LAUTERBACH
DEVELOPMENT TOOLS

Onchip/NOR FLASH Programming User's Guide

Onchip/NOR FLASH Programming User's Guide

Onchip/NOR FLASH Programming User's Guide

Version 06-Jun-2024

Introduction

This manual contains all important information for programming

- On-chip FLASH memory
- Off-chip NOR FLASH devices

The programming of off-chip NAND FLASH devices is described in **"NAND FLASH Programming User's Guide"** (nandflash.pdf).

The programming of off-chip serial FLASH devices is described in **"Serial FLASH Programming User's Guide"** (serialflash.pdf).

The programming of off-chip eMMC FLASH devices is described in **"eMMC FLASH Programming User's Guide"** (emmcflash.pdf).

Standard Approach

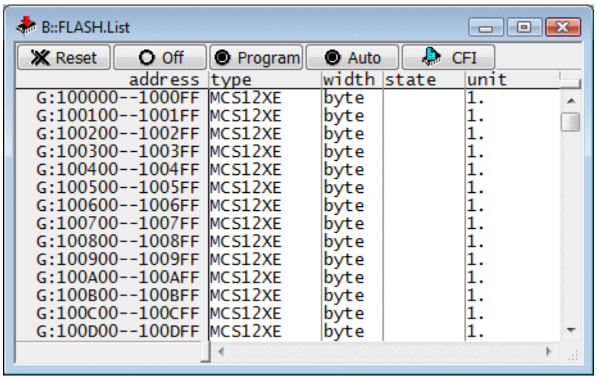
Standard Approach provides a compact description of the steps required to program on-chip/NOR FLASH memory. The description is knowingly restricted to standard cases. A detailed description of the FLASH programming concepts is given in the subsequent chapters of this manual.

On-chip FLASH

Integrated On-chip FLASH Programming

Integrated on-chip FLASH programming means that the FLASH programming algorithm is part of the TRACE32 software.

If the programming of the on-chip FLASH is **integrated** into the TRACE32 software, the on-chip FLASH is automatically declared when the target CPU is selected. The FLASH declaration is listed in the **FLASH.List** window.



```
          address type     width state    unit
  G:100000--1000FF MCS12XE byte           1.
  G:100100--1001FF MCS12XE byte           1.
  G:100200--1002FF MCS12XE byte           1.
  G:100300--1003FF MCS12XE byte           1.
  G:100400--1004FF MCS12XE byte           1.
  G:100500--1005FF MCS12XE byte           1.
  G:100600--1006FF MCS12XE byte           1.
  G:100700--1007FF MCS12XE byte           1.
  G:100800--1008FF MCS12XE byte           1.
  G:100900--1009FF MCS12XE byte           1.
  G:100A00--100AFF MCS12XE byte           1.
  G:100B00--100BFF MCS12XE byte           1.
  G:100C00--100CFF MCS12XE byte           1.
  G:100D00--100DFF MCS12XE byte           1.
```

If the on-chip FLASH is relocatable the address assignment for the FLASH is automatically performed before the on-chip FLASH is accessed by TRACE32. The address assignment is based on the settings in the corresponding configuration registers of the CPU.

Full example for integrated on-chip FLASH programming, here for the MCS12/S12X architecture:

```
; establish the communication between the debugger and the CPU
 SYStem.CPU Auto
 SYStem.Up

; program FLASH
 FLASH.AUTO ALL
 Data.LOAD.COSMIC demo.h12
 FLASH.AUTO off

; verify the FLASH contents
 Data.LOAD.COSMIC demo.h12 /DIFF
 IF FOUND()
     PRINT "Verify error after FLASH programming"
 ELSE
     PRINT "FLASH programming completed successfully"
...
```

Videos about the target-controlled on-chip flash programming can be found here:
support.lauterbach.com/kb/articles/flash-programming

Target-controlled on-chip FLASH programming means that the FLASH programming algorithm is not part of the TRACE32 software. An external programming algorithm usually provided by Lauterbach has to be linked to the TRACE32 software. This approach is demonstrated in example scripts. They can be found in the TRACE32 installation directory:
`~~/demo/`**`<architecture>`**`/flash/`**`<cpu>`**`.cmm`

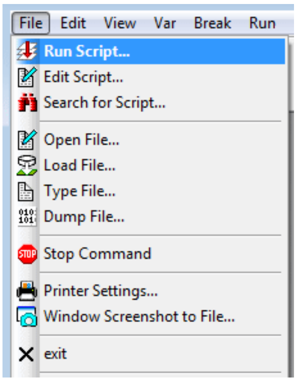
```
e.g. ~~/demo/powerpc/flash/jpc564xbc.cmm
     ~~/demo/arm/flash/mk20.cmm
```

Where `~~` is expanded to the `<TRACE32_installation_directory>`, which is `c:/T32` by default. Using target-controlled FLASH programming for the on-chip FLASH has the advantage that the FLASH algorithm can be updated very easily. In most cases no update of the TRACE32 software is required.

If available Lauterbach uses the FLASH programming libraries provided by the chip manufacturer. Using the provided libraries ensures that the FLASH algorithm fulfills the manufacturer´s requirements. The FLASH algorithm provided by Lauterbach is interfacing between the TRACE32 software and the library algorithm.

The FLASH programming example scripts are written for the following major use cases:

1. **Program FLASH directly after TRACE32 PowerView was started.**



Choose **File** menu > **Run Script** or use the following command to establish the debug communication and to program the on-chip FLASH:

```
DO ~~/demo/powerpc/flash/jpc564xbc.cmm
```

The script queries all necessary information via suitable dialog boxes.

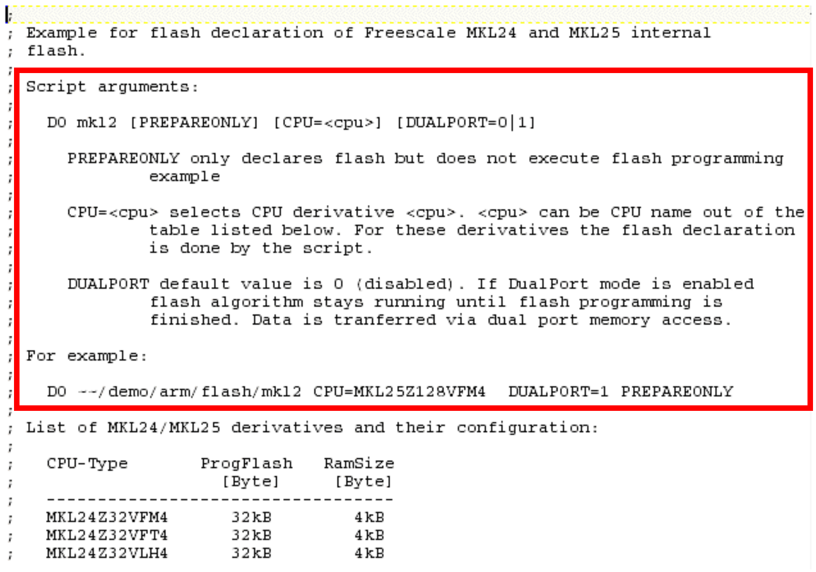
2. Use FLASH programming script in your start-up script.

If you create your own start-up script for your target hardware, please call the flash programming script from there. If you leave the flash programming script unchanged, you can always replace it with its most current version.

The following parameters can be used, when the flash programming script is called:

CPU=*<cpu>*	If a FLASH programming script supports a CPU family, you can provide your target CPU as parameter.
PREPAREONLY	Advise the FLASH programming script to prepare the FLASH programming by declaring the FLASH sectors and by linking the appropriate programming binary. The FLASH programming commands are bypassed.
DUALPORT=0|1	Disable/enable DualPort FLASH programming. For all processors/cores that allow to write to physical memory while the CPU is running a higher FLASH programming performance can be achieved by the use of **DualPort FLASH Programming**

Not every script supports all parameters. The parameters relevant for your script are described in the comment section of the script.



```
;
; Example for flash declaration of Freescale MKL24 and MKL25 internal
; flash.
;
; Script arguments:
;
;   DO mkl2 [PREPAREONLY] [CPU=<cpu>] [DUALPORT=0|1]
;
;     PREPAREONLY only declares flash but does not execute flash programming
;             example
;
;     CPU=<cpu> selects CPU derivative <cpu>. <cpu> can be CPU name out of the
;             table listed below. For these derivatives the flash declaration
;             is done by the script.
;
;     DUALPORT default value is 0 (disabled). If DualPort mode is enabled
;             flash algorithm stays running until flash programming is
;             finished. Data is tranferred via dual port memory access.
;
; For example:
;
;   DO ~~/demo/arm/flash/mkl2 CPU=MKL25Z128VFM4  DUALPORT=1 PREPAREONLY
;
; List of MKL24/MKL25 derivatives and their configuration:
;
;   CPU-Type       ProgFlash    RamSize
;                   [Byte]      [Byte]
;   ----------------------------------
;   MKL24Z32VFM4      32kB        4kB
;   MKL24Z32VFT4      32kB        4kB
;   MKL24Z32VLH4      32kB        4kB
```

The following framework can be used to call the flash programming script from your start-up script.

```
…

DO <flash_script> [CPU=<cpu>] PREPAREONLY [DUALPORT=0|1]

; program file to on-chip FLASH
FLASH.ReProgram ALL /Erase
Data.LOAD.Elf <file>
FLASH.ReProgram off

; reset processor/chip
; it might be necessary to reset all target settings made by the
; flash programming script
SYStem.Up

; continue with start-up script
…
```

⚠	Before the first use of the FLASH programming scripts, it is recommended to read the comment section of the script. In some cases the target memory layout or the programming clock has to be adapted. The comments in the example script describe the necessary adjustments.

The **FLASH.ReProgram** command used in a script can be replaced by the **FLASH.AUTO** command if a too old version of the TRACE32 software is used.

For some on-chip FLASHs the command **FLASH.Program** might fail due to:

- ECC protection of the on-chip FLASH

  Each ECC row can only be programmed once, but the file format fragmentation does not match the ECC row size.

- The on-chip FLASH programming sequence requires a specific number of bytes to be written simultaneously.

Off-chip FLASH Devices Supporting CFI

Here we focus on programming of CFI-conform FLASH devices, since most NOR FLASHs support this standard.

CFI stands for ***C**ommon **F**lash memory **I**nterface*. It is an open standard that describes how self-identifying information is provided by a FLASH device. Most relevant are:

- information about the FLASH programming algorithm
- device size and block configuration

TRACE32 queries this information to perform an easy declaration for off-chip NOR FLASH devices.

The following framework can be used to program CFI-conform FLASH devices:

```
                                        ; set up the CPU and configure the
                                        ; external bus interface

FLASH.RESet                             ; reset the FLASH declaration

FLASH.CFI …                             ; declare FLASH sectors via
                                        ; CFI query

FLASH.UNLOCK ALL                        ; unlock FLASH if required

FLASH.ReProgram ALL                     ; enable the FLASH for programming

Data.LOAD.auto …                        ; load the programming file

FLASH.ReProgram off                     ; program the FLASH and disable
                                        ; the FLASH programming
```

The following gives a description of the individual steps.

CPU Setup

FLASH programming with TRACE32 requires, that the communication between the debugger and the target CPU is established. The following commands are available to set up this communication:

SYStem.CPU *<cpu>*	Specify your target CPU
SYStem.Up	Establish the communication between the debugger and the target CPU

```
SYStem.CPU MCF5272       ; select ColdFire MCF5272 as target CPU

SYStem.Up                ; establish the communication between the
                         ; debugger and the target CPU
```

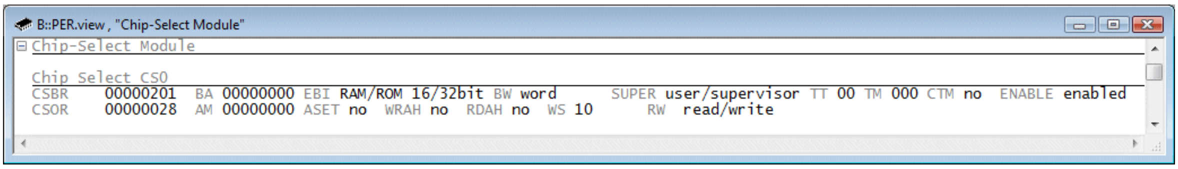
Bus Configuration

Programming of an off-chip FLASH devices requires a proper initialization of the external bus interface. The following settings in the bus configuration might be necessary:

- Write access has to be enabled for the FLASH devices
- Definition of the base address of the FLASH devices
- Definition of the size of the FLASH devices
- Definition of the data bus size that is used to access the FLASH devices
- Definition of the timing (number of wait states for the access to the FLASH devices)

Use the **PER.view** command to check the settings in the bus configuration registers.

```
PER.Set MOV:0xc0f %Long 0x10000001    ; specify the base address for the
                                      ; special function registers

                                      ; the FLASH is connected to Chip
                                      ; Select CS0, most settings are
                                      ; already correct after reset
PER.Set SD:0x10000044 %Long 0x28      ; set the number of wait states
                                      ; to 10

PER.view , "Chip-Select Module"       ; display the CS0 configuration
```



```
Chip-Select Module

Chip Select CS0
CSBR     00000201  BA 00000000 EBI RAM/ROM 16/32bit BW word      SUPER user/supervisor TT 00 TM 000 CTM no  ENABLE enabled
CSOR     00000028  AM 00000000 ASET no  WRAH no  RDAH no  WS 10      RW  read/write
```

FLASH Declaration

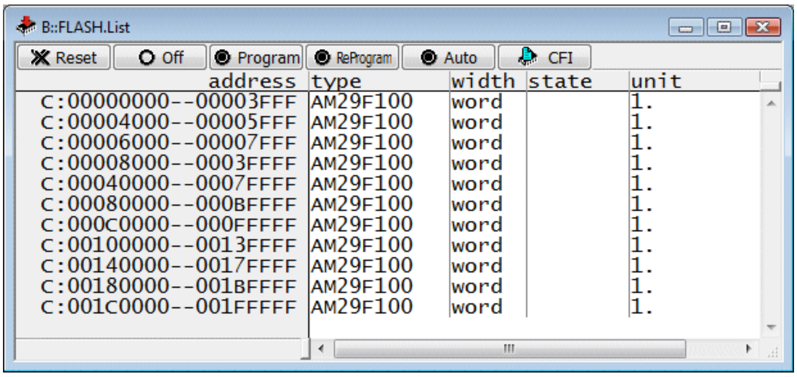
The following commands are available to set up the FLASH declaration:

FLASH.RESet	Reset the FLASH declaration
FLASH.CFI *<start_address> <bus_width>*	Set up a declaration for CFI-conform FLASH devices
FLASH.List	Display the declaration

Parameters for FLASH.CFI command	
<start_address>	Defines the start address of the FLASH devices.
<bus_width>	Defines the width of the data bus between the target CPU and the FLASH devices.

Example:

```
FLASH.RESet

FLASH.CFI 0x0 Word

FLASH.List
```

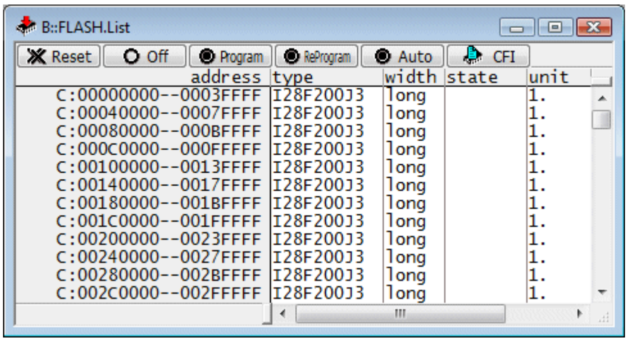


```
            address  type       width  state      unit
 C:00000000--00003FFF AM29F100   word              1.
 C:00004000--00005FFF AM29F100   word              1.
 C:00006000--00007FFF AM29F100   word              1.
 C:00008000--0003FFFF AM29F100   word              1.
 C:00040000--0007FFFF AM29F100   word              1.
 C:00080000--000BFFFF AM29F100   word              1.
 C:000C0000--000FFFFF AM29F100   word              1.
 C:00100000--0013FFFF AM29F100   word              1.
 C:00140000--0017FFFF AM29F100   word              1.
 C:00180000--001BFFFF AM29F100   word              1.
 C:001C0000--001FFFFF AM29F100   word              1.
```

If two or more identical FLASH devices are used in parallel to implement the needed data bus width, it is sufficient for the FLASH declaration to specify this *<bus_width>*.

Example: Two Intel Strata FLASH devices 28F128J3 in 16-bit mode are used in parallel to implement a 32-bit data bus.

```
FLASH.RESet

FLASH.CFI 0x0 Long

FLASH.List
```



```
                address type       width state   unit
   C:00000000--0003FFFF I28F200J3  long          1.
   C:00040000--0007FFFF I28F200J3  long          1.
   C:00080000--000BFFFF I28F200J3  long          1.
   C:000C0000--000FFFFF I28F200J3  long          1.
   C:00100000--0013FFFF I28F200J3  long          1.
   C:00140000--0017FFFF I28F200J3  long          1.
   C:00180000--001BFFFF I28F200J3  long          1.
   C:001C0000--001FFFFF I28F200J3  long          1.
   C:00200000--0023FFFF I28F200J3  long          1.
   C:00240000--0027FFFF I28F200J3  long          1.
   C:00280000--002BFFFF I28F200J3  long          1.
   C:002C0000--002FFFFF I28F200J3  long          1.
```

Unlocking the FLASH Devices

Many FLASH devices provide a sector/block protection to avoid unintended erasing or programming operations.

Since some FLASH devices are locked after power-up the protection has to be unlocked in order to erase or program the FLASH devices. Please refer to the data sheet of your FLASH device, to find out if your FLASH provides sector/block protection.

FLASH.UNLOCK ALL	Unlock FLASH devices

```
FLASH.UNLOCK ALL
```

Programming the FLASH Devices

The following command are available to program the FLASH:

FLASH.ReProgram ALL	Enable all declared FLASH devices for programming
FLASH.ReProgram off	Program the FLASH devices and disable the FLASH programming afterwards

Data.LOAD.auto *<file>*	Load the programming file (in most cases an automatic detection of the file format is possible)
Data.LOAD.*<file_format> <file>*	Please refer to the section **Compilers** in the chapter **Support** of your **Processor Architecture Manual** for the supported file formats
Data.LOAD.Elf *<file>*	Load the programming file (ELF/DWARF format)
Data.LOAD.Binary *<file> <start_address>*	Load the programming file (binary file) and specify the *<start_address>* of the FLASH devices
Data.LOAD.S3record *<file>*	Load the programming file (S3 record file)

Example:

```
FLASH.ReProgram ALL

Data.LOAD.auto demo.x

FLASH.ReProgram off
```

```
; select ColdFire MCF5272 as target CPU
SYStem.CPU MCF5272

; establish the communication between the debugger and the target CPU
SYStem.Up

; specify the base address for the special function registers
PER.Set MOV:0xc0f %Long 0x10000001

; the FLASH is connected to Chip Select CS0, most settings are already
; correct after reset - set the number of wait states to 10
PER.Set SD:0x10000044 %Long 0x28

; reset the FLASH declaration
FLASH.RESet

; declare the FLASH sectors by CFI query
FLASH.CFI 0x0 Word

; unlock the FLASH device if required for a power-up locked device
; FLASH.UNLOCK ALL

; enable the programming for all declared FLASH devices
FLASH.ReProgram ALL

; specify the file that should be programmed
Data.LOAD.auto demo.x

; program the file and disable the FLASH programming
FLASH.ReProgram off

; verify the FLASH contents
Data.LOAD.auto demo.x /DIFF

IF FOUND()
     PRINT "Verify error after FLASH programming"
ELSE
     PRINT "FLASH programming completed successfully"
...
```

Target-controlled FLASH Programming

The FLASH programming steps described so far are easy to carry out, but FLASH programming is slow. The programming time can be considerably improved by using so-called **target-controlled FLASH programming**.

Target-controlled FLASH programming means that the underlying FLASH programming algorithm is no longer part of the TRACE32 software. FLASH programming works now in principle as follows:

1. The FLASH algorithm is downloaded to the target RAM.
2. The programming data are downloaded to the target RAM.
3. The FLASH algorithm running in the target RAM programs the data to the FLASH devices.

This way the communication between the host and the debugger hardware is minimized.

Ready-to-run binary files for target-controlled FLASH programming are available for all common processor architectures in the folder `~~/demo/`**`<architecture>`**`/flash`. Where `~~` is expanded to the `<trace32_installation_directory>`, which is `c:/T32` by default. TRACE32 loads the appropriate FLASH programming algorithm automatically from this directory when target-controlled FLASH programming with CFI is used.

In order to initialize the communication between the TRACE32 software and the external FLASH programming algorithm the following command is used:

FLASH.CFI *<start_address> <bus_width>* **/TARGET** *<code_range> <data_range>*

Parameters

- *<start_address>*

  Defines the start address of the FLASH devices.

- *<bus_width>*

  Defines the width of the data bus between the target CPU and the FLASH devices.

- *<code_range>*

  Define an address range in the target´s RAM to which the external FLASH programming algorithm is loaded.

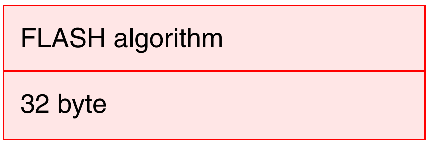
  

  Figure: Memory mapping for the *<code_range>*

  Required size for the code is `size_of(`*<flash_algorithm>*`) + 32 byte`

- *<data_range>*

  Define the address range in the target´s RAM where the programming data are buffered for the FLASH algorithm.

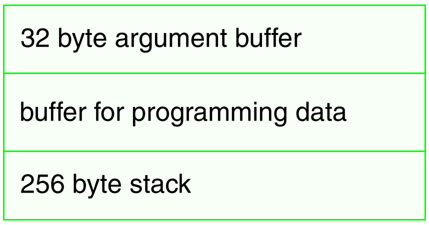
  

  Figure: Memory mapping for the *<data_range>*

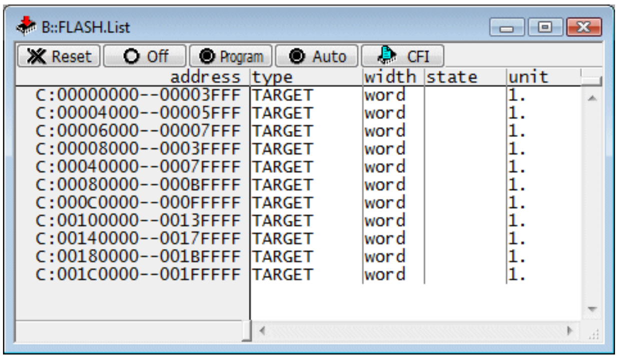
  The argument buffer used for the communication between the TRACE32 software and the FLASH algorithm is located at the first 32 bytes of *<data_range>*. The 256 byte stack is located at the end of *<data_range>*.

  `<buffer_size> =`
  `size_of(`*<data_range>*`) - 32 byte argument buffer - 256 byte stack`

  `<buffer_size>` is the maximum number of bytes that are transferred from the TRACE32 software to the external FLASH programming algorithm in one call.

Example:

```
FLASH.RESet

FLASH.CFI 0x0 Word /TARGET 0x20000000++0xfff 0x20001000++0xfff

FLASH.List
```



```
           address type        width state  unit
 C:00000000--00003FFF TARGET   word          1.
 C:00004000--00005FFF TARGET   word          1.
 C:00006000--00007FFF TARGET   word          1.
 C:00008000--0003FFFF TARGET   word          1.
 C:00040000--0007FFFF TARGET   word          1.
 C:00080000--000BFFFF TARGET   word          1.
 C:000C0000--000FFFFF TARGET   word          1.
 C:00100000--0013FFFF TARGET   word          1.
 C:00140000--0017FFFF TARGET   word          1.
 C:00180000--001BFFFF TARGET   word          1.
 C:001C0000--001FFFFF TARGET   word          1.
```

```
; select ColdFire MCF5272 as target CPU
SYStem.CPU MCF5272

; establish the communication between the debugger and the target CPU
SYStem.Up

; specify the base address for the special function registers
PER.Set MOV:0xc0f %Long 0x10000001

; the FLASH is connected to Chip Select CS0, most settings are already
; correct after reset - set the number of wait states to 10
PER.Set SD:0x10000044 %Long 0x28

; reset the FLASH declaration
FLASH.RESet

; set up the FLASH declaration for target-controlled programming
; target RAM at address 0x20000000
FLASH.CFI 0x0 Word /TARGET 0x20000000++0xfff 0x20001000++0xfff

; unlock the FLASH device if required for a power-up locked device
; FLASH.UNLOCK ALL

; enable the programming for all declared FLASH devices
; the option Erase ensures that unused sectors are erased
FLASH.ReProgram ALL /Erase

; specify the file that should be programmed
Data.LOAD.auto demo.x

; program all modified sectors and disable the FLASH programming
FLASH.ReProgram off

; verify the FLASH contents
Data.LOAD.auto demo.x /DIFF

IF FOUND()
     PRINT "Verify error after FLASH programming"
ELSE
     PRINT "FLASH programming completed successfully"
...
```

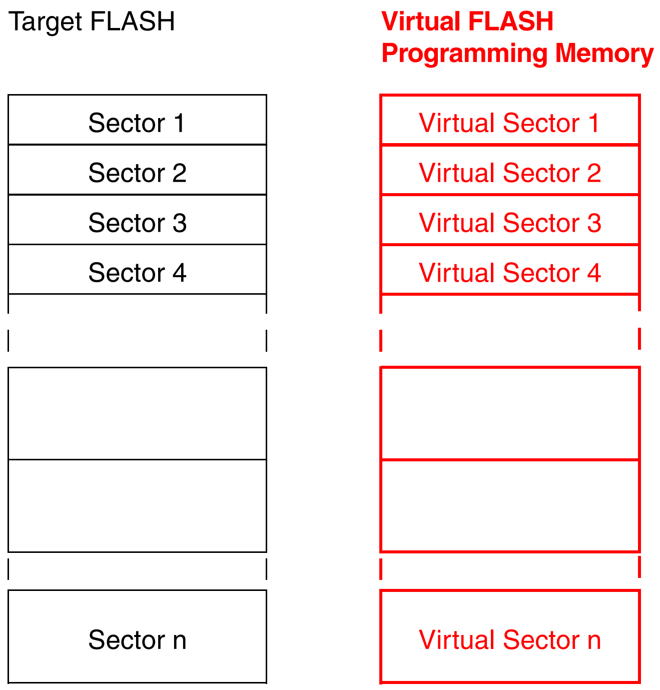
For all processors/cores that allow to write to physical memory while the CPU is running a higher FLASH programming performance can be achieved by the use of **DualPort FLASH Programming**.

FLASH.ReProgram Command (Target-controlled)

The command **FLASH.ReProgram** is the best choice for target-controlled FLASH programming. It provides an optimum FLASH programming performance by reducing the erasing and programming cycles to a minimum:

- Only not-empty sectors are erased.
- Only modified sectors are programmed.

TRACE32 allocates a *Virtual FLASH Programming Memory* to implement the **FLASH.ReProgram** command:



The following command sequence is recommended when using the **FLASH.ReProgram** command:

```
FLASH.ReProgram ALL /Erase          ; switch target FLASH to
                                    ; reprogramming state and
                                    ; erase virtual FLASH programming
                                    ; memory

Data.LOAD.auto …                    ; write the contents of the
                                    ; programming file to the virtual
                                    ; FLASH programming memory

FLASH.ReProgram off                 ; program only changed sectors to
                                    ; the target FLASH and erase
                                    ; obsolete code in unused sectors
```

Details

FLASH programming by using the **FLASH.ReProgram** command works in detail as follows:

```
FLASH.ReProgram ALL /Erase          ; switch target FLASH to
                                    ; reprogramming state and
                                    ; erase virtual FLASH programming
                                    ; memory
```

It is recommended to initialize the *Virtual FLASH Programming Memory* as erased (option **/Erase**). This inhibits that obsolete code is remaining in unused sectors.

When the command **FLASH.ReProgram /Erase** is entered:

1. The *Virtual FLASH Programming Memory* is erased.

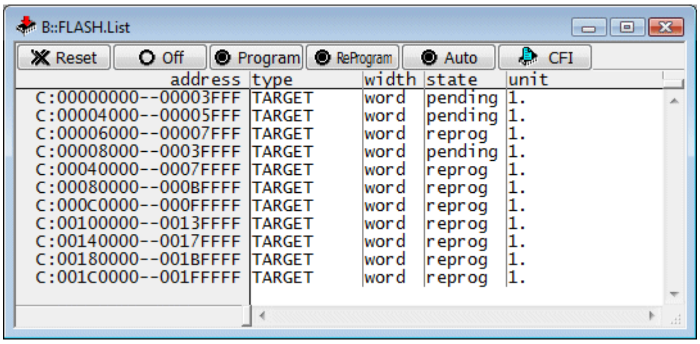
2. The target-controlled FLASH algorithm is called to deliver the following information:

   - The checksum for the target FLASH sector

   - The information if the target FLASH sector is erased

   - The information if an erased FLASH bit is 0 or 1

Afterwards all not-empty FLASH sectors are marked as **pending**. All empty FLASH sectors are marked as **reprog**.



```
                 address type       width state    unit
  C:00000000--00003FFF TARGET     word  pending  1.
  C:00004000--00005FFF TARGET     word  pending  1.
  C:00006000--00007FFF TARGET     word  reprog   1.
  C:00008000--0003FFFF TARGET     word  pending  1.
  C:00040000--0007FFFF TARGET     word  reprog   1.
  C:00080000--000BFFFF TARGET     word  reprog   1.
  C:000C0000--000FFFFF TARGET     word  reprog   1.
  C:00100000--0013FFFF TARGET     word  reprog   1.
  C:00140000--0017FFFF TARGET     word  reprog   1.
  C:00180000--001BFFFF TARGET     word  reprog   1.
  C:001C0000--001FFFFF TARGET     word  reprog   1.
```

```
Data.LOAD.auto …                    ; write the contents of the
                                    ; programming file into the
                                    ; virtual FLASH programming
                                    ; memory
```

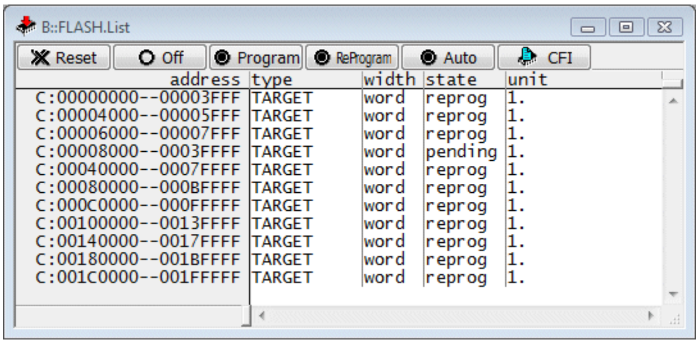
The following actions are taken, when TRACE32 performs a write access to a sector:

3. The new data is copied to the corresponding sector in the *Virtual FLASH Programming Memory.*

4. The checksum for the virtual sector is calculated.

5. The state of the sector is changed from **pending** to **reprog** if the checksum of the target FLASH sector is equal the checksum of the virtual sector.

   The state of the sector is changed from **reprog** to **pending** if the checksum of the target FLASH sector is different from the checksum of the virtual sector.

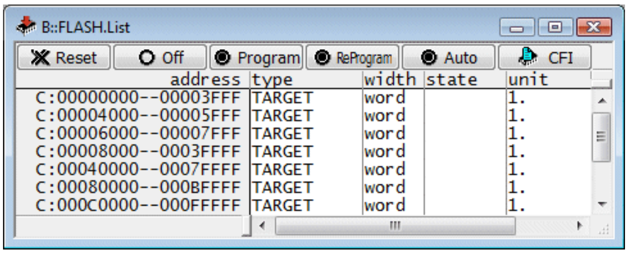


```
           address type       width state   unit
C:00000000--00003FFF TARGET   word  reprog  1.
C:00004000--00005FFF TARGET   word  reprog  1.
C:00006000--00007FFF TARGET   word  reprog  1.
C:00008000--0003FFFF TARGET   word  pending 1.
C:00040000--0007FFFF TARGET   word  reprog  1.
C:00080000--000BFFFF TARGET   word  reprog  1.
C:000C0000--000FFFFF TARGET   word  reprog  1.
C:00100000--0013FFFF TARGET   word  reprog  1.
C:00140000--0017FFFF TARGET   word  reprog  1.
C:00180000--001BFFFF TARGET   word  reprog  1.
C:001C0000--001FFFFF TARGET   word  reprog  1.
```

```
FLASH.ReProgram off                 ; program only changed sectors to
                                    ; the target FLASH and erase
                                    ; obsolete code in unused sectors
```

When the command **FLASH.ReProgram off** is executed, all sectors marked as pending are programmed to the target FLASH. In this process TRACE32 only erases not-empty sectors before programming.

After all pending sectors are programmed FLASH programming is disabled. This is indicated by an empty state column in the **FLASH.List** window.



```
           address type       width state   unit
C:00000000--00003FFF TARGET   word          1.
C:00004000--00005FFF TARGET   word          1.
C:00006000--00007FFF TARGET   word          1.
C:00008000--0003FFFF TARGET   word          1.
C:00040000--0007FFFF TARGET   word          1.
C:00080000--000BFFFF TARGET   word          1.
C:000C0000--000FFFFF TARGET   word          1.
```

NOTE:	Please be aware that TRACE32 PowerView displays the contents of the *Virtual FLASH Programming Memory*, if the FLASH is in reprogramming state and the current state of the FLASH sector is pending.

NOTE:	If the FLASH is in reprogramming state, the command **FLASH.ReProgram CANCEL** can be used to undo all changes and re-start from scratch.

The command syntax:

FLASH.ReProgram ALL | *<address_range>* | *<unit_number>* [**/Erase**]	Switch FLASH device to reprogramming state for optimized FLASH programming
FLASH.ReProgram off	Program only changed sectors
FLASH.ReProgram CANCEL	Cancel FLASH programming without programming pending changes.

TRACE32 commands that perform a write access on the memory:

Data.LOAD.auto *<file>*	Write code from the programming file to memory (if an automatic detection of file format is possible)
Data.Set [*<address>*|*<range>* **%***<format>* *<value>*]	Write *<value>* to the specified memory location
Data.PATTERN *<range>* [**/***<option>*]	Fill memory with a predefined pattern
…	

Full example:

```
...

; reset the FLASH declaration
FLASH.RESet

; set up the FLASH declaration for target-controlled programming
; target RAM at address 0x20000000
FLASH.CFI 0x0 Word /TARGET 0x20000000++0xfff 0x20001000++0xfff

; switch all declared FLASH sectors to reprogramming state
FLASH.ReProgram ALL /Erase

; copy the code from the programming file to the corresponding sectors
; in the virtual FLASH and mark all changed sectors
Data.LOAD.auto demo.x

; program only the changed sectors
FLASH.ReProgram off

; verify the FLASH contents
Data.LOAD.auto demo.x /DIFF

IF FOUND()
     PRINT "Verify error after FLASH programming"
ELSE
     PRINT "FLASH programming completed successfully"
...
```

FLASH.ReProgram Command (TRACE32 Tool-based)

If TRACE32 tool-based FLASH programming is used, the **reprog** state is handled differently:

The following actions are taken, when TRACE32 performs a write access on a sector:

1. The corresponding sector in the *Virtual FLASH Programming Memory* is marked as **pending**.

   Due to performance reasons no erase status is checked and no checksum is calculated for the target FLASH sector.

2. The corresponding sector is erased in the *Virtual FLASH Programming Memory*.

3. The new data is copied to the corresponding sector in the *Virtual FLASH Programming Memory*.

If the recommended command sequence for the **FLASH.ReProgram** is used, no performance benefit is reached compared to the usage of **FLASH.Erase** / **FLASH.Program**.

- Since no erase status is maintained for a target FLASH sector empty sectors are erased.
- Since no checksum is calculated for a target FLASH sector not-modified sectors are programmed.

A reasonable performance benefit is reached, when the *Virtual FLASH Programming Memory* is not erased before the contents of the programming file is copied. Such a proceeding however includes the risk, that obsolete code remains in unused target FLASH sectors.

```
FLASH.ReProgram ALL                     ; switch target FLASH to
                                        ; reprogramming state

Data.LOAD.auto …                        ; write the contents of the
                                        ; programming file to the virtual
                                        ; FLASH programming memory

FLASH.ReProgram off                     ; erase and program all those
                                        ; sectors to the target FLASH to
                                        ; which the contents of the
                                        ; programming file was written
```

FLASH.Erase / FLASH.Program Command

Beside the **FLASH.ReProgram** command there are also the commands **FLASH.Erase** and **FLASH.Program** to program the FLASH devices. Both commands work identically for TRACE32 tool-based and target-controlled programming.

```
FLASH.Erase ALL                      ; erase the FLASH devices

FLASH.Program ALL                    ; enable all FLASH devices for
                                     ; programming

Data.LOAD.auto …                     ; write the contents of the
                                     ; programming file to the FLASH
                                     ; devices

FLASH.Program off                    ; disable the programming for the
                                     ; FLASH devices
```

FLASH programming by using the **FLASH.Erase** / **FLASH.Program** commands works in detail as follows:

```
FLASH.Erase ALL                      ; erase the FLASH devices
```

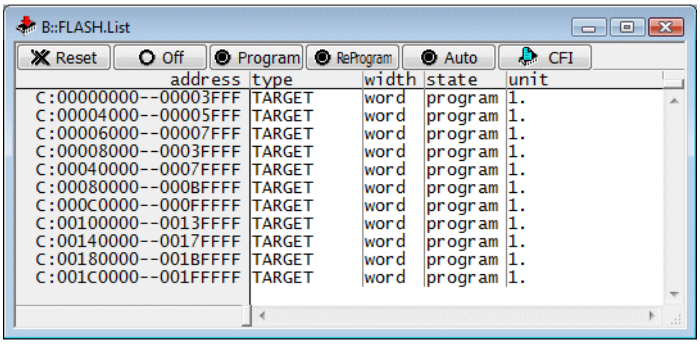
All declared FLASH sectors are erased.

TRACE32 is using *full chip erase*/*bulk erase* whenever possible. If the FLASH devices are declared manually it is strongly recommended:

- To declare each FLASH device with **all sectors**. Otherwise if there are undeclared FLASH sectors for a FLASH device, these sectors are also erased when a *full chip erase*/*bulk erase* is used.
- To declare each FLASH device with its **own *<unit_number>***, if two or more FLASH devices are used in series to implement the needed FLASH memory size. Otherwise if the same unit number is used for 2 or more FLASH devices only the first FLASH device is erased.

```
FLASH.Program ALL                         ; enable the FLASH devices for
                                          ; programming
```

When the command **FLASH.Program** is entered, the state of all FLASH sectors is changed to **program**.



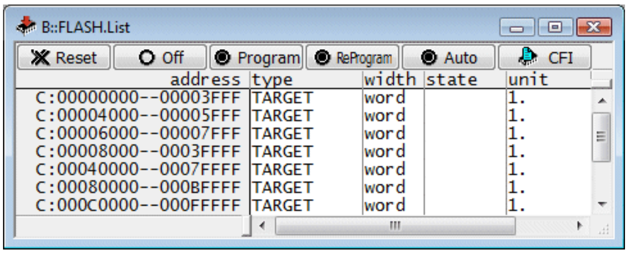
```
address                type    width state   unit
C:00000000--00003FFF   TARGET  word  program 1.
C:00004000--00005FFF   TARGET  word  program 1.
C:00006000--00007FFF   TARGET  word  program 1.
C:00008000--0003FFFF   TARGET  word  program 1.
C:00040000--0007FFFF   TARGET  word  program 1.
C:00080000--000BFFFF   TARGET  word  program 1.
C:000C0000--000FFFFF   TARGET  word  program 1.
C:00100000--0013FFFF   TARGET  word  program 1.
C:00140000--0017FFFF   TARGET  word  program 1.
C:00180000--001BFFFF   TARGET  word  program 1.
C:001C0000--001FFFFF   TARGET  word  program 1.
```

If a FLASH sector is in **program** state each write access by TRACE32 on this sector is directly converted to a FLASH programming command sequences. A typical TRACE32 command that writes to FLASH sectors is:

```
Data.LOAD.auto …                          ; write the contents of the
                                          ; programming file to the FLASH
                                          ; devices
```

```
FLASH.Program off                         ; disable the programming for the
                                          ; FLASH devices
```

With the command **FLASH.Program off** FLASH programming is disabled. This is indicated by an empty state column in the **FLASH.List** window.



```
address                type    width state   unit
C:00000000--00003FFF   TARGET  word          1.
C:00004000--00005FFF   TARGET  word          1.
C:00006000--00007FFF   TARGET  word          1.
C:00008000--0003FFFF   TARGET  word          1.
C:00040000--0007FFFF   TARGET  word          1.
C:00080000--000BFFFF   TARGET  word          1.
C:000C0000--000FFFFF   TARGET  word          1.
```

The command syntax:

FLASH.Erase ALL | *<address_range>* | *<unit_number>*	Erase the specified FLASH sectors
FLASH.Program ALL | *<address_range>* | *<unit_number>*	Enable the specified FLASH sectors for programming
FLASH.Program off	Disable the programming state for all FLASH sectors.

Full example:

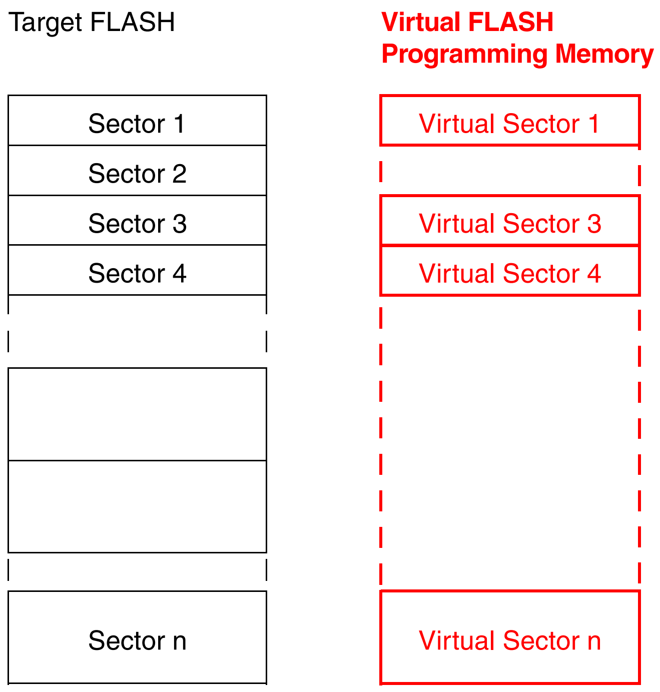
```
...

; reset the FLASH declaration
FLASH.RESet

; set up the FLASH declaration for target-controlled programming
; target RAM at address 0x20000000
FLASH.CFI 0x0 Word /TARGET 0x20000000++0xfff 0x20001000++0xfff

; erase all FLASH sectors
FLASH.Erase ALL

; enable all FLASH sectors for programming
FLASH.Program ALL

; write the code from the programming file to the target FLASH
Data.LOAD.auto demo.x

; disable the programming state for all sectors
FLASH.Program off

; verify the FLASH contents
Data.LOAD.auto demo.x /DIFF

IF FOUND()
     PRINT "Verify error after FLASH programming"
ELSE
     PRINT "FLASH programming completed successfully"
...
```

The FLASH.AUTO Command

The command **FLASH.AUTO** is a special command that allows:

- to set software breakpoints to FLASH
- to patch code located in FLASH

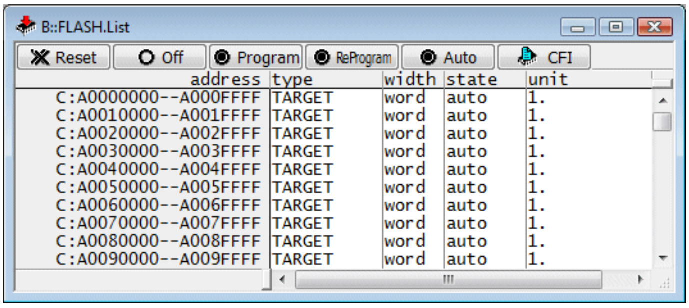
TRACE32 is using a *Virtual FLASH Programming Memory* to implement the **FLASH.AUTO** command:



Using the **FLASH.AUTO** command to set software breakpoints to FLASH works as follows:

```
FLASH.AUTO ALL                          ; switch the target FLASH into auto
                                        ; state
```

When the command **FLASH.AUTO ALL** is entered, the state of all FLASH sectors is changed to **auto**.



```
                  address type         width state    unit
  C:A0000000--A000FFFF TARGET       word  auto     1.
  C:A0010000--A001FFFF TARGET       word  auto     1.
  C:A0020000--A002FFFF TARGET       word  auto     1.
  C:A0030000--A003FFFF TARGET       word  auto     1.
  C:A0040000--A004FFFF TARGET       word  auto     1.
  C:A0050000--A005FFFF TARGET       word  auto     1.
  C:A0060000--A006FFFF TARGET       word  auto     1.
  C:A0070000--A007FFFF TARGET       word  auto     1.
  C:A0080000--A008FFFF TARGET       word  auto     1.
  C:A0090000--A009FFFF TARGET       word  auto     1.
```

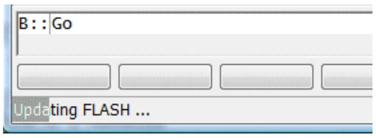
```
Break.Set … /Program /SOFT              ; set a software breakpoint to a
                                        ; program address located in FLASH
```

If a software breakpoint is set to an address within the FLASH, the following actions are taken:

1. *Virtual FLASH Programming Memory* is allocated for the affected sector.
2. The code from the target FLASH sector is copied to the virtual sector.
3. The code at the location of the software breakpoint is saved by TRACE32.
4. The software breakpoint is patched into the virtual sector.

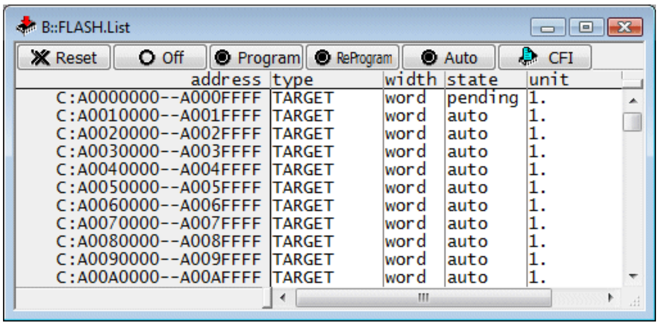
```
Go                                        ; start the program execution
```

All virtual sectors to which software breakpoints were patched are copied to the target FLASH when the program execution is started.



```
B::Go
Updating FLASH ...
```

The state of the affected FLASH sectors is changed to **pending** to indicate that software breakpoints were programmed into these sectors.



B::FLASH.List

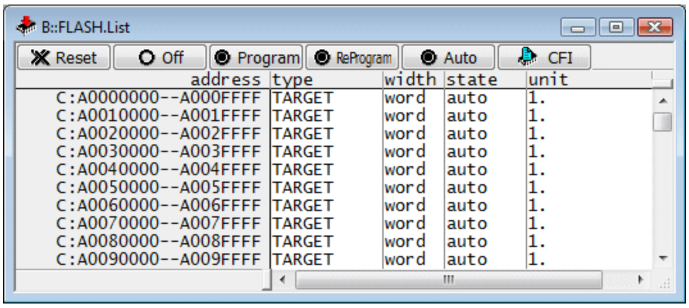
```
             address type      width state   unit
   C:A0000000--A000FFFF TARGET    word  pending 1.
   C:A0010000--A001FFFF TARGET    word  auto    1.
   C:A0020000--A002FFFF TARGET    word  auto    1.
   C:A0030000--A003FFFF TARGET    word  auto    1.
   C:A0040000--A004FFFF TARGET    word  auto    1.
   C:A0050000--A005FFFF TARGET    word  auto    1.
   C:A0060000--A006FFFF TARGET    word  auto    1.
   C:A0070000--A007FFFF TARGET    word  auto    1.
   C:A0080000--A008FFFF TARGET    word  auto    1.
   C:A0090000--A009FFFF TARGET    word  auto    1.
   C:A00A0000--A00AFFFF TARGET    word  auto    1.
```

```
Break.Delete … /Program /SOFT           ; remove the software breakpoint
```

When the software breakpoint is deleted, the original code at the location of the software breakpoint is restored in the virtual sector.

All virtual sectors that contain such a restoration are programmed to the target FLASH when the program execution is started.

The state of the restored FLASH sectors is changed back to **auto**.



```
                address type     width state unit
C:A0000000--A000FFFF TARGET   word  auto  1.
C:A0010000--A001FFFF TARGET   word  auto  1.
C:A0020000--A002FFFF TARGET   word  auto  1.
C:A0030000--A003FFFF TARGET   word  auto  1.
C:A0040000--A004FFFF TARGET   word  auto  1.
C:A0050000--A005FFFF TARGET   word  auto  1.
C:A0060000--A006FFFF TARGET   word  auto  1.
C:A0070000--A007FFFF TARGET   word  auto  1.
C:A0080000--A008FFFF TARGET   word  auto  1.
C:A0090000--A009FFFF TARGET   word  auto  1.
```

Please execute the command **FLASH.AUTO off** before you exit TRACE32. This guarantees that all software breakpoints are removed from the target FLASH and the original code is restored.

The following command sequence is recommended when using the **FLASH.AUTO** command:

```
FLASH.AUTO ALL                          ; switch the target FLASH to auto
                                        ; state to allow debugging with
                                        ; software breakpoints in FLASH

…

Break.Set … /Program /SOFT

…

FLASH.AUTO off                          ; use this command to restore the
                                        ; original code at the locations of
                                        ; the software breakpoints back to
                                        ; the target FLASH

                                        ; exit TRACE32
```

The command syntax:

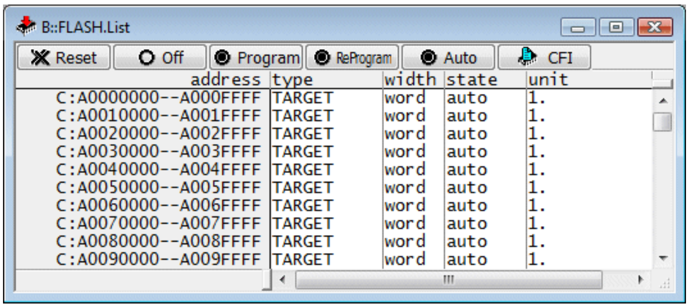
FLASH.AUTO ALL | *<address_range>* | *<unit_number>*	Switch FLASH to auto state to allow debugging with software breakpoints in FLASH
FLASH.AUTO off	Restore the original code in the FLASH for all software breakpoints and disable the auto state

NOTE:	Please be aware that TRACE32 PowerView displays the contents of the *Virtual FLASH Programming Memory*, if the FLASH is in auto state and the current state of the FLASH sector is pending.

Using the **FLASH.AUTO** command to patch code located in FLASH works as follows:

```
FLASH.AUTO ALL                            ; switch the target FLASH into auto
                                          ; state
```

When the command **FLASH.AUTO ALL** is entered, the state of all FLASH sectors is changed to **auto**.



```
         address  type      width state   unit
C:A0000000--A000FFFF TARGET word  auto    1.
C:A0010000--A001FFFF TARGET word  auto    1.
C:A0020000--A002FFFF TARGET word  auto    1.
C:A0030000--A003FFFF TARGET word  auto    1.
C:A0040000--A004FFFF TARGET word  auto    1.
C:A0050000--A005FFFF TARGET word  auto    1.
C:A0060000--A006FFFF TARGET word  auto    1.
C:A0070000--A007FFFF TARGET word  auto    1.
C:A0080000--A008FFFF TARGET word  auto    1.
C:A0090000--A009FFFF TARGET word  auto    1.
```

```
Data.Assemble …                               ; patch code in FLASH

Data.Set …                                    ; patch hex. value to FLASH
```

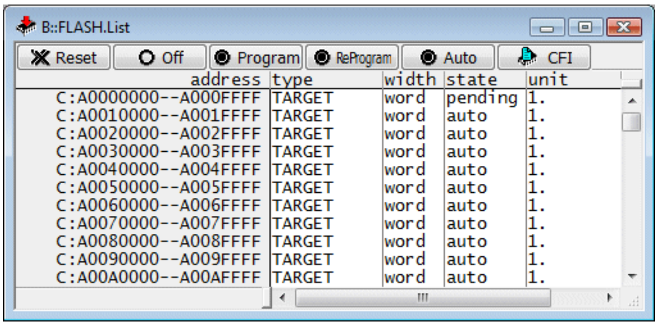
If code located in FLASH is patched, the following actions are taken:

1. *Virtual FLASH Programming Memory* is allocated for the affected sector.

2. The code from the target FLASH sector is copied to the virtual sector.

3. The patch is copied into the virtual sector.

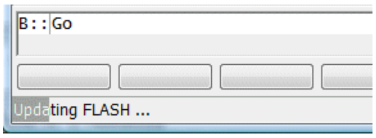
4. The state of the virtual sector is change to **pending** to indicate, that a patch needs to be programmed.



```
         address  type      width state   unit
C:A0000000--A000FFFF TARGET word  pending 1.
C:A0010000--A001FFFF TARGET word  auto    1.
C:A0020000--A002FFFF TARGET word  auto    1.
C:A0030000--A003FFFF TARGET word  auto    1.
C:A0040000--A004FFFF TARGET word  auto    1.
C:A0050000--A005FFFF TARGET word  auto    1.
C:A0060000--A006FFFF TARGET word  auto    1.
C:A0070000--A007FFFF TARGET word  auto    1.
C:A0080000--A008FFFF TARGET word  auto    1.
C:A0090000--A009FFFF TARGET word  auto    1.
C:A00A0000--A00AFFFF TARGET word  auto    1.
```

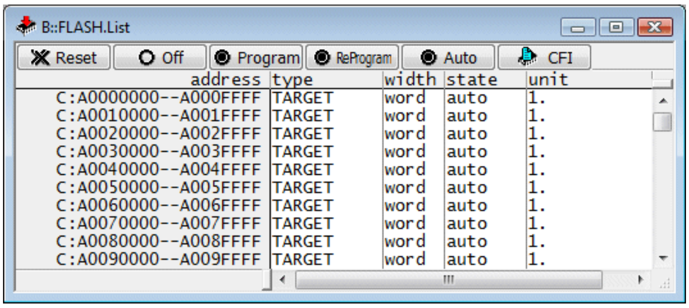
```
Go                                  ; start the program execution
```

All virtual sectors which contain patches are copied to the target FLASH when the program execution is started.



```
B::Go
Updating FLASH ...
```

The state of the affected FLASH sectors is changed back to **auto** after the patch is programmed.



```
           address type    width state  unit
 C:A0000000--A000FFFF TARGET  word  auto   1.
 C:A0010000--A001FFFF TARGET  word  auto   1.
 C:A0020000--A002FFFF TARGET  word  auto   1.
 C:A0030000--A003FFFF TARGET  word  auto   1.
 C:A0040000--A004FFFF TARGET  word  auto   1.
 C:A0050000--A005FFFF TARGET  word  auto   1.
 C:A0060000--A006FFFF TARGET  word  auto   1.
 C:A0070000--A007FFFF TARGET  word  auto   1.
 C:A0080000--A008FFFF TARGET  word  auto   1.
 C:A0090000--A009FFFF TARGET  word  auto   1.
```

```
FLASH.AUTO off                      ; disable the auto state and
                                    ; program all pending patches
```

The following command sequence is recommended when using the **FLASH.AUTO** command to patch code located in FLASH:

```
FLASH.AUTO ALL                      ; switch the target FLASH to auto
                                    ; state to allow code patches in
                                    ; FLASH

…

Data.Assemble …

…

FLASH.AUTO off                      ; program all pending patches and
                                    ; disable the auto state
```

The command syntax:

FLASH.AUTO ALL | *<address_range>* | *<unit_number>*	Switch FLASH to auto state to allow code patches is FLASH
FLASH.AUTO off	Program all pending patches and disable the auto state

NOTE:	Please be aware that TRACE32 PowerView displays the contents of the *Virtual FLASH Programming Memory*, if the FLASH is in auto state and the current state of the FLASH sector is pending.

CENSORSHIP Option

A FLASH sector can contain sensitive data e.g. bytes that unsecure the chip and enable it for debugging. An unintended or incorrect write to this data might secure the chip and lock it for debugging.

The FLASH programming algorithm provided by Lauterbach is aware of sensitive data. It discards all erase and write operations to their addresses.

The **CENSORSHIP** option enables the erasing/programming of the sensitive data.

FLASH.AUTO *<address_range>* **/CENSORSCHIP**	Enable erasing/programming of sensitive data

Example for Kinetis MK30DX64VEX7, the first FLASH sector contains sensitive data.

```
FLASH.Create 1. 0x00000000--0x0000FFFF 0x800 TARGET Long

…

; a FLASH programming algorithm that is aware
; of sensitive data is defined
FLASH.TARGET … ~~/demo/arm/flash/long/ftfl1x.bin

…

; the FLASH sector with sensitive data is explicitly enabled for
; erasing/programming
FLASH.AUTO 0--0x7ff /CENSORSHIP

Data.Set …
Data.Set …
…

; program all changed data and disable the erasing/programming of
; sensitive data
FLASH.AUTO OFF
```

Unlocking Command

Many FLASH devices provide a sector/block protection to avoid unintended erasing and programming operations. Most of them are locked after power-up. They need to be unlocked in order to be erased or programmed..

FLASH.UNLOCK ALL | *<address_range>* | *<unit_number>*

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).

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, TRACE32 tool-based programming

```
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
```

Example for 2 (parallel unlocking):

INTEL 28F128J3 at address 0x0, connected to the CPU via a 16-bit data bus, each sector 128 KByte, target-controlled programming.

Please be aware that the flash device in this example only supports a full-device unlock. This means a single **FLASH.UNLOCK** command unlocks the complete device.

```
; reset FLASH declaration
FLASH.RESet

; declare FLASH sectors via CFI query
FLASH.CFI 0x0 Word /TARGET 0x10000000++0xfff 0x10001000++0xfff

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

; erasing and programming
…
```

The FLASH devices can be re-locked after programming to avoid unintended erasing and programming operations while debugging. Re-locking has to be executed usually sector by sector.

FLASH.LOCK ALL | *<address_range>* | *<unit_number>*

Re-locking is not recommended if you like to use:

- Software breakpoints in FLASH
- Code patches in FLASH

Please refer to **“The FLASH.AUTO Command”** in Onchip/NOR FLASH Programming User’s Guide, page 31 (norflash.pdf) for details.

DualPort FLASH Programming

Benefits

Dualport FLASH programming reduces the FLASH programming time for all processors/cores that allow to write to physical memory while the CPU is running. This time reduction is achieved by the simultaneous execution of the following: the next block of programming data is downloaded to the target RAM while the FLASH algorithm is programming the current block of data.

The best results can be achieved if the following times are nearly the same:

- Sector erase time
- Download time of a block of programming data (host to target RAM)
- Programming time of a block of data

The average time reduction is 5% to 30%. However, under favorable conditions, the programming time can be shortened by up to 70%.

Preconditions

1. DualPort FLASH programming is only supported for target-controlled FLASH programming.
2. DualPort FLASH programming can only be used for processors/cores that allow to write to physical memory while the CPU is running. For details on this feature refer to **“Run-time Memory Access”** in TRACE32 Glossary, page 42 (glossary.pdf).
3. DualPort FLASH programming requires a FLASH binary that is position independent. This is the case for nearly all FLASH binaries provided by Lauterbach.
4. FLASH programming in general requires that the data cache is disabled for the entire address range of the FLASH.
5. For DualPort FLASH programming the data cache has to be disabled also for the RAM area that is used to buffer the programming data, because run-time memory access can only write to physical memory.

Usage

DualPort FLASH programming achieves a time reduction for the following programming commands:

- FLASH.ReProgram
- FLASH.Auto

DualPort FLASH programming is enabled by the use of one of the following commands:

FLASH.CFI … **/TARGET** *<code_range> <data_range>* **/DualPort**

FLASH.CFI … **/TARGET** *<code_address> <data_address>* [*<buffer_size>*] **/DualPort**

FLASH.TARGET *<code_range> <data_range> <file>* **/DualPort**

FLASH.TARGET *<code_address> <data_address>* [*<buffer_size>*] *<file>* **/DualPort**

If you use a processor/core that allows to write to physical memory while the CPU is running, but the option DUALPORT is not accepted by TRACE32 PowerView, DualPort FLASH programming is not yet supported for your processor architecture. Please contact **flash-support@lauterbach.com** for details.

The following framework can be used for DualPort FLASH programming:

```
                                       ; set up the CPU and configure the
                                       ; external bus interface

FLASH.RESet                            ; reset the FLASH declaration

FLASH.CFI … /TARGET … /DualPort        ; declare FLASH sectors via
                                       ; CFI query

FLASH.UNLOCK ALL                       ; unlock FLASH if required

FLASH.ReProgram ALL /Erase             ; enable the FLASH for programming

Data.LOAD.auto …                       ; load the programming file

FLASH.ReProgram off                    ; program the FLASH and disable
                                       ; the FLASH programming
```

Full Example

```
; set up the CPU and configure the external bus interface

; reset the FLASH declaration
FLASH.RESet

; set up the FLASH declaration for target-controlled programming
; target RAM at address 0x20000000
FLASH.CFI 0x0 Word /TARGET 0x20000000++0xfff 0x20001000++0xfff /DualPort

; enable the programming for all declared FLASH devices
FLASH.ReProgram ALL /Erase

; specify the file that should be programmed
Data.LOAD.auto demo.x

; program only modified sectors and erase obsolete code in unused sectors
; disable the FLASH programming
FLASH.ReProgram off
```

FLASH.CFI … /TARGET *<code_range> <data_range>* **/DualPort**

<data_range> is automatically extended by the **access class** E: if DualPort FLASH programming is used.

Full Example (ARM/Cortex)

Because ARM/Cortex perform the run-time memory access via the AHB bus, the access class AHB: has to specified explicitly for *<data_range>.*

```
; set up debug communication for LPC4357 and configure
; the external bus interface

; reset the FLASH declaration
FLASH.RESet

; set up the FLASH declaration for target-controlled programming
; target RAM at address 0x10000000
FLASH.CFI 0x1C000000 Long /TARGET 0x10000000 EAHB:0x10001000 0x2000 /DualPort

; enable the programming for all declared FLASH devices
FLASH.ReProgram ALL /Erase

; specify the file that should be programmed
Data.LOAD.auto demo.x

; program only modified sectors and erase obsolete code in unused sectors
; disable the FLASH programming
FLASH.ReProgram off
```

Special Features for Onchip FLASHs

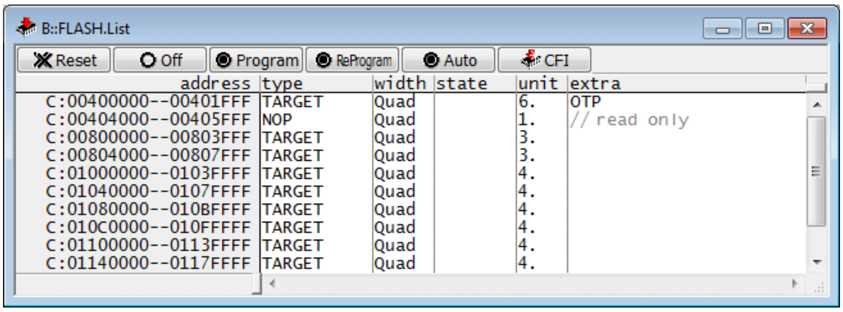
The ready-to-run scripts for onchip FLASH programming provided by Lauterbach use in addition to the classical FLASH programming commands special **commands and options** to handle characteristics of onchip FLASHs.

OTP Sector Programming

Nowadays many onchip FLASHs contain OTP sectors. OTP sectors can not be erased, that's why they are called **O**ne **T**ime **P**rogrammable sectors. The ready-to-run FLASH programming scripts provided by Lauterbach use the option **OTP** to protect those sectors from unintentional programming.

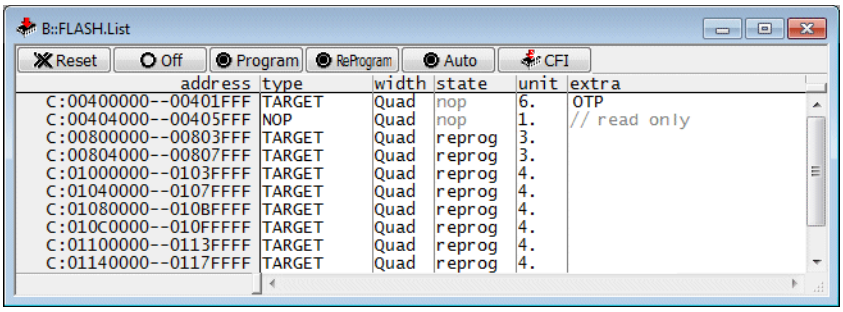
FLASH.Create …**TARGET** … **/OTP**	Protect OTP sector from unintentional programming

```
FLASH.Create 6. 0x00400000--0x00401FFF TARGET Quad /OTP
```



```
          address type    width state   unit extra
C:00400000--00401FFF TARGET  Quad          6.   OTP
C:00404000--00405FFF NOP     Quad          1.   // read only
C:00800000--00803FFF TARGET  Quad          3.
C:00804000--00807FFF TARGET  Quad          3.
C:01000000--0103FFFF TARGET  Quad          4.
C:01040000--0107FFFF TARGET  Quad          4.
C:01080000--010BFFFF TARGET  Quad          4.
C:010C0000--010FFFFF TARGET  Quad          4.
C:01100000--0113FFFF TARGET  Quad          4.
C:01140000--0117FFFF TARGET  Quad          4.
```

Whenever FLASH programming is activated by one of the following commands: **FLASH.ReProgram**, **FLASH.AUTO** or **FLASH.Program** the state of all OTP sectors changes to **nop** to indicate that all FLASH erasing and programming commands are blocked for these sectors.



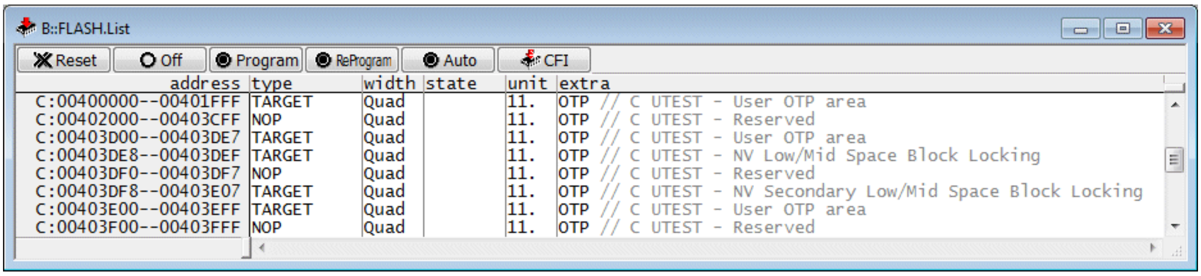
```
          address type    width state   unit extra
C:00400000--00401FFF TARGET  Quad  nop     6.   OTP
C:00404000--00405FFF NOP     Quad  nop     1.   // read only
C:00800000--00803FFF TARGET  Quad  reprog  3.
C:00804000--00807FFF TARGET  Quad  reprog  3.
C:01000000--0103FFFF TARGET  Quad  reprog  4.
C:01040000--0107FFFF TARGET  Quad  reprog  4.
C:01080000--010BFFFF TARGET  Quad  reprog  4.
C:010C0000--010FFFFF TARGET  Quad  reprog  4.
C:01100000--0113FFFF TARGET  Quad  reprog  4.
C:01140000--0117FFFF TARGET  Quad  reprog  4.
```

In order to program an OTP sector the following command sequence is recommended:

```
FLASH.Program 0x00400000--0x00401FFF /OTP        ; enable FLASH
                                                  ; programming for the
                                                  ; specified OTP sector

Data.Set …                                        ; program the data
                                                  ; to sector


FLASH.Program off                                 ; disable FLASH
                                                  ; programming
```

Already programmed OTP sectors can be declared with **NOP** as *<family_code>*. TRACE32 PowerView discards all erase and write operations to No OPeration sectors.

```
FLASH.Create 11.0x402000++0x1cff NOP Quad /OTP /Info "C UTEST - Reserved"
```



```
           address type      width state    unit extra
C:00400000--00401FFF TARGET  Quad           11.  OTP // C UTEST - User OTP area
C:00402000--00403CFF NOP     Quad           11.  OTP // C UTEST - Reserved
C:00403D00--00403DE7 TARGET  Quad           11.  OTP // C UTEST - User OTP area
C:00403DE8--00403DEF TARGET  Quad           11.  OTP // C UTEST - NV Low/Mid Space Block Locking
C:00403DF0--00403DF7 NOP     Quad           11.  OTP // C UTEST - Reserved
C:00403DF8--00403E07 TARGET  Quad           11.  OTP // C UTEST - NV Secondary Low/Mid Space Block Locking
C:00403E00--00403EFF TARGET  Quad           11.  OTP // C UTEST - User OTP area
C:00403F00--00403FFF NOP     Quad           11.  OTP // C UTEST - Reserved
```

Mirrored FLASH Addresses

The TRACE32 NOR FLASH programming might fail, if an on-chip FLASH or an off-chip FLASH device is addressed in more than one way.

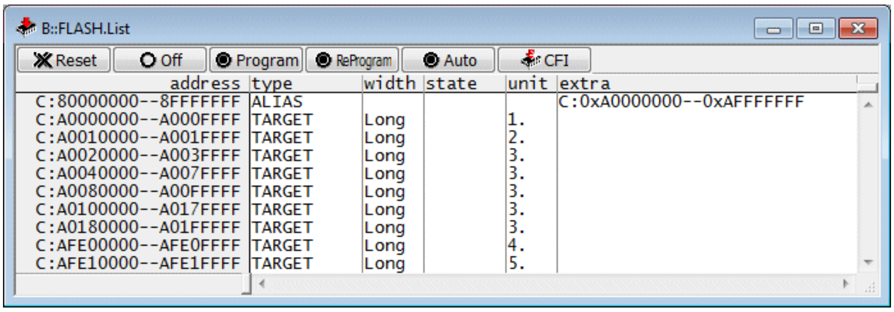
Non-Cached/Cached Addresses

Some processor architectures (e.g. TriCore, MIPS) provide separate address spaces for cached and non-cached memory. Compilers may generate code for non-cached addresses for the boot sequence and code for cached addresses for the application.

In order to program the FLASH successfully it is recommended to declare the FLASH for non-cached addresses and use the command **FLASH.CreateALIAS** to mirror these addresses to the cached address space.

```
; example for the TriCore architecture
; non-cached on-chip FLASH starts at address 0xa0000000
; cached on-chip FLASH starts at address 0x80000000

FLASH.RESet
; declare program FLASH in non-cached address space
FLASH.Create 1. 0xa0000000--0xa000ffff 0x4000  TARGET Long
FLASH.Create 2. 0xa0010000--0xa001ffff 0x4000  TARGET Long
FLASH.Create 3. 0xa0020000--0xa003ffff 0x20000 TARGET Long
FLASH.Create 3. 0xa0040000--0xa007ffff 0x40000 TARGET Long
FLASH.Create 3. 0xa0080000--0xa01fffff 0x80000 TARGET Long

; declare data FLASH in non-cached address space
FLASH.Create 4. 0xafe00000--0xafe0ffff 0x10000 TARGET Long
FLASH.Create 5. 0xafe10000--0xafe1ffff 0x10000 TARGET Long

; declare FLASH programming algorithm
FLASH.TARGET 0xd4000000 0xd0000000 0x1000
             ~~/demo/tricore/flash/long/tc1796.bin

; all FLASH write cycles to the address range 0x80000000--0x8fffffff
; (cached) are redirected to the address range 0xa0000000--0xafffffff
; (non-cached)

FLASH.CreateALIAS 0x80000000--0x8fffffff 0xa0000000
…
```

The alias is also displayed in the listing of the FLASH declarations.



```
                address  type    width state  unit extra
 C:80000000--8FFFFFFF  ALIAS                       C:0xA0000000--0xAFFFFFFF
 C:A0000000--A000FFFF  TARGET  Long         1.
 C:A0010000--A001FFFF  TARGET  Long         2.
 C:A0020000--A003FFFF  TARGET  Long         3.
 C:A0040000--A007FFFF  TARGET  Long         3.
 C:A0080000--A00FFFFF  TARGET  Long         3.
 C:A0100000--A017FFFF  TARGET  Long         3.
 C:A0180000--A01FFFFF  TARGET  Long         3.
 C:AFE00000--AFE0FFFF  TARGET  Long         4.
 C:AFE10000--AFE1FFFF  TARGET  Long         5.
```

FLASH mirrored to Boot Area

Some CPUs allow to mirror the FLASH to the boot code address space. In order to program the FLASH successfully it is recommended to declare the FLASH for its primary address space and use the command **FLASH.CreateALIAS** to mirror these addresses to the boot code address space.

Hardvard Architecture with Unified Memory

For CPUs with Harvard architecture and unified FLASH memory, it is recommended to declare the FLASH for the program memory address space and use the command **FLASH.CreateALIAS** to mirror these addresses to the data address space.

```
; generate FLASH declaration by CFI
FLASH.CFI P:0x0 Word

…

; all FLASH write cycles to the data address space are redirected
; to the program address space
FLASH.CreateALIAS D:0x0++0xffffff P:
```

Group Code NOP

TRACE32 PowerView discards all erase and write operations to No OPeration sectors. NOP sectors are used for the following purposes:

- Some FLASH sectors are already programmed by the processor/chip manufacturer. If you debug without a proper FLASH declaration for these sectors an unintended erase or write operation may result in a bus error or something similar.

  Declaring such sectors as NOP sectors guarantees an error-free debugging.

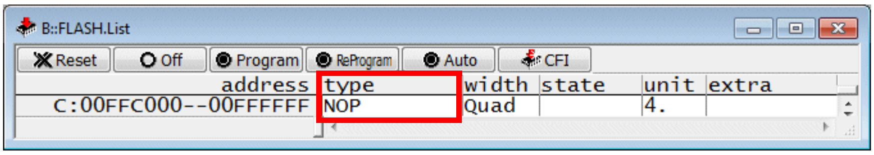
- Same FLASH sectors contain sensitive information. An unintended overwrite can harm the system or lock the processor/chip for debugging. Examples for sensitive sectors are: shadow raws, boot sectors, FLASH sectors that contain the debug monitor.

  TRACE32 PowerView forces the user to handle such sectors with care by declaring them as NOP sectors.

  The chapter **"FLASH.CHANGETYPE Command"**, page 56 introduces a command sequence that is recommended for the programming of sensitive sectors.

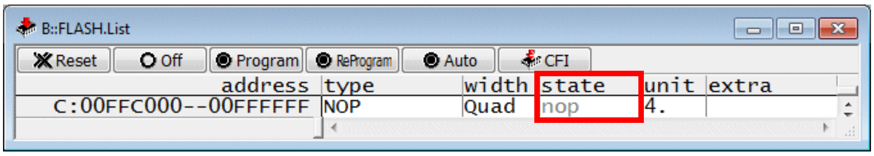
FLASH.Create *<unit_number> <address_range>* [*<sector_size>*] **NOP** *<bus_width>*

```
FLASH.Create 4. 0xFFC000--0xFFFFFF NOP Quad
```



```
         address type     width state     unit extra
C:00FFC000--00FFFFFF NOP  Quad            4.
```

Whenever FLASH programming is activated by one of the following commands: **FLASH.ReProgram**, **FLASH.AUTO** or **FLASH.Program** the state of all NOP sectors changes to **nop** to indicate that all FLASH erasing and programming commands are discarded for these sectors.



```
         address type     width state     unit extra
C:00FFC000--00FFFFFF NOP  Quad  nop       4.
```

INFO Option

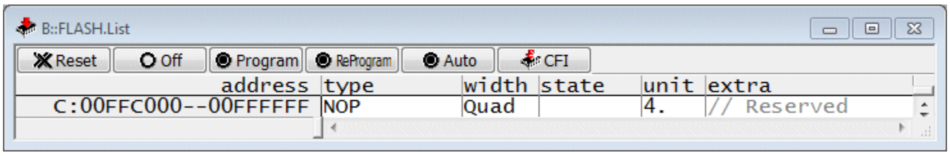
TRACE32 PowerView allows to add comments to FLASH sectors.

A comment can be up to 64 characters long. TRACE32 PowerView allocated 4 kBytes for all comments.

FLASH.Create … **/INFO** *<comment>*

```
FLASH.Create 4. 0xFFC000--0xFFFFFF NOP Quad /INFO "Reserved"
```

The comment is displayed in the **extra** column of the **FLASH.List** window.



```
           address type       width state    unit extra
C:00FFC000--00FFFFFF NOP        Quad           4.   // Reserved
```

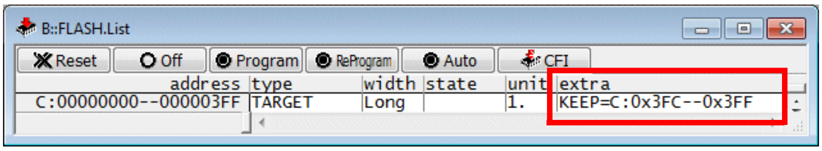
KEEP Option

FLASH sectors may contain data that should not be deleted. An example are chip trimming data stored in FLASH.

Use the option **KEEP**, if you want to advise TRACE32 PowerView to preserve the data in the specified *<address_range>*.

FLASH.Create … **/KEEP** *<address_range>*

```
FLASH.Create 1. 0x00000000--0x0001FFFF … /KEEP 0x0003FC--0x0003FF
```



```
           address type   width state unit extra
C:00000000--000003FF TARGET Long        1.   KEEP=C:0x3FC--0x3FF
```

The preservation of the FLASH content is implemented differently depending on the used FLASH programming command.

FLASH.ReProgram (for details refer to **"FLASH.ReProgram Command (Target-controlled)"**, page 21): After a virtual FLASH sector is erased, the information to be preserved is written back to the virtual sector. This approach assumes that the *<address_range>* to be preserved is not overwritten by data loaded from the file to be programmed.

FLASH.Erase: After the FLASH sector is erased, the information is restored. This approach assumes that the *<address_range>* to be preserved is not overwritten by data loaded from the file to be programmed.

FLASH.AUTO (for details refer to **“The FLASH.AUTO Command”**, page 31): Since the data from the target sector are copied to the virtual FLASH sectors, the option **KEEP** is not required. Not overwritten original data are programmed back to the target sectors.

BootModeHeaDer Option

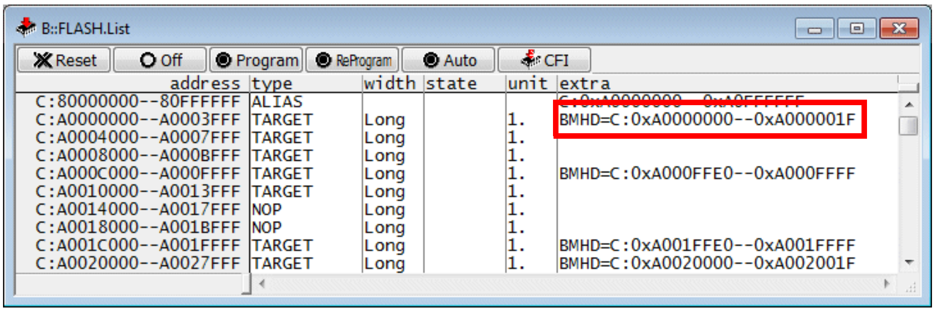
For the Astep version of the TC27x debugging was locked if the onchip FLASH does not provide a valid Boot Mode HeaDer. To avoid that the onchip FLASH contains no valid BMHD after programming, TRACE32 takes the following preventive measures:

1. TRACE32 tries to preserve all valid BMHDs.
2. The FLASH programming scripts warns you if the FLASH data to be programmed do not contain a valid BMHD. For details refer to your FLASH programming script.

More details to 1: The option **BootModeHeaDer** advises TRACE32 to preserve the contents of *<address_range>* if *<address_range>* contains a valid BMHD.

FLASH.Create … **/BootModeHeaDer** *<address_range>*

```
FLASH.Create  1. 0xA0000000--0xA000BFFF … /BMHD 0xA0000000--0xA000001F
```



```
          address |type     |width |state |unit |extra
 C:80000000--80FFFFFF |ALIAS    |      |      |     |C:0xA0000000--0xA0FFFFFF
 C:A0000000--A0003FFF |TARGET   |Long  |      |1.   |BMHD=C:0xA0000000--0xA000001F
 C:A0004000--A0007FFF |TARGET   |Long  |      |1.   |
 C:A0008000--A000BFFF |TARGET   |Long  |      |1.   |
 C:A000C000--A000FFFF |TARGET   |Long  |      |1.   |BMHD=C:0xA000FFE0--0xA000FFFF
 C:A0010000--A0013FFF |TARGET   |Long  |      |1.   |
 C:A0014000--A0017FFF |NOP      |Long  |      |1.   |
 C:A0018000--A001BFFF |NOP      |Long  |      |1.   |
 C:A001C000--A001FFFF |TARGET   |Long  |      |1.   |BMHD=C:0xA001FFE0--0xA001FFFF
 C:A0020000--A0027FFF |TARGET   |Long  |      |1.   |BMHD=C:0xA0020000--0xA002001F
```

The preservation of the BMHDs is implemented differently depending on the used FLASH programming command. For details refer to **“KEEP Option”**, page 50.

EraseALIAS Option

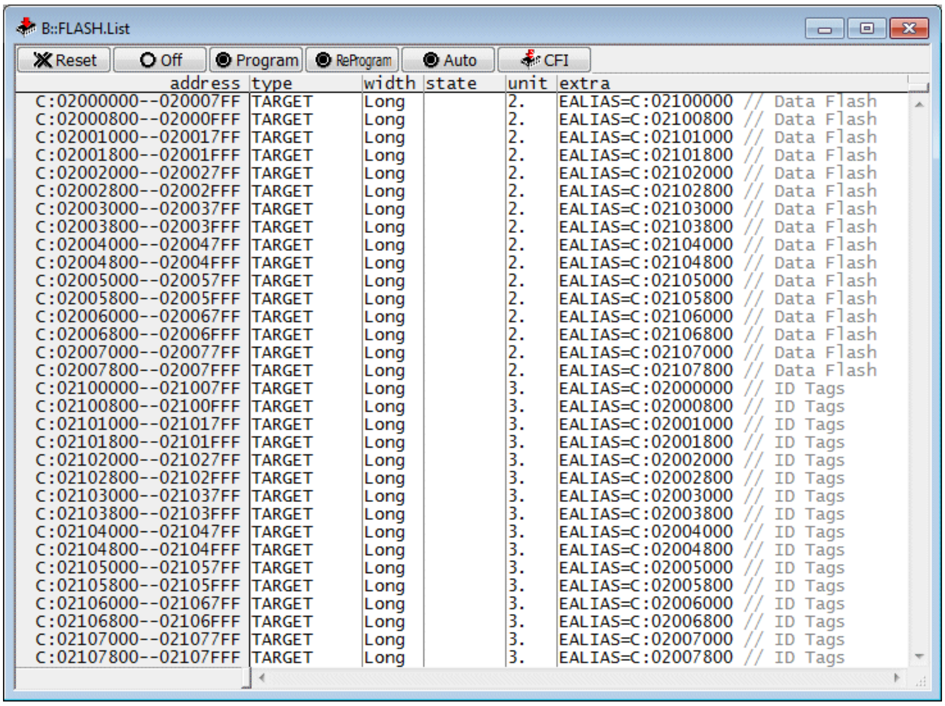
A physical FLASH sector can be split up into two or more logical address spaces, if it maintains different types of information. FLASH programming writes usually to the logical address spaces, while FLASH erasing applies to the physical FLASH sector.

To understand the use cases of the option **EraseALIAS** it is important to remember that the commands **FLASH.ReProgram** or **FLASH.AUTO** erase/program only modified sectors. The option **EraseALIAS** guarantees:

- That content of logical address spaces is preserved, if a physical sector has to be erased, in order to program one of its modified logical address spaces,
- That a physical sector is only erased once while the modified FLASH content is programmed.

FLASH.Create … **/EraseALIAS** *<address_range>*

```
FLASH.Create 2. 0x2000000++0x7fff /EALIAS 0x2100000++0x7fff /INFO "Data Flash"

FLASH.Create 3. 0x2100000++0x7fff /EALIAS 0x2000000++0x7fff /INFO "ID Tags"
```



```
            address type       width state   unit extra
  C:02000000--020007FF TARGET   Long          2.   EALIAS=C:02100000 // Data Flash
  C:02000800--02000FFF TARGET   Long          2.   EALIAS=C:02100800 // Data Flash
  C:02001000--020017FF TARGET   Long          2.   EALIAS=C:02101000 // Data Flash
  C:02001800--02001FFF TARGET   Long          2.   EALIAS=C:02101800 // Data Flash
  C:02002000--020027FF TARGET   Long          2.   EALIAS=C:02102000 // Data Flash
  C:02002800--02002FFF TARGET   Long          2.   EALIAS=C:02102800 // Data Flash
  C:02003000--020037FF TARGET   Long          2.   EALIAS=C:02103000 // Data Flash
  C:02003800--02003FFF TARGET   Long          2.   EALIAS=C:02103800 // Data Flash
  C:02004000--020047FF TARGET   Long          2.   EALIAS=C:02104000 // Data Flash
  C:02004800--02004FFF TARGET   Long          2.   EALIAS=C:02104800 // Data Flash
  C:02005000--020057FF TARGET   Long          2.   EALIAS=C:02105000 // Data Flash
  C:02005800--02005FFF TARGET   Long          2.   EALIAS=C:02105800 // Data Flash
  C:02006000--020067FF TARGET   Long          2.   EALIAS=C:02106000 // Data Flash
  C:02006800--02006FFF TARGET   Long          2.   EALIAS=C:02106800 // Data Flash
  C:02007000--020077FF TARGET   Long          2.   EALIAS=C:02107000 // Data Flash
  C:02007800--02007FFF TARGET   Long          2.   EALIAS=C:02107800 // Data Flash
  C:02100000--021007FF TARGET   Long          3.   EALIAS=C:02000000 // ID Tags
  C:02100800--02100FFF TARGET   Long          3.   EALIAS=C:02000800 // ID Tags
  C:02101000--021017FF TARGET   Long          3.   EALIAS=C:02001000 // ID Tags
  C:02101800--02101FFF TARGET   Long          3.   EALIAS=C:02001800 // ID Tags
  C:02102000--021027FF TARGET   Long          3.   EALIAS=C:02002000 // ID Tags
  C:02102800--02102FFF TARGET   Long          3.   EALIAS=C:02002800 // ID Tags
  C:02103000--021037FF TARGET   Long          3.   EALIAS=C:02003000 // ID Tags
  C:02103800--02103FFF TARGET   Long          3.   EALIAS=C:02003800 // ID Tags
  C:02104000--021047FF TARGET   Long          3.   EALIAS=C:02004000 // ID Tags
  C:02104800--02104FFF TARGET   Long          3.   EALIAS=C:02004800 // ID Tags
  C:02105000--021057FF TARGET   Long          3.   EALIAS=C:02005000 // ID Tags
  C:02105800--02105FFF TARGET   Long          3.   EALIAS=C:02005800 // ID Tags
  C:02106000--021067FF TARGET   Long          3.   EALIAS=C:02006000 // ID Tags
  C:02106800--02106FFF TARGET   Long          3.   EALIAS=C:02006800 // ID Tags
  C:02107000--021077FF TARGET   Long          3.   EALIAS=C:02007000 // ID Tags
  C:02107800--02107FFF TARGET   Long          3.   EALIAS=C:02007800 // ID Tags
```

AutoInc Option

Some FLASH algorithms need additional information to program the onchip FLASH. Typical information are:

- the FLASH control register base address
- a sector number

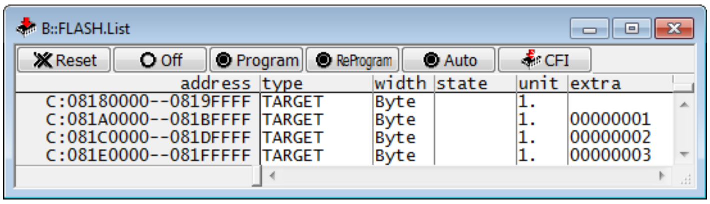
If additional information is required, it is the last parameter of the **FLASH.Create** command.

FLASH.Create *<unit_number> <address_range> <sector_size>* **TARGET** *<bus_width> <add_info>*

The **AutoInc** option allows to shorten the FLASH declaration if increasing sector numbers are needed. The **extra** column in the FLASH.List window shows the sector number as a hex. number.

Example 1:

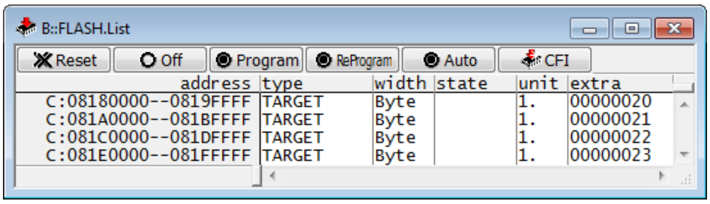
```
FLASH.Create 1. 0x08180000--0x081FFFFF 0x20000 TARGET Byte /AutoInc
```



```
           address type    width state   unit extra
C:08180000--0819FFFF TARGET  Byte          1.
C:081A0000--081BFFFF TARGET  Byte          1.   00000001
C:081C0000--081DFFFF TARGET  Byte          1.   00000002
C:081E0000--081FFFFF TARGET  Byte          1.   00000003
```

Example 2 shows that it is possible to specify the starting sector number:

```
FLASH.Create 1. 0x08180000--0x081FFFFF 0x20000 TARGET Byte 0x20 /AutoInc
```



```
           address type    width state   unit extra
C:08180000--0819FFFF TARGET  Byte          1.   00000020
C:081A0000--081BFFFF TARGET  Byte          1.   00000021
C:081C0000--081DFFFF TARGET  Byte          1.   00000022
C:081E0000--081FFFFF TARGET  Byte          1.   00000023
```

FLASH.TARGET Command

STACKSIZE Option

Target-controlled FLASH programming (for details refer to **“Target-controlled FLASH Programming”**, page 17) uses 256 bytes of stack. If the FLASH programming algorithm requires more stack, the option **STACKSIZE** can be used to define the required stack size.

FLASH.TARGET … **/STACKSIZE** *<size>*

```
FLASH.TARGET 0x0 0x2000 0x1000 lpc4300.bin /STACKSIZE 0x200
```

FirmWareRAM Option

Some processors provide their FLASH programming algorithm in their firmware ROM. The option **FirmWareRAM** can be used to declare the RAM *<address_range>* needed by the firmware FLASH algorithm.

The option **FirmWareRAM** guarantees that the contents of this *<address_range>* is saved before and restored after FLASH programming.

FLASH.TARGET … **/FirmWareRAM** *<address_range>*

```
FLASH.TARGET 0x0 0x2000 0x1000 … /FirmWareRAM 0x10089FF0--0x10089FFF
```

FLASH.CLocK Command

Some onchip FLASHs require a FLASH programming clock within a specified frequency range. The FLASH programming clock is derived from the system clock in most cases. The command **FLASH.CLocK** allows:

- to specify the system clock.

```
FLASH.CLocK 10.MHz
```

- to ask TRACE32 to measure the system clock.

```
FLASH.CLocK AUTO
```

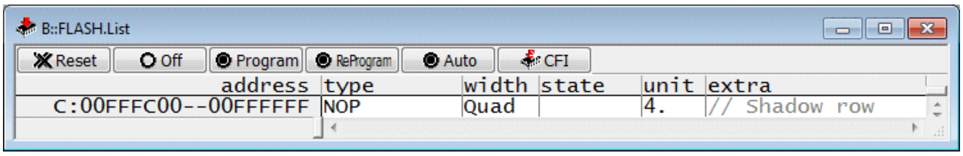
TRACE32 passes the system clock to the FLASH programming algorithm, which is then responsible for deriving the FLASH programming clock.

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

FLASH.CHANGETYPE Command

Sensitive FLASH sectors are declared as NOP sectors to protect them from unintended overwrite.

```
FLASH.Create 4. 0xFFFC00--0xFFFFFF NOP Quad /INFO "Shadow row"
```



```
           address type      width state unit extra
  C:00FFFC00--00FFFFFF NOP   Quad        4.   // Shadow row
```

The following command sequence is recommended if you want to program a sensitive sector:

```
FLASH.CHANGETYPE 0x00FFFC00++0x3FF TARGET        ; change FLASH sector
                                                 ; from NOP sector to
                                                 ; sector programmable by
                                                 ; the FLASH programming
                                                 ; algorithm specified by
                                                 ; the preceding
                                                 ; FLASH.TARGET command

FLASH.Program 0x00FFFC00++0x3FF                  ; enable FLASH sector for
                                                 ; programming

Data.Set …                                       ; program the data
                                                 ; to sector

FLASH.Program off                                ; disable sector for
                                                 ; programming

FLASH.CHANGETYPE 0x00FFFC00++0x3FF NOP           ; change FLASH sector
                                                 ; back to NOP sector
```

FLASH.UNSECUREerase Command

Some chips/processors are secured and require a key-code to allow debugging. Entering the keycode (**SYStem.Option.KEYCODE** *<key_code>*) command unsecures the chip and allows to establish a debug communication. Please refer to your **Processor Architecture** manual for details.

The key-code is for most chips stored in the onchip FLASH.

If the keycode is unknown you can use the command **FLASH.UNSECUREerase**. This command erases the onchip FLASH completely in order to remove the key-code. The chip is unsecured afterwards.

NOTE:	Please be aware that the command **FLASH.UNSECUREerase** is locked if it is not implemented for the selected CPU (**SYStem.CPU** *<cpu>*).

```
SYSTem.CPU MK10DN32VLH5

FLASH.UNSECUREerase

SYStem.Up
```

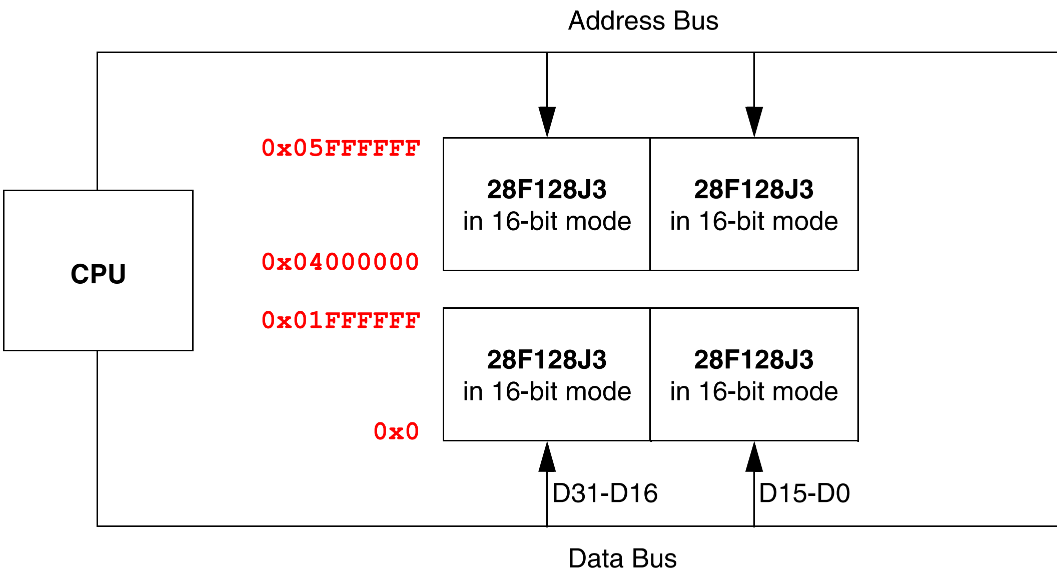
Further Applications for FLASH Declarations Using CFI

Identical FLASH Devices in Series

If two identical FLASH devices are used in series to implement the needed FLASH memory size, the **FLASH.CFI** command has to be performed for each FLASH device.

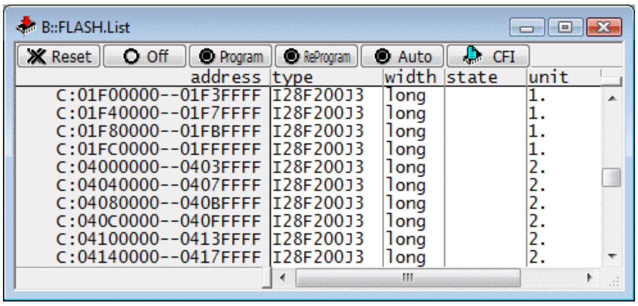
Example:

- Four Intel Strata FLASH devices 28F128J3 in 16-bit mode are used to implement 64 MByte of FLASH memory.
- Therefrom two Intel Strata FLASH devices 28F128J3 are used in parallel to implement a 32-bit data bus.
- Target RAM at 0xa0000000.



FLASH declaration command for TRACE32 tool-based programming:

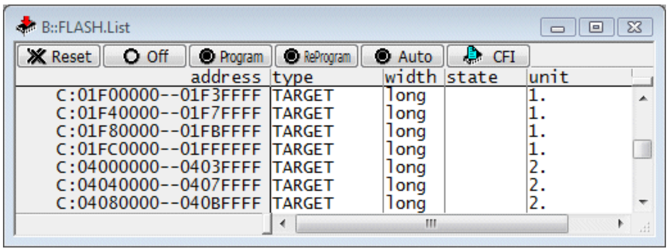
```
FLASH.RESet
; FLASH.CFI <start_address> <bus_width>
FLASH.CFI 0x00000000 Long
FLASH.CFI 0x04000000 Long
```



```
            address type     width state  unit
 C:01F00000--01F3FFFF I28F200J3 long        1.
 C:01F40000--01F7FFFF I28F200J3 long        1.
 C:01F80000--01FBFFFF I28F200J3 long        1.
 C:01FC0000--01FFFFFF I28F200J3 long        1.
 C:04000000--0403FFFF I28F200J3 long        2.
 C:04040000--0407FFFF I28F200J3 long        2.
 C:04080000--040BFFFF I28F200J3 long        2.
 C:040C0000--040FFFFF I28F200J3 long        2.
 C:04100000--0413FFFF I28F200J3 long        2.
 C:04140000--0417FFFF I28F200J3 long        2.
```

FLASH declaration command for target-controlled programming:

```
FLASH.RESet
; FLASH.CFI <start_address> <bus_width> /TARGET <code_range> <data_range>
FLASH.CFI 0x0 Long 0x00000000 /TARGET 0xa0000000++0xfff 0xa0001000++0x1fff
FLASH.CFI 0x0 Long 0x04000000 /TARGET 0xa0000000++0xfff 0xa0001000++0x1fff
```



```
            address type   width state  unit
 C:01F00000--01F3FFFF TARGET long       1.
 C:01F40000--01F7FFFF TARGET long       1.
 C:01F80000--01FBFFFF TARGET long       1.
 C:01FC0000--01FFFFFF TARGET long       1.
 C:04000000--0403FFFF TARGET long       2.
 C:04040000--0407FFFF TARGET long       2.
 C:04080000--040BFFFF TARGET long       2.
```

TRACE32 allocates a so-called *<unit_number>* for each FLASH device. The *<unit_number>* allows to handle each FLASH device separately and to perform a *full chip erase/bulk erase* correctly.

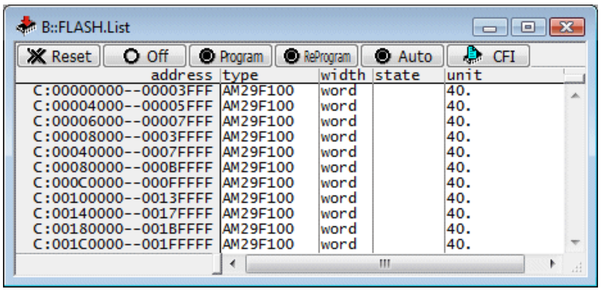
Assigning a Fixed Unit Number

For some scripts it might be helpful to assign a fixed *<unit_number>* to a FLASH device. For these cases the *<unit_number>* can be used as a parameter for the **FLASH.CFI** command.

FLASH.CFI *<unit_number> <start_address> <bus_width>*	*<unit_number>* allows to assign a fixed unit number to a FLASH device

FLASH.CFI *<unit_number> <start_address> <bus_width>* **/TARGET** *<code_range> <data_range>*

```
FLASH.CFI 40. 0x0 Long
```



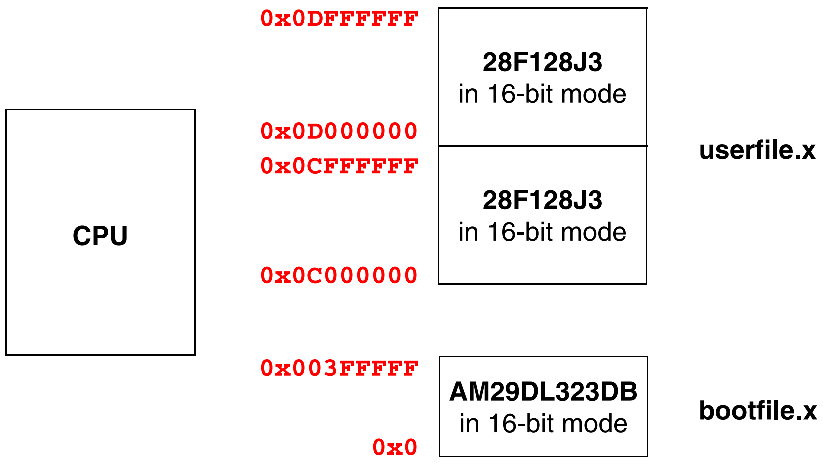
```
               address type        width state  unit
  C:00000000--00003FFF AM29F100    word         40.
  C:00004000--00005FFF AM29F100    word         40.
  C:00006000--00007FFF AM29F100    word         40.
  C:00008000--0003FFFF AM29F100    word         40.
  C:00040000--0007FFFF AM29F100    word         40.
  C:00080000--000BFFFF AM29F100    word         40.
  C:000C0000--000FFFFF AM29F100    word         40.
  C:00100000--0013FFFF AM29F100    word         40.
  C:00140000--0017FFFF AM29F100    word         40.
  C:00180000--001BFFFF AM29F100    word         40.
  C:001C0000--001FFFFF AM29F100    word         40.
```

Heterogeneous FLASH Devices in Series

Since TRACE32 can only handle one external FLASH algorithm at a time, a special proceeding is required if target-controlled FLASH programming is used to program two or more FLASH devices with different FLASH algorithms.

Example 1:

- AM29DL323DB FLASH device in 16-bit mode as boot FLASH
- Two Intel Strata FLASH devices 28F128J3 in 16-bit mode as user FLASH
- Target RAM at 0x00400000
- One programming file per FLASH device



FLASH declaration command for target-controlled programming:

```
; FLASH.CFI <start_address> <bus_width> /TARGET <code_range> <data_range>

; boot FLASH
FLASH.RESet
FLASH.CFI 0x0 Word /TARGET 0x00400000++0xfff 0x00401000++0x1fff
FLASH.ReProgram ALL
Data.LOAD.auto bootfile.x
FLASH.ReProgram off

; user FLASH 1 + 2
FLASH.RESet
FLASH.CFI 0x0c000000 Word /TARGET 0x00400000++0xfff 0x00401000++0x1fff
FLASH.CFI 0x0d000000 Word /TARGET 0x00400000++0xfff 0x00401000++0x1fff
FLASH.ReProgram ALL
Data.LOAD.auto userfile.x
FLASH.ReProgram off
```

No special proceeding is required if TRACE32 tool-based programming is used:

```
FLASH.RESet
; FLASH.CFI <start_address> <bus_width>
FLASH.CFI 0x00000000 Word                              ; boot FLASH
FLASH.CFI 0x0c000000 Word                              ; user FLASH 1
FLASH.CFI 0x0d000000 Word                              ; user FLASH 2

FLASH.ReProgram ALL
Data.LOAD.auto bootfile.x
Data.LOAD.auto userfile.x
FLASH.ReProgram off
```

Example 2:

- On-chip FLASH of TriCore TC1796
- Two AMD FLASH devices Am29BL162CB in parallel to implement a 32-bit data bus
- PMI Scratch-Pad RAM at address 0xD4000000, DMI Scratch-Pad RAM at address 0xD0000000
- One programming file for both FLASHs

FLASH declaration command for target-controlled programming:

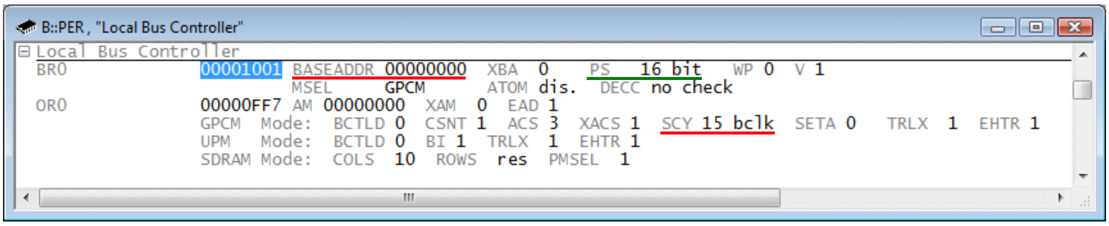
```
; on-chip FLASH
FLASH.RESet
FLASH.Create 1. 0xA0000000++0x1FFFF 0x004000 TARGET Long
FLASH.Create 2. 0xA0020000++0x1FFFF 0x020000 TARGET Long
FLASH.Create 2. 0xA0040000++0x3FFFF 0x040000 TARGET Long
FLASH.Create 2. 0xA0080000++0x7FFFF 0x080000 TARGET Long
FLASH.Create 2. 0xA0100000++0x6FFFF 0x008000 TARGET Long
FLASH.TARGET 0xD4000000++0xFFF 0xD0000000++0x1FFF \
             ~~/demo/tricore/flash/long/tc1796.bin

FLASH.ReProgram ALL
Data.LOAD.Elf demo.elf 0xA0000000++0x16FFFF
FLASH.ReProgram off

; off-chip FLASH
FLASH.RESet
FLASH.CFI 0xA1000000 Long /TARGET 0xD4000000++0xFFF 0xD0000000++0x1FFF

FLASH.ReProgram ALL
Data.LOAD.Elf demo.elf 0xA1000000++0x3FFFFF
FLASH.ReProgram off
```

FLASH.CFI.SIZE(*<address>***,***<bus_width>***)**	Returns the size of single or parallel CFI-conform FLASH devices as a hex. number. Returns 0 if TRACE32 can´t read the CFI information.

```
PRINT FLASH.CFI.SIZE(P:0x0,Word)
```

Expert example for the MPC85xx architecture:

Preconditioned the boot configuration works correctly it is possible to set up the FLASH declaration and the required bus configuration to program the FLASH automatically by a script.



```
⊟ Local Bus Controller
  BR0          00001001 BASEADDR 00000000  XBA  0     PS   16 bit    WP 0  V 1
                        MSEL       GPCM    ATOM dis.   DECC no check
  OR0          00000FF7 AM 00000000  XAM  0  EAD 1
               GPCM  Mode:  BCTLD 0  CSNT 1  ACS 3  XACS 1  SCY 15 bclk  SETA 0   TRLX  1  EHTR 1
               UPM   Mode:  BCTLD 0  BI 1  TRLX  1  EHTR 1
               SDRAM Mode:  COLS  10  ROWS  res  PMSEL  1
```

Bus configuration after reset

1. Configure the start address of the FLASH devices by setting BR0/BASEADDR to 0xff800000 (Boot ROM Location). This setting is preliminary and will be corrected later.

```
&flashbase=0xff800000
Data.Set ANC:iobase()+0x00005000 %Long \
(Data.LONG(ANC:iobase()+0x00005000)&0x00007FFF)|&flashbase
```

2. Read BR0/PS to determine the data *<bus_width>* between the CPU and the FLASH devices.

```
&port_size=(Data.LONG(ANC:iobase()+0x00005000)>>11.)&0x00000003
IF &port_size==1
     &bus_width="BYTE"
ELSE IF &port_size==2
     &bus_width="WORD"
ELSE IF &port_size==3
     &bus_width="LONG"
ELSE
(
     PRINT %ERROR "ERROR: Invalid bus width"
     ENDDO
)
```

3. Determine the flash size via a CFI query.

```
&flash_size=FLASH.CFI.SIZE(ANC:&flashbase,&bus_width)
IF (&flash_size==0)
(
     PRINT %ERROR "ERROR: FLASH module could not be detected"
     ENDDO
)
```

4. Calculate the start address of the FLASH device.

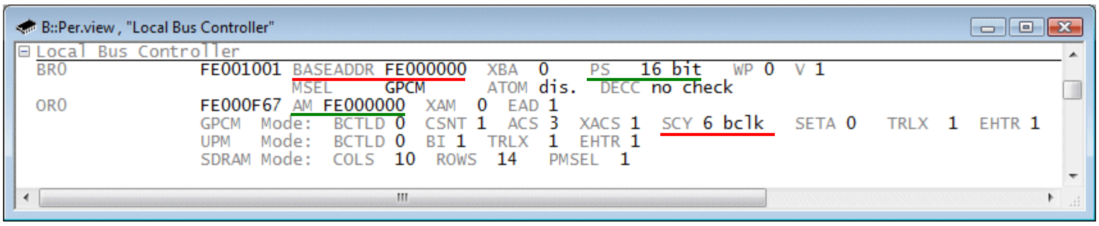
```
&end_address=0xffffffff
&address==&end_address-&flash_size+0x1
```

5. Correct the start address of the FLASH devices by setting BR0/BASEADDR to the calculated &address.

```
Data.Set ANC:iobase()+0x00005000 %Long\
(Data.LONG(ANC:iobase()+0x00005000)&0x00007FFF)|&address
```

6. Reduce the wait states for the FLASH devices to improve the programming performance by setting OR0/SCY to 6.

```
&waitstates=6.
Data.Set ANC:iobase()+0x00005004 %Long \
(Data.LONG(ANC:iobase()+0x00005004)&0x00007F0F)\
|&flashbase|(&waitstates<<4.)
```



```
B::Per.view, "Local Bus Controller"
Local Bus Controller
BR0           FE001001 BASEADDR FE000000  XBA  0     PS    16 bit   WP 0  V 1
                       MSEL      GPCM     ATOM dis.  DECC no check
OR0           FE000F67 AM  FE000000  XAM  0  EAD 1
              GPCM  Mode:  BCTLD 0  CSNT 1  ACS 3  XACS 1  SCY 6 bclk   SETA 0   TRLX  1  EHTR 1
              UPM   Mode:  BCTLD 0  BI 1   TRLX  1  EHTR 1
              SDRAM Mode:  COLS  10  ROWS  14   PMSEL  1
```

Correct bus configuration for the FLASH programming

7. Program the FLASH.

```
; program FLASH device
FLASH.Reset
FLASH.CFI &address &bus_width
FLASH.Erase
FLASH.ReProgram ALL
Data.LOAD.auto * /WORD
FLASH.ReProgram off
```

The script for this example is also available in the TRACE32 folder:
`~~/demo/powerpc/hardware/mpc85xx/all_boards/flash_cfi.cmm`

Truncating the FLASH Size to the CPU Address Space

If the FLASH size is bigger then the address space of the CPU, it is necessary to specify the usable address range for the **FLASH.CFI** command.

Example:

- In order to have more GPIO pins an Infineon XC2xxx CPU is using only 18 of the 24 address lines. Thus the external address space of the CPU is 256 KByte.
- Due to any reason (better availability, smaller packages, better price) a 1 MByte FLASH is used.
- Target RAM at 0x00e00000

FLASH declaration command for TRACE32 tool-based programming:

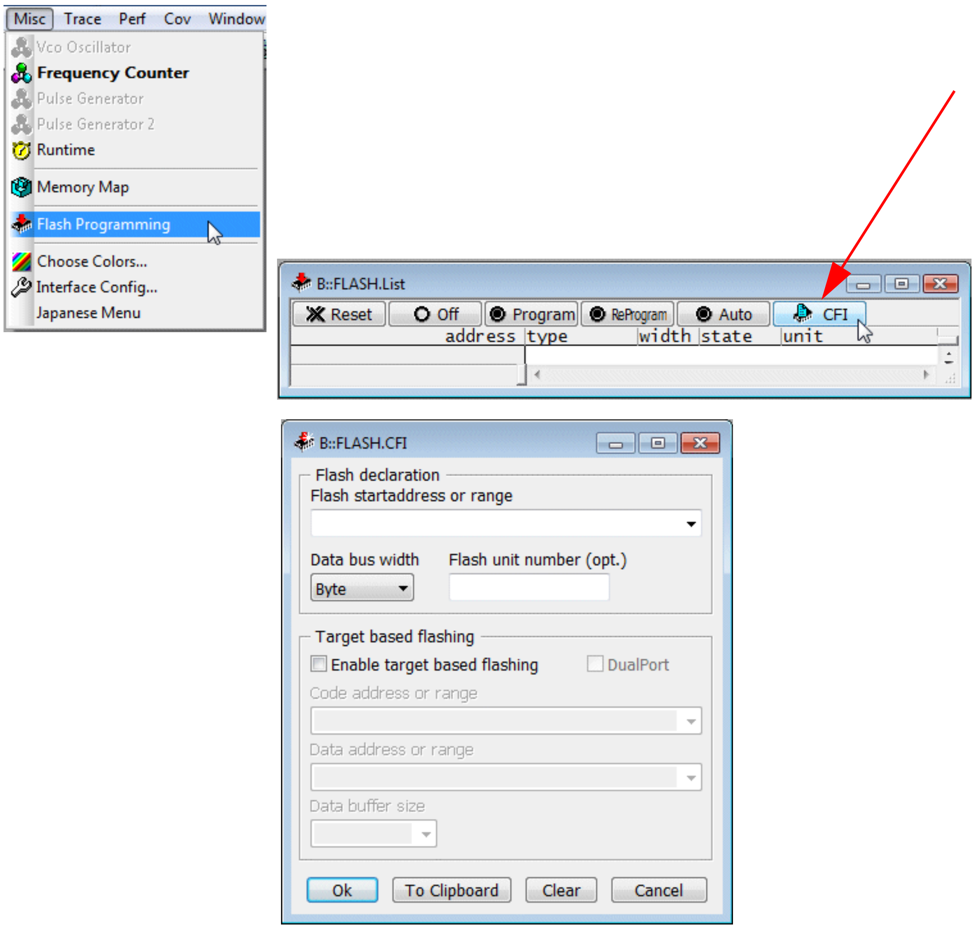
```
FLASH.RESet
; FLASH.CFI <address_range> <bus_width>
FLASH.CFI 0x0x00100000++0x3ffff Word
```

FLASH declaration command for target-controlled programming:

```
; FLASH.CFI <address_range> <bus_width> /TARGET <code_range> <data_range>

FLASH.RESet
FLASH.CFI 0x0x00100000++0x3ffff Word /TARGET 0x00e00000++0xfff 0x00e01000++0x1fff
```

FLASH Declaration via FLASH.CFI Dialog Window

FLASH declaration with CFI is mostly used in scripts so the command line version is more common then the corresponding dialog window.



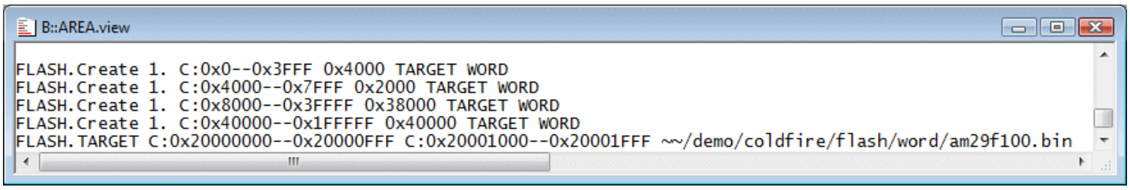
The FLASH.CFI-dialog opens also if the command **FLASH.CFI** is used without parameters.

```
FLASH.CFI                                       ; open FLASH.CFI dialog
```

Generation of Equivalent FLASH.Create Commands

TRACE32 displays the equivalent commands for the manual FLASH declaration in the TRACE32 message area, if the command **FLASH.CFI** is used. This is especially helpful to check which binary file is loaded as FLASH programming algorithm.

```
AREA.CLEAR                                  ; clear the message area

AREA.view                                   ; open the message area

FLASH.CFI 0x0 Word /TARGET 0x20000000++0xfff 0x20001000++0xfff
```



```
FLASH.Create 1. C:0x0--0x3FFF 0x4000 TARGET WORD
FLASH.Create 1. C:0x4000--0x7FFF 0x2000 TARGET WORD
FLASH.Create 1. C:0x8000--0x3FFFF 0x38000 TARGET WORD
FLASH.Create 1. C:0x40000--0x1FFFFF 0x40000 TARGET WORD
FLASH.TARGET C:0x20000000--0x20000FFF C:0x20001000--0x20001FFF ~~/demo/coldfire/flash/word/am29f100.bin
```

A detailed description of the **FLASH.Create** command is given in the next chapter.

Declarations for not CFI-conform FLASH Devices

Not CFI-conform FLASH devices require that all characteristics are provided in the FLASH declaration.

Manual FLASH Declaration (TRACE32 Tool-based)

FLASH.Create *<unit_number> <address_range> <sector_size> <family_code> <bus_width>*

Parameters:

- *<unit_number>*

  TRACE32 maintains each FLASH device by its own *<unit_number>*.

- *<address_range>*

  Specifies the address range of the FLASH devices.

- *<sector_size>*

  Specifies the size of the individual sectors within the FLASH devices.

- *<family_code>*

  Specifies the TRACE32 tool-based programming algorithm. The FLASH device and the corresponding *<family_code>* is listed under **"List of Supported FLASH Devices"** (flashlist.pdf).

- *<bus_width>*

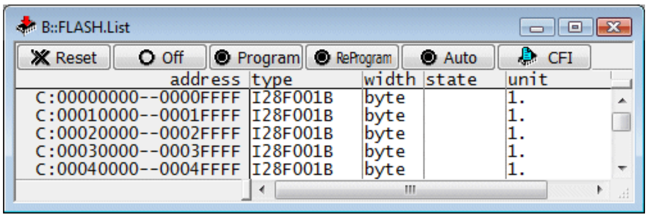
  Defines the width of the data bus between the target CPU and the FLASH devices.

The syntax for the manual FLASH declaration for target-controlled programming is described in **"Converting TRACE32 Tool-based to Target-controlled FLASH Programming"** in Onchip/NOR FLASH Programming User's Guide, page 85 (norflash.pdf).

Example:

- Sharp LH28F016 FLASH device
- 32 sectors each with 64 KBytes
- 8-bit FLASH

```
FLASH.RESet

; FLASH.Create <unit_number> <address_range> <sector_size> <family_code> <bus_width>

FLASH.Create 1. 0x0--0x1fffff 0x10000 I28F001B Byte
```



```
          address type      width state    unit
 C:00000000--0000FFFF I28F001B byte           1.
 C:00010000--0001FFFF I28F001B byte           1.
 C:00020000--0002FFFF I28F001B byte           1.
 C:00030000--0003FFFF I28F001B byte           1.
 C:00040000--0004FFFF I28F001B byte           1.
```

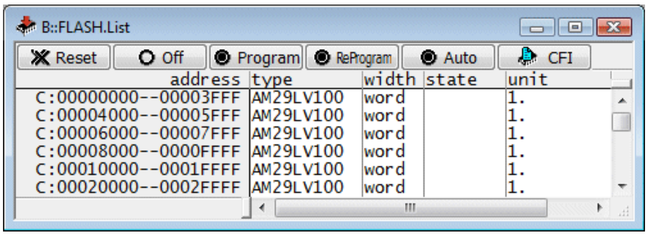
FLASH Devices with Sectors of Different Size

If a FLASH device contains sectors of different size an extra **FLASH.Create** command has to be used for each address range with the same *<sector_size>*. TRACE32 knows that all the commands are related to a single FLASH device if the same *<unit_number>* is used with all **FLASH.Create** commands.

Example:

- Spansion S29AL008D bottom boot sector device
- 1 sector with 16 KByte, 2 sectors with 8 KByte, 1 sector with 32 KByte, 15 sectors with 64 KByte
- 16-bit mode

```
FLASH.RESet

; FLASH.Create <unit_number> <address_range> <sector_size> <family_code> <bus_width>

FLASH.Create 1. 0x00000--0x03fff 0x04000 AM29LV100 Word
FLASH.Create 1. 0x04000--0x07fff 0x02000 AM29LV100 Word
FLASH.Create 1. 0x08000--0x0ffff 0x08000 AM29LV100 Word
FLASH.Create 1. 0x10000--0xfffff 0x10000 AM29LV100 Word
```

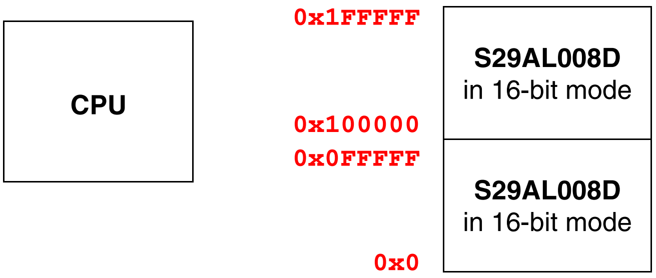


```
          address type      width state    unit
C:00000000--00003FFF AM29LV100 word           1.
C:00004000--00005FFF AM29LV100 word           1.
C:00006000--00007FFF AM29LV100 word           1.
C:00008000--0000FFFF AM29LV100 word           1.
C:00010000--0001FFFF AM29LV100 word           1.
C:00020000--0002FFFF AM29LV100 word           1.
```

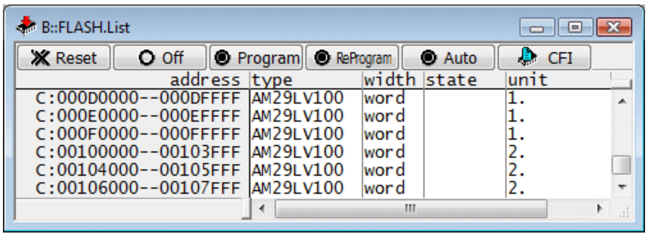
If two or more identical FLASH devices are used in series to implement the needed FLASH memory size each FLASH device has to be declared with a different *<unit_number>*.

Example:

- Two Spansion S29AL008D bottom boot sector devices
- Each providing 1 sector with 16 KByte, 2 sectors with 8 KByte, 1 sector with 32 KByte, 15 sectors with 64 KByte
- 16-bit mode



```
FLASH.RESet

; FLASH.Create <unit_number> <address_range> <sector_size> <family_code> <bus_width>

; declaration for the first FLASH device
FLASH.Create 1. 0x000000--0x003fff 0x04000 AM29LV100 Word
FLASH.Create 1. 0x004000--0x007fff 0x02000 AM29LV100 Word
FLASH.Create 1. 0x008000--0x00ffff 0x08000 AM29LV100 Word
FLASH.Create 1. 0x010000--0x0fffff 0x10000 AM29LV100 Word

; declaration for the second FLASH device
FLASH.Create 2. 0x100000--0x103fff 0x04000 AM29LV100 Word
FLASH.Create 2. 0x104000--0x107fff 0x02000 AM29LV100 Word
FLASH.Create 2. 0x108000--0x10ffff 0x08000 AM29LV100 Word
FLASH.Create 2. 0x110000--0x1fffff 0x10000 AM29LV100 Word
```



```
           address  type       width state      unit
C:000D0000--000DFFFF AM29LV100 word             1.
C:000E0000--000EFFFF AM29LV100 word             1.
C:000F0000--000FFFFF AM29LV100 word             1.
C:00100000--00103FFF AM29LV100 word             2.
C:00104000--00105FFF AM29LV100 word             2.
C:00106000--00107FFF AM29LV100 word             2.
```

FLASH Devices in Parallel

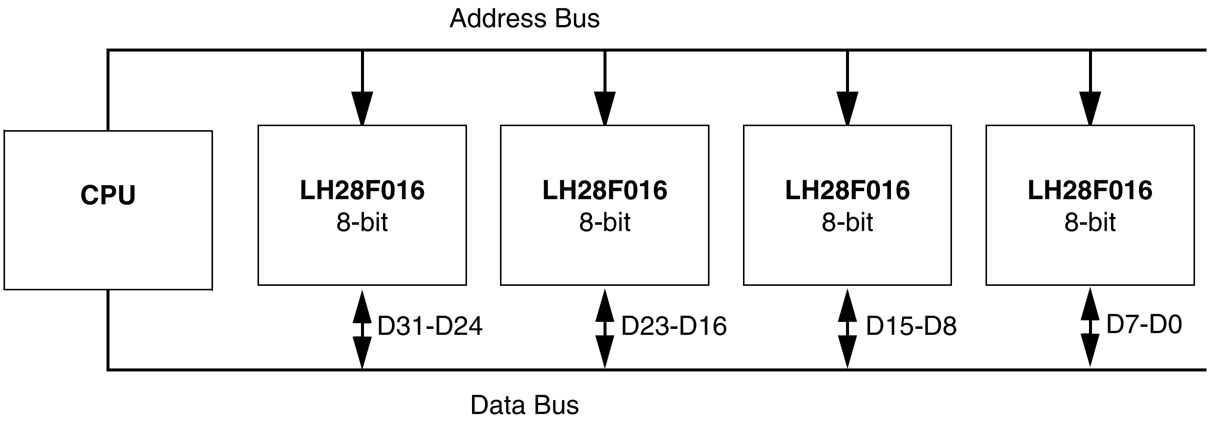
If two or more identical FLASH devices are used in parallel to implement the needed data bus width, the FLASH declaration has to be performed as follows:

n identical FLASH devices in parallel

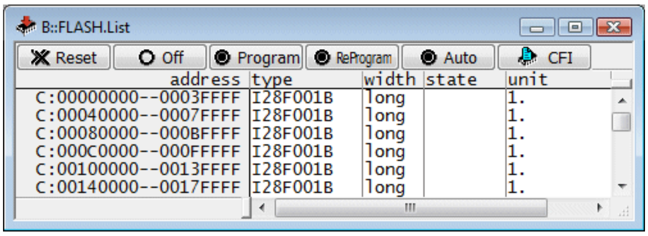
- *<address_range>* = n x *<address_range_of_single_device>*
- *<sector_size>* = n x *<sector_size_of_single_device>*
- *<bus_width>* = n x *<bus_width_of_single_device>*

Example:

- Four 8-bit Sharp LH28F016 FLASH devices are used in parallel to implement a 32-bit data bus
- Each providing 32 sectors with 64 KBytes



```
FLASH.RESet

; FLASH.Create <unit_number> <address_range> <sector_size> <family_code> <bus_width>
; <address_range> = 4 x 0x200000 = 0x800000
; <sector_size> = 4 x 0x10000 = 0x40000
; <bus_width> = 4 x 8 bit = Long

FLASH.Create 1. 0x0--0x7fffff 0x40000 I28F001B Long
```



```
             address type        width state     unit
 C:00000000--0003FFFF I28F001B    long            1.
 C:00040000--0007FFFF I28F001B    long            1.
 C:00080000--000BFFFF I28F001B    long            1.
 C:000C0000--000FFFFF I28F001B    long            1.
 C:00100000--0013FFFF I28F001B    long            1.
 C:00140000--0017FFFF I28F001B    long            1.
```

- Declare each FLASH device with **all** sectors.

  TRACE32 is using *full chip erase/bulk erase* if possible. In doing so also not declared FLASH sectors are erased.

- Use the same unit number for all sector declarations applying to the same FLASH device.

- If two or more identical FLASH devices are used in parallel to implement the needed data bus width with the CPU, use the same unit number and calculate the parameters for the **FLASH.Create** command as follows:

  n identical FLASH devices in parallel

  <address_range> = n x *<address_range_of_single_device>*

  <sector_size> = n x *<sector_size_of_single_device>*

  <bus_width> = n x *<bus_width_of_single_device>*

- If two or more FLASH devices are used in series to implement the needed FLASH memory size, declare each FLASH device with its own unit number.

  TRACE32 is using *full chip erase/bulk erase* if possible. In doing so only the sectors within the first FLASH device are erased, even if other FLASH sectors are declared with the same unit number.

TRACE32 Tool-based vs. Target-controlled FLASH Programming

TRACE32 provides two techniques to program off-chip FLASH devices:

- TRACE32 tool-based programming

  The FLASH programming algorithm is part of the TRACE32 software and runs on the host.

- Target-controlled programming

  The FLASH programming algorithm is not part of the TRACE32 software. It is linked to TRACE32 and downloaded to the target RAM if required.

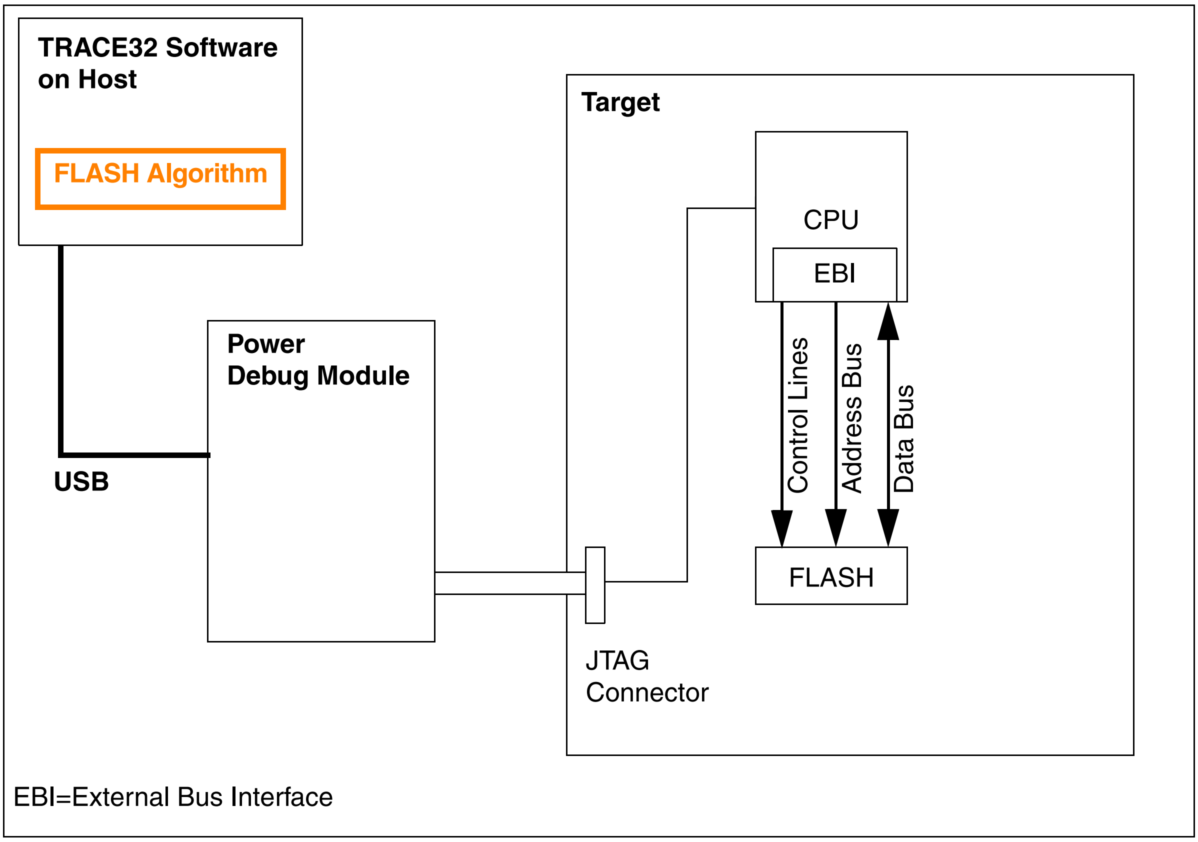
TRACE32-tool based programming	**Target-controlled programming**
Simple to set up because no target resources are required	Simple setup, but the usage of more target resources adds sources of errors
Slow	Very fast
Update of FLASH programming algorithm requires complete TRACE32 software update	FLASH programming algorithm can be updated independently from the TRACE32 software
	Very flexible, every FLASH device can be supported. If required refer to **“How to Write your own FLASH Algorithm”** (flash_app_own_algorithm.pdf)
	Specifics in the target design (e.g. switched data lines) can be corrected by the FLASH algorithm.

Despite the obvious disadvantages of TRACE32 tool-based programming it is recommended to start with this technique, because errors are less likely since no target resources are required.

If TRACE32 tool-based FLASH programming runs faultless, you can almost be sure that:

- The bus configuration registers for the FLASH devices are set up correctly.
- The interface between the CPU and the FLASH devices on your target hardware works faultless.
- TRACE32 can erase and program the FLASH devices correctly.

After TRACE32 tool-based FLASH programming works correctly, you can easy migrate to the faster target-controlled FLASH programming. The migration procedure is described in **“Converting TRACE32 Tool-based to Target-controlled FLASH Programming”** in Onchip/NOR FLASH Programming User’s Guide, page 85 (norflash.pdf).



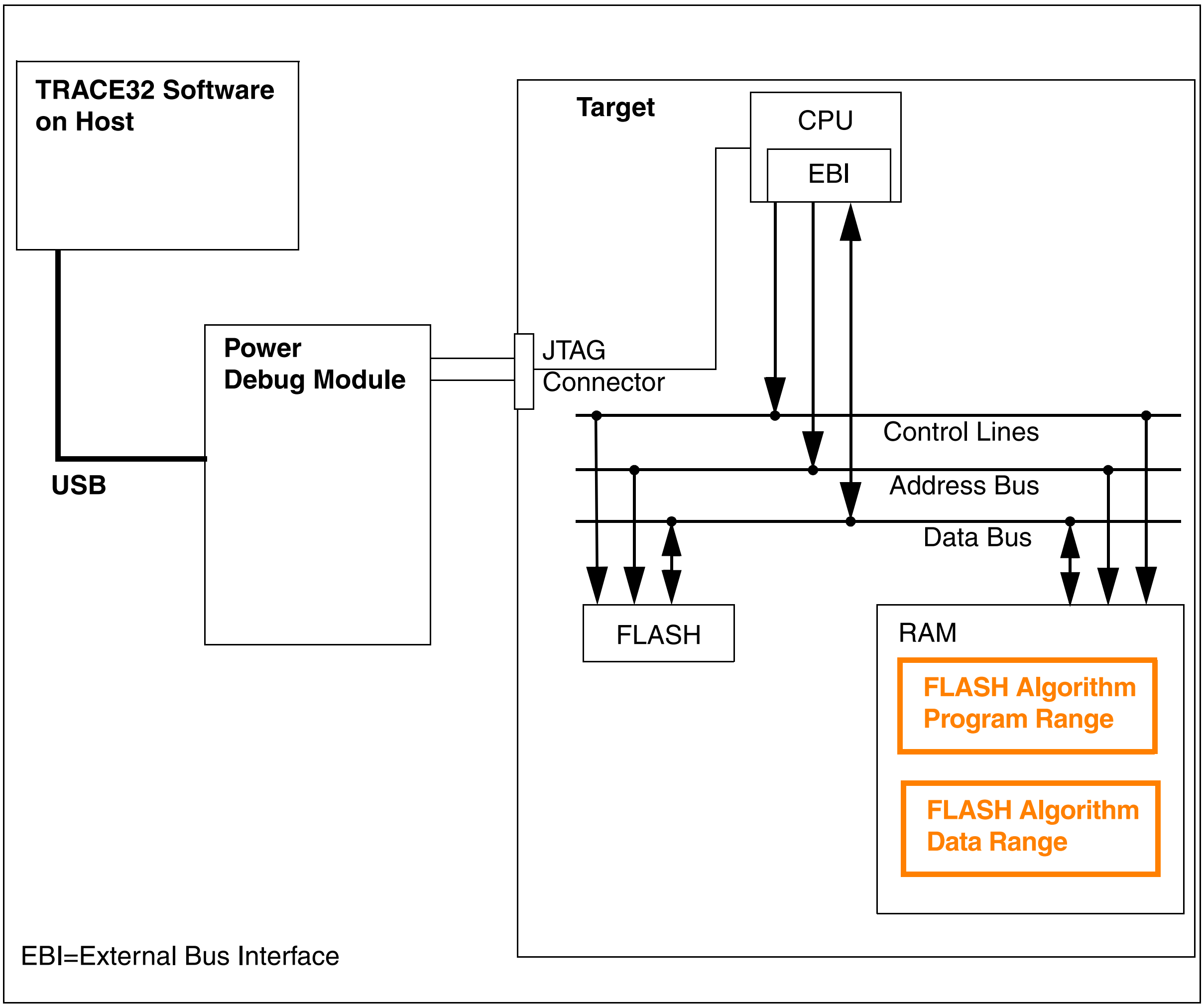
The FLASH declaration requires only information on the FLASH devices since the FLASH algorithm is integrated into the TRACE32 software.
FLASH declaration commands:

1. FLASH declaration via CFI query

```
FLASH.CFI <start_address> <bus_width>
```

2. FLASH declaration for not CFI-conform FLASH devices

```
FLASH.Create <unit_number> <address_range> <sector_size> <family_code> <bus_width>
```



If target-controlled FLASH programming is used, the FLASH algorithm is not part of the TRACE32 software. FLASH programming works now in principle as follows:

If any TRACE32 command is used that unlocks or locks, erases, programs the FLASH devices:

- The TRACE32 software is saving the RAM contents of *FLASH Algorithm Program/Data Rang*e.
- The TRACE32 software is saving the register context.
- The TRACE32 software is loading the external FLASH algorithm to the *FLASH Algorithm Program Range* and sets a software breakpoint at the exit of the FLASH algorithm.

To execute any action on the FLASH device (unlock or lock, block erase, chip erase, program) by the FLASH algorithm

1. The TRACE32 software is loading the argument buffer for the FLASH algorithm.
2. The TRACE32 software is loading the data to *FLASH Programming Data Range*.
3. The PC, stack pointer and the registers for the argument passing are set.
4. The external FLASH algorithm is started.
5. After the software breakpoint at the exit of the FLASH algorithm is reached, the TRACE32 software checks if there are any further actions to perform. Step 1 - 4 are repeated until all actions are performed.

After the TRACE32 FLASH command is done:

- The TRACE32 software is restoring the contents of the *FLASH Algorithm Program/Data Range*.
- The TRACE32 software is restoring the register context.

Only one external FLASH algorithm can be used at a time.

This procedure requires additional setups in order to program off-chip NOR FLASH devices. This includes:

- The definition of the external FLASH programming algorithm
- The definition of the *FLASH Algorithm Program Range*
- The definition of the *FLASH Algorithm Data Range*
- The definition of the maximum number of bytes that are transferred from the TRACE32 software.

Before these requirements are described in detail a short command overview:

FLASH declaration commands for CFI-conform FLASH devices:

FLASH.CFI *<start_address> <bus_width>* **/TARGET** *<code_range> <data_range>*

FLASH.CFI *<start_address> <bus_width>* **/TARGET** *<code_address> <data_address>* [*<buffer_size>*]

FLASH declaration commands for not CFI-conform FLASH devices:

FLASH.Create *<unit_number> <address_range> <sector_size>* **TARGET** *<bus_width>*
FLASH.TARGET *<code_range> <data_range> <file>*

FLASH.Create *<unit_number> <address_range> <sector_size>* **TARGET** *<bus_width>*
FLASH.TARGET *<code_address> <data_address>* [*<buffer_size>*] *<file>*

A binary file is used as external FLASH algorithm.

Ready-to-run binary files for target-controlled FLASH programming are available for the most common processor architectures in the folder `~~/demo/`**`<architecture>`**`/flash`; where `~~` is expanded to the `<TRACE32_installation_directory>`, which is `c:\T32` by default.

CFI-conform FLASH devices

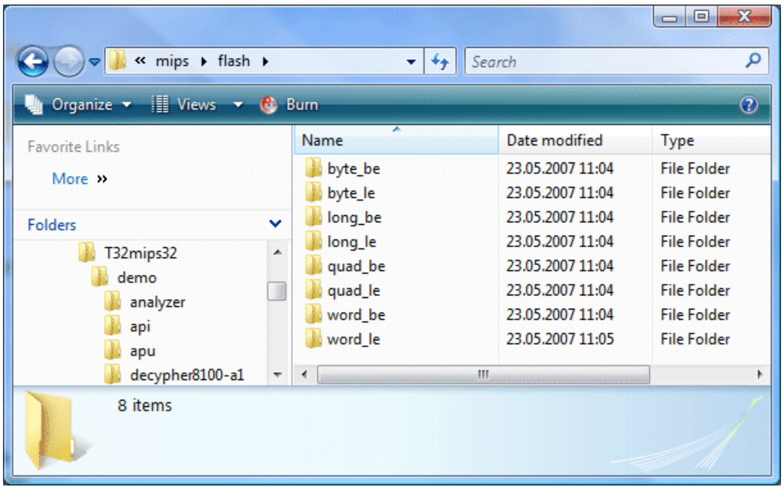
TRACE32 loads the appropriate FLASH programming algorithm automatically from `~~/demo/`**`<architecture>`**`/flash` when target-controlled FLASH programming for CFI-conform FLASH devices is used.

Not CFI-conform FLASH devices

The file name and the path for the FLASH programming algorithm needs to be specified explicitly for not CFI-conform FLASH devices.

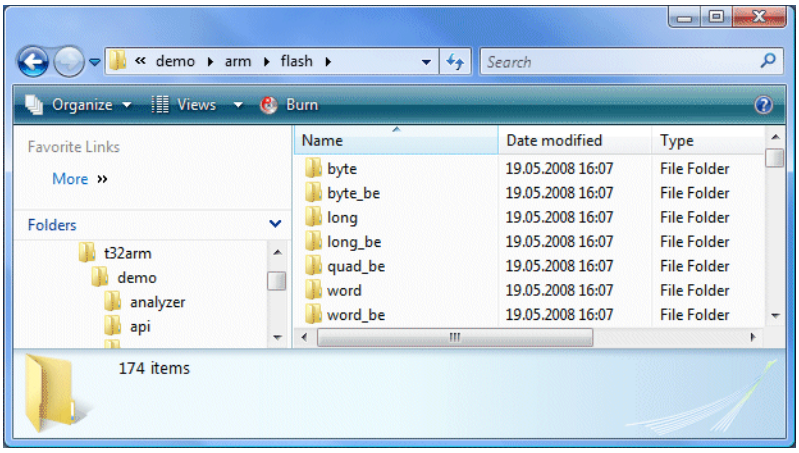
In the directory `~~/demo/`**`<architecture>`**`/flash` the FLASH algorithms are organized by *<bus_width>* and by *<endianness>.*

- *<bus_width>*_**be** stands for FLASH support for big endian mode.
- *<bus_width>*_**le** stands for FLASH support for little endian mode.



Name	Date modified	Type
byte_be	23.05.2007 11:04	File Folder
byte_le	23.05.2007 11:04	File Folder
long_be	23.05.2007 11:04	File Folder
long_le	23.05.2007 11:04	File Folder
quad_be	23.05.2007 11:04	File Folder
quad_le	23.05.2007 11:04	File Folder
word_be	23.05.2007 11:04	File Folder
word_le	23.05.2007 11:05	File Folder

If your processor architecture has a preferred endianness, this *<endianness>* is left out and only the *<bus_width>* is listed. The preferred endianness for the ARM architecture as an example is little endian mode.



Name	Date modified	Type
byte	19.05.2008 16:07	File Folder
byte_be	19.05.2008 16:07	File Folder
long	19.05.2008 16:07	File Folder
long_be	19.05.2008 16:07	File Folder
quad_be	19.05.2008 16:07	File Folder
word	19.05.2008 16:07	File Folder
word_be	19.05.2008 16:07	File Folder

The name of the binary file for the FLASH algorithm corresponds to the name listed in the CODE column for the FLASH device in **“List of Supported FLASH Devices”** (flashlist.pdf).

Example 1:

- MIPS32 CPU in big endian mode
- Intel Strata FLASH 28L256L30
- 16-bit data bus width between CPU and FLASH

Programming algorithm: `~~/demo/`**`mips`**`/flash/`**`word_be`**`/`**`i28f200k3.bin`**

Example 2:

- ARM9 CPU in little endian mode
- AM29DL323C FLASH
- 16-bit data bus width between CPU and FLASH

Programming algorithm: `~~/demo/`**`arm`**`/flash/`**`word`**`/`**`am29lv100.bin`**

If the binary file for the FLASH algorithm is not provided in `~~/demo/`**`<architecture>`**`/flash` please contact **flash-support@lauterbach.com**.

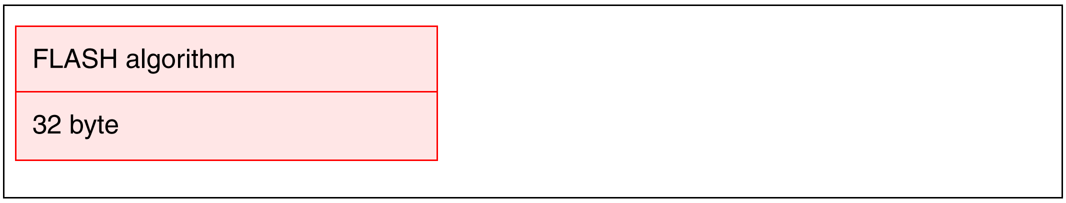
If you would like to write your own FLASH programming algorithm, please refer to the application note **“How to Write your own FLASH Algorithm”** (flash_app_own_algorithm.pdf).

If target-controlled FLASH programming is used TRACE32:

1. Downloads the external FLASH algorithm to the target RAM (*FLASH Algorithm Program Range*).
2. Downloads the programming data to the target RAM (*FLASH Algorithm Data Range*).

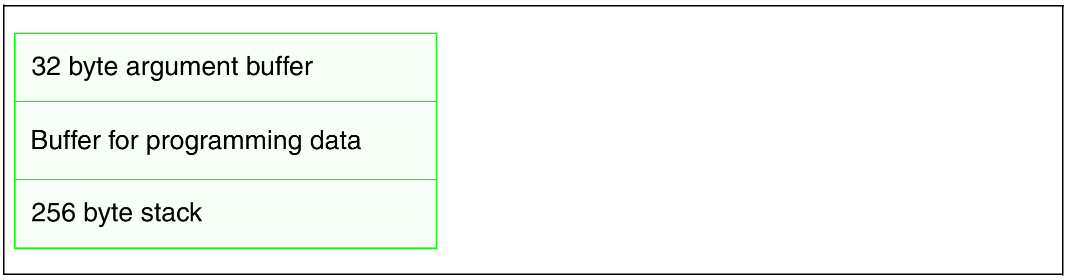
This proceeding requires the specification of both address ranges in the FLASH declaration.

Memory mapping for the *FLASH Algorithm Program Range*:



Required size for the external FLASH algorithm is `size_of(`*<flash_algorithm>*`) + 32 byte`

Memory mapping for the *FLASH Algorithm Data Range*:



- The **argument buffer** used for the communication between the TRACE32 software and the FLASH algorithm is located at the first 32 bytes of the *FLASH Algorithm Data Range*.
- The 256 byte **stack** is located at the end of the *FLASH Algorithm Data Range*.
- The size of the **buffer for programming data** (*<buffer_size>*) specifies the maximum number of bytes that are transferred from the TRACE32 software to the external FLASH programming algorithm in one call.

TRACE32 supports two formats to provide this information.

Format 1: *<code_range> <data_range>*

```
; FLASH.CFI <start_address> <bus_width> /TARGET <code_range> <data_range>
FLASH.CFI 0x0 Word /TARGET 0x20000000++0x7ff 0x20001000++0xfff
```

- *<code_range>*

  TRACE32 downloads the external FLASH algorithm to *<code_range>*.

- *<data_range>*

  TRACE32 loads the programming data to *<data_range>* in the target RAM.

- The maximum number of bytes that are transferred from the TRACE32 software to the external FLASH programming algorithm in one call is calculated out of the *<data_range>* as follows:

  `size_of(`*<data_range>*`) - 32 byte argument buffer - 256 byte stack`

Format 2: *<code_address> <data_address> <buffer_size>*

```
; FLASH.CFI <start_address> <bus_width> \
;/TARGET <code_address> <data_address> <buffer_size>
FLASH.CFI 0x0 Word /TARGET 0x20000000 0x20001000 0x4000
```

- *<code_address>*

  TRACE32 loads the external FLASH algorithm to the target RAM starting at *<code_address>*.

- *<data_address>*

  TRACE32 loads the programming data to the target RAM starting at *<data_address>*.

- *<buffer_size>* specifies the maximum number of bytes that are transferred from the TRACE32 software to the external FLASH programming algorithm in one call.

  If *<buffer_size>* is not specified 4 KByte is used by default.

CFI-conform FLASH Devices

Conversion from TRACE32 tool-based FLASH declaration to target-controlled FLASH declaration:

Add the option **/TARGET** and the information about the *FLASH Algorithm Program/Data Range* to the **FLASH.CFI** command.

```
&FLASH_PROGRAMMING_METHOD="Tool-based"
; &FLASH_PROGRAMMING_METHOD="Target-controlled"

FLASH.RESet
(
IF "&FLASH_PROGRAMMING_METHOD"=="Tool-based"

     ; FLASH.CFI <start_address> <bus_width>
     FLASH.CFI 0x0 Word
)

IF "&FLASH_PROGRAMMING_METHOD"=="Target-controlled"
(
 ; FLASH.CFI <start_address> <bus_width>/TARGET <code_range> <data_range>
 FLASH.CFI 0x0 Word /TARGET 0x20000000++0xfff 0x20001000++0xfff
)
```

Conversion from TRACE32 tool-based FLASH declaration to target controlled FLASH declaration:

1. Replace the *<family_code>* by the keyword **TARGET** when using the **FLASH.Create** command.

2. Use the **FLASH.TARGET** command to specify the *<code_range>*, *<data_range>* and the *<flash_algorithm>*. Please remember to use the FLASH algorithm from the directory with the adequate *<bus_width>* and the correct *<endianness>*. The name of *<flash_algorithm>* matches with the *<family_code>*.

```
&FLASH_PROGRAMMING_METHOD="Tool-based"
; &FLASH_PROGRAMMING_METHOD="Target-controlled"

FLASH.RESet
(
IF "&FLASH_PROGRAMMING_METHOD"=="Tool-based"

  ; FLASH.Create <unit_number> <address_range> <sector_size> <family_code> <bus_width>
  FLASH.Create 1. 0x0++0x3fffff 0x20000 I28F200B Long
)

IF "&FLASH_PROGRAMMING_METHOD"=="Target-controlled"
(
  ; FLASH.Create <unit_number> <address_range> <sector_size> TARGET <bus_width>
  FLASH.Create 1. 0x0++0x3fffff 0x20000 TARGET Long
  ; FLASH.Target <code_range> <data_range> <flash_algorithm>
  FLASH.TARGET 0x20000000++0xfff 0x20001000++0xfff ~~/demo/arm/flash/long_be/i28f200b.bin
)
```

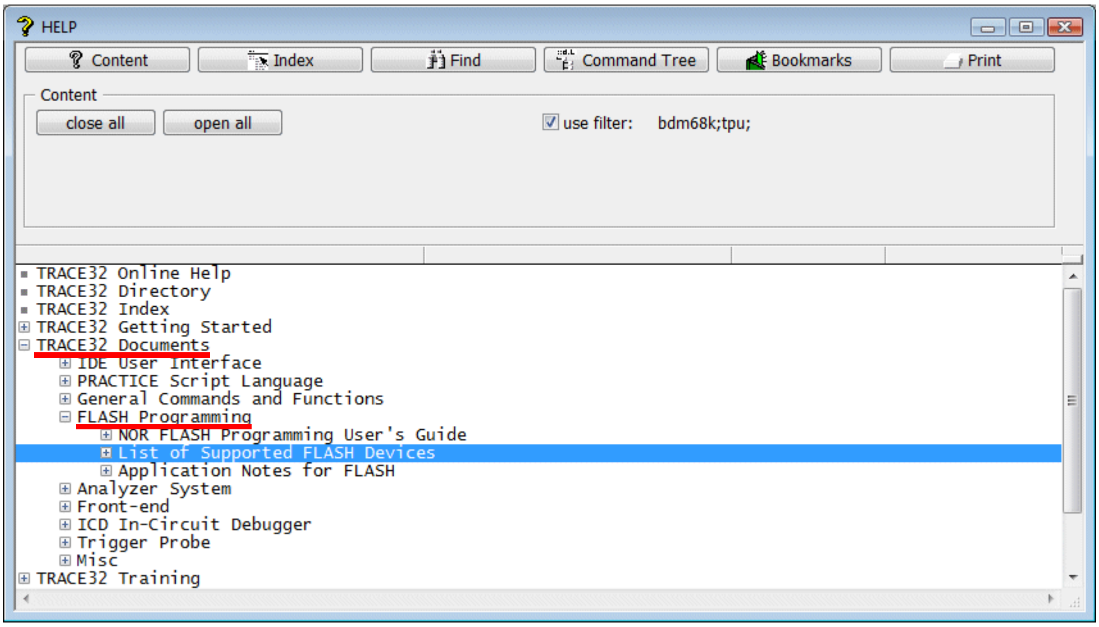
Maintaining the Declared FLASH Devices

TRACE32 maintains all declared FLASH devices in the so-called **FLASH declaration table**. The following commands are provided:

FLASH.List	List the contents of the FLASH declaration table
FLASH.REset	Clear the FLASH declaration table and reset all FLASH programming setups within TRACE32 to its default value
FLASH.Delete ALL | *<range>* | *<unit_number>*	Remove entries from the FLASH declaration table

List of Supported FLASH Devices

A list of all supported FLASH devices plus the corresponding *<family_code>* can be found in the online help under:

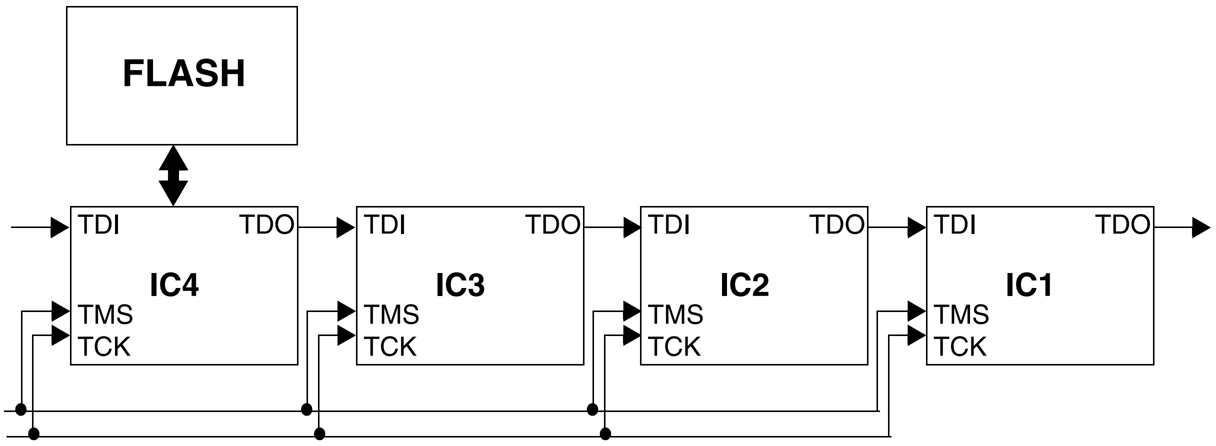


```
■ TRACE32 Online Help
■ TRACE32 Directory
■ TRACE32 Index
⊞ TRACE32 Getting Started
⊟ TRACE32 Documents
    ⊞ IDE User Interface
    ⊞ PRACTICE Script Language
    ⊞ General Commands and Functions
    ⊟ FLASH Programming
        ⊞ NOR FLASH Programming User's Guide
        ⊞ List of Supported FLASH Devices
        ⊞ Application Notes for FLASH
    ⊞ Analyzer System
    ⊞ Front-end
    ⊞ ICD In-Circuit Debugger
    ⊞ Trigger Probe
    ⊞ Misc
⊞ TRACE32 Training
```

An up-to-date list of all supported FLASH devices is always available on our web-page under:
https://www.lauterbach.com/ylist.html.

FLASH Programming via Boundary Scan

External NOR FLASH memories can be programmed via boundary scan, if the FLASH memory is connected to an IC with a boundary scan chain and this boundary scan chain is accessible to the debugger. After initializing the boundary scan FLASH mode, the tool based FLASH programming is used. All FLASH commands like **FLASH.CFI**, **FLASH.Program**, ... can be used (target based programming is not possible!).



To avoid disturbances of the FLASH programming due to communication between the debugger and the target CPU, the system should be set to down state:

```
SYStem.Down
```

Boundary scan chain configuration

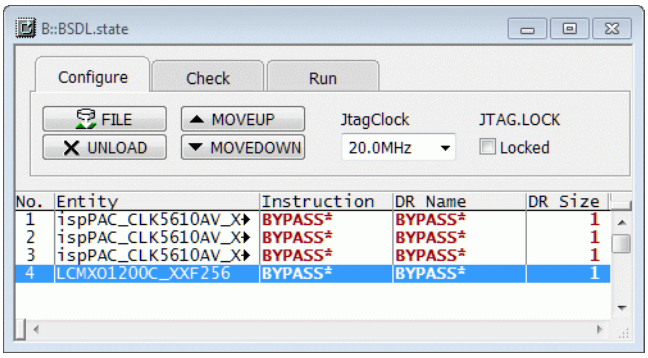
The first step for FLASH programming via boundary scan is the scan chain configuration. Configuration is done by loading the BSDL files in the right order. The following commands are available for the scan chain configuration:

BSDL.UNLOAD ALL	Removes all previous boundary scan chain configurations
BSDL.FILE *<file>*	Loads a BSDL file

The BSDL file for the IC, which is closest to the board TDO connector, must be loaded first:

```
BSDL.UNLOAD ALL                        ; remove previous configuration

BSDL.FILE ispCLOCK5610Av_isc.bsm       ; load the BSDL file for IC1
BSDL.FILE ispCLOCK5610Av_isc.bsm        ; load the BSDL file for IC2

BSDL.FILE ispCLOCK5610Av_isc.bsm        ; load the BSDL file for IC3

BSDL.FILE LCMXCO1200C_ftBGA256.bsdl     ; load the BSDL file for IC4
```

The scan chain configuration can be viewed in the **BSDL.state** window:



```
No. Entity              Instruction DR Name  DR Size
 1  ispPAC_CLK5610AV_X> BYPASS*     BYPASS*        1
 2  ispPAC_CLK5610AV_X> BYPASS*     BYPASS*        1
 3  ispPAC_CLK5610AV_X> BYPASS*     BYPASS*        1
 4  LCMXO1200C_XXF256   BYPASS*     BYPASS*        1
```

With the commands **BSDL.BYPASSall** and **BSDL.IDCODEall** (or the functions **bsdl.check.bypass()** and **bsdl.check.idcode()** in a PRACTICE script) the correct function of the boundary scan chain can be verified.

FLASH interface definition

The FLASH interface definition is done with the two commands:

BSDL.FLASH.IFDefine	Defines the FLASH memory interface
BSDL.FLASH.IFMap	Maps the generic FLASH ports to the driving IC ports

The definition requires the number of the IC in the boundary scan chain, to which the FLASH memory is connected, the number of address and data ports.

```
BSDL.FLASH.IFDefine RESet               ; remove previous configuration

BSDL.FLASH.IFDefine NOR 4. 24. 16.      ; defines a NOR FLASH memory with
                                        ; 24 address bits and 16 data
                                        ; bits
                                        ; the FLASH is connected to IC4
```

```
BSDL.FLASH.IFMap CE PR7C                ; map the FLASH chip enable port
                                        ; to pin PR7C of IC4

BSDL.FLASH.IFMap OE PR7D                ; map the FLASH output enable
                                        ; port to port PR7D of IC4

BSDL.FLASH.IFMap A0 PB7E                ; map the FLASH address bit 0
                                        ; to port PB7E of IC4
```

If two or more FLASH memories are used in parallel, up to 4 sets of control ports are provided (e.g. CE, CE2, CE3 and CE4).

The FLASH interface configuration can be checked with the command **BSDL.FLASH.IFCheck** (or **bsdl.check.flashconf()** in a PRACTICE script).

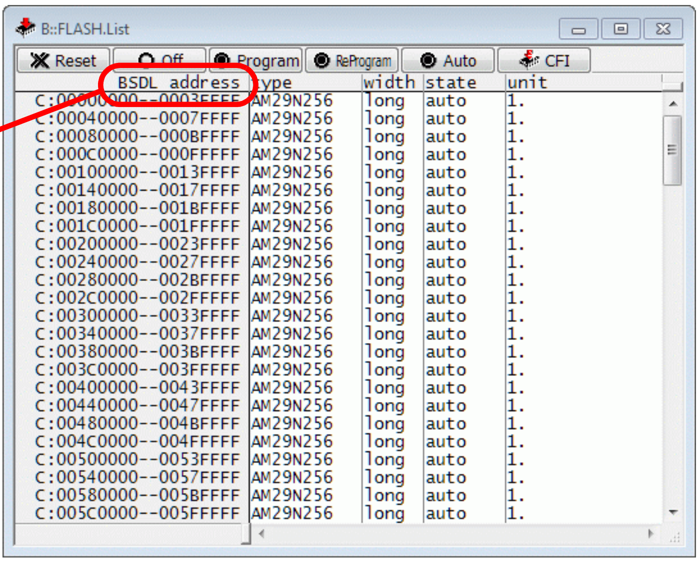
FLASH Programming

To start the FLASH programming, the boundary scan chain must be initialized and the boundary scan FLASH mode must enabled:

BSDL.FLASH.INIT SAFE	Initializes the boundary scan chain
FLASH.BSDLACCESS ON	Switch to boundary scan mode for FLASH access

```
BSDL.FLASH.INIT SAFE        ; Initializes the boundary scan chain: sets the
                            ; required instructions (BYPASS or EXTEST),
                            ; set all ports, which are not used for FLASH
                            ; programming to SAFE state according BSDL
                            ; file and sets the FLASH ports to idle state

FLASH.RESET                 ; remove previous FLASH configuration

FLASH.BSDLaccess ON         ; switch to boundary scan FLASH mode
```

Title changes
to “BSDL address”
when FLASH BSDL
access is on



```
BSDL address          type       width state unit
C:00000000--0003FFFF AM29N256   long  auto  1.
C:00040000--0007FFFF AM29N256   long  auto  1.
C:00080000--000BFFFF AM29N256   long  auto  1.
C:000C0000--000FFFFF AM29N256   long  auto  1.
C:00100000--0013FFFF AM29N256   long  auto  1.
C:00140000--0017FFFF AM29N256   long  auto  1.
C:00180000--001BFFFF AM29N256   long  auto  1.
C:001C0000--001FFFFF AM29N256   long  auto  1.
C:00200000--0023FFFF AM29N256   long  auto  1.
C:00240000--0027FFFF AM29N256   long  auto  1.
C:00280000--002BFFFF AM29N256   long  auto  1.
C:002C0000--002FFFFF AM29N256   long  auto  1.
C:00300000--0033FFFF AM29N256   long  auto  1.
C:00340000--0037FFFF AM29N256   long  auto  1.
C:00380000--003BFFFF AM29N256   long  auto  1.
C:003C0000--003FFFFF AM29N256   long  auto  1.
C:00400000--0043FFFF AM29N256   long  auto  1.
C:00440000--0047FFFF AM29N256   long  auto  1.
C:00480000--004BFFFF AM29N256   long  auto  1.
C:004C0000--004FFFFF AM29N256   long  auto  1.
C:00500000--0053FFFF AM29N256   long  auto  1.
C:00540000--0057FFFF AM29N256   long  auto  1.
C:00580000--005BFFFF AM29N256   long  auto  1.
C:005C0000--005FFFFF AM29N256   long  auto  1.
```

For the initialization of the boundary scan chain, the SAFE mode is recommended. Other modes are: SAMPLE (the current state of the driving IC is sampled and these values are used during FLASH programming), ZERO and ONE (sets all unused boundary scan register bits to ’0’ or ’1’). With the mode NONE, the unused boundary scan register bits are not initialized and a previous configuration is used.

⚠	Caution: Initializing the unused boundary scan register bits to all zero or one could enable output drivers which leads to unintended behavior or could damage the board.

After switching to boundary scan mode, the FLASH commands for tool based programming are used. The boundary scan FLASH mode is terminated either by FLASH.BSDLaccess OFF or FLASH.RESet.

```
; stop the debugger
SYStem.Down

; configure the boundary scan chain
BSDL.UNLOAD ALL                             ; remove previous configuration
BSDL.FILE ispCLOCK5610Av_isc.bsm            ; load the BSDL file for IC1
BSDL.FILE ispCLOCK5610Av_isc.bsm            ; load the BSDL file for IC2
BSDL.FILE ispCLOCK5610Av_isc.bsm            ; load the BSDL file for IC3
BSDL.FILE LCMXCO1200C_ftBGA256.bsdl         ; load the BSDL file for IC4

; configure the FLASH interface
BSDL.FLASH.IFDefine RESet                   ; remove previous configuration
BSDL.FLASH.IFDefine NOR 4. 24. 16.          ; defines a 16 bit NOR FLASH
                                            ; with 24 address bits on IC4
BSDL.FLASH.IFMap CE   PR7C                  ; map the FLASH CE to port PR7C
BSDL.FLASH.IFMap OE   PR7D                  ; map the FLASH OE to port PR7D
BSDL.FLASH.IFMap WE   PB4C                  ; map the FLASH WE to port PB4C
; map the address ports
BSDL.FLASH.IFMap A0   PB7E
BSDL.FLASH.IFMap A1   PB7F
BSDL.FLASH.IFMap A2   PB8A
BSDL.FLASH.IFMap A3   PB8B
BSDL.FLASH.IFMap A4   PB8C
BSDL.FLASH.IFMap A5   PB8D
BSDL.FLASH.IFMap A6   PB8E
BSDL.FLASH.IFMap A7   PB8F
BSDL.FLASH.IFMap A8   PB9A
BSDL.FLASH.IFMap A9   PB9B
BSDL.FLASH.IFMap A10  PB9C
BSDL.FLASH.IFMap A11  PB9D
BSDL.FLASH.IFMap A12  PB9E
BSDL.FLASH.IFMap A13  PB9F
BSDL.FLASH.IFMap A14  PB10A
BSDL.FLASH.IFMap A15  PB10B
BSDL.FLASH.IFMap A16  PB10C
BSDL.FLASH.IFMap A17  PB10D
BSDL.FLASH.IFMap A18  PB10F
BSDL.FLASH.IFMap A19  PB11A
BSDL.FLASH.IFMap A20  PB11B
BSDL.FLASH.IFMap A21  PB11C
BSDL.FLASH.IFMap A22  PB11D
BSDL.FLASH.IFMap A23  PB6E
```

```
; map the data ports
BSDL.FLASH.IFMap DQ0  PR16B
BSDL.FLASH.IFMap DQ1  PR16A
BSDL.FLASH.IFMap DQ2  PR15B
BSDL.FLASH.IFMap DQ3  PR15A
BSDL.FLASH.IFMap DQ4  PR14D
BSDL.FLASH.IFMap DQ5  PR14C
BSDL.FLASH.IFMap DQ6  PR14B
BSDL.FLASH.IFMap DQ7  PR14A
BSDL.FLASH.IFMap DQ8  PR13D
BSDL.FLASH.IFMap DQ9  PR13C
BSDL.FLASH.IFMap DQ10 PR13B
BSDL.FLASH.IFMap DQ11 PR13A
BSDL.FLASH.IFMap DQ12 PR12D
BSDL.FLASH.IFMap DQ13 PR12C
BSDL.FLASH.IFMap DQ14 PR12B
BSDL.FLASH.IFMap DQ15 PR12A


; check the boundary scan chain
if bsdl.check.bypass()
(
    if bsdl.check.idcode()
    (

        ; initialize boundary scan chain
        BSDL.FLASH.INIT SAFE

        ; reset the FLASH declaration
        FLASH.RESet

        ; switch to boundary scan FLASH mode
        FLASH.BSDLaccess ON

        ; declare the FLASH sectors by CFI query
        FLASH.CFI 0x0 Word

        ; unlock the FLASH device if required
        ; FLASH.UNLOCK ALL

        ; enable the programming for all declared FLASH devices
        FLASH.ReProgram ALL

        ; specify the file that should be programmed
        Data.LOAD.auto demo.x

        ; program the file and disable the FLASH programming
        FLASH.ReProgram off

        ; verify FLASH data
        FLASH.AUTO ALL
        Data.LOAD.auto demo.x /ComPare
        FLASH.AUTO off
```

```
            ; finish the boundary scan FLASH mode
            FLASH.BSDLaccess OFF
        )
    )
```

FAQ

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

Further Information

List of supported FLASH devices	**"List of Supported FLASH Devices"** (flashlist.pdf) or **https://www.lauterbach.com/ylist.html**
Command list (NOR FLASH)	**FLASH** command list in **"General Commands Reference Guide F"** (general_ref_f.pdf).
Troubleshooting	**"Tips to Solve NOR FLASH Programming Problems"** (flash_diagnosis.pdf)
Write your own FLASH programming algorithm	**"How to Write your own FLASH Algorithm"** (flash_app_own_algorithm.pdf)