

Application Note FLASH Programming TriCore

Release 02.2025



TRACE32 Online Help	
TRACE32 Directory	
TRACE32 Index	
TRACE32 Documents	Þ
ICD In-Circuit Debugger	þ
Processor Architecture Manuals	
TriCore	Þ
TriCore Application Notes	
Application Note FLASH Programming TriCore	1
Introduction	4
FLASH Programming Commands	5
Organization of TriCore FLASH Scripts FLASH Declaration Scripts FLASH Feature Scripts FLASH Demo Scripts	6 6 7 7
TC2xx Devices Erased FLASH TRACE32 Methods for Safe FLASH Programming	11 11 13
NOP Sectors Protecting Boot Mode Headers Recommended Procedures	13 14 15
Boot Mode Headers User Configuration Blocks HSM OTR and WOR Sectors	15 16 18 21
TC3xx Devices Erased FLASH TRACE32 Mechanisms for Safe FLASH Programming	22 22 24
NOP Sectors Recommended Procedures User Configuration Blocks HSM	24 25 25 30
OTP and WOP Sectors DFLASH Single-Ended and Complement Sensing Mode Support for SOTA	33 34 35

Version 13-Feb-2025

Introduction

TriCore AURIX devices have many configurations that are located in FLASH (e.g., User Configurations Blocks). For security and safety reasons, the chip offers the possibility, e.g., to install intended locking of the debug access, or prohibit programming of specific FLASH sectors. But programming invalid content to the FLASH may result in locking the device forever.

TRACE32 provides several methods to minimize the risks of accidentally writing wrong content to FLASH and consequently locking the device without recovering possibilities. To fully benefit from these mechanisms, it is mandatory to understand the TRACE32 FLASH programming strategies for TriCore AURIX devices and to learn about recommended procedures and good practices when writing FLASH scripts. This is the intention of this document.

This document is only discussing peculiarities of TRACE32 internal FLASH programming for TriCore AURIX devices. The document "Onchip/NOR FLASH Programming User's Guide" (norflash.pdf) covers TRACE32 internal FLASH programming in general and is recommended as basic knowledge for this application note.

The first section summarizes the TRACE32 FLASH programming commands and which methods have to be used or avoided according to the TriCore specificities. The second section presents different types of TRACE32 FLASH scripts for TriCore, their location in the TRACE32 installation and their intended usage. The last two sections are separately covering FLASH programming for TC2xx and TC3xx. Chip specific behaviors, protection mechanisms, practical use-cases and scripting examples are provided.

In this document, the term AURIX is used to designate all TC2xx and TC3xx devices.

The procedures and scripts presented in this document are designed to avoid common risks and mistakes resulting in a locked device. TRACE32 FLASH scripts consider critical content as valid when it passes number of formal checks. Formal checks are, e.g., correct address alignment, presence of magic patterns matching CRCs, etc. When formal checks have passed, the Startup Software (SSW) will consider the content as valid and will evaluate it.
The TRACE32 FLASH scripts are continuously improved and extended to cover a maximum of protection but there is no guarantee that all cases are covered. For example, the scripts may not warn if a user actively enables locking. In no case using TRACE32 FLASH scripts will replace reading the TriCore User Manuals and Errata Sheets.

TRACE32 offers three FLASH programming methods. Each method uses different groups of FLASH programming commands:

- FLASH.Erase / FLASH.Program
- FLASH.ReProgram
- FLASH.AUTO

Details on how these commands work and information about their parameters and option are explained in "Onchip/NOR FLASH Programming User's Guide" (norflash.pdf).

For TriCore, the commands **FLASH.Erase** and **FLASH.Program** have to be used with care. Using these commands might result in incorrect ECC and erroneous FLASH content. This especially happens when the programmed file format fragmentation does not cover the entire ECC line.

The command **FLASH.ReProgram** is the convenient choice when programming a content that will make big changes to many FLASH sectors. Loading an application ELF file is a typical example.

The command **FLASH.AUTO** is the best fit when making small patches or changing some bytes in the FLASH.

Three categories of FLASH scripts for the TriCore architecture are available in the TRACE32 installation:

- FLASH declaration scripts
- FLASH feature scripts
- FLASH demo scripts

This section presents their organization and intended usage for writing user's FLASH scripts.

FLASH Declaration Scripts

Internal FLASH programming for AURIX devices is using a target-controlled method, where the FLASH programming algorithm (or FLASH driver) is executed from the device RAM under control of TRACE32 PowerView. Every AURIX device has a different FLASH structure and consequently a different FLASH mapping. Thus it is necessary to specify the FLASH mapping before the FLASH programming commands can be used.

The FLASH mapping declaration is done via the TRACE32 FLASH declaration scripts located in the installation directory under "~~/demo/tricore/flash". Each device series has a FLASH declaration script. For example, the script tc39x.cmm is intended to be used for all TC39x devices. Obviously, the script tc27x.cmm is for all TC27x devices.

The FLASH declaration scripts are intended as "library" scripts and have to be used as-is. Those scripts must not be modified or partly copied to the user's FLASH scripts. Instead, they have to be called from the user's FLASH scripts with the appropriate arguments.

Example:

DO ~~/demo/tricore/flash/tc39x.cmm CPU=TC397XE PREPAREONLY DUALPORT=1

The *"PREPAREONLY"* argument instructs the FLASH declaration script to declare the FLASH and exit without FLASH programming.

The argument "*CPU=<cpu>*" selects the used CPU derivative. This argument is optional if the debugger to target communication is already established.

The argument "**DUALPORT=1**" enables data transfer via dual-port memory access and ensures a continuous running of the FLASH algorithm until the FLASH programming is finished. This results in a higher FLASH programming performance.

Other important information and details about all the script arguments can be found in the script header.

Some devices, using the same CPU selection, might be available with different FLASH sizes and consequently require different FLASH declarations. This is handled transparently if the device is already supported by the used FLASH declaration script version. A TRACE32 software update can be requested

from *support@lauterbach.com* if the device is not supported by the used TRACE32 software version (i.e. the CPU selection is missing) and/or by the FLASH declaration script (an error "*Unsupported Pflash Size*" is thrown by the script).

NOTE: Do not move the FLASH declaration scripts from their default location. This leads to problems when performing a TRACE32 software update, since the copied scripts won't be updated to the newest version. This prevents to benefit from eventual script improvements and support of new devices.

FLASH Feature Scripts

Like FLASH declaration scripts, FLASH feature scripts are located in the installation directory under "~~/demo/tricore/flash". They have to be used as-is by calling them from the user's FLASH scripts using appropriate arguments.

The feature scripts are used to help programming various features including checks for valid content.

Examples of FLASH feature scripts for TC2xx:

- tc2xx-ucb.cmm
- tc2xx-bmhd.cmm
- tc2xx-hsm-config.cmm
- tc23x-hsm.cmm, tc27x-hsm.cmm, tc29x-hsm.cmm

Examples of FLASH feature scripts for TC3xx:

- tc3xx-ucb.cmm
- tc3xx-hsm-config.cmm
- tc3xx-swap.cmm

Details about the scripts' arguments and intended usage are documented in the script headers. Use-cases and examples will be detailed later in this document.

FLASH Demo Scripts

The TRACE32 demo directory for the TriCore architecture includes FLASH demo scripts for Infineon reference boards. These demo scripts are located in the installation under "~~/demo/tricore/hardware" and are organized by board name and chip derivative.

Example:

PEDIT ~~/demo/tricore/hardware/triboard-tc3x7/tc387qp/demo_flash.cmm

These scripts are intended to be copied and used as templates to create customised users' scripts.

TRACE32 script search dialog can be used to search an appropriate FLASH demo script. The dialog can be opened from the menu "*Help -> Demo Scripts...*". Using the chip name and appropriate keywords to refine the search, the matching scripts from the TRACE32 installation are listed.

P Search for scripts		
Search Selection Manuals		
Example search: TC26* multi-core		
tc387qp flash demo v 💥 🛉 Search 6 demo files	found.	
Filter		
O Rone ○ Chip ○ Board		
Search for newest scripts at https://www.lauterbach.com/scripts.html		
2		
CONFIG Eg: Tree view III LIST CONFIG		
Title	Chip	Board ^
Demo script for TC387QP on TriBoard-TC3x7 (SMP, Flash, multisieve app)	TC 387QP	TriBoard-TC3x7 🔥
Demo script for TC387QP on TriBoard-TC3x7 (Flash, sieve app)	TC 387QP	TriBoard-TC3x7
Tricore TC387 Serial SPI <mark>flash</mark> program	TC 38*	-
Generic FLASH script file for TC38x devices	TC38*	-
Flash script file for TC3xx UCB sectors.	TC3*	-
Flash script file for TC3xx SOTA (Software Over The Air).	TC3*	- ~
		н. <

In this example, the matching FLASH demo script is entitled "*Demo script for TC387QP on TriBoard-TC3x7 (Flash, sieve app)*". By selecting the script, a more detailed description is displayed.

P Search for scripts						
Search Selection	Manuals					
File: demo_flash.cmm	Title: Demo script for TC387QP on TriBoard-TC3x7 (Fla	ash, sieve app)	Niou/Start			
Chip: TC387QP Board: TriBoard-TC3x7	Programs the sieve demo application (si internal flash and sets up a demo debuc used as a template for flashing an app	ingle-core) into the processor g scenario. This script can be lication.	Edit			
Related Documentation: readme.	Related Documentation: readme.bxt					
			New Search			
CONFIG	LISTCONFIG					
Title		Chip Boạr	d ^ _			
Demo script for TC387QP on Demo script for TC387QP on	TriBoard-TC3x7 (SMP, Flash, multisieve app Tri <mark>Board-TC3x7 (</mark> Flash <mark>, sieve app)</mark>) TC387QP Trie TC387QP Trie	Board-TC3x7			
Tricore TC387 Serial SPI f Generic FLASH script file f Flash script file for TC3xx	IST program or TC38x devices UCB sectors.	TC38* - TC38* - TC3* -				
<pre>chiputile for IC3XX </pre>	SUTA (SUTEware Over The ATr).	IC3" -	× 			

The FLASH demo scripts have a unified layout with three main parts.

The first part of the script initiates the debugger/target communication. This includes setting the CPU selection and setting up all the required configurations to start the debugger/target communication successfully.

TRACE32 PowerView for TriCore							
File Edit View Var Break Run CPU Misc Trace Perf Cov TC38x Window Help							
M M ⊷ ↓ 4′ Ċ ▶ II ? 1⁄2 ◎ 圖 @ ֎ ֎ ֎ ● ≟ ≫							
[B::CD.PSTEP "C:/T32/demo/tricore/hardware/triboard-tc3x7/tc387qp/demo_flash.cmm"]							
🖅 Step 🚿 Over 🖄 Up 😰 Continue 🕮 Stop 🗊 Enddo 🥰 Skip 🕰 Macros 📓 Edit 😰 Breakpoints							
<pre>Step # Over 20 Up 20 Continue 20 Stop 20 Skip Control 20 Skip 20 Macros 20 Skip 20 Skip 20 Macros 20 Skip 20 Macros</pre>							
30 Statem. op 31 LOCAL &elfFile &progFlash &bmhdResult 32 33 B::							
components trace Data Var List PERE SYStem other previous							
P:A0000020 system ready X MIX UP							

Before starting the FLASH programming, it is very important to disable all external (and internal) watchdogs to avoid resetting the target while the FLASH programming is in progress. It is also very important to start FLASH programming from a "clean" known state of the target. In this example, the command **SYStem.Up** is used to reset the target and halt the CPU at the reset vector.

After a successful startup of the debugger/target communication, the state "*system ready*" is displayed in the status bar.

The second part is the FLASH preparation. This is done by calling the FLASH declaration script with the *"PREPAREONLY"* argument.

```
DO ~~/demo/tricore/flash/tc38x.cmm CPU=TC387QP PREPAREONLY
```



After the FLASH preparation the command FLASH.List can be used to examine the declared FLASH mapping.

B::flash.li	st										×
🌲 State	📣 Reset	0 01	ff	ntel 🍜	📣 Pr	ogram	📣 RePr	ogram	📣 Auto	🍀 CF	1
C:AF07E	ad 3000AF0	dress 7BFFF	type TARG	ET I	width Long	state	unit 20.	extra	PROM		-
C:AF070 C:AF070	000AF0	7CFFF 7DFFF	TARG TARG	ET I	Long		20.	// EEF // EEF	PROM		
C:AF07E C:AF07F	000AF0	7EFFF 7FFFF	TARG TARG	ET I	Long Long		20. 20.	// EEF // EEF	PROM		
C:AF400 C:AF400	0000AF4	001FF 003FF	NOP NOP		Long Long		21.	// UCE	300 BMHD0 301 BMHD1	ORIG	
C:AF400 C:AF400	0400AF40	005FF	NOP		Long		21.	// UCE	302 BMHD2 303 BMHD3	ORIG	
C:AF400	000AF40	009FF	NOP		Long		21.	// UCE	304 55W 305 USER		
C:AF400	E00AF4	00FFF	NOP		Long		21.	// UCE	307 HSMCF(308 BMHD0	COPY	
C:AF401 C:AF401	200AF4	013FF 015FF	NOP		Long		21.	// UCE	309 BMHD1 310 BMHD2	COPY	
C:AF401 C:AF401	600AF4	017FF 019FF	NOP	i	Long		21. 21.	// UCE	311 BMHD3 312 REDSEG	COPY	~
			<							>	

The third part of a FLASH demo script is the FLASH programming. The command **FLASH.ReProgram ALL** enables the FLASH programming for all "**TARGET**" sectors, then the ELF file of the demo application (or any other binary format) is loaded using the appropriate **Data.LOAD** command.

缰 [B::C	IB::CD.PSTEP "C:/T32/demo/tricore/hardware/triboard-tc3x7/tc387qp/demo_flash.cmm"]									
Ste	🛞 Over 🖉 Up 😥 Continue 🏽 Stop 🗊 Enddo 🥰 Skip 🕰 Macros 📓 Edit 🕸 Breakpo	oints								
42	(F FOUND()	^								
	; ==== Step 1: Program TriCore code ====									
46	DIALOG.YESNO "Program Lauterbach demo into flash memory?"									
47	IF (&progFlash)									
49	(; enable flash programming									
51	FLASH.ReProgram ALL									
54	; load demo application									
54										
56)	~								
_	< >	e a								

After executing FLASH.ReProgram OFF the effective FLASH programming to the device is started.

The rest of this demo script shows the FLASH programming of the BMHDs. This part will be detailed in the following sections.

Erased FLASH

For TriCore AURIX devices the data is stored in FLASH with error correcting codes "ECC" in order to protect against data corruption. These ECC can correct a certain amount of bit errors.

For PFLASH, the ECC are calculated over the data and the address bits. After an erase, the state of the data as well as the state of the ECC is "all-0". As the ECC of 0x0 is not 0x0, the stored ECC does not match with the ECC of the actual data. Thus, an erased PFLASH range is resulting in ECC errors.

The ECC is automatically evaluated by the chip when reading data. Thus, reading from an erased PFLASH sector returns "bus error" indicated as "???????" in, e.g., a **Data.dump** window.

101 [B::Data.dump (C:0xA03C2100 /	/Long /NoAsc	ii /SpotLight]	l					×
address	0	4	8	C	0	4	8	C	
D:A03C2100	→??????????	?????????	?????????	????????	????????	????????	?????????	?????????	~
D:A03C2120	????????	?????????	?????????	?????????	????????	????????	?????????	?????????	=
D:A03C2140	????????	?????????	?????????	?????????	?????????	????????	?????????	?????????	-
D:A03C2160	?????????	?????????	?????????	?????????	?????????	?????????	?????????	?????????	~
D:A03C2180	?????????	?????????	?????????	?????????	?????????	????????	?????????	?????????	\wedge
D:A03C21A0	?????????	?????????	?????????	?????????	?????????	????????	?????????	?????????	
D:A03C21C0	?????????	?????????	?????????	?????????	?????????	????????	?????????	?????????	
D:A03C21E0	?????????	?????????	?????????	?????????	?????????	????????	?????????	?????????	\sim
	<							>	

For DFLASH, a different ECC algorithm is used. ECC are calculated over the data bits only and the "all-0" state for data and checksums do not result in "bus errors" when reading from erased DFLASH sectors.

Bus errors that are caused by uncorrectable ECC errors can be disabled by configuring *FLASH0_MARP.TRAPDIS* and *FLASH0_MARD.TRAPDIS* register fields.

After changing the configuration of *FLASHO_MARP.TRAPDIS* field to disable the uncorrectable ECC error reporting, the erased range is displayed like in the following screenshot.

🛷 B::PER , "PMU (Program Memory Unit),FLASH0,FLASH0_MARP" /SpotLight								
FLASHO_MARI	P 00008	3000 TRAPDIS SELP1 0000 TRAPDIS SELD0	5 <mark>Yes</mark> Standard 5 No Standard	SELPO SPNDE HMARG) Standard RR No error IN Tight0	SPND	< >	
<						>		
B::Data.dump C:0 address D:A03C2100 D:A03C2120 D:A03C2140	xA03C2100 /Long / 0 00000000 0000 00000000 0000 00000000	NoAscii /SpotLigh 4 00000 0000000 00000 0000000 00000 000000	t 8 C 00 0000000 00 0000000 00 00000000 00 00000000	0 00000000 00 00200000 00 00000000 00	4 0000000 0000000 000000 00000000000000	C 3 C 0 00000000 0 00000000 0 00000000 0 00000000	*	
D:A03C2160 D:A03C2180 D:A03C21A0 D:A03C21C0 D:A03C21E0	00000000 0000 00000000 0010 00000000 0000 00000000	00000 0000000 00000 0000000 00000 000000	00 0000000 00 0000000 00 0000000 00 000000	00000000 00 00000000 00 00000000 00 000000	0000000 000000000000000000000000000000	0 0000000 0 0000000 0 0000000 0 0000000 0 000000	*	

Some bits still show the value 1. The displayed values don't reflect the real physical FLASH content, but this reflects the content obtained after applying the ECC correction. The result is not the same for all the PFLASH lines because for PFLASH, the address of the ECC line is used for calculating the ECC.

The real PFLASH content can be displayed by additionally disabling the ECC correction in the corresponding PFLASH register filed *FLASH0_ECCRPp.ECCORDIS* like the example in the following screenshot.

🤝 B	::PER , "PMU (I	Program Mem	ory Unit),FLAS	H0,FLASH0_N	/IARP" /SpotLi	ght				×
	FLASH0_MA	RP	0008000	TRAPDIS	Yes Standar		P0 51	andard		^
	FLASH0_MA	RD	00000000	TRAPDIS	No	SPI	NDERR NO	o error	SPND	
<				SELDO	SLandari	u HM/	AKGIN	Ignico	2	≻ .a
										_
🥌 B	::PER , "PMU (I	Program Mem	ory Unit),FLAS	H0, PF1 Regist	ters, FLASH0_E	CCRP1" /Spot	Light			×
	FLASH0_ FLASH0	ECCRP1 CBABCEG1	8000000	ECCORD	IS <mark>Yes</mark> No		EDCERRI CLR	INJ No er No ef	ror fect	^
	FLASH0_	CBABSTAT1	0000000	VLD9	Low		VLD8	Low		~
 <									2	.
010 101 B	::Data.dump C	:0xA03C2100 /	Long /NoAsci	i /SpotLight				[×
	address	0	4	8	C	0	4	8	C	
D	: A03C2100	♦00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	^
D	:A03C2120	00000000	000000000	000000000	000000000	00000000	000000000	00000000	000000000	=
D	:A03C2160	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	~
D	:A03C2180	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	^
	: 403C21A0	000000000	00000000	000000000	00000000	00000000	00000000	000000000000000000000000000000000000000	00000000	
D	:A03C21E0	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	~
		<							2)

NOTE:

Configuring *FLASH0_MARP* and *FLASH0_MARD* to mask the bus error traps or disabling the ECC correction in *FLASH0_ECCRPp* are not presented here as programming options. These are only mentioned for better understanding of the chip behavior.

To ensure that all PFLASH and ECCs are initialized, the command **FLASH.ReProgram** can be used with the option *"/FILL"* to fill all erase sectors with 0.

Example:

FLASH.ReProgram ALL /Erase /FILL Data.LOAD.Elf application.elf FLASH.ReProgram OFF

NOP Sectors

TRACE32 FLASH declaration scripts declare sensitive sectors as "**NOP**" to protect them against unwanted FLASH programming. TRACE32 PowerView silently discards all erase and write operations to "**NOP**" sectors.

When it is required to make changes to "**NOP**" sectors, the command **FLASH.CHANGEtype** has to be used to modify the FLASH sector type to "**TARGET**". Only then, FLASH operations for that range are performed by TRACE32 PowerView.



To prevent unintended bricking of the device the FLASH declaration scripts for TC2xx declare the tuning protection configurations and the HSM code as "*NOP*" sectors. The following is an extract from tc27x.cmm FLASH declaration script.

FLASH.Create 1. 0xA0014000++0x3FFF 0x4000 NOP Long /INFO "Tuning Protec."
FLASH.Create 1. 0xA0018000++0x3FFF 0x4000 NOP Long /INFO "HSM code sect."

These sectors can safely be used for the application if the user makes sure that the code he is programming will not lock the device.

The FLASH programming scripts for TC2xx does not provide checks to ensure that the code programmed to the tuning protection sectors or HSM code sectors is safe. For the tuning protection sectors, the user has to ensure this by himself. For the HSM code sectors it is strongly recommended to use the checks provided by the HSM feature scripts e.g. tc29x-hsm.cmm. Details and scripting examples are provided in the recommended procedures paragraph for HSM.

NOTE:	It is strongly recommended to use the command FLASH.CHANGEtype to
	change a sector from "NOP" to "TARGET". After programming, change the
	sector back to "NOP" to avoid unintended programming with invalid content.

For TC2xx devices, the option *"/BootModeHeaDer"* is used with the command FLASH.Create to make TRACE32 PowerView aware of the boot mode header regions. TRACE32 will then ensure that the old content of the boot mode headers will be preserved if nothing is programmed to them after an erase.

Example:

&s0_s2="0xA0000000--0xA000BFFF" ; PS0, S0..S2 &bmhd0="0xA0000000--0xA000001F" ; Range of BMHD0 FLASH.Create 1. &s0_s2 0x4000 TARGET Long **/BootModeHeaDer** &bmhd0

The resulting FLASH declaration can be viewed using the command **FLASH.List**. The following FLASH declaration is obtained after executing the FLASH declaration script for TC277TF device using the parameter "*PREPAREONLY*".

DO ~~/demo/tricore/flash/tc27x.cmm CPU=TC277TF **PREPAREONLY**

🐡 B::FLASH.List					3
🌲 State 🛛 🚜 Reset 🛛 🔾	Off 🛛 券 Cancel	📣 Program	📌 ReProgram	📣 Auto 🛛 👹 CFI	
address t C:800000080FFFFF A C:A000000A0003FFF T C:A0004000A0007FFF T C:A0008000A0008FFF T C:A000000A0013FFF T C:A0010000A0013FFF N C:A0018000A013FFF N C:A0016000A001FFF T C:A0020000A0027FFF T	ype wid LLIAS ARGET Lon ARGET Lon ARGET Lon ARGET Lon ARGET Lon IOP Lon IOP Lon ARGET Lon ARGET Lon	th state u g 1 g 1 g 1 g 1 g 1 g 1 g 1 g 1 g 1 g 1 g 1 g 1 g 1 g 1 g 1 g 1 g 1 g 1	nit extra C:A000000 BMHD=C:A00 BMHD=C:A00 BMHD=C:A00 C C C C C C C C C C C C C C C C C C	0A0FFFFF 000000A000001F 00FFE0A000FFFF Protection de sector 01FFE0A001FFFF 020000A002001F	
C:A0028000A002FFFF	arget Lon <	g I	•	>	

In order to prevent unwanted locking of a device the following procedures must be followed. They combine the FLASH declaration script and the FLASH feature scripts.

```
NOTE: The procedures and script examples presented in this document requires TRACE32 release 2020/02 or newer.
```

Boot Mode Headers

When booting an AURIX device, at least one boot mode header should contain valid data. For some early TC2xx devices, having no valid boot mode header is fatal. This will lock the device with no recovering possibilities.

The FLASH declaration scripts can be used to check if at least one valid Boot Mode Header is going to be programmed to the device. The script needs to be called using the argument "*CHECKBMHD*" as follows:

DO ~~/demo/tricore/flash/tc27x.cmm CHECKBMHD

This check has to be performed before the effective FLASH programming is started via the command **FLASH.ReProgram OFF**.

If at least one valid boot mode header is found, the script execution returns the result "**BMHD_OK**". Then the user script can continue the FLASH programming. If no valid boot mode header is found, the check will return "**BMHD_MISSING**" and the user script must abort the FLASH programming.

Here is the script example with the full programming flow:

```
; Enable FLASH programming
FLASH.ReProgram ALL
; Load the demo application
Data.LOAD.Elf application.elf
; Check if there is at least one valid BMHD
DO ~~/demo/tricore/flash/tc27x.cmm CHECKBMHD
ENTRY & bmhdResult
IF ("&bmhdResult"=="BMHD OK")
(
  ; At least one valid BMHD is detected => make the changes to the device
  FLASH.ReProgram OFF
)
ELSE
(
 DIALOG.OK "No valid Boot Mode Header found!" "Reverting loaded data"
  FLASH.ReProgram CANCEL
  ENDDO
)
```

The FLASH feature script tc2xx-ucb.cmm is used to help programming UCBs and test for valid content.

The following example shows how the PFLASH UCB can be programmed when it is in a confirmed state (indicated by the register field *FPRO.PROINP*). The main steps to achieve this are the following:

- 1. Perform the FLASH declaration using the appropriate FLASH declaration script.
- 2. Activate UCB_PFLASH programming by calling the UCB programming script using the argument "ACTIVATE". This combines the following steps:
 - Unlocking the UCB_PFLASH password protection. This is only required if UCB_PFLASH password protection is active.
 - Changing the UCB_PFLASH sector type from "**NOP**" to "**TARGET**" to allow FLASH programming by TRACE32 PowerView.
 - Enabling FLASH programming for UCB_PFLASH using FLASH.AUTO command. The latter is used to preserve the unchanged content of the affected FLASH sector.

Unlocking the UCB password protection requires the unlock password to be provided. This is achieved by additionally providing the argument "*PWD=<password>*" like in the following example:

&pwd="0x0_0x1_0x2_0x3_0x4_0x5_0x6_0x7"
D0 ~~/demo/tricore/flash/tc2xx-ucb.cmm UCB=PFLASH ACTIVATE PWD=&pwd

Alternatively, a dialog can be used to enter the password manually. This is done by using the argument "*DIALOG*" instead of "*PWD=<password>*".

👠 Dialog	
Enter password for	r PFLASH
0x00000000	^
0x00000001	
0x0000002	
0x0000003	
0x00000004	
0x00000005	
0x0000006	
0x0000007	¥
ОК С	ancel

- 3. Make the desired changes to the UCB via, e.g., **Data.Set** commands for both original and copy UCB contents.
- 4. Program the modifications by calling the UCB programming script using the argument "**PROGRAM**". This will perform the following:
 - Do a formal verification of the UCB_PFLASH content.
 - If the provided content is formally correct the changes are programmed to the device and the script returns "UCBOK".
 - If the formal checks detected invalid content the FLASH programming is aborted and the script returns the result "*UCBFAIL*".

In both cases the NOP protection of UCB_PFLASH sector is restored.

Here is the script example with the full programming flow:

```
: Perform the FLASH declaration
DO ~~/demo/tricore/flash/tc27x.cmm PREPAREONLY CPU=TC277TF
; Activate UCB modification:
; - Unlock password protection to allow programming of UCB_PFLASH
; - Allow FLASH programming by changing the sector type from NOP to
   TARGET
;
; - Enable FLASH programming for UCB_PFLASH using FLASH.AUTO command
DO ~~/demo/tricore/flash/tc2xx-ucb.cmm UCB=PFLASH ACTIVATE DIALOG
; Disable all write protections in PROCONPO original and copy content
Data.Set 0xAF100000 %Long 0x0
Data.Set 0xAF100010 %Long 0x0
: Program the modifications:
; - Formal verification of the UCB PFLASH content
; - The content is formally correct => Program the changes to the device
; - The check detected invalid content => Abort FLASH programming
; - In any case restore NOP protection for UCB PFLASH range
DO ~~/demo/tricore/flash/tc2xx-ucb.cmm UCB=PFLASH PROGRAM
ENTRY &result
IF ("&result"=="UCBOK")
(
 PRINT "UCB PFLASH programming successful"
)
```

Enabling HSM boot when no valid HSM code is present will lock the device permanently. Series-specific HSM FLASH scripts can be used to program with maximum safety.

Example 1:

This example shows how the following HSM feature scripts can be used to check for valid HSM code and enable the HSM boot.

- tc23x-hsm.cmm
- tc27x-hsm.cmm
- tc29x-hsm.cmm

Assuming that the TriCore application is already programmed to the device, the recommended FLASH programming flow for HSM programming is the following:

1. Disable NOP protection of HSM code and configuration sectors:

Per default, the FLASH declaration scripts declare HSM code sectors as "**NOP**". It is necessary to change these sectors types to "**TARGET**" to allow their programming by TRACE32 PowerView. HSM FLASH scripts can be used to achieve this using the script argument "**PREPAREONLY**".

DO ~~/demo/tricore/flash/tc27x-hsm.cmm CPU=TC277TF **PREPAREONLY**

2. Program the HSM application code:

The FLASH programming of HSM application can be done via, e.g., HSM specific ELF file or any other binary format. The FLASH programming commands **FLASH.ReProgram** have to be used.

```
FLASH.ReProgram ALL
Data.LOAD.Elf hsm_app.elf /NoClear /NoRegister /NosYmbol
FLASH.ReProgram OFF
```

3. Check for valid HSM code and enable HSM boot:

After programming the HSM code, a post programming check could be done to verify if the application is correctly programmed to the target FLASH using the command **Data.LOAD** with the option *"/DIFF"*.

If any difference is found, HSM boot needs to be disabled by calling the HSM FLASH script with the argument "*DISABLE*".

If the HSM application file is correctly programmed the HSM FLASH script can safely be called with the argument *"ENABLE"*. This will verify if the HSM programmed boot vectors are valid and enable HSM by programming the register *UCB_HSMCOTP*. Otherwise, the HSM is kept disabled.

```
Data.LOAD.Elf hsm_app.elf /NoClear /NoRegister /NosYmbol /DIFF
IF FOUND()
(
 ; Ensure that HSM is kept disabled to avoid locking the device!
  DO ~~/demo/tricore/flash/tc27x-hsm.cmm DISABLE
  ENDDO
)
ELSE
(
 ; Check HSM code area and enable HSM
  DO ~~/demo/tricore/flash/tc27x-hsm.cmm ENABLE
)
```

4. Restore the NOP protection of HSM code and configuration sectors: After HSM FLASH programming is finished, restore the NOP protection of HSM sectors by calling the HSM FLASH script using the argument *"FINISH"*.

DO ~~/demo/tricore/flash/tc27x-hsm.cmm FINISH

Here is the script example with the full programming flow:

```
DO ~~/demo/tricore/flash/tc27x-hsm.cmm CPU=TC277TF PREPAREONLY
FLASH.ReProgram ALL
Data.LOAD.Elf hsm_app.elf /NoClear /NoRegister /NosYmbol
FLASH.ReProgram OFF
; Check the HSM Elf file is correctly programmed to the device
Data.LOAD.Elf hsm app.elf /NoClear /NoRegister /NosYmbol /DIFF
IF FOUND()
  ; Ensure that HSM is kept disabled to avoid locking the device!
 DO ~~/demo/tricore/flash/tc27x-hsm.cmm DISABLE
 ENDDO
)
ELSE
(
  : Check HSM code area and enable HSM
 DO ~~/demo/tricore/flash/tc27x-hsm.cmm ENABLE
)
DO ~~/demo/tricore/flash/tc27x-hsm.cmm FINISH
```

Example 2:

This example shows how the HSM feature script *tc2xx-hsm-config.cmm* can be used for an easier configuration of the HSM related chip setting.

The script can be started in dialog mode as follows.

DO ~~/demo/tricore/flash/tc2xx-hsm-config.cmm

The HSM related setups are configured via various check-boxes. The "**Refresh**" button reloads the currently programmed configuration. After selecting the desired HSM configurations the button "**Program**" stores the new configuration to the device. If the selected setup enables the HSM boot, the script will first check if the HSM boot vectors are valid, otherwise the FLASH programming is aborted to prevent permanent locking of the device.

▲ TC277TF HSM settings	
HSM Debug Configuration	
HSM Debug Disable	Test Interface Locked
Debug Interface Locked	HSM Test Disable
HSM Configuration	
HSM Boot Enable	Enable HSM Triggering Resets
SSW Wait	Destructive Debug Entry
HSM can force pins	Block Flash Analysis on all ranges
HSM Sector Exclusive Protection	HSM Sector OTP Protection
Code Sector 06	Code Sector 06
Code Sector 16	Code Sector 16
Code Sector 17	Code Sector 17
🗹 Data Sector	
Program Refr	resh Show

The button "*Show*" generates a script snippet with all the settings to be made, that can be copied to the user's FLASH script.

For TriCore AURIX devices, beside write protection with password, two types of sector specific programming protection can be distinguished.

- FLASH sectors that are configured with OTP (One-Time Programmable) protection can not be erased or programmed. Otherwise, FLASH programming errors will be reported by the hardware.
- FLASH sectors that are configured with WOP (Write-page Once Protection) can be programmed once. These sectors can only be programmed if they are in erased state. Erasing these sectors is prevented by hardware after the protection is activated. The hardware will report an error when trying to erase them.

TRACE32 defines the FLASH sectors that must not be erased or programmed as "*NOP*" sectors. Thus OTP protected sectors (as defined by Infineon) are declared by TRACE32 FLASH declaration scripts as NOP sectors.

TRACE32 uses the "*OTP*" term to designate FLASH sectors that can be programmed once because erase is prohibited. Thus, the FLASH sectors that are protected with WOP (as defined by Infineon) are declared by TRACE32 FLASH declaration scripts using the option "*/OTP*".

Sectors declared with *"/OTP*" can only be programmed with the command FLASH.Program and the option *"/OTP*" must be specified.

Example:

```
FLASH.Program 0xA000000++0x3FFF /OTP
Data.Set ...
FLASH.Program OFF
```

More details about FLASH programming of TRACE32 OTP sectors can be found in "Onchip/NOR FLASH Programming User's Guide" (norflash.pdf).

Erased FLASH

For TriCore AURIX devices the data is stored in FLASH with error correcting codes "ECC" in order to protect against data corruption. These ECC can correct a certain amount of bit errors.

For PFLASH, the ECCs are calculated over the data and the address bits. After an erase, the state of the data as well as the state of the ECC is "all-0". As the ECC of 0x0 is not 0x0, the stored ECC does not match with the ECC of the actual data. Thus, an erased PFLASH range is resulting in ECC errors.

The ECC is automatically evaluated by the chip when reading data. Thus, reading from an erased PFLASH sector returns "bus error" indicated as "???????" in, e.g., a **Data.dump** window.

810 B::Data.dump C	:0xA0100200 /I	.ong /NoAscii	i/SpotLight						×
address	0	4	8	C	0	4	8	C	
D:A0100200	??????????	????????	????????	?????????	?????????	?????????	????????	?????????	~
D:A0100220	????????	?????????	?????????	?????????	?????????	?????????	?????????	?????????	=
D:A0100240	?????????	?????????	?????????	?????????	?????????	?????????	?????????	?????????	-
D:A0100260	?????????	?????????	?????????	?????????	?????????	?????????	?????????	?????????	~
D:A0100280	????????	?????????	?????????	?????????	?????????	?????????	?????????	?????????	÷
	<							>	

For DFLASH, a different ECC algorithm is used. ECC are calculated over the data bits only and the "all-0" state for data and checksums do not result in "bus errors" when reading from erased DFLASH sectors.

Bus errors that are caused by uncorrectable ECC errors can be masked by changing the register field *FLASHCON1.MASKUECC* of the CPU associated to the corresponding PFLASH unit.

After changing the configuration in the CPU0 register field *FLASHCON1.MASKUECC* for disabling the ECC error reporting, the erased range is displayed like in the following screenshot.

🛷 B::per , "CPU (C	(PU Subsystem)	,CPU0,Flash C	onfiguration	Registers" /Sp	ootLight						- • •
Flash Confi FLASHCON0 2	iguration A 2 3222120	Registers TAG4	35	Т	AG3	34	TAG2	33	Т	AG1	32 ^
FLASHCON1 FLASHCON2	2010000 AA020A0A	MASKUECC ZBABCLR ECCSCLR RECDIS	No en No en No en No	ffect M ffect M	TALL BABCLR SEL	Error No effect Standard	DBABCI HMARGI	.R No ef IN Low	fect s	BABCLR CCCORDIS	No effect No
<											>
왜 B::Data.dump C	:0xA0100200 /l	.ong /NoAscii	/SpotLight								
胧 B::Data.dump C address	C:0xA0100200 /I	.ong /NoAscii 4	/SpotLight 8	C	0	4	8	C			
端 B::Data.dump C address D:A0100200	C:0xA0100200 /L 0 ♦00000000	ong /NoAscii 4 00000000	/SpotLight <u>8</u> 00000000	C 00001000	00000000	4 00000000	<u>8</u> 00000000	C 00000000			
Image: Bit Data.dump C addr ess D:A0100200 D:A0100220 D:A0100220	:0xA0100200 /L 0 ◆00000000 00000000	ong /NoAscii 4 00000000 00000000	/SpotLight 8 00000000 20000000	C 00001000 00000000	0 00000000 00000000	4 00000000 00000000	8 00000000 00000000	C 00000000 00000000			
B::Data.dump C address D:A0100200 D:A0100220 D:A0100220 D:A0100240 D:A0100240	:0xA0100200 /I	ong /NoAscii 4 00000000 00000000 80000000	/SpotLight 8 00000000 2000000 00000000 00000000	C 00001000 00000000 00000000	0 00000000 0000000 0000000	4 00000000 00000000 10000000	8 00000000 00000000 00000000	C 00000000 00000000 00000000 00000000			
B::Data.dump C address D:A0100200 D:A0100220 D:A0100240 D:A0100260 D:A0100280	C:0xA0100200 /I 0 00000000 00000000 00000000 00000000	ong /NoAscii 4 00000000 00000000 80000000 00000000 000000	/SpotLight 8 00000000 20000000 00000000 00000000	C 00001000 00000000 00000000 00000000 000000	0 00000000 00000000 0000000 00000000 0000	4 00000000 00000000 10000000 00000000 000000	8 00000000 00000000 00000000 00000000 0000	C 00000000 00000000 00000000 00000000 0000			

Some bits still show the value 1. The displayed values don't reflect the real physical FLASH content, but this reflects the content obtained after applying the ECC correction. The result is not the same for all the FLASH cells because the ECC is calculated from the data values and the corresponding address.

The real erased FLASH content can be displayed by additionally disabling the ECC correction in *FLASHCON2.ECCCORDIS* like in the following screenshot.

4	🗢 B::per , "CPU (CPU	J Subsystem)	,CPU0,Flash C	onfiguration	Registers" /Sp	ootLight					[×
6	∃ Flash Config FLASHCONO 23	uration R	tac4	35	т	AG3	34	TAG2	33		TAG1	32	^
	FLASHCON1 02	010000	MASKUECO	No ei	rror S	TALL	Error	00406	5.5 D. No. 1		CRARCER	No 66 +	
	FLASHCONZ AA	020A06	ECCSCLR	No e	ffect M	SEL	Standard	HMARG	LK NOE EN Low	errect	ECCCORDIS	NO ETTECT Yes	
			RECDIS	No									~
Ŀ	<											>	
Ŷ	器 B::Data dumn C:0	×Δ0100200 /I	ona /NoAscii	/SpotLight							ſ		x
Ē	address	0	4	7 Spoteigne 8	C	0	4	8	C		l		
ŀ	D:A0100200 +	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000				~
	D:A0100220	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000				≡
	D:A0100240	000000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000				¥
	D:A0100280	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000				ĉ

NOTE:	Configuring FLASHCON1 to mask the bus error traps or disabling the ECC
	correction in FLASHCON2 are not presented here as programming options.
	These are only mentioned for better understanding of the chip behavior.

To ensure that all PFLASH and ECCs are initialized, the command **FLASH.ReProgram** can be used with the option *"/FILL"* to fill all erase sectors with 0.

Example:

FLASH.ReProgram ALL /Erase /FILL Data.LOAD.Elf application.elf FLASH.ReProgram OFF

NOP Sectors

TRACE32 FLASH declaration scripts declare sensitive sectors as "*NOP*" to protect them against unwanted FLASH programming. TRACE32 PowerView silently discards all erase and write operations to "*NOP*" sectors.

When it is required to make changes to "**NOP**" sectors, the command **FLASH.CHANGEtype** has to be used to modify the FLASH sector type to "**TARGET**". Only then, FLASH operations for that range are performed by TRACE32 PowerView.



FLASH declaration scripts for TC3xx declare the User Configuration Blocks (UCBs) and Configuration Sector Layout (CFS) as "*NOP*" sectors.

Example:

FLASH.Create 21. 0xAF400000++0x1FF 0x0200 NOP Long /INFO "BMHD0 ORIG" FLASH.Create 22. 0xAF800000++0xFFFF 0x0200 NOP Long /INFO "CFS" More safety in FLASH programming can be achieved by using the FLASH declaration scripts and the FLASH feature scripts in the recommended way.

For TC3xx devices, more automation of FLASH pre-checks was introduced starting from TRACE32 release **2018/02**. The FLASH declaration scripts install automatic checkers for the FLASH programming commands to warn the user about dangerous operations and to block fatal operations.

The pre-check has configuration options to control the user confirmation. These options are documented in the header of the FLASH declaration scripts.

To benefit from these automatic checks, the user's FLASH script must call the device-specific FLASH declaration script using the parameter "*PREPAREONLY*".

```
DO ~~/demo/tricore/flash/tc39x.cmm CPU=TC397XE PREPAREONLY
```

From the user's FLASH script, no call to check scripts is required before effective FLASH programming. For TC3xx these checks are performed automatically by TRACE32 PowerView.

The FLASH feature scripts for TC3xx are used to help programming the UCBs. Also, they are used as an additional check against unwanted device locking.

```
NOTE: The procedures and script examples presented in this document requires TRACE32 release 2020/02 or newer.
```

User Configuration Blocks

The FLASH feature script tc3xx-ucb.cmm is used to help programming different UCBs and to test the UCB content. Details about the supported UCBs are documented in the script header.

Example 1: BMHD UCBs

This example shows how the tc3xx-ucb.cmm script is used for programming the UCB_BMHD when they are in a confirmed state. This is indicated by the fields **PROINBMHDxO** and **PROINBMHDxC** of the register **DMU_HF_CONFIRMO**.

The recommended strategy is to start by programming the COPY UCBs first and the ORIG UCBs second. Please refer to the Infineon documentation for details on the recommended strategy.

The main steps to achieve this are the following:

- 1. Perform the FLASH declaration using the appropriate FLASH declaration script.
- 2. Activate UCB_BMHD0_COPY programming by calling the UCB programming script using the "ACTIVATE" argument. This combines the following steps:

- Unlocking the UCB_BMHD password protection. This is only required if the password protection for BMHDs is active.
- Changing the UCB_BMHD0_COPY sector type from "NOP" to "TARGET" to allow FLASH programming by TRACE32 PowerView.
- Enabling FLASH programming for UCB_BMHD0_COPY using FLASH.AUTO command. The latter is used to preserve the unchanged content of the affected FLASH sector.

Unlocking the UCB password protection requires the unlock password to be provided. This is achieved by additionally providing the argument "*PWD=<password>*" like in the following example:

```
&pwd="0x0_0x1_0x2_0x3_0x4_0x5_0x6_0x7"
D0 ~~/demo/tricore/flash/tc3xx-ucb.cmm UCB=BMHD0 COPY ACTIVATE PWD=&pwd
```

Alternatively, a dialog can be used to enter the password manually. This is done by calling the UCB programming script using the argument "*DIALOG*" instead of "*PWD=<password>*".

👠 Dialog	- • •
Enter password for BMHD p	rotection
0x0000000	^
0x0000001	
0x0000002	
0x0000003	
0x00000004	
0x0000005	
0x0000006	
0x0000007	~
OK Cancel	

- 3. Make the desired changes to the UCB_BMHD0_COPY via, e.g., **Data.LOAD** command.
- 4. Program the modifications by calling the UCB programming script using the argument "**PROGRAM**". This will perform the following:
 - Do a formal verification of the UCB_BMHD0_COPY content.
 - If the provided content is formally correct the changes are programmed to the device and the script returns "UCBOK".
 - If the formal checks detected invalid content the FLASH programming is aborted and the script returns the result "UCBFAIL".

In both cases the NOP protection of UCB_BMHD0_COPY sector is restored.

DO ~~/demo/tricore/flash/tc3xx-ucb.cmm **UCB=BMHD0_COPY PROGRAM**

After BMHD0_COPY is programmed, repeat steps 2 to 4 to program BMHD0_ORIG. For this, in step 2 and step 4 the script tc3xx-ucb.cmm must be called with the argument "*UCB=BMHD0_ORIG*" instead of "*UCB=BMHD0_COPY*". Also the address range of UCB_COPY must be replaced by the one for UCB_ORIG as parameter of the Data.LOAD command.

The following example summarizes the full programming flow for BMHD0_COPY.

```
: Perform the FLASH declaration
DO ~~/demo/tricore/flash/tc39x.cmm PREPAREONLY CPU=TC397XE
; Activate UCB modification:
; - Unlock password protection to allow programming of BMHDs
; - Allow FLASH programming by changing the sector type from NOP to
; TARGET
; - Enable FLASH programming for UCB BMHD0 COPY using FLASH.AUTO command
DO ~~/demo/tricore/flash/tc3xx-ucb.cmm UCB=BMHD0 COPY ACTIVATE DIALOG
ENTRY &result
IF ("&result"=="UCBFAIL")
(
 PRINT "UCB BMHD unlock failed"
 ENDDO
)
; Load data of UCB BMHD0 COPY
Data.LOAD.S3record bmhds.s3 0xAF401000++0x1FF
; Program the modifications:
; - Formal verification of the UCB_BMHD0_COPY content
; - The content is formally correct => Program the changes to the device
; - The check detected invalid content => Abort FLASH programming
; - In any case restore NOP protection for UCB_BMHD0_COPY range
DO ~~/demo/tricore/flash/tc3xx-ucb.cmm UCB=BMHD0_COPY PROGRAM
ENTRY &result
IF ("&result"=="UCBOK")
(
 PRINT "UCB_BMHD0_COPY programming successful"
)
```

Example2: PFLASH UCBs

This example shows how the tc3xx-ucb.cmm script is used for programming the PFLASH UCBs when they are in a confirmed state (indicated by the fields **PROINPO** and **PROINPC** of the register **DMU_HF_CONFIRM1**).

The recommended strategy is to start by programming the COPY UCB then the ORIG UCB.

This example is only showing the steps for programming PFLASH_COPY.

- 1. Perform the FLASH declaration using the appropriate FLASH declaration script.
- Activate the UCB modification: This includes unlocking the password protection of the UCB, changing the UCB sector type from "NOP" to "TARGET" and enabling FLASH programming of UCB_PFLASH_COPY.
- 3. Make the desired changes to the UCB.
- Program the modifications: A formal verification of UCB_PFLASH_COPY content will be performed and the changes are programmed to the device only if the formal check passes.

Here is an example script with the full programming flow of the UCB_PFLASH_COPY:

```
; Perform the FLASH declaration
DO ~~/demo/tricore/flash/tc39x.cmm PREPAREONLY CPU=TC397XE
; Activate UCB modification:
; - Unlock password protection to allow programming of PFLASH UCBs
; - Allow FLASH programming by changing the sector type from NOP to
; TARGET
; - Enable FLASH programming for UCB PFLASH COPY using FLASH.AUTO command
&pwd="0x0 0x1 0x2 0x3 0x4 0x5 0x6 0x7"
DO ~~/demo/tricore/flash/tc3xx-ucb.cmm UCB=PFLASH COPY ACTIVATE PWD=&pwd
ENTRY &result
IF ("&result"=="UCBFAIL")
(
  PRINT "UCB PFLASH unlock failed"
 ENDDO
)
: Load data of the PFLASH COPY UCB
Data.LOAD.S3record ucbs.s3 0xAF403000++0x1FF
; Program the modifications:
; - Formal verification of the UCB_PFLASH_COPY content
; - The content is formally correct => Program the changes to the device
; - The check detected invalid content => Abort FLASH programming
; - In any case restore NOP protection for UCB_PFLASH_COPY range
DO ~~/demo/tricore/flash/tc3xx-ucb.cmm UCB=PFLASH COPY PROGRAM
ENTRY &result
IF ("&result"=="UCBOK")
(
  PRINT "UCB_PFLASH_COPY programming successful"
)
```

To extend the script for programming PFLASH_ORIG, the steps 2 to 4, need to be repeated using the address range of the ORIG UCB. Additionally the argument "*UCB=PFLASH_ORIG*" needs to be used instead of "*UCB=PFLASH_COPY*" when calling the script tc3xx-ucb.cmm.

The script tc3xx-ucb.cmm is also used to unlock sector specific write protection of different PFLASHs. In this case the script needs to be called with the argument "UNLOCK" and the corresponding "PRO=PFLASHx" (x in [0..5]).

Example:

```
; Unlock write protection of PFLASH0 enabled in the PFLASH UCBS DO ~~/demo/tricore/flash/tc3xx-ucb.cmm PRO=PFLASH0 UNLOCK DIALOG
```

```
; Unlock write protection of PFLASH5 enabled in the PFLASH UCBS DO ~~/demo/tricore/flash/tc3xx-ucb.cmm PRO=PFLASH5 UNLOCK DIALOG
```

The arguments "**PRO=PFLASHx**" are not to be confused with "**PRO=PFLASH**". The latter is used, to disable global write protection for all PFLASHs, and to unlock the programming of PFLASH UCBs when they are in confirmed state.

For more details, please refer to the description of the register *DMU_HF_PROTECT* in the AURIX[™] TC3xx User's Manual from Infineon.

HSM

When enabling HSM boot it is very important to verify that the HSM contains valid code in its FLASH area and that the boot configuration register **PROCONHSMCBS** is correctly initialized. Otherwise the device gets locked forever and there is no way for the debugger to recover the chip from this state.

TRACE32 FLASH scripts for TC3xx offer two ways to do this safely. The HSM feature script tc3xx-hsmconfig.cmm can be used for easier configuration of the HSM boot. The FLASH declaration scripts install automatic pre-checks for the HSM boot vectors to warn about dangerous operations. The FLASH declaration scripts block any detected fatal operation that may result, e.g., in enabled HSM with invalid boot code and HSM boot settings.

Assuming that the TriCore application is already programmed to the device, the main programming flow is as follows:

- 1. Perform the FLASH declaration using the appropriate FLASH declaration script.
- 2. Enable FLASH programming of the HSM code sectors via the FLASH programming command **FLASH.ReProgram**.
- 3. Load the HSM code using the appropriate **Data.LOAD** command.
- 4. Perform FLASH programming via the command FLASH.ReProgram OFF.
- 5. Program the HSM boot configuration via the HSM feature script tc3xx-hsm-settings.cmm.

Example:

```
; Prepare FLASH programming
DO ~~/demo/tricore/flash/tc39x.cmm CPU=TC397XE PREPAREONLY
; HSM boot address
&bootaddr=0x80090000
; Enable FLASH programming of the address range reserved for HSM
FLASH.ReProgram &bootaddr++0xFFFF
; Load HSM application code
Data.LOAD.Elf hsm_app.elf /NoClear /NoRegister /NosYmbol
FLASH.ReProgram OFF
; Set the HSM boot configuration
&hsm_cfg="UCBHSM_COTP0 BOOTADDR=&bootaddr BOOTINDEX=0 HSMBOOT=ENABLE"
DO ~~/demo/tricore/flash/tc3xx-hsm-config.cmm PROGRAM SETTINGS &hsm cfg
```

The HSM feature script is called either using the argument "**PROGRAM_SETTING**" or "**PROGRAM_REGISTER**".

In both cases the argument "UCBHSM_COTP0" or "UCBHSM_COTP1" is used to specify whether the settings are to be programmed to the UCBs HSMCOTP0 or HSMCOTP1.

The argument "**PROGRAM_SETTING**" is for programming some of the HSM settings by human readable arguments:

- The arguments "*BOOTINDEX=<value> BOOTADDR=<value>*" are to be set together. "*BOOTINDEX*" selects the index [0..3] of the *PROCONHSMCBS* to be programmed, "*BOOTADDR*" sets the boot sector address for the HSM.
- *"HSMBOOT=ENABLE/DISABLE"* is used to enable or disable HSM booting.

Using the argument "**PROGRAM_REGISTER**", the calling script needs to supply the registers values to be programmed to the HSM UCBs. The script tc3xx-hsm-config.cmm starts an HSM configuration dialog to calculate these registers values:

; Open the TC3xx HSM configuration programming dialog **DO** ~~/demo/tricore/flash/tc3xx-hsm-config.cmm

The dialog can also be started from the PowerView using the "HSM configuration" menu in the chip specific menu.

HSM Debug Configuration HSM Test Disable Debug Interface Locked HSM Trace Disable Test Interface Locked HSM Trace Disable Test Interface Locked HSM Trace Disable Configuration HSM Boot Enable SSW Wait Enable HSM Triggering Resets Destructive Debug Entry Destructive Debug Entry HSM RAM Clear: Aldress: Boot address selection Boot SEL2: DxXD Dx80090000 BOOTSEL1: DxXD DxXD Dx80000000 BOOTSEL2: DxXD DxXD Dx80000000 BOOTSEL1: DxXD DxXD Dx80000000 BOOTSEL2: DxXD Dx80000000 None D9	A TC387QP HSM settings						
HSM Debug Disable HSM Test Disable Debug Interface Locked HSM Trace Disable Test Interface Locked HSM Type of Trace: Address+Data Configuration HSM Boot Enable HSM Boot Enable Bable SSW Wait Enable HSM Triggering Resets HSM Data Sectors Exclusive Destructive Debug Entry HSM And Clear: All POR and System Reset Boot address selection Index: Address: BootTSEL0: Dx24 Dx80090000 BOOTSEL2: Dx0 BOOTSEL1: Dx0 Dx80000000 BOOTSEL3: Dx0 MSM Code Sector Exclusive Protection Index: Address: MSM Code Sector Exclusive Protection Index Index 39 38 37 36 35 34 33 32 11 Index HSM Code OTP Protection Index Index Index Address 9 8 7 6 5 4 3 2 11 Index HSM Code Sector Exclusive Protection Index Index Index Index Index I	🗆 HSM Debug Configurati	on					
□ Debug Interface Locked □ HSM Trace Disable □ Test Interface Locked □ HSM Type of Trace: Address+Data ▼ Configuration □ HSM can force pins □ SSW Wait □ Enable HSM Triggering Resets □ SSW Wait □ Destructive Debug Entry HSM RAM Clear: All POR and System Reset ♥ Block Flash Analysis on all ranges Boot address selection Index: Address: BooTSEL0: 0x24 0x80090000 BOOTSEL2: 0x0 0x80000000 BOOTSEL1: 0x0 0x80000000 BOOTSEL3: 0x0 0x80000000 BOOTSEL1: 0x0 0x80000000 BOOTSEL3: 0x0 0x80000000 HSM Code Sector Exclusive Protection □ □ □ □ □ 39 38 37 36 35 34 33 32 11 0 None 9 8 7 6 5 4 3 22 11 0 19 18 17 16 15 14 13 12 11 0 29 28 27 <td>HSM Debug Disable</td> <td></td> <td></td> <td>HSM Te</td> <td>est Disable</td> <td></td> <td></td>	HSM Debug Disable			HSM Te	est Disable		
□ Test Interface Locked HSM Type of Trace: Address+Data ▼ □ Configuration □ HSM Boot Enable □ HSM can force pins □ SSW Wait □ Enable HSM Triggering Resets □ SSW Wait □ Destructive Debug Entry HSM RAM Clear: All POR and System Reset ♥ Block Flash Analysis on all ranges Boot address selection Index: Address: Boot SEL0: 0x24 0x8000000 BOOTSEL2: 0x0 0x80000000 BOOTSEL1: 0x0 0x80000000 BOOTSEL3: 0x0 0x80000000 HSM Code Sector Exclusive Protection □ □ □ □ 39 38 37 36 35 34 33 32 11 0 9 28 27 26 25 24 23 22 11 0 9 8 7 6 5 4 3 32 31 30 All 29 28 27 26 25 24 23 22 11 0 9 8 7 6 5 4 <td>Debug Interface Lock</td> <td>ed</td> <td></td> <td>HSM Tr</td> <td>ace Disable</td> <td></td> <td></td>	Debug Interface Lock	ed		HSM Tr	ace Disable		
Configuration	Test Interface Locked	ł		HSM Type	of Trace:	Address+Dat	ta ~
Configuration HSM Boot Enable HSM Can force pins SSW Wait Enable HSM Triggering Resets HSM Data Sectors Exclusive Destructive Debug Entry HSM RAM Clear: All POR and System Reset Block Flash Analysis on all ranges Boot address selection Index: Address: BOOTSEL0: 0x24 0x80090000 BOOTSEL2: 0x0 BOOTSEL1: 0x0 0x80000000 BOOTSEL3: 0x0 0x80000000 BOOTSEL1: 0x0 0x80000000 BOOTSEL3: 0x0 0x80000000 HSM Code Sector Exclusive Protection 039 38<037							
HSM Boot Enable HSM can force pins SSW Wait Enable HSM Triggering Resets HSM Data Sectors Exclusive Destructive Debug Entry HSM RAM Clear: All POR and System Reset Boot address selection Index: Index: Address: BOOT SEL0: 0x24 0x8000000 BOOTSEL2: 0x0 0x80000000 BOOT SEL1: 0x0 0x80000000 BOOT SEL3: 0x0 0x80000000 BOOT SEL1: 0x0 0x80000000 BOOT SEL3: 0x0 0x80000000 BOOT SEL3: 0x0 0x1 0x80000000 BOOT SEL3: 0x0 0x2 22 11 10 12 11 13 12 19 18 17 16 18 17 16 15 19 18 11 10 19 18 11 10	Configuration						
SSW Wait Enable HSM Triggering Resets HSM Data Sectors Exclusive Destructive Debug Entry HSM RAM Clear: All POR and System Reset Block Flash Analysis on all ranges Boot address selection Index: Address: BOOTSEL0: 0x24 0x80090000 BOOTSEL2: 0x0 0x80000000 BOOTSEL1: 0x0 0x80000000 BOOTSEL3: 0x0 0x80000000 HSM Code Sector Exclusive Protection 0x30 0x80000000 BOOTSEL3: 0x0 0x80000000 HSM Code Sector Exclusive Protection 0x39 0x3 022 021 00 HSM Code Sector Exclusive Protection 0x33 0x3 0x3 0x3 0x3 0x3 HSM Code OTP Protection 0x33 0x3 0x3 0x3 0x3 0x3 HSM Code OTP Protection 0x33 0x3 0x3 0x3 0x3 0x3 HSM Code OTP Protection 0x3 0x3 0x3 0x3 0x3 0x3 HSM Code OTP Protection 0x3 0x3 0x3 0x3 0x3 0x3 19 18 17	HSM Boot Enable			🗌 HSM ca	n force pins		
□ HSM Data Sectors Exclusive □ Destructive Debug Entry HSM RAM Clear: All POR and System Reset ✓ Boot address selection Index: Address: BOOTSEL0: 0x24 0x80090000 BOOTSEL2: 0x0 0x80000000 BOOTSEL1: 0x0 0x80000000 BOOTSEL3: 0x0 0x80000000 HSM Code Sector Exclusive Protection □ □ □ □ 19 18 17 16 15 14 13 12 11 0 HSM Code OTP Protection □ □ □ □ □ □ □ □ □ □ 19 18 17 16 15 14 13 12 11 0 None 19 28 27 26 25 24 23 22 1 0 HSM Code OTP Protection □	SSW Wait			Enable	HSM Triggerii	ng Resets	
HSM RAM Clear: All POR and System Reset ✓ Block Flash Analysis on all ranges Boot address selection Index: Address: Index: Address: BOOTSEL0: 0x24 0x80090000 BOOTSEL2: 0x0 0x80000000 BOOTSEL1: 0x0 0x80000000 BOOTSEL3: 0x0 0x80000000 HSM Code Sector Exclusive Protection	HSM Data Sectors Ex	clusive		Destruc	tive Debug En	itry	
Boot address selection Index: Address: Index: Address: BOOTSEL0: 0x24 0x80090000 BOOTSEL2: 0x0 0x80000000 BOOTSEL1: 0x0 0x80000000 BOOTSEL3: 0x0 0x80000000 HSM Code Sector Exclusive Protection	HSM RAM Clear: All PC	R and System Reset	~	Block Fl	ash Analysis (on all ranges	5
Boot address selection Index: Address: Index: Address: BOOTSEL0: 0x24 0x8009000 BOOTSEL2: 0x0 0x8000000 BOOTSEL1: 0x0 0x80000000 BOOTSEL3: 0x0 0x80000000 HSM Code Sector Exclusive Protection							
Index: Address: Index: Address: BOOTSEL0: 0x24 0x80090000 BOOTSEL2: 0x0 0x80000000 BOOTSEL1: 0x0 0x80000000 BOOTSEL3: 0x0 0x80000000 HSM Code Sector Exclusive Protection	Boot address selection						
BOOTSEL0: 0x24 0x80090000 BOOTSEL2: 0x0 0x8000000 BOOTSEL1: 0x0 0x80000000 BOOTSEL3: 0x0 0x80000000 HSM Code Sector Exclusive Protection	Index:	Address:			Index:	Address:	
BOOTSEL1: 0x0 0x8000000 BOOTSEL3: 0x0 0x8000000 HSM Code Sector Exclusive Protection	BOOTSELO: 0x24	0x80090000) E	BOOTSEL2:	0x0	0x8000	0000
BOOTSEL1: 0x0 0x80000000 BOOTSEL3: 0x0 0x80000000 HSM Code Sector Exclusive Protection		_					
HSM Code Sector Exclusive Protection 39 38 37 36 35 34 33 32 31 30 All 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 None 9 8 7 6 5 4 3 2 1 0 HSM Code OTP Protection 39 38 37 36 35 34 33 32 31 30 All 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 None 9 8 7 6 5 4 3 2 1 0 Select HSMCOTP: HSM_COTP0 Program Refresh Show	BOOTSEL1: 0x0	0x8000000) E	BOOTSEL3:	0x0	0x8000	0000
HSM Code Sector Exclusive Protection 39 38 37 36 35 34 33 32 31 30 All 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 None 9 8 7 6 5 4 3 2 1 0 None 9 18 17 16 15 14 3 2 1 0 None 9 8 7 6 5 24 23 22 21 20 139 38 37 36 35 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 None 9 8 7 6 5 4 3 2 1 0 None 9 8 7							
HSM Code Sector Exclusive Protection 39 38 37 36 35 34 33 32 31 30 All 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 None 9 8 7 6 5 4 3 2 1 0 None HSM Code OTP Protection							
39 38 37 36 35 34 33 32 31 30 All 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 None 9 8 7 6 5 4 3 2 1 0 None 9 8 7 6 5 4 3 2 1 0 HSM Code OTP Protection 39 38 37 36 35 34 33 32 31 30 All 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 None 9 8 7 6 5 4 3 2 1 0 None 9 8 7 6 <t< td=""><td>HSM Code Sector Exclu</td><td>sive Protection</td><td></td><td></td><td></td><td></td><td></td></t<>	HSM Code Sector Exclu	sive Protection					
29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 None 9 8 7 6 5 4 3 2 1 0 None HSM Code OTP Protection	39 38 37	36 35	34	33	32 31	30	All
19 18 17 16 15 14 13 12 11 10 None 9 8 7 6 5 4 3 2 1 0 None HSM Code OTP Protection	29 28 27	26 25	24	23	22 21	20	
9 8 7 6 5 4 3 2 1 0 HSM Code OTP Protection 39 38 37 36 35 34 33 32 31 30 All 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 None 9 8 7 6 5 4 3 2 1 0 Select HSMCOTP: HSM_COTP0 Program Refresh Show		16 15	14	13	12 11	10	None
HSM Code OTP Protection 39 38 37 36 35 34 33 32 31 30 All 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 None 9 8 7 6 5 4 3 2 1 0 Select HSMCOTP: HSM_COTP0 Program Refresh Show	9 8 7	6 5	4	3	2 1	0	
HSM Code OTP Protection 39 38 37 36 35 34 33 32 31 30 All 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 None 9 8 7 6 5 4 3 2 1 0 None Select HSMCOTP: HSM_COTP0 Program Refresh Show							
39 38 37 36 35 34 33 32 31 30 All 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 None 9 8 7 6 5 4 3 2 1 0 Select HSMCOTP: HSM_COTP0 Program Refresh Show	HSM Code OTP Protect	ion					
29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 None 9 8 7 6 5 4 3 2 1 0 Select HSMCOTP: HSM_COTPO Program Refresh Show	39 38 37	36 35	34	33	32 31	30	All
19 18 17 16 15 14 13 12 11 10 None 9 8 7 6 5 4 3 2 11 0 Select HSMCOTP: HSM_COTPO Program Refresh Show	29 28 27	26 25	24	23	22 21	20	
9 8 7 6 5 4 3 2 1 0 Select HSMCOTP: HSM_COTP0 Program Refresh Show	19 18 17	16 15	14	13	12 11	10	None
Select HSMCOTP: HSM_COTP0 V Program Refresh Show	9 8 7	6 5	4	3	2 1	0	
Select HSMCOTP: HSM_COTP0 V Program Refresh Show							
	Select HSMCOTP: HSM	1_COTP0 ~	Program	1	Refresh		Show

- The combo-box "*Select HSMCOTP*" needs to be configured first depending whether HSMCOTP0 or HSMCOTP1 shall be programmed.
- The button "*Program*" is used to apply the changes to the device. This will automatically check if the new configuration is valid according to the HSM code programmed to the device. The changes are discarded otherwise.
- The button "*Refresh*" updates the dialog with the current configuration of the HSM UCBs as programmed in the chip.
- The button "*Show*" generates a script snippet with all the changed settings. The snippet can be copied to the user's FLASH script.

For TriCore AURIX devices, beside write protection with password, two types of sector specific programming protection can be distinguished.

- FLASH sectors that are configured with OTP (One-Time Programmable) protection can not be erased or programmed. Otherwise, FLASH programming errors will be reported by the hardware.
- FLASH sectors that are configured with WOP (Write-page Once Protection) can be programmed once. These sectors can only be programmed if they are in erased state. Erasing these sectors is prevented by hardware after the protection is activated. The hardware will report an error when trying to erase them.

TRACE32 defines the FLASH sectors that must not be erased or programmed as "*NOP*" sectors. Thus OTP protected sectors (as defined by Infineon) are declared by TRACE32 FLASH declaration scripts as NOP sectors.

TRACE32 uses the "*OTP*" term to designate FLASH sectors that can be programmed once because erase is prohibited. Thus, the FLASH sectors that are protected with WOP (as defined by Infineon) are declared by TRACE32 FLASH declaration scripts using the option "*/OTP*".

Sectors declared with *"/OTP*" can only be programmed with the command FLASH.Program and the option *"/OTP*" must be specified.

Example:

```
FLASH.Program 0xA000000++0x3FFF /OTP
Data.Set ...
FLASH.Program OFF
```

More details about FLASH programming of TRACE32 OTP sectors can be found in "Onchip/NOR FLASH Programming User's Guide" (norflash.pdf).

The DFLASH of TC3xx devices supports two operation modes.

- The single-ended mode is the normal operation mode were the full DFLASH size is available.
- The complement sensing mode is a redundant mode were only half the DFLASH size is available and the other, invisible part is used to store the redundancy information as bit-complement.

FLASH programming for the complement sensing mode is supported since TRACE32 release **2019/09**. Whether DFLASH is configured to operate in single ended or complement sensing mode is transparently handled by TRACE32. No special handling by the user's FLASH script is required. As usual, the FLASH declaration is to be done by calling the corresponding FLASH declaration script using the option *"PREPAREONLY"*. Depending on the current device configuration, the corresponding FLASH mapping is declared.

Example:

DO ~~/demo/tricore/flash/tc39x.cmm CPU=TC397XE **PREPAREONLY**

In complement-sensing mode, erased DFLASH is flickering. The **Data.dump** window shows random values, including bus errors. This is a chip behavior. Only a programmed DFLASH will show stable and correct values.

The majority of the TC3xx devices support a product feature named "Software Updates Over The Air (SOTA)". This designates the ability of the device to integrate received software updates by supporting the SWAP mechanism. The basic idea of SWAP is to split the PFLASH into two groups of banks, "**A**" and "**B**", allowing the application to program a new version to the inactive banks while the current version of the application is executing from the active banks. When programming has been completed, the banks are switched. After the next reboot of the chip, the previously inactive banks will become the active banks and the new version of the application will start. The SWAP mechanism ensures that the application will always execute from the same addresses, no matter which physical banks it was programmed to.

When SWAP is enabled, a chip-internal remapping redirects all read-and-fetch accesses to the active address map while the FLASH programming operations (erase/write) keep using the physical system address of PFLASH.



NOTE:	The SOTA feature allows the application to update itself by receiving an update over
	the air. TRACE32 SWAP support is only required during the SOTA development
	phase where FLASH programming and bank switching is implemented and tested.

FLASH Programming with SWAP Enabled

TC3xx FLASH programming with SWAP enabled needs special handling. This is only supported with TRACE32 release **2020/09** or newer.

TRACE32 SWAP support ensures that the address mapping (Standard Address Map or Alternate Address Map) is always displayed as seen by the application, depending on the current configuration.

TRACE32 ensures that FLASH is programmed according to the currently selected address map. FLASH declaration is to be done by calling the corresponding FLASH declaration script using the option *"PREPAREONLY"*.

Example:

DO ~~/demo/tricore/flash/tc39x.cmm CPU=TC397XE **PREPAREONLY**

When performing the FLASH declaration, the *SWAPEN* status of the device and the currently selected address map is checked and the FLASH declaration is performed according to the current setup of the device. The user does not need to know about the current mapping. The user just needs to know if he wants to load the application to the currently active or inactive banks.

If the application has to be programmed to the inactive banks, then it has to be relocated by the user. To achieve this, TRACE32 internal debugger address translation must be configured and enabled using the **TRANSlation** command group.

The following example configures the relocation for tc37x devices:

```
TRANSlation.RESet
TRANSlation.Create C:0x8000000--0x802FFFFF A:0x80300000
TRANSlation.Create C:0xA0000000--0xA02FFFFF A:0xA0300000
TRANSlation.ON
```

SWAP demo scripts are located in the installation directory under "~~/demo/tricore/etc/swap".

NOTE: After FLASH programming to the currently inactive banks is performed the relocation must be disabled before loading the debug symbols. This can be achieved using the command **TRANSlation.RESet**.

	 The mapping between the banks is not linear. For example, for TC39x: PF0 and PF1 are mapped to PF2 and PF3 PF4 is mapped to PF5 The upper 2 Mbytes of PF4 are not accessible when the alternate address map is installed.
--	---

The SWAP Feature Script

For convenience, TRACE32 offers the FLASH feature script tc3xx-swap.cmm. This script can be used to easily configure the SWAP related chip setting. Calling the script with the appropriate arguments results in programming the corresponding UCBs. The changes become effective with the next device reset after the UCBs are re-evaluated by the microcontroller's firmware (SSW).

Example:

DO ~~/demo/tricore/flash/tc3xx-swap.cmm SWAPEN=ON OTP=OTP1 MAP=SWITCH

The argument "SWAPEN=ON" enables swapping in **PROCONTP** register of the OTPx UCB (COPY and ORIG) specified by the argument "**OTP=<otp>**". If not specified, OTP0 is used per default.

Note that **"SWAPEN=OFF**" is only performed for the selected OTP (or OTP0 if no "**OTP=<otp>**" argument is specified). For completely disabling the swapping feature the user needs to make sure that swapping is disabled in all the remaining OTPs.

"MAP=" parameter results in programming UCB_SWAP as follows:

- *"MAP=ALTERNATE"* results in activating the alternate address map after the device reset.
- *"MAP=STANDARD"* results in activating the standard address map after the device reset.
- *"MAP=SWITCH"* results in activating the currently inactive banks after the device reset. The alternate address map will become active when the standard address map was active before reset and vice versa.
- *"MAP=KEEP"* results in preserving the current configuration. Note that disabling the swapping feature automatically switches back to the standard address map.

More details about all the script arguments and other important information are documented in the script header.



When changing any of the SWAP configurations, the FLASH declaration scripts must be called again after the next chip reboot to adjust the FLASH declaration according to the new FLASH mapping. The FLASH declaration script detects the chip configuration and declares FLASH accordingly.