LAUTERBACH
DEVELOPMENT TOOLS

General Commands Reference Guide F

MANUAL

General Commands Reference Guide F

General Commands Reference Guide F

Version 06-Jun-2024

History

22-Mar-2024	New commands FLASHFILE.LOAD.JSON, FLASHFILE.LOAD.SPARSE, and FLASHFILE.MMC.GETHealth.
26-Jun-2023	New option /**SpotLight** for the command FLASHFILE.DUMP.
17-Mar-2023	New command FLASHFILE.ReProgram.
28-Jul-2022	New command FLASHFILE.Delete.
12-Apr-2022	Added parameters to the command FLASH.List.

FDX

Trace method FDX

The Fast Data Exchange (FDX) enables transferring universal data between the target and the host. The protocol implementation on target side is included in the target application. The source code (C) is provided by LAUTERBACH. On the host side the transmitted data can be processed by a user application communicating with the TRACE32 Application Interface or through Named Pipes. In a non-interactive mode TRACE32 can also read or write normal files as a general data source and sink.

A special application of the FDX is the FDX software trace. Through the trace application interface the target application can send trace information to the host. TRACE32 is capable to interpret the FDX data stream and handle it as ordinary trace information device. The trace method FDX is mainly used when no trace extension is available.

The basic packet transport method differs dependent on the target. TRACE32 supports memory mapped buffered transfer through dual port memory access or normal access at breakpoints or spot breakpoints. Some target devices support a Debug Communication Channel (DCC), which can be used to transfer FDX data in real-time.

For more information, please refer to **"Application Note for FDX"** (app_fdx.pdf).

See also

- FDX.ADDRESS
- FDX.CLEAR
- FDX.CLOSE
- FDX.DISableChannel
- FDX.ENableChannel
- FDX.InChannel
- FDX.METHOD
- FDX.Mode
- FDX.Out
- FDX.OutChannel
- FDX.PipeREAD
- FDX.PipeWRITE
- FDX.Rate
- FDX.READ
- FDX.TImestamp
- FDX.TraceChannel
- FDX.WRITE
- LOGGER
- Trace.METHOD

▲ 'Using ARTI Hooks' in 'Application Note Profiling on AUTOSAR CP with ARTI'
▲ 'General Function' in 'Application Note for FDX'
▲ 'FDX-specific Command' in 'General Commands Reference Guide F'
▲ 'Generic FDX Trace Commands' in 'General Commands Reference Guide F'
▲ 'Release Information' in 'Legacy Release History'

FDX.ADDRESS — Specify memory space for FDX traces

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

Specifies memory space for FDX traces.

See also

■ FDX

▲ 'General Data Transfer' in 'Application Note for FDX'

FDX.CLEAR — Clear FDX communication buffers

Format: **FDX.CLEAR** [*<address>*]

Clears the communication buffers of an FDX channel. All buffer contents are lost. Without arguments all FDX channels will be cleared.

See also

■ <trace>.state ■ FDX

▲ 'General Data Transfer' in 'Application Note for FDX'

FDX.CLOSE — Close FDX files

Format: **FDX.CLOSE** [*<address>*]

Closes the file related to the given FDX channel. Without arguments all files used by FDX are closed.

See also

■ <trace>.state ■ FDX

▲ 'General Data Transfer' in 'Application Note for FDX'

FDX.DISableChannel — Disable FDX communication

Format: **FDX.DISableChannel** [*<address>*]

Disables an FDX communication channel. Without parameters all channels are disabled. Disabling keeps the buffer contents of FDX. Communication can be re-enabled with **FDX.ENableChannel**.

See also

■ <trace>.state ■ FDX

▲ 'General Data Transfer' in 'Application Note for FDX'

FDX.ENableChannel — Enable FDX communication

Format: *<trace>*.**ENableChannel** [*<address>*]

Enables the data transfer over a FDX channel. Without parameters all existing FDX channels are enabled.

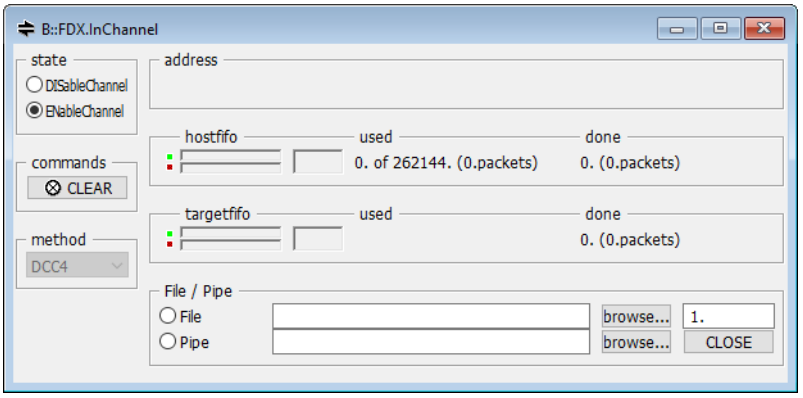
See also

■ <trace>.state ■ FDX

▲ 'General Data Transfer' in 'Application Note for FDX'

Format: **FDX.InChannel** [*<address>*]

Show the state of the input channel. An address parameter has to be specified if FDX doesn't use DCC (see **FDX.METHOD**).



See also

■ <trace>.state ■ FDX

▲ 'General Data Transfer' in 'Application Note for FDX'

FDX.METHOD Select communication channel

Format: **FDX.METHOD** [*<method>*]

<method>: **BufferE** | **BufferC** | **BufferS** | **DCC3** | **DCC** | **DCC4D** | **DCC7** | **DCC8**

If a *<method>* is specified, the command defines the low-level communication channel. Without *<method>*, the command displays the currently selected *<method>* in the TRACE32 message line.

BufferE and **DCC** are methods where the CPU does not need to be stopped to transfer data. These modes require a hardware support from the CPU. The **BUFFER** methods require a buffer at the target side. The host is accessing these buffers by memory access to its addresses.

The DCC channel is a real-time debug communication channel which needs to be supported by the CPU. DCC, DCC3, DCC4D require a 4-byte wide channel. DCC7 and DCC8 require a 8-byte wide channel. **BufferC** and **BufferS** require a stopped CPU to transfer data that's why this methods are very slow compared to the other methods.

BufferE	Transfers the data via Dual Port Memory access. This method requires a buffer at the target side and a special CPU feature that allow to read memory while CPU is running
BufferC	Transfers the data after the CPU is stopped.
BufferS	Transfers the data when the CPU has reached a SPOT breakpoint
DCC3	4-byte wide channel, where the first byte defines if the last 3 bytes are used.
DCC	4-byte wide channel (e.g. ARM family)
DCC4D	4-byte wide channel without FDX high level protocol, every package is 4 byte wide
DCC7	8-byte wide channel, where the first byte defines if the last 8 bytes are used.
DCC8	8-byte wide channel

See also

■ <trace>.state ■ FDX

▲ 'General Data Transfer' in 'Application Note for FDX'
▲ 'Release Information' in 'Legacy Release History'

FDX.Mode — Set the trace operation mode

Format:	**FDX.Mode** [*<mode>*]
<mode>:	**Fifo** | **Stack**
	Compress

Selects the trace operation mode. The most common operation modes are:

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.
Compress	Compressed information transfer for FDX trace.

See also

■ FDX ■ <trace>.Mode

▲ 'General Data Transfer' in 'Application Note for FDX'

FDX.Out — Send FDX data

Format:	**FDX.Out** *<address>* [**%***<format>*] [*<data>* | *<string>*]
<format>	**Byte** | **Word** | **Long** | **Quad** | **TByte** | **PByte** | **HByte** | **SByte** | **Float** | **LE** | **BE**

Send data to the FDX channel specified with the *<address>* parameter.

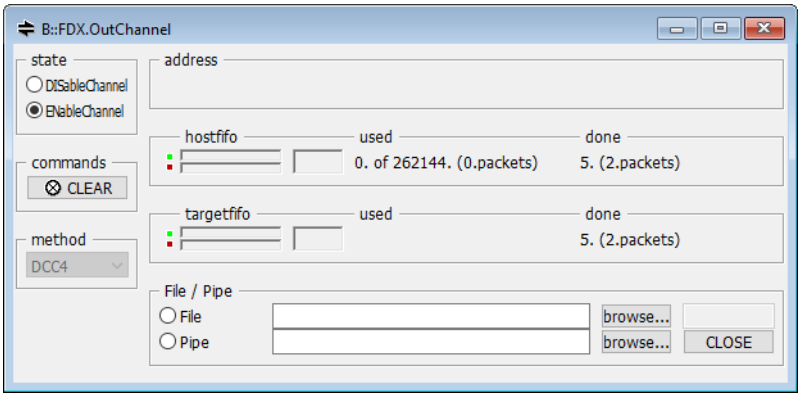
See also

■ <trace>.state ■ FDX

▲ 'General Data Transfer' in 'Application Note for FDX'

Format: **FDX.OutChannel** [*<address>*]

Show the state of the output channel. An address parameter has to be specified if FDX doesn't use DCC (see **FDX.METHOD**).



See also

■ <trace>.state ■ FDX

▲ 'General Data Transfer' in 'Application Note for FDX'
▲ 'Release Information' in 'Legacy Release History'

FDX.PipeREAD Define named pipe for input channel

Format: **FDX.PipeREAD** *<address> <pipe_name>*

Defines a named pipe for FDX input channel.

See also

■ <trace>.state ■ FDX

▲ 'General Data Transfer' in 'Application Note for FDX'

Format: **FDX.PipeWRITE** *<address> <pipe_name>*

Defines a named pipe for FDX output channel.

See also

■ <trace>.state ■ FDX

▲ 'General Data Transfer' in 'Application Note for FDX'

FDX.Rate Select sampling rate

Format: **FDX.Rate** [*<rate>* | *<resolution>*]

Selects the sampling rate in samples/s.

See also

■ <trace>.state ■ FDX

▲ 'General Data Transfer' in 'Application Note for FDX'

FDX.READ Define FDX input file

Format: **FDX.READ** *<address> <file>*

Defines FDX input file.

See also

■ <trace>.state ■ FDX

▲ 'General Data Transfer' in 'Application Note for FDX'

Format: **FDX.TImestamp OFF** | **Up** | **Down** | **Rate** *<rate>*

Default: OFF.

Configure timestamps for the FDX trace. The FDX 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 FDX target code does not store timestamps in the FDX 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 FDX target code stores timestamps in the FDX trace records and if the timestamp increments with each timer tick.
Down	Enable timestamp counter, counting downwards. Use this setting if the FDX target code stores timestamps in the FDX trace records and if the timestamp decrements with each timer tick.
Rate *<rate>*	Frequency of the timestamp in ticks per second.

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

```
FDX.TimeStamp Up
FDX.TimeStamp Rate 16000000.
```

See also

■ <trace>.state ■ FDX

▲ 'General Data Transfer' in 'Application Note for FDX'

FDX.TraceChannel Define FDX trace channel

Format: **FDX.TraceChannel** [*<address>*]

Defines FDX trace channel. An address parameter has to be specified if FDX doesn't use DCC (see **FDX.METHOD**).

Examples:

```
; Example for DCC methods:
  FDX.METHOD DCC4
  FDX.Mode COMPRESS ON
  FDX.SIZE 100000.
  FDX.TraceChannel
  FDX.OFF

; Example for buffered methods:
  FDX.METHOD BUFFERE
  FDX.Mode COMPRESS ON
  FDX.SIZE 100000.
  FDX.TraceChannel FdxTraceSendBuffer
  FDX.OFF
```

See also

■ <trace>.state ■ FDX

▲ 'General Data Transfer' in 'Application Note for FDX'

FDX.WRITE — Define FDX output file

Format: **FDX.WRITE** *<address> <file>*

Defines FDX output file.

See also

■ <trace>.state ■ FDX

▲ 'General Data Transfer' in 'Application Note for FDX'

Generic FDX Trace Commands

FDX.Arm — Arm the trace

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

FDX.AutoArm — Arm automatically

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

FDX.AutoInit — Automatic initialization

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

FDX.BookMark — Set a bookmark in trace listing

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

FDX.Chart — Display trace contents graphically

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

FDX.ComPare — Compare trace contents

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

FDX.DISable — Disable the trace

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

FDX.DRAW — Plot trace data against time

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

FDX.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).

FDX.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).

FDX.Find — Find specified entry in trace

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

FDX.FindAll — Find all specified entries in trace

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

FDX.FindChange — Search for changes in trace flow

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

FDX.GOTO — Move cursor to specified trace record

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

FDX.Init — Initialize trace

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

FDX.List — List trace contents

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

FDX.ListNesting — Analyze function nesting

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

FDX.ListVar — List variable recorded to trace

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

FDX.LOAD — Load trace file for offline processing

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

FDX.OFF — Switch off

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

FDX.PROfileChart — Profile charts

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

FDX.PROTOcol — Protocol analysis

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

FDX.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).

FDX.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).

FDX.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).

FDX.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).

FDX.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).

FDX.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).

FDX.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).

FDX.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).

FDX.REF — Set reference point for time measurement

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

FDX.RESet — Reset command

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

FDX.SAVE — Save trace for postprocessing in TRACE32

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

FDX.SelfArm — Automatic restart of trace recording

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

FDX.SIZE — Define buffer size

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

FDX.SnapShot — Restart trace capturing once

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

FDX.state — Display trace configuration window

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

FDX.STATistic — Statistic analysis

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

FDX.Timing — Waveform of trace buffer

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

FDX.Timing — Waveform of trace buffer

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

FDX.TRACK — Set tracking record

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

FDX.View — Display single record

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

FDX.ZERO — Align timestamps of trace and timing analyzers

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

FIFO — Display on-chip trace FIFO

Format: **FIFO**

Shows the raw data of the on-chip trace fifo (where applicable).

FLASH — Memory mapped FLASH memories

Memory mapped FLASH devices can be programmed and erased using the **FLASH** command group.

For more information about programming NOR and internal FLASH memories, please refer to **"Onchip/NOR FLASH Programming User's Guide"** (norflash.pdf).

Please refer to the **FLASHFILE** command group for information about programming non-memory mapped FLASH memories as NAND and serial FLASH devices.

See also

- FLASH.AUTO
- FLASH.CFI
- FLASH.CLocK
- FLASH.CreateALIAS
- FLASH.EPILOG
- FLASH.GETID
- FLASH.List
- FLASH.MultiProgram
- FLASH.Program
- FLASH.ReProgram
- FLASH.SPI
- FLASH.TARGET
- FLASH.UNLOCK
- FLASHFILE
- FLASH.CFI.SIZE()
- FLASH.BSDLaccess
- FLASH.CHANGEtype
- FLASH.Create
- FLASH.Delete
- FLASH.Erase
- FLASH.HOOKSCRIPT
- FLASH.LOCK
- FLASH.OFFSET
- FLASH.PROLOG
- FLASH.RESet
- FLASH.state
- FLASH.TARGET2
- FLASH.UNSECUREerase
- BSDL.FLASH
- FLASH.List.STATE.PENDING()

▲ 'FLASH Functions' in 'General Function Reference'
▲ 'Introduction' in 'Onchip/NOR FLASH Programming User's Guide'

FLASH.AUTO — Auto programming of FLASH

Format: **FLASH.AUTO** [*<unit>* | *<address_range>* | **ALL** | **off** | **CANCEL** | **/CENSORSHIP**]

Activates the auto programming mode for the selected FLASH memory unit, address range or all declared devices.

The auto programming mode can be used:

- To patch code in FLASH.
- To set software breakpoints into FLASH.

NOTE:	The **FLASH.AUTO off** and **FLASH.ReProgram off** commands automatically erase the modified sectors before writing them. Consequently, do *not* use **FLASH.Erase** when using the auto or FLASH reprogramming mode. If you do, you will lose the advantage of reprogramming only modified sectors, which will result in a loss of performance.

<unit>	Activate the auto programming mode for all sectors of the specified unit.
ALL	Activate the auto programming mode for all FLASH sectors.
off	With parameter “off” or without argument the auto programming mode is terminated. Terminating the auto programming mode lets you program only the modified sectors.
CANCEL	Abort without programming pending changes.
CENSORSHIP	CPU specific option: Allows you to program the range where the FLASH security bytes are located.

Example:

```
; (--)                                ; No FLASH.Erase!
FLASH.AUTO ALL                        ; Activate all FLASHs for programming

Data.LOAD.Binary data.bin             ; load binary file

FLASH.AUTO off                        ; Erase and program all modified
                                      ; sectors

Data.LOAD.Binary data.bin /DIFF       ; compare the contents of the FLASH
                                      ; with the file contents

IF FOUND()                            ; FOUND() returns TRUE if a
     PRINT "Not ok!"                  ; difference was found
ELSE
     PRINT "FLASH ok!"
```

See also

■ FLASH ■ FLASH.state ❏ FLASH.ProgramMODE()

▲ ’Programming Commands’ in ’Onchip/NOR FLASH Programming User’s Guide’

Format: **FLASH.BSDLaccess ON** | **OFF**

Enables or disables FLASH memory access via boundary scan. The boundary scan chain must be configured with **BSDL.FLASH.IFDefine** and **BSDL.FLASH.IFMap**.

See also

- FLASH
- BSDL.FLASH.IFDefine
- BSDL.CHECK.FLASHCONF()
- FLASH.state
- BSDL.FLASH.IFMap

▲ 'FLASH Programming via Boundary Scan' in 'Onchip/NOR FLASH Programming User's Guide'

FLASH.CFI Generate FLASH declaration by CFI

Format 1: **FLASH.CFI** [[*<unit>*] *<address>* | *<range>* *<bus_width>*]
[**/BSPLIT** *<increment>* *<offset>* *<width>*]

Format 2: **FLASH.CFI** [*<unit>*] *<address>* | *<range>* *<bus_width>*
/TARGET *<code_address>* *<data_address>* *<buffer_size>*
[**/DualPort** | **/BSPLIT** *<increment>* *<offset>* *<width>*]

Format 3: **FLASH.CFI** [*<unit>*] *<address>* | *<range>* *<bus_width>*
/TARGET2 *<code_address>* *<data_address>* *<buffer_size>*
[**/DualPort** | **/BSPLIT** *<increment>* *<offset>* *<width>*]

<bus_width>: **Byte** | **Word** | **Long** | **Quad** | **TByte** | **PByte** | **HByte** | **SByte** | **AUTOwidth**

Generates the FLASH declaration by using the CFI information stored in an off-chip FLASH device.

CFI (Common FLASH memory Interface) is an open standard, that specifies how FLASH identification information can be provided by FLASH devices. The identification information includes the memory size, block configuration, voltage and timing information etc.

Without any parameters a **FLASH.CFI** dialog is opened.

<bus_width> **AUTOwidth**	The *<bus_width>* parameter defines the external data bus size. If the **AUTOwidth** parameter is used, TRACE32 detects the data bus width automatically. This is only recommended if a memory access with the wrong bus width does not result in an error.
TARGET and **TARGET2** *<code_address>* *<data_address>* *<buffer_size>*	Specify that target controlled FLASH programming is used: • TARGET uses the flash family code TARGET • TARGET2 uses the flash family code TARGET2 See also **FLASH.TARGET** and **FLASH.TARGET2**. The flash algorithm is loaded to the *<code_address>*. *<data_address>* specifies the start address for data and stack used by the flash algorithm. *<buffer_size>* specifies the data buffer size.
DualPort	Dual-port can be used for target controlled FLASH programming only. For an explanation of the **DualPort** option, see **FLASH.TARGET**.
BSPLIT *<increment>* *<offset>* *<width>*	Loads only certain bytes of the memory. For an illustration of *<increment>*, *<offset>*, and *<width>*, see **FLASH.Create**. *<increment>* number of bytes which the other two parameters refer to. *<offset>* defines the offset of the bytes being programmed. *<width>* defines the number of bytes being programmed.

Format 1

If **Format 1** is used, a FLASH declaration for **TRACE32 tool based FLASH programming** is generated. Refer to **"TRACE32 Tool-based vs. Target-controlled FLASH Programming"** in Onchip/NOR FLASH Programming User's Guide, page 75 (norflash.pdf) for for a description of the TRACE32 FLASH programming techniques,

```
FLASH.RESet                  ; reset all FLASH declarations

FLASH.CFI 0xc0000000 Long    ; perform FLASH declaration by CFI

FLASH.List                   ; display the FLASH declaration

AREA.view                    ; display the monitoring of the TRACE32
                             ; FLASH declaration commands
```

If several FLASH devices of the same type are used in serial, a **FLASH.CFI** for each FLASH device has to be used.

```
FLASH.RESet                   ; reset all FLASH declarations

FLASH.CFI 0xc0000000 Long     ; perform FLASH declaration by CFI

FLASH.CFI 0xc2000000 Long     ; perform FLASH declaration by CFI

FLASH.List                    ; display the FLASH declaration

AREA.view                     ; display the monitoring of the TRACE32
                              ; FLASH declaration commands
```

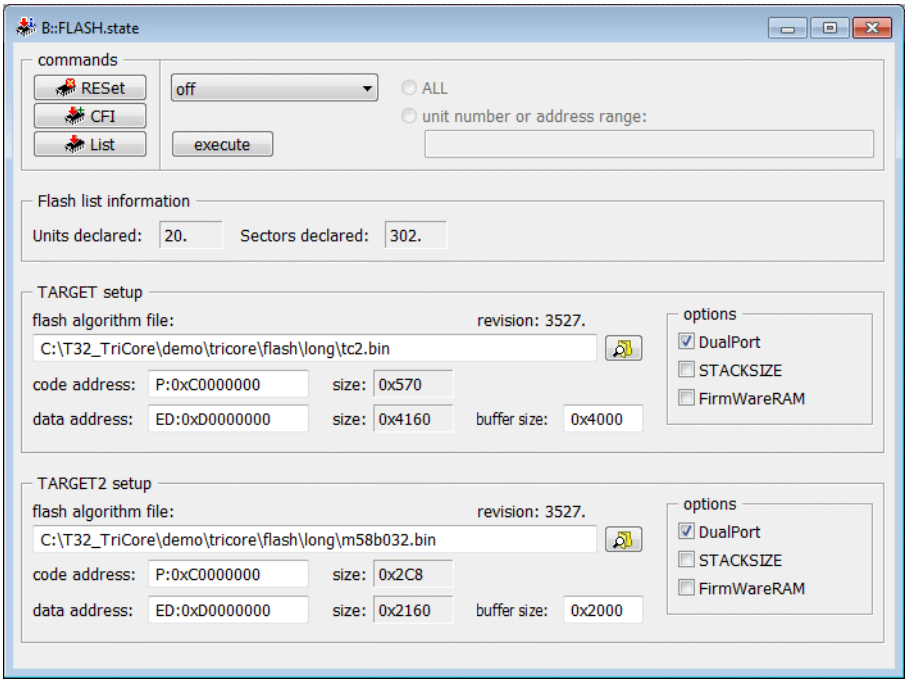
Format 2

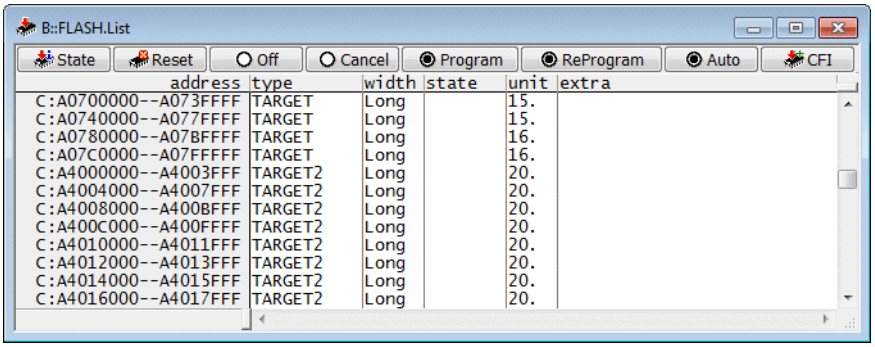
If **Format 2** is used, a FLASH declaration for **target controlled FLASH programming** is generated. Refer to **"TRACE32 Tool-based vs. Target-controlled FLASH Programming"** in Onchip/NOR FLASH Programming User's Guide, page 75 (norflash.pdf) for a description of the TRACE32 FLASH programming techniques,

```
; reset all FLASH declarations
FLASH.RESet

; FLASH.CFI <address> <bus_width> /TARGET <code_address> <data_address>
; <buffer_size>
FLASH.CFI 0xc0000000 Long /TARGET 0x1000 0x2000 0x1000

; display flash programming dialog to check settings
FLASH.state

; display the FLASH declaration
FLASH.List

; display the monitoring of the TRACE32 FLASH declaration commands
AREA.view
```

If **Format 3** is used, a second FLASH declaration for **target controlled FLASH programming** is generated. This is needed, for example, to program processor internal and processor external NOR flash or HyperFlash together. Refer to **"TRACE32 Tool-based vs. Target-controlled FLASH Programming"** in Onchip/NOR FLASH Programming User's Guide, page 75 (norflash.pdf) for a description of the TRACE32 FLASH programming techniques,

```
…

; script to set up onchip flash configuration
DO ~~/demo/tricore/flash/tc29x.cmm CPU=TC298TF PREPAREONLY DUALPORT=1

; configure external bus interface
…

; perform flash declaration for external NOR flash by CFI
FLASH.CFI 0xA4000000 Long /TARGET2 0xC0000000 0xD0000000 0x2000 /DUALPORT

; check configuration
FLASH.state
FLASH.List
```





See also

- FLASH
- FLASH.List
- FLASH.state
- FLASH.TARGET
- FLASH.TARGET2
- FLASH.CFI.SIZE()

▲ 'Standard Approach' in 'Onchip/NOR FLASH Programming User's Guide'
▲ 'Release Information' in 'Legacy Release History'

[Example]

Format:	**FLASH.CHANGEtype ALL** | *<unit>* | *<address_range>* [*<type>* *<width>* *<extra_value>* **/***<option>*]
<type>:	**TARGET** | **TARGET2** | …
<width>:	**Byte** | **Word** | **Long** | **Quad** | **TByte** | **PByte** | **HByte** | **SByte**
<option>:	**AutoInc** [*<increment>*] | **EraseALIAS** | **BootModeHeaDer** | **SWAP** **KEEP** *<address_range>* | **OTP** | **INFO** *<string>* | **DISableBulkErase**

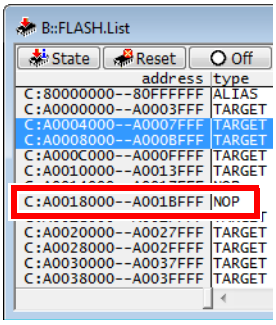
Changes the FLASH type (FLASH family code) for the specified range. All other options remain unchanged.

ALL	Changes the FLASH declaration of all FLASH sectors.
<unit>	Change the FLASH declaration of the specified unit.
<address_range>	Change the FLASH declaration of the specified address range.
<type>	FLASH type, also referred to as FLASH family code. For more information, see **FLASH.Create**.
<width>	For information about the data bus width, see **FLASH.Create**.
<extra_value>	An extra value that is passed to the FLASH algorithm. The extra value is displayed in the **extra** column of the **FLASH.List** window.
<option>	For a description of the options, see **FLASH.Create**.

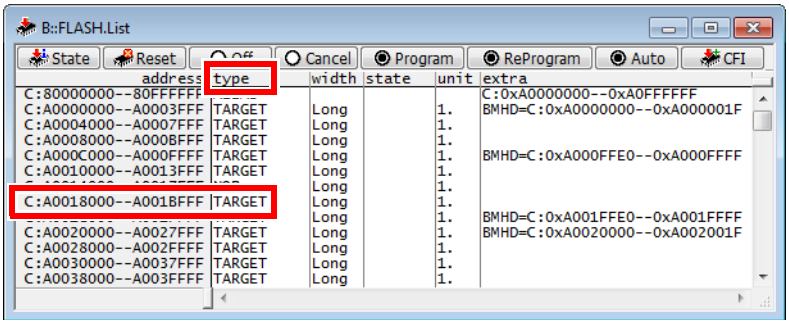
Example:

```
;For the specified address range, change current type to type 'TARGET'
FLASH.CHANGEtype 0xA0018000--0xA001BFFF TARGET
```

Before



After



See also

■ FLASH ■ FLASH.state

▲ 'Release Information' in 'Legacy Release History'

FLASH.CLocK — Setup input clock for processor internal flash

Format: **FLASH.CLocK** *<frequency>* | **AUTO**

The command is used to set up the FLASH module input clock for processor internal flash (target-controlled on-chip FLASH programming only, refer to **"TRACE32 Tool-based vs. Target-controlled FLASH Programming"** in Onchip/NOR FLASH Programming User's Guide, page 75 (norflash.pdf) for a description of the TRACE32 FLASH programming techniques).

Examples:

```
FLASH.CLocK AUTO              ; the FLASH programming algorithm measures
                              ; the input clock before programming or
                              ; erasing the FLASH
```

```
FLASH.CLocK 10MHz             ; a fixed input clock is used
```

See also

■ FLASH ■ FLASH.state ❑ FLASH.CLocK.Frequency()

Format:	**FLASH.Create** *<unit> <physical_range> <sector_size> <family_code> <bus_width> <extra_value>* [*/<option>*]
<family_code>:	**TARGET** | **TARGET2** | …
<bus_width>:	**Byte** | **Word** | **Long** | **Quad** | **TByte** | **PByte** | **HByte** | **SByte**
<option>:	**AutoInc** [*<increment>*] | **EraseALIAS** | **BootModeHeaDer** | **SWAP** **KEEP** *<address_range>* | **CENSORSHIP** *<address_range>* | **OTP** | **INFO** *<string>* | **BSPLIT** *<increment> <offset> <width>* | **DISableBulkErase**

Declaration of FLASH memories.

A *<unit>* number has to be declared for each device. If the processor has on-chip FLASH that is declared automatically by TRACE32, the unit numbers 1 … n are already in use for the on-chip FLASH. Please use the **FLASH.List** command to check the next available unit number.

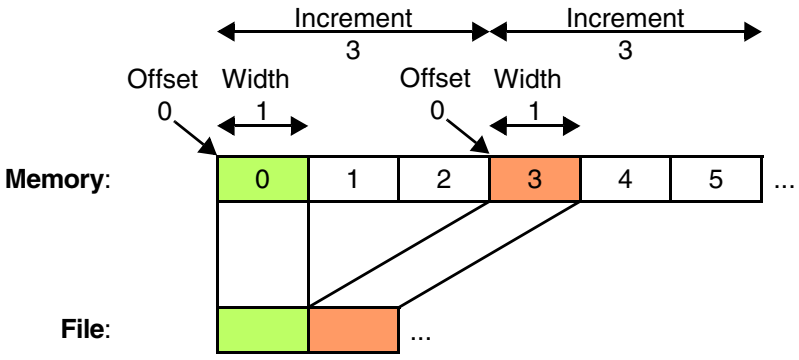
If the FLASH has sectors of different size (boot block devices) one **FLASH.Create** command has to be entered for each sector size.

<bus_width>	The *<bus_width>* parameter defines the external data bus 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) If the data bus size is wider than the bus size of the FLASH device, it will be assumed that FLASH devices of the same type are in parallel.
<extra_value>	An extra value that is passed to the FLASH algorithm. The extra value is displayed in the **extra** column of the **FLASH.List** window.
<family_code>	The *<family_code>* determines the programming algorithm. The FLASH types and the corresponding *<family_codes>* are listed under **"List of Supported FLASH Devices"** (flashlist.pdf).
<sector_size>	If no *<sector_size>* is specified, then one sector occupying the full *<physical_range>* will be assumed.

AutoInc *<increment>*	Increments the **FLASH.Create** extra value for every created sector by *<increment>*. The default *<increment>* is 1. See example. The **AutoInc** option is only available for **TARGET**, **TARGET2**, and **NOP** and a few other FLASH family codes.
BootModeHeaDer *<address_range>*	Only available for the TriCore debugger. See example. Does not delete a valid TriCore boot mode header when the commands **FLASH.Erase** and **FLASH.ReProgram** are executed. But, after the FLASH reprogramming mode has been activated (see **FLASH.ReProgram**), the TriCore boot mode header can be overwritten.
BSPLIT *<increment>* *<offset>* *<width>*	Loads only certain bytes of the memory. For an illustration of *<increment>*, *<offset>*, and *<width>*, see below. *<increment>* number of bytes which the other two parameters refer to. *<offset>* defines the offset of the bytes being programmed. *<width>* defines the number of bytes being programmed.
CENSORSHIP *<address_range>*	The FLASH security bytes are **not** erased or changed by any FLASH command. If you want to change or erase them, you have to use the **FLASH.AUTO ... /CENSORSHIP** command.
DISableBulkErase	Prevents unintended chip erase for truncated declaration of flash address ranges.
EraseALIAS *<address_range>*	The **EraseALIAS** option allows you to apply the commands **FLASH.Auto.off** and **FLASH.ReProgram.off** to non-contiguous physical sectors. The **EraseALIAS** option is, for example, used for a processor-internal FLASH with visible ECC sectors.
INFO *<string>*	Adds a user-defined comment to the **extra** column of the **FLASH.List** window (max. length 64 characters). See example.
KEEP *<address_range>*	Does not delete the specified address range when the commands **FLASH.Erase** and **FLASH.ReProgram** are executed. See example. But, after the FLASH reprogramming mode has been activated (see **FLASH.ReProgram**), the specified address range can be overwritten.

OTP	States in a declaration that the specified range is an OTP sector (**O**ne **T**ime **P**rogrammable). See example. All regular FLASH erase and FLASH programming commands have no effect on the OTP sector. In the **FLASH.List** window, the **state** column displays “nop” for this OTP sector. To activate the FLASH programming mode for an OTP sector, use the **FLASH.Program ... /OTP** command. In the **FLASH.List** window, the **state** column displays “otp” for this OTP sector.
SWAP	Supports **SOTA/FOTA** FLASH configurations. In case sectors are physically swapped, FLASH programming is redirected to physical address of declared swap sector.

BSPLIT: illustration of <increment> = 3, <offset> = 0, and <width> =1



Example 1

FLASH declaration for an Am29LV640 in 16 bit mode on a 16 bit bus. The FLASH provides 128 sectors each with 64 KByte:

```
FLASH.RESet
FLASH.Create 1. 0x0--0x7FFFFF 0x10000 AM29LV100 Word
```

Example 2

FLASH declaration for 2 Intel 28F128J3 in 16 bit mode, 2 FLASHs in parallel on a 32 bit bus. Each FLASH provides 128 sectors with 128 KByte. Since the FLASHs are in parallel the sector size used with the **FLASH.Create** command is 2 x 128 KByte:

```
FLASH.RESet
FLASH.Create 1. 0x0--0x1FFFFFF 0x40000 I28F200J3 Long
```

Example 3

FLASH declaration for an Am29DL322DB in 8 bit mode on a 8 bit bus (FLASH sectors of different size - bottom boot block device). The 8 boot blocks have a size of 8 KByte, the 63 main blocks have a size of 64 KBytes:

```
FLASH.RESet
FLASH.Create 1. 0x00000--0x00FFFF 0x02000 AM29LV100B Byte
FLASH.Create 1. 0x10000--0x3FFFFF 0x10000 AM29LV100B Byte
```

You will find an up-to-date list of all supported FLASHs at the LAUTERBACH website under: **https://www.lauterbach.com/ylist.html**. The list is also available under **"List of Supported FLASH Devices"** (flashlist.pdf).

The *<family_code>* **TARGET** must be selected if target controlled FLASH programming is used. The target-based FLASH programming is configured via the **FLASH.Target** command.

EEPROM can be used to declare an **EEPROM** area. When activated write accesses result in EEPROM programming sequences. This is useful if both FLASH and EEPROM areas are used by the download file.

If no data accesses are allowed to a specific memory area, the option **NOP** can be used, to prevent any access when data transfer commands are used.

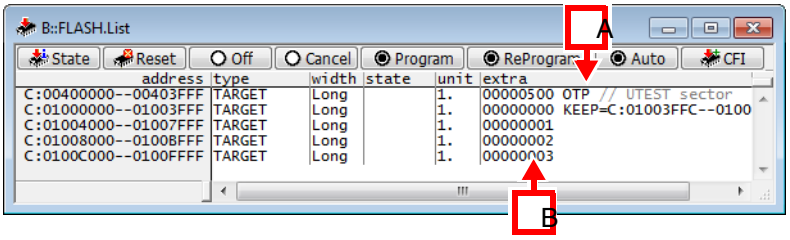
Compatible types are supported even if they are not explicitly listed. (Please note that not all FLASH types have been tested in all configurations).

It is recommended to use **Long** or **Word** option for load or memory copy command, corresponding to the *<bus_width>* defined with the **FLASH.Create** command.

On some ATMEL flash types the data written must be aligned to the flash page size. Use binary file format or copy from virtual memory if possible (refer to **Data.Load <file> /VM** for more information). TRACE32 software based programming is not working on all FLASHs.

Example 4 - /AutoInc, /KEEP, /OTP, and /INFO

```
FLASH.RESet

FLASH.Create 0x1000000--0x100FFFF 0x4000 TARGET Long 0. \
                                    /AutoInc /KEEP 0x1003FFC--0x1003FFF

FLASH.Create 0x0400000--0x0403FFF 0x4000 TARGET Long 0x500 \
                                    /OTP /INFO "UTEST sector"

FLASH.List
```



A A user-defined comment is added by **/INFO**. **B** Sector IDs are incremented by **/AutoInc**.

Example 5 - /BootModeHeaDer

The **BootModeHeaDer** option is only available for TriCore.

Example for a TriCore internal FLASH:

```
FLASH.Create 1. 0xA0000000--0xA0003FFF TARGET Long \
                                    /BootModeHeaDer 0xA0000000--0xA000001F
```

See also

- FLASH
- FLASH.state
- FLASH.SECTOR.EXTRAvalue()

- 'FLASH Declaration in Detail' in 'Onchip/NOR FLASH Programming User's Guide'
- 'Release Information' in 'Legacy Release History'

Format: **FLASH.CreateALIAS** *<address_range> <alias_address>*

All FLASH write cycles to *<address_range>* are performed to the address range starting with *<alias_address>*.

This command is useful when the on-chip FLASH is mapped to different addresses, for example:

- At address 0x0C000000--0x0C01FFFF for uncached accesses
- At address 0x08000000--0x0801FFFFF for cached accesses

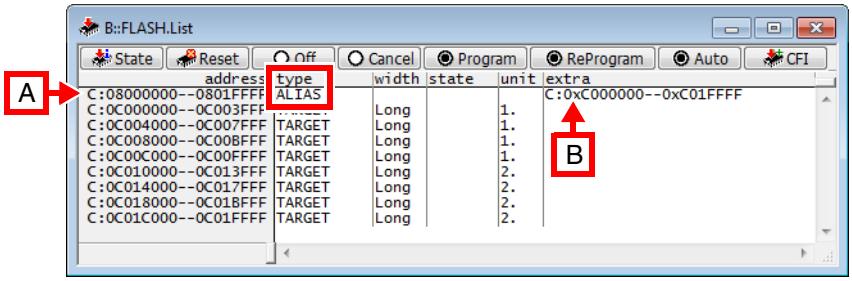
Due to this, a compiler may generate bootcode for the uncached address range and application code for the cached address range. This situation complicates the programming of the on-chip FLASH. With the command **FLASH.CreateALIAS** flash programming is only executed in the uncached address range.

The command **FLASH.CreateALIAS** might also be useful for mirrored FLASH address ranges.

Example:

```
;reset FLASH declaration table
FLASH.RESet

;declare FLASH sectors
FLASH.Create 1. 0x0C000000--0x0C00FFFF  0x4000 TARGET Long
FLASH.Create 2. 0x0C010000--0x0C01FFFF  0x4000 TARGET Long

;all FLASH write cycles to the address range 0x08000000--0x0801FFFF
;(cached) are actually performed to the address range
;0x0C000000--0x0C01FFFF (not cached)
FLASH.CreateALIAS 0x08000000--0x0801FFFF 0x0C000000

;declare the target controlled FLASH algorithm
FLASH.TARGET 0x1FFFE000 0x20000000 0x1000 \
                                       ~~/demo/arm/flash/long/xmc4000.bin
;open the window
FLASH.List
```

The alias is also displayed in the listing of the FLASH declarations, see **FLASH.List** window:



A Cached address range. **B** Destination address for uncached memory.

See also

■ FLASH ■ FLASH.state

▲ 'Special Features for Onchip FLASHs' in 'Onchip/NOR FLASH Programming User's Guide'

FLASH.Delete Delete entry in FLASH declaration table

Format: **FLASH.Delete** *<unit>* | *<address_range>* | **ALL** | **off**

The specified FLASH entry is removed from the FLASH declaration table. Use the command **FLASH.RESet** to clear the whole list and the entire FLASH target configuration.

<unit>	Deletes the specified unit.
ALL	Deletes all FLASH entries from the FLASH declaration table.
off	(no effect)

Example:

```
FLASH.Delete 2.                 ; delete unit 2 from the FLASH declaration
                                ; table. A unit can consist of multiple
                                ; flash sectors.

FLASH.Delete 0x2000--0xffff     ; delete the specified flash sectors.
                                ; from the FLASH declaration table.

FLASH.Delete ALL                ; delete all flash sectors from the
                                ; FLASH declaration table.
```

See also

■ FLASH ■ FLASH.state

▲ 'FLASH Declaration in Detail' in 'Onchip/NOR FLASH Programming User's Guide'

The **FLASH.EPILOG** command group allows to define a sequence of read/write accesses that are automatically performed directly after the execution of one of the following commands. **FLASH.Erase**, **FLASH.Program**, **FLASH.AUTO** or **FLASH.ReProgram**. The complementary command **FLASH.PROLOG** performs read/write accesses before the execution of the command.

See also

- FLASH
- FLASH.state

FLASH.EPILOG.CONDition Define condition for FLASH epilog

Format:	**FLASH.EPILOG.CONDition** *<condition>*
<condition>:	*<memory_access>* **&** *<mask>* **==** *<value>* *<memory_access>* **&** *<mask>* **!=** *<value>*
<memory_access>:	**Data.Byte(***<address>***)** | **Data.Word(***<address>***)** | **Data.Long(***<address>***)**

Defines a condition on which the command sequence defined with **FLASH.EPILOG.SEQuence** will be executed.

<memory_access>	Supported **Data.*()** functions are: • **Data.Byte()** and its short form **D.B()** • **Data.Long()** and its short form **D.L()** • **Data.Word()** and its short form **D.W()**

FLASH.EPILOG.CORE Select core for FLASH epilog

Format:	**FLASH.EPILOG.CORE** *<core_number>*

Selects the core for which you want to define one or more FLASH epilogs.

Prerequisite: You have successfully configured an SMP system with the **CORE.ASSIGN** command.

Example: The following example shows how to define a FLASH epilog that is executed on core 3 of a multicore chip.

```
;Select the core for which you want to define a FLASH epilog
FLASH.EPILOG.CORE 3.

;Define the FLASH epilog for core 3
FLASH.EPILOG.CONDition <your_code>
FLASH.EPILOG.SEQuence  <your_code>
```

For information on how to configure two different FLASH epilogs, see **FLASH.EPILOG.SELect**.

FLASH.EPILOG.OFF — Switch FLASH epilog off

Format: **FLASH.EPILOG.OFF**

Disables the execution of the **FLASH.EPILOG** sequence.

FLASH.EPILOG.ON — Switch FLASH epilog on

Format: **FLASH.EPILOG.ON**

Enables the execution of the **FLASH.EPILOG** sequence.

FLASH.EPILOG.RESet — Reset all FLASH epilogs

Format: **FLASH.EPILOG.RESet**

Switches the **FLASH.EPILOG** feature off and clears all settings.

Format: **FLASH.EPILOG.SELect** *<index_number>*

Increments the index number for each new FLASH epilog. This is useful, for example, if you need two separate FLASH epilogs with each FLASH epilog having its own **FLASH.EPILOG.CONDition**.

TRACE32 automatically assigns the index number 1. to the 1st **FLASH.EPILOG.SEQuence**. If you require a 2nd, separate FLASH epilog sequence, then increment the *<index_number>* to 2. Otherwise the 2nd FLASH epilog will overwrite the 1st FLASH epilog. You can define a maximum of 10 FLASH epilogs.

FLASH.EPILOG.SEQuence Define FLASH epilog sequence

Format: **FLASH.EPILOG.SEQuence** *<command>* …

<command>:
SET *<address>* **%***<format>* *<data>*
SETI *<address>* **%***<format>* *<data>* *<increment>*
SETS *<address>*
GETS *<address>*

Defines a sequence of **Data.Set** commands that are automatically executed by the TRACE32 software directly after the execution of the FLASH command.

SET Parameters: *<address>* **%***<format>* *<value>*
Write *<value>* with data type *<format>* to *<address>*

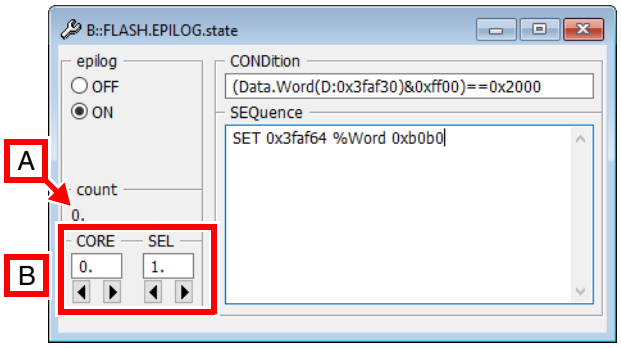
SETI Parameters: *<address>* **%***<format>* *<start>* *<increment>*
At the first time performed, write *<start>* to *<address>*.
<start> is incremented by *<increment>* on each successive call.

GETS Parameters: *<address>* **%***<format>*
Reads the value at *<address>* and stores it into an internal data buffer. The internal data buffer can contain multiple records and is reset when the command **FLASH.EPILOG.Sequence** is called.

SETS Parameters: *<address>* **%***<format>*
If the internal data buffer contains a record for *<address>*, the stored value is written to the processor.

Format:	**FLASH.EPILOG.state**

Opens the **FLASH.EPILOG.state** window, where you can configure FLASH epilogs.



A Counts the number of times the **FLASH.EPILOG.SEQuence** command has been executed.

B Lets you create and view the FLASH epilogs of a particular core. This example shows the 2nd FLASH epilog of core 1.
The **CORE** field is grayed out for single-core targets.

Format: **FLASH.Erase** *<unit>* | *<address_range>* | **ALL** | **off**

Erases the selected FLASH memory unit or an address range or all declared sectors.

<unit>	Erases the specified unit.
ALL	Erases all FLASH sectors of a FLASH memory.
off	(no effect)

Examples:

```
FLASH.Erase 1.                  ; erase unit 1 from the FLASH memory.

FLASH.Erase 0x0--0x1FFFF        ; erase the specified FLASH sectors
                                ; from the FLASH memory.

FLASH.Erase ALL                 ; erase all flash sectors
                                ; of a FLASH memory.
```

See also

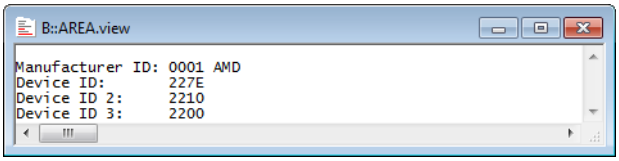
■ FLASH ■ FLASH.state

▲ 'TRACE32 Tool-based Programming' in 'Tips to Solve NOR FLASH Programming Problems'
▲ 'Programming Commands' in 'Onchip/NOR FLASH Programming User's Guide'

Format:	**FLASH.GETID** *<address>* [*<bus_width>* | *<option>* …]
<bus_width>:	**Byte** | **Word** | **Long** | **Quad** | **TByte** | **PByte** | **HByte** | **SByte**
<option>:	**ByteSWAP** | **BSPLIT**

Prints the FLASH manufacturer ID and the device IDs to the message area **A000**. To view the output, open an **AREA.view** window.

The **FLASH.GETID** command is mainly used to identify FLASH devices that do not provide CFI information. The command **FLASH.GETID** together with the function **FLASH.ID()** allows you to program PRACTICE scripts (*.cmm) with device-specific behavior.



<address>	Start address of the FLASH device.
<bus_width>, etc.	For descriptions of the command line arguments, see **FLASH.Create**.

See also

■ FLASH ■ FLASH.state ❑ FLASH.ID()

[Examples]

Format:	**FLASH.HOOKSCRIPT** *<hook_script>*

If a so-called *<hook_script>* is specified, it is started when one of the following FLASH programming commands is entered: **FLASH.ReProgram**, **FLASH.AUTO**, **FLASH.Erase**, **FLASH.Program**, **FLASH.LOCK** or **FLASH.UNLOCK**.

The *<hook_script>* can perform checks, setups etc. to guarantee that the FLASH programming works properly afterwards, e.g. to avoid fatal problems that might occur when the FLASH programming erases or modifies FLASH sectors that contain information that is necessary to operate the debug interface or the chip.

The entered FLASH programming command can then be invoked in the *<hook_script>* by using the **/NoHOOK** option.

<hook_script>	The *<hook_script>* is usually provided by Lauterbach.

Examples:

```
; FLASH declaration

; specification of hook script
FLASH.HOOKSCRIPT ~~~~/flash_hookscript.cmm

; FLASH programming command
FLASH.ReProgram ALL
```

TRACE32 redirects the FLASH programming command to the *<hook_script>* by using the following PRACTICE script call:

```
DO ~~~~/flash_hookscript.cmm HOOKCMD="FLASH.ReProgram ALL"
```

If the *<hook_script>* starts with the following script commands, it is able to read the actual entered FLASH programming command and invoke it, when its checks or setups are done.

```
; read HOOKCMD=<flash_cmd>
LOCAL &param
ENTRY %LINE &param

&flash_cmd=STRing.SCANAndExtract(STRing.UPpeR("&param"),"HOOKCMD=","")
; convert string to command
&flash_cmd=&flash_cmd
…
; invode FLASH programming command, but skip <hook_script>
&flash_cmd /NoHOOK
```

See also

■ FLASH ■ FLASH.state

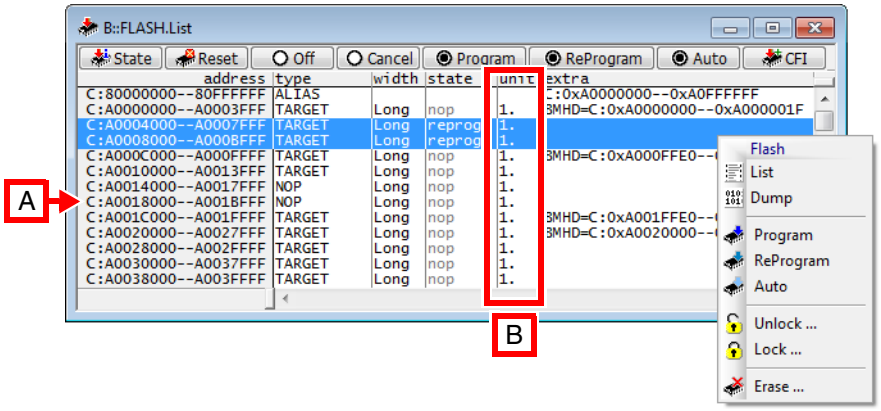
▲ 'Release Information' in 'Legacy Release History'

Format: **FLASH.List all** | *<unit>* | *<address>*

Displays the address range, family code, bus width, state (if programming or auto is activated), and the unit number of declared FLASH sectors. For a description of the columns in the **FLASH.List** window, see table below.

<unit> Opens the window pointing to unit start address.

<address> Opens the window pointing to *<address>*.



A Each row stands for one sector.

B Unit 1. consists of multiple sectors.

Description of Columns in the FLASH.List Window

address	Address range of a sector.
type	FLASH family code. An ALIAS entry is created with **FLASH.CreateALIAS**.
width	Data bus width (Byte, Word, Long, Quad).

state	Values in the **state** column: • **program**: FLASH programming for the selected sector is activated by **FLASH.Program**. • **reprog**: FLASH programming for the selected sector is activated by **FLASH.ReProgram**. • **pending**: The contents in the virtual host memory differs from the real FLASH contents. FLASH will be programmed by **FLASH.AUTO.off** or **FLASH.ReProgram.off**. • **nop**: Sectors that are *not* affected by **FLASH.Program**, **FLASH.AUTO**, or **FLASH.ReProgram**. • **otp**: OTP sector programming is enabled by **FLASH.Program... /OTP**.
unit	A unit consists of multiple sectors. Each row stands for one sector. Related sectors have the same unit number.
extra	Displays the extra value and the options created with **FLASH.Create**, including user-defined comments.

See also

- FLASH
- FLASH.state
- FLASH.CFI
- FLASH.List.STATE.PENDING()

▲ 'Standard Approach' in 'Onchip/NOR FLASH Programming User's Guide'
▲ 'FLASH Declaration in Detail' in 'Onchip/NOR FLASH Programming User's Guide'

FLASH.LOCK — Lock FLASH

Format: **FLASH.LOCK** *<unit>* | *<address_range>* | **ALL** | **off**

Locks the selected sectors of a FLASH device.

<unit>	Locks the specified unit.
ALL	Locks all FLASH sectors.
off	(no effect)

Example:

```
FLASH.LOCK 1.                   ; lock unit 1. in the FLASH memory.

FLASH.LOCK 0x0--0x1FFFF         ; lock the specified range
                                ; in the FLASH memory.

FLASH.LOCK ALL                  ; lock all FLASH sectors
                                ; in the FLASH memory.
```

See also

■ FLASH ■ FLASH.state ■ FLASH.UNLOCK

▲ 'TRACE32 Tool-based Programming' in 'Tips to Solve NOR FLASH Programming Problems'

Available on: MPC555 (K1, K2, K3)

Format: **FLASH.MultiProgram** *<range>*

Simultaneous programming of the internal FLASH is supported for the masks **K1**, **K2**, **K3** and **M** of the **MPC555**. The MPC555 supports simultaneous programming of all 14 flash modules, 8 blocks of FLASH module A and 6 blocks of FLASH module B.

Example:

```
; initialize the virtual memory of TRACE32-ICD with 0xff
Data.Set VM:0x0++0x6ffff %Long 0xffffffff

; load the code for the internal FLASH into the virtual memory
Data.LOAD.Elf file.elf /VM

; Start the simultaneous programming
FLASH.MultiProgram 0x0++0x6ffff
```

See also

- FLASH
- FLASH.state

FLASH.OFFSET Change FLASH control address

Format: **FLASH.OFFSET** [*<offset>*]

On some FLASH memory types a dummy address is used to control FLASH programming and erasure. For this purpose normally the lowest address of the FLASH is used (default offset: 0). If this address can not be accessed, since it is occupied by other peripherals, this command can be used to change these control address.

Example:

```
; 80196 FLASH example
; access to location 0 is not possible (overlaid by register block)

FLASH.Create P:0x0--0x1ffff AM29F100 Word
FLASH.OFFSET 0x1000
```

See also

- FLASH
- FLASH.state

Format: **FLASH.Program** [*<unit>* | *<address_range>* | **ALL** | **off** | **CANCEL**] [**/OTP**]

Activates the FLASH programming mode for the selected FLASH memory unit or address range or for all declared sectors.

While the programming mode is active, all writes from TRACE32 to the activated FLASH memory range are executed as FLASH program cycles. The programming data must be written to the correct location and access class of the FLASH memory.

<unit>	Activates the FLASH programming mode for the specified unit.
ALL	Programs all FLASH sectors.
off	With parameter “off” or without argument the programming mode is terminated.
CANCEL	Abort without programming pending changes.
OTP	**FLASH.Create ... /OTP** declares an OTP sector (**O**ne **T**ime **P**rogrammable). Use **FLASH.Program ... /OTP** to activate the FLASH programming mode for the OTP sector.

NOTE:	Before using the **FLASH.Program** command, it is necessary to erase the FLASH memory with **FLASH.Erase**. This is **not** required with **FLASH.ReProgram** or **FLASH.AUTO** because they automatically erase the sectors before writing them.

Example:

```
FLASH.Erase ALL                          ; Reset FLASH contents
                                         ; (mandatory)

FLASH.Program ALL                        ; Activate all FLASHs for
                                         ; programming

Data.LOAD.Binary data.bin 0x0 /Word      ; load binary file at offset 0x0

FLASH.Program off                        ; deactivate FLASH programming

Data.LOAD.Binary data.bin 0x0 /DIFF      ; compare the contents of the
                                         ; FLASH

IF FOUND()                               ; with the file contents
     PRINT "Not ok!"
ELSE                                     ; FOUND() returns TRUE if a
     PRINT "FLASH ok!"                   ; difference was found
```

See also

■ FLASH ■ FLASH.ReProgram ■ FLASH.state ❑ FLASH.ProgramMODE()

▲ 'Programming Commands' in 'Onchip/NOR FLASH Programming User's Guide'

The **FLASH.PROLOG** command group allows to define a sequence of read/write accesses that are automatically performed before the execution of one of the following commands. **FLASH.Erase**, **FLASH.Program**, **FLASH.AUTO** or **FLASH.ReProgram**. The complementary command **FLASH.EPILOG** performs read/write accesses after the execution of the command.

See also

- FLASH
- FLASH.state

FLASH.PROLOG.CONDition Define condition for FLASH prolog

Format:	**FLASH.PROLOG.CONDition** *<condition>*
<condition>:	*<memory_access>* **&** *<mask>* **==** *<value>* *<memory_access>* **&** *<mask>* **!=** *<value>*
<memory_access>:	**Data.Byte(***<address>***)** | **Data.Word(***<address>***)** | **Data.Long(***<address>***)**

Defines a condition on which the command sequence defined with **FLASH.PROLOG.SEQuence** will be executed.

<memory_access>	Supported **Data.*()** functions are: • **Data.Byte()** and its short form **D.B()** • **Data.Long()** and its short form **D.L()** • **Data.Word()** and its short form **D.W()**

FLASH.PROLOG.CORE Select core for FLASH prolog

Format:	**FLASH.PROLOG.CORE** *<core_number>*

Selects the core for which you want to define one or more FLASH prologs.

Prerequisite: You have successfully configured an SMP system with the **CORE.ASSIGN** command.

Example: The following example shows how to define a FLASH prolog that is executed on core 3 of a multicore chip.

```
;Select the core for which you want to define a FLASH prolog
FLASH.PROLOG.CORE 3.

;Define the FLASH prolog for core 3
FLASH.PROLOG.CONDition <your_code>
FLASH.PROLOG.SEQuence  <your_code>
```

For information on how to configure two different FLASH prologs, see **FLASH.PROLOG.SELect**.

FLASH.PROLOG.OFF — Switch FLASH prolog off

Format: **FLASH.PROLOG.OFF**

Disables the execution of the **FLASH.PROLOG** sequence.

FLASH.PROLOG.ON — Switch FLASH prolog on

Format: **FLASH.PROLOG.ON**

Enables the execution of the **FLASH.PROLOG** sequence.

FLASH.PROLOG.RESet — Reset all FLASH prologs

Format: **FLASH.EPILOG.RESet**

Switches the **FLASH.PROLOG** feature off and clears all settings.

Format: **FLASH.PROLOG.SELect** *<index_number>*

Increments the index number for each new FLASH prolog. This is useful, for example, if you need two separate FLASH prologs with each FLASH prolog having its own **FLASH.PROLOG.CONDition**.

TRACE32 automatically assigns the index number 1. to the 1st **FLASH.PROLOG.SEQuence**. If you require a 2nd, separate FLASH prolog sequence, then increment the *<index_number>* to 2. Otherwise the 2nd FLASH prolog will overwrite the 1st FLASH prolog. You can define a maximum of 10 FLASH prologs.

FLASH.PROLOG.SEQuence Define FLASH prolog sequence

Format: **FLASH.PROLOG.SEQuence** *<command>* …

<command>: **SET** *<address>* **%***<format>* *<data>*
SETI *<address>* **%***<format>* *<data>* *<increment>*
SETS *<address>*
GETS *<address>*

Defines a sequence of **Data.Set** commands that are automatically executed by the TRACE32 software directly before the execution of the FLASH command.

SET Parameters: *<address>* **%***<format>* *<value>*
Write *<value>* with data type *<format>* to *<address>*

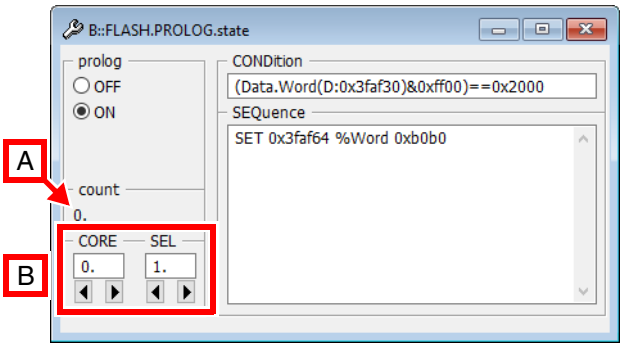
SETI Parameters: *<address>* **%***<format>* *<start>* *<increment>*
At the first time performed, write *<start>* to *<address>*.
<start> is incremented by *<increment>* on each successive call.

GETS Parameters: *<address>* **%***<format>*
Reads the value at *<address>* and stores it into an internal data buffer. The internal data buffer can contain multiple records and is reset when the command **FLASH.PROLOG.Sequence** is called.

SETS Parameters: *<address>* **%***<format>*
If the internal data buffer contains a record for *<address>*, the stored value is written to the processor.

Format:	**FLASH.PROLOG.state**

Opens the **FLASH.PROLOG.state** window, where you can configure FLASH prologs.



A Counts the number of times the **FLASH.PROLOG.SEQuence** command has been executed.

B Lets you create and view the FLASH prologs of a particular core. This example shows the 2nd FLASH prolog of core 1.
The **CORE** field is grayed out for single-core targets.

[Example]

Format: **FLASH.ReProgram** [*<unit>* | *<address_range>* | **ALL** | **off** | **CANCEL**] [**/Erase** | **/FILL**]

Activates the FLASH reprogramming mode for the selected FLASH unit, address range or for all declared devices. In this mode, all changes are written to the virtual host memory of TRACE32. As soon as **FLASH.ReProgram.off** is executed, the FLASH sectors are programmed but only if their contents have changed. The programming time for target-controlled FLASH programming is considerably improved. Refer to **"TRACE32 Tool-based vs. Target-controlled FLASH Programming"** in Onchip/NOR FLASH Programming User's Guide, page 75 (norflash.pdf) for a description of the TRACE32 FLASH programming techniques.

<unit>	Activates the reprogramming mode for the specified unit.
ALL	Activates the reprogramming mode for all FLASH sectors.
off	With parameter "off" or without parameters the FLASH reprogramming mode is terminated. Terminating the FLASH reprogramming mode lets you program only the modified sectors.
CANCEL	Abort without programming pending changes.

Erase	Flags the activated sectors in the virtual host memory as to be erased. The **Erase** option is used to remove obsolete code from sectors that are not reprogrammed (e.g. because a smaller program is flashed into the target).
FILL	Fills all erase sectors with 0 or 1, depending on the FLASH device. Use the **FILL** option to prevent that ECC errors occur after FLASH programming in case of an address-sensitive ECC calculation.

NOTE: The **FLASH.AUTO.off** and **FLASH.ReProgram.off** commands automatically erase the modified sectors before writing them.

- Consequently, do *not* use **FLASH.Erase** when using the auto or FLASH reprogramming mode.
- If you do, you will lose the advantage of reprogramming only modified sectors, which will result in a loss of performance.

Example:

```
; (--)                                ; No FLASH.Erase!
FLASH.ReProgram ALL /Erase            ; Activate all FLASHs for programming

Data.LOAD.Binary data.bin             ; load binary file

FLASH.ReProgram off                   ; Erase and program all modified
                                      ; sectors

Data.LOAD.Binary data.bin /DIFF       ; compare the contents of the FLASH
                                      ; with the file contents

IF FOUND()                            ; FOUND() returns TRUE if a
     PRINT "Not ok!"                  ; difference was found
ELSE
     PRINT "FLASH ok!"
```

See also

■ FLASH ■ FLASH.Program ■ FLASH.state ❑ FLASH.ProgramMODE()

▲ 'Programming Commands' in 'Onchip/NOR FLASH Programming User's Guide'
▲ 'Release Information' in 'Legacy Release History'

FLASH.RESet — Reset FLASH declaration table

Format: **FLASH.RESet**

The debugger-internal table with FLASH-related declarations is cleared and all settings are restored to their default values. Use the command **FLASH.Delete** to remove a single entry or parts of the FLASH list.

This command does not erase the FLASH sectors. Use **FLASH.Erase** for that purpose.

See also

■ FLASH ■ FLASH.state

▲ 'Standard Approach' in 'Onchip/NOR FLASH Programming User's Guide'

See also

- FLASH
- FLASH.state

FLASH.SPI.CFI Generate SPI FLASH sector declaration by CFI

Format:	**FLASH.SPI.CFI** [*<unit>*] *<address>* | *<range>* *<bus_width>* [**/***<option>*]
<bus_width>:	**Byte** | **Word** | **Long** | **Quad** | **TByte** | **PByte** | **HByte** | **SByte**
<option>:	**TARGET** [*<code_address>* *<data_address>* *<buffer_size>* *<file>*] **TARGET2** [*<code_address>* *<data_address>* *<buffer_size>* *<file>*] **DualPort** **OctaPI** **QuadPI**

Generates the FLASH declaration by using the CFI information stored in an SPI FLASH device.

CFI (Common FLASH memory Interface) is an open standard, that specifies how FLASH identification information can be provided by FLASH devices. The identification information includes the memory size, block configuration, voltage and timing information etc.

The target algorithm has to be specified with the **/TARGET[2]** option or with the **FLASH.TARGET** / **FLASH.TARGET2** command.

If the **/TARGET[2]** option is used without further parameters, then the target algorithm has to have been already specified with **FLASH.TARGET** / **FLASH.TARGET2**. The option only specifies in this case which of the two possible algorithms should be used. If no **/TARGET[2]** option is specified, the algorithm that is declared with **FLASH.TARGET** will be used.

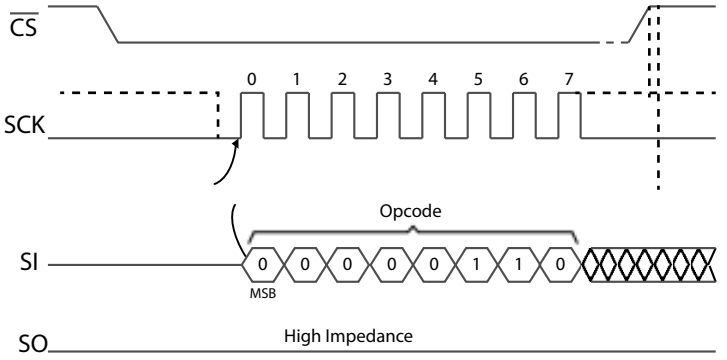
Default serial peripheral interface (SPI) devices use a single data bit (in/out separate), while Quad SPI (QPI) moves for data bits at a time and Octal SPI (OPI) moves eight data bits.

Default	Supports SPI Mode (1-1-1).
QuadPI	Supports QPI Mode (4-4-4).
OctaPI	Supports Octal Mode (8-8-8).

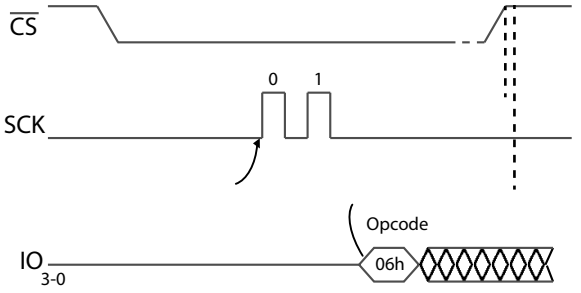
n-n-n means here *<cmd phase iowidth>-<address phase iowidth>-<data phase iowidth>*.

The following graphics display a write enable command phase for the 3 different spi modes.

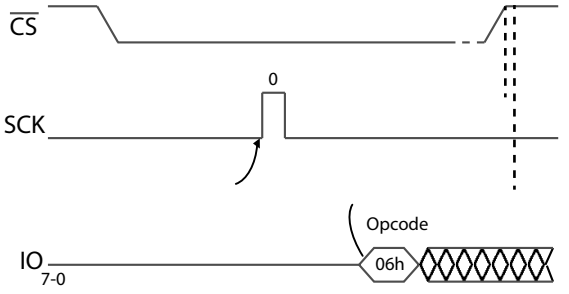
SPI Mode:



QPI Mode:



OPI Mode:



Examples:

Example 1:

```
FLASH.RESet
FLASH.TARGET2 <code_address> <data_address> <buffer_size> <file>
FLASH.SPI.CFI <address> Byte /TARGET2
FLASH.List
AREA.view
```

Example 2:

```
FLASH.RESet
FLASH.SPI.CFI <address> Byte /TARGET <code_address> <data_address> \
                                                    <buffer_size> <file>
FLASH.List
AREA.view
```

FLASH.SPI.CMD Send data to SPI FLASH device

[Examples]

Format:	**FLASH.SPI.CMD** *<unit>* | *<range>* [**%***<format>*] *<data>* … [**/***<option>*] **FLASH.SPICMD** (deprecated)
<format>:	**Byte** | **Word** | **Long** | …
<option>:	**READ** *<number_of_bytes>* [*<address>*] **QPI** **TARGET** **TARGET2**

Sends FLASH-device-specific *<data>* from TRACE32 to a SPI FLASH device. The *<data>* can be any valid FLASH command sequence. For *<data>* that can be sent, refer to the FLASH manufacturer's documentation.

<unit> *<range>*	Sends the SPI FLASH command to the specified FLASH device identified by its unit number or address range.
Byte (default), **Word**, **Long**, …	Data size for integer constants. See **"Keywords for <width>"** (general_ref_d.pdf).
<data>	Any valid FLASH command sequence you want to send to the SPI FLASH, such as commands, addresses, dummy cycles, etc. Examples of SPI-FLASH-device commands: • **0x9F** // the READ ID command • **0x03 0x00 0x10 0x00** // the READ command with arguments 0x3 is the READ command. The three arguments 0x00, 0x10, and 0x00 stand for the SPI FLASH address 0x001000.
READ *<number_of_bytes>*	TRACE32 requests the SPI FLASH to read the specified *<number_of_bytes>*, and then prints the bytes read by the SPI FLASH to the **AREA.view** window.

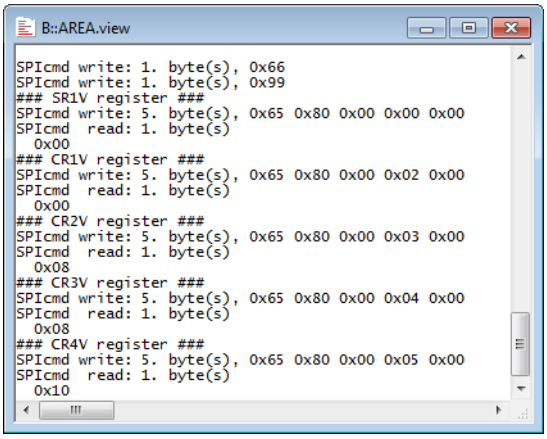
READ *<number_of_bytes>* *<address>*	TRACE32 requests the SPI FLASH to read the specified *<number_of_bytes>*, and then sends the bytes read by the SPI FLASH to the specified *<address>*; typically an address in the TRACE32 virtual memory (VM:). See example 1.
QPI	Execute SPI FLASH access in 4-4-4 QPI mode.
TARGET	Execute target FLASH algorithm declared with **FLASH.TARGET**.
TARGET2	Execute target FLASH algorithm declared with **FLASH.TARGET2**.

Example 1: The SPI FLASH reads the *<number_of_bytes>*, and then TRACE32 sends the bytes read by the SPI FLASH to the specified *<address>* in the TRACE32 virtual memory (VM:).

```
;TRACE32 transmits the command 0x9f (= read ID), and the host receives 4
;bytes of data. Then TRACE32 sends the data received from the host to
;the TRACE32 virtual memory address 0x0.
FLASH.SPI.CMD 1. 0x9f /READ 4. VM:0x0

&data=Data.LONG(VM:0x0)
PRINT "SPI data : 0x" &data //print the FLASH Manufacture and Device ID
```

Example 2: This script shows how to output the contents of internal SPI FLASH registers to the **AREA.view** window.

```
SCREEN.OFF
FLASH.SPI.CMD 1. 0x66  ;enable the SPI FLASH software reset
FLASH.SPI.CMD 1. 0x99  ;perform the SPI FLASH software reset
SCREEN.ON

&address=0x800000

;if the SPI flash is the 3-byte address mode
PRINT "### SR1V register ###"
FLASH.SPI.CMD 1. 0x65 0x80 0x00 0x00 0x00 /READ 0x1

PRINT "### CR1V register ###"
FLASH.SPI.CMD 1. 0x65 0x80 0x0 0x2 0x0 /READ 0x1

PRINT "### CR2V register ###"
FLASH.SPI.CMD 1. 0x65 0x80 0x0 0x3 0x0 /READ 0x1

PRINT "### CR3V register ###"
FLASH.SPI.CMD 1. 0x65 0x80 0x0 0x4 0x0 /READ 0x1

PRINT "### CR4V register ###"
FLASH.SPI.CMD 1. 0x65 0x80 0x0 0x5 0x0 /READ 0x1

;let's open the AREA window to view the result
AREA.view
```



Format:	**FLASH.SPI.GETSFDP** *<unit>* | *<range>* [**%***<format>*] *<data>* …
<format>:	**Byte** | **Word** | **Long** | …

Reads Serial Flash Discovery Parameters from SPI FLASH. Several parameters values can be afterwards utilized with the **FLASH.SPI.SFDP()** function.

FLASH.SPI.RESetMemory Reset SPI FLASH volatile register

[build 134704 - DVD 09/2021]

Format:	**FLASH.SPI.RESetMemory** *<unit>* | *<address>* | *<range>* [**/***<option>*]
<option>:	**TARGET** (Available if sector is not declared) **TARGET2** (Available if sector is not declared)

Default: TARGET.

Resets SPI FLASH internal volatile register values.

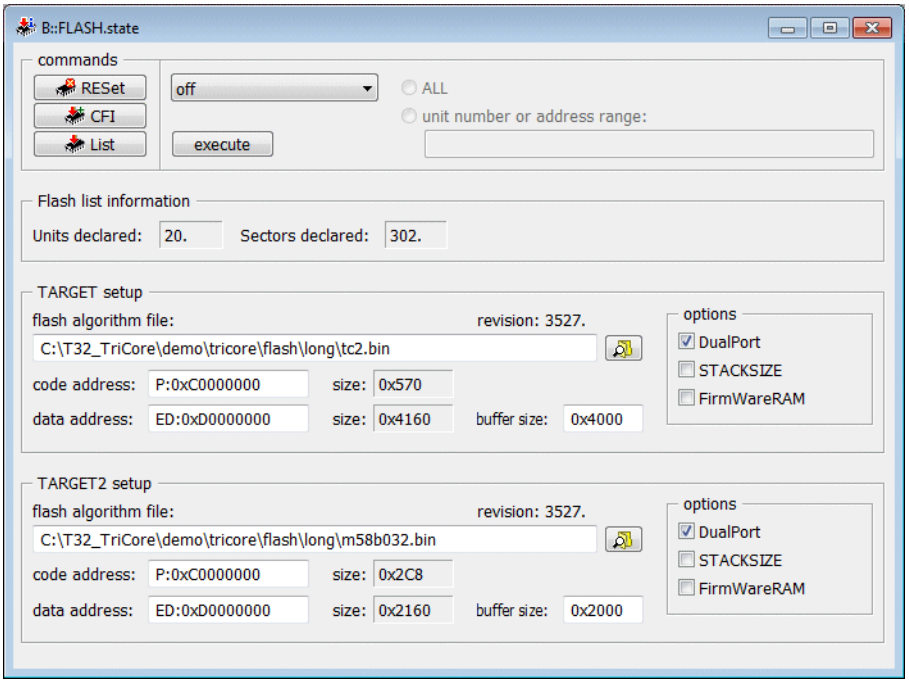
TARGET Execute target FLASH algorithm declared with **FLASH.TARGET**.

TARGET2 Execute target FLASH algorithm declared with **FLASH.TARGET2**.

Format: **FLASH.state**

Opens the **FLASH.state** window, where you can create, check, and modify the setup for memory mapped FLASH programming.

```
; script to program onchip and off-chip flash together

SYStem.CPU TC298TF

…

; script to set up onchip flash configuration
DO ~~/demo/tricore/flash/tc29x.cmm CPU=TC298TF PREPAREONLY DUALPORT=1

; configure external bus interface
…

; prepare off-chip flash configuration
FLASH.CFI 0xA4000000 Long /TARGET2 0xC0000000 0xD0000000 0x2000 /DualPort

FLASH.state
```



See also

- FLASH
- FLASH.AUTO
- FLASH.BSDLaccess
- FLASH.CFI
- FLASH.CHANGEtype
- FLASH.CLocK
- FLASH.Create
- FLASH.CreateALIAS

■ FLASH.Delete
■ FLASH.EPILOG
■ FLASH.Erase
■ FLASH.GETID
■ FLASH.HOOKSCRIPT
■ FLASH.List
■ FLASH.LOCK
■ FLASH.MultiProgram
■ FLASH.OFFSET
■ FLASH.Program
■ FLASH.PROLOG
■ FLASH.ReProgram
■ FLASH.RESet
■ FLASH.SPI
■ FLASH.TARGET
■ FLASH.TARGET2
■ FLASH.UNLOCK
■ FLASH.UNSECUREerase

▲ 'Release Information' in 'Legacy Release History'

FLASH.TARGET Define target controlled algorithm

[Examples]

Format 1: **FLASH.TARGET** *<code_range>* *<data_range>* [*<file>*] \
[**/STACKSIZE** *<size>* | **/FirmWareRAM** *<address_range>*] \
[**/DualPort** | **/CORE** *<number>*]

Format 2: **FLASH.TARGET** *<code_address>* *<data_address>* [*<buffer_size>*] [*<file>*] \
[**/STACKSIZE** *<size>*] | **/FirmWareRAM** *<address_range>*] \
[**/DualPort** | **/CORE** *<number>*]

Prepares target controlled FLASH programming.

STACKSIZE	Declares the stack size for the target algorithm. The default size is 256 bytes.
FirmWareRAM	Declares the *<address_range>* used by the on-chip FLASH programming firmware. This range is protected before programming and restored after FLASH programming, just as is the case for the *<data_range>* and the *<code_range>*. See example.
DualPort	**DualPort** can be used for target controlled FLASH programming only. While FLASH programming is being executed, the TRACE32 host software simultaneously downloads the next command and data via dual-port memory access into the target RAM. **DualPort** can reduce the programming time significantly. See examples.
CORE *<number>*	Defines that the target controlled algorithm is executed on the specified core.

Example for Format 1

FLASH: Intel Strata FLASH 28F320J3A, 16 bit mode, word access
CPU: ARM core

Memory configuration:

- FLASH from 0x0--0x3FFFFF
- RAM starting at address 0xA0000000

```
; Format 1

FLASH.RESet
FLASH.Create 1. 0x0--0x3fffff 0x20000 TARGET Word

;            <code_range>      <data_range>      <flash_algorithm>
FLASH.Target 0xA0000000++0xFFF 0xA0001000++0xFFF \
                               ~~/demo/arm/flash/word/i28f200j3.bin
                               ; binary file only!
```

- `<code_range>`
  - The code for the `<flash_algorithm>` is downloaded to `<code_range>`
  - Required size for the code is `size_of(<flash_algorithm>) + 32 byte`
- `<data_range>`
  - The specified data range is used for `<flash_algorithm>`
  - `<buffer_size>` is the number of bytes which are transferred from the TRACE32 software to the target controlled FLASH programming routine in one step. Recommended buffer size is 4 KByte, smaller buffer sizes are also possible. The max. buffer size is 16 KByte.
  - `<buffer_size> = size_of(<data_range>) - 32 byte argument buffer - 256 byte stack`
- `<flash_algorithm>`

  The FLASH algorithm can be found in: `~~/demo/arm/flash/word/i28f200j3.bin`

Programming procedure:

1. The TRACE32 software takes care of saving the contents of

   `<code_range>`
   `<data_range>`
   used CPU registers

   before the commands **FLASH.Erase**, **FLASH.UNLOCK**, **FLASH.LOCK** are started
   before patching the code in FLASH or setting a software breakpoint to FLASH is performed after **FLASH.AUTO** is used
   when FLASH programming is enabled with the **FLASH.Program** command
2. The FLASH program is loaded to the target and the requested FLASH programming commands are performed.

3. The saved context (register and data) is restored

   after the commands **FLASH.Erase**, **FLASH.UNLOCK**, **FLASH.LOCK** are finished
   after patching the code in FLASH or setting a software breakpoint to FLASH is done when **FLASH.AUTO** is used
   when FLASH programming is disabled with the **FLASH.Program** command.

Complete example:

```
; Format 1

FLASH.RESet
FLASH.Create 1. 0x0--0x3fffff 0x20000 TARGET Word
FLASH.Target 0xA0000000++0xFFF 0xA0001000++0xFFF \
                                    ~~/demo/arm/flash/word/i28f200j3.bin

FLASH.Erase 1.
FLASH.Program 1.
Data.LOAD.Elf my_application.elf /Word
FLASH.Program off

Data.LOAD.Elf my_application.elf /Word /DIFF

IF FOUND()
     Print "Not ok!"
ELSE
     Print "FLASH ok!"
```

Example for Format 2

FLASH: Intel Strata FLASH 28F320J3A, 16 bit mode, word access

CPU: ARM core

Memory configuration:

- FLASH from 0x0--0x3FFFFF
- RAM starting at address 0xA0000000

```
; Format 2

FLASH.RESet
FLASH.Create 1. 0x0--0x3FFFFF 0x20000 TARGET Word

;        <code_address> <data_addr> <buffer_size> <flash_algorithm>
FLASH.Target 0xA0000000 0xA0001000  0x1000 \
                                ~~/demo/arm/flash/word/i28f200j3.bin
                                ; binary file only!
```

- `<code_address>`
  - The code for the `<flash_algorithm>` is downloaded to the target RAM starting at `<code_address>`
  - Required size for the code is `size_of(<flash_algorithm>) + 32 byte`
- `<data_address>`
  - The address range starting with `<data_address>` is used as data range for `<flash_algorithm>`
  - Required size for the data is `<buffer_size> + 32 byte argument buffer + 256 byte stack`
- `<buffer_size>`

  `<buffer_size>` is the number of bytes which are transferred from the TRACE32 software to the target controlled FLASH programming routine in one step. Recommended buffer size is 4 KByte, smaller buffer sizes are also possible. The max. buffer size is 16 KByte.
- `<flash_algorithm>`

  The FLASH algorithm can be found in `~~/demo/arm/flash/word/i28f200j3.bin`

Programming procedure:

1. The TRACE32 software takes care of saving the contents of

   <code_address>++(size_of(<flash_algorithm>)+32 byte)
   <data_address>++(<buffer_size> + 32 byte argument buffer + 256 byte stack>)
   used CPU registers

   before the commands **FLASH.Erase**, **FLASH.UNLOCK**, **FLASH.LOCK** are started
   before patching the code in FLASH or setting a software breakpoint to FLASH is performed after **FLASH.AUTO** is used
   when FLASH programming is enabled with the **FLASH.Program** command

2. The FLASH program is loaded to the target and the requested FLASH programming commands are executed.

3. The saved context (register and data) is restored.

   after the commands **FLASH.Erase**, **FLASH.UNLOCK, FLASH.LOCK** are finished
   after patching the code in FLASH or setting a software breakpoint to FLASH is done when **FLASH.AUTO** is used
   when FLASH programming is disabled with the **FLASH.Program** command.

Complete example:

```
; Format 2

FLASH.RESet
FLASH.Create 1. 0x0--0x3FFFFF 0x20000 TARGET Word
FLASH.Target 0xA0000000 0xA0001000 0x1000 \
                                    ~~/demo/arm/flash/word/i28f200j3.bin

FLASH.Erase 1.
FLASH.Program 1.
Data.LOAD.Elf my_application.elf /Word
FLASH.Program off

Data.LOAD.Elf my_application.elf /Word /DIFF

IF FOUND()
     Print "Not ok!"
ELSE
     Print "FLASH ok!"
```

Example 3 (Recommended to Test Your Own FLASH Algorithm)

Tips for writing your own FLASH algorithm can be found in **"How to Write your own FLASH Algorithm"** (flash_app_own_algorithm.pdf).

FLASH: Intel Strata FLASH 28F320J3A, 16 bit mode, word access

Memory configuration:

- FLASH from 0x0--0x3FFFFF
- RAM starting at address 0xA0000000

```
; Format 1 using your own FLASH algorithm

FLASH.RESet
FLASH.Create 1. 0x0--0x3fffff 0x20000 TARGET Word

;             <code_range>      <data_range>
FLASH.Target 0xA0000000++0xFFF 0xA0001000++0xFFF

Data.LOAD.Elf my_flash_algorithm.elf

FLASH.Erase ALL
FLASH.Program ALL
Data.LOAD.Binary flash_contents.bin
FLASH.Program off
```

- `<code_range>`
  - The code for `my_flash_algorithm.elf` is downloaded to `<code_range>`
  - Required size for the code is `size_of(my_flash_algorithm.elf) + 32 byte`
- `<data_range>`
  - The specified data range is used for `my_flash_algorithm.elf`
  - `<buffer_size>` is the number of bytes which are transferred from the TRACE32 software to the target controlled FLASH programming routine in one step. Recommended buffer size is 4 KByte, smaller buffer sizes are also possible. The max. buffer size is 16 KByte.
  - `<buffer_size> = size_of(<data_range>) - 32 byte argument buffer - 256 byte stack`

Programming procedure:

1. The FLASH program is loaded to the target by the user.
2. The TRACE32 software saves the used CPU registers:
   - Before the commands **FLASH.Erase**, **FLASH.UNLOCK**, **FLASH.LOCK** are started or
   - When FLASH programming is enabled.
3. The requested FLASH commands are executed.
4. The saved CPU registers are restored:
   - After the commands **FLASH.Erase**, **FLASH.UNLOCK**, **FLASH.LOCK** are finished or
   - When FLASH programming is disabled.

Examples for /DualPort

Example for processor-internal FLASH:

```
;             <code>         <data>       <buffer_size>
FLASH.TARGET 0x40000000    0x40002000      0x1000         *.bin  /DualPort
```

Example for Cortex/ARM using dual-port access via AHB bus:

```
;             <code>        <data>             <buffer_size>
FLASH.TARGET 0x20000000   EAHB:0x20001000      0x1000     *.bin  /DualPort
```

Example for /FirmWareRAM

FLASH declaration of LPC43xx internal FLASH:

```
FLASH.TARGET 0x10000000 0x10001000 0x2000 \
                    ~~/demo/arm/flash/long/lpc4300.bin \
                    /STACKSIZE 0x200 /FirmWareRAM 0x10089FF0--0x10089FFF
```

See also

- FLASH.TARGET2
- FLASH.CFI
- FLASH.TARGET.BUILD()
- FLASH.TARGET.DATARANGE()
- FLASH
- FLASH.state
- FLASH.TARGET.CODERANGE()
- FLASH.TARGET.FILE()

▲ 'FLASH Declaration in Detail' in 'Onchip/NOR FLASH Programming User's Guide'
▲ 'Release Information' in 'Legacy Release History'

Format 1: **FLASH.TARGET2** *<code_address> <data_address> <buffer_size> <file>* \
[**/STACKSIZE** *<size>*] [**/FirmWareRAM** *<address_range>*] \
[**/DualPort** | **/CORE** *<number>*]

Format 2: **FLASH.TARGET2** *<code_range> <data_range>* [*<file>*] \
[**/STACKSIZE** *<size>*] [**/FirmWareRAM** *<address_range>*] \
[**/DualPort** | **/CORE** *<number>*]

Supports the simultaneous programming of multiple flash devices with two different flash algorithms. This allows to program for example processor internal and processor external NOR flash, HyperFlash or QSPI flash with a single programming command sequence.

The command syntax is the same as for the **FLASH.TARGET** command.

STACKSIZE, etc. For a description of the options, see **FLASH.TARGET**.

Example:

```
; set up declaration for off-chip flash 1
FLASH.Create 1. 0x00000000--0x0001FFFF 0x10000 TARGET Long
FLASH.TARGET 0xC0000000 0xD0000000 0x2000 \
                                    ~~/demo/arm/flash/long/am29lv100.bin

; set up declaration for off-chip flash 2
FLASH.Create 2. 0xA4000000--0xA401FFFF 0x4000 TARGET2 Long
FLASH.Create 2. 0xA4020000--0xA43FFFFF 0x20000 TARGET2 Long
FLASH.TARGET2 0xC0000000 0xD0000000 0x2000 \
                                     ~~/demo/arm/flash/long/am29n256.bin
…

; single flash programming sequence for both flash devices
FLASH.ReProgram ALL /Erase
Data.LOAD.Elf my_program.elf
FLASH.ReProgram off
```

See also

- FLASH.TARGET
- FLASH.CFI
- FLASH.TARGET.BUILD()
- FLASH.TARGET2.DATARANGE()
- FLASH
- FLASH.state
- FLASH.TARGET2.CODERANGE()
- FLASH.TARGET2.FILE()

▲ 'Release Information' in 'Legacy Release History'

Format:	**FLASH.UNLOCK** *<unit>* | *<address_range>* | **ALL** | **off**

Many FLASH devices provide a sector/block protection to avoid unintended erasing or programming operations. This protection has to be unlocked in order to erase or program a FLASH device.

<unit>	Unlock the specified unit.
ALL	Unlock all FLASH sectors.
off	(no effect)

Two unlocking schemes are used by FLASH devices:

1. Each individual sector/block has to be unlocked (individual unlocking).
2. The execution of a single unlock command sequence on an address range unlocks the complete FLASH device (parallel unlocking).

Re-locking has to be executed usually sector by sector.

Please refer to the data sheet of your FLASH device, to find out which scheme is used by your FLASH device.

Example for 1 (individual unlocking): INTEL 28F128L18 at address 0x0, connected to the CPU via a 16 bit data bus

```
FLASH.RESet                          ; reset FLASH declaration

FLASH.CFI 0x0 Word                   ; declare FLASH sectors via
                                     ; CFI query

FLASH.UNLOCK ALL                     ; unlock each sector individually

…                                    ; erasing and programming

FLASH.LOCK ALL                       ; re-lock each sector individually
```

Example for 2 (parallel unlocking): INTEL 28F128J3 at address 0x0, connected to the CPU via a 16 bit data bus, each sector 128KByte

```
FLASH.RESet                             ; reset FLASH declaration

FLASH.CFI 0x0 Word                      ; declare FLASH sectors via
                                        ; CFI query

FLASH.UNLOCK 0x0--0x1ffff               ; execute a single unlock command
                                        ; by using an address range
                                        ; inside of a FLASH sector (faster)

…                                       ; erasing and programming

FLASH.LOCK ALL                          ; re-lock each sector individually
```

See also

■ FLASH ■ FLASH.LOCK ■ FLASH.state

▲ 'TRACE32 Tool-based Programming' in 'Tips to Solve NOR FLASH Programming Problems'
▲ 'Programming Commands' in 'Onchip/NOR FLASH Programming User's Guide'

Format: **FLASH.UNSECUREerase**

Debug access is no longer possible if the CPU is secured by programming special FLASH address locations.

In these cases, the **FLASH.UNSECUREerase** command can be used to unsecure a device by sending special commands via the debug interface. As a result, the FLASH is completely erased by the unsecure sequence. Debug access is now possible again.

NOTE: **FLASH.UNSECUREerase** is a CPU specific command.

Example:

```
SYStem.CPU <cpu_type>
FLASH.UNSECUREerase
SYStem.Up
```

See also

- FLASH
- FLASH.state

FLASHFILE Non-memory mapped FLASH devices

Non-memory mapped FLASH memories as NAND FLASH devices and serial FLASH devices can be programmed and erased by the **FLASHFILE** command group.

- For a description of the NAND FLASH programming concept, see **"NAND FLASH Programming User's Guide"** (nandflash.pdf).
- For a description of the Serial FLASH programming concept, see **"Serial FLASH Programming User's Guide"** (serialflash.pdf).
- For a description of the eMMC FLASH programming concept, see **"eMMC FLASH Programming User's Guide"** (emmcflash.pdf).

See also

- FLASHFILE.BSDLaccess
- FLASHFILE.CONFIG
- FLASHFILE.COPYSPARE
- FLASHFILE.Delete
- FLASHFILE.Erase
- FLASHFILE.GETEXTCSD
- FLASHFILE.GETONFI
- FLASHFILE.LOAD
- FLASHFILE.LOADECC
- FLASHFILE.LOCK
- FLASHFILE.PATTERN
- FLASHFILE.RESet
- FLASHFILE.SAVEALL
- FLASHFILE.SAVESPARE
- FLASHFILE.SETEXTCSD
- FLASHFILE.TARGET
- FLASHFILE.UNLOCK
- BSDL.FLASH
- FLASHFILE.BSDLFLASHTYPE
- FLASHFILE.COPY
- FLASHFILE.Create
- FLASHFILE.DUMP
- FLASHFILE.GETBADBLOCK
- FLASHFILE.GETID
- FLASHFILE.List
- FLASHFILE.LOADALL
- FLASHFILE.LOADSPARE
- FLASHFILE.MSYSDLL
- FLASHFILE.ReProgram
- FLASHFILE.SAVE
- FLASHFILE.SAVEECC
- FLASHFILE.Set
- FLASHFILE.SPI
- FLASHFILE.TEST
- FLASH
- FLASHFILE.SPAREADDRESS()

- 'FLASHFILE Functions' in 'General Function Reference'
- 'Introduction' in 'NAND FLASH Programming User's Guide'
- 'Introduction' in 'Serial FLASH Programming User's Guide'

Format:	**FLASHFILE.BSDLaccess ON** | **OFF**

Enables or disables FLASH memory access via boundary scan. The boundary scan chain must be configured with **BSDL.FLASH.IFDefine** and **BSDL.FLASH.IFMap**.

See also

- FLASHFILE

FLASHFILE.BSDLFLASHTYPE Define FLASH type

Format:	**FLASHFILE.BSDLFLASHTYPE** *<flash_family_code>*

Declaration of FLASH memories.

<flash_family_code>	Determines the programming algorithm. Supported flash family codes are listed on **https://www.lauterbach.com/ylist.html** in the **Code** column.

See also

- FLASHFILE

Format 1: NAND FLASH	**FLASHFILE.CONFIG** *<cmd_reg> <addr_reg> <io_reg>*
Format 2: Serial FLASH	**FLASHFILE.CONFIG** *<spi_tx> <spi_rx> <cs_reg> <cs_port>*

NAND FLASH

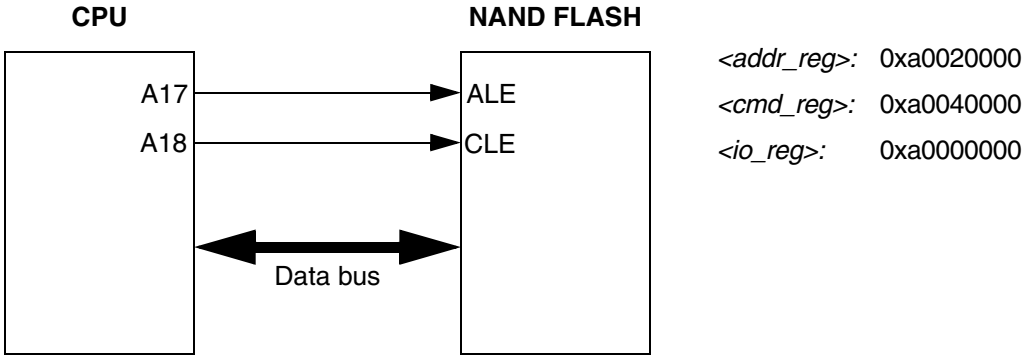
NAND FLASH devices use a common multiplexed bus for data, address and command transfers. With the two signals (Command Latch Enable, Address Latch Enable) the access is latched to the appropriate register. If the two signals are inactive, the access is performed to the I/O register.

Normally CLE and ALE are part of the address on the external bus. Thus NAND FLASHs data, address and command register can be mapped to the processors external memory space.

The user has to define the following three addresses:

<cmd_reg>	Address of the NAND FLASH command register
<addr_reg>	Address of the NAND FLASH address register
<io_reg>	Address of the NAND FLASH I/O register

Example: The NAND FLASH device chip select is assigned to the address 0xa0000000.



For a processor with integrated NAND FLASH controller this command can be omitted. Instead a script for the configuration of such a controller must be executed. Scripts for the supported NAND FLASH controllers are located in the demo directory under `~~/demo/<arch>/flash/<cpu>-nand*.cmm`.

See also

- FLASHFILE

▲ 'Scripts for NAND Flash Programming' in 'NAND FLASH Programming User's Guide'

Format:	**FLASHFILE.COPY** <*source_range*> <*target_address*> [*/*<*option*>]
<*option*>:	**ComPare** | **DIFF** | **wordSWAP** | **LongSWAP** | **QuadSWAP**

The source range data is copied to the defined target address. It is useful to write some memory data to FLASH memory.

<*option*>	For a description of the options, see **FLASHFILE.LOAD.binary**.

Example:

```
; Copy the 2MB virtual memory data at 0x0 to the NAND FLASH address at
0x100000, the writing scheme is the skipped way
FLASHFILE.COPY VM:0x0--0x1FFFFF 0x100000

; Compare the data between virtual memory and NAND FLASH
FLASHFILE.COPY VM:0x0--0x1FFFFF 0x100000 /ComPare
```

See also

- FLASHFILE

FLASHFILE.COPYSPARE Copy to spare area of NAND FLASH

Format:	**FLASHFILE.COPYSPARE** <*source_range*> <*spare_area_address*> [*/*<*option*>]
<*option*>:	**ComPare** | **DIFF** | **wordSWAP** | **LongSWAP** | **QuadSWAP**

The source range data is copied from the TRACE32 virtual memory (VM:) to the defined spare area address of the NAND flash.

<*option*>	For a description of the options, see **FLASHFILE.LOAD.binary**.

Example:

```
; NAND Page Size (example for main/spare: 2KB/64B)
; From the TRACE32 virtual memory, copy 64B to the spare area that
; corresponds to the NAND FLASH main address 0x100000
FLASHFILE.COPYSPARE   VM:0x0--0x3F   FLASHFILE.SPAREADDRESS(0x100000)

;or

;the spare area address 0x8000 corresponds to the main area 0x100000
FLASHFILE.COPYSPARE   VM:0x0--0x3F    0x8000
```

See also

- FLASHFILE
- FLASHFILE.SPAREADDRESS()

Format:	**FLASHFILE.Create** *<address_range> <erase_unit_size> <bus_width>*
<erase_unit_size>:	*<sector>* | *<block>*
<bus_width>:	**Byte** | **Word**

Declaration of FLASH memories.

Generally, the **FLASHFILE.Create** command instructs TRACE32 to correctly calculate the non-uniform flash block size/flash sector size since it has a non-uniform erase unit size.

<address_range>	Address range within the flash memory.
<erase_unit_size>	The erase unit is called *sector* for serial flash and *block* for NAND flash memories. **In case of a NAND flash:** A block equals the size of an ERASE BLOCK of the NAND command (0x60/0xD0). **In case of a Serial flash:** A sector equals the size of an ERASE SECTOR of the Serial flash erase command (0x20/0x40/0xD8).
<bus_width>	The bus width parameter defines the external flash memory bus size (8-bit, 16-bit).

Example: In this script, TRACE32 is informed that the serial flash device S25FL129 has 32 boot sectors, each with a size of 4Kbyte, and 254 main sectors, each with a size of 64KByte.

```
FLASHFILE.Create 0x0--0x1FFFF 0x1000 Byte         ; boot sectors
FLASHFILE.Create 0x20000--0xFFFFFF 0x10000 Byte   ; main sectors
```

See also

- FLASHFILE
- FLASHFILE.List

Format: **FLASHFILE.Delete** *<address_range>* | **ALL**

The specified FLASH block is removed from the FLASH declaration table. Use the command **FLASHFILE.RESet** to clear the whole list and the entire FLASH target configuration.

Example:

```
FLASHFILE.LIST

FLASHFILE.CREATE 0x0--0xFFFFF 0x10000 Byte      ; create 1MByte FLASH
                                                with 16 erase blocks of
                                                64KByte

FLASHFILE.Delete 0x0--0xFFFF                    ; delete the first
                                                64KByte block
```

See also

■ FLASHFILE

FLASHFILE.DUMP Dump FLASH

Format: **FLASHFILE.DUMP** [*<address>* | *<range>*] [*/<format>* | *<option>* ...]

<format>: **NoHex** | **NoAscii**
Byte | **Word** | **Long** | **Quad** | **TByte** | **HByte** | **SByte**
BE | **LE**

<option>: **MAIN**
SPARE
Track
ReFresh
SpotLight

Reads the FLASH memory to the **Data.dump** window. You can read the main and spare data from NAND FLASH devices. You can find the matched spare area easily from the main area dump window by the **Track** option.

See also

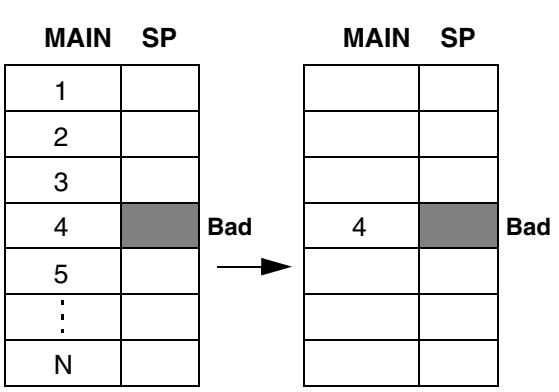
■ FLASHFILE

▲ 'Release Information' in 'Legacy Release History'

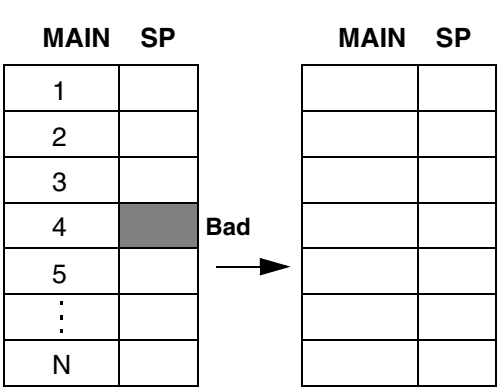
Format:	**FLASHFILE.Erase** *<range>* **/EraseBadBlocks**

Erases the FLASH memory.

EraseBadBlocks	Includes bad blocks in the operation.



FLASHFILE.Erase



FLASHFILE.ERASE /EraseBadBlocks

Examples:

```
;Erase NAND FLASH except bad blocks
FLASHFILE.Erase 0x0--0xFFFF
```

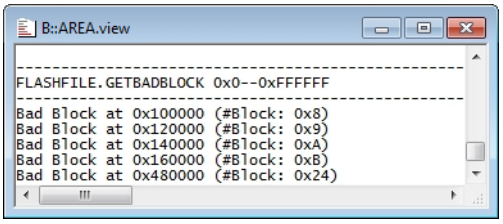
```
;Erase NAND FLASH including bad blocks
FLASHFILE.Erase 0x0--0xFFFF /EraseBadBlocks
```

See also

- FLASHFILE

Format: **FLASHFILE.GETBADBLOCK** *<range>*

Displays the bad block addresses within the specified range. If the NAND FLASH block #0 is bad, then TRACE32 cannot detect any bad blocks in the NAND FLASH device.



See also

- FLASHFILE
- FLASHFILE.GETBADBLOCK.COUNT()
- FLASHFILE.GETBADBLOCK.NEXT()

FLASHFILE.GETEXTCSD Get the extended CSD register

Format: **FLASHFILE.GETEXTCSD**

Gets the extended CSD register, which defines the MMC flash properties and selected modes. Each register byte is described in the JEDEC standard eMMC documentation.

Example:

```
FLASHFILE.GETEXTCSD
AREA.view             ;display the result in the AREA.view window
```

See also

- FLASHFILE
- FLASHFILE.SETEXTCSD

Format: **FLASHFILE.GETID**

Gets the manufacture and device ID from FLASH. This command is mainly used to test the communication between the CPU and FLASH device.

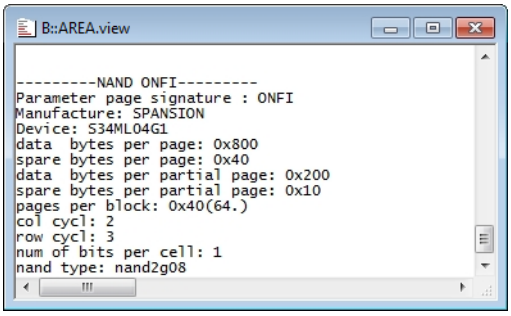
See also

- FLASHFILE

FLASHFILE.GETONFI Display ONFI

Format: **FLASHFILE.GETONFI**

Displays the Open NAND FLASH Interface (ONFI) specification. This command works only for NAND flash devices that support ONFI.



See also

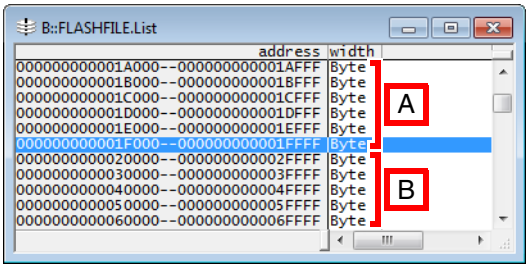
- FLASHFILE

Format: **FLASHFILE.List**

Opens the **FLASHFILE.List** window, displaying the list of blocks or sectors defined with the **FLASHFILE.Create** command. Each block or sector corresponds to one erase unit size of the flash memory.

Example: In this script, TRACE32 is informed that the serial flash device S25FL129 has 32 boot sectors, each with a size of 4Kbyte, and 254 main sectors, each with a size of 64KByte.

```
FLASHFILE.Create 0x0--0x1FFFF 0x1000 Byte          ;boot sectors
FLASHFILE.Create 0x20000--0xFFFFFF 0x10000 Byte    ;main sectors
```



A Each row corresponds to one 4KB boot sector. **B** Each row corresponds to one 64KB main sector.

See also

- FLASHFILE
- FLASHFILE.Create

Using the **FLASHFILE.LOAD** command group, you can load binary, ELF, Intel HEX, and Srecord files to the NAND FLASH.

See also

- FLASHFILE.LOAD.binary
- FLASHFILE.LOAD.Elf
- FLASHFILE.LOAD.IntelHex
- FLASHFILE.LOAD.JSON
- FLASHFILE.LOAD.SPARSE
- FLASHFILE.LOAD.Srecord
- FLASHFILE

FLASHFILE.LOAD.binary Write FLASH

[Examples]

Format:	**FLASHFILE.LOAD.binary** *<file>* [*<address>* *<range>*] **/***<option>*
<option>:	**WriteBadBlocks** | **ComPare** | **ComPareChecKSUM** | **DIFF** | **wordSWAP** | **LongSWAP** | **QuadSWAP** | **SKIP** *<offset>* | **ZIPLOAD** | **NoFF** | **NoZero**

The data from the binary *<file>* is programmed to the flash. If the command is called without *<address>* and *<range>*, the complete file will be programmed to flash beginning from target address 0.

<address>	• File format without address information (e.g. binary): base address of the flash memory • File format with address information: address offset of the flash memory
<range>	• If specified, only the data within the address range will be loaded. Data outside this address range will be ignored. • If specified for file formats without address information, the start address of *<range>* is used as base address and *<address>* will be ignored.
ComPare	Verify the each byte of the file(or data) against the flash memory.
ComPareChecK-SUM	Verify the checksum bytes from the certain size of the file (or data) against the flash memory, faster verification.
DIFF	Compare the contents of the FLASH with the file contents. See example.
LongSWAP	Swaps high and low bytes of a 32-bit word during load.
NoFF	Skips the pages of a *<file>* that contain only 0xFF.

NoZero	Skips the pages of a *<file>* that contain only 0x00. **NOTE**: Pages containing only 0xFF or 0x00 can cause unexpected ECC code generation on the NAND flash spare area. The purpose of the options **NoFF** and **NoZero** is to prevent unexpected ECC code generation by skipping these pages. For information about ECC, see **"NAND FLASH Programming User's Guide"** (nandflash.pdf).
QuadSWAP	Swaps high and low bytes of a 64-bit word during load.
SKIP	If the option **SKIP** *<offset>* is specified, the first *<offset>* bytes of the file are omitted.
wordSWAP	Swaps high and low bytes of a 16-bit word during load.
WriteBadBlock	Program NAND Flash including bad blocks. (Does not apply to serial flash). See example.
ZIPLOAD	Speeds up the transfer of compressed files through the JTAG interface, decompresses and programs the files to the FLASH memory on the target board. Requirements for using **ZIPLOAD**: • SDRAM > 256 KB; SDRAM > 512 KB is recommended. • Configure the CPU clock and SDRAM clock. See example.

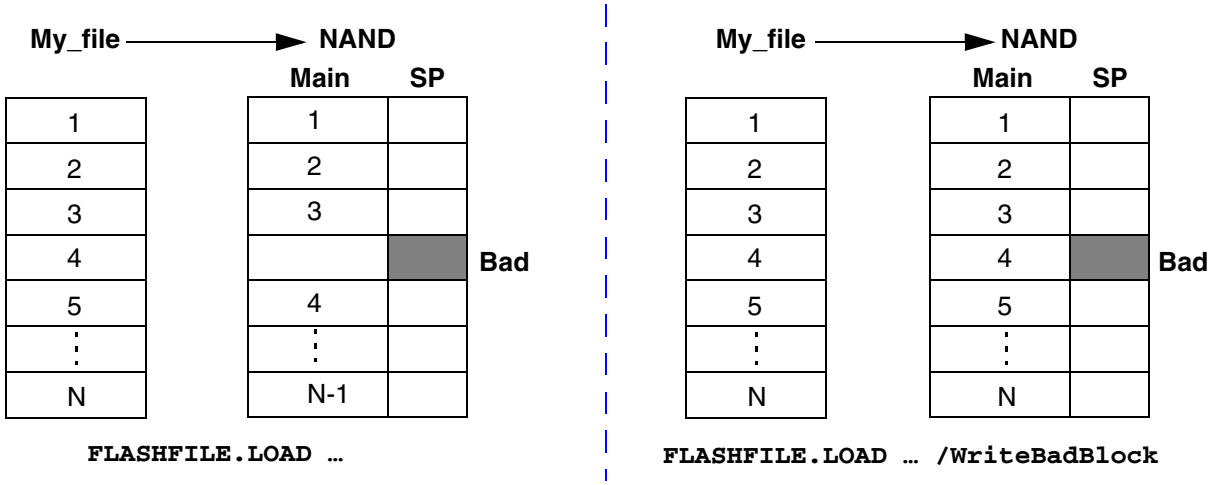
Example 1

```
FLASHFILE.RESet

; FLASHFILE.Config <cmd_reg>   <addr_reg>   <io_reg>
FLASHFILE.Config 0xA0040000 0xA0020000 0xA0000000

; FLASHFILE.TARGET <code_range>     <data_range>        <file>
FLASHFILE.TARGET 0x1000++0x1FFF 0x3000++0x1FFFF \
                                     ~~/demo/arm/flash/word/nand1216.bin

; get the flash identification
FLASHFILE.GETID

; erase first 1 MByte
FLASHFILE.Erase 0++0xFFFFF

; write file to NAND FLASH
FLASHFILE.LOAD my_file.bin 0++0xFFFFF
```

Example 2 - ZIPLOAD option:

```
FLASHFILE.LOAD  *  0x0  /ZIPLOAD  ;the asterisk (*) lets you browse
                                  ;for the compressed file you want
                                  ;to load
```

Example 3 - Illustration of the WriteBadBlock option:



Example 4 - DIFF option:

```
; compare the contents of the NAND FLASH with the file contents
FLASHFILE.LOAD.Elf data.elf /DIFF

; FOUND() returns TRUE if a difference was found
IF FOUND()
   PRINT "Not ok!"
ELSE
   PRINT "NAND FLASH ok!"
```

See also

- FLASHFILE.LOAD

Format:	**FLASHFILE.LOAD.Elf** *<file>* [*<address>* *<range>*] [**/***<option>*]
<option>:	**ComPare** | **DIFF** **wordSWAP** | **LongSWAP** | **QuadSWAP**

Loads an ELF file to the FLASH memory on the target.

<option>	For a description of the parameters and options, see **FLASHFILE.LOAD.binary**.

Example:

```
FLASHFILE.LOAD.ELF *.*  0x20000000  0x0--0xFFFFFF
```

See also

- FLASHFILE.LOAD

FLASHFILE.LOAD.IntelHex Load Intel hex file

Format:	**FLASHFILE.LOAD.IntelHex** *<file>* [*<address>* *<range>*] [**/***<option>*]
<option>:	**ComPare** | **DIFF** **wordSWAP** | **LongSWAP** | **QuadSWAP**

Loads a file in the intel HEX format to the NAND Flash.

<option>	For a description of the parameters and options, see **FLASHFILE.LOAD.binary**.

See also

- FLASHFILE.LOAD

Format:	**FLASHFILE.LOAD.JSON** *<file>* [*<address>* *<range>*] [*/<option>*]
<option>:	**ComPare** | **DIFF** **wordSWAP** | **LongSWAP** | **QuadSWAP**

Loads the json query “`flash_files`” so that TRACE32 can program to the FLASH memory by parsing the json file.

<option>	For a description of the parameters and options, see **FLASHFILE.LOAD.binary**.

Example:

```
;====================flasher_args.json====================
{
...
 “flash_files” : {
        “0x0” : “bootloader/bootloader.bin”,
        “0x10000” : “blink.bin”,
        “0x8000” : “partition_table/partition-table.bin”
 },
...
;=========================================================
```

```
FLASHFILE.ReProgram ALL
FLASHFILE.LOAD.JSON flasher_args.json
FLASHFILE.ReProgram OFF

FLASHFILE.LOAD.JSON flasher_args.json /ComPare

ENDDO
```

See also

- FLASHFILE.LOAD

Format:	**FLASHFILE.LOAD.SPARSE** *<file>* [*<address>* *<range>*] [**/***<option>*]
<option>:	**ComPare** | **DIFF** **wordSWAP** | **LongSWAP** | **QuadSWAP**

Loads SPARSE image.

<option>	For a description of the parameters and options, see **FLASHFILE.LOAD.binary**.

See also

- FLASHFILE.LOAD

Format:	**FLASHFILE.LOAD.S1record** *<file>* [*<address>* *<range>*] [**/***<option>*] **FLASHFILE.LOAD.S2record** *<file>* [*<address>* *<range>*] [**/***<option>*] **FLASHFILE.LOAD.S3record** *<file>* [*<address>* *<range>*] [**/***<option>*]
<option>:	**ComPare** | **DIFF** **wordSWAP** | **LongSWAP** | **QuadSWAP**

Loads a file in the Srecord format to the NAND Flash.

<option>	For a description of the parameters and options see **FLASHFILE.LOAD.binary**.

See also

- FLASHFILE.LOAD

FLASHFILE.LOADALL Load to main area and spare area

Format:	**FLASHFILE.LOADALL** *<file>* [*<address>* *<range>*] [**/***<option>*]
<option>:	**ComPare** | **DIFF** | **WriteBadBlocks** **wordSWAP** | **LongSWAP** | **QuadSWAP**

Loads *<file>* to the main area and the spare area of the NAND flash memory. This is useful for cloning a NAND flash device.

<option>	For a description of the parameters and options see **FLASHFILE.LOAD.binary**.

See also

- FLASHFILE

Format: **FLASHFILE.LOADECC** *<params>* (deprecated)
Use FLASHFILE.SAVEECC + FLASHFILE.LOADSPARE.

See also

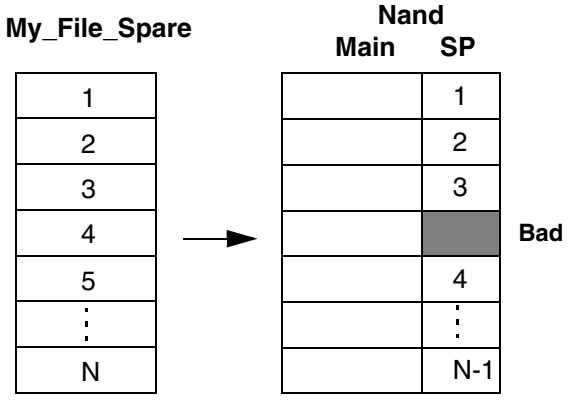
■ FLASHFILE ■ FLASHFILE.SAVEECC

▲ 'Scripts for NAND Flash Programming' in 'NAND FLASH Programming User's Guide'

FLASHFILE.LOADSPARE Write NAND FLASH spare area

Format: **FLASHFILE.LOADSPARE** *<file>* *<range>* **/WriteBadBlocks /ComPare**

The data from *<file>* is written to the spare area on the address and size specified by *<range>*. At first only binary files are supported.



FLASHFILE.LOADSPARE

See also

■ FLASHFILE ❏ FLASHFILE.SPAREADDRESS()

Format: **FLASHFILE.LOCK** *<range>*

Locks the FLASH device.

See also

- FLASHFILE
- FLASHFILE.UNLOCK

Format:	**FLASHFILE.MMC.GETHealth** [**/***<option>*]
<option>:	**VM**

Reports the health status of the eMMC (Embedded Multi-Media Card), allowing user to monitor device usage and determine the lifetime of the eMMC.

This command applies to eMMC memories over version 5.0.

VM	Allows to copy the read out data to VM memory.

FLASHFILE.MSYSDLL Access an M-Systems DiskOnChip flash device

Format:	**FLASHFILE.MSYSDLL** *<file>* [*<cmd>* …]

Accesses an M-Systems DiskOnChip flash device.

See also

- FLASHFILE

FLASHFILE.PATTERN Erase and fill flash memory

Format:	**FLASHFILE.PATTERN** *<addressrange>* [**/***<option>*]
<option>:	**ComPare**

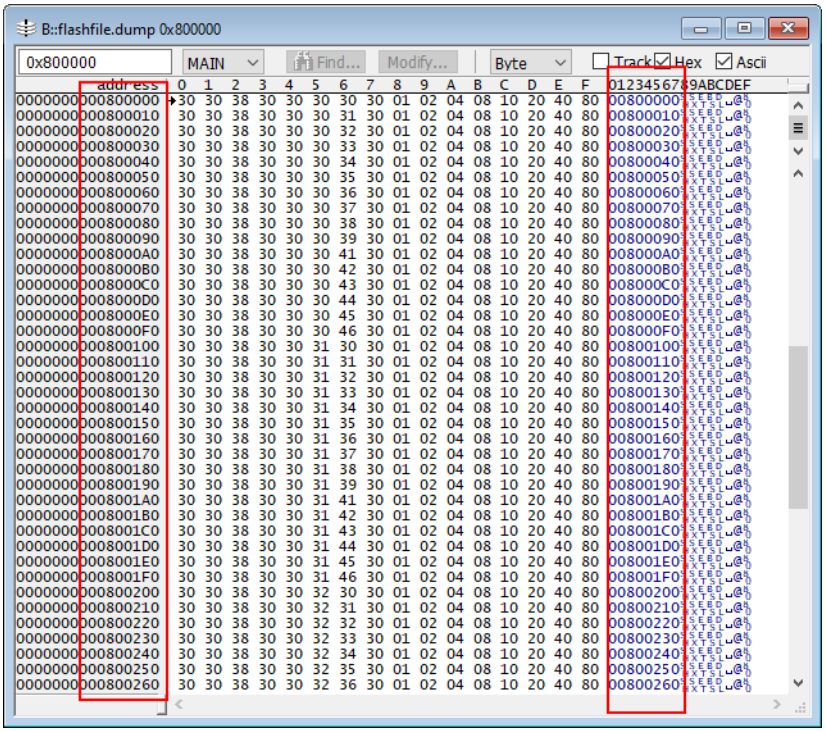
Erases and fills the flash memory with a predefined pattern for the specified address range. The predefined pattern is as follows:

```
(8 byte hexadecimal address) 01 02 04 08 10 20 40 80
```

<addressrange>	Flash block align address for the erase.

Example: This example shows how to test the flash write per flash address.

```
; 1MB erase & fill pattern data
FLASHFILE.PATTERN 0x0--0xfffff

; Display the pattern data in the flash
FLASHFILE.DUMP 0x0

; Compare flash memory against pattern
FLASHFILE.PATTERN 0x0--0xfffff /ComPare
```



See also

- FLASHFILE

FLASHFILE.ReProgram

Re-program FLASH

[build 157321 - DVD 09/2023]

Format: **FLASHFILE.ReProgram** [*<address_range>* | **ALL** | **off** | **CANCEL**]

Activates the FLASH reprogramming mode for the selected address range or for all declared devices. In this mode, all changes are written to the virtual host memory of TRACE32. As soon as **FLASHFILE.ReProgram.off** is executed, the FLASH sectors are programmed but only if their contents have changed.

When using **FLASHFILE.ReProgram**, it is required to define the flash memory (**FLASHFILE.Create**), whether the memory is uniform or non-uniform.

ALL	Activates the reprogramming mode for all FLASH sectors.
off	With parameter “off” or without parameters the FLASH reprogramming mode is terminated. Terminating the FLASH reprogramming mode lets you program only the modified sectors.
CANCEL	Abort without programming pending changes.

Example:

```
; create FLASH block/sector
FLASHFILE.Create 0x0--0xFFFF 0x1000 Byte

; Activate all FLASHs for programming
FLASHFILE.ReProgram ALL

; load the programming file
FLASHFILE.LOAD.Elf data.elf
   ; or
; modify FLASH data
FLASHFILE.Set ...

; Program all modified sectors
FLASHFILE.ReProgram off
```

See also

- FLASHFILE

FLASHFILE.RESet — Reset FLASHFILE declaration within TRACE32

Format: **FLASHFILE.RESet**

The setup for the FLASH device is cleared and set to the default values within TRACE32.

This command does not erase the FLASH sectors.

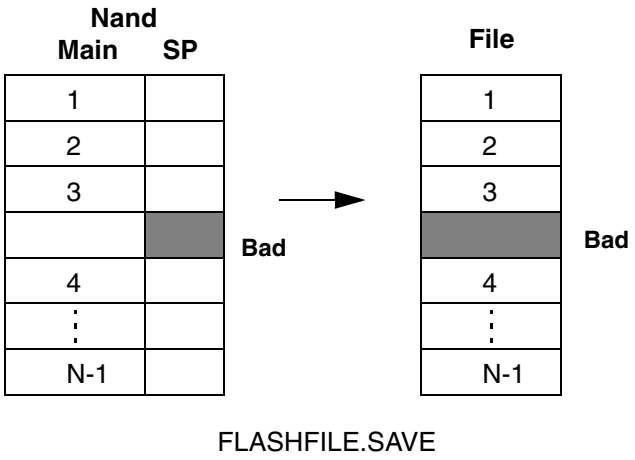
See also

- FLASHFILE

- ▲ ’Scripts for eMMC Controllers’ in ’eMMC FLASH Programming User’s Guide’
- ▲ ’Scripts for NAND Flash Programming’ in ’NAND FLASH Programming User’s Guide’
- ▲ ’Scripts for SPI Controllers’ in ’Serial FLASH Programming User’s Guide’

Format: **FLASHFILE.SAVE** *<file> <range>*

The data whose address and size is specified by *<range>* is read from the NAND FLASH main area and written to *<file>*.



FLASHFILE.SAVE

See also

- FLASHFILE

FLASHFILE.SAVEALL Save the main area and the spare area

Format: **FLASHFILE.SAVEALL** *<file> <range>* [**/SkipBadBlocks**]

Saves the main area and the spare area of the NAND flash memory to *<file>*. This is useful for cloning a NAND flash device.

SkipBadBlocks	Skips bad blocks while saving the contents of the NAND Flash main area to *<file>*.

See also

- FLASHFILE

The **FLASHFILE.SAVEECC** command group is used to generate and save error correction code (ECC) to file. The ECC is generated from the target application residing in the main area of a flash device.

Depending on the **FLASHFILE.SAVEECC.*** command used, TRACE32 generates the ECC with one of the following algorithms:

- Bose-Chaudhuri-Hocquenghem (BCH) algorithm
- Hamming algorithm
- Reed-Solomon algorithm

Use **FLASHFILE.LOADECC** to load ECC files to the spare area of a flash device.

See also

- FLASHFILE.SAVEECC.BCH
- FLASHFILE.SAVEECC.ReedSolomon
- FLASHFILE.LOADECC
- FLASHFILE.SAVEECC.hamming
- FLASHFILE

FLASHFILE.SAVEECC.BCH Save ECC with BCH algorithm

[Example]

Format: **FLASHFILE.SAVEECC.BCH** *<params>* [**/REVERSE** | **/SWAP** | **/LAYOUT** *<layout_block_size> <offset> <bytes_per_block>*]

<params>: *<file> <range> <ref_size> <g_field> <err_correc>*

Saves the error correction code (ECC) to file using the BCH algorithm.

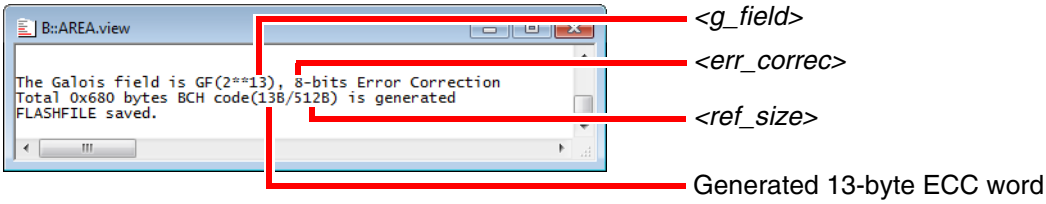
<file>	Destination file for the error correction code (ECC).
<range>	Source memory range for which you want to save the ECC to file.
<ref_size>	Number of bytes that will be represented by an ECC word. **Example**: One ECC word is generated for every `512.` bytes of the *<range>* if the *<ref_size>* is set to `512.` bytes.
<g_field>	Dimension of the Galois field.
<err_correc>	Error correction capability (measured in bit) of the BCH code.

SWAP	Swaps the read byte order (byte access) of each ECC word. **Example** for the 4 byte ECC word 0x12345678: Without **/SWAP** `0x|12|34|56|78|` With **/SWAP** `0x|78|56|34|12|`
REVERSE	Reverses the read bit order (bit access) of each ECC word. **Example**: The ECC word is 0xC1, i.e. 1 byte [7:0]. The **REVERSE** option changes bit 7 to bit 0, bit 6 to bit 1, and so on. Thus, the resulting reversed ECC word is 0x83. Without **/REVERSE** `0x| C | 1` → `0y|1100|0001` With **/REVERSE** `0x| 8 | 3` ← `0y|1000|0011`
LAYOUT	For a description of **LAYOUT**, see **FLASHFILE.SAVEECC.hamming**. For examples for **FLASHFILE.SAVEECC.BCH ... /LAYOUT**, see examples 2 and 3 below.

Examples for FLASHFILE.SAVEECC.BCH

Example 1:

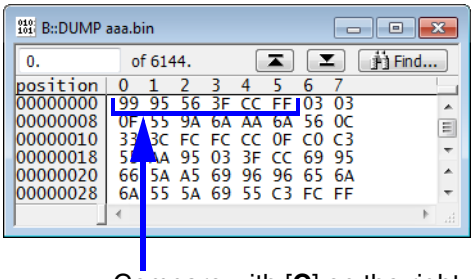
```
;save the ECC file:                <range>   <ref_s.><g_field><err_correc>
FLASHFILE.SAVEECC.BCH file.bch 0x0--0xFFFF   512.     13.         8.

;open an AREA window to view the result
AREA.view

;optionally, view the ECC file with the DUMP command
;for WIDTH use the width of the generated ECC word
DUMP file.bch /WIDTH 13.
```



Example 2:

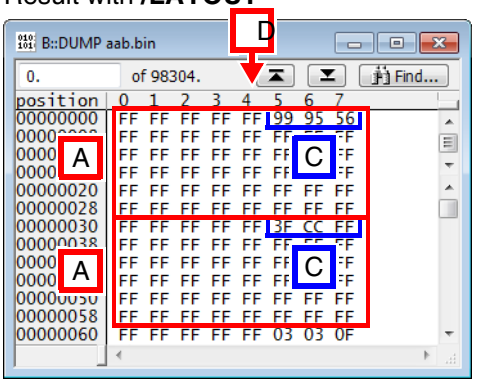
```
;without LAYOUT: ECC in linear form
FLASHFILE.SAVEECC.BCH aaa.bin 0x0--0xFFFFF  512. 13. 8.
DUMP aaa.bin

;with LAYOUT: ECC in non-linear form
FLASHFILE.SAVEECC.BCH aab.bin 0x0--0xFFFFF  512. 13. 8.  \
                                                /LAYOUT 0x30 5. 3.
DUMP aab.bin
```

Result without **/LAYOUT**



Compare with [**C**] on the right.

Result with **/LAYOUT**



A *<layout_block_size>* = 0x30

B *<offset>* = 5

C *<bytes_per_block>* = 3

Example 3:

```
FLASHFILE.SAVEECC.BCH spare-bch.bin 0x0--0xFFFFF  512. 13. 8.   \
                                                  /LAYOUT 0x10 2. 13.
```

See also

- FLASHFILE.SAVEECC

▲ 'Scripts for NAND Flash Programming' in 'NAND FLASH Programming User's Guide'

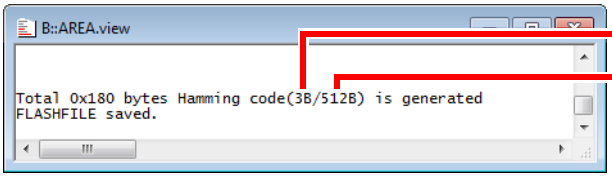
Format: **FLASHFILE.SAVEECC.hamming** *<file> <range> <ref_size>*
[**/SWAP** | **/LAYOUT** *<layout_block_size> <offset> <bytes_per_block>*]

Saves the error correction code (ECC) to file using the Hamming algorithm. By default, the ECC words generated by the Hamming algorithm are fixed (3 bytes). Consequently, you do not need to specify the *<ecc_size>*, but only *<file>*, *<range>*, and *<ref_size>*.

<file>	Destination file for the error correction code (ECC).
<range>	Source memory range for which you want to save the ECC to file.
<ref_size>	Number of bytes that will be represented by an ECC word. **Example**: One ECC word is generated for every `512.` bytes of the *<range>* if the *<ref_size>* is set to `512.` bytes.
SWAP	In the generated 3-byte ECC words, the location of ECC0 and ECC1 is swapped. The location of ECC2 remains fixed. **Example**: Without **/SWAP** **ECC0 ECC1 ECC2** FF 3C 0F 69 AA 96 With **/SWAP** **ECC0 ECC1 ECC2** 3C FF 0F AA 69 96
LAYOUT	Without **LAYOUT**: The error correction code (ECC) is written to file in linear form. With **LAYOUT**: The ECC is written to file in non-linear form and the layout of the ECC is modified by the arguments *<layout_block_size>*, *<offset>*, and *<bytes_per_block>*. See examples 2 and 3 below.

Example 1:

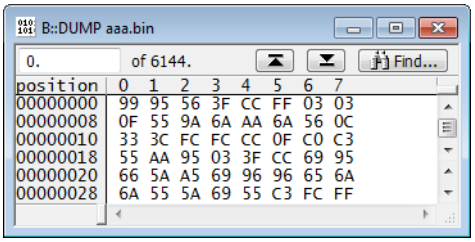
```
;save the ECC file:                          <range>    <ref_size
FLASHFILE.SAVEECC.hamming    file.hmg    0x0--0xFFFF      512.

AREA.view ;open an AREA window to view the result

;optionally, view the ECC file with the DUMP command
;use WIDTH 3. because each ECC word is 3 bytes
DUMP file.hmg /WIDTH 3.
```



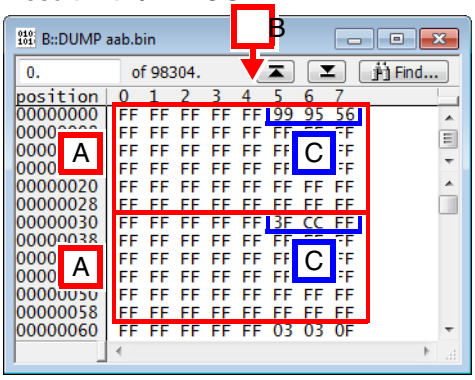
Example 2:

```
;without LAYOUT: ECC in linear form
FLASHFILE.SAVEECC.hamming aaa.bin 0x0--0xFFFFF 512.
DUMP aaa.bin

;with LAYOUT: ECC in non-linear form
FLASHFILE.SAVEECC.hamming aab.bin 0x0--0xFFFFF 512. /LAYOUT 0x30 5. 3.
DUMP aab.bin
```

Result without **/LAYOUT**



Result with **/LAYOUT**



A *<layout_block_size>* = 0x30

B *<offset>* = 5

C *<bytes_per_block>* = 3

Example 3:

```
FLASHFILE.SAVEECC.hamming spare-ham.bin 0x0--0xFFFFF  256. \
                                                /LAYOUT 0x40 40. 24.
```

See also

- FLASHFILE.SAVEECC
- ▲ 'Scripts for NAND Flash Programming' in 'NAND FLASH Programming User's Guide'

Format: **FLASHFILE.SAVEECC.ReedSolomon** *<file> <range>*
[**/LAYOUT** *<layout_block_size> <offset> <bytes_per_block>*]

Saves the error correction code (ECC) to file using the Reed-Solomon algorithm.

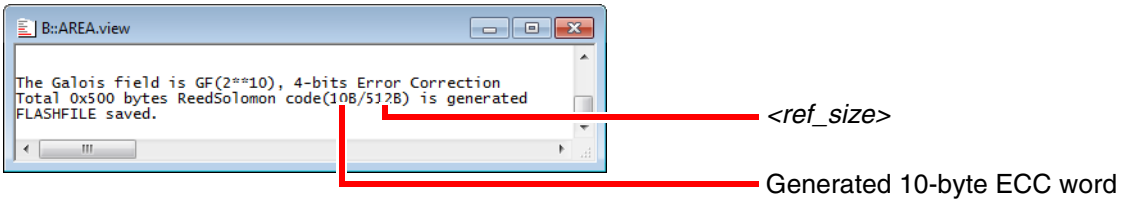
By default, one ECC word is 10 bytes for 4-bits error correction. Consequently, you do not need to specify the *<ecc_size>*, but only *<file>* and *<range>*.

<file>	Destination file for the error correction code (ECC).
<range>	Source memory range for which you want to save the ECC to file.
LAYOUT	For a description of **LAYOUT**, see **FLASHFILE.SAVEECC.hamming**. For examples of **FLASHFILE.SAVEECC.ReedSolomon ... /LAYOUT**, see examples 2 and 3 below.

Examples for FLASHFILE.SAVEECC.ReedSolomon

Example 1:

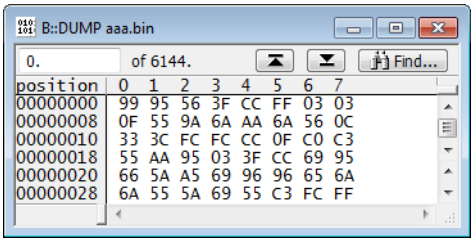
```
;view the result of FLASHFILE.SAVEECC.ReedSolomon in the AREA window
AREA.view

;                                              <range>
FLASHFILE.SAVEECC.ReedSolomon   file.rs    0x0--0xFFFF

;optionally, view the ECC file with the DUMP command
;use WIDTH 10. because each ECC word is 10 bytes
DUMP file.rs /WIDTH 10.
```



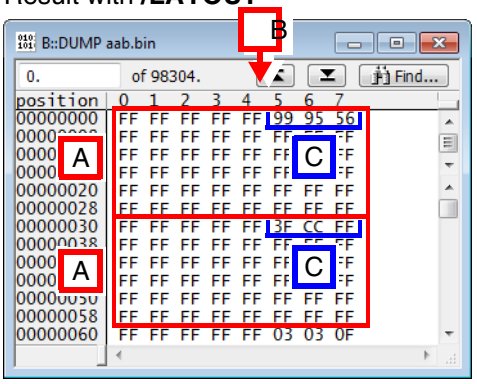
Example 2:

```
;without LAYOUT: ECC in linear form
FLASHFILE.SAVEECC.ReedSolomon   aaa.bin  0x0--0xFFFF
DUMP aaa.bin

;with LAYOUT: ECC in non-linear form
FLASHFILE.SAVEECC.ReedSolomon   aab.bin  0x0--0xFFFF /LAYOUT 0x30 5. 3.
DUMP aab.bin
```

Result without **/LAYOUT**



Result with **/LAYOUT**



A *<layout_block_size>* = 0x30

B *<offset>* = 5

C *<bytes_per_block>* = 3

Example 3:

```
FLASHFILE.SAVEECC.ReedSolomon   spare-rs.bin  0x0--0xFFFF     \
                                                /LAYOUT 0x10  6. 10.
```

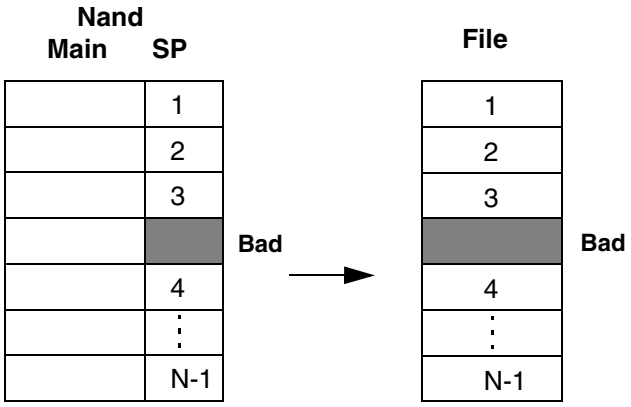
See also

- FLASHFILE.SAVEECC

▲ 'Scripts for NAND Flash Programming' in 'NAND FLASH Programming User's Guide'

Format: **FLASHFILE.SAVESPARE** *<file> <range>*

The data whose address and size is specified by *<range>* is read from the NAND FLASH spare area and written to *<file>*.



FLASHFILE.SAVESPARE

See also

- FLASHFILE
- FLASHFILE.SPAREADDRESS()

FLASHFILE.Set Modify FLASH data

Format: **FLASHFILE.Set** *<address>* | *<range>* **%***<format>* *<data>*

<format>: **Byte** | **Word** | **Long** | **Quad** | …

Modifies the contents of FLASH data at defined *<address>*. The maximum size of modifying is a block size of the flash.

```
; Write 4Bytes data 0x12345678 at the address 0x100000;
FLASHFILE.Set 0x100000 %LE %Long 0x12345678

; Write data 0x0 at the address range 0x100000++0xFFF;
FLASHFILE.Set 0x100000++0xFFF %Byte 0x0
```

See also

- FLASHFILE

Format: **FLASHFILE.SETEXTCSD** *<index> <data>*

Modifies the extended CSD register by using the MMC command CMD6.

Example: Partition switch in the eMMC flash

```
;The Extended CSD register index 179. is the PARTITIOM_CONFIG register.
;The register bit flag describes in the JEDEC standard eMMC
;documentation.

FLASHFILE.SETEXTCSD  179. 0x00   ; access: partition null, no boot,
                                 ; access: no boot partition

FLASHFILE.SETEXTCSD  179. 0x49   ; access: partition boot 1

FLASHFILE.SETEXTCSD  179. 0x4A   ; access: partition boot 2
```

See also

- FLASHFILE
- FLASHFILE.GETEXTCSD

See also

- FLASHFILE

FLASHFILE.SPI.CFI Generate SPI FLASH sector declaration by CFI

Format: **FLASHFILE.SPI.CFI** [*/<option>*]

<option>: **QPI**

Generates the FLASH declaration (**FLASHFILE.List**) by using the CFI information stored in a non-memory mapped SPI FLASH device.

QPI This option is needed when the SPI FLASH is in QPI mode.

[Examples]

Format:	**FLASHFILE.SPI.CMD** [**%***<format>*] *<data>* … [**/***<option>*] **FLASHFILE.SPICMD** (deprecated)
<format>:	**Byte** | **Word** | **Long** | …
<option>:	**READ** *<number_of_bytes>* [*<address>*] **QPI**

Sends FLASH-device-specific *<data>* from TRACE32 to a SPI FLASH device. The *<data>* can be any valid FLASH command sequence. For *<data>* that can be sent, refer to the FLASH manufacturer's documentation.

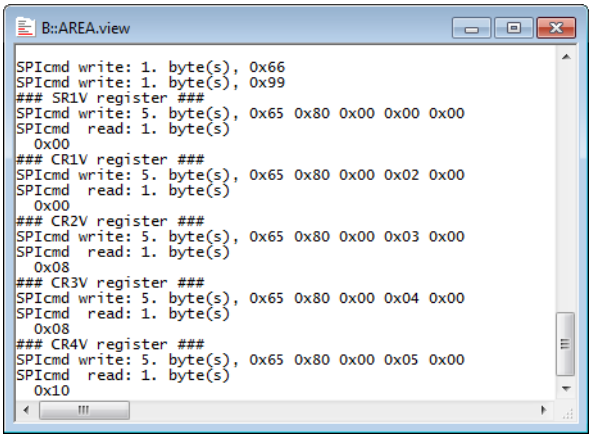
Byte (default), **Word**, **Long**, …	Data size for integer constants. See **"Keywords for <width>"** (general_ref_d.pdf).
<data>	Any valid FLASH command sequence you want to send to the SPI FLASH, such as commands, addresses, dummy cycles, etc. Examples of SPI-FLASH-device commands: • **0x9F** // the READ ID command • **0x03 0x00 0x10 0x00** // the READ command with arguments 0x3 is the READ command. The three arguments 0x00, 0x10, and 0x00 stand for the SPI FLASH address 0x001000.
READ *<number_of_bytes>*	TRACE32 requests the SPI FLASH to read the specified *<number_of_bytes>*, and then prints the bytes read by the SPI FLASH to the **AREA.view** window.
READ *<number_of_bytes>* *<address>*	TRACE32 requests the SPI FLASH to read the specified *<number_of_bytes>*, and then sends the bytes read by the SPI FLASH to the specified *<address>*; typically an address in the TRACE32 virtual memory (VM:). See example 1.
QPI	Execute SPI FLASH access in 4-4-4 QPI mode.

Example 1: The SPI FLASH reads the *<number_of_bytes>*, and then TRACE32 sends the bytes read by the SPI FLASH to the specified *<address>* in the TRACE32 virtual memory (VM:).

```
;TRACE32 transmits the command 0x9f (= read ID), and the host receives 4
;bytes of data. Then TRACE32 sends the data received from the host to
;the TRACE32 virtual memory address 0x0.
FLASHFILE.SPI.CMD 0x9f /READ 4. VM:0x0

&data=Data.LONG(VM:0x0)
PRINT "SPI data : 0x" &data //print the FLASH Manufacture and Device ID
```

Example 2: This script shows how to output the contents of internal SPI FLASH registers to the **AREA.view** window:

```
;Flash name: S25FS series (Spansion SPI Flash memory)

SCREEN.OFF
FLASHFILE.SPI.CMD 0x66  ;enable the SPI FLASH software reset
FLASHFILE.SPI.CMD 0x99  ;perform the SPI FLASH software reset
SCREEN.ON

;if the SPI FLASH is the 3-byte address mode
PRINT "### SR1V register ###"
FLASHFILE.SPI.CMD 0x65 0x80 0x00 0x00 0x00    /READ 0x1

PRINT "### CR1V register ###"
FLASHFILE.SPI.CMD 0x65 0x80 0x00 0x02 0x00    /READ 0x1

PRINT "### CR2V register ###"
FLASHFILE.SPI.CMD 0x65 0x80 0x00 0x03 0x00    /READ 0x1

PRINT "### CR3V register ###"
FLASHFILE.SPI.CMD 0x65 0x80 0x00 0x04 0x00    /READ 0x1

PRINT "### CR4V register ###"
FLASHFILE.SPI.CMD 0x65 0x80 0x00 0x05 0x00    /READ 0x1

;let's open the AREA window to view the result
AREA.view
```



FLASHFILE.SPI.GETSFDP Read FLASH discovery parameters

Format:	**FLASHFILE.SPI.GETSFDP** *<unit>* | *<range>* [**%***<format>*] *<data>* …
<format>:	**Byte** | **Word** | **Long** | …

Reads Serial Flash Discovery Parameters from SPI FLASH. Several parameters values can be afterwards.

FLASHFILE.SPI.RESetMemory Reset volatile register values

[build 134704 - DVD 09/2021]

Format:	**FLASHFILE.SPI.RESetMemory**

Resets all volatile register values.

Format: **FLASHFILE.TARGET** *<code_range> <data_range>* [*<file>*] [**/DualPort** | **/KEEP**]

Specifies the FLASH programming algorithm and where it runs in the target RAM.

<code_range> defines the address range in the target memory where the code of the FLASH programming algorithm is downloaded to.

required size for the code is `size_of(file) + 32 byte`

FLASH algorithm
32 byte

Memory mapping for the *<code_range>*

<data_range> specifies the address range in the target memory where data buffer, argument buffer and stack are placed.

```
<data_buffer_size> =
size_of(<data_range>) - 64 byte argument buffer - 256 byte stack
```

<data_buffer_size> is the number of bytes which are transferred from the TRACE32 software to the FLASH programming routine in one step. Recommended buffer size is 4 KByte, smaller buffer sizes are also possible. The max. buffer size is 16 KByte.

64 byte argument buffer
Data buffer for data transfer between TRACE32 and FLASH algorithm `<buffer_size>` calculated as described above
256 byte stack

Memory mapping for the *<data_range>*

<file> specifies the FLASH programming algorithm file.

DualPort (for FLASHFILE.TARGET)

DualPort can be used for target controlled FLASH programming. While FLASH programming is being executed, the TRACE32 host software simultaneously downloads the next command and data via dual-port memory access into the target RAM. **DualPort** can reduce the programming time significantly.

The **DualPort** option can only be used together with the memory class **E:** for *<data_range>*:

```
;                 <code_range>        <data_range>
FLASHFILE.TARGET  0x20000000++0x1FFF  EAHB:0x20002000++0x4FFF  \
~~/demo/arm/flash/byte/nand2g08_cortexm3.bin  /DualPort /KEEP
```

KEEP (for FLASHFILE.TARGET)

After FLASH programming, the *<code_range>* and *<data_range>* on the target remain (=**KEEP**) reserved for FLASH programming. TRACE32 does *not* restore the original data that existed in *<code_range>* and *<data_range>* before the FLASH programming was executed.

So TRACE32 can save the restore overhead, for example, when **FLASHFILE** commands are executed within a large *<data_range>* or when **FLASHFILE** commands are executed frequently, such as the **FLASHFILE.DUMP** command.

See also

- FLASHFILE

- ▲ 'Scripts for eMMC Controllers' in 'eMMC FLASH Programming User's Guide'
- ▲ 'Scripts for NAND Flash Programming' in 'NAND FLASH Programming User's Guide'

FLASHFILE.TEST Non-memory mapped FLASH test

Format: **FLASHFILE.TEST** *<address_range>* [*/<option>*]

Performs an integrity test of the non-memory mapped FLASH in the specified *<address_range>* and prints a message indicating success or failure of the test.

<address_range> The start and end addresses have to be aligned to the FLASH block size, otherwise the error message "test address range is wrong" is printed.

Prime	The defined range is completely filled with a test pattern and is subsequently verified. **NOTE**: The length of the test pattern is a prime, but not the data itself. Original memory contents are lost. This test detects address line failures or mirrored partitions within a memory.
RANDOM	Pattern is a random sequence.
PRANDOM	Pattern is a pseudo random sequence.
Repeat	the test is repeated, in case there are no parameters, until it is stopped manually.

Examples:

```
FLASHFILE.TEST 0x0--0xFFFF /Prime  ;test flash address 0x0--0xFFFF(64KB)
```

```
FLASHFILE.TEST 0x200000++0xFFFFF /Prime /Repeat 10.  ;test 10 times,1MB
size from the flash address 0x200000
```

See also

- FLASHFILE
- Data.Test

FLASHFILE.UNLOCK — Unlock FLASH device

Format: **FLASHFILE.UNLOCK** <*range*>

Unlocks a FLASH device.

Some FLASH devices are locked after power-up. These devices have to be unlocked in order to erase or program the FLASH device.

See also

- FLASHFILE
- FLASHFILE.LOCK

FPU — Access to FPU registers

In order to access the FPU (Floating-Point Unit) state, the **FPU.ON** command must be set.

Standalone
Emulation probes which support standalone FPU operation (socket inside the emulation probe), this FPU must be activated by the **SYStem.Option.FPU ON** command first. This option must not be set, if an FPU in the target is used.

OS Awareness
The FPU's registers are **not** a part of the tasks, rather, are jointly available only once to all tasks. ’

HLL
HLL-debugging with the FPU is fully supported, i.e. register variables in FPU and stack back traces through FPU stacks are supported.

Assembler Stepping with 68881/68882
When stepping floating point commands, it is possible, that the FPU operation is stopped by the single step during the execution of the operation. This state is indicated by the message 'Mid-Instruction Break' in the FPU window. The displayed PC value may be wrong in this case. The FPU operation will be completed with the next assembler step.

See also

- FPU.Init
- FPU.OFF
- FPU.ON
- FPU.RESet
- FPU.Set
- FPU.TARGET
- FPU.view
- FPU()
- FPUCR()

▲ ’FPU Functions (Floating Point Unit)’ in ’General Function Reference’

FPU.Init — Initialize FPU registers

Format: **FPU.Init**

Sets the FPU registers to their default values.

See also

- FPU
- FPU.view

FPU.OFF — FPU access off

Format: **FPU.OFF**

All other FPU commands are locked. The FPU in the target is not accessed.

See also

- FPU
- FPU.view

FPU.ON — FPU access on

Format: **FPU.ON**

The FPU register set is activated. The FPU must be either in the target or in the emulation-probe. FPU registers are handled similarly to CPU registers (stack tracing, HLL register variables, etc.).

See also

- FPU
- FPU.view

FPU.RESet — Reset command

Format: **FPU.RESet**

The access to the FPU is turned off.

See also

- FPU
- FPU.view

FPU.Set — Modify FPU registers

Format: **FPU.Set** *<register>* [*<expression>* | *<float>*]

The modification can either set floating-point values, or hexadecimal values.

Examples:

```
FPU.Set fpiar 0x1000          ; set FPIAR
FPU.Set fp0 1.23              ; FP0 on 1.23
FPU.Set fp0 1 2 3 4           ; the upper most bytes from
                              ; FP0 to 01 02 03 04
```

See also

- FPU
- FPU.view

FPU.TARGET — Define FPU access agent

Format: **FPU.TARGET** *<code_range>* [*<data_range>*]

Defines the code and data address ranges used by the target agent to read or write FPU registers. Both ranges must define RAM areas that can be used for code or data access. This command is only required for the ARM9 vfp.

See also

- FPU
- FPU.view

FPU.view — Display FPU registers

Format: **FPU.view** [*/<option>*]

<option>: **RAW** (Armv8 only)

All visible FPU registers are displayed in a window. The contents of this window are strongly dependent on the CPU type.

RAW (Armv8 only) — Displays the hexadecimal (raw) interpretation of the FPU registers without floating point view.

See also

- FPU
- FPU.Init
- FPU.OFF
- FPU.ON
- FPU.RESet
- FPU.Set
- FPU.TARGET
- FPU()
- FPUCR()

▲ 'Release Information' in 'Legacy Release History'

Frame — Call-tree and context

Using the **Frame** command group, you can display the call-tree-hierarchy including parameters and local variables belonging to the previous levels (frames). The Up/Down buttons allow the user to see CPU context of previous frames which is then used to investigate and find the exact location of a software-bug.

See also

- Frame.CONFIG
- Frame.COPY
- Frame.Down
- Frame.GOTO
- Frame.Init
- Frame.REDO
- Frame.SkipFunc
- Frame.SkipLine
- Frame.SWAP
- Frame.TASK
- Frame.UNDO
- Frame.Up
- Frame.view
- sYmbol.List.FRAME

▲ 'Release Information' in 'Legacy Release History'

Frame.CONFIG — Fine-tune stack unwinding

Using the **Frame.CONFIG** command group, the user can optimize the results of the frame command by turning on/off compiler-generated frame-unwind rules or other parts of the frame algorithm.

See also

- Frame.CONFIG.Asm
- Frame.CONFIG.EABI
- Frame.CONFIG.EPILOG
- Frame.CONFIG.PROLOG
- Frame.CONFIG.RELOAD
- Frame.CONFIG.SignalHandler
- Frame.CONFIG.sYmbol
- Frame
- Frame.view

Frame.CONFIG.Asm — Frame back-trace mode

Format: **Frame.CONFIG.Asm** [**ON** | **OFF**]
sYmbol.ASMFRAME (deprecated)

Default: ON.

If this option is set, the system will try to trace back over assembler generated frames and frames without debug information (command **Frame.view**). This back trace will fail, if the stack is modified by a function in a 'non standard' way.

See also

- Frame.CONFIG

Format: **Frame.CONFIG.EABI** [**ON** | **OFF**]

Default: OFF.

When turned on, this command forces the stack unwinding algorithm to assume that the frame pointers of each call are chained with each other. The stack unwinding algorithm then reads the previous frame pointer and the return address consecutively from the current frame pointer address. The user should only use this command if he is sure that the architecture supports frame pointers and the compiler has chained all the frame pointers.

See also

- Frame.CONFIG

Frame.CONFIG.EPILOG Use epilog code for frame display

Format: **Frame.CONFIG.EPILOG** [**MaxInstr** *<value>* | **ON** | **OFF** | **UNRESTRICTED** | **RESTRICTED**]
sYmbol.FRAME.EPILOG (deprecated)

Default: ON.

When turned on, the epilog of each function is run through the simulator to find the return address thus the calling function.

MaxInstr *<value>*	Specify the maximum number of simulated instructions used to find the return address in the Frame window.
ON	The simulator is used to find the return address.
OFF	The simulator is not used for stack unwinding.
RESTRICTED	The Simulator does not follow indirect jumps.
UNRESTRICTED	The Simulator follows indirect jumps.

See also

- Frame.CONFIG

Frame.CONFIG.PROLOG — Use prolog code for frame display

Format: **Frame.CONFIG.PROLOG** [**ON** | **OFF**]
sYmbol.FRAME.PROLOG (deprecated)

Default: ON.

When turned on, the prolog of each function is analyzed and the effect of the prolog is reversed to find the calling function.

See also

- Frame.CONFIG

Frame.CONFIG.RELOAD — Generate frame information again

Format: **Frame.CONFIG.RELOAD**
sYmbol.FRAME.Reload (deprecated)

The PowerView assumes the stability of the executable code and analyzes it only once. This command forces it to reanalyze the executable code. It is useful when the executable code is modified during a session and the Frame results become corrupted.

See also

- Frame.CONFIG

Frame.CONFIG.SignalHandler — Stack unwinding

[build 135371 - DVD 09/2021]

Format: **Frame.CONFIG.SignalHandler** *<addressrange> <name>*

Unwinds the Stack inside a Signal Handler if the SignalHandler has been declared with **Frame.CONFIG.SignalHandler**.

See also

- Frame.CONFIG

Format: **Frame.CONFIG.sYmbol** [**ON** | **OFF**]
sYmbol.FRAME.sYmbol (deprecated)

Default: ON.

This command is an on-and-off switch for debug information from the debug file to find the calling function.

ON	Use the symbolic debug information the compiler generated during compile time (if any).
OFF	Use a platform-dependent built-in algorithm to make an educated and sophisticated “guess” about frame size and layout.

See also

- Frame.CONFIG

Format: **Frame.COPY**
Register.COPY (deprecated)

The processor register set is copied to the TRACE32-internal register set.

This command is useful, when the current register values need to be saved and restored later. The TRACE32-internal register set can be copied back by the **Frame.SWAP** command.

See also

■ Frame ■ Frame.view

Frame.Down Show state one level down in stack nesting

Format: **Frame.Down**
Register.Down (deprecated)

The registers are set to the values they had one subroutine nesting level deeper. This command is used after the command **Frame.Up** to restore the original register values.

See also

■ Frame ■ Frame.view ❑ FLAG.READ() ❑ FLAG.WRITE()

Frame.GOTO Change source code view temporarily

Format: **Frame.GOTO** [*<address>*]
Register.GOTO (deprecated)

Sets the PC to the specified address in order to change the view of the source listing (**Data.List**).

The current PC value is saved and will be restored before the application is started again.

See also

■ Frame ■ Frame.view

Format: **Frame.Init**

Resets the CPU registers to their default values after a processor reset. Registers that are undefined after a reset are set to zero.

CPU	Behavior of Frame.Init

ARC	`STATUS  <= 0x02000000`
	`STATUS32  <= 0x00000001`
	`DEBUG  <= 0x11000000`
	`IRQ_LV12  <= 0x00000002` (Resets any interrupt flags)
	`IENABLE  <= 0xffffffff`
	`SEMAPHORE <= 0x00000000`
	
	All other registers are set to zero.
	
	If **SYStem.Option.ResetDetection** is used with a semaphore bit (**Sem0**...**Sem3**), **Frame.Init** sets the corresponding semaphore bit in the SEMAPHORE register.

ARM, Cortex, XScale	**ARM7/9/10/11, Cortex-A/R, XScale:** Rx = 0 SPSRx = 0x10 CPSR = 0xd3 (ARM7/9/10, XScale), 0x1d3 (ARM11, Cortex-A/R) R15 (PC) = 0, 0xffff0000 if high exception vectors selected **Cortex-M:** Rx = 0 R15 (PC) = [vector table base + 4] xPSR = 0x01000000 MSP = [vector table base + 0] PSP = 0 R13 (SP) = MSP or PSP depending on the mode
C166	The CP is set to 0xFC00. The sixteen registers R0 - R15 are set to 0x0. DPP0 = 0x0, DPP1 = 0x1, DPP2 = 0x2 and DPP3 = 0x3. Stack registers STKUN is set to 0xFC00 and STKOV is set to 0xFA00. The Stack Pointer SP is set to 0xFC00 The Instruction Pointer IP is set to zero. The Code Segment Pointer CSP and the VECSEG are set to the initial value after **SYStem.Mode Up**. All other registers are set to zero.
CEVA-X	MODA and MODA shadow register are set to 0x1E. All other registers are set to zero.

CPU	Behavior of Frame.Init
DSP56K	**Family 56000 and 56100** The eight 16-bit modifier registers M[0-7] are set to 0xFFFF. This specifies linear arithmetic as the default type for address register update calculations.The Operating Mode Register (OMR) is set to the initial value after **SYStem.Mode Up**. Values of bits MA, MB and MC of the OMR register are preserved.The program counter is set to zero. All interrupts are masked by setting the Status Register (SR) to 0x300. **Family 56300 and 56720 Dualcore** The eight 24-bit modifier registers M[0-7] are set to 0xFFFFFF. This specifies linear arithmetic as the default type for address register update calculations. The Operating Mode Register (OMR) is set to the initial value after **SYStem.Mode Up**. Values of bits MA, MB, MC and MD of the OMR register are preserved. All interrupts are masked by setting Status Register (SR) to 0x300. The program counter is set to zero. **Family 56800 and 56800E** The eight 16-bit modifier registers M[0-7] are set to 0xFFFF. This specifies linear arithmetic as the default type for address register update calculations. The Operating Mode Register (OMR) is set to the initial value after **SYStem.Mode Up**. Values of bits MA and MB of the OMR register are preserved. All interrupts are masked by setting Status Register (SR) to 0x300. The program counter is set to zero.
HCS08	The Program Counter is set to the value read at 0xFFFE. The Stack Pointer SP is set to 0xFF and the CCR is set to 0x68. All other registers are set to zero.
HC11	The Program Counter is set to the value read at 0xFFFE. The Stack Pointer SP is set to a default value dependent on the derivative. The CCR is set to 0xD8. All other registers are set to zero.
HC12 / S12 / S12X	The Program Counter is set to the value read at 0xFFFE. The CCR is set to 0xD8. All other registers are set to zero.
MIPS32 / MIPS64 /NEC-VR	Program Counter, Status register and Config register are set to their initial values after reset (read during **SYStem.Mode Up**). PRID and Cause register are updated, all other registers are set to zero.
MMDSP	Sets all registers to their initial value after a reset. This is done via a **soft reset** of the core. **NOTE**: This may have effects besides updating the contents of architectural registers.
PowerPC	Program counter and MSR are set to their initial values after reset (read during **SYStem.Mode Up**). GPRs and SPR appearing in the Register window are set to zero.
Microblaze	All registers are set to zero.
PCP	All registers are set to zero.

CPU	Behavior of Frame.Init
Teak	MOD0 and MOD0S registers SATA bit is set. MOD1 and MOD1S registers CMD bit is set. MOD3 and MOD3S registers CREP, CPC and CCNTA bits are set. All other registers are set to zero.
Teaklite / Teaklite-II /Oak	All registers are set to zero.
Teaklite-III	MOD2 register SATA and SATP bits are set. All other registers are set to zero.
x86	EDX is set to a cpu specific value defining the family/model/stepping of the core if a **SYStem.Mode Up** has been executed at some point before, otherwise EDX is set to 0. EAX,EBX,ECX,ESI,EDI,ESP,EBP are set to 0. EIP is set to 0xFFF0 and EFLAGS to 2. CR0 is set to 0x60000010 and CR2-4 to 0. DR0-3 are set to 0. DR6 to 0xFFFF0FF0 and DR7 to 0x400. IDT and GDT: Base = 0 and Limit = 0xFFFF. LDT and TR: Selector = 0, Base = 0, Limit = 0xFFFF, Access = 0x82. CS: Selector = 0xF000, Base = 0xFFFF0000, Limit = 0xFFFF, Access = 0x93. DS,ES,FS,GS,SS: Selector = 0, Base = 0, Limit = 0xFFFF, Access = 0x93. **NOTE**: In a multicore system the above holds for the main bootstrap processor. For the other processors the following differences apply: EIP is set to 0x10000 and CR0 to 0x10.
TMS320	All registers except SSR, IER and TSR are set to zero.
TriCore	All registers except PC, PSW, ICR and BTV are set to zero.
XTENSA	Program Counter (PC), Program Status (PS) and dependent on the XTENSA configuration some other Special Registers are read from the CPU. All other registers are set to zero.
ZSP	All registers are set to zero.

Example:

```
B::                             ; example for the debugger

…

SYStem.Up                       ; establish the communication between the
                                ; processor and the debugger

Frame.Init                      ; initialize the general purpose registers

…
```

See also

- Frame
- Frame.view

Format: **Frame.REDO**
Register.REDO (deprecated)

Recovers the state of the registers before the last **Frame.UNDO** command was executed.

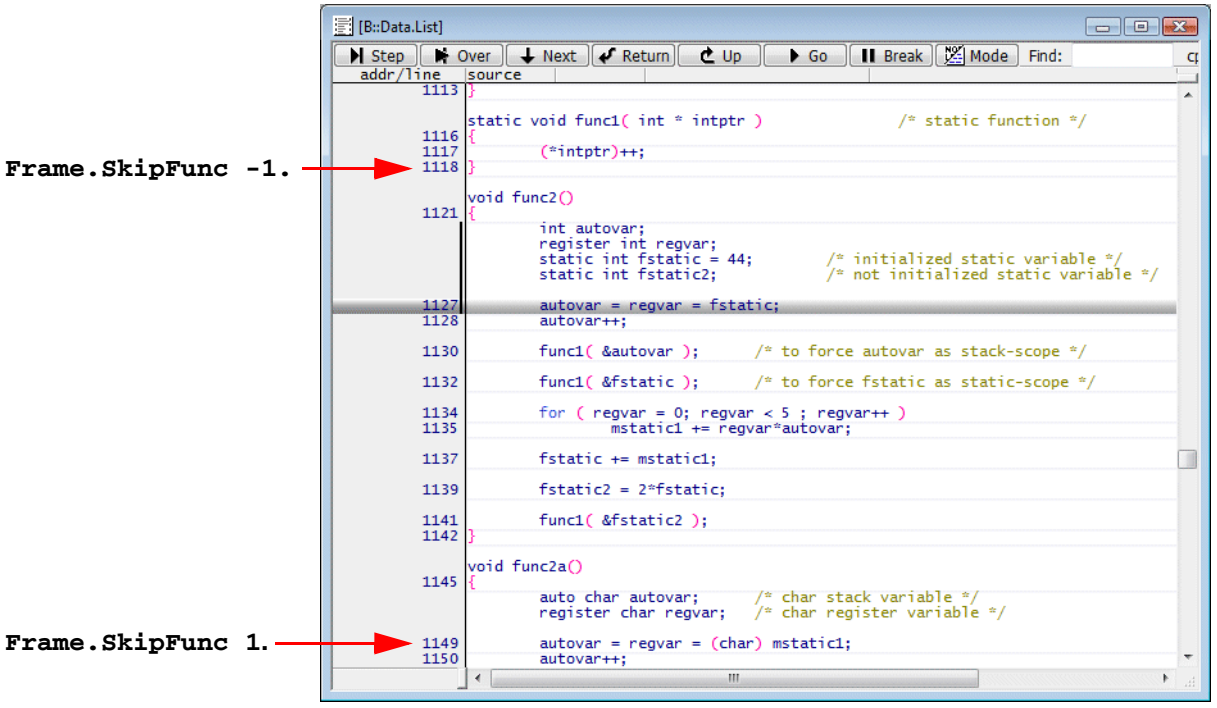
See also

- Frame
- Frame.view

Frame.SkipFunc Change view to previous/subsequent function

Format: **Frame.SkipFunc** *<number>*
Register.SkipFunc (deprecated)

Sets the PC temporarily to one of the subsequent (positive number) or previous (negative number) functions in the source. The current PC value is saved and will be restored before the application is started again.



See also

- Frame
- Frame.view

Format: **Frame.SkipLine** *<number>*
Register.SkipLine (deprecated)

Sets the PC temporarily to one of the subsequent (positive number) or previous (negative number) HLL lines in the source. The current PC value is saved and will be restored before the application is started again.

Example:

```
Register.SkipLine +3.                    ; set PC 3 HLL line forward
```

See also

- Frame
- Frame.view

Frame.SWAP Swap TRACE32 registers

Format: **Frame.SWAP**
Register.SWAP (deprecated)

The processor register set and TRACE32-internal register sets are swapped. This command is useful when an operating system-call or HLL-function call fails to restore the original registers.

See also

- Frame
- Frame.view

Frame.TASK Change view to specified task

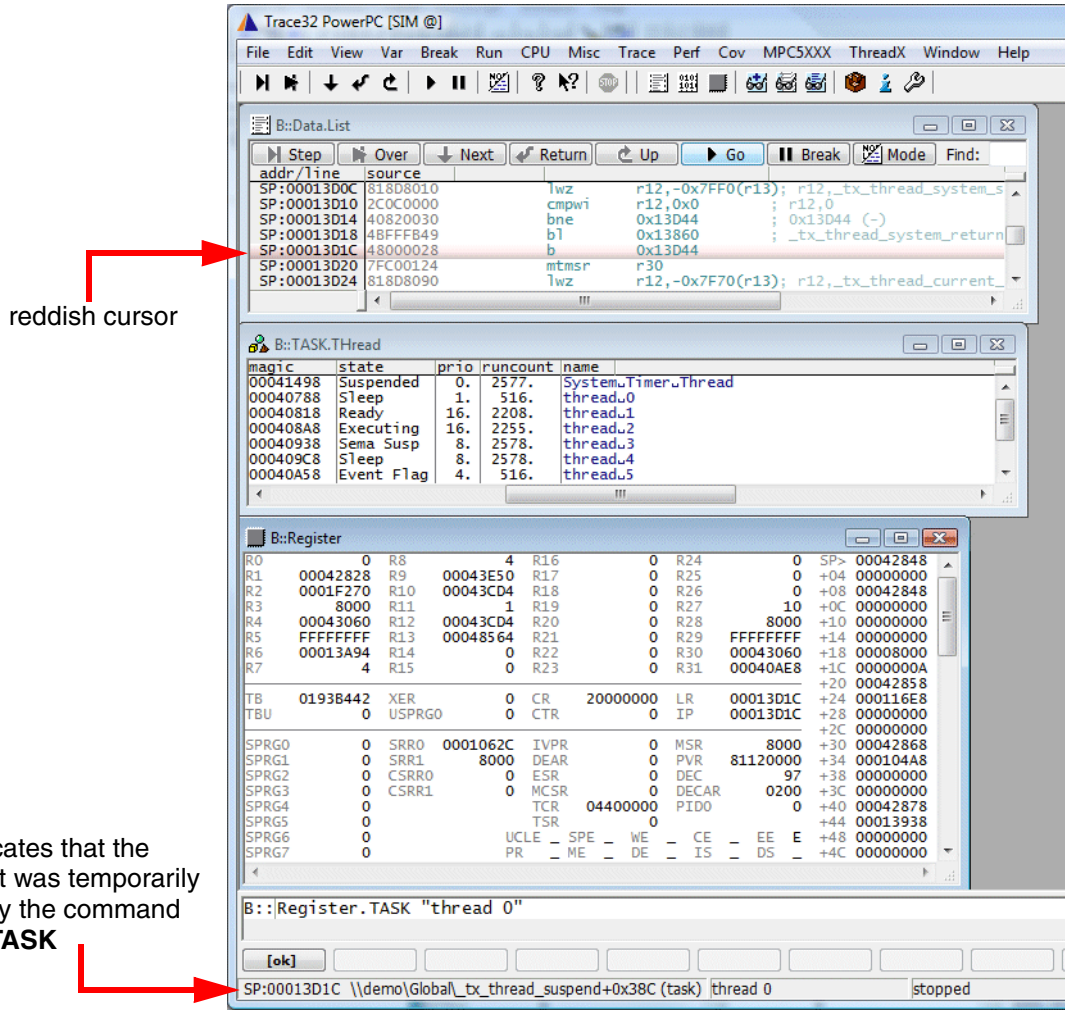
[Examples]

Format: **Frame.TASK** *<task_magic>* | *<task_id>* | *<task_name>*
Register.TASK (deprecated)

Display register set of the specified task temporarily. This will also change the source listing display (**Data.List**).

The register set of the current task is saved and will be restored before the application is started again.

A reddish cursor at the program counter indicates a manipulation by **Register.TASK**.



<task_magic>, etc.	See also **"What to know about the Task Parameters"** (general_ref_t.pdf).

Examples:

```
Register.TASK 0x41498                   ; <task_magic>

Register.TASK "thread 0"                ; <task_name>
```

See also

- Frame
- Frame.view

Format: **Frame.UNDO**
Register.UNDO (deprecated)

Recovers the state of the registers before the last register change or execution command. Repeating the command will recover earlier registers. The command **Frame.REDO** can be used to “undo” this command.

See also

- Frame
- Frame.view

Frame.Up Show state one level up in stack nesting

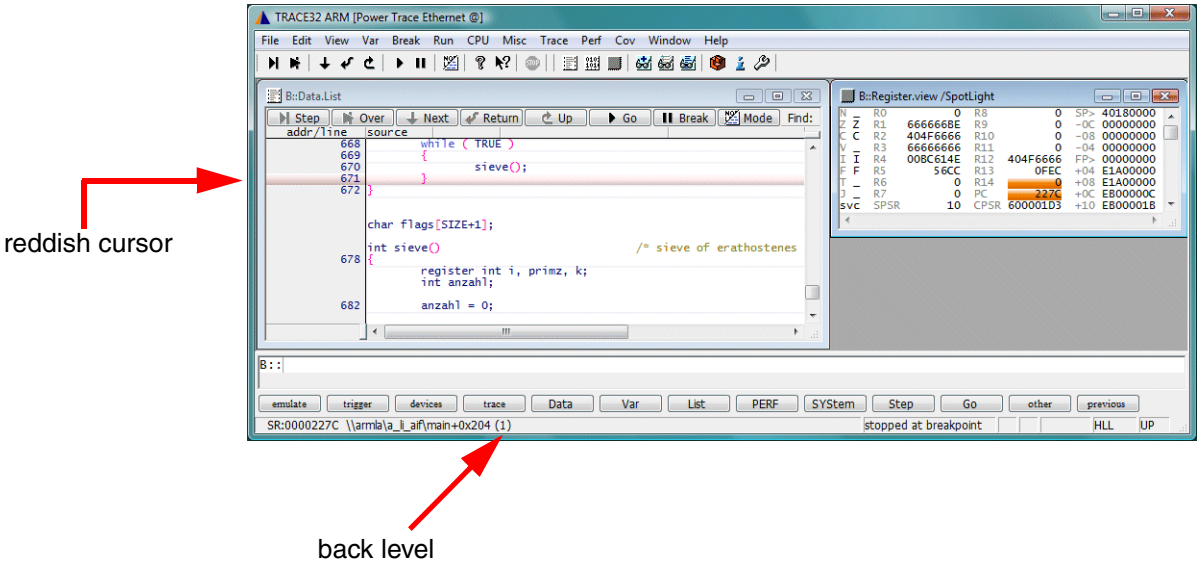
Format: **Frame.Up**
Register.Up (deprecated)

The debugger “virtually” goes up one level of the call stack by switching to the stack frame of the calling function and reconstructing its registers. The command can be repeated to traverse multiple levels of the call stack. The current offset from the actual stack frame is shown in the status line as (2), (3) etc.

The register window will show the values of the newly selected stack frame. In **Data.List** windows, the bar indicating the PC position changes its color to a reddish tone to indicate that it is not the actual PC of the processor.

The actual register set in the processor is not affected by this. Therefore a **Go** would continue from the place where the processor stopped before issuing the **Frame.Up** command.

The corresponding command for going down in the call hierarchy is **Frame.Down**.



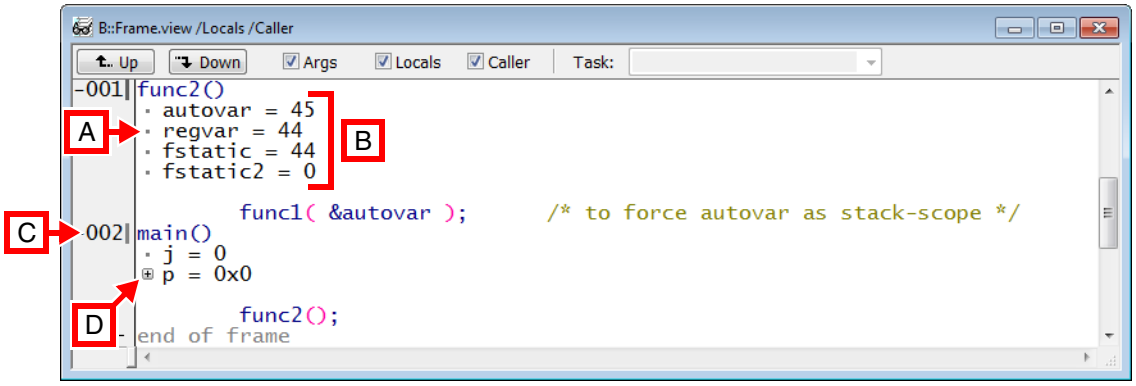
See also

- Frame
- Frame.view

[Examples]

Format:	**Frame.view** [**%***<format>*] [**/***<option>*] **Var.Frame** (deprecated)
<display_option>:	**NoVar** | **Args** | **Locals** | **Caller** | **MODule** | **Size** | **NoSize** | **LIMIT** *<levels>*
<context_option>:	**MACHINE** *<machine_number>* [**/VCPU** *<vcpu_number>*] **TASK** *<task_magic>* | *<task_id>* | "*<task_name>*" **CORE** *<number>* | **REGSET** *<number>*

Displays a stack back trace. By default the stack frame for the current program context is displayed. The command **Frame.CONFIG.Asm** configures whether the back trace stops at frames that do not belong to HLL code or attempts to trace back over those frames.



A By clicking the little, gray square to the left of a variable, you can switch between the formats hex, decimal, and binary for a value.

B By double-clicking a variable, you can modify its value.

C The numbers in the gray scale area of the **Frame.view** window indicate the nesting level. Double-clicking a number will execute a **Frame.Up** command to get to this level.

D Click the **+/-** toggle button to view the variable in more detail.

<context_option>	*<context_option>* allows to specify a different program context.
<display_option>	Without *<display_option>*, solely the functions together with the arguments are displayed.
<format>	The format parameters can modify the display in various ways. For information about format parameters, refer to section "**Display Formats**" in **"General Commands Reference Guide V"** (general_ref_v.pdf).

Args	Display of arguments (default). If the argument is prefixed with a colon ":" (e.g. ":state") then the displayed value is the initial argument value, otherwise the displayed value is the current argument value.
Caller	Display of the high level language block, from which the function was called.
CORE	Choose a core for which you want to display the frame.
LIMIT	Limit the number of frame levels to be displayed.
Locals	Local variables of functions.
MACHINE *<machine_id>*	Choose a virtual machine for which you want to display the stack frame. It is possible to specify additionally the number of the VCPU. See also **"What to know about the Machine Parameters"** (general_ref_t.pdf).
MODule	Display the module (e.g. library) to which the frame entry belongs.
NoVar	No display of variables and arguments.
REGSET	Choose one of the switchable register sets.
Size, **NoSize**	Show/hide the frame size.
TASK *<task_magic>*, etc.	Choose a task for which you want to display the frame. See also **"What to know about the Task Parameters"** (general_ref_t.pdf).
VCPU	A virtual core used by a virtual machine. See also VCPU and virtual machine in the **"TRACE32 Concepts"** (trace32_concepts.pdf).

Examples:

```
; display stack frame for task1 on the virtual machine 2
Frame.view /MACHINE 2 /TASK "task1"
```

```
; display stack frame with call information and local variables
Frame.view /Caller /Locals
```

See also

- Frame
- Frame.CONFIG
- Frame.COPY
- Frame.Down
- Frame.GOTO
- Frame.Init
- Frame.REDO
- Frame.SkipFunc
- Frame.SkipLine
- Frame.SWAP
- Frame.TASK
- Frame.UNDO

■ Frame.Up ■ Var.View ❑ FLAG() ❑ FLAG.READ()
❑ FLAG.WRITE()

▲ 'Release Information' in 'Legacy Release History'
▲ 'Display Variables' in 'Training Source Level Debugging'

FXU — FXU registers (extended floating point unit)

RH850

The **FXU** command group is used to display and modify the FXU (extended floating point unit) registers for RH850.

See also

- FXU.Init
- FXU.Set
- FXU.view
- CPU.FEATURE()
- FXU()

▲ 'FXU Function' in 'General Function Reference'

FXU.Init — Initialize FXU registers

RH850

Format: **FXU.Init**

Sets all registers of the active FXU extension to zero.

See also

- FXU
- FXU.view

FXU.Set — Modify FXU registers

RH850

Format: **FXU.Set** *<register> <value>* …

Modifies the FXU registers.

See also

- FXU
- FXU.view

Format:	**FXU.view**

Opens an FXU register window.

See also

- FXU
- FXU.Init
- FXU.Set