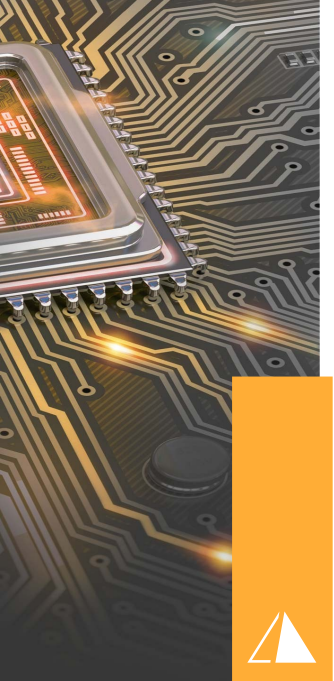


ARM TrustZone

Alex Merkle, Lauterbach Engineering GmbH & Co. KG

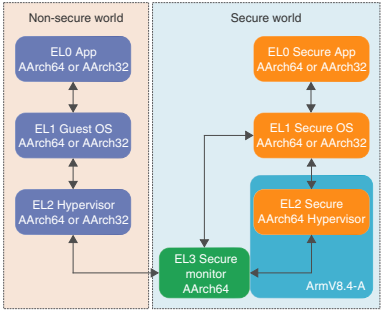
February 1, 2022

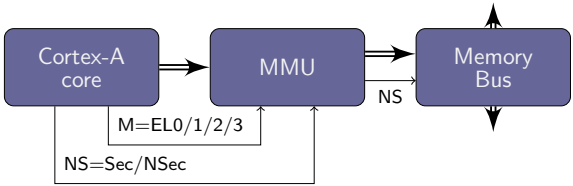
AGENDA

1. TrustZone Basics
2. TrustZone Cortex-A
3. TrustZone Cortex-M
4. Usage Cortex-A
5. Example Cortex-A
6. Usage Cortex-M

TrustZone Cortex-A

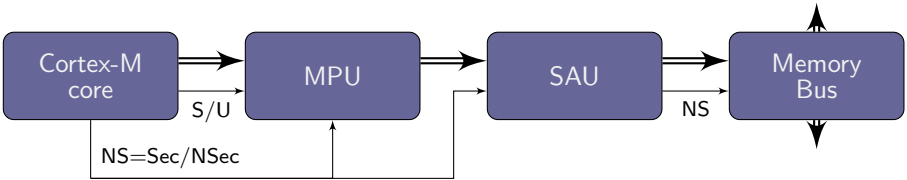
- Cortex-A cores feature multiple *Exception Level* EL0/1/2/3
  EL0 = User, EL1 = Supervisor, EL2 = Hypervisor, EL3 = Monitor
- Additionally there is a *Security State* Secure/NonSecure
- Both *Exception Level* and *Security State* control the MMU
- *Security State* is also connected to the SoC interconnect

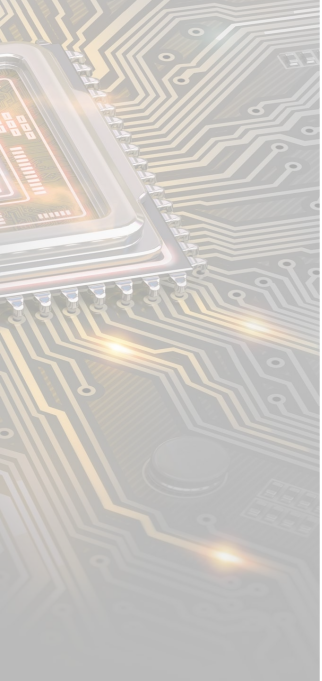




TrustZone Cortex-M

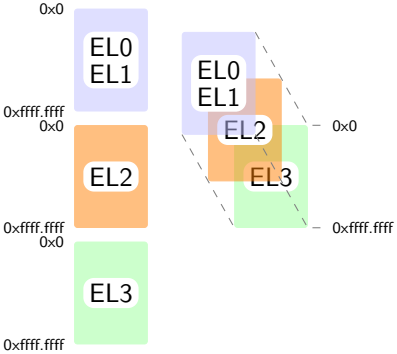
- Cortex-M cores feature Handler and Thread mode
  Handler = Supervisor , Thread = User/NonPrivileged
- Additionally there is a *Security State* Secure/NonSecure
- *Security State* controls the SAU
- *Security State* is also connected to the SoC interconnect





Address Ambiguity

- As the *Exception Level* & *Security State* are input signals for the MMU there exists for every combination a separate address map
- Compilers work with offsets and are not aware of *Exception Level* & *Security State*!
- Conclusion: Address-Offset are ambiguous!
- Every *Exception Level* & *Security State* combination has its own access class within TRACE32

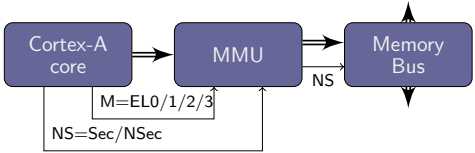


TRACE32 Access Classes

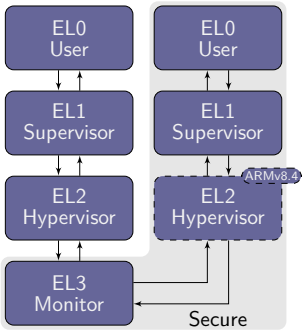
- TRACE32 Access Classes allow to access virtual memory of a specific mode
- TRACE32 combines the specified access class with the current CPU-Mode if the address is not fully qualified

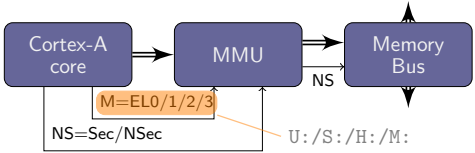
	TrustZone (NS)	
Mode (M)	NonSecure	Secure
EL0/User	NU:<addr>	ZU:<addr>
EL1/Supervisor	NS:<addr>	ZS:<addr>
EL2/Hypervisor	H:<addr>[1]	ZH:<addr>[2]
EL3/Monitor	M:<addr>	

[1] H: is a an alias for NH:
[2] starting from ARMv8.4

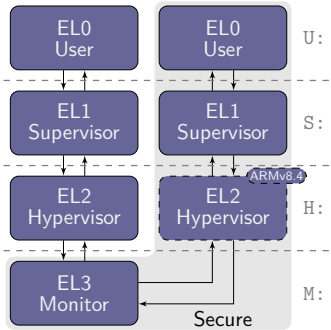


Mode (M)	TrustZone (NS) NonSecure	Secure
EL0/User	NU:<addr>	ZU:<addr>
EL1/Supervisor	NS:<addr>	ZS:<addr>
EL2/Hypervisor	H:<addr>	ZH:<addr>
EL3/Monitor	M:<addr>	

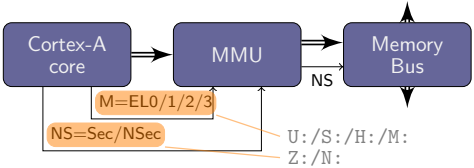


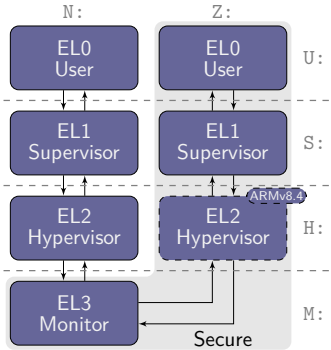


Mode (M)	TrustZone (NS) NonSecure	Secure
EL0/User	NU:<addr>	ZU:<addr>
EL1/Supervisor	NS:<addr>	ZS:<addr>
EL2/Hypervisor	H:<addr>	ZH:<addr>
EL3/Monitor	M:<addr>	



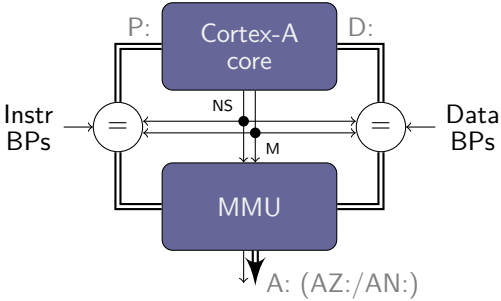
Mode (M)	TrustZone (NS) NonSecure	Secure
EL0/User	NU:<addr>	ZU:<addr>
EL1/Supervisor	NS:<addr>	ZS:<addr>
EL2/Hypervisor	H:<addr>	ZH:<addr>
EL3/Monitor	M:<addr>	





Onchip Breakpoints

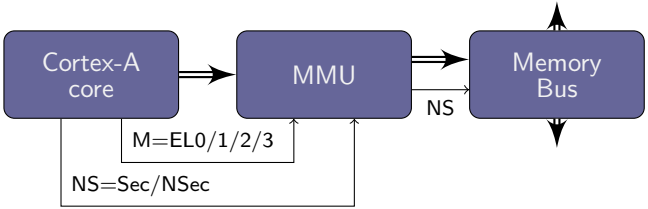
- Onchip Breakpoints can trigger on *Security State* Secure/NonSecure
- Onchip Breakpoints can trigger on *Exception Level* EL0+1/EL2/EL3
- No Onchip Breakpoints to physical memory

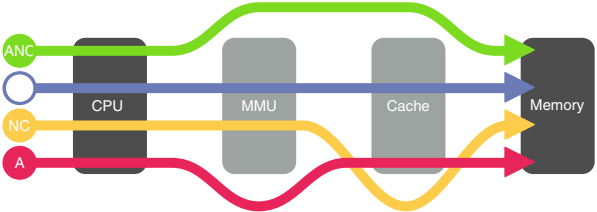


Physical Memory

Access Class A: can be combined with *Security State* as it's connected to the SoC interconnect

`AZ: <-> AN:`

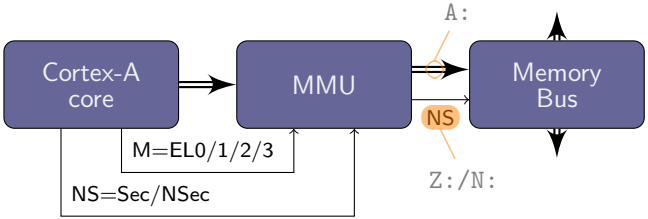


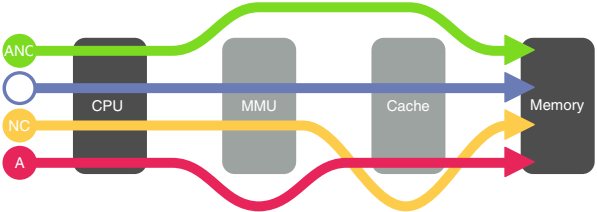


Physical Memory

Access Class A: can be combined with *Security State* as it's connected to the SoC interconnect

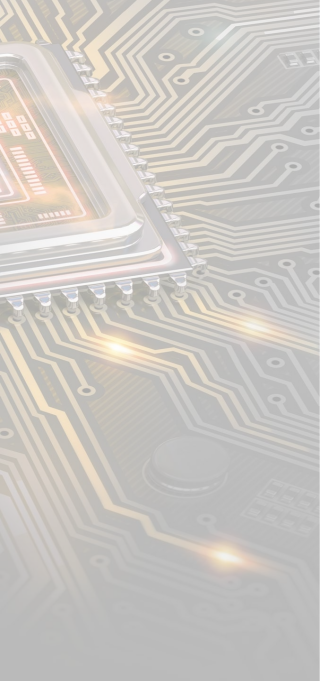
`AZ: <-> AN:`





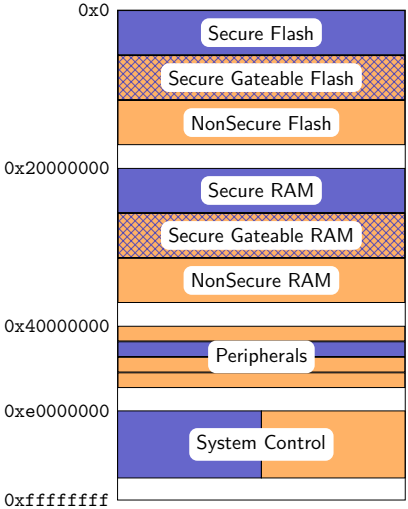
TRACE32 behavior

- *Security State* & *Exception Level* combination is called *ZONE* in TRACE32
- Default - `SYStem.Option ZONESPACES OFF`
  - sYmbol Database is not separated/symbol offsets may be ambiguous
- `SYStem.Option ZONESPACES ON`
  - sYmbol Database is strictly separated
- Default - `Break.CONFIG.MatchZONE OFF`
  - Onchip Breakpoints ignore *Security State* as well as *Exception Level*
- `Break.CONFIG.MatchZONE ON`
  - Onchip Breakpoints respect *Security State* as well as *Exception Level*



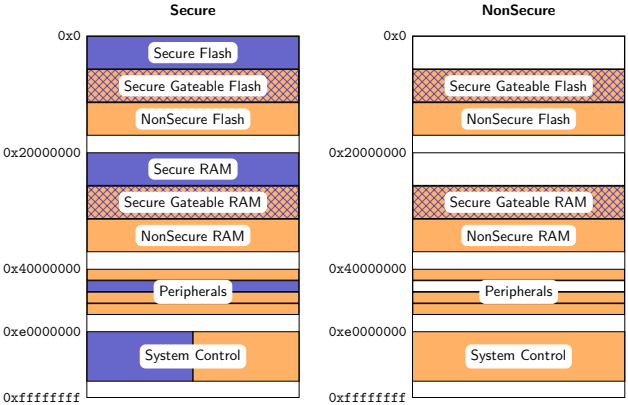
Memory Map

- For Cortex-M addresses are non-ambiguous
- The *Security State* is a input signal for the SAU
- The SAU is software configurable and allows to restrict accesses into memory when the core is in *NonSecure* state
- Secure Gateable region contains sg instructions that allows *NonSecure* code to call *Secure* routines



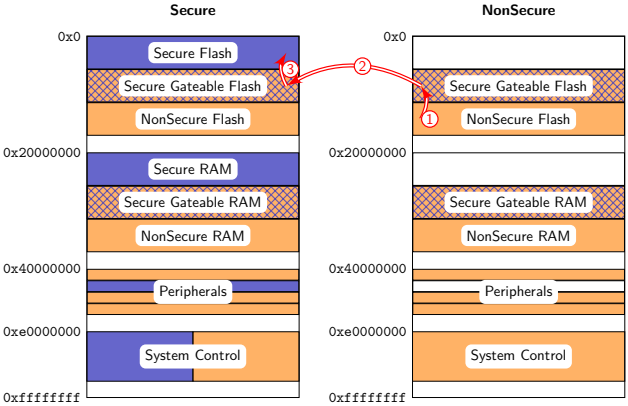
Memory Map

- The SAU works like a blinker/blinder for NonSecure



NonSecure to Secure switch

- Secure Gateable Region is used as a veneer



TRACE32 Access Classes

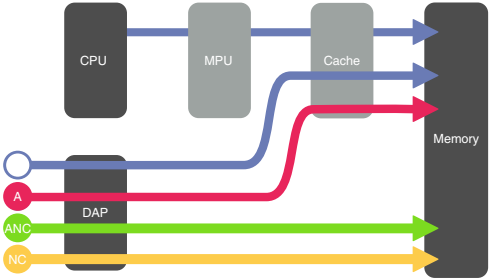
- TRACE32 Access Classes allow to access memory of a specific *Security State*
- TRACE32 combines the specified access class with the current CPU-Mode if the address is not fully qualified

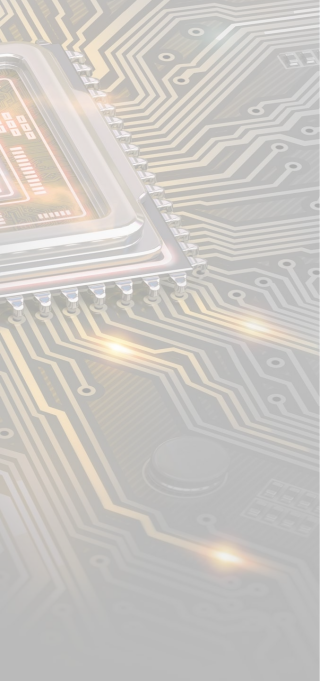
	TrustZone (NS)	
Mode (M)	NonSecure	Secure
EL0/User	NU:<addr>[1]	ZU:<addr>[1]
EL1/Supervisor	NS:<addr>	ZS:<addr>

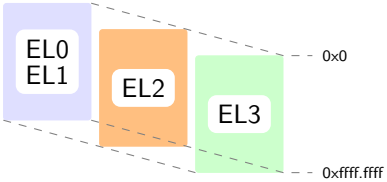
[1] Cortex-M debugger accesses bypass MPU => U: is ignored

Physical Memory

- Access Class A: can be combined with *Security State* as it's connected to the SoC interconnect
  `AZ: <-> AN:`
- Addresses for Cortex-M are always physical







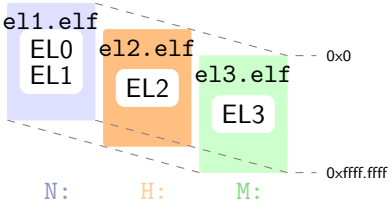
Configuration

```
SYStem.Option ZONESPACES ON
Data.LOAD.Elf el1.elf N:
Data.LOAD.Elf el2.elf H:
Data.LOAD.Elf el3.elf M:
```

Symbol access

Code

Variables



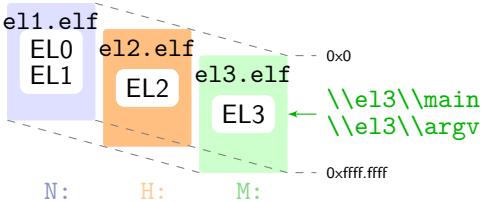
Configuration

```
SYStem.Option ZONESPACES ON
Data.LOAD.Elf el1.elf N:
Data.LOAD.Elf el2.elf H:
Data.LOAD.Elf el3.elf M:
```

Symbol access

Code

Variables

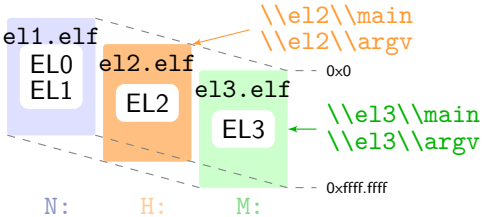


Configuration

```
SYStem.Option ZONESPACES ON
Data.LOAD.Elf el1.elf N:
Data.LOAD.Elf el2.elf H:
Data.LOAD.Elf el3.elf M:
```

Symbol access

Code	Variables
`List \\el3\\main`	`Var.View \\el3\\argv`

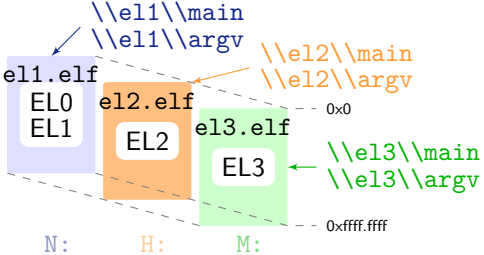


Configuration

```
SYStem.Option ZONESPACES ON
Data.LOAD.Elf el1.elf N:
Data.LOAD.Elf el2.elf H:
Data.LOAD.Elf el3.elf M:
```

Symbol access

Code	Variables
`List \\el3\\main`	`Var.View \\el3\\argv`
`List \\el2\\main`	`Var.View \\el2\\argv`

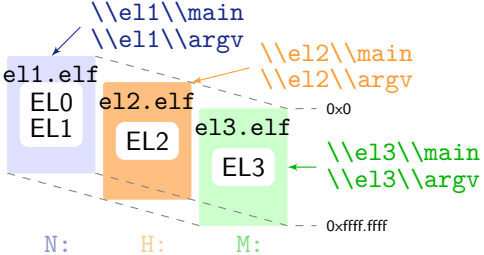


Configuration

```
SYStem.Option ZONESPACES ON
Data.LOAD.Elf el1.elf N:
Data.LOAD.Elf el2.elf H:
Data.LOAD.Elf el3.elf M:
```

Symbol access

Code	Variables
`List \\el3\\main`	`Var.View \\el3\\argv`
`List \\el2\\main`	`Var.View \\el2\\argv`
`List \\el1\\main`	`Var.View \\el1\\argv`



Configuration

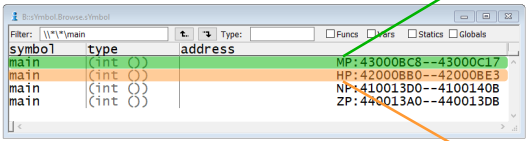
```
SYStem.Option ZONESPACES ON
Data.LOAD.Elf el1.elf N:
Data.LOAD.Elf el2.elf H:
Data.LOAD.Elf el3.elf M:
```

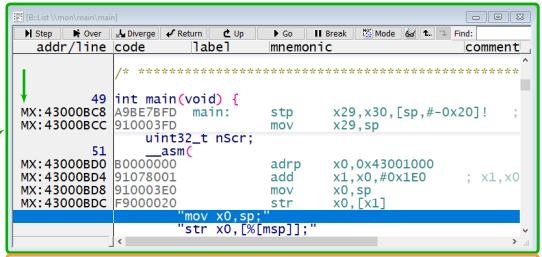
Symbol access

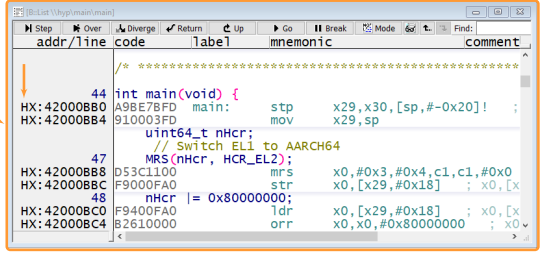
Code	Variables
`List \\el3\\main`	`Var.View \\el3\\argv`
`List \\el2\\main`	`Var.View \\el2\\argv`
`List \\el1\\main`	`Var.View \\el1\\argv`

⇒ TRACE32 allows to combine symbol offsets (from the compiler) with the correct CPU-Mode and thus also the correct underlying virtual memory

Symbols


B:s\?mbol.Browse.s\?mbol
Filter: **\?main
Type:
Funcs
Vars
Statics
Globals
symbol type address
main (int ()) MP:43000BC8--43000C17
main (int ()) HP:42000BB0--42000BE3
main (int ()) NP:4100013D0--4100140B
main (int ()) ZP:440013A0--440013DB

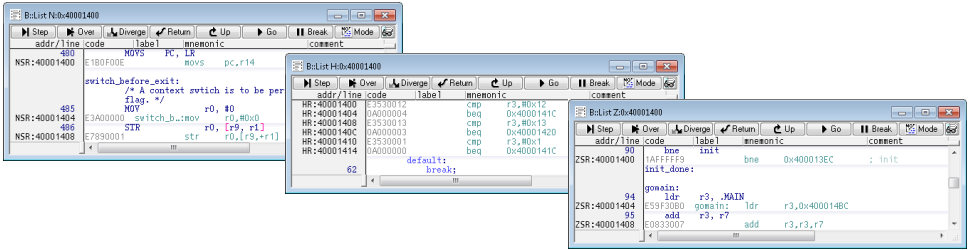

B:List \\mon\main\main
Step
Over
Diverge
Return
Up
Go
Break
Mode
Find:
addr/line code label mnemonic comment
/* ********
49 int main(void) {
MX:43000BC8 A9BE7BFD main: stp x29,x30,[sp,#-0x20]! ;
MX:43000BCC 910003FD mov x29,sp
uint32_t nScr;
51 __asm(
MX:43000BD0 B0000000 adrp x0,0x43001000
MX:43000BD4 91078001 add x1,x0,#0x1E0 ; x1,x0
MX:43000BD8 910003E0 mov x0,sp
MX:43000BDC F9000020 str x0,[x1]
"mov x0,sp;"
"str x0,[%[msp]];"


B:List \\hyp\main\main
Step
Over
Diverge
Return
Up
Go
Break
Mode
Find:
addr/line code label mnemonic comment
/* ********
44 int main(void) {
HX:42000BB0 A9BE7BFD main: stp x29,x30,[sp,#-0x20]! ;
HX:42000BB4 910003FD mov x29,sp
uint64_t nHcr;
// Switch EL1 to AARCH64
47 MRS(nHcr, HCR_EL2);
HX:42000BB8 D53C1100 mrs x0,#0x3,#0x4,c1,c1,#0x0
HX:42000BBC F9000FA0 str x0,[x29,#0x18] ; x0,[x
48 nHcr |= 0x80000000;
HX:42000BC0 F9400FA0 ldr x0,[x29,#0x18] ; x0,[x
HX:42000BC4 B2610000 orr x0,x0,#0x80000000 ; x0

Symbols

```
SYStem.Option ZONESPACES ON
Break.CONFIG.MatchZone   ON
; -----------------------------------------------------
; load sYmbols
Data.LOAD.Elf <code_on_el3.elf>      M: /NoCODE
Data.LOAD.Elf <code_on_el2.elf>      H: /NoCODE
Data.LOAD.Elf <code_on_el1_ns.elf>   N: /NoCODE
Data.LOAD.Elf <code_on_el1_s.elf>    Z: /NoCODE
```

Different code and different sourcecode & memory for each zone



sYmbols

Functions & Variables are automatically accessed with the correct zone

```
B::Var.View %Location flags RoundKey
[ND:0x40005F94] flags = (1, 1, 1, 0, 1, 1, 0, 1,
[ZD:0x40002A5C] RoundKey = (43, 126, 21, 22, 40,
```

```
B::Ymbol.name main
        path        symbol  type      address
   \\hv\hypdemo\    main    (int ())   NP:4000135C--4000136F
 \\aes\securedemo\  main    (int ())   ZP:40001680--4000170F
\\freertos\rtosdemo\ main   (int ())   NP:40001E38--40001FD3
```

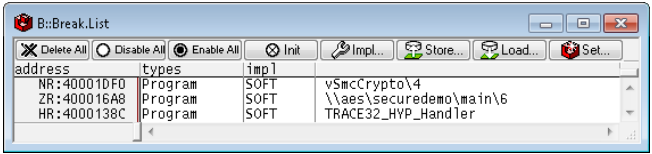
```
[B::List \\freertos\\main ]
Step  Over  Return  Up  Go  Break  Mode
addr/line   code      label   mnemonic   commen
      209   int main() {
NSR:40001E38  E92D4810  main:  push   {r4
NSR:40001E3C  E28D8008         add    r11
NSR:40001E40  E24DD014         sub    r13
NSR:40001E44  E59F415C         ldr    r4,
NSR:40001E48  E08F4004         add    r4,
      210     Init();
NSR:40001E4C  EBFFFDBC         bl     0x4
```

```
B::List \\hv\\main
Step  Over  Return  Up
addr/line   code      label   mnemonic
       31   int main() {
HR:4000135C  E52DB004  main:  push   {r11
HR:40001360  E28DB000         add    r11
       33     asm volatile ("bkpt #0; no
HR:40001364  E1200070         bkpt   #0x0
```

```
B::List \\aes\\main
Step  Over  Return  Up  Go  Break  Mode
addr/line   code      label   mnemonic   commen
       25   int main(int argc, char* argv[])
            {
ZSR:40001680  E92D4800  main:  push   {r11,r14}
ZSR:40001684  E28DB004         add    r11,r13,#0x4     ; r11,
ZSR:40001688  E24DD010         sub    r13,r13,#0x10    ; r13,
ZSR:4000168C  E50B0010         str    r0,[r11,#-0x10]
ZSR:40001690  E50B1014         str    r1,[r11,#-0x14]
       26         int counter = '1';
ZSR:40001694  E3A03031         mov    r3,#0x31         ; r3,#
```

Breakpoints

```
Break.CONFIG.MatchZone ON
; ----------------------------------------------------
; set BPs
Break.Set  M:<addr> [/Onchip] </Program|/ReadWrite|/Read|/Write>
Break.Set  H:<addr> [/Onchip] </Program|/ReadWrite|/Read|/Write>
Break.Set  NS:<addr> [/Onchip] </Program|/ReadWrite|/Read|/Write>
Break.Set  ZS:<addr> [/Onchip] </Program|/ReadWrite|/Read|/Write>

Break.Set <symbol_name> [/Onchip]
```



Break.CONFIG.MatchZONE

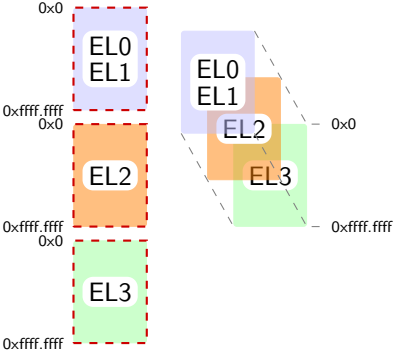
Configuration

```
SYStem.Option ZONESPACES ON
Break.CONFIG.MatchZone  <ON|OFF>
```

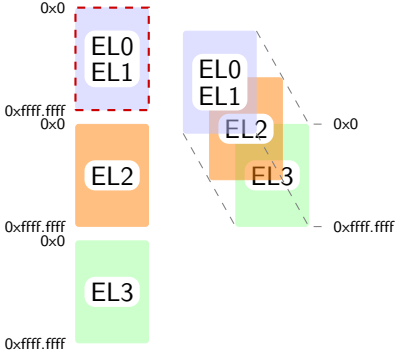
```
Break.Set <Zone>:<offset> /Onchip

            ON
        MatchZone
           OFF

Break.Set <Zone>:<offset> /Onchip
          IGNORE
```

Break.CONFIG.MatchZONE

`Break.CONFIG.MatchZONE OFF`
`Break.Set N:<offset> /Onchip`



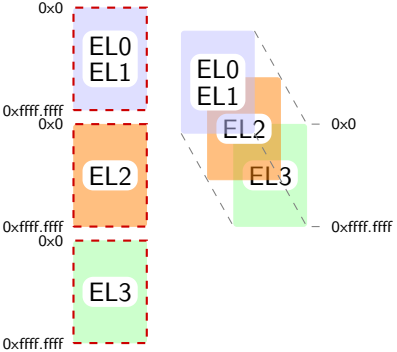
`Break.CONFIG.MatchZONE ON`
`Break.Set N:<offset> /Onchip`



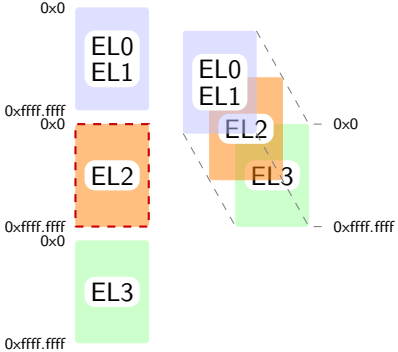
Hint: Example shows NonSecure only

Break.CONFIG.MatchZONE

```
Break.CONFIG.MatchZONE OFF
Break.Set H:<offset> /Onchip
```



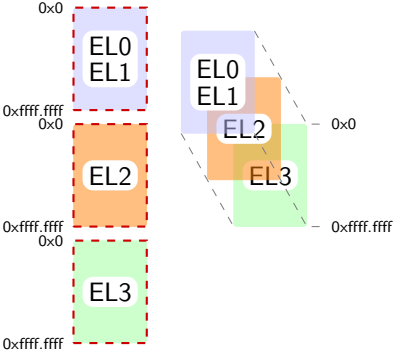

```
Break.CONFIG.MatchZONE ON
Break.Set H:<offset> /Onchip
```



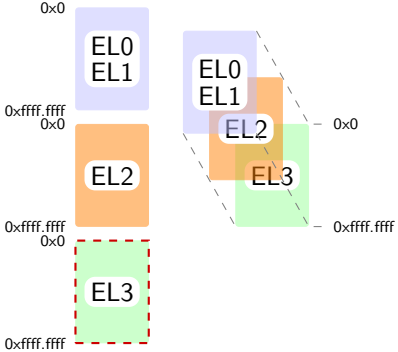
Hint: Example shows NonSecure only

Break.CONFIG.MatchZONE

`Break.CONFIG.MatchZONE OFF`
`Break.Set M:<offset> /Onchip`



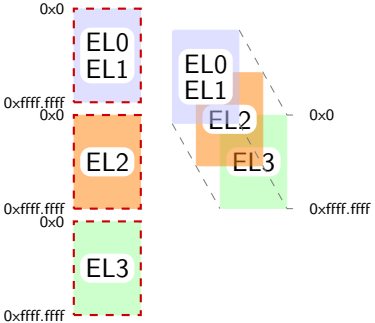
`Break.CONFIG.MatchZONE ON`
`Break.Set M:<offset> /Onchip`



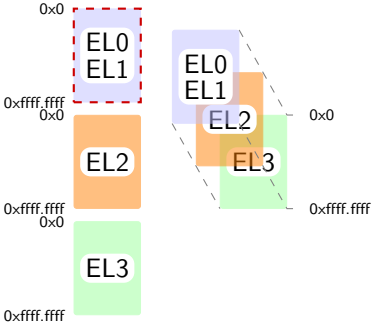
Hint: Example shows NonSecure only

Break.CONFIG.MatchZONE

```
Break.CONFIG.MatchZONE OFF
Data.LOAD sieve.elf N:
Break.Set myVar /ReadWrite /Onchip
```



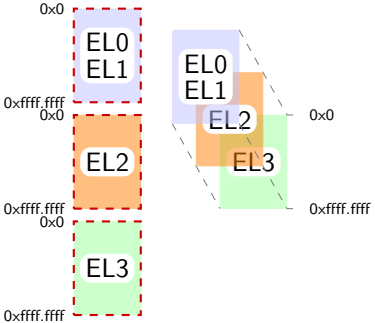

```
Break.CONFIG.MatchZONE ON Data.LOAD
sieve.elf N:
Break.Set myVar /ReadWrite /Onchip
```



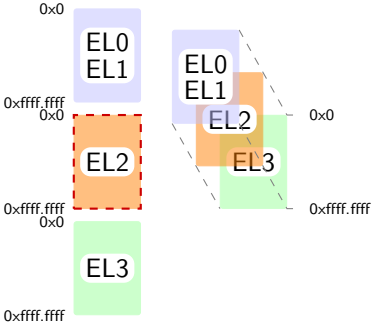
Hint: Example shows NonSecure only

Break.CONFIG.MatchZONE

```
Break.CONFIG.MatchZONE OFF
Data.LOAD sieve.elf H:
Break.Set myVar /ReadWrite /Onchip
```



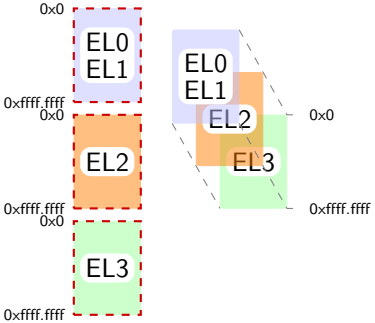

```
Break.CONFIG.MatchZONE ON
Data.LOAD sieve.elf H:
Break.Set myVar /ReadWrite /Onchip
```



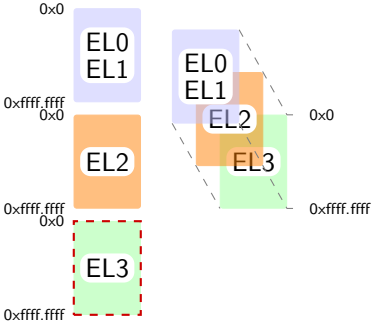
Hint: Example shows NonSecure only

Break.CONFIG.MatchZONE

```
Break.CONFIG.MatchZONE OFF
Data.LOAD sieve.elf M:
Break.Set myVar /ReadWrite /Onchip
```




```
Break.CONFIG.MatchZONE ON
Data.LOAD sieve.elf M:
Break.Set myVar /ReadWrite /Onchip
```



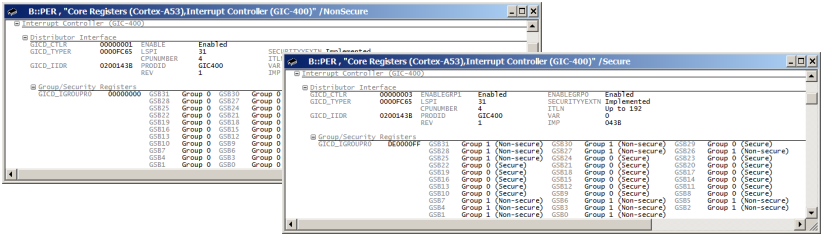
Hint: Example shows NonSecure only ⇒ Access class is derived from symbol

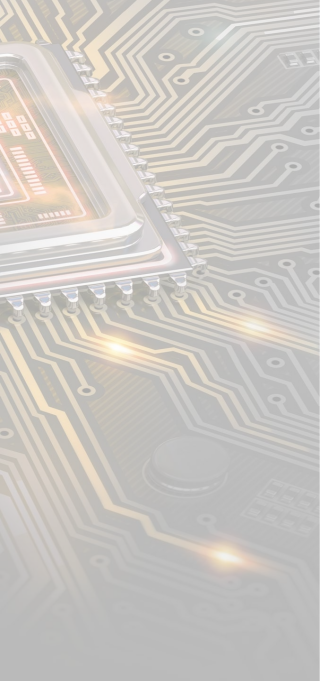
Access Memory

```
; ----------------------------------------------------
; access memory
Data.Set   M:<addr> <value>
Data.Set   H:<addr> <value>
Data.Set  NS:<addr> <value>
Data.Set  ZS:<addr> <value>
Data.dump  M:<addr>
Data.dump  H:<addr>
Data.dump NS:<addr>
Data.dump ZS:<addr>
; ----------------------------------------------------
; access physical memory
Data.Set   A:<addr> <value>
Data.Set  AZ:<addr> <value>
Data.Set  AN:<addr> <value>
Data.dump  A:<addr>
Data.dump AZ:<addr>
Data.dump AN:<addr>
```

Peripherals

```
; ---------------------------------------------------------------
; access physical memory
Data.Set  <A|AZ|AN>:<addr> <value>
Data.dump <A|AZ|AN>:<addr>
; ---------------------------------------------------------------
; peripheral view
PER
PER , /NonSecure
PER , /Secure
PER , /ACCESS <NS:|H:|M:|ZS:>
```





Debugging TF-A

```
; ----------------------------------------------------
; connect
SYStem.CPU 88F3720-A0
SYStem.Option ZONESPACES ON
Break.CONFIG.MatchZone   ON
CORE.ASSIGN 1.
SYStem.Up

; ----------------------------------------------------
; load Symbols
; Bootloader 1: bl1 running in Monitor/EL3
Data.LOAD.Elf ~~~~/atf/build/a3700/debug/bl1/bl1.elf   M: /NoCODE
; Bootloader 2: bl2 running in Secure Supervisor/EL2
Data.LOAD.Elf ~~~~/atf/build/a3700/debug/bl2/bl2.elf   Z: /NoCODE /NoCLEAR
; Bootloader 3: bl31 running in Monitor/EL3
Data.LOAD.Elf ~~~~/atf/build/a3700/debug/bl31/bl31.elf M: /NoCODE /NoCLEAR
; Bootloader 4: u-boot running in Hypervisor/EL2
Data.LOAD.Elf ~~~~/u-boot/u-boot                       H: /NoCODE /NoCLEAR

Go bl1_entrypoint /Onchip /PROGRAM
WAIT !STATE.RUN()

Go bl2_entrypoint /Onchip /PROGRAM
WAIT !STATE.RUN()

Go bl31_entrypoint /Onchip /PROGRAM
WAIT !STATE.RUN()

Go __image_copy_start /Onchip /PROGRAM
WAIT !STATE.RUN() 10s
```

Debugging TF-A

Hints

- Keep the complete code in the bootsource (e.g. Flash / SD-Card)
  ⇒ Boot sequence is automatized
- Use `/Onchip` Breakpoints to trigger at every bootstage entry
  ⇒ Possibility to patch the code
- Use
  ```
  TRANSlation.TableWalk ON
  TRANSlation.ON
  ```
  to enable TRACE32 automatic MMU-TableWalk
  ⇒ Software Breakpoints within every zone even though the `.text` segment is read-only (via MMU)

Debugging TF-A
Pitfalls

- Bootflow is SoC specific - check BSP
- To debug into the bootloaders requires to connect the debugger early after reset
  Highly SoC & Board specific!
- Not every SoC supports that!
  ⇒patch an endless loop into the bootimage (last resort)
- TRACE32 is preconfigured for SMP debugging after `SYStem.CPU`
  ⇒ use `CORE.ASSIGN <x>` to select the bootcore(s) only

⇒ example scripts `~~/demo/arm/hardware/<chip|board>`

⇒ support@lauterbach.com

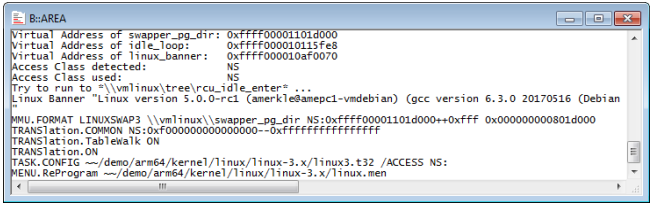
Debugging Linux + bl31

```
; ----------------------------------------------------
; connect
SYStem.CPU 88F3720-A0
SYStem.Option ZONESPACES ON
SYStem.Option MMUSPACES  ON
Break.CONFIG.MatchZone   ON
SYStem.Mode Attach
Break

; ----------------------------------------------------
; load Symbols
; bl31 running in Monitor/EL3
Data.LOAD.Elf ~~~~/atf/build/a3700/debug/bl31/bl31.elf M:  /NoCODE
; linux running in NonSecure Supervisor/EL1
Data.LOAD.Elf ~~~~/linux/vmlinux                       NS: /NoCODE /NoCLEAR

; ----------------------------------------------------
; configure debugger address translation
MMU.FORMAT LINUXSWAP3 \\vmlinux\\swapper_pg_dir NS:<virtual range>  <physical>
TRANSlation.COMMON NS:0xf000000000000000--0xffffffffffffffff
TRANSlation.TableWalk ON
TRANSlation.ON

TASK.CONFIG    ~~/demo/arm/kernel/linux/awareness/linux.t32 /ACCESS NS:
MENU.ReProgram ~~/demo/arm/kernel/linux/awareness/linux.men
```

- Use ~~/demo/arm/kernel/linux/boards/generic-template/detect_translation.cmm
- Use ~~/demo/arm/kernel/linux/boards/generic-template/linux-attach.cmm

detect_translation.cmm

- First official release DVD2019.02
- Supports KAISER/MELTDOWN patch
- Steps
  - Linux must be up and running
  - Debugger is connected ⇒ `SYStem.Mode Attach`
  - Debug symbols must be loaded ⇒ `Data.LOAD.Elf ...`
  - DO `~~/demo/arm/kernel/linux/boards/generic-template/detect_translation.cmm`



- See also: `~~/demo/arm/kernel/linux/boards/generic-template/linux-attach.cmm`

Debugging OP-TEE

```
; ---------------------------------------------------
; connect
SYStem.CPU ZYNQ-ULTRASCALE+-APU
SYStem.Option ZONESPACES ON
SYStem.Option MMUSPACES  ON
Break.CONFIG.MatchZone   ON
SYStem.Mode Attach
Break

; ---------------------------------------------------
; load Symbols
; op-tee running in Secure Supervisor/EL1
Data.LOAD.Elf out/arm-plat-zynqmp/core/tee.elf ZS: /NoCODE

; ---------------------------------------------------
; configure debugger address translation
TRANSlation.Create ZS:<optee virtual range> AZ:<optee physical>
TRANSlation.COMMON ZS:<optee virtual range>
TRANSlation.TableWalk ON
TRANSlation.ON

TASK.CONFIG    ~~/demo/arm/kernel/op-tee/optee.t32 /ACCESS ZS:
MENU.ReProgram ~~/demo/arm/kernel/op-tee/optee.men
```

Debugging OP-TEE

Hints

- Op-Tee uses a paging approach with ambiguous TA address space
  ⇒ `SYStem.Option MMUSPACES ON`
- The Op-Tee default translation needs to be configured manually, by Nov 2019 we do not support `MMU.FORMAT`
  ⇒ `TRANSlation.Create`
- Use
  `TRANSlation.TableWalk ON`
  `TRANSlation.ON`
  to enable TRACE32 automatic MMU-TableWalk
  ⇒ Software Breakpoints are available for TAs despite the `.text` segment is read-only (via MMU)

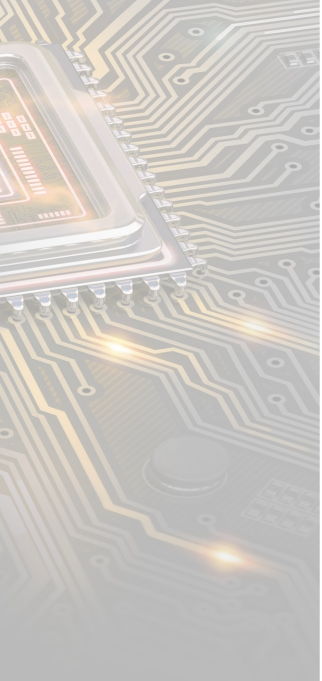
Debugging OP-TEE - Limitations

Limitations

Status November 2019

- The standard workflow foresees that a TA is
  - loaded - `TEEC_OpenSession`
  - debugged - `TEEC_InvokeCommand`
  - unloaded - `TEEC_CloseSession`

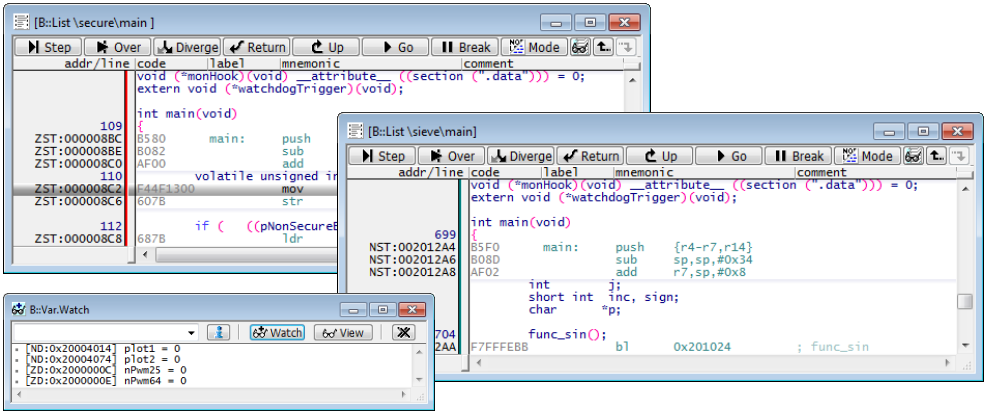
  triggered by the optee menu *Debug Trusted Application from Entry*.
- For TAs that are loaded persistently (no `TEEC_CloseSession`), possibly ambiguous Onchip-Breakpoints need to be used



Symbols

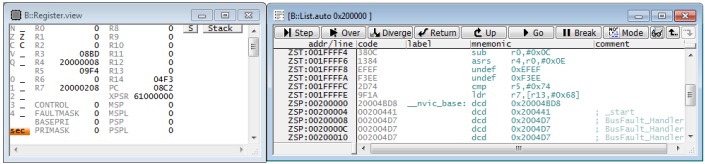
```
; ----------------------------------------------------
; load sYmbols
Data.LOAD.Elf <secure.elf>     Z: /NoCODE
Data.LOAD.Elf <non_secure.elf> N: /NoCODE /NoCLEAR
```

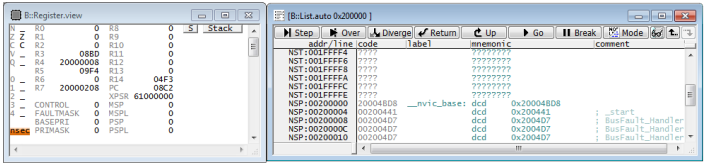
Functions & Variables are automatically accessed with the correct zone



Access Memory

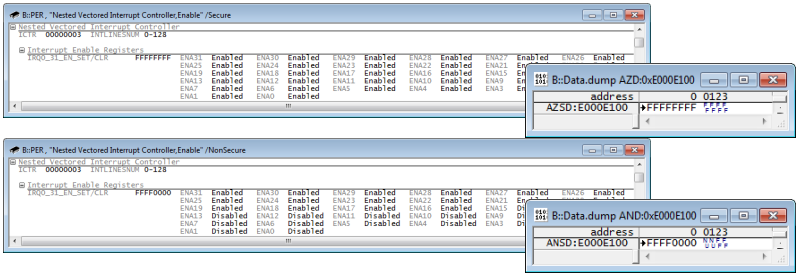
```
; ------------------------------------------------------
; access memory
Data.Set  <N|Z>:<addr> <value>
Data.Set  <addr> <value>
Data.dump <N|Z>:<addr>
Data.dump <addr>
```





Peripherals

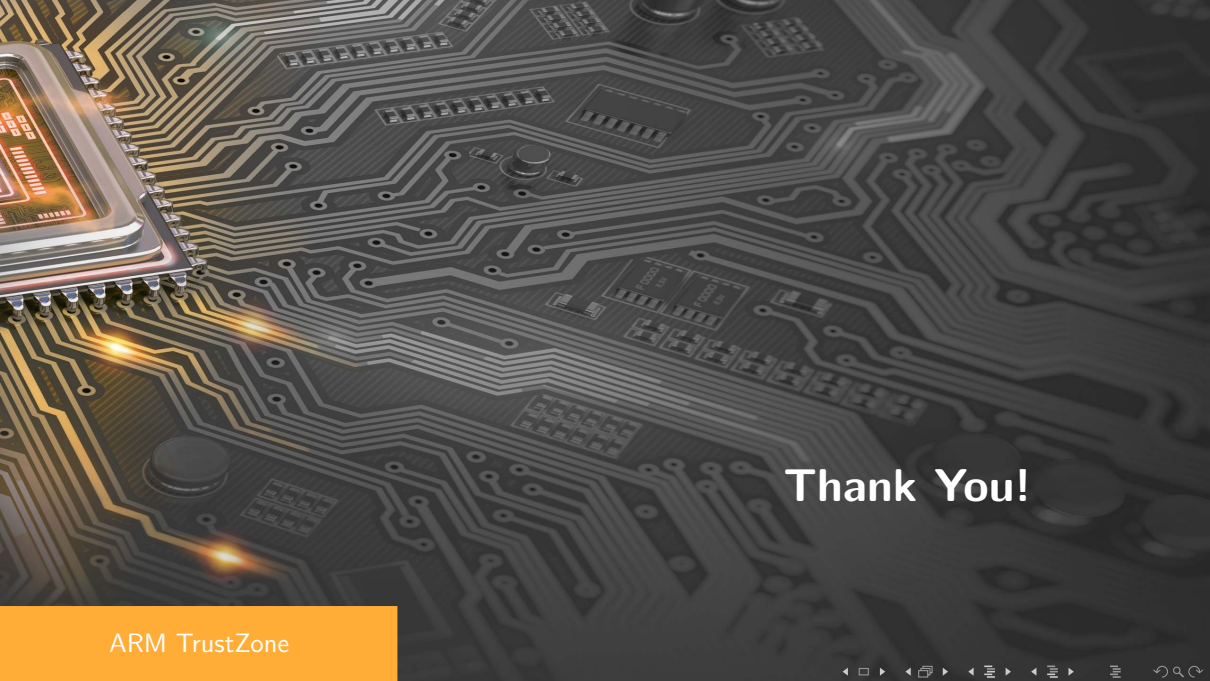
```
; ----------------------------------------------------
; access physical memory
Data.Set  <A|AZ|AN>:<addr> <value>
Data.dump <A|AZ|AN>:<addr>
; ----------------------------------------------------
; peripheral view
PER
PER , /NonSecure
PER , /Secure
```



Flash programming

```
SYStem.CPU <cpu>
SYStem.Up

; ------------------------------------------------------------------------------
; Flash programming
; prepare flash programming (declarations)
DO ~~/demo/arm/flash/<driver>.cmm PREPAREONLY
; ReProgram Flash
FLASH.ReProgram ALL
Data.LOAD.Elf <secure.elf>        /NosYmbol
Data.LOAD.Elf <non_secure.elf>    /NosYmbol
FLASH.ReProgram OFF

; ---------------------------------------------------
; load sYmbols
Data.LOAD.Elf <secure.elf>     Z: /NoCODE
Data.LOAD.Elf <non_secure.elf> N: /NoCODE /NoCLEAR
```

Questions?



Thank You!