameters, which initialize the local variables **Local0**…**Local7** of the sequence.
<seq_name>	For a description of *<seq_name>*, see **JTAG.SEQuence.ADD**.
CORE current	Executes the JTAG sequence on the currently selected core, e.g. the core that is currently selected in the **CORE.SHOWACTIVE** window.
CORE ALL	Executes the JTAG sequence on all cores.
CORE *<core>*	Specifies the number of the logical core on which you want to execute the JTAG sequence.

What happens when the JTAG.SEQuence.Execute command is being executed?

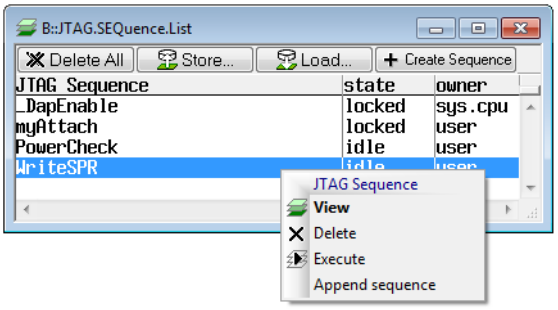
1. If the debugger is in **SYStem.Mode Down** or **NoDebug**, the debugger will automatically enable the pin drivers.
2. If the debugger is in **SYStem.Mode Up** or **Prepare**, the debugger will execute JTAG actions *before* and *after* the execution of the chosen JTAG sequence:
   - *Before* the execution the debugger will go from the JTAG park-state of the debug driver to the TAP state which is selected by **SYStem.CONFIG.TAPState**.
   - *After* the execution the debugger will go back to the JTAG park-state of the debug driver from the TAP state which is selected by **SYStem.CONFIG.TAPState**.
3. The JTAG sequence is executed.
4. After successful execution, the value of the global sequence variable **Result0** is displayed in the **AREA** window unless **JTAG.SEQuence.Execute** was run with the pre-command **SILENT** and unless the option **CORE ALL** was used.
   The values of **Result0** and **Result1** set by the last run of **JTAG.SEQuence.Execute** can be read via the PRACTICE function **JTAG.SEQuence.RESULT(0|1)**.

See also

■ JTAG.SEQuence ❑ JTAG.SEQuence.RESULT()

Format: **JTAG.SEQuence.List**

The command **JTAG.SEQuence.List** opens a window to display the names of all available user-defined and TRACE32-internal JTAG sequences.



Description of Toolbar and Popup Menu in the JTAG.SEQuence.List Window

Delete All	Delete all sequences which are not locked. See **JTAG.SEQuence.Delete**.
Store...	Saves all current sequences with owner “user” to a PRACTICE script (via command **STOre * JtagSEQuence**)
Load...	Executes a PRACTICE script. (See command **DO**) Typically you use that to execute the script previously created with **Store...**
Create Sequence...	Writes command **JTAG.SEQuence.Create** to the command line
View	Displays the selected sequence in a **JTAG.SEQuence.View** window.
Delete	Deletes the selected sequence. See **JTAG.SEQuence.Delete**.
Execute	Executes the selected sequence. See **JTAG.SEQuence.Execute**.
Append sequence	Writes the command **JTAG.SEQuence.Append** to the command line to append another sequence to the selected one.

Description of Columns in the JTAG.SEQuence.List Window:

JTAG sequence	The names of the currently existing JTAG sequences. The names of internal sequences (with owner “sys.cpu”) start with an underscore.
state	**idle**: Default state for a user sequence which was not yet attached to an event. **empty**: The sequence was created and is valid but does not yet contain any action. **invalid**: The sequence contains an invalid action or calls an invalid sequence. Invalid sequences can’t be executed or attached to an event **locked**: The sequence can be modified. That is the case when the sequence is an internal sequences (with owner “sys.cpu”) or when the sequence is attached to an event e.g. by **SYStem.CONFIG.MULTITAP.JtagSEQuence**
owner	**user**: User-defined JTAG sequence. **sys.cpu**: Internal sequences, specific for the current **SYStem.CPU** Internal sequences are created by TRACE32 *after* the **SYStem.CPU** command has been executed. They are always locked.

See also

- JTAG.SEQuence

JTAG.SEQuence.MemAccess.ADD Register sequence for memory access

ARC, ARM [build 142495 - DVD 02/2022]

Format:	**JTAG.SEQuence.MemAccess.ADD** *<addressrange> <seq_name> <mau>* [*<rtAccMode>*]
<mau>	**Byte** | **Word** | **Long** | **Quad** | **TByte** | **PByte** | **HByte** | **SByte**
<rtAccMode>	**stopped** | **SYStem** | **PREPARE** | **ALways**

This command registers a JTAG sequence to be used to read or write data via access class **JSEQ:**

<addressrange>	Address range for which a JTAG sequence should provide memory access
<seq_name>	Name of a previously created JTAG sequence, which should provide memory access for the given address range.

<mau>	Minimal Addressable Unit (MAU) (see below)
<rtAccMode>	Run-time access mode: Controls in which system states the memory access is allowed (see below)

When accessing the registered address range via the access class **JSEQ:** e.g. via command **Data.dump JSEQ:*<address>*** , the following parameters are set to the variables before calling the sequence:

- **Local0**: address (in MAU)
- **Local1**: length (in MAU)
- **Local2**: 1 on Write, 0 on read-access
- **Local3**: size of Minimal Addressable Unit (MAU) in bytes
- **Local4**: access-width (in bytes)
- **Result1**: 1 (must be set to 0 on success, otherwise the access is invalid)

On a read request the called JTAG sequence has to write the requested data to the buffer via sequence action **ToByteBuffer** in little-endian byte order and then set the global variable **Result1** to 0.

On a write request the called JTAG sequence has to read the requested data to the buffer via sequence action **FromByteBuffer** in little-endian byte order and then set the global variable **Result1** to 0.

The maximum amount of data, which should be read or write during one call is 512 Bytes.

If the transaction was successful, you have to set **Result1** to 0.
If the memory, you try to access, is read-only, do nothing on a write, except setting **Result1** to 0.

Minimal Addressable Unit (MAU):
While PowerView will access memory via class JSEQ always byte addressable, however setting MAU to any other value than Byte will have the following effect:
The addresses and the access size send to the JTAG sequence are always provided in MAUs instead of bytes. Thus the data read/write via JTAG sequences are always aligned to a multiple of the MAU.
When a user tries to read memory not aligned to the MAU, TRACE32 will request aligned data containing the requested data and extract the requested data from there.
When a user tries to write memory not aligned to the MAU, TRACE32 will read aligned data containing the requested data via the JTAG sequence, modify the affected data and then write back the modified data.

Run-time access mode:
When registering a JTAG sequence for memory access, you can specify under which system states the memory access is possible at all:

rtAccMode	System.Mode Down	System.Mode PREPARE	System.Mode Up & Stopped	System.Mode Up & Running
stopped (default)	no access	no access	EJSEQ: | JSEQ	no access
SYStem	no access	no access	EJSEQ: | JSEQ	EJSEQ: only if SYS.MemAccess()!= "DENIED"
PREPARE	no access	EJSEQ: | JSEQ	EJSEQ: | JSEQ	EJSEQ:
ALways	EJSEQ: | JSEQ:	EJSEQ: | JSEQ	EJSEQ: | JSEQ	EJSEQ:

Example:

```
// Create JTAG sequences:
JTAG.SEQ.Create myMemRead
JTAG.SEQ.ADD , ReadWrite 0x0A 32.  Local0        ; 0: set address
JTAG.SEQ.ADD , ReadWrite 0x09  4.  0x04          ; 1: trigger read access
JTAG.SEQ.ADD , ReadWrite 0x08  3.  0x00  Local2  ; 2: read state
JTAG.SEQ.ADD , Jump      3. Local2 & 0x05 != 0x04; 3: transaction done?
JTAG.SEQ.ADD , ReadWrite 0x0B 32.  0x00  Result0 ; 4: get result

JTAG.SEQ.Create myMemWrite
JTAG.SEQ.ADD , ReadWrite 0x0A 32.  Local0        ; 0: set address
JTAG.SEQ.ADD , ReadWrite 0x0B 32.  Local1        ; 1: set data
JTAG.SEQ.ADD , ReadWrite 0x09  4.  0x00          ; 2: trigger read access
JTAG.SEQ.ADD , ReadWrite 0x08  4.  0x00  Local2  ; 3: read state
JTAG.SEQ.ADD , Jump      3. Local2 & 0x05 != 0x04; 4: transaction done?

JTAG.SEQ.Create myMemAccess
;  Local0: address (in MAU)
;  Local1: length (in MAU)
;  Local2: 1 on Write, 0 on read-access
JTAG.SEQ.ADD , PrePostRelative  -4. +4. -1. +1.  ; 0
JTAG.SEQ.ADD , IRWidth           4.              ; 1
JTAG.SEQ.ADD , Jump     13. Local1 == 0x00       ; 2 : all accesses done?
JTAG.SEQ.ADD , Jump     7.  Local2 == 0x01       ; 3 : write access?
JTAG.SEQ.ADD , CALL     myMemRead  Local0        ; 4 : read 32-bit value
JTAG.SEQ.ADD , ToByteBuffer   Local0 Long Result0; 5 : save value
JTAG.SEQ.ADD , Jump     10.                      ; 6
JTAG.SEQ.ADD , FromByteBuffer Local0 Long Local6 ; 7 : load val. to write
JTAG.SEQ.ADD , CALL     myMemWrite Local0 Local6 ; 8 : write 32-bit value
JTAG.SEQ.ADD , Jump     10.                      ; 9
JTAG.SEQ.ADD , ASSIGN   Local0  =  Local0  + 1   ;10: increment address
JTAG.SEQ.ADD , ASSIGN   Local1  =  Local1  - 1   ;11: decrement length
JTAG.SEQ.ADD , Jump     2.                       ;12: goto next iteration
JTAG.SEQ.ADD , ASSIGN   Result1 =  0x00          ;13: indicate success

// Register JTAG sequence for accessing the given address range:
JTAG.SEQ.MemAccess.ADD JSEQ:0x10000--0x1FFFF myMemAccess Long PREPARE

// Display memory:
Data.dump JSEQ:0x10000--0x1FFFF
```

JTAG.SEQuence.MemAccess.List View registered memory accesses

ARC, ARM [build 142495 - DVD 02/2022]

Format: **JTAG.SEQuence.MemAccess.List**

This command opens a window, which shows your all address ranges which are currently registered for accesses via JTAG sequences.

To add a new entry to this window use command **JTAG.SEQuence.MemAccess.ADD**.

JTAG.SEQuence.MemAccess.ReMove Delete registered memory accesses

ARC, ARM [build 142495 - DVD 02/2022]

Format: **JTAG.SEQuence.MemAccess.ReMove** [*<addressrange>* | *<address>*]

Delete all memory accesses via JTAG sequences previously registered via **JTAG.SEQuence.MemAccess.ADD** which overlap with the given address or address-range.

Calling **JTAG.SEQuence.MemAccess.ReMove** without any address or address-range will delete all registered memory accesses.

JTAG.SEQuence.MemAccess.Replace Replace registered memory access

ARC, ARM [build 142495 - DVD 02/2022]

Format: **JTAG.SEQuence.MemAccess.Replace** *<addressrange> <seq_name> <mau> <rtAccMode>*

This command is a shortcut and does the same than executing the following two commands:
JTAG.SEQuence.MemAccess.ReMove.
JTAG.SEQuence.MemAccess.ADD.

JTAG.SEQuence.ReMove — Remove action from sequence

ARC, ARM, Ceva-X, RH850, RISC-V, SDMA, TeakLite, TriCore, Xtensa

Format: **JTAG.SEQuence.ReMove** *<seq_name>* [*<index>*]

Removes the indexed action from the specified sequence.

If the *<index>* parameter is omitted, all actions are removed from the sequence. However, unlike **JTAG.SEQuence.Delete**, the sequence name still exists, but the sequence itself is empty.

<seq_name> — For a description of *<seq_name>*, see **JTAG.SEQuence.ADD**.

<index> — Index of a command within the given JTAG sequence.
Parameter Type: Decimal.

See also

- JTAG.SEQuence

JTAG.SEQuence.Replace — Replace action inside sequence

ARC, ARM, Ceva-X, RH850, RISC-V, SDMA, TeakLite, TriCore, Xtensa

Format: **JTAG.SEQuence.Replace** *<seq_name> <index> <action>*

Like **JTAG.SEQuence.Add**, but instead of adding a new action to a sequence it replaces the indexed action *<index>* inside the specified sequence with a new *<action>*. For an illustrated example, see **JTAG.SEQuence.View**.

<seq_name> — For a description of *<seq_name>*, see **JTAG.SEQuence.ADD**.

<index> — Index of a command within the given JTAG sequence.
Parameter Type: Decimal.

<action> — For a description of *<action>*, see **JTAG.SEQuence.ADD**.

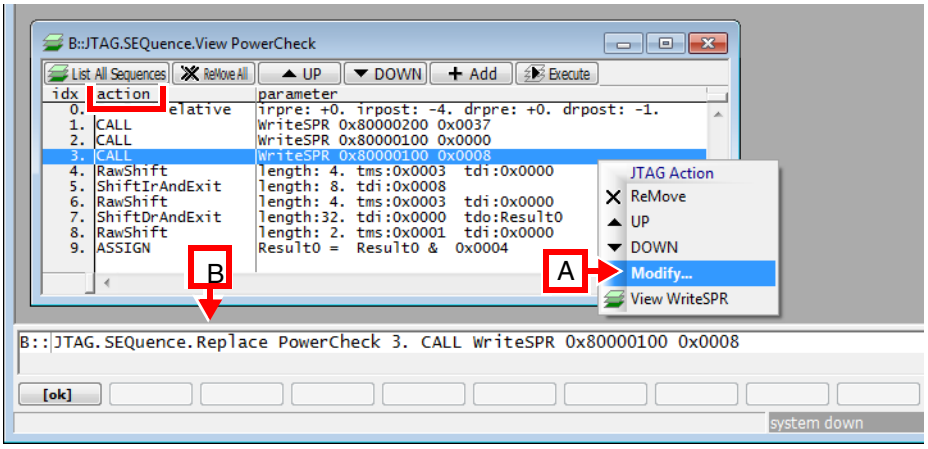
See also

- JTAG.SEQuence

Format: **JTAG.SEQuence.View** *<seq_name>*

Opens the **JTAG.SEQuence.View** window, displaying the specified JTAG sequence.

<seq_name> For a description of *<seq_name>*, see **JTAG.SEQuence.ADD**.



A To modify an individual action of a JTAG sequence, right-click the action and then select **Modify**.

B This inserts the command **JTAG.SEQuence.Replace** into the TRACE32 command line. You can now modify the action as required.

Description of Toolbar and Popup Menu in the JTAG.SEQuence.View Window

List All Sequences	Lists the sequence names in the **JTAG.SEQuence.List** window.
ReMove All	Removes all actions of the displayed sequence. The sequence name, however, is retained. See **JTAG.SEQuence.ReMove**.
UP, DOWN	Allow you to rearrange the individual actions of a sequence.
Add	Inserts the **JTAG.SEQuence.Add** command into the TRACE32 command line. You can now add a new action to the sequence.
Execute	Executes the displayed sequence. See **JTAG.SEQuence.Execute**.
View *<seq_name>* Only available for the action CALL	Displays the sequence that will be called from this sequence in a new **JTAG.SEQuence.View** window.

See also

- JTAG.SEQuence

Format: **JTAG.SHIFTREG** *<pattern>*

This command can be used to shift data if you are in the Shift-IR or Shift-DR state of the JTAG state machine. The shift is limited to 1024 bit.

It behaves the same way as **JTAG.SHIFTTDI**, but on the last bit the TMS is set `high` to leave the shift state. The last bit is still a valid data. This way you can shift a complete value with one command.

The function **JTAG.SHIFT()** can be used to read out the TDO data of the last shift. LSB first. Up to 64 bit.

Example: "JTAG.SHIFTREG 1 1 1 1 0 0 1 0 1 0 0 0 0 1 1 1" produces 16 TCK pulses having the specified TDI pattern. The left-most bit will be sent first. TMS keeps its level (low), on the last bit it gets high.

The pattern can also be specified for example in 8-bit, 16-bit, 32-bit formats:

```
JTAG.SHIFTREG %Byte 0x23 0x45 0x67
```

```
JTAG.SHIFTREG %Word 0x2345
```

```
JTAG.SHIFTREG %Long 0x23444565 0x22556667
```

For a more advanced example, see **JTAG**.

See also

- JTAG.SHIFTTDI
- JTAG.SHIFTTMS
- JTAG.SEQuence
- JTAG
- JTAG.LOCK
- JTAG.PIN
- JTAG.UNLOCK

▲ 'Custom JTAG Access' in 'Application Note JTAG Interface'

Format: **JTAG.SHIFTTDI** *<pattern>*

This command can be used to shift data if you are in the Shift-IR or Shift-DR state of the JTAG state machine. The size of the shift is limited to 1024 bits.

The function **JTAG.SHIFT()** can be used to read out the TDO data of the last shift. LSB first. Up to 64 bit.

Example: “JTAG.SHIFTTDI 1 1 1 1 0 0 1 0 1 0 0 0 0 1 1 1” produces 16 TCK pulses having the specified TDI pattern. The left-most bit will be sent first. TMS keeps its level (low).

The pattern can also be specified for example in 8-bit, 16-bit, 32-bit formats:

```
JTAG.SHIFTTDI %Byte 0x23 0x45 0x67
```

```
JTAG.SHIFTTDI %Word 0x2345
```

```
JTAG.SHIFTTDI %Long 0x23444565 0x22556667
```

For a more advanced example, see **JTAG**.

See also

- JTAG.SHIFTTMS
- JTAG.SHIFTREG
- JTAG.SEQuence
- JTAG
- JTAG.LOCK
- JTAG.PIN
- JTAG.UNLOCK
- JTAG.SHIFT()

▲ ’Custom JTAG Access’ in ’Application Note JTAG Interface’

Format: **JTAG.SHIFTTMS** *<pattern>*

This command can be used to walk to a different state in the JTAG state machine. The shift is limited to 1024 bit.

Example: “JTAG.SHIFTTMS 1 1 0 0 1 0 1” produces 7 TCK pulses having the specified TMS bit pattern. The left-most bit will be sent first. TDI keeps its level.

The pattern can also be specified for example in 8-bit, 16-bit, 32-bit formats:

```
JTAG.SHIFTTMS %Byte 0x23 0x45 0x67
```

```
JTAG.SHIFTTMS %Word 0x2345
```

```
JTAG.SHIFTTMS %Long 0x23444565 0x22556667
```

For a more advanced example, see **JTAG**.

See also

- JTAG.SHIFTTDI
- JTAG.SHIFTREG
- JTAG.SEQuence
- JTAG
- JTAG.LOCK
- JTAG.PIN
- JTAG.UNLOCK

▲ ’Custom JTAG Access’ in ’Application Note JTAG Interface’

JTAG.SWD.Init — Initialize the debug port

Format: **JTAG.SWD.Init**

Initializes the debug port and switches to SWD.

JTAG.SWD.ReadDapBus — Read register from DAP

Format: **JTAG.SWD.ReadDapBus** [**AP** | **DP** | **IDCODE**]

Reads a register from the DAP via SWD. **JTAG.SWD.ReadDapBus** additionally reads out the DP:RDBUF register when an AP register is read.

JTAG.SWD.ReadScan — Read register from DAP

Format: **JTAG.SWD.ReadScan** [**AP** | **DP** | **IDCODE**]

Reads a register from the DAP via SWD.

JTAG.SWD.Select — Configure SWD multi drop target selection

Format: **JTAG.SWD.Select** *<pattern>*

Selects to configure SWD multi drop target selection.

Format: **JTAG.SWD.SHIFT** *<pattern>*

This command can shift data by using the SWIO pin with the configured clock. The command accepts single symbols for single cycles or larger words using masks to mark positions where data is read back. The function **JTAG.SHIFT()** returns in maximum 64 bit shift data counting all cycles passed to **JTAG.SWD.SHIFT**.

Examples:

```
; 1. shift 3 times symbol ‘1’ to target
; 2. shift 3 cycles and read back SWIO from target
; 3. shift 3 cycles symbol ‘0’ to target
JTAG.SWD.SHIFT 1 1 1 X X X 0 0 0
; read back result
print "result of X X X " (JTAG.SHIFT()>>>3.)&0x7
```

```
; 1. shift 1 cycle and read back SWIO from target
; 2. shift 32 bit data
; 3. shift 1 bit '1'
JTAG.SWD.SHIFT X %LONG 0xC0FFEAFFE 1
```

```
; 1. shift 32 cycles and read back SWIO from target
; 2. shift 1 cycle and read back SWIO from target
JTAG.SWD.SHIFT %long 0xXXXXXXXX X
```

```
; shift line reset
JTAG.SWD.SHIFT %long 0xFFFFFFFF %word 0xFFFF 1 1 0 0
```

```
; shift 'park' and 'trn' symbol (just don't drive swio line)
JTAG.SWD.SHIFT X X
```

JTAG.SWD.WriteDapBus Write register to DAP

Format: **JTAG.SWD.WriteDapBus** [**AP** | **DP** | **ABORT**]

Writes a register to the DAP via SWD.

Format: **JTAG.SWD.WriteScan** [**AP** | **DP** | **ABORT**]

Writes a register to the DAP via SWD.

JTAG.UNLOCK Hand the JTAG port control back to the debugger

Format: **JTAG.UNLOCK**

Re-enables all debugger activity on the JTAG port which might have been disabled by **JTAG.LOCK** in order to avoid debugger JTAG access in between a manual control sequence.

You need to return to the JTAG pause parking position before performing the UNLOCK. Refer to **JTAG.LOCK** for more information.

See also

- JTAG
- JTAG.LOCK
- JTAG.PIN
- JTAG.SHIFTREG
- JTAG.SHIFTTDI
- JTAG.SHIFTTMS

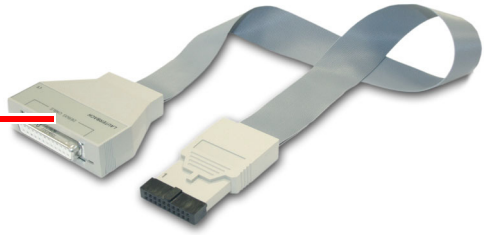
▲ 'Custom JTAG Access' in 'Application Note JTAG Interface'

Format: **JTAG.USECLOCK** [**ON** | **OFF**]

Default: OFF

The command **JTAG.USECLOCK** affects only Lauterbach hardware labeled with DEBUG CABLE.

DEBUG CABLE



ON The debug clock set with **SYStem.JtagClock** is taken into account when the JTAG.SHIFT* commands are used:
- **JTAG.SHIFTREG**
- **JTAG.SHIFTTDI**
- **JTAG.SHIFTTMS**

OFF The debug clock is ignored.

Example:

```
SYStem.JtagClock 1Mhz      ; configure JTAG clock
JTAG.USECLOCK ON           ; use the JTAG clock instead of pin toggling
                           ; for DEBUG CABLE based solutions
JTAG.LOCK                  ; prevent debugger from interrupting the
                           ; sequence
JTAG.PIN ENable            ; enable output buffers of DEBUG CABLE
JTAG.SHIFTTMS 1 1 1 1 1 0  ; reset TAP controller and
                           ; go to RUN-TEST-IDLEstate
```

See also

- JTAG

[Examples]

Format: **JTAG.X7EFUSE** [*/<option>*]

<option>:
DEVICE *<device>*
SLR *<slr>*
IRPRE *<value>*
IRPOST *<value>*
DRPRE *<value>*
DRPOST *<value>*
USER *<value>*
USERL *<value>*
USERH *<value>*
KEY *<key_file>*
NOKEY
ADD
UNSAFE
ENABLE
CFG_AES_ONLY
AES_EXCLUSIVE
W_EN_B_KEY_USER
R_EN_B_KEY
R_EN_B_USER
W_EN_B_CNTL

Programs the eFuse array in Xilinx 7-series (Virtex-7, Kintex-7 and Artix-7) FPGAs. These fuses contain the following information:

- AES key for loading encrypted bitstreams (256 bit)
- USER code accessible to loaded FPGA design (32 bit)
- Protection flags for eFuses and configuration (6 flags)

It is highly recommended to read the Xilinx application note XAPP 1239, *Using Encryption to Secure a 7 Series FPGA Bitstream*, prior to using this command.

The command prints information about its progress, including the unique device identifier DNA, to the **AREA.view** window. It is recommended to use **AREA.OPEN** and **AREA.CLOSE** to set up persistent logging of the performed operations to a file.

After the command has been executed, information about its outcome can be programmatically queried using the PRACTICE functions **JTAG.X7EFUSE.RESULT()**, **JTAG.X7EFUSE.CNTL()**, **JTAG.X7EFUSE.DNA()**, **JTAG.X7EFUSE.KEY()** and **JTAG.X7EFUSE.USER()**.

.

DEVICE *<device>*	Selects the device type to be configured. The following devices are known to TRACE32: • XC7A15T, XA7A15T • XC7A35T, XA7A35T • XC7A50T, XA7A50T, XQ7A50T • XC7K70T • XC7A75T, XA7A75T • XC7A100T, XA7A100T • XQ7A200T, XC7A200T, XQ7A200T, XC7K160T • XC7K325T, XQ7K325T, XC7K355T • XC7K410T, XQ7K410T • XC7K420T • XC7K480T • XC7VX330T • XC7VX415T • XC7VX485T • XC7VX550T • XC7V585T • XC7VX690T • XC7VX980T • XC7VX1140T (*) • XC7V2000T (*) • XC7VH580T (*) • XC7VH870T (*) (*) These devices contain multiple Super Logic Regions (SLRs) and require the **SLR** option.
SLR *<slr>*	Selects the Super Logic Region (SLR) to be configured. The largest 7-series devices are composed of multiple SLRs. In this case, each SLR contains its own eFuse array. All eFuse arrays must be separately programmed to the exact same values. Using this option, one SLR can be programmed at a time. The valid indices are as follows: • XC7VX1140T: 0, 1, 2, 3 • XC7V2000T: 0, 1 • XC7VH580T: 0, 1 • XC7VH870T: 0, 1, 2 • All others: 0 (the **SLR** option can also be omitted)
IRPRE *<value>* **IRPOST** *<value>* **DRPRE** *<value>* **DRPOST** *<value>*	Configures the JTAG chain parameters. These options work like the corresponding commands in **SYStem.Option**: They specify how the FPGA device to be programmed can be reached in the JTAG chain. All values default to 0 if the corresponding option is not specified.

USER *<value>* **USERL** *<value>* **USERH** *<value>*	Specifies the USER code to be programmed. The USER code is a 32-bit value that is accessible to the logic inside the FPGA. Unless explicitly disabled, it can also be read via the JTAG port. The value is stored in two parts: USER[31:8] can be programmed independently of the AES key, while USER[7:0] must be programmed at the same time. If only part of the USER code should be programmed, use **USERL** and **USERH** to program USER[7:0] and USER[24:0], respectively. These commands also expect a 32-bit value, but require that the high or low bits be zero. If **USERL** or **USER** is specified, it is also required to specify an AES key with **KEY** or explicitly accept that no key can be loaded in the future using **NOKEY**
KEY *<key_file>*	Specifies the *.nky file containing the AES key. The *.nky file is generated by the Xilinx Vivado toolchain. When using **KEY**, the values for USER[7:0] must also be permanently programmed. If no USER code should ever be programmed, use **USERL** 0.
NOKEY	Explicitly specifies that no key should be programmed. Used in conjunction with **USERL** or **USER**.
ADD	Allow adding information to a partially programmed eFuse array. Under some circumstances, it is possible to perform further programming: • Program KEY and USER[7:0] if neither have already been programmed and none of the W_EN_KEY_USER, R_EN_KEY and R_EN_USER fuses are set. • Program USER[31:8] if not already programmed and none of the W_EN_KEY_USER, R_EN_KEY and R_EN_USER fuses are set. • Program further CNTL bits if W_EN_CNTL is not set. When programming the fuses for the first time, dedicated test fuses are blown to verify correct programming. This is not possible if the array has already been partially programmed. It is therefore recommended to program KEY, USER and CNTL bits using a single invocation of the **JTAG.X7EFUSE** command.
UNSAFE	Allows combinations of operations that could leak key data. When programming the AES key, it is recommended to immediately set the R_EN_KEY CNTL flag to disallow reading the key. Additionally, if the R_EN_KEY or R_EN_USER flags are set, writing to KEY or USER is disabled. It is therefore recommended to also set W_EN_KEY_USER as a further precaution. When using this flag, programming will begin operation even if these conditions are not met. Without this flag, it will report an error instead, before any fuses are programmed.
ENABLE	Allows writing to an eFuse array. This option is required to write to any eFuse. If the option is missing, but the other options specify that a write operation is necessary, the command will fail and report the list of operations that would have been performed. This is the equivalent to a dry-run operation.

CFG_AES_ONLY	Forces configuring the FPGA through the AES decryptor. If this option is set, it will be impossible to load the FPGA without knowledge of the AES key. See XAPP1239 for further details.
AES_EXCLUSIVE	Disables partial reconfiguration from external interfaces. See XAPP1239 for further details.
W_EN_B_KEY_USER	Disables writing to KEY and USER values. See XAPP1239 for further details.
R_EN_B_KEY	Disables reading the KEY value. As a side effect, this also disables writing to KEY and USER values. See XAPP1239 for further details.
R_EN_B_USER	Disables reading the USER value through the JTAG interface. The USER value will still be accessible to user logic inside the FPGA. As a side effect, this also disables writing to KEY and USER values. See XAPP1239 for further details.
W_EN_B_CNTL	Disables writing to the CNTL flags. See XAPP1239 for further details.

Preconditions

Before programming, please verify the following:

- Any external configuration sources must be disabled. TRACE32 will clear the FPGA configuration upon excution of this command, but cannot prevent reconfiguration from external sources. For eFuse programming, the FPGA must be unconfigured.
- **SYStem.JtagClock** must be set to exactly 1 MHz.
- Observe the I_{FS} and T_j conditions from the device data sheet.
- The V_{CCINT} must be within 0.97 V and 1.03 V. This applies to all devices, including -2L and -1L devices.
- Verify the reliability of the communication between TRACE32 and the FPGA by loading a bitstream using **JTAG.LOADBIT**.

Failure Conditions

The command **JTAG.X7EFUSE** first reads and verifies the IDCODE register multiple times, then resets the FPGA configuration. It then reads the current state of the CNTL bits and any accessible eFuse data. If a communication error occurs or it is determined that the requested operations cannot be performed, the command fails and **JTAG.X7EFUSE.RESULT()** returns the value 2. If it is determined that the requested values are already programmed, the command succeeds and **JTAG.X7EFUSE.RESULT()** returns 0.

If programming is to be performed, the command first blows dedicated test fuses to verify correct operation. If an error is discovered during this period, the command fails and **JTAG.X7EFUSE.RESULT()** returns 3. If the operation is requested by the user, the command writes and verifies KEY and USER data. Regardless of the result of this operation, the command will attempt to program the CNTL bits as specified by the user. This

is done to ensure that any key security settings also protect incorrectly written keys. If programming or verification of KEY or USER failed, but programming and verifying CNTL succeeded, **JTAG.X7EFUSE.RESULT()** returns 4. If programming or verifying CNTL fails, **JTAG.X7EFUSE.RESULT()** returns 5 to indicate that the device could potentially leak confidential information.

If no error occurs, but at least one eFuse was programmed, **JTAG.X7EFUSE.RESULT()** returns 1.

Examples

Example 1: This script programs the KEY, USER and CNTL flags with a single command, treats KEY as a secret value and leaves USER readable via JTAG. In addition, further writes to control flags are disabled:

```
SYStem.JtagClock 1.0MHz

JTAG.X7EFUSE /DEVICE XC7K325T /KEY my_key.nky /USER 0xC0DEC0DE \
             /R_EN_B_KEY /W_EN_B_KEY_USER /W_EN_B_CNTL \
             /ENABLE

IF JTAG.X7EFUSE.RESULT()!=1.
(
    ; handle the case where programming or verifying failed
)
```

Example 2: This script initially programs only the USER[31:8] code, and at a later time programs the KEY. The FPGA configuration and reconfiguration is forced to use the AES decryptor:

```
SYStem.JtagClock 1.0MHz

JTAG.X7EFUSE /DEVICE XC7K325T /USERH 0xC0DEC000 /ENABLE
; omitted: do error checking as above

; later, possibly in a separate TRACE32 session:

JTAG.X7EFUSE /DEVICE XC7K325T /KEY my_key.nky /USERL 0x000000DE \
             /R_EN_B_KEY /W_EN_B_KEY_USER /CFG_AES_ONLY /AES_EXCLUSIVE \
             /W_EN_B_CNTL /ENABLE /ADD
; omitted: do error checking as above
```

Example 3: This script reads all available information from a device:

```
SYStem.JtagClock 1.0MHz

JTAG.X7EFUSE /DEVICE XC7K325T

IF JTAG.X7EFUSE.RESULT()==0.
(
    PRINT "Device DNA: "+JTAG.X7EFUSE.DNA()
    PRINT "Device CNTL:"+JTAG.X7EFUSE.CNTL()
    PRINT "Device KEY: "+JTAG.X7EFUSE.KEY()
    PRINT "Device USER:"+JTAG.X7EFUSE.USER()
)
ELSE
(
    ; some error occurred
)
```

Depending on the CNTL flags, it may be impossible to read the KEY or USER, in which case an all-zero KEY or USER value will be returned.

See also

■ JTAG

▲ 'JTAG Functions' in 'General Function Reference'

Format: **JTAG.XUSEFUSE** [*/<option>*]

<option>:
READ *<value>*
IRPRE *<value>*
IRPOST *<value>*
DRPRE *<value>*
DRPOST *<value>*
SLR *<value>*
RSA *<value>*
KEY *<value>*
USER *<value>*
USER128 *<value>*
CNTL *<value>*
SEC *<value>*
NKZ *<key_file>*

Programs the eFUSE registers in Xilinx UltraScale FPGAs. UltraScale+ devices are **not** supported. These fuses contain the following information:

- SHA-3 hash for RSA bitstream authentication
- AES key for loading encrypted bitstreams (256 bit)
- Short USER code accessible to loaded FPGA design (32 bit)
- Long USER code readable via special JTAG instruction (128 bit)
- Configuration flags for accessing the eFuse registers (10 flags)
- Security flags for configuring e. g. the use of the AES key (7 flags)

It is highly recommended to read the Xilinx User Guide UG 570, *UltraScale Architecture Configuration* as well as Xilinx application note XAPP 1283, *Internal Programming of BBRAM and eFUSEs*, prior to using this command.

The command prints information about its progress, including the unique device identifier DNA, to the **AREA.view** window. It is recommended to use **AREA.OPEN** and **AREA.CLOSE** to set up persistent logging of the performed operations to a file.

After the command has been executed, information about its outcome can be programmatically queried using the PRACTICE function **JTAG.XUSEFUSE.RESULT()**. If the command has been invoked with the **/READ** option, the following PRACTICE functions can be used to get the current eFUSE configuration of the FPGA:

- **JTAG.XUSEFUSE.CNTL()**, **JTAG.XUSEFUSE.DNA()**,
- **JTAG.XUSEFUSE.KEY()**, **JTAG.XUSEFUSE.RSA()**,
- **JTAG.XUSEFUSE.SEC()**, **JTAG.XUSEFUSE.USER()**, and **JTAG.XUSEFUSE.USER128()**

READ *<value>*	Reads the values currently stored in the eFUSE registers. If not prevented by the configuration bits in the CNTL register, the values can afterwards be obtained by using the PRACTICE functions **JTAG.XUSEFUSE.*()**. **NOTE**: Because the private AES key cannot be directly read, the user must provide a value as key for comparing. **NOTE**: This option can only be combined with the options **IRPRE**, **IRPOST**, **DRPRE**, **DRPOST**, and **SLR**.
IRPRE *<value>* **IRPOST** *<value>* **DRPRE** *<value>* **DRPOST** *<value>*	Configures the JTAG chain parameters. These options work like the corresponding commands in **SYStem.Option**: They specify how the FPGA device to be programmed can be reached in the JTAG chain. All values default to 0 if the corresponding option is not specified.
SLR *<slr>*	Selects the Super Logic Region (SLR) to be configured. The largest UltraScale series devices are composed of multiple SLRs. In this case, each SLR contains its own eFUSE array. All eFUSE arrays must be separately programmed. Using this option, one SLR can be programmed at a time. The valid indices are as follows: • XCKU085: 0, 1 • XCKU115: 0, 1 • XCVU125: 0, 1 • XCVU160: 0, 1, 2 • XCVU190: 0, 1, 2 • XCVU440: 0, 1, 2 • All others: 0 (the SLR option can also be omitted)
RSA *<value>*	Specifies the hash for RSA authentication to be programmed. The value is the SHA-3 hash of the public key used for RSA bitstream authentication.
KEY *<value>*	Specifies the AES key for bitstream decryption.
USER *<value>*	Specifies the USER code to be programmed. The USER code is a 32-bit value that is accessible to the logic inside the FPGA. Unless explicitly disabled, it can also be read via the JTAG port.
USER128 *<value>*	Specifies the 128-bit USER code to be programmed.
CNTL *<value>*	Specifies the value to be programmed into the eFUSE Control Register. The Control Register contains 10 flags which can be used to restrict the access to the different eFUSE registers. A more detailed information on the Control Register can be found in the Xilinx User Guide UG570.

SEC *<value>*	Specifies the value to be programmed into the eFUSE Security Register. The Security Register contains 7 flags which can be used to configure the usage of the different eFUSE registers. A more detailed information on the Security Register can be found in the Xilinx User Guide UG570.
NKZ *<key_file>*	Specifies the *.nkz file containing several values to be programmed. All values in the given *.nkz file are programmed to the eFUSEs. The *.nkz file can be generated by the Xilinx Vivado toolchain and may contain the values for each eFUSE register. Example 3 includes a *.nkz file with all possible keywords and values. **NOTE**: This option can only be combined with the options **IRPRE**, **IRPOST**, **DRPRE**, **DRPOST**, and **SLR**.

Preconditions

Before programming, please verify the following:

- Any external configuration sources must be disabled. TRACE32 will clear the FPGA configuration upon execution of this command, but cannot prevent reconfiguration from external sources. For eFUSE programming, the FPGA must be unconfigured.
- **SYStem.JtagClock** must be set to exactly 1 MHz.
- Observe the I_{FS} and T_j conditions from the device data sheet.
- The V_{CCINT} must be within 0.97 V and 1.03 V. This applies to all devices, including -2L and -1L devices.

Verify the reliability of the communication between TRACE32 and the FPGA by loading a bitstream using **JTAG.LOADBIT**.

Failure Conditions

The command **JTAG.XUSEFUSE** first reads and verifies the IDCODE register, then resets the FPGA configuration. Afterwards it either reads the current accessible eFUSE data or writes the given values to the eFUSES. If a communication error occurs or it is determined that the requested operations cannot be performed, the command fails and **JTAG.XUSEFUSE.RESULT()** returns the value 1. If it is determined that the requested values are already programmed, the command succeeds and **JTAG.XUSEFUSE.RESULT()** returns 0.

If programming is to be performed, the command verifies the written values. If an error is discovered during this period or during reading, the command fails and **JTAG.XUSEFUSE.RESULT()** returns 2. An exception is made when the AES key value could not be verified. In this case **JTAG.XUSEFUSE.RESULT()** returns 3.

If no error occurs when executing this command, **JTAG.XUSEFUSE.RESULT()** returns 0.

Examples

Example 1: This script programs the KEY, USER and CNTL registers with a single **JTAG.XUSEFUSE** command:

```
SYStem.JtagClock 1.0MHz

JTAG.XUSEFUSE /KEY 0x0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF\
0123456789ABCDEF /USER 0x01234567 /CNTL 0x00080

IF JTAG.X7EFUSE.RESULT()!=0.
(
    ; handle the case where programming or verifying failed
)
```

Example 2: This script reads all available information from a device:

```
SYStem.JtagClock 1.0MHz

JTAG.XUSEFUSE /READ 0xBE99CB71765F308C8EC0D67ACCAA3146\
FBD2A1C67E1394400937873B8A4BDFFF

IF JTAG.XUSEFUSE.RESULT()==0.
(
  PRINT "Device DNA: 0x"+JTAG.XUSEFUSE.DNA()
  PRINT "Device CNTL: 0x" +JTAG.XUSEFUSE.CNTL()
  PRINT "Device Key: 0x"+JTAG.XUSEFUSE.KEY()
  PRINT "Device SEC: 0x" +JTAG.XUSEFUSE.SEC()
  PRINT "Device RSA: 0x"+JTAG.XUSEFUSE.RSA()
  PRINT "Device User: 0x" +JTAG.XUSEFUSE.USER()
  PRINT "Device 128: 0x"+JTAG.XUSEFUSE.USER128()
)
ELSE IF JTAG.XUSEFUSE.RESULT()==3.
(
  ; handle an error
  PRINT "Device DNA: 0x"+JTAG.XUSEFUSE.DNA()
  PRINT "Device CNTL: 0x" +JTAG.XUSEFUSE.CNTL()
  PRINT "The given AES key does not match the stored value"
  PRINT "Device SEC: 0x" +JTAG.XUSEFUSE.SEC()
  PRINT "Device RSA: 0x"+JTAG.XUSEFUSE.RSA()
  PRINT "Device User: 0x" +JTAG.XUSEFUSE.USER()
  PRINT "Device 128: 0x"+JTAG.XUSEFUSE.USER128()
)
ELSE
(
  PRINT "An error occurred while reading the eFUSE registers"
)
```

Depending on the CNTL flags, it may be impossible to read a eFUSE register, in which case an all-one value will be returned.

Example 3: This script writes all available information from a device within a JTAG chain:

```
SYStem.JtagClock 1.0MHz

JTAG.XUSEFUSE /IRPRE 0x6 /DRPRE 0x12 /NKZ my_values.nkz

IF JTAG.XUSEFUSE.RESULT()!=0.
(
    ; some error occurred
)
```

The following shows the contents of the file my_values.nkz:

```
;This is an example of a .nkz file containing all information*/
Device xcku040;
EncryptKeySelect EFUSE;

Key0 0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF;

Program FUSE_USER 13579BDF;

Program FUSE_USER_128 0123456789ABCDEF0123456789ABCDEF;

Program FUSE_CNTL_0_R_DIS_KEY 1;
Program FUSE_CNTL_1_R_DIS_USER 0;
Program FUSE_CNTL_2_R_DIS_SEC 0;
Program FUSE_CNTL_5_W_DIS_CNTL 0;
Program FUSE_CNTL_6_R_DIS_RSA 0;
Program FUSE_CNTL_7_W_DIS_KEY 1;
Program FUSE_CNTL_8_W_DIS_USER 0;
Program FUSE_CNTL_9_W_DIS_SEC 0;
Program FUSE_CNTL_15_W_DIS_RSA 1;
Program FUSE_CNTL_16_W_DIS_USER_128 1;

Program FUSE_SEC_0_CFG_AES_ONLY 1;
Program FUSE_SEC_1_EFUSE_AES_KEY 0;
Program FUSE_SEC_2_RSA_AUTH 0;
Program FUSE_SEC_3_JTAG_DISABLE 0;
Program FUSE_SEC_4_SCAN_DISABLE 0;
Program FUSE_SEC_5_CRYPT_DISABLE 1;
Program FUSE_SEC_6_OBFUSCATED_KEY_ENABLE 0;

RsaPublicKeyDigest 0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF\
0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF;
```

NOTE: The first three lines of the file above contain information which is omitted by the command. All other lines contain a key-value-pair which is evaluated. The key and associated register value must not be spread over separate lines. As shown for the key `RsaPublicKeyDigest`.

See also

■ JTAG ❑ JTAG.MIPI34() ❑ JTAG.PIN() ❑ JTAG.SHIFT()

▲ 'JTAG Functions' in 'General Function Reference'
▲ 'Release Information' in 'Legacy Release History'

The **JTAG.CJTAG** command group allows a manual control of the IEEE 1149.7 (aka CJTAG). An example script is available under `~~/demo/etc/jtag/test_cjtag_idcode.cmm`.

See also

- JTAG.CJTAG.COMMAND
- JTAG.CJTAG.START
- JTAG

JTAG.CJTAG.COMMAND Send command to the chip

Format: **JTAG.CJTAG.COMMAND** *<cp0> <cp1>*

The IEEE 1149.7 (aka CJTAG) defines a method to send commands to the chip internal 1149.7 controller module.

These commands are made up of two 5-bit numbers, “cp0” and “cp1”.

The exact shift sequence how to send such a command is complicated. **JTAG.CJTAG.COMMAND** provides an interface to execute such a command; you have to provide the “cp0” and “cp1” number values.

See also

- JTAG.CJTAG

JTAG.CJTAG.START Access the target via CJTAG

Format: **JTAG.CJTAG.START**

If you intend to access your target via IEEE 1149.7 (aka CJTAG) in 2-pin mode, then when the connection is initialized, the debugger will first of all switch the target to 2-pin mode.

If you want to access the target via raw JTAG shifts before the debugger has initialized the connection, then you have to explicitly switch to 2-pin mode, before accessing the target via IEEE 1149.7 (aka CJTAG) in 2-pin mode.

See also

- JTAG.CJTAG

The command **JTAG.CJTAG.START** executes this explicit switch to 2-pin mode.