LAUTERBACH
DEVELOPMENT TOOLS

General Commands Reference Guide L

Release 09.2024

MANUAL

General Commands Reference Guide L

General Commands Reference Guide L

Version 05-Oct-2024

History

22-Apr-2024 New option **/NoExec** for LA.IMPORT.TARMAC command.

25-Mar-2024 New option **/COVerableItems** for List.EXPORT.auto and List.EXPORT.Mix commands.

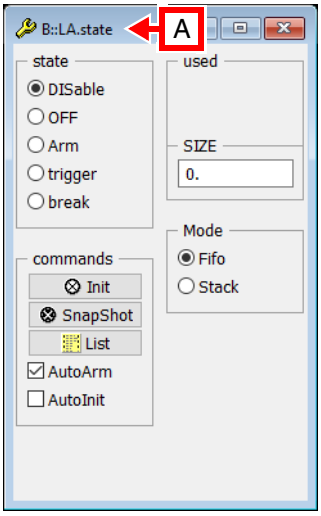
23-Feb-2024 New command LA.IMPORT.TARMAC.

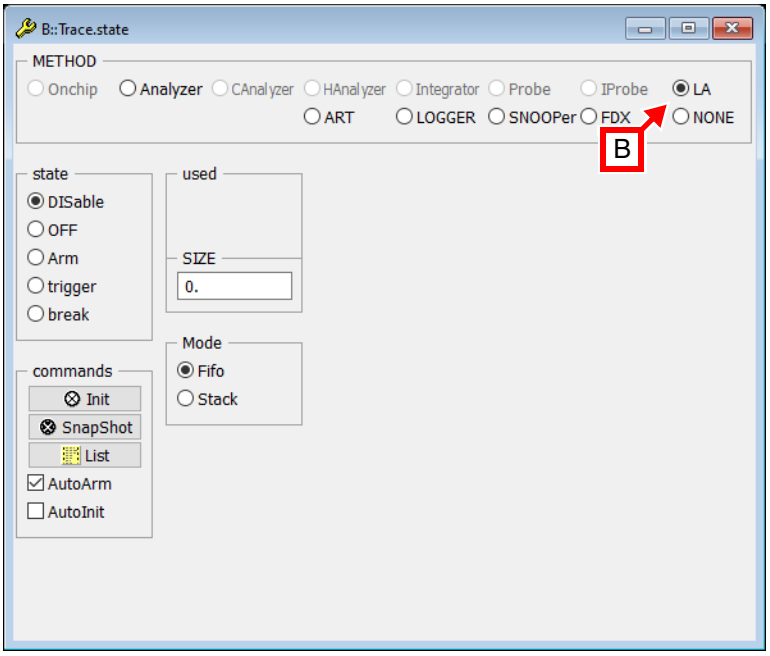
LA Logic analyzer

The trace method LA is used for remote control of logic analyzer systems.

For selecting and configuring the trace method LA, use the TRACE32 command line or a PRACTICE script (*.cmm) or the **LA.state** window [**A**].

Alternatively, use the **Trace.state** window: click the option **LA** or execute the command **Trace.METHOD LA** in order to select the trace method **LA** [**B**].





The chapter **"LA-specific Trace Commands"**, page 8 describes the LA-specific configuration commands. While the chapter **"Generic LA Trace Commands"**, page 20 lists the LA trace analysis and display commands, which are shared with other TRACE32 trace methods.

See also

- Trace.METHOD

- 'Generic LA Trace Commands' in 'General Commands Reference Guide L'
- 'Release Information' in 'Legacy Release History'

LA.IMPORT

Import trace information

[Example]

The **LA.IMPORT** command group is used to load trace data from a file into TRACE32 and to analyze it just like data recorded with a TRACE32 trace tool.

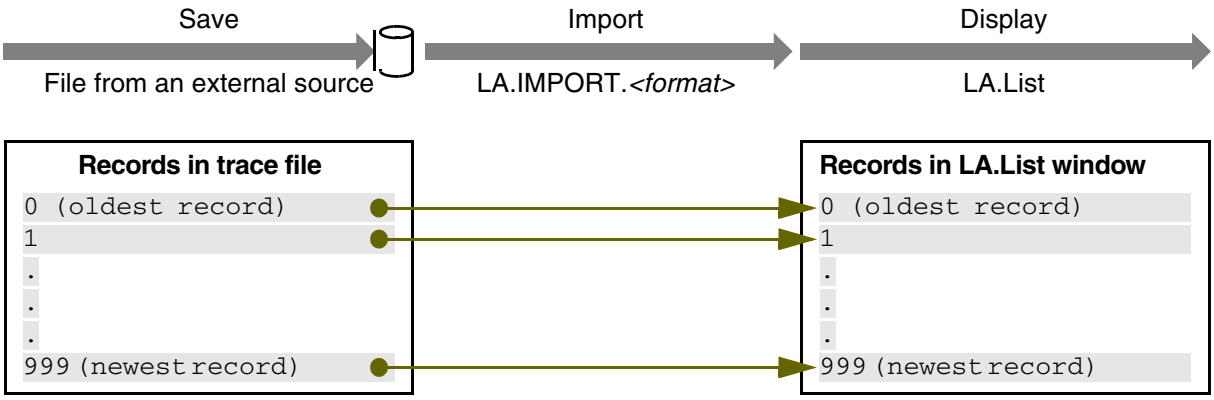
The trace data can be obtained by the application software itself or by another tool or by TRACE32 in a previous debug session in which the processing could not be performed for some reasons.

Trace data successfully obtained and analyzed by TRACE32 can be stored by **<trace>.SAVE** and re-viewed by using the **<trace>.LOAD** command. This is the more convenient way because **<trace>.SAVE** stores a lot of additional information used for the analysis. **LA.IMPORT** imports only the trace raw data. For proper processing you need to inform the debugger about all the trace-relevant circumstances.

All kind of trace postprocessing is only possible with the trace method **LA** (Logic Analyzer). Therefore you need to use **LA.IMPORT** and **LA.*** command group for all analysis commands or better switch the trace method to **LA** (**Trace.METHOD LA**) and use the command group **Trace.*** for all further operations.

LA.IMPORT supports different kinds of trace data and formats. Therefore different commands are provided.

Most trace data is stored in the file in the timely order the data had been generated.

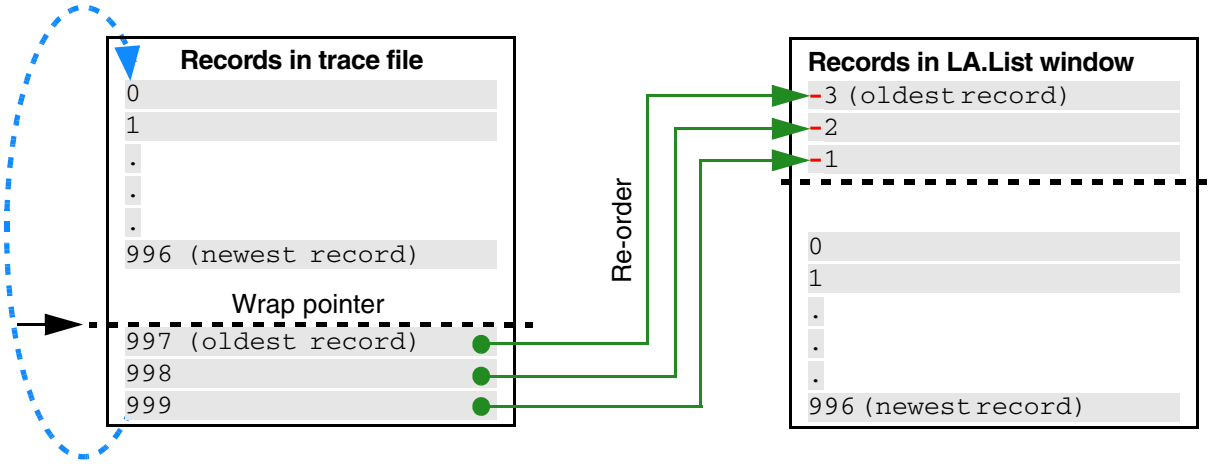


An exception is the on-chip trace buffer, which is typically used as a circular ring buffer overwriting the trace data all the time until the point of interest is reached.

If this buffer is saved into a file, you need to know the wrap pointer for being able to get the data in a timely order. **LA.IMPORT.WRAP** and **LA.IMPORT.GUESSWRAP** will deal with this concern.

Process:

Result:



For post processing trace data loaded by **LA.IMPORT** you need to take the following steps:

1. Start TRACE32 to run as simulator (config.t32 -> `PBI=SIM`). You neither need a debugger hardware nor a target. You can run TRACE32 as debugger as well, but for the postprocessing this is not needed.

2. Adjust all trace relevant settings like for a real target by running the start-up script you used for generating the trace data. For postprocessing an ETMv4 even further setups might be needed which normally the debugger would read out from the ETM module (ETM.COND, ETM.INSTPO, ETM.QE).

   If the start-up script is **not** available, then try this:

   - At best selecting the chip you are debugging (**SYStem.CPU** ...) is sufficient.
   - For trace data coming from a ARM CoreSight system, all commands describing the trace system on the chip are required (**SYStem.CONFIG** ...).
   - Further all settings for the trace sources done at recording time are needed (e.g. ETM. ...).

3. Load your target application (**Data.LOAD** ...).

4. Import the trace raw data (**LA.IMPORT**. ...).

5. Now you can use all trace display and analysis functions, e.g.

```
LA.List TP TPC TPINFO DEFault List.NoDummy.OFF ; with diagnostics
```

Example:

```
SYStem.CPU CortexA15

SYStem.CONFIG COREDEBUG.Base 0x82010000
SYStem.CONFIG.ETM.Base 0x8201c000
SYStem.CONFIG.FUNNEL.Base 0x80040000
SYStem.CONFIG.FUNNEL.ATBSource ETM 0
SYStem.CONFIG.ETB.Base 0x80010000

ETM.PortMode.Wrapped
ETM.TraceID 0x55

SYStem.Up

Data.LOAD.Elf myfile.elf

Trace.METHOD.LA
Trace.IMPORT.ETB mydata.bin
Trace.IMPORT.GUESSWRAP

Trace.List TP TPC TPINFO DEFault List.NoDummy.OFF
```

See also

- LA.IMPORT.CoreByteStream
- LA.IMPORT.ELA
- LA.IMPORT.GUESSWRAP
- LA.IMPORT.StartValid
- LA.IMPORT.STPByteStream
- LA.IMPORT.TraceFile
- LA.IMPORT.UltraSOC
- LA.IMPORT.WRAP
- LA.IMPORT.cycles
- LA.IMPORT.ETB
- LA.IMPORT.StartInvalid
- LA.IMPORT.STP
- LA.IMPORT.TARMAC
- LA.IMPORT.TracePort
- LA.IMPORT.VCD
- <trace>.EXPORT

LA.IMPORT.CoreByteStream — Import pure single core trace data

Format: **LA.IMPORT.CoreByteStream** *<file>*

Imports pure single core trace data (e.g. for unwrapped single core trace data or x86 IPT traces).

See also

- LA.IMPORT

LA.IMPORT.cycles — Import bus trace data

Format: **LA.IMPORT.cycles** *<file>*

Re-imports a file that has been exported with **<trace>.EXPORT.cycles**. This bus trace data comes from capturing the fetched instructions and data accesses done on an external bus to figure out the program behavior. It works only if no cache is used and if the bus accesses can be captured. This command can be used to import traces from external tools or simulators.

See also

- LA.IMPORT
- ▲ 'Release Information' in 'Legacy Release History'

LA.IMPORT.ELA — Import ELA trace data

Format: **LA.IMPORT.ELA** *<file>*

Imports a pure binary trace data file obtained from an ARM CoreSight Embedded Logic Analyzer (**ELA**).

See also

- LA.IMPORT

LA.IMPORT.ETB — Import on-chip trace data

Format: **LA.IMPORT.ETB** *<file>*

Imports a pure binary trace data file obtained from an on-chip trace buffer like ARM CoreSight ETB, ETF, ETR.

You additionally need to use **LA.IMPORT.WRAP** or **LA.IMPORT.GUESSWRAP** if the following conditions apply:

- The on-chip trace buffer was used as a circular ring buffer.
- The on-chip trace data was stored as is, it was not read out in the timely order starting from the write pointer position.

LA.Mode FlowTrace will automatically be set when using this command.

See also

- LA.IMPORT
- ▲ 'Release Information' in 'Legacy Release History'

Format: **LA.IMPORT.GUESSWRAP** [*<record_number>*]

Reformats external trace data loaded to TRACE32 in a timely order. The external trace data of a circular ring buffer is loaded to TRACE32 using **LA.IMPORT.ETB**. The command **LA.IMPORT.GUESSWRAP** scans the loaded trace data and guesses where the wrap pointer might have been.

Optionally, you can pass a record number where the search for the wrap pointer shall start. Without a parameter it starts from the beginning.

Use **LA.IMPORT.WRAP** if you know where the wrap pointer is.

See the figures in the introduction to **<trace>.IMPORT**.

See also

- LA.IMPORT
- LA.IMPORT.WRAP

LA.IMPORT.StartInvalid Set start of trace as invalid

Format: **LA.IMPORT.StartInvalid**

Reverts the setting done with **LA.IMPORT.StartValid**.

See also

- LA.IMPORT

Format: **LA.IMPORT.StartValid** [*<address1>* *<address2>* …]

This command informs the debugger that the start of the loaded trace is valid and that it should not wait for Sync packets.

<address1> *<address2>* …	The trace is set as valid starting from the given address. On SMP systems, the first address corresponds to the first core, the second to the second core...

See also

- LA.IMPORT

LA.IMPORT.STP Import STP recording from file (nibble)

Format: **LA.IMPORT.STP** *<file>*

Imports an STP trace from *<file>* to process it within TRACE32. One trace record is generated per nibble.

In order to unwrap the trace information for processing, TRACE32 needs to know the following information: STM base address and the STP protocol version.

If TRACE32 is aware of the chip characteristic, setting up the chip is sufficient.

Example:

```
SYStem.CPU OMAP4430APP1

LA.IMPORT.STP my_recording.stp

STMLA.List
```

Otherwise the following setup has to be done.

```
SYStem.CONFIG.STM.Base DAP:0xd4161000      ; any base address != 0x0 is
                                           ; fine

SYStem.CONFIG.STM.Mode STPv2               ; specify the STP protocol
                                           ; version

LA.IMPORT.STP my_recording.stp

STMLA.List
```

See also

- LA.IMPORT

LA.IMPORT.STPByteStream — Import STP recording from file (byte)

Format:	**LA.IMPORT.STPByteStream** *<file>*

Same as **LA.IMPORT.STP**, but one trace record is generated per byte.

See also

- LA.IMPORT

▲ 'Release Information' in 'Legacy Release History'

LA.IMPORT.TARMAC — Import TARMAC trace file

[build 165454 - DVD 02/2024]

Format:	**LA.IMPORT.TARMAC** *<file>* [**/***<option>*]
<option>:	**NoExec**

Imports a trace data file in Tarmac format.

NoExec	Allows the import process to ignore the execution state information (exec/notexec) from the file.

See also

- LA.IMPORT

Format: **LA.IMPORT.TraceFile** *<file>*

Re-imports trace data stored by **<trace>.SAVE** for re-processing. This is useful if processing was not possible when the trace recording was made. For example if you had no access to the target code at that moment.

Only the trace raw data will be extracted from the saved (*.ad) file.

LA.Mode FlowTrace will automatically be set when using this command.

See also

■ LA.IMPORT

▲ 'Release Information' in 'Legacy Release History'

LA.IMPORT.TracePort Import off-chip trace data

Format: **LA.IMPORT.TracePort** *<file>*

Imports a pure binary trace data file from an external trace port like an ARM CoreSight TPIU. Unlike on-chip trace data, off-chip trace data includes synchronization packages and depend on the port size of the trace port.

LA.Mode FlowTrace will automatically be set when using this command.

See also

■ LA.IMPORT

▲ 'Release Information' in 'Legacy Release History'

Format: **LA.IMPORT.UltraSOC** *<file>*

This command allows to load raw UltraSOC flow trace data.

See also

- LA.IMPORT

LA.IMPORT.VCD Import recorded signals in VCD file format

Format: **LA.IMPORT.VCD** *<file>*

Imports a VCD (Value Change Dump) file, which is an industrial standard format for waveforms (not for program trace). It is used for visualizing and analyzing the captured signals in the **<trace>.Timing** window.

See also

- LA.IMPORT

Format: **LA.IMPORT.WRAP** *<record_number>*

Reformats external trace data loaded to TRACE32 in a timely order. The external trace data of a circular ring buffer is loaded to TRACE32 using **LA.IMPORT.ETB**.

<record_number> You pass the *<record_number>* of the first trace record in time (wrap pointer). This is the write pointer location of a circular ring buffer the moment the data has been stored.

NOTE: On a CoreSight trace, the write pointer points to a 32-bit value. You need to multiply this value by 4 because each CoreSight trace record is 8 bit in size.

Use **LA.IMPORT.GUESSWRAP** if you do not know where the wrap pointer is.

See the figures in the introduction to **LA.IMPORT**.

See also

- LA.IMPORT
- LA.IMPORT.GUESSWRAP

Format:	**LA.Mode** [*<mode>*]
<mode>:	**Fifo** **Stack** **FlowTrace**

Selects the trace operation mode.

Fifo	If the trace is full, new records will overwrite older records. The trace records always the last cycles before the break.
Stack	If the trace is full recording will be stopped. The trace always records the first cycles after starting the trace.
FlowTrace	FlowTrace mode.

See also

- <trace>.Mode

LA.ACCESS Define access path to program code for trace decoding

See command **<trace>.ACCESS** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 131).

LA.Arm Arm the trace

See command **<trace>.Arm** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 134).

LA.AutoArm Arm automatically

See command **<trace>.AutoArm** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 135).

LA.AutoInit Automatic initialization

See command **<trace>.AutoInit** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 140).

LA.BookMark Set a bookmark in trace listing

See command **<trace>.BookMark** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 140).

LA.Chart Display trace contents graphically

See command **<trace>.Chart** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 144).

LA.CLOCK — Clock to calculate time out of cycle count information

See command **<trace>.CLOCK** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 191).

LA.ComPare — Compare trace contents

See command **<trace>.ComPare** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 192).

LA.ComPareCODE — Compare trace with memory

See command **<trace>.ComPareCODE** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 194).

LA.DISable — Disable the trace

See command **<trace>.DISable** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 197).

LA.DRAW — Plot trace data against time

See command **<trace>.DRAW** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 201).

LA.EXPORT — Export trace data for processing in other applications

See command **<trace>.EXPORT** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 212).

LA.FILE — Load a file into the file trace buffer

See command **<trace>.FILE** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 233).

LA.Find — Find specified entry in trace

See command **<trace>.Find** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 235).

LA.FindAll — Find all specified entries in trace

See command **<trace>.FindAll** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 237).

LA.FindChange — Search for changes in trace flow

See command **<trace>.FindChange** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 238).

LA.FLOWPROCESS — Process flowtrace

See command **<trace>.FLOWPROCESS** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 241).

LA.FLOWSTART — Restart flowtrace processing

See command **<trace>.FLOWSTART** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 241).

LA.GOTO — Move cursor to specified trace record

See command **<trace>.GOTO** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 244).

LA.Init — Initialize trace

See command **<trace>.Init** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 246).

LA.List — List trace contents

See command **<trace>.List** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 248).

LA.ListNesting — Analyze function nesting

See command **<trace>.ListNesting** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 263).

LA.ListVar — List variable recorded to trace

See command **<trace>.ListVar** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 266).

LA.LOAD — Load trace file for offline processing

See command **<trace>.LOAD** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 270).

LA.OFF — Switch off

See command **<trace>.OFF** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 278).

LA.PROfileChart — Profile charts

See command **<trace>.PROfileChart** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 283).

LA.PROfileSTATistic — Statistical analysis in a table versus time

See command **<trace>.PROfileSTATistic** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 322).

LA.PROTOcol — Protocol analysis

See command **<trace>.PROTOcol** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 339).

LA.PROTOcol.Chart Graphic display for user-defined protocol

See command **<trace>.PROTOcol.Chart** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 339).

LA.PROTOcol.Draw Graphic display for user-defined protocol

See command **<trace>.PROTOcol.Draw** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 341).

LA.PROTOcol.EXPORT Export trace buffer for user-defined protocol

See command **<trace>.PROTOcol.EXPORT** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 342).

LA.PROTOcol.Find Find in trace buffer for user-defined protocol

See command **<trace>.PROTOcol.Find** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 343).

LA.PROTOcol.list Display trace buffer for user-defined protocol

See command **<trace>.PROTOcol.list** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 344).

LA.PROTOcol.PROfileChart Profile chart for user-defined protocol

See command **<trace>.PROTOcol.PROfileChart** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 347).

LA.PROTOcol.PROfileSTATistic Profile chart for user-defined protocol

See command **<trace>.PROTOcol.PROfileSTATistic** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 348).

LA.PROTOcol.STATistic — Display statistics for user-defined protocol

See command **<trace>.PROTOcol.STATistic** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 350).

LA.REF — Set reference point for time measurement

See command **<trace>.REF** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 357).

LA.RESet — Reset command

See command **<trace>.RESet** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 357).

LA.SAVE — Save trace for postprocessing in TRACE32

See command **<trace>.SAVE** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 358).

LA.SelfArm — Automatic restart of trace recording

See command **<trace>.SelfArm** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 362).

LA.SIZE — Define buffer size

See command **<trace>.SIZE** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 373).

LA.SnapShot — Restart trace capturing once

See command **<trace>.SnapShot** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 373).

LA.state — Display trace configuration window

See command **<trace>.state** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 376).

LA.STATistic — Statistic analysis

See command **<trace>.STATistic** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 378).

LA.Timing — Waveform of trace buffer

See command **<trace>.Timing** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 499).

LA.TRACK — Set tracking record

See command **<trace>.TRACK** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 502).

LA.View — Display single record

See command **<trace>.View** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 504).

LA.ZERO — Align timestamps of trace and timing analyzers

See command **<trace>.ZERO** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 505).

List

Features program listing display

Program Listing Across Multiple Display Formats

The **List** command group displays a program listing:

- Source code and object code or source code only in the **List.auto** window dynamically determined by the **current debug mode**.
- Source code and object code in the **List.Mix** window
- Object code only in the **List.Asm** window
- Source code only in the **List.HII** window
- Java byte code in the **List.Java** window

Displayed Program Section

If no address is specified, the window automatically tracks the program counter (PC).

```
List.Auto
```

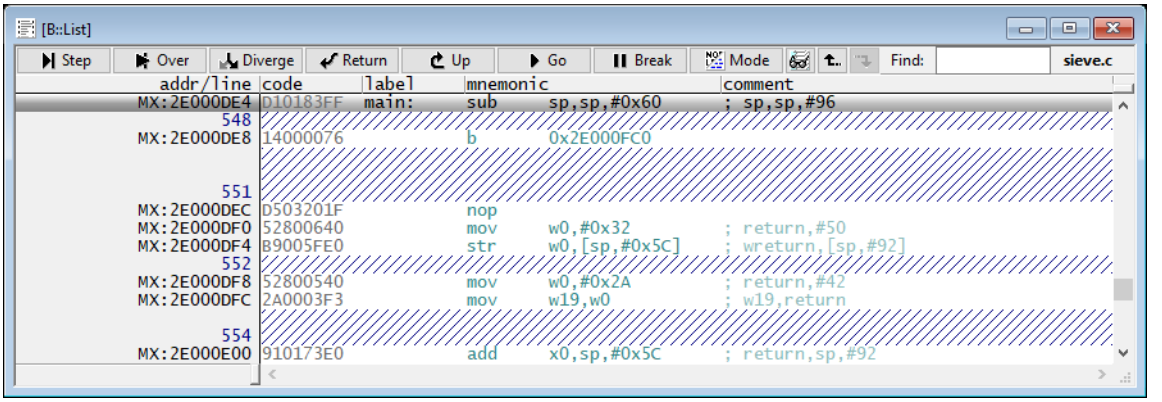
If an address or symbol is used, the program display begins at that address. The window remains permanently anchored to this address.

```
List.Auto func5
```

Disassembler Format

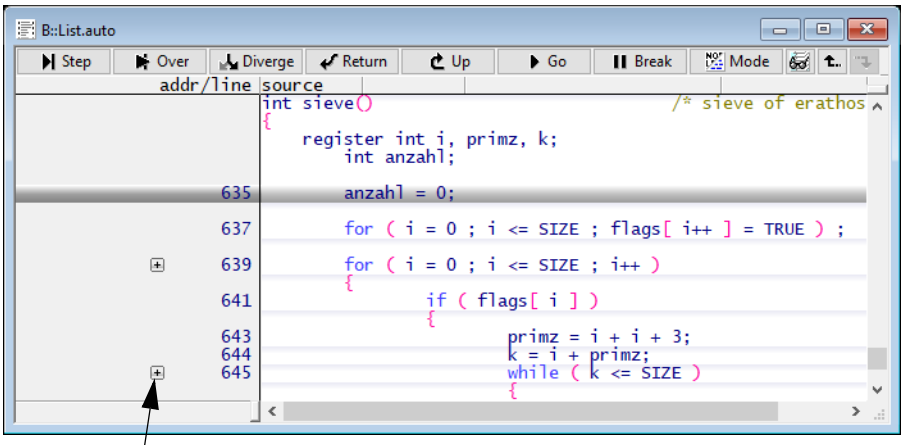
You can customize the disassembler format using the **SETUP.DIS** command.

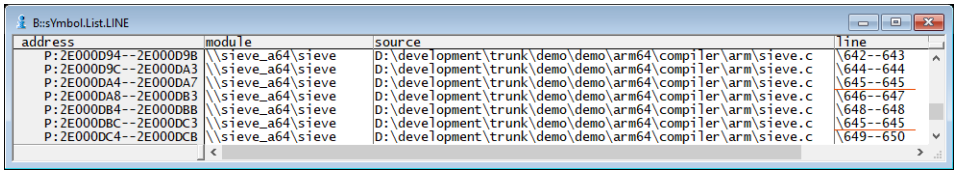
Hatched Lines



The debug information for the executable also includes the paths to the source files. Hatched lines indicate that TRACE32 cannot locate the source files at the specified paths. Use a command from the **sYmbol.SourcePATH** command group to correct the paths to the source files.

If the source listing is displayed in HLL mode, some code lines may be marked with a tree button. This tree button indicates that the compiler generated detached blocks of assembler code at more than one address location for this source code line. This is very common for **for-while** loops as well as for code compiled with a high optimization level.

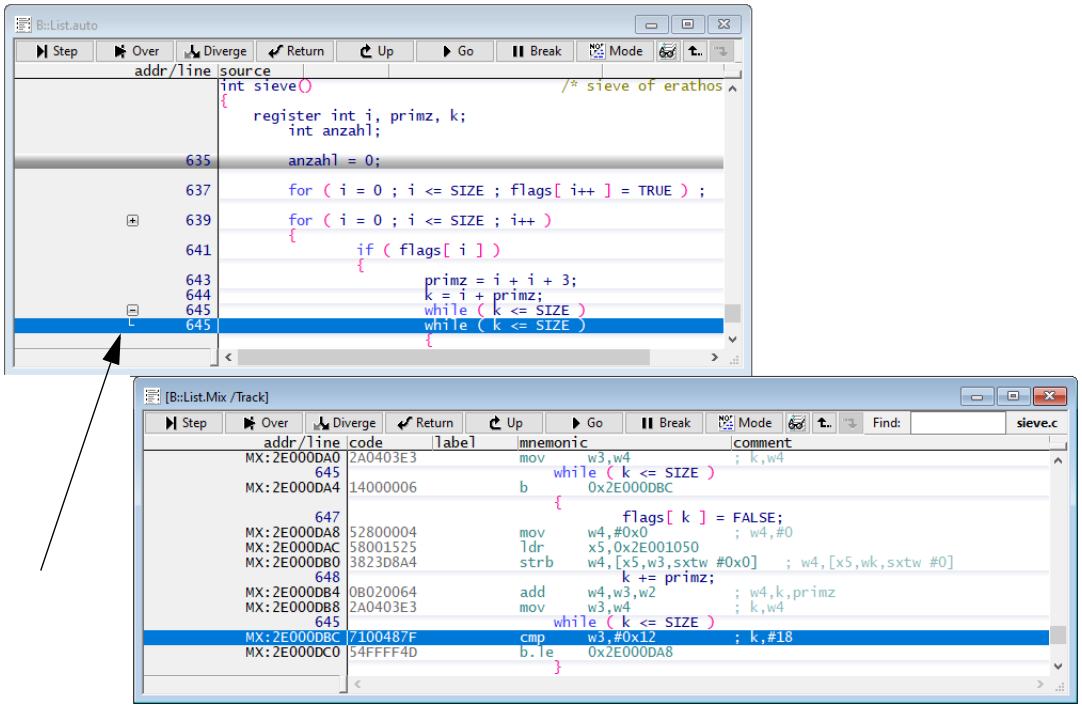




Two detached blocks of assembler code were generated for the source code at line 645.

If you want to inspect this in detail, the following commands might be helpful:

```
List.Hll                                ; display the source code in HLL
                                        ; mode (source order)

List.Mix /Track                         ; display the source code in Mixed
                                        ; mode (target line order)
```



The expansion of the tree button shows how many detached blocks of assembler code are generated for the HLL line.

Export of Program Listing

In addition, the program listing can be exported with **List.EXPORT**. The export is primarily utilized by the TRACE32 Coverage Report Utility to generate code coverage reports for the ObjectCode coverage metric. see **"Appendix A: TRACE32 Coverage Report Utility"** in Application Note for Trace-Based Code Coverage, page 117 (app_code_coverage.pdf).

General Options of the List Commands

Mark	The **Mark** option highlights individual lines, depending on the breakpoint type.
MarkPC	The **MarkPC** option highlights all HLL source lines belonging to the current PC. See example.
DIVERGE	This option is mainly intended for internal diagnostic purposes: It attaches tags to executed and not executed ASM and HLL lines. The tags are displayed in the scale area of the **List.*** windows. You can open the scale area by dragging the slider control to the right. • For information about the tags, see example. • See also **Step.Diverge**.

COVerage	Display source listing together with code coverage information. Two examples are provided below: • Object code coverage example. • MCDC coverage example.
CACHE	Displays cache hit information and marks currently cached code.
Track	Tracks the window to the reference position of other windows. The window tries first to track to the PROGRAM reference, and if this reference is not valid, it tracks to the DATA reference.
TOrder	List source lines in **target line order**. This is the default for assembly and mixed mode displays.
SOrder	List source lines in **source line order**. This is the default for source level displays.
ISTAT [*<parameter>*]	Display source listing together with the information provided by the instruction trace database **(ISTATistic.ListFunc**). Instructions that have not been executed are highlighted in yellow. • For a description of the *<parameters>*, see table below. • **ISTAT** used without parameter, see example. • **ISTAT** used with the **COVerage** parameter, see example.

Parameters for the ISTAT Option

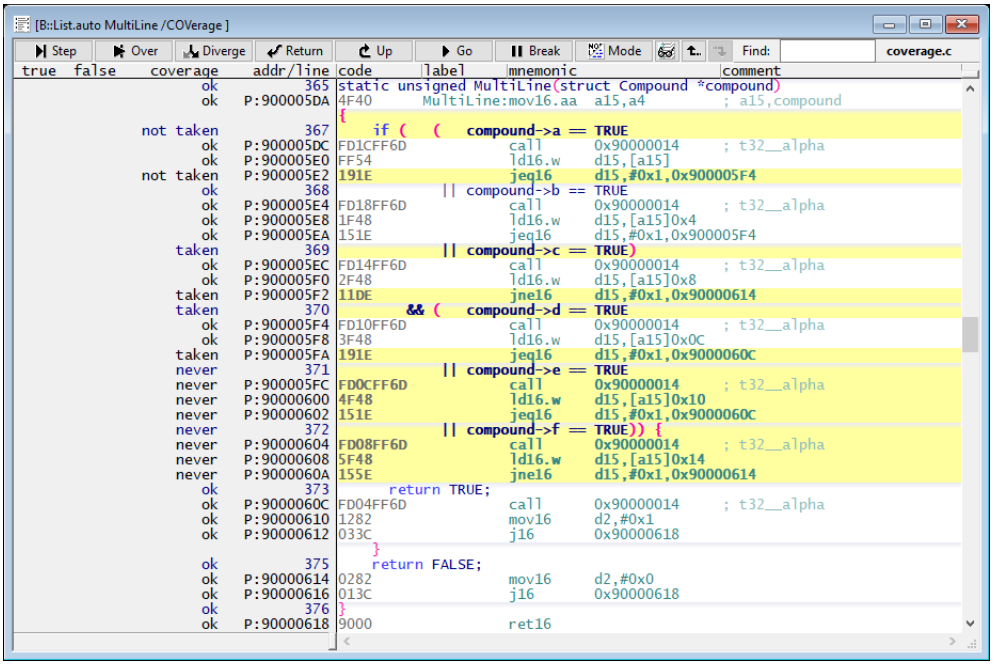
[Back]

DEFault	Display the default information provided by the ISTAT database.
ALL	Display all information provided by the ISTAT database.
CLOCK	Display the clock and CPI information provided by the ISTAT database.
TCLOCK	(only for special purposes)
SAMPLES	Display recorded samples, time and ratio.
COVerage	Display the code coverage information provided by the ISTAT database.

General Examples

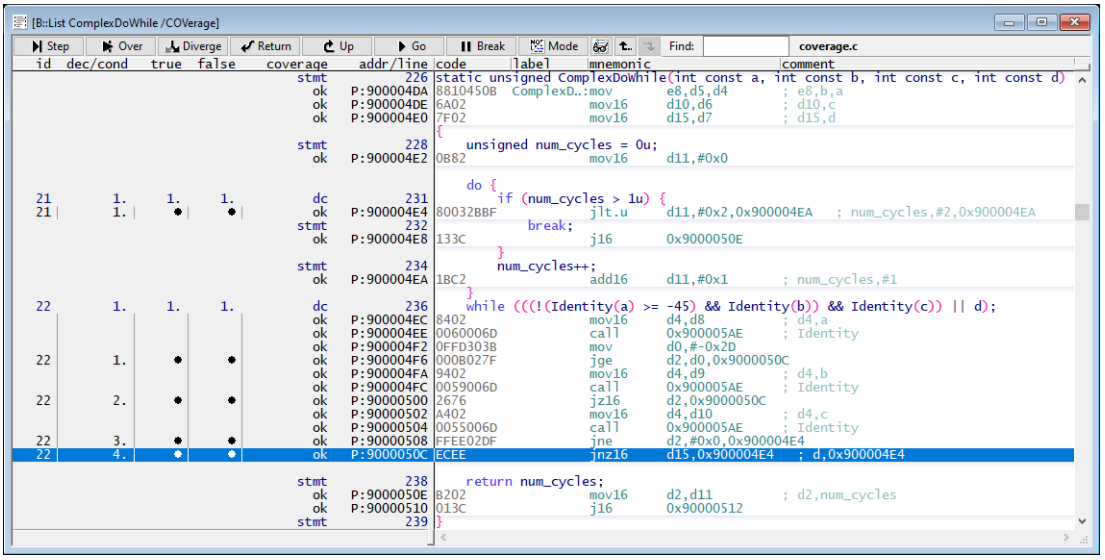
Example for the COVerage option - ObjectCode coverage

```
List.auto MultiLine /COVerage
```



Object code statements that are not tagged with 'ok' (indicating full coverage) are highlighted in bold with a yellow background.

```
List.auto ComplexDoWhile /COVerage
```

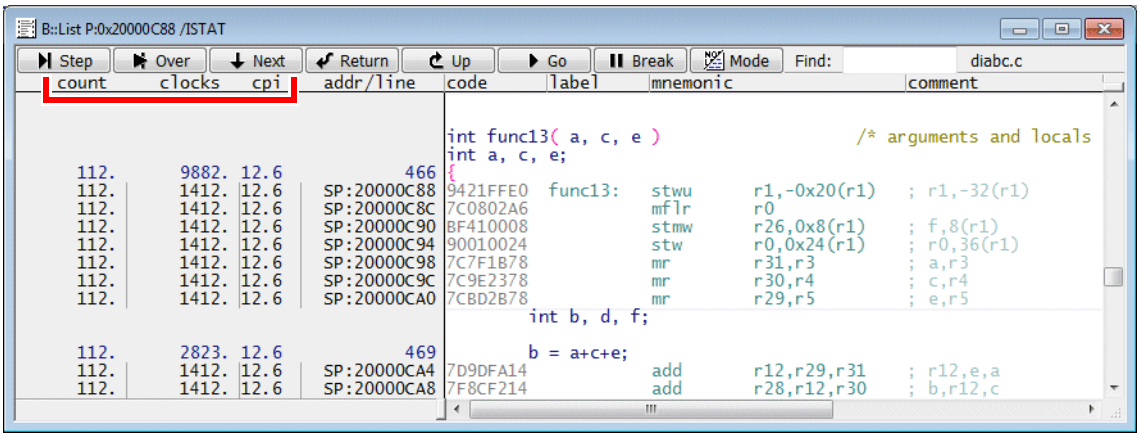


TRACE32 assigns an ID to each decision, with the atomic conditions that comprise the decision being numbered sequentially. Ideally, each condition is represented at the object code level by a conditional branch, as illustrated in the screenshot above.

Example for the ISTAT option - without <parameter>

[Back]

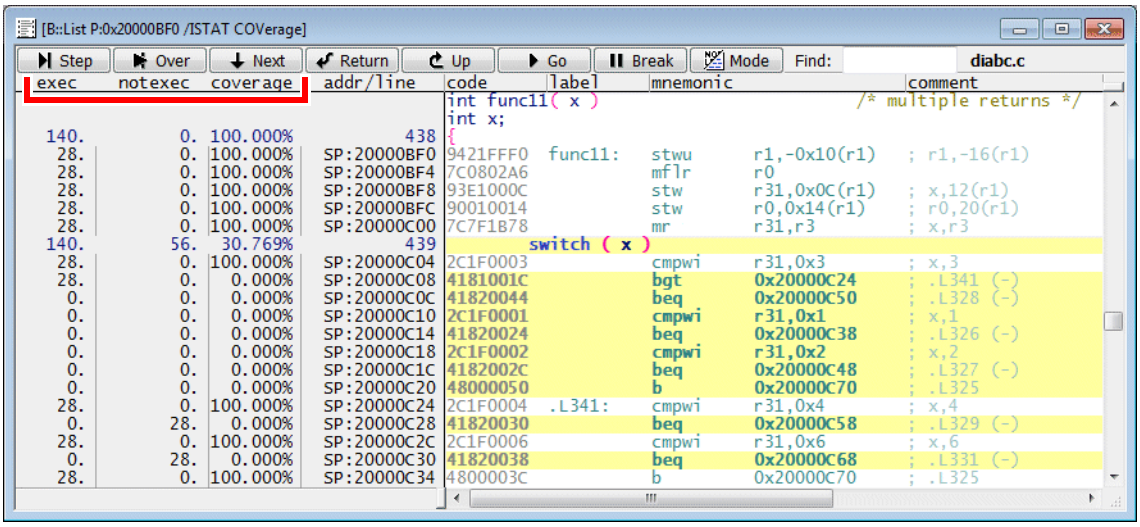
```
List.Asm func13 /ISTAT                  ; list instruction run-time
                                        ; statistic
```



count	Total number of instruction executions
clocks	Total number of clocks for the instruction
cpi	Average clocks per instruction

[Back]

```
List func11 /ISTAT COVerage                ; list instruction coverage
```



exec	**conditional instructions:** number of times the instruction was executed because the condition was true. **other instructions:** number of times the instruction was executed
notexec	**conditional instructions:** number of times the instruction wasn't executed because the condition was false.
coverage	Instruction coverage

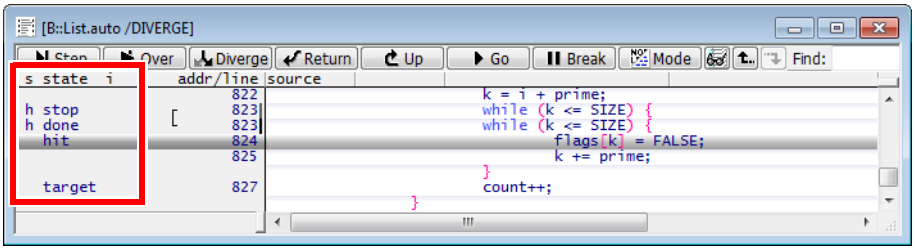
If exec or/and notexec is 0 for an instruction with condition, the instruction is bold-printed on a yellow background. All other instruction are bold-printed on a yellow background if they were not executed.

```
List                            ; display source listing around the
                                ; current PC

List /Mark Program              ; display source listing, bold print
                                ; all instructions / HLL lines on a
                                ; yellow background if a program
                                ; breakpoint is set

List /Mark                      ; remove bold printing on yellow
                                ; background
```

```
List /Track                     ; track the window to a reference, e.g.
                                ; analyzer

List Register(a0)               ; follow the register A0 of the CPU
```

Example for the DIVERGE option

[Back]

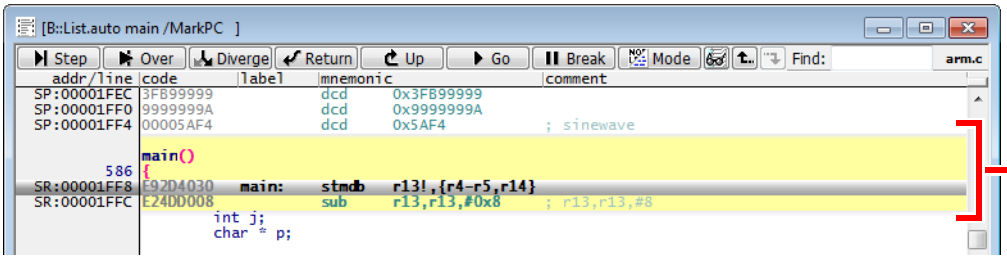


Tags in the columns s, state, and i	
h	Line in HLL mode (**Mode.Hll**).
a	Address in ASM mode (**Mode.Asm**).
stop	Reached by a breakpoint.
done	An executed line or address.
hit	A reached line or address that has not yet been executed, e.g. in a linear code sequence.
target	This line or address is a possible target of the next **Step.Diverge**. Once reached, **target** is replaced with **hit**.
i	Indirect branch instruction.

Example for the MarkPC option

[Back]

```
Register.Set PC main      ; set the Program Counter to the label main
List.auto main /MarkPC    ; highlight all instructions related to the
                          ; current HLL line
```



See also

- List.Asm
- List.auto
- List.EXPORT
- List.Hll
- List.Java
- List.Mix
- Data.dump
- Data.LOAD
- Go
- SETUP.DIS
- SETUP.LISTCLICK
- SETUP.sYmbol
- SETUP.TIMEOUT
- Step
- WinOverlay
- ADDRESS.isPHYSICAL()
- ADDRESS.OFFSET()
- ADDRESS.SEGMENT()
- ADDRESS.STRACCESS()
- ADDRESS.WIDTH()
- sYmbol.END()
- sYmbol.EXIT()

▲ 'Release Information' in 'Legacy Release History'

Format:	**List.auto** [*<address>* | *<range>*] [**/***<option>*]
<option>:	**Mark** *<break>* **Flag** *<flag>* (EF) **DIVERGE** **COVerage** **CACHE** **Track** **TOrder** | **SOrder** **ISTAT** [*<parameter>*]
<flag>:	**Read** | **Write** | **NoRead** | **NoWrite**
<break>:	**Program** | **Hll** | **Spot** | **Read** | **Write** | **Alpha** | **Beta** | **Charly** | **Delta** | **Echo**
<parameter>:	**DEFault** | **ALL** | **CLOCKS** | **TCLOCKS** | **SAMPLES** | **COVerage**

The display format of the **List.auto** window (assembler, mixed, or HLL) is dynamically determined by the current debug mode. You can change the debug mode by clicking the **Mode** button in the local toolbar of any List window or by using the **Mode** command.

<option>	For a description of the options, see **"General Options of the List Commands"**, page 30.

See also

- List
- List.Asm
- List.EXPORT
- List.Hll
- List.Java
- List.Mix
- SETUP.DIS
- SETUP.LISTCLICK
- SETUP.sYmbol
- SETUP.TIMEOUT
- ADDRESS.isPHYSICAL()
- ADDRESS.OFFSET()
- ADDRESS.SEGMENT()
- ADDRESS.STRACCESS()
- ADDRESS.WIDTH()
- sYmbol.END()
- sYmbol.EXIT()

▲ 'Release Information' in 'Legacy Release History'

Format:	**List.Asm** [*<address>*] [**/***<option>*]
<option>:	**Mark** *<break>* **Flag** *<flag>* (EF) **COVerage** **CACHE** **Track** **MarkPC** **TOrder** | **SOrder** **IgnoreSymbols**

Displays the program in disassembled format.

<option>	For a description of the options, see **"General Options of the List Commands"**, page 30.
IgnoreSymbols	Let the disassembler ignore any symbols for deciding at which byte of the machine code the disassembling should start. This option does normally only make sense on architectures with different instruction lengths.

See also

- List
- List.auto
- List.Hll
- List.Java
- List.Mix
- Mode
- DEBUGMODE()

List.EXPORT — Export a listing to an XML file

Using the **List.EXPORT** command group, you can export a source or disassembler listing or both listings to an XML file.

In addition, TRACE32 provides an XSL transformation template for formatting the XML file. The formatting is automatically applied to the XML file when it is opened in an external browser window. Prerequisite: The XSL file is placed in the same folder as the XML file.

For demo scripts, see **List.EXPORT.Mix**.

See also

- List.EXPORT.Asm
- List.EXPORT.auto
- List.EXPORT.HII
- List.EXPORT.Mix
- List
- List.auto
- COVerage.EXPORT
- ISTATistic.EXPORT

List.EXPORT.Asm — Export disassembler listing

Format:	**List.EXPORT.Asm** *<file>* *<range>* [*/<option>*]
<option>:	**COVerage** | **ISTATistic** **Append** | **NoData**

Creates an XML file containing the disassembler listing. For an example and a description of the options, see **List.EXPORT.Mix**.

See also

- List.EXPORT

List.EXPORT.auto — Export source and disassembler listing

Format:	**List.EXPORT.auto** *<file>* *<range>* [*/<option>*]
<option>:	**COVerage** | **ISTATistic** | **TOrder** | **SOrder** **Append** | **NoData** | **COVerableItems**

Creates an XML file containing the source listing and the disassembler listing - same as **List.EXPORT.Mix**. For an example and a description of the options, see **List.EXPORT.Mix**.

See also

- List.EXPORT

Format:	**List.EXPORT.HII** *<file> <range>* [**/***<option>*] **Data.ListEXPORTHII** (as an alternative)
<option>:	**COVerage** | **ISTATistic** | **Append** | **NoData**

Creates an XML file containing just the source listing. For an example and a description of the options, see **List.EXPORT.Mix**.

See also

- List.EXPORT

List.EXPORT.Mix Export source and disassembler listing

[Example]

Format:	**List.EXPORT.Mix** *<file> <range>* [**/***<option>*]
<option>:	**COVerage** | **ISTATistic** | **SOrder** | **TOrder** | **Append** | **NoData** | **COVerableItems**

Creates an XML file containing the source listing and the disassembler listing.

<file>	Name of the XML file that stores a listing of the source and disassembler code. The file extension *.xml can be omitted.
<range>	Address filter for exporting the specified range.
Append	Appends the listing to an existing XML file - without overwriting the current file contents.
COVerage	Listing additionally contains code coverage information.
COVerableItems	Exports only coverage results for measurable source lines.
ISTATistic	Listing additionally contains information provided by the ISTATistics module.
NoData	Excludes data-only sections from the XML output.
SOrder	Export the source code lines in **source order**.
TOrder	Export the source code lines in **target order** (default).

Example 1: The prerequisites for the following example are that the debug symbols have already been loaded, the address bookmarks have been created, and trace data has been recorded.

```
COVerage.ADD                  ;update the coverage database
Data.List /COVerage /Track    ;display source listing
COVerage.ListFunc             ;display coverage for HLL functions

;export all bookmarks
BookMark.EXPORT "~~/list.xml"

;export the source listing of the functions "main" and "sieve"
List.EXPORT.Mix "~~/list.xml"  main   /COVerage  /Append
List.EXPORT.Mix "~~/list.xml"  sieve  /COVerage  /Append

;for demo purposes: let's open the unformatted result in TRACE32
EDIT "~~/list.xml"

;place the transformation template in the same folder as the XML file
COPY "~~/demo/coverage/single_file_report/t32transform.xsl" \
     "~~/t32transform.xsl"

;you can now open the formatted result in an external browser window
OS.Command start iexplore.exe "file:///C:/t32/list.xml"
```

The tildes ~~ expand to your TRACE32 system directory, by default c:\t32.

Example 2: A more complex demo script is included in your TRACE32 installation. To access the script, run this command:

```
B::CD.PSTEP ~~/demo/coverage/example.cmm
```

See also

- List.EXPORT

Format:	**List.HII** [*<address>*] [**/***<option>*]
<option>:	**Mark** *<break>* **Flag** *<flag>* (EF) **COVerage** **CACHE** **Track** **MarkPC** **TOrder** | **SOrder**

Displays the program in source format. If the starting address in not an HLL (High Level Language) line, assembler code is displayed to the next source code line found in the code segment.

<option>	For a description of the options, see **"General Options of the List Commands"**, page 30.

See also

- List
- List.Asm
- List.auto
- List.Java
- List.Mix
- Mode
- DEBUGMODE()

List.Java Display Java byte code

[Example]

Format:	**List.Java** [*<address>*] [**/***<option>*]
<option>:	**Mark** *<break>* **Flag** *<flag>*(EF) **COVerage** **CACHE** **Track** **TOrder** | **SOrder**

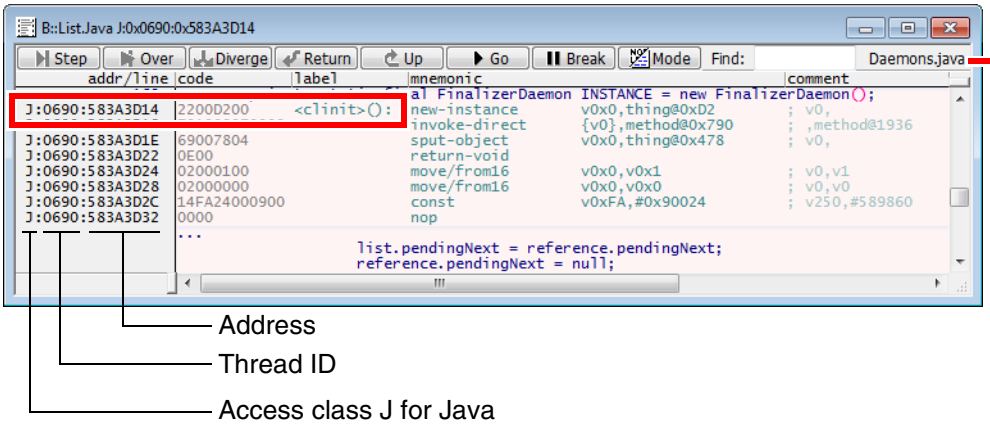
Displays the program in Java byte code format. The functionality is the same as the **List.auto** command.

This command is NOT required when an address range is specified as Java byte code area with the Java byte code debugger or when the processor implements a special byte code mode (e.g. ARM Jazelle).

<option>	For a description of the options, see **"General Options of the List Commands"**, page 30.

Android is used in this example:

```
;list all running tasks by magic number, task name, task ID, etc.
TASK.List.tasks

;change view to a task by specifying the magic number, task name or ID
Frame.TASK 0xEFD29700

;display a source listing at this address for the task specified above
List.Java J:0x0690:0x583A3D14

;alternatively, use the label to display the same source listing.
List.Java `<clinit>()`          ;enclose the label in backticks `...`
```



See also

- List
- List.Asm
- List.auto
- List.Hll
- List.Mix

Format:	**List.Mix** [*<address>*] [*/<option>*]
<option>:	**Mark** *<break>* **Flag** *<flag>* (EF) **COVerage** **CACHE** **Track** **MarkPC** **TOrder** | **SOrder**

The code is displayed in HLL (High Level Language) and additionally disassembled from the memory.

<option>	For a description of the options, see **"General Options of the List Commands"**, page 30.

See also

- List
- List.Asm
- List.auto
- List.Hll
- List.Java
- Mode
- DEBUGMODE()

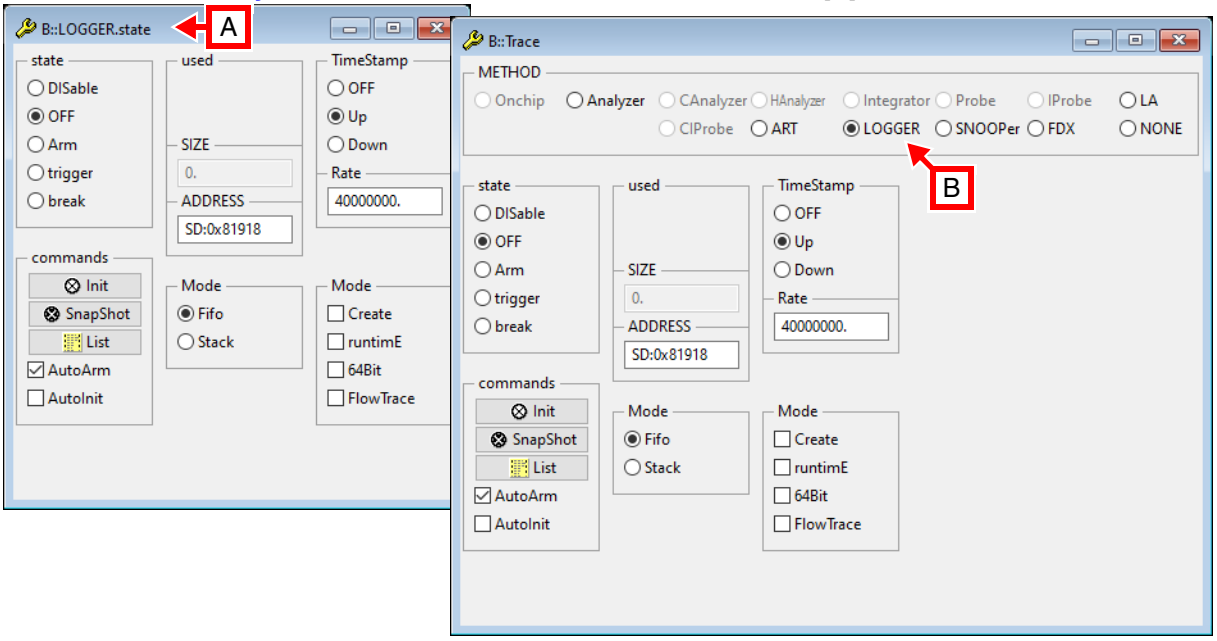
LOGGER Trace method LOGGER, recording and analysis commands

LOGGER is a software trace method where the target application writes the required trace information to the a reserved buffer on the target RAM. TRACE32 loads then the trace information from the target RAM for display and processing.

Please refer to the **"Application Note for the LOGGER Trace"** (app_logger.pdf) for more information.

For selecting and configuring the trace method LOGGER, use the TRACE32 command line or a PRACTICE script (*.cmm) or the **LOGGER.state** window [**A**].

Alternatively, use the **Trace.state** window: click the option **LOGGER** or execute the command **Trace.METHOD Analyzer** in order to select the trace method **LOGGER** [**B**].



The chapter **"LOGGER-specific Trace Commands"**, page 47 describes the LOGGER-specific configuration commands. While the chapter **"Generic LOGGER Trace Commands"**, page 49 lists the LOGGER trace analysis and display commands, which are shared with other TRACE32 trace methods.

See also

- FDX
- Trace.METHOD

LOGGER.ADDRESS — Software trace address

Format: **LOGGER.ADDRESS** [*<address>*]

Defines the address of the logger trace control block in target memory.

LOGGER.Mode — Set LOGGER operation mode

Format: **LOGGER.Mode** [*<mode>*]

<mode>: **Fifo** | **Stack**

Create
runtimE
64bit

FlowTrace

Selects the trace operation mode.

Fifo	If the trace is full, new records will overwrite older records. The trace records always the last cycles before the break.
Stack	If the trace is full recording will be stopped. The trace always records the first cycles after starting the trace.
Create	Create LOGGER software trace control block by debugger.
runtimE	Dualport access.
64Bit	LOGGER mode for 64-bit traces.
FlowTrace	Special mode where the LOGGEr is used to sample the program flow. The LOGGER trace listing reconstructs the program flow based on the sampled information. Only supported for PowerPC and SH4.

See also

- <trace>.Mode

Format: **LOGGER.TimeStamp OFF** | **Up** | **Down** | **Rate** *<rate>*

Configure timestamps for the **LOGGER** trace. The LOGGER trace record format includes a timestamp field for up to 48 bit timestamps. The direction and rate information passed by this command is required to convert the timestamp into the time in seconds.

OFF (default)	Disable timestamps. Use this setting if the LOGGER target code does not store timestamps in the LOGGER trace records. When this setting is used, the x-direction in chart views is the record number axis instead of the time axis.
Up	Enable timestamp counter, counting upwards. Use this setting if the LOGGER target code stores timestamps in the LOGGER trace records and if the timestamp increments with each timer tick.
Down	Enable timestamp counter, counting downwards. Use this setting if the LOGGER target code stores timestamps in the LOGGER trace records and if the timestamp decrements with each timer tick.
Rate *<rate>*	Frequency of the timestamp in ticks per second.
AllCycles [**ON** | **OFF**] SH only	Set timestamp generation frequency. • **OFF** (default): Generate a single timestamp for 6 trace cycles. • **ON**: Generate dedicated timestamps for all trace cycles.

Example: The timestamp used by the LOGGER target code increments at a rate of 16 million per second (16 MHz):

```
LOGGER.TimeStamp Up
LOGGER.TimeStamp Rate 16000000.
```

LOGGER.ACCESS Define access path to program code for trace decoding

See command **<trace>.ACCESS** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 131).

LOGGER.Arm Arm the trace

See command **<trace>.Arm** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 134).

LOGGER.AutoArm Arm automatically

See command **<trace>.AutoArm** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 135).

LOGGER.AutoInit Automatic initialization

See command **<trace>.AutoInit** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 140).

LOGGER.BookMark Set a bookmark in trace listing

See command **<trace>.BookMark** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 140).

LOGGER.BookMarkToggle Toggles a single trace bookmark

See command **<trace>.BookMarkToggle** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 143).

LOGGER.Chart — Display trace contents graphically

See command **<trace>.Chart** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 144).

LOGGER.ComPare — Compare trace contents

See command **<trace>.ComPare** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 192).

LOGGER.DISable — Disable the trace

See command **<trace>.DISable** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 197).

LOGGER.DRAW — Plot trace data against time

See command **<trace>.DRAW** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 201).

LOGGER.EXPORT — Export trace data for processing in other applications

See command **<trace>.EXPORT** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 212).

LOGGER.FILE — Load a file into the file trace buffer

See command **<trace>.FILE** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 233).

LOGGER.Find — Find specified entry in trace

See command **<trace>.Find** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 235).

LOGGER.FindAll — Find all specified entries in trace

See command **<trace>.FindAll** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 237).

LOGGER.FindChange — Search for changes in trace flow

See command **<trace>.FindChange** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 238).

LOGGER.FLOWPROCESS — Process flowtrace

See command **<trace>.FLOWPROCESS** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 241).

LOGGER.FLOWSTART — Restart flowtrace processing

See command **<trace>.FLOWSTART** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 241).

LOGGER.GOTO — Move cursor to specified trace record

See command **<trace>.GOTO** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 244).

LOGGER.Init — Initialize trace

See command **<trace>.Init** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 246).

LOGGER.List — List trace contents

See command **<trace>.List** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 248).

LOGGER.ListNesting — Analyze function nesting

See command **<trace>.ListNesting** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 263).

LOGGER.ListVar — List variable recorded to trace

See command **<trace>.ListVar** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 266).

LOGGER.LOAD — Load trace file for offline processing

See command **<trace>.LOAD** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 270).

LOGGER.OFF — Switch off

See command **<trace>.OFF** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 278).

LOGGER.PROfileChart — Profile charts

See command **<trace>.PROfileChart** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 283).

LOGGER.PROfileSTATistic — Statistical analysis in a table versus time

See command **<trace>.PROfileSTATistic** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 322).

LOGGER.PROTOcol — Protocol analysis

See command **<trace>.PROTOcol** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 339).

LOGGER.PROTOcol.Chart — Graphic display for user-defined protocol

See command **<trace>.PROTOcol.Chart** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 339).

LOGGER.PROTOcol.Draw Graphic display for user-defined protocol

See command **<trace>.PROTOcol.Draw** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 341).

LOGGER.PROTOcol.EXPORT Export trace buffer for user-defined protocol

See command **<trace>.PROTOcol.EXPORT** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 342).

LOGGER.PROTOcol.Find Find in trace buffer for user-defined protocol

See command **<trace>.PROTOcol.Find** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 343).

LOGGER.PROTOcol.list Display trace buffer for user-defined protocol

See command **<trace>.PROTOcol.list** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 344).

LOGGER.PROTOcol.PROfileChart Profile chart for user-defined protocol

See command **<trace>.PROTOcol.PROfileChart** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 347).

LOGGER.PROTOcol.PROfileSTATistic Profile chart for user-defined protocol

See command **<trace>.PROTOcol.PROfileSTATistic** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 348).

LOGGER.PROTOcol.STATistic Display statistics for user-defined protocol

See command **<trace>.PROTOcol.STATistic** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 350).

LOGGER.REF — Set reference point for time measurement

See command **<trace>.REF** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 357).

LOGGER.RESet — Reset command

See command **<trace>.RESet** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 357).

LOGGER.SAVE — Save trace for postprocessing in TRACE32

See command **<trace>.SAVE** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 358).

LOGGER.SelfArm — Automatic restart of trace recording

See command **<trace>.SelfArm** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 362).

LOGGER.SIZE — Define buffer size

See command **<trace>.SIZE** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 373).

LOGGER.SnapShot — Restart trace capturing once

See command **<trace>.SnapShot** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 373).

LOGGER.state — Display trace configuration window

See command **<trace>.state** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 376).

LOGGER.STATistic — Statistic analysis

See command **<trace>.STATistic** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 378).

LOGGER.Timing — Waveform of trace buffer

See command **<trace>.Timing** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 499).

LOGGER.TRACK — Set tracking record

See command **<trace>.TRACK** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 502).

LOGGER.View — Display single record

See command **<trace>.View** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 504).

LOGGER.ZERO — Align timestamps of trace and timing analyzers

See command **<trace>.ZERO** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 505).

LUA Support for the Lua script language

The TRACE32 Lua API allows the user to load and execute Lua scripts directly in the debugger. This feature can be used to accelerate execution of certain debug commands by avoiding the interaction between the TRACE32 host software and the debug driver. A Lua interpreter is built into the debug box, supporting the complete Lua language. Please refer to the official website of Lua www.lua.org for documentation.

In addition to the standard language elements, Lauterbach extended Lua with a set of TRACE32 specific libraries. This allows you to, for example, use the JTAG shift functions directly in the Lua script. For a description of the TRACE32 specific libraries, refer to **"TRACE32 Lua Library"** (lua_library.pdf).

The TRACE32 host software maintains an input buffer with size 0x1000 bytes to store input parameters for the Lua script. While sending the command to execute a certain Lua script, TRACE32 also packs and sends the input buffer to the debugger. Use **LUA.Data.SET** to manipulate the input buffer and **LUA.Data.ShowInput** to view the current content in the input buffer. Loading the input buffer from a binary file is also supported. From within the Lua scripts, the data in the input buffer can be retrieved using functions from the **"TRACE32 Lua Library"** (lua_library.pdf).

When executing a Lua script, the TRACE32 host software blocks until it returns. Inside the Lua script, functions from the **"TRACE32 Lua Library"** (lua_library.pdf) can be used to store output data into an output buffer (again 0x1000 bytes). The output buffer will be automatically sent back to the TRACE32. Use **LUA.Data.ShowOutput** to view the output data. It is also supported to save the output buffer into a binary file.

The Lua API can be used both as TRACE32 commands and through the remote C API. The **LUA** command group is described in the following sections and the C API is introduced in chapter **Remote Lua API**.

LUA.Data.Loadinput Load content from a file into the input buffer

Format: **LUA.Data.Loadinput** *<file>* [*/<load_option>*]

Load content from a binary file into the input buffer. Use **LUA.Data.ShowInput** to check the current content of the input buffer.

Offset *<offset>* — Starting position in the binary file to be loaded into the input buffer

Length *<length>* — Number of bytes to be loaded into the input buffer

Format:	**LUA.Data.Saveoutput** *<file>* [*/<save_option>*]

Save content of the output buffer into a file. Use **LUA.Data.ShowOutput** to check the current content of the output buffer.

Append Append data to the end of the output file.

LUA.Data.SET Modify the Lua input buffer

Format:	**LUA.Data.SET** *<index>* **%***<format>* *<value>*
<format>:	**Byte** | **Word** | **Long** | **Quad** | **TByte** | **HByte** | **PByte** | **SByte** **BE** | **LE**

Writes byte-wise data to the input buffer. Use **LUA.Data.ShowInput** to view the current content of the input buffer. The input buffer will be sent to the debugger upon executing a **LUA.Program.RUN** command.

<index>	The byte position of the input buffer to be written.
Byte, **Word**, …	Data size. • **Byte** (8-bit accesses) **Word** (16-bit accesses) • **TByte** (24-bit accesses) **Long** (32-bit accesses) • **PByte** (40-bit accesses) **HByte** (48-bit accesses) • **SByte** (56-bit accesses) **Quad** (64-bit accesses) •
BE, **LE**	Define byte endianness: big endian or little endian.
<value>	The value to be written to the buffer.

LUA.Data.ShowInput — Show current content of the input buffer

Format: **LUA.Data.ShowInput**

Displays the current content of the input buffer in the **AREA** window.

LUA.Data.ShowOutput — Show current content of the output buffer

Format: **LUA.Data.ShowOutput**

Displays the current content of the output buffer in the **AREA** window.The output buffer contains the return values from the Lua script. Inside the Lua script, functions from the **"TRACE32 Lua Library"** (lua_library.pdf) can be used to write to the output buffer.

LUA.Program.List — List the current Lua scripts

Format: **LUA.Program.List**

Lists the Lua scripts that have been loaded into the debugger. The output is redirected to the **AREA** window and has the following format: *<id>* : *<label>, <attribute>, <file>.* See **example**.

ID	An auto-generated sequential index of the current Lua scripts. Note that the ID of a certain script may change after loading/unloading another script.
label	A unique string to identify a Lua script.
attribute	EXE: an executable script. LIB: a library script. See **LUA.Program.LOAD** for more details.
filename	Path to the Lua file.

LUA.Program.LOAD — Load a Lua script to debugger

Format:	**LUA.Program.LOAD** *<file>* [*<label>*] [**/***<options>*]
<option>:	**Program** | **Library**

Loads the Lua script to the debugger without executing it. The script is uniquely identified by its label. See **example** here.

<file>	Path and file name of the Lua script to be loaded.
<label>	A unique string to identify the Lua script. If not specified, a default label will be generated.
Program	The Lua script is loaded as an executable script. This is the default option.
Library	The Lua script is loaded as a library script. The Lua functions in a library script will be made available to all other Lua scripts. Although marked as a library, the script itself can still be executed normally. However, we recommend to develop separate Lua scripts for libraries and executables.

LUA.Program.RESet — Reset the Lua context

Format:	**LUA.Program.RESet**

Resets the Lua context, unloads all scripts from the debugger and clears all input and output buffers.

LUA.Program.RUN — Execute a Lua script

Format:	**LUA.Program.RUN** *<id>* | *<label>*

Executes the Lua script with given index or label. Use **LUA.Program.List** to check IDs and labels of the Lua scripts that are currently available in the debugger.

See **example**.

Format:	**LUA.Program.UNLOAD** *<id>* | *<label>*

Removes a Lua script from the debugger. Use **LUA.Program.List** to see the scripts currently loaded.

Example

```
; load the Lua script lib.lua as a library
LUA.Program.LOAD c:\lua\lib.lua "mylib" /Library
; load the Lua script jtag.lua as an executable
; use default option "/Program" and default label
LUA.Program.LOAD c:\lua\jtag.lua

; see the current list of Lua scripts
LUA.Program.List
; you should see the following:
; 0 : mylib, LIB, C:\lua\lib.lua
; 1 : jtag.lua, EXE, C:\lua\jtag.lua

; set input parameter
LUA.Data.SET 0x0 %l 0x12345678
LUA.Data.ShowInput

; execute the Lua script jtag.lua using its index
LUA.Program.RUN 1
; execute the Lua script jtag.lua using its label
LUA.Program.RUN "jtag.lua"

; view the output buffer
LUA.Data.ShowOutput

; remove a Lua script
LUA.Program.UNLOAD 0
; Note that now the Lua script with index 0 (lib.lua) is removed
; and the indexing has changed
LUA.Program.List
; now you should see the following
; 0 : jtag.lua, EXE, C:\lua\jtag.lua
; the Lua script jtag.lua now has the index 0
LUA.Program.UNLOAD "jtag.lua"

; clear the context
LUA.Program.RESet
```