LAUTERBACH
DEVELOPMENT TOOLS

Peripheral Files Programming

MANUAL

Peripheral Files Programming

Peripheral Files Programming

Version 06-Jun-2024

History

14-Sept-2022 New feature **inherit** for REPEAT command.

21-Jan-2022 Added %y... placeholder to **BITLD** and **ABITFLD**.

Introduction

This document describes the commands which are used to write peripheral files. This allows to display/manipulate configuration registers and the on-chip peripheral registers at a logical level. Registers and their contents are visible and accessible in the **PER.view** window.

Peripherals in MCU can be displayed and manipulated with the **PER** commands. TRACE32 offers configurable window for displaying memory or I/O structures. Displaying the state of peripheral components or memory based structures is very comfortable.

User can define 'chip macros' and put them together to generate 'project files'. These files describe the port structure for a specific hardware system.

Examples for different microcontrollers reside in the directory **~~/demo/per/**.

Passing Arguments

You can pass arguments from a PRACTICE script to a PER file (peripheral file). These arguments can be strings, hex and decimal values. See below for an example and an illustration and explanation of the example.

Example

PRACTICE script (*.cmm) - Bold and red are used to highlight the information flow:

```
;Declare four PRACTICE macros and assign values to the PRACTICE macros
LOCAL &addr &reg64bit &name &idx

&addr=0xE0000000  ;Base address of the PER file called with PER.view.
&reg64bit=1.      ;Show the 64bit or the 32bit specific register group.
&name="My Module" ;Module description of the register group.
&idx=35.          ;Show a specific register out of an array of
                  ;memory-mapped registers.
;... your code
SYStem.Up

;View the peripheral file and pass the four arguments
PER.view "per_with_args.per"  &addr  &reg64bit  "&name"  &idx  "*"

;Open the peripheral file in the built-in TRACE32 editor PER.Program
PER.Program   "per_with_args.per"    ;Do not pass arguments here!
;Set a different peripheral file as temporary new default file
PER.ReProgram "per_with_args.per"    ;Do not pass arguments here!
```

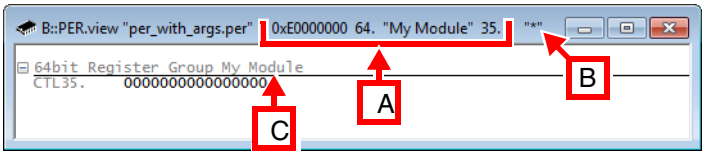
The above PRACTICE script (*.cmm) calls this PER file (*.per):

```
CONFIG 16. 8.
WIDTH 10.

;The PER.view command arguments are passed to the ENTRY command arguments
ENTRY &baseaddr=0x0 &reg64bit=0. &modulename="foo" &index=1.

BASE D:&baseaddr

SIF (&reg64bit==1.)
  TREE "64bit Register Group &modulename"
   GROUP.QUAD (0x8*&index)++0x07
    LINE.QUAD 0x00 "CTL&index,Control Register &index"
  TREE.END
ELSE
  TREE "32bit Register Group &modulename"
   GROUP.LONG (0x4*&index)++0x07
    LINE.LONG 0x00 "CTL&index,Control Register &index"
  TREE.END
ENDIF
```

NOTE: Although the **ENTRY** command arguments may look like PRACTICE macros, they are *not* PRACTICE macros and do *not* behave like PRACTICE marcos:

- When you try to create PRACTICE macros with the **LOCAL** command inside a PER file and compile it, you receive the error message “unknown command”.
- When you try to assign an **ENTRY** command argument to another **ENTRY** command argument (`&arg2=&arg1`) inside a PER file and compile it, you also receive the error message “unknown command”.

Our example produces this **PER.view** window:



A The four values passed to the PER file are displayed in the window caption.

B `"*"` displays all branches. For more information, see **PER.view**.

C Result of the information flow highlighted in bold and red in the above example (see `&name`).

NOTE: In the PER file, valid default values must be assigned to each **ENTRY** command argument. See highlighted values in the **ENTRY** line.

The default values in the **ENTRY** line ensure that no “syntax error” is reported when a PER file is compiled in the built-in TRACE32 editor **PER.Program**.

```
;Define default values for the ENTRY command arguments
ENTRY &baseaddr=0x0 &reg64bit=0. &modulename="foo" &index=0.
```

As valid default values in a PER file, our example uses:

- **`0x0`** for hex values.
- **`0.`** for decimal values.
- **`"foo"`** for strings.

When the PRACTICE macro values are passed to the same PER file, the passed values override the default values in the **ENTRY** line of the PER file.

Format: *<access_class>*:*<base_address>*

<access_class> Appropriate access method to memory class (**D**, **SD**, **A**, **AD**, **AP**, **ANC**,**DC**, **IC**, **NC**, **ED**, **EAD**, **VM**, **P, etc.**)

<base_address> Base address of the peripheral module.

Comma-Separated-Values (CSV) File Format for *.per Files

Peripheral files can be formatted as comma-separated values, i.e. the same format as in *.csv files. However, the file extension for peripheral files remains *.per, as usual. The CSV format extends the regular peripheral command set and offers you an alternative way to create and maintain peripheral files more easily in a spreadsheet. Therefore it usually offers better readability. Peripheral files in CSV format can also be generated more easily from binary files (such as netlists, etc.) by automated tools.

Example: Regular *.per file format (excerpt from ~~/demo/per/percsv_nocsv.per):

```
TREE "Common Registers"
GROUP 0xE80++0x01
LINE.WORD 0x0 "ADCR1,ADC Control Register 1"
BITFLD.WORD 0 14. "STOP,Stop", "Normal operation,Stop"
BITFLD.WORD 0 13. "START,Start Conversion", "No action,Start"
BITFLD.WORD 0 12. "SYNC,Sync Select","START bit,sync input or START bit"
GROUP 0xF80++0x01
LINE.WORD 0x0 "ADCR2,ADC Control Register 2"
HEXMASK.WORD.BYTE 0 0.--3. 1. "DIV,Clock Divisor Select"
TREE.END
```

The same register definitions in CSV format and displayed in a spreadsheet editor (excerpt from ~~/demo/per/percsv_simple.per):

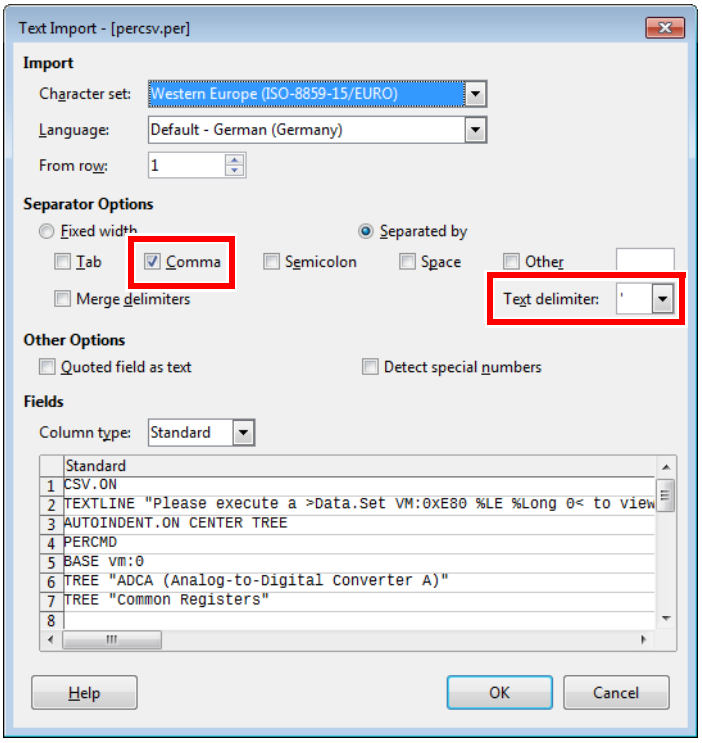
PERCMD	Address	AccessWidth	Name	Tooltip	From	To	Choices
TREE "Common Registers"							
	0xe80	16.	ADCR1	ADC Control Register 1			
			STOP	Stop	14.	14.	Normal operation,Stop
			START	Start Conversion	13.	13.	No action,Start
			SYNC	Sync Select	12.	12.	START bit,sync input or START bit
	0xf80	16.	ADCR2	ADC Control Register 2			
			DIV	Clock Divisor Select	0.	3.	
TREE.END							

Whenever necessary, you can still mix the regular and CSV file format.

NOTE: Microsoft Excel is not capable of exporting true comma-separated-values files on machines based in Europe (instead semicolons will be used as separators due to system-wide Region and Language settings).
Therefore it is recommended to use LibreOffice Calc or any other spreadsheet editor.

1. Do one of the following:

   - Create an empty file, or
   - Open/Import an existing *.per file. Make sure comma is selected as separator and the single quote as text delimiter:



2. The first command in the *.per file (except comments) must enable CSV capabilities:

```
CSV.ON
```

3. Optional step: Use your preferred auto-indent style (see **AUTOINDENT**):

```
AUTOINDENT.ON CENTER TREE
```

4. Optional step: Define the columns (see **PERCMD**).

   - The column name arguments of the **PERCMD** command will serve as column headers in your spreadsheet, see [**X**] below.

CSV.ON								
AUTOINDENT.ON CENTER TREE								
PERCMD	**X** →	**Address**	**AccessWidth**	**Name**	**Tooltip**	**From**	**To**	**Choices**
BASE x:0								
TREE "ADCA (Analog-to-Digital Converter A)"								

   - To freeze the headers, choose **View** menu > **Freeze Rows and Columns**.
   - If you omit the **PERCMD** command: The first column must always contain peripheral file commands only and must be kept empty otherwise!

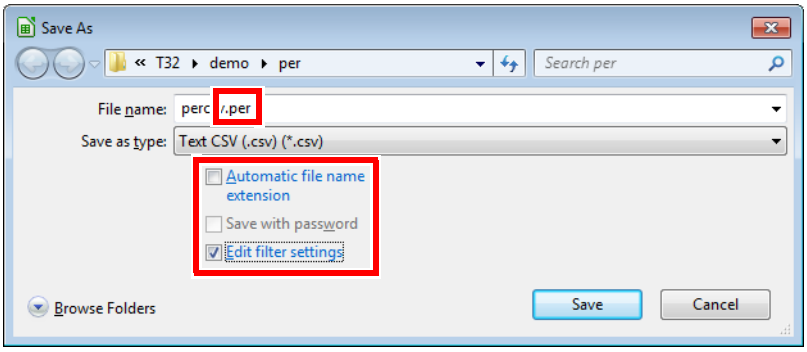
5. Optional step: Use **BASE** and **TREE** commands in the subsequent rows to create an environment.

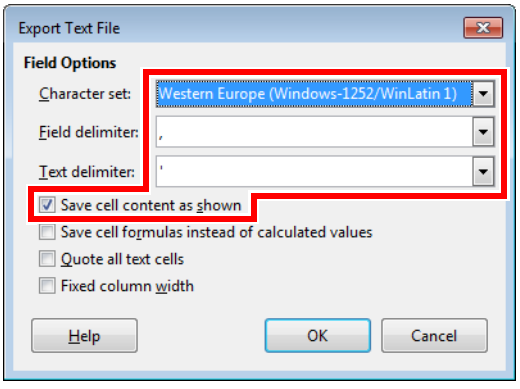
6. Define the registers and bits:

CSV.ON							
AUTOINDENT.ON CENTER TREE							
PERCMD	**Address**	**AccessWidth**	**Name**	**Tooltip**	**From**	**To**	**Choices**
BASE x:0							
TREE "ADCA (Analog-to-Digital Converter A)"							
TREE "Common Registers"							
A →	0xe80	16.	ADCR1	ADC Control Register 1			
			STOP	Stop	14.	14.	Normal operation,Stop
			START	Start Conversion	13.	13.	No action,Start
			SYNC	Sync Select	12.	12.	Initiated by START bit
NEWLINE							
			EOSIE	End-of-Scan Interrupt Enable	11.	11.	Disabled,Enabled
			ZCIE	Zero Crossing Interrupt Enable	10.	10.	Disabled,Enabled
			LLMTIE	Low Limit Interrupt Enable	9.	9.	Disabled,Enabled
NEWLINE							
			HLMTIE	High Limit Interrupt Enable	8.	8.	Disabled,Enabled
			CHNCFG[3]	Channel Configure AN6-AN7	7.	7.	Single ended AN6-AN
			CHNCFG[2]	Channel Configure AN4-AN5	6.	6.	Single ended AN4-AN
NEWLINE							
			CHNCFG[1]	Channel Configure AN2-AN3	5.	5.	Single ended AN2-AN
B →			CHNCFG[0]	Channel Configure AN0-AN1	4.	4.	Single ended AN0-AN
			SMODE	ADC Mode Control	0.	2.	Once sequential,Once
	0xe82	16.	ADCR2	ADC Control Register 2			
C →			DIV	Clock Divisor Select	0.	3.	
TREE.END							
TREE.END							

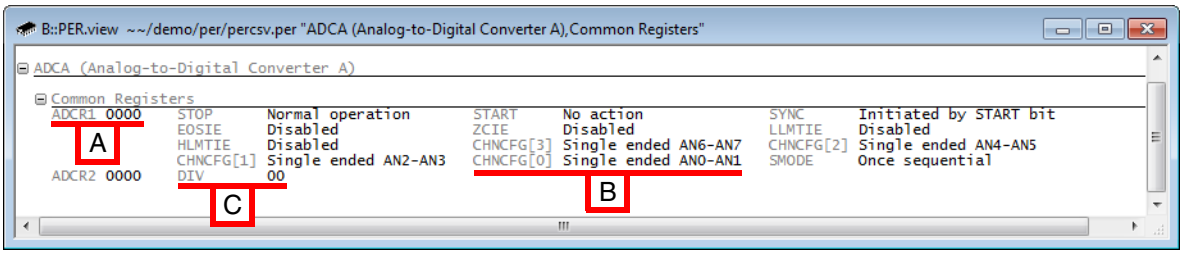
A to **C** For a description, see **Rules** below.

7. When done, save/export the spreadsheet in CSV format as shown below:





Output:



A to **C** For a description, see **Rules** below.

Rules:

- A new register [**A**] will be created if at least one of the following conditions applies:
  - The **Address** value is the first non-empty entry in the spreadsheet.
  - The **Address** value differs from the previous one.
  - The **AccessWidth** value differs from the previous one.
  - **From** and **To** values are empty.
- A new customized bit description [**B**] will be created if the following conditions are all true:
  - The **Address** value does not change, or the entry is empty.
  - The **AccessWidth** value does not change, or the entry is empty.
  - The **Choices** value is not empty.
- A new bit or bit range [**C**] is displayed as hexadecimal if the following conditions are all true:
  - The **Address** value does not change, or the entry is empty.
  - The **AccessWidth** value does not change, or the entry is empty.
  - The **Choices** value is empty.

In order to simplify matters, peripheral files in CSV format do not offer the full functional range of regular *.per files. However, you can easily include regular *per commands in the first column:

Excerpt from ~~/demo/per/percsv_mixed.per:

PERCMD	Address	AccessWidth	Name	Tooltip	From	To	Choices
TREE "Common Registers"							
	0xe80	16.	ADCR1	ADC Control Register 1			
IF (Data.Long(vm:0xe80)==0x0)							
			STOP	Stop	14.	14.	Normal operation,Stop
			START	Start Conversion	13.	13.	No action,Start
			SYNC	Sync Select	12.	12.	START bit,sync input or START bit
ELSE							
			EOSIE	End-of-Scan Interrupt Enable	11.	11.	Disabled,Enabled
			ZCIE	Zero Crossing Interrupt Enable	10.	10.	Disabled,Enabled
			LLMTIE	Low Limit Interrupt Enable	9.	9.	Disabled,Enabled
ENDIF							
NEWLINE							
			HLMTIE	High Limit Interrupt Enable	8.	8.	Disabled,Enabled
			CHNCFG[3]	Channel Configure AN6-AN7	7.	7.	Single ended AN6-AN7,AN6 + and AN7 -
			CHNCFG[2]	Channel Configure AN4-AN5	6.	6.	Single ended AN4-AN5,AN4 + and AN5 -
	0xf80	16.	ADCR2	ADC Control Register 2			
			DIV	Clock Divisor Select	0.	3.	
TREE.END							

In above example we utilize the regular peripheral commands **TREE**, **IF** and **NEWLINE**. In all other cases, the first column must remain empty!

Manual Peripheral File Generation

To start writing the peripheral file, please create a file with extension *.per.
“.per” is the TRACE32 standard extension for peripheral files.

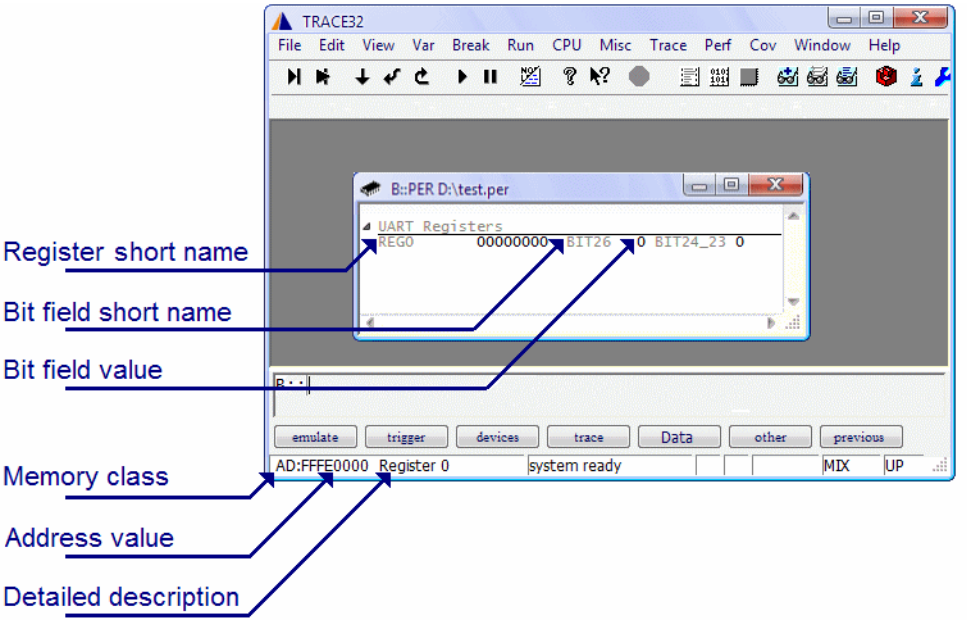
The syntax of a peripheral file is line oriented. Blanks and empty lines can be inserted to define the structure of the program. Comment lines start with semicolon.

Examples of the peripheral file reside in the directory **~~/demo/per**.

At the beginning of the file, the commands **WIDTH** and **CONFIG** should be placed. The next step is to define the base address using **BASE** command. Each implemented module has to be started with **TREE** command and ended with the **TREE.END** command.

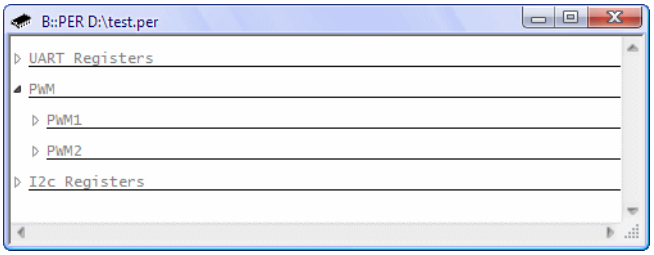
A typical peripheral file implementation is showed below:

```
; "dots" mean decimal format
CONFIG 16. 8.

; 0x means hex format
WIDTH 0xb

; "Treeview" of the module
TREE "Module Registers"

; base address of the module
BASE ad:0xf0000000

; GROUP definition
GROUP.LONG 0x00++0x3
  ; register definition
  LINE.LONG 0x00 "REG0,Register 0"

     ; one bit filed definition
     BITFLD.LONG 0x00 26. " BIT26 ,Bit 26" "0,1"

     ; 2-bit field definition
     BITFLD.LONG 0x00 23.--24. " BIT24_23 ,Bits 24 to 23" "0,1,2,3"

; end of the tree
TREE.END
```

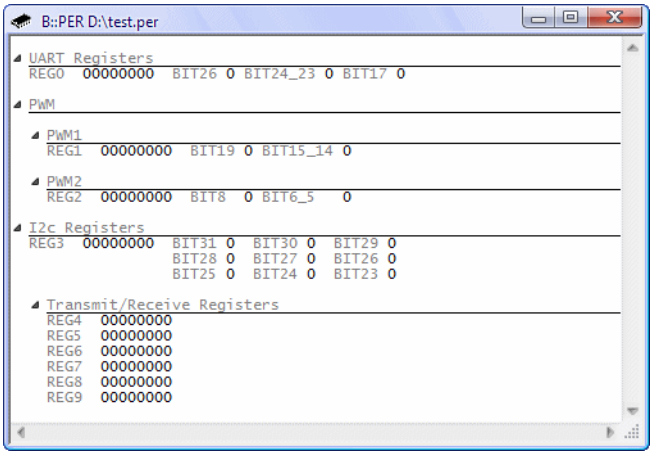

TRACE32
File Edit View Var Break Run CPU Misc Trace Perf Cov Window Help
B::PER D:\test.per
UART Registers
REG0 00000000 BIT26 0 BIT24_23 0
emulate trigger devices trace Data other previous
AD:FFFE0000 Register 0
system ready
MIX
UP
Register short name
Bit field short name
Bit field value
Memory class
Address value
Detailed description

```
TREE "UART Registers"
  BASE ad:0xfffe0000
  GROUP.LONG 0x00++0x3
       LINE.LONG 0x00 "REG0,Register 0"
            BITFLD.LONG 0x00 26. " BIT26 ,Bit 26" "0,1"
            BITFLD.LONG 0x00 23.--24. " BIT24_23 ,Bits 24 to 23" "0,1,2,3"
            BITFLD.LONG 0x00 26. " BIT17 ,Bit 17" "0,1"
TREE.END

TREE.OPEN "PWM"
  TREE "PWM1"
       BASE ad:0xfffe1000
       GROUP.LONG 0x00++0x3
            LINE.LONG 0x00 "REG1,Register 1"
                 BITFLD.LONG 0x00 19. " BIT19 ,Bit 19" "0,1"
                 BITFLD.LONG 0x00 14.--15. " BIT15_14 ,Bits 15 to 14" "0,1,2,3"
  TREE.END
  TREE "PWM2"
       BASE ad:0xfffe2000
       GROUP.LONG 0x00++0x3
            LINE.LONG 0x00 "REG2,Register 2"
                 BITFLD.LONG 0x00 8. " BIT8  ,Bit 8" "0,1"
                 BITFLD.LONG 0x00 5.--6. " BIT6_5   ,Bits 6 to 5" "0,1,2,3"
  TREE.END
TREE.END
TREE "I2c Registers"
  BASE ad:0xfffe3000
  GROUP.LONG 0x00++0x3
       LINE.LONG 0x00 "REG3,Register 3"
            BITFLD.LONG 0x00 31. " BIT31 ,Bit 31" "0,1"
            BITFLD.LONG 0x00 30. "  BIT30 ,Bit 30" "0,1"
            BITFLD.LONG 0x00 29. "  BIT29 ,Bit 29" "0,1"
            TEXTLINE "                "
            BITFLD.LONG 0x00 28. " BIT28 ,Bit 28" "0,1"
            BITFLD.LONG 0x00 27. "  BIT27 ,Bit 27" "0,1"
            BITFLD.LONG 0x00 26. "  BIT26 ,Bit 26" "0,1"
            TEXTLINE "                "
            BITFLD.LONG 0x00 25. " BIT25 ,Bit 25" "0,1"
            BITFLD.LONG 0x00 24. "  BIT24 ,Bit 24" "0,1"
            BITFLD.LONG 0x00 23. "  BIT23 ,Bit 23" "0,1"
  TREE "Transmit/Receive Registers"
       GROUP.LONG 0x10++0x17
            LINE.LONG 0x00 "REG4,Register 4"
            LINE.LONG 0x04 "REG5,Register 5"
            LINE.LONG 0x08 "REG6,Register 6"
            LINE.LONG 0x0c "REG7,Register 7"
            LINE.LONG 0x10 "REG8,Register 8"
            LINE.LONG 0x14 "REG9,Register 9"
  TREE.END
TREE.END
```

Peripheral modules are organized in a tree structure.



Contents of peripheral modules is also organized in a tree structure.



GROUP Commands

The **GROUP** commands describe how data is basically read or written to/from memory.

Format:	**GROUP.***<size>* *<datagrp>*|*<fifogroup>* ["*<name>*"]
<datagrp>:	*<address>***++***<number_of_read_bytes>***-1** or *<start_address>***--***<end_address>*
<fifogroup>:	*<address>* *<address_range>*

The GROUP commands control the debugger access to the target memory.

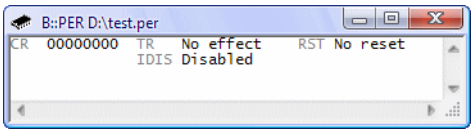
<size>	Size of registers (**Byte**, **Word**, **TByte**, **Long**, **Quad**) or **auto**.
<datagrp>	Maximum size 4 kB (4096 bytes).
<name>	Optional text.

If a name is given, the GROUP is separated from the previous lines and the name is used as headline in the per window. Using numerical values (without memory access class) in address parameter, the address is calculated by the entered value plus the base address (defined by the last **BASE** command). The GROUP can either use normal memory access or fifo access (reads all bytes from the same address). The whole address range of the GROUP command is read at once. Reading from reserved address range may cause a bus error.

Example 1:

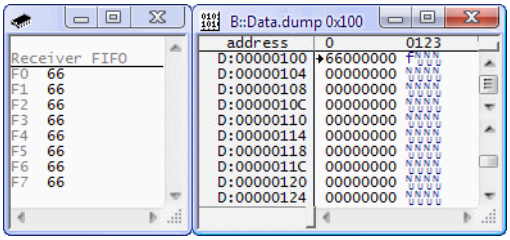
```
BASE ud:0x200                     ;data bytes at address sd:0x100--0x101
GROUP sd:0x100--0x101 "PortA"


GROUP 0x50--0x51                  ;data bytes at address ud:0x250--0x251


GROUP.LONG sd:0x60--0x6f          ;read memory with 32-bit access


GROUP sd:0x300 0x10               ;fifo at location sd:0x300, 16 bytes
                                  ;deep
GROUP 0x10 0x4                    ;fifo at ud:0x210, 4 bytes deep
```

```
BASE ad:0x00000000
GROUP 0x00++0x03
  LINE.LONG 0x00 "CR,Control Register"
       BITFLD.LONG 0x00 24. " TR   ,Transfer" "No effect,Transferred"
       BITFLD.LONG 0x00 5. "  RST ,Software Reset" "No reset,Reset"
       TEXTLINE "             "
       BITFLD.LONG 0x00 1. " IDIS ,Interrupt Enable" "Disabled,Enabled"
```



Example 2:

```
BASE ad:0x00000000
GROUP.BYTE 0x100 0x8 "Receiver FIFO"
  LINE.BYTE 0x0 "F0,FIFO position 0"
  LINE.BYTE 0x1 "F1,FIFO position 1"
  LINE.BYTE 0x2 "F2,FIFO position 2"
  LINE.BYTE 0x3 "F3,FIFO position 3"
  LINE.BYTE 0x4 "F4,FIFO position 4"
  LINE.BYTE 0x5 "F5,FIFO position 5"
  LINE.BYTE 0x6 "F6,FIFO position 6"
  LINE.BYTE 0x7 "F7,FIFO position 7"
```



HGROUP — Define read-once/write GROUP

Format:	**HGROUP.***<size>* *<datagrp>*|*<fifogroup>*["*<name>*"]
<datagrp>:	*<address>***++***<number_of_read_bytes>***-1**> or *<start_address>***--***<end_address>*
<fifogroup>:	*<address>* *<address_range>*

Similar to GROUP, but this definition is useful for ports which are cleared by a read access. Refer to the **GROUP** command description. HGROUP command prevents target memory from the periodic read access and is useful for 'write-only' ports. In hidden GROUPs only hidden elements e.g. **HIDE** command should be used.

<size>	Size of registers (byte, word, tbyte, long, quad).
<datagrp>	Maximum size 4 kB (4096 bytes).
<name>	Optional text.

Format:	**RGROUP.***<size>* *<datagrp>*|*<fifogroup>* ["*<name>*"]
<datagrp>:	*<address>***++***<number_of_read_bytes>***-1** or *<start_address>***--***<end_address>*
<fifogroup>:	*<address>* *<address_range>*

Similar to GROUP, but this definition is useful for 'read-only' ports. Refer to the **GROUP** command description.

<size>	Size of registers (**Byte**, **Word**, **TByte**, **Long**, **Quad**).
<datagrp>	Maximum size 4 kB (4096 bytes).
<name>	Optional text.

WSGROUP Define write-only and shadow GROUP

Format:	**WSGROUP.***<size>* *<wr_acc_addr>* *<rd_acc_addr>*

WSGROUP is a specific GROUP command, which forces the debugger to access different registers for read and for write accesses. It is only useful, if the core has write-only registers and their contents are duplicated in shadow registers, which are read- and writable.

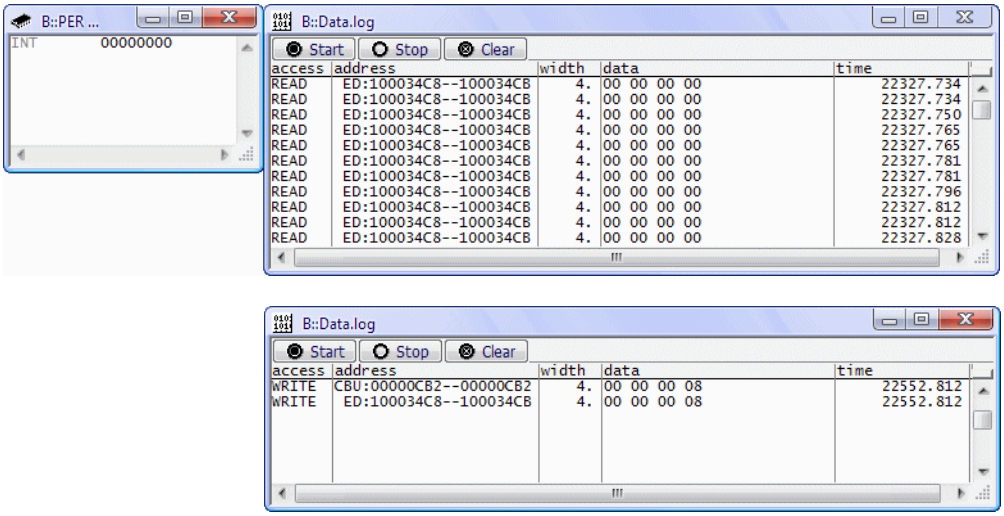
<size>	Size of registers (byte, word, tbyte, long, quad).
<wr_acc_addr>	Address of the register where data is to be written into.
<rd_acc_addr>	Address of the register where data is to be read from.

Read-/write accesses have following effects:

- **write access**: Data is written to write-only registers (dataGROUP) as well as to the shadow registers.
- **read access**: Data is read from the shadow registers.

Example:

```
WSGROUP.LONG (ecbu:0x0CB2)++0 (ed:0x100034C8)
  LINE.LONG 0x0 "INT,Self-interrupt register"
```



WGROUP

Define write-only GROUP

Format:	**WGROUP.***<size>* *<datagrp>*|*<fifogroup>* ["*<name>*"] [**/SET**|**/CLEAR**]
<datagrp>:	*<address>***++***<number_of_read_bytes>***-1** > or *<start_address>***--***<end_address>*
<fifogroup>:	*<address> <address_range>*

Similar to GROUP command. This definition is useful for 'write-only' ports. The current state of the port is held in the emulation memory (must be mapped at this location). Refer to the **GROUP** command description.

<size>	Size of registers (byte, word, tbyte, long, quad).
<datagrp>	Maximum size 4 kB (4096 bytes).
<name>	Optional text.
/SET	Only has an effect if **WGROUP** contains a **BITFLD** command. All bits outside the **BITFLD** range will be set to '1' on a write access.
/CLEAR	Only has an effect if **WGROUP** contains a **BITFLD** command. All bits outside the **BITFLD** range will be set to '0' on a write access.

Example:

```
WGROUP sd:0x50--0x51          ;the port at address sd:0x50--0x51
                              ;is a write-only port (e.g. 74xx374)
                              ;but the state can be read via
                              ;dual-port access
```

SGROUP — Define sequence GROUP

Format: **SGROUP** ["*<name>*"]

Sequence of memory accesses done to get/set the data.

<name> Optional text.

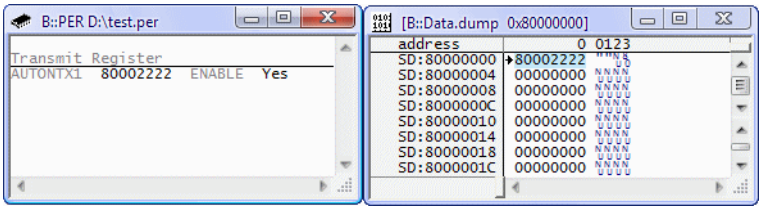
Usually GROUP commands specify the target memory accesses and the following commands e.g. BITFLD, HEXMASK, etc. define how the data are displayed in the per window.
With SGROUP data is not accessed with **SGROUP** itself, but by a sequence of special commands, which transfer data from memory to the “SGROUP data buffer” or from the “SGROUP data buffer” back to memory. The size of the buffer is 256 bytes.
Afterwards this sequence of special commands the data in the buffer can be displayed by following commands e.g. BITFLD, HEXMASK.

To read/write data from/to memory to/from SGROUP buffer you can use the following commands (which are only allowed in **SGROUP**s):

Command	Function
SET *<address>* %*<format>* *<value>*	Constant value **-->** memory(address)
SETX *<address>* %*<format>* *<index>*	Buffer(index) **-->** memory(address)
GETX *<address>* %*<format>* *<index>*	Memory(address) **-->** buffer(index)
CONSTX *<index>* %*<format>* *<value>*	Constant value **-->** buffer(index)
VARX *<index>* %*<format>* *<expression>*	Variable value **-->** buffer(index)
WRITEBACK	Separate write part from a read part

Example:

```
SGROUP "Transmit Register"              ; define sequence GROUP
GETX d:0x80000000 %l 0                  ; read data at 0x80000000 and store
                                        ; them in buffer + offset 0
WRITEBACK                               ; next commands only done for
CONSTX 2 %w 0x2222                      ; per.set
                                        ; write 0x2222 to buffer + offset 2
SETX d:0x80000000 %l 0                  ; write data from buffer + offset 0
                                        ; to memory at 0x80000000
LINE.LONG 0x0                           ; display AUTONTX1 register with
"AUTONTX1,Autonegotiation Next          ; contents of buffer[0…3]
Page Transmit Register 1"
BITFLD.LONG 0 31. "ENABLE" "No,Yes"     ; define bit "Enable"
```



SET — Write constant value to memory

Format:	**SET** *<address>* **%***<format>* *<value>*

SET command writes data to memory.

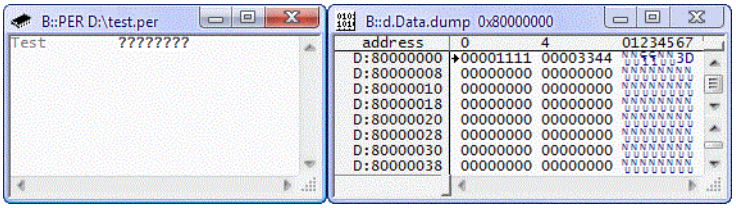
The given value is written to the target memory at the specified address or at the base address with added offset. The specified value is written continuously.

<address>	Target address.
<format>	Defines specific format (**Byte**, **Word**, **TByte**, **Long**, **Quad**, **LE**, **BE**).
<value>	Constant value. The value may be a hexadecimal o mask or binary mask. (E.g.: `0yxxxx10xx`)

Command is only allowed in **SGROUP**.

Example:

```
BASE d:0x80000000                       ; set base address to d:0x80000000
SGROUP                                  ; define sequence GROUP
SET d:0x80000000 %l 0x1111              ; write 0x1111 to d:80000000
SET 4 %l 0x3344                         ; write 0x3344 to base address
                                        ; (d:80000000) + offset 4
LINE.LONG 0x0 "Test,Test Register"
```



SETX — Write SGROUP buffer to memory

Format:	**SETX** *<address>* **%***<format>* *<index>*

SETX command writes a buffered value to the memory.

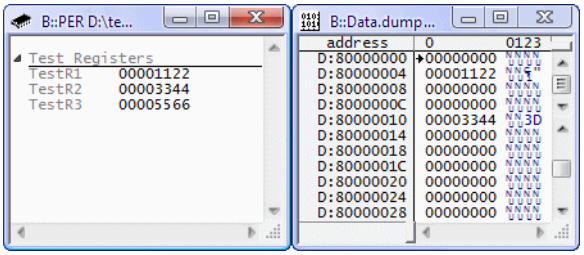
A value stored in a buffer at the given buffer offset is written to the target memory at the specified address or base address with added offset. The value is written only once.

<address>	Target address.
<format>	Defines specific format (**Byte**, **Word**, **TByte**, **Long**, **Quad**).
<index>	Constant value.

Command is only allowed in **SGROUP**.

Example:.

```
CONFIG 16. 8.
WIDTH  10.
BASE 0x80000000
TREE "Test Registers"
;write into buffer : 0x1122 at offet [0], 0x3344 at [4], 0x5566 at [8]
SGROUP

CONSTX 0 %l 0x1122
CONSTX 4 %l 0x3344
CONSTX 8 %l 0x5566
   LINE.LONG 0x0 "TestR1,Test Register 1"
   LINE.LONG 0x4 "TestR2,Test Register 2"
   LINE.LONG 0x8 "TestR3,Test Register 3"

;write buffer contents into target memory : [0..3] at 0x80000004,...
SETX 4    %l 0
SETX 0x10 %l 4
TREE.END
```



GETX — Read from memory to the SGROUP buffer

Format:	**GETX** *<address>* **%***<format>* *<index>*

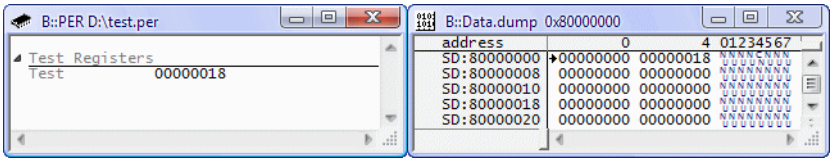
GETX command reads data from the memory and puts it to the buffer. The memory contents from the given address is read using specified access width format. The read data is stored in a buffer at the defined offset.

<address>	Target address equals base address + offset.
<format>	Defines specific format (**Byte**, **Word**, **TByte**, **Long**, **Quad**).
<index>	Defines buffer number.

Command is only allowed in **SGROUP**.

Example:

```
BASE d:0x80000000
TREE "Test Registers"
SGROUP                                  ; define sequence GROUP
SET d:0x80000004 %l 0x18                ; write value 0x18 to target memory
                                        ; at d:80000004
GETX 4 %l 0                             ; read out target memory at base
                                        ; address d:80000000+offset 4 and
                                        ; store it at buffer+offset 0
LINE.LONG 0x0 "Test,Test Register"      ; display data of buffer[0…3]
TREE.END
```



CONSTX

Write constant value to the SGROUP buffer

Format:	**CONSTX** *<index>* **%***<format>* *<value>*

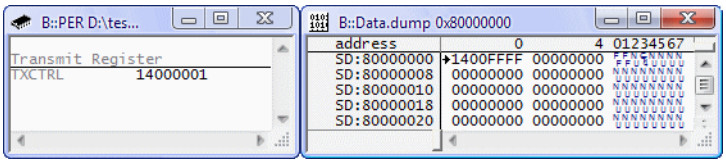
CONSTX command writes a constant value to the buffer. This data is **not** written to the target memory. The data can be displayed with a following line command.

<index>	Defines indexed offset.
<format>	Defines specific format (**Byte**, **Word**, **TByte**, **Long**, **Quad**, **LE**, **BE**).
<value>	Defines a constant value. The value may be a hexadecimal or mask or binary mask. (E.g.: `0yxxxx10xx`)

Command is only allowed in **SGROUP**.

Example:

```
SGROUP "Transmit Register"          ; define sequence GROUP
SET 0x80000000 %l 0x1400ffff        ; write value 1400ffff to target
                                    ; memory at d:80000000
GETX  d:0x80000000 %l 0x00          ; read out target memory at 80000000
                                    ; and store it at buffer + offset 0
CONSTX 2 %w 0x1                     ; write 0x0001 at buffer + offset 2
LINE.LONG 0x0 "TXCTRL,Transmit      ; display data of buffer[0…3]
Control Register"
```



VARX — Write expression to SGROUP buffer

Format: **VARX** *<index>* **%***<format>* *<expression>*

VARX command writes a variable value to the SGROUP buffer. This data is **not** written to the target memory. The data can be displayed with a following line command.

<index>	Defines indexed offset.
<format>	Defines specific format (**Byte**, **Word**, **TByte**, **Long**, **Quad**, **LE**, **BE**).
<expression>	Defines a PRACTICE expression. The expression will be parsed whenever the **PER** window updates and its result will be assigned to the **SGROUP** buffer

The **VARX** command is very similar to the **CONSTX** command. However the value, which should be assigned to the **SGROUP** buffer may be based on PRACTICE functions, whose values may change during the display of the **PER** window.
With **VARX** you can modify the **SGROUP** buffer in any way you like by using the following PRACTICE functions, which access the **SGROUP** buffer:

PER.Buffer.Byte(*<index>***)** **PER.B.B(***<index>***)**	Returns a byte at position *<index>* from the SGROUP buffer.
PER.Buffer.Word(*<index>***)** **PER.B.W(***<index>***)**	Returns a 16 bit word at position *<index>* from the SGROUP buffer.

PER.Buffer.Long(<*index*>**)**
PER.B.L(<*index*>**)**
Returns a 32 bit word at position <*index*> from the SGROUP buffer.

PER.Buffer.Quad(<*index*>**)**
PER.B.Q(<*index*>**)**
Returns a 64 bit at position <*index*> from the SGROUP buffer.

Due to performance reasons you should use **VARX** only, if there is no other solution possible.

Command is only allowed in **SGROUP**.

Example:

```
SGROUP "Dummy Counter"                          ; begin Sequence-GROUP
  varx 0 %quad os.timer()                       ; read timer from OS
  varx 9 %q (PER.B.Q(0)/1000.)                  ; define quad data from
                                                  SGROUP buffer at index 0 by
                                                  1000 and store the result
                                                  at index 9 as quad
  textline ""                                   ; display data at index 0 as
  decmask.quad 0 0--63. 1 "  milliseconds:"      decimal
  textline ""                                   ; display data at index 9 as
  decmask.quad 9 0--63. 1 "  seconds:     "      decimal
  textline ""                                   ; Newline
```

WRITEBACK

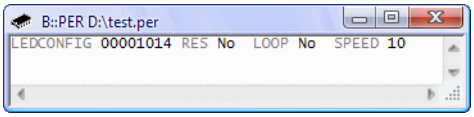
Separate write a part from a read part

Format:	**WRITEBACK**

Separates the write part of a sequence from the read part. Command is only allowed in **SGROUP**.

Example 1:

```
SGROUP
  SET 0 %l 0x1014
  GETX 0 %l 0
  WRITEBACK
  CONSTX 2 %w 0x2014
  SETX 0 %l 0
  LINE.LONG 0x0 "LEDCONFIG,LED Configuration Register (20)"
       BITFLD.LONG 0x0 31. "RES ,Reset" "No,Yes"
       BITFLD.LONG 0x0 30. " LOOP ,Loopback" "No,Yes"
       BITFLD.LONG 0x0 29. " SPEED ,Speed" "10,100"
```

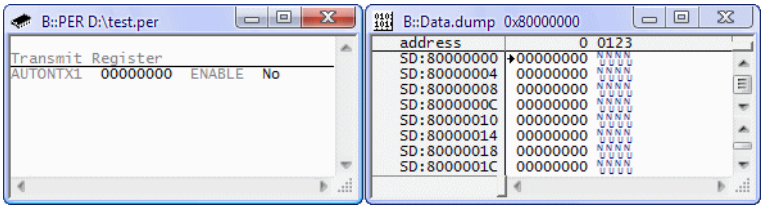


The commands after write back are executed only if **PER.Set** command is used. For displaying the data in the PER-window these commands are ignored.

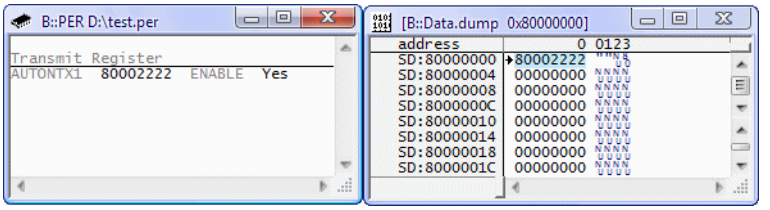
Example 2:

```
SGROUP "Transmit Register"          ; define sequence GROUP
GETX d:0x80000000 %l 0              ; read data at 0x80000000 and store
                                    ; them in buffer + offset 0
WRITEBACK                           ; next commands only executed, if a
                                    ; write access is done in per-window
CONSTX 2 %w 0x2222                  ; write 0x2222 to buffer + offset 2
SETX d:0x80000000 %l 0              ; write data from buffer + offset 0
                                    ; to memory at 0x80000000
LINE.LONG 0x0 "AUTX1,Transmit Reg." ; display AUTX1 register with
BITFLD.LONG 0 31. "ENABLE " "No,Yes"; contents of buffer[0…3]
                                    ; if bit 31 is changed/written
                                    ; constx and setx are done
```

Opening the per-window results in displaying data from memory.



Changing state of the ENABLE bit results also in writing constant value 0x2222 to the register.



ASSERT — Abort if condition not met

Format: **ASSERT** *<expression>* [*<string>*]

With **ASSERT** you can ensure that your environment meets a certain condition, before TRACE32 should go on with the parsing of the PER file.

If you omit the optional string with an error message, the following message will be shown instead:

```
Assertion failed: <expression>
```

<expression>	Expression which must evaluate to a boolean. If the result of the expression is FALSE, the parsing of the PER file will be stopped and an error message will be shown.
<string>	Optional string containing an error message, which will be shown if *<expression>* evaluates to FALSE.

Example: This code line ensures that a PER file is only parsed by "TRACE32 for ARM"

```
ASSERT CPUFAMILY()=="ARM" "Sorry, this PER file is only for ARM cores"
```

AUTOINDENT — Indent content of peripheral file automatically

[Examples]

Format: **AUTOINDENT.***<command>* *<alignment>* *<type>* [*<number>* | *<columns>* *<width>*]

<command>	**ON** | **OFF** | **PUSH** | **POP**
<alignment>:	**LEFT** | **RIGHT** | **CENTER**
<type>:	**TREE** | **LINE** | **GRID**

Default: OFF

Switches automatic indentation **ON** or **OFF**. Only available for TRACE32 versions >= 97444.

AUTOINDENT ignores all leading and trailing space characters within subsequent definitions and rearranges the contents according to the specified *<alignment>* and *<type>*. It affects all entries within a **TREE** and should therefore only be activated or changed outside of a **TREE**. Otherwise the result may be undefined.

PUSH	Pushes current AutoIndent configuration on the stack.
POP	Recovers previously pushed AutoIndent configuration from the stack.
<alignment>	Alignment of the values in relation to their description: **LEFT**, **RIGHT**, **CENTER**. Default: **LEFT** For examples, see here.
<type>	Indentation type of description-value pairs: **TREE**, **LINE**, **GRID**. Default: **TREE** For examples, see here.
<number>	Proximity range. Only available if *<type>* = **PROXIMITY**. Default: 5
<columns>	Number of columns. Only available if *<type>* = **GRID**. Default: 5
<width>	Width of a column in characters. Only available if *<type>* = **GRID**. Default: 16.

NOTE: **AUTOINDENT** affects only the following statements:

- **ASCII**
- **BITFLD**, **EVENTFLD**, **RBITFLD**, **SETCLRFLD**
- **BUTTON**
- **DECMASK**, **FLOATMASK**, **HEXMASK**
- **HEXFLD**
- **HIDE**
- **IN**
- **LINE**
- **NEWLINE**

It explicitly does not affect the following statements:

- **BIT**
- **TEXTFLD**, **TEXTLINE**

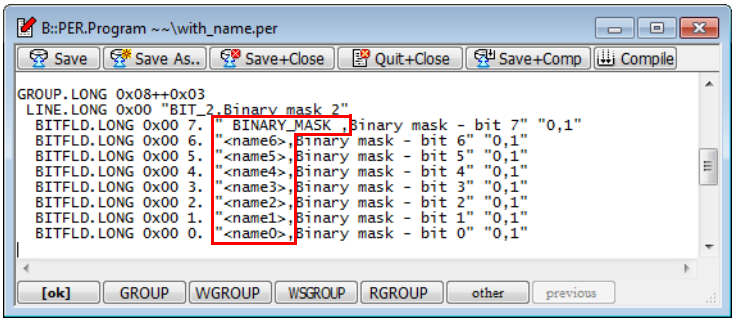
It makes the following statements obsolete:

- **WIDTH**
- **CONFIG** (If no BIT command is being used)

Sometimes you may want to concatenate bits or include text fragments *without switching auto-indentation OFF*. To override auto-indentation in this special case, omit the *<name>* entry of the **HEXMASK** or **BITFLD**.

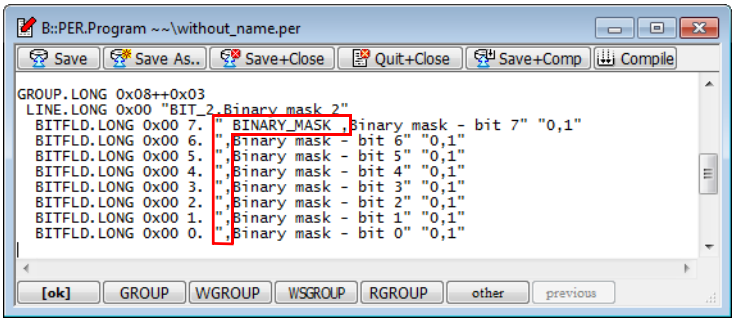
Let's illustrate the override effect by comparing two source code snippets, one with *<name>* and the other one without *<name>*. The relevant part in each source code snippet is highlighted in red in the two **PER.Program** windows. The results are displayed directly below in the two **PER.view** windows.

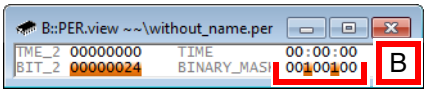
With <name>:



A The bits are not concatenated if a *<name>* is specified in **BITFLD**.

Without <name>:





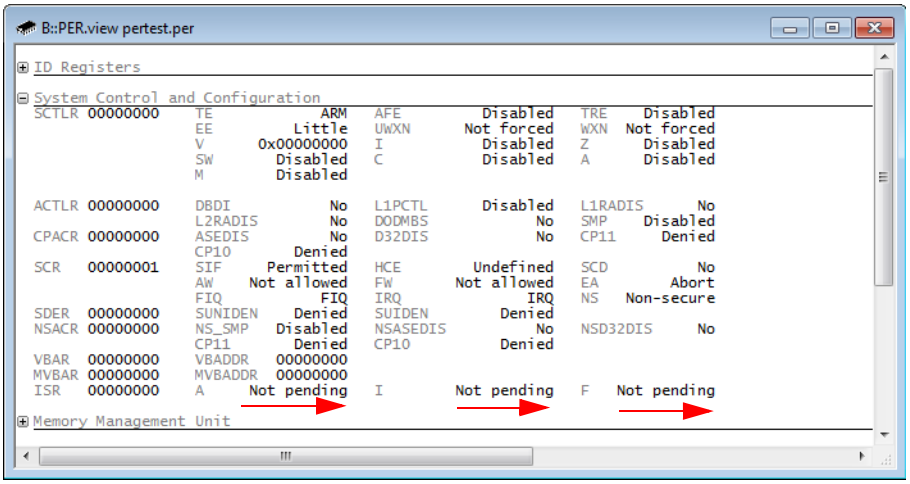
B If *<name>* is omitted from **BITFLD**, then the bits are concatenated.

Code Example

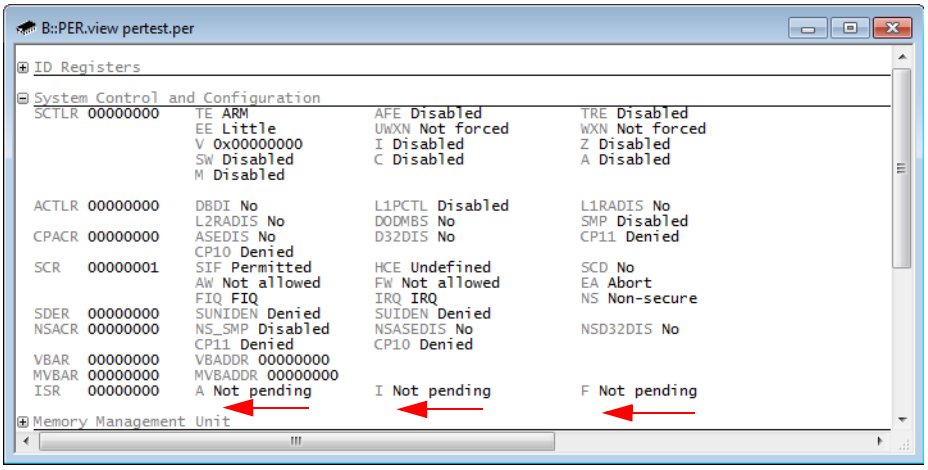
```
ASSERT version.build()>=97444. "Please update TRACE32"

AUTOINDENT.ON left tree                          ; AUTOINDENT using
TREE "Tree 1"                                    ; <alignment> = left and
  GROUP.LONG ...                                 ; <type> = tree
  LINE.LONG 0, "Reg1,First register"
    BITFLD.LONG 0, 0.--1. "Fld1,First field" "1,2,3,4"
  ...
TREE.END
AUTOINDENT.ON right tree                         ; Second tree looks
TREE "Tree 2"                                    ; better with
  GROUP.LONG ...                                 ; <alignment> = right
  LINE.LONG 0, "Reg32,32nd register"
    BITFLD.LONG 0, 0.--1. "Fld1,First field" "1,2,3,4"
  ...
TREE.END
AUTOINDENT.OFF                                   ; Sometimes you do
TREE "Tree 3"                                    ; not want to use
  GROUP.LONG...                                  ; AUTOINDENT
  LINE.LONG 0, "   Reg99   ,99th register"
    BITFLD.LONG 0, 0.--1. "  Fld1  ,First field" "1,2,3,4"
  ...
```

<alignment> Examples

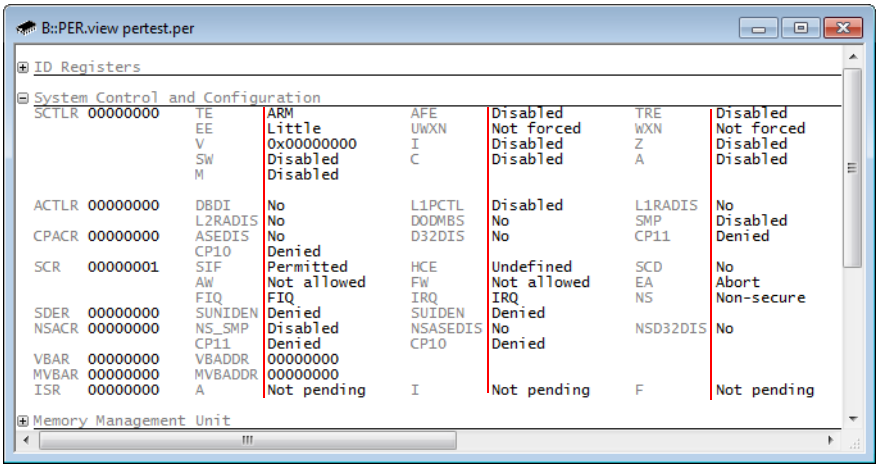
Example 1: AUTOINDENT.ON RIGHT TREE aligns all values to the right.



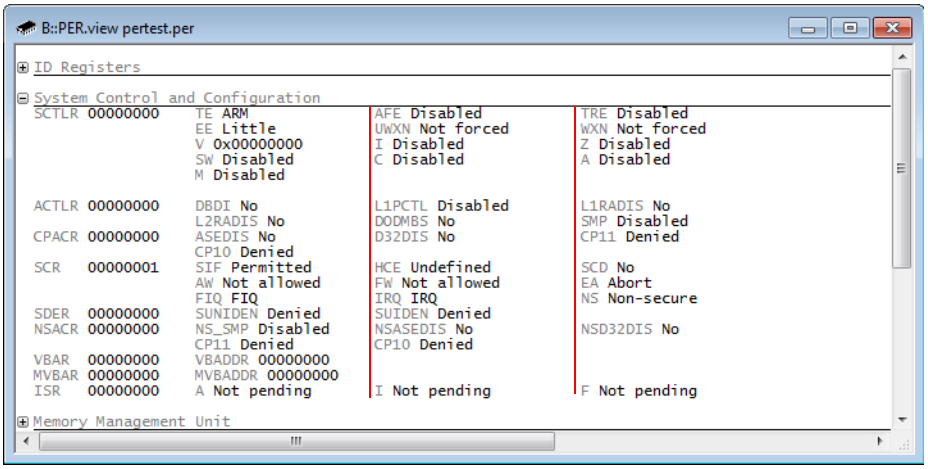
Example 2: AUTOINDENT.ON LEFT TREE aligns all values to the left next to their descriptions.



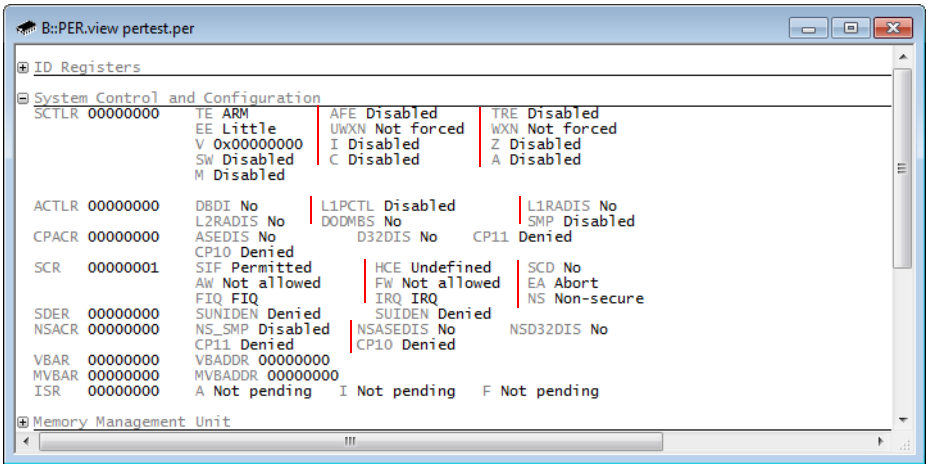
Example 3: AUTOINDENT.ON CENTER TREE moves the values somewhere to the middle so they are aligned.



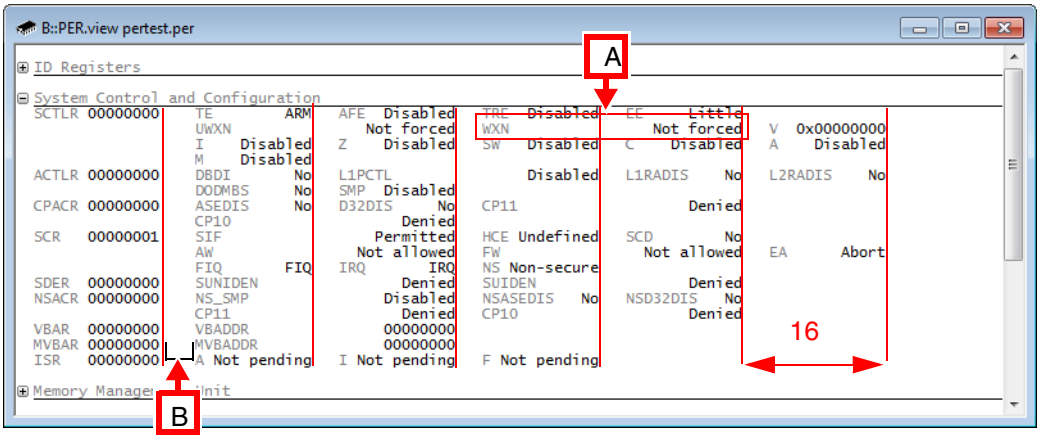
Example 1: AUTOINDENT.ON LEFT TREE aligns all description-value pairs within a **TREE**.



Example 2: AUTOINDENT.ON LEFT LINE aligns all description-value pairs within a **LINE**.



Example 4: AUTOINDENT.ON RIGHT GRID 5 16. divides the window into the given number of *<columns>* which are *<width>* characters wide each.



A In case a description-value pair does not fit within a column, two (or more) columns will be merged -> see red box above.

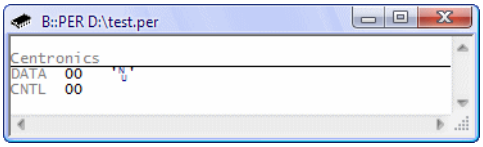
B When defining the *<width>* of the columns, please take the first 3 separation characters into account.

This command is useful for peripheral files which have been generated automatically and do not contain any **NEWLINE** statements. These will be added automatically if a **LINE** contains more than *<columns>* subentries. **NEWLINE** statements, however, can still be added manually.

Format:	**BASE** *<address>*|*<offset>*

This command sets the start address for the peripheral module and refers to simple offset ranges. This expression is permanently recalculated. If the parameters contain functions or symbols, it reflects later changes in the parameters. The BASE command specifies memory class which is responsible for setting appropriate addressing mode. Memory classes are described in **Memory Classes** section.

<address>	Fixed address or expression which evaluates to the start address of the peripheral groups following the **BASE** command.
<offset>	Fixed offset or expression which evaluates to the start address of the peripheral groups following the **BASE** command. The access class will be taken from the <address> of a preceding **BASE** command.



Example:

```
// use fixed base
BASE d:0xffff0000
  GROUP.LONG 0x00++0x3
       LINE.LONG 0x00 "Reg_0,Register 0"

// use offset base
BASE 0x100

// use variable base
BASE (SYStem.BASE()&0x0f)*0x1000

// use variable base
BASE Data.Long(base_pointer)
```

Format:	**BASEOUT** *<addr_expr>* *<address>* [**%***<format>*] *<data>*
<format>:	**Byte** | **Word** | **Long** | **Quad** | **TByte** | **HByte** **Float.** [**Ieee** | **IeeeDbl** | **IeeeeXt** | *<others>*] **BE** | **LE**

Like the **BASE** command **BASEOUT** defines a start address for the peripheral group definitions following the **BASEOUT** command. This address is usually frequently calculated by the given address expression.

Unlike the **BASE** command **BASEOUT** writes a certain value (*<data>*) to a specified address (*<address>*) before evaluating the expression which sets the start address for the following group definitions. If a bit-mask is used the specified address will be read and modified before it will be written.

NOTE: If *<addr_expr>* is a constant address, no data will be written to *<address>*.

<addr_expr>	Expression which evaluates to the start address of the peripheral groups following the **BASEOUT** command.
<address>	Address which should be written before evaluating the address expression.
<data>	Data which should be send to the specified address before evaluating the address expression. This could also be a bit-mask.

Please consider: As the display is refreshed permanently the memory at *<address>* is modified permanently as well.

Example 1: Write 0x01 to address 0x100 before reading the base address from address 0x104.
The GROUP command will then read the first three lines at that base address.

```
BASEOUT Data.Long(D:0x104) D:0x100 %Long 0x01
GROUP 0x00++0x3
       LINE.LONG 0x00 "Reg_0,Register 0"
```

Example 2: Set the LSB in address 0x200 before reading the base address from 0x202.

```
BASEOUT Data.Word(D:0x202) D:0x200 %Word 0yXXXXXXXXXXXXXXX1
GROUP 0x00++0x3
       LINE.WORD 0x00 "TIMER_CTRL_0,Timer 0 Control register"
```

Format:	**BASESAVEOUT** *<addr_expr>* *<address>* [**%***<format>*] *<data>*
<format>:	**Byte** | **Word** | **Long** | **Quad** | **TByte** | **HByte** **Float.** [**Ieee** | **IeeeDbl** | **IeeeeXt** | *<others>*] **BE** | **LE**

Outputs a value before calculating a base address with restore. This command is almost the same like **BASEOUT**. However, unlike **BASEOUT** the data on the specified address gets restored after evaluating the address expression.

<addr_expr> Expression which evaluates to the start address of the peripheral groups following the **BASESAVEOUT** command.

<address> Address which should be written before evaluating the address expression.
The original content gets saved before evaluating the expression and es restored afterwards.

<data> Data which should be send to the specified address before evaluating the address expression. This could also be a bit-mask.

CONFIG Configure default access width and line break for BIT

[Example]

Format:	**CONFIG** *<access_width>* [*<bits_per_line>*]

Configures the default access width used with **GROUP.auto**, aligns the field description after a **LINE** statement, and configures the bits-per-line emitted by the **BIT** statement.

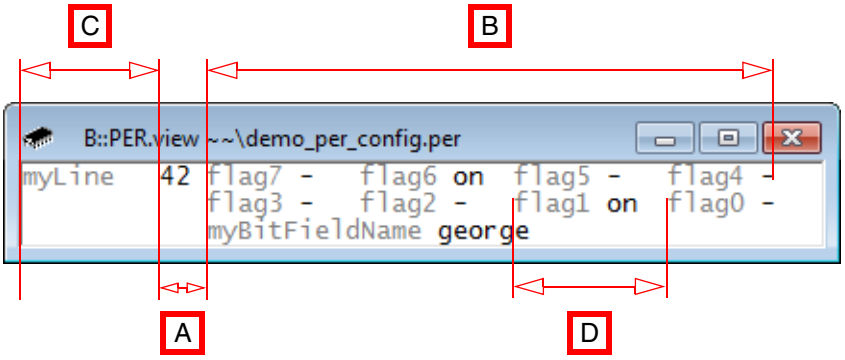
<access_width> By default the *<access_width>* is set to 8, which means (a) byte accesses to the memory by **GROUP.auto** and (b) no extra white space after any **LINE** statement. The access width in bits configures two things:

1. The default data access width in bytes of a **GROUP**, which does not specify its access width (**GROUP.auto**).
   The access width in bytes is calculated as follows:
   `(access width + 7) / 8 = result` (max. result: 8)
2. The minimum display width of the hex nibbles of a **LINE** statement.
   The minimum width is calculated as follows:
   `(access width + 3) / 4 + 1 = result` (max. result: 17)

<bits_per_line> By default *<bits_per_line>* is set to *<access_width>*. The bits per line set the number of bits shown in one line with the **BIT** statement before an automatic line break.
This setting affects only the **BIT** statement, but not the **BITFLD** statement (or others).

Example:

```
WIDTH   9. 10.
CONFIG 16.  4.
GROUP.auto D:0x000++1
LINE.BYTE 0x00 "myLine"
    BIT 7. "flag7" "-,on"
    BIT 6. "flag6" "-,on"
    BIT 5. "flag5" "-,on"
    BIT 4. "flag4" "-,on"
    BIT 3. "flag3" "-,on"
    BIT 2. "flag2" "-,on"
    BIT 1. "flag1" "-,on"
    BIT 0. "flag0" "-,on"
    NEWLINE
    BITFLD.BYTE 0x00 0--1 "myBitFieldName " "john,paul,george,ringo"
```



A Display width of the hex value emitted by the **LINE** statement. This width is the first parameter of the **CONFIG** statement.
In this example, *<access_width>* is 16 bits, i.e. (*<access_width>* + 3) / 4 + 1 = 5 characters.

B Number of **BIT** items in one single line before an automatic line break. This is configured with the second parameter of the **CONFIG** statement. (here: 4 **BIT** in one line).

C Width of the register name emitted by the **LINE** statement. This width is configured with the first parameter of the **WIDTH** statement. (here: 9 characters)

D Width of a bit displayed by the **BIT** statement. This width is configured with second parameter of the **WIDTH** statement. (here: 10 characters)

CSV — Enables CSV capabilities

Format:	**CSV.**[**ON** | **OFF**]

Enables or disables the new CSV file format for *.per files. For more information, see **"Comma-Separated-Values (CSV) File Format for *.per Files"**, page 12.

Refer to the **IF** command.

ELIF — Conditional GROUP display

Refer to the **IF** command.

ENDIAN — Define little or big endian

Format: **ENDIAN** [**BE** | **LE** | **DEF**]

With DEF parameter the endianness is set due to the configuration of the debugger. With this command the debugger accesses the target data with the specified endianness. This is done independent of the target and the system endianness settings.

Default: **ENDIAN** DEF

Example:

```
ENDIAN.LE                               ; little endian

ENDIAN.BE                               ; big endian

ENDIAN.DEF                              ; target default endian
```

ENDIF — Conditional GROUP display

Refer to the **IF** command.

ENTRY — Assign parameters to macros

Assign parameters used to open the peripheral file to macros, to parametrize the peripheral view (similar to the PRACTICE **ENTRY** command).

Refer to **“Passing Arguments”**, page 8.

Format:	**HELP.Winhelp** "*<file>*,*<item>*" **HELP.Online** "*<item>*"

Defines a button in the last GROUP header or tree control. HELP.Online calls the TRACE32 online manual. HELP.Winhelp calls a windows help file (available on Windows only).

IF Conditional GROUP display

Format:	**IF** *<condition>*
	ELIF *<condition>*
	ELSE
	ENDIF

<condition>:	Condition examples: - eval()==*<condition_val>* - %*<parameter>*==*<condition_val>* - (((data.*<size>*(*<address>*))&*<bit_mask>*)==*<condition_val>*)

GROUPs can be displayed conditionally using IF...ENDIF commands.

GROUPs defined in different IF and ELIF statements are overlaid at the same place in the window.

Only GROUPs which reside within the fulfilled condition are displayed. The ELSE part is displayed only when no other condition is true. All conditions are dynamically recalculated to reflect the current state of the peripheral.

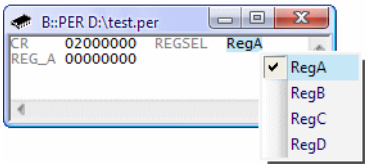
NOTE:	The **IF** command cannot be used inside a **GROUP**. (Please use **IF** always before a new **GROUP**.).

NOTE:	Unlike in the C programming language, the IF statement always evaluates all expressions also for logical operators && and ||.

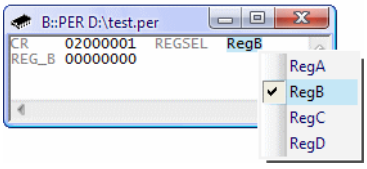
Example:

```
IF (((Data.Long(d:0x00))&0xf)==0x0)
  GROUP.LONG d:0x0++0x7
    LINE.LONG 0x0 "CR,Control register"
      BITFLD.LONG 0x0 0.--1. " REGSEL  ,Register select" "RegA,RegB,RegC,RegD"
    LINE.LONG 0x4 "REG_A,Register A"
ELIF (((Data.Long(d:0x00))&0xf)==0x1)
  GROUP.LONG d:0x0++0x7
    LINE.LONG 0x0 "CR,Control register"
      BITFLD.LONG 0x0 0.--1. " REGSEL  ,Register select" "RegA,RegB,RegC,RegD"
    LINE.LONG 0x4 "REG_B,Register B"
ELIF (((Data.Long(d:0x00))&0xf)==0x2)
  GROUP.LONG d:0x0++0x7
    LINE.LONG 0x0 "CR,Control register"
      BITFLD.LONG 0x0 0.--1. " REGSEL  ,Register select" "RegA,RegB,RegC,RegD"
    LINE.LONG 0x4 "REG_C,Register C"
ELSE
  GROUP.LONG d:0x0++0x7
    LINE.LONG 0x0 "CR,Control register"
      BITFLD.LONG 0x0 0.--1. " REGSEL  ,Register select" "RegA,RegB,RegC,RegD"
    LINE.LONG 0x4 "REG_D,Register D"
ENDIF
```

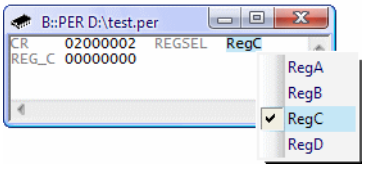
Register REG_A is selected if the value of the REGSEL bit field equals 0.



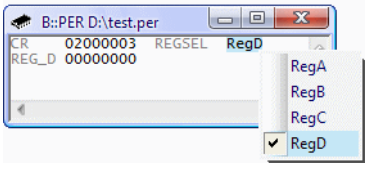
Register REG_B is selected if the value of the REGSEL bit field equals 1.



Register REG_C is selected if the value of the REGSEL bit field equals 2.



Register REG_D is selected if the value of the REGSEL bit field equals 3.



Format	**INCLUDE** *<file>*

Includes another peripheral file.

<file>	Path to another peripheral file

MENCONFIG PERMENU configuration

[build 158791 - DVD 09/2023]

Format:	**MENCONFIG** *<level>*

Configures the number of submenu levels for the **PERMENU** command. Has no effect for normal peripheral file processing. Overwrites the third optional parameter of **PERMENU**.

PERCMD Row definition in CSV-formatted *.per file

Format:	**PERCMD,***<column_list>*
<column_list>:	**Address,AccessWidth,Name,Tooltip,From,To,Choices**[**,RW**][**,Ignore**]

Optional definition of the columns of a peripheral file in CSV format.

- Default: If the **PERCMD** command is omitted in the CSV-formatted *.per file, then the sequence of columns must be: Address,AccessWidth,Name,Tooltip,From,To,Choices
- If the **PERCMD** command is included in a CSV-formatted *.per file, then *<column_list>* must contain all column names that are flagged as mandatory in the table below. Column names are case sensitive!

NOTE:	With the **PERCMD** command included in the CSV-formatted *.per file, you are free to arrange the mandatory and optional columns in any order.

Please also refer to **"Comma-Separated-Values (CSV) File Format for *.per Files"**, page 12.

Column Names	Meaning in the spreadsheet
Address (mandatory)	Absolute address of a register consisting of access class and value, or the offset from a previously defined **BASE** command. If empty, the value is assumed to be the same as the last known one. An address different from the previous one corresponds to the **LINE** command.
AccessWidth (mandatory)	Access width of the register. Valid values are: 8. 16. 32. and 64. If empty, the value is assumed to be the same as the last known one. An access width different from the previous one corresponds to the **LINE** command.
Name (mandatory/optional)	Name of the register. Mandatory for registers, optional for register fields (see **AutoIndent** -> binary mask).
Tooltip (mandatory)	Tooltip or more meaningful name of the register, e.g. the long form of the register name.
From (mandatory)	Lower boundary of a bit field of a register.
To (mandatory)	Upper boundary of a bit field of a register.
Choices (mandatory)	• Not empty: Comma-separated list of choices which will appear in the **PER.view** window in drop-down lists. Corresponds to the **BITFLD** command. A spreadsheet editor automatically adds the surrounding single quotes when the *.per file is exported in CSV file format. Otherwise the single quotes must be added manually. • • Empty: Corresponds to the **HEXMASK** command.
RW (optional)	Access rights to the register or register field. Valid values are: • **RD** (read) • **WR** (write) • **RW** (read/write) If empty, **WR** (write) will be taken as default.
ClearAddress (optional)	• Not empty: Defines a **SETCLRFLD** command, see **ClearFrom**. • Empty: Defines a **HEXMASK**, **BITFLD** or **EVENTFLD** command, see **ClearFrom**.

Column Names	Meaning in the spreadsheet
ClearFrom (optional)	• Not empty and column **ClearAddress** empty: Bit(s) of a register which can only be cleared by writing a '1'. Corresponds to the **EVENTFLD** command. This value must be the same as in the **From** column while the range is defined as **To** - **From**. • Not empty and columns **ClearAddress**, **SetAddress** and **SetFrom** not empty: Defines a register status bit with associated set and clear bits. See **SETCLRFLD** command. • Empty: Corresponds to **HEXMASK** or **BITFLD** command.
SetAddress (optional)	• Not empty: Defines a **SETCLRFLD** command, see **ClearFrom**. • Empty: Corresponds to **HEXMASK** or **BITFLD** command.
SetFrom (optional)	• Not empty: Defines a **SETCLRFLD** command, see **ClearFrom**. • Empty: Corresponds to **HEXMASK** or **BITFLD** command.
Ignore (optional)	• Ignores a column that is irrelevant for a *.per file, e.g. redundant columns extracted from binaries. • User-defined column names will also be ignored in the *.per files.

Example: The two last columns **Ignore** and **myCol1** will not have any effect.

```
PERCMD,Address,AccessWidth,Name,Tooltip,From,To,Choices,Ignore,myCol1
```

Format:	**REPEAT** *<count>* **(***<argument1>***) (***<argument2>***)** ... *<block>* **REPEAT.end**
<argument>:	**increment** *<start> <step>* **list** *<item1> <item2>* ... **strings** *<item1> <item2> ...* **function "***<PRACTICE_function>"*

Repeat the enclosed *<block>* of peripheral commands *<count>* times. Within the *<block>*, placeholders can be used in order to take into account iteration-specific register names and addresses. These placeholders are denoted as **$1**, **$2**, etc. and refer to *<argument1>*, *<argument2>* and so on.

increment	Placeholders within *<block>* have an initial value of *<start>* and get incremented by *<step>* on each iteration.
list	Placeholders within *<block>* will be assigned the *<items>*, which must be addresses or numeric values. There is a maximum of 16 list items.
strings	Same as **list**, but *<items>* must be text strings instead of numbers.
function	PRACTICE function which may contain placeholders. Quotes within the PRACTICE function must be escaped by a second quote.

Example 1: A GROUP and a LINE command get repeated 4 times:

```
REPEAT 4. (increment 0x0 0x1) (list 0x0 0x8 0xC 0x14)
  GROUP.LONG $2++0x3
  LINE.LONG 0x00 "MyRegister_$1,My test register $1"
REPEAT.end
```

Which is equivalent to:

```
  GROUP.LONG 0x0++0x3
  LINE.LONG 0x00 "MyRegister_0,My test register 0"
  GROUP.LONG 0x8++0x3
  LINE.LONG 0x00 "MyRegister_1,My test register 1"
  GROUP.LONG 0xC++0x3
  LINE.LONG 0x00 "MyRegister_2,My test register 2"
  GROUP.LONG 0x14++0x3
  LINE.LONG 0x00 "MyRegister_3,My test register 3"
```

Example 2: PRACTICE function

```
SIF COMPonentNUMBER("ETM")>0
  REPEAT COMPonentNUMBER("ETM") (increment 0 1)
                                (function "COMPonentNAME(""ETM"", $1)")
                                (function "COMPonent.BASE(""$2"",0)")
    BASE $3
    TREE $2
      ...
    TREE.END
  REPEAT.END
ENDIF
```

NOTE: When using placeholders as addresses or offsets within the *<block>*, the following restrictions apply:

- Placeholders cannot be used in expressions except for the 'add' operation ('+' sign). In that case the placeholder must be written first. E.g. 'LINE.LONG $1+0x100'.
- GROUP commands must use the '<placeholder>++<size>' format.
- Expressions in GROUP definitions must be enclosed by parenthesis, e.g. 'GROUP.LONG ($1+0x100)++0x3'.

The following restrictions apply to placeholders:

	Decimal / Hexadecimal numbers	Addresses	Strings
increment	x	only *<start>*	
list	x	x	
strings			x

Format: **REPEAT.REPLAY**

Replays the last complete **REPEAT** block. A block is considered as completed after the final **REPEAT.END**. The **REPLAY** command is typically used whenever more than 16 **list** items are required:

```
;The following example assumes 32 identical peripheral modules named
;'MyPeripheral_0' to 'MyPeripheral_31'. Their base addresses are
;distributed randomly and will thus not fit into the list argument.

TREE "MyPeripherals"
    BASE D:0x1720900
    TREE "MyPeripheral_0"
        REPEAT 1.
        <block>
        REPEAT.END
    TREE.END
    BASE D:0x1310900
    TREE "MyPeripheral_1"
        REPEAT.REPLAY
    TREE.END
    ...
    BASE D:0x1687000
    TREE "MyPeripheral_31"
        REPEAT.REPLAY
    TREE.END
TREE.END
```

SIF Conditional interpretation

Format: **SIF (CPU()=="***<cpu_name>***")**
SIF (CPUIS("*<cpu_name>***"))**
SIF (*<logical_comparison>***)**

According to the condition a block between **SIF** and **ENDIF** (or **SIF** and **ELSE**) will be interpreted when the peripheral file is opened or reparsed. The **SIF** command can be used also inside the **GROUPs**.

Example:

```
SIF (cpu()=="MIPS4KC")
GROUP.LONG CP0:16.++0.
  LINE.LONG 0x0 "Config,Configuration Register"
   BITFLD.LONG 0x00 31. " M ,Config1 register is implemented" "no,yes"
   ...
ELIF (cpu()=="MIPS4KEC")
GROUP.LONG 0x0 "Config,Configuration Register"
   BITFLD.LONG 0x00 31. " M ,Config1 register is implemented" "no,yes"
   ...
ELSE
GROUP.LONG 0x0 "Config,Configuration Register"
   BITFLD.LONG 0x00 31. " M ,Config1 register is implemented" "no,yes"
   ...
ELSE
ENDIF
```

Conventions :

SIF is only to be used to distinguish between CPUs, memory accesses should be avoided (not possible in system.mode down).

Using once a GROUP command inside a **SIF** block, all trees of the **SIF** block must contain GROUP commands. Also the next command after a finished **SIF** block must be a GROUP command then.

Using the command **PER.TestProgram** the error will be detected.

TREE

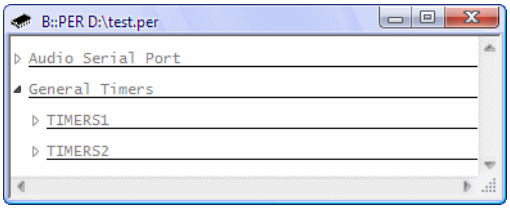
Define hierarchic display

Format: **TREE** "*<name>*"
TREE.OPEN "*<name>*"
TREE.END

Defines a “Treeview” of peripheral modules. The tree can be displayed/hidden by a tree control (+/-). It is possible to nest trees.

Example:

```
TREE "Audio Serial Port"          ; tree GROUP displayed closed by
;                                 default
TREE.END                          ; definition of the GROUP members
TREE.OPEN "General Timers"
  TREE "TIMERS1"                  ; tree GROUP displayed opened in the
  ;                               ; peripheral window
  TREE.END
TREE.END
```



WIDTH — Width of register names and a BIT description

Format	**WIDTH** [*<register_name>*] [*<bit_width>*]

Configures width of **LINE** register names and a **BIT** description.

<register_name> (default: 6.)	Sets the width of the register name emitted by the **LINE** statement.
<bit_width> (default: 9.)	Sets the width reserved for the output of a **BIT** statement. This setting affects only the **BIT** statement, but not the **BITFLD** statement (or others).

Example: For an example, see the **CONFIG** statement.

WAIT — Wait with PER windows until system is ready

Format	**WAIT** [*<address>* | *<expression>* | *<boolean_expression>*]

The **WAIT** command is available for all architectures and **PER** files, but it should only be used when required (i.e. **SIF** with target-dependent values). Most architectures will probably *not* require **WAIT**. But if there is a need to use **WAIT**, then the recommendation is to use **WAIT** at the beginning of a **PER** file.

<address>	Target address which has to be accessible; see example 2.
<expression>	TRACE32 expression which can be evaluated; see example 3.
<boolean_expression>	Boolean expression which has to be true; see example 4.

There are four ways to use the **WAIT** command, see examples 1 to 4.

Example 1: Wait with compilation until the target is up and regular memory can be accessed (this usually means that the target is stopped).

```
WAIT
```

Example 2: Wait with compilation until the target is up and the given memory address can be accessed (it is never really accessed).

```
WAIT ETM:0
```

Example 3: Wait with compilation until the target is up and the expression can be evaluated (the result does not matter).

```
WAIT Data.Long(D:0)
```

Example 4: Wait with compilation until the target is up and the boolean expression evaluates to true.

```
WAIT Data.Long(D:0)!=0
```

These commands are only useful inside a GROUP (**GROUP**, **RGROUP**, **WGROUP**, **HGROUP**, **SGROUP**).

Beside the commands **INDEX**, **SAVEINDEX** and **BUTTON**, which extend the memory access by a GROUP, the commands define how the data fetched by a GROUP command should be displayed and/or modified.

ABITFLD — Assign values to BITFLD choice items

[build 134843 - DVD 09/2021]

Format: **ABITFLD.***<size>* *<offset>* *<bit_range1>* [*<bit_range2>*] "*<display_name>*,*<tooltip>*" ["*<choices>*[**,%d...**|**%x...**|*<string>***...**]"] …

Same as **BITFLD**, but allows to assign values to the choice items:

```
ABITFLD.BYTE 0x00 0.--7. "Lock" "0xA5=Yes,0x5A=No"
```

The value and choice text must be separated by the equal sign and without blanks in between! Value/text pairs not listed will be output as hexadecimal value.

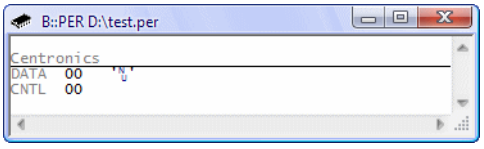
ASCII — Display ASCII character

Format: **ASCII**

The previously defined byte is displayed as an **ASCII** character.

Example:

```
GROUP.BYTE sd:0x100--0x101 "Centronics"
  LINE.BYTE 0x0 "DATA,Centronics Data Register"
       ASCII
  LINE.BYTE 0x1 "CNTL,Centronics Control Register"
```



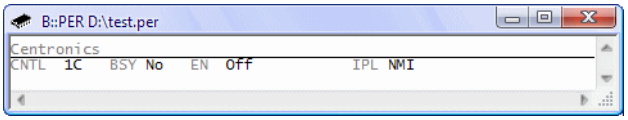
Format: **BIT** *<bit>*|*<bitrange>* "*<display_name>*,*<tooltip>*" "*<choices>*"

These fields are in fixed positions in the per window. The bit numbers must be entered from MSB to LSB. The size of a field depends on the number of bits and the size of the name header.

<bit> | *<bitrange>*	Defines bit's number and range. LSB is defined as the first, MSB as the second character.
<display_name>	Short name (abbreviation) of corresponding bit.
<tooltip>	The sentence accurately describing a bits functionality.
<choices>	Indicates states with bit field may take. LSB is defined as the first, MSB as the last one. Each state is separated by a comma.

Example:

```
GROUP sd:0x100--0x101 "Centronics"
  LINE.BYTE 0x00 "CNTL,Centronics Control Register"
       BIT 7 "BSY,Centronics Busy" "No,Yes"
       BIT 6 "EN,Centronics Enable" "Off,On"
       BIT 2--4 "IPL,Centronics Interrupt Level" "Off,1,2,3,4,5,6,NMI"
```



BITFLD Define bits individually

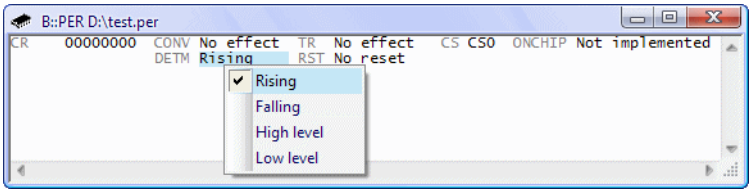
Format: **BITFLD.***<size>* *<offset>* *<bit_range1>* [*<bit_range2>*] "*<display_name>*,*<tooltip>*" ["*<choices>*[**,%d...**|**%x...**|**%y...**|*<string>***...**]"] …

BITFLD is used to display the bit field name and its contents in a free format. The fields are chained together in a line. A new line can be created by a **TEXTLINE** command.

<size>	Size of register (**Byte**, **Word**, **TByte**, **Long**, **Quad**).
<offset>	The bit field offset refers to the start address of the GROUP command.
<bit_range1>	Defines a range of bits (or a single bit) that belong to a bit field. The lower bit number has to come before the higher bit number, e.g. `3.--7.`

<bit_range2>	For disjunct bit fields (= where not all bits are in one block), you can define a second range of bits (or a single bit). Please see **examples**.
<short_name>	Short name (abbreviation) of corresponding bit field.
<long_name>	The sentence accurately describing a bit field functionality.
<choices>	Defines the possible values (in words) which the bit field may take. LSB is defined as the first, MSB as the last one. Each state is separated by a comma. If you define fewer *<choices>* than required for the *<bit_range>*, then append %x...
%d...	Placeholder for reserved/unused values at the end of *<choices>*. The values will be formatted as decimal numbers when displayed in the **PER.view** window. The field width is defined by the *<choices>*. If the decimal value is too large to fit into the field, a question mark is displayed. Please see **examples.**
%x...	Placeholder for reserved/unused values at the end of *<choices>*. The values will be formatted as hexadecimal numbers when displayed in the **PER.view** window. The field width is defined by the *<choices>*. If the hex value is too large to fit into the field, a question mark is displayed.
%y...	Placeholder for reserved/unused values at the end of *<choices>*. The values will be formatted as binary numbers when displayed in the **PER.view** window. The field width is defined by the *<choices>*. If the hex value is too large to fit into the field, a question mark is displayed.
<string>...	Placeholder for reserved/unused values at the end of *<choices>*. The values will be displayed as strings in the **PER.view** window.

```
BASE d:0x00000000
GROUP 0x00++0x03
    LINE.LONG 0x00 "CR,Control Register"
         BITFLD.LONG 0x00 31. " CONV ,Conversion Bit" "No effect,Conv"
         BITFLD.LONG 0x00 24. " TR ,Transfer" "No effect,Transferred"
         BITFLD.LONG 0x00 16.--19. " CS ,Chip Select"
 "CS0,CS1,CS2,CS3,CS4,CS5,CS6,CS7,CS8,CS9,CS10,CS11,CS12,CS13,CS14,CS15"
         BITFLD.LONG 0x00 5. " ONCHIP ,On chip trace implemented" "Not
implemented,Implemented"
TEXTLINE " "
         BITFLD.LONG 0x00 1. 3. " DETM ,Detection mode"
"Rising,Falling,High level,Low level"
         BITFLD.LONG 0x00 0. " RST ,Reset mode" "No reset,Reset"
```



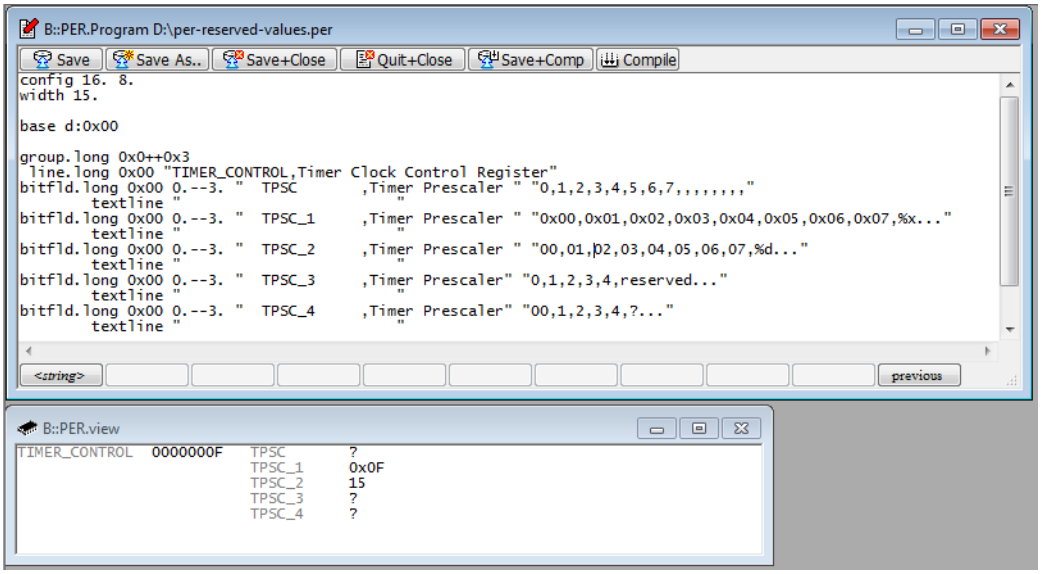
Examples

Example for bitranges:

Example 1:	31	...	15	14	13	12	11	10	9	8	7	6	5	4	3	**2**	1	0
Example 2:	31	...	15	14	13	12	11	10	9	**8**	**7**	**6**	**5**	**4**	**3**	**2**	1	0
Example 3:	31	...	15	**14**	13	12	11	10	9	**8**	**7**	**6**	**5**	**4**	**3**	**2**	1	0
Example 4:	31	...	**15**	**14**	13	12	11	10	9	**8**	**7**	**6**	**5**	**4**	**3**	**2**	1	0
Example 5:	31	...	15	**14**	13	12	11	10	9	8	7	6	5	4	3	**2**	1	0
Example 6:	31	...	**15**	**14**	13	12	11	10	9	8	7	6	5	4	3	**2**	1	0

```
;Example 1          <bit_range1>
BITFLD.<size> 0x00   2.

;Example 2          <bit_range1>
BITFLD.<size> 0x00   2.--8.

;Example 3          <bit_range1>  <bit_range2>
BITFLD.<size> 0x00   2.--8.        14.

;Example 4          <bit_range1>  <bit_range2>
BITFLD.<size> 0x00   2.--8.        14.--15.

;Example 5          <bit_range1>  <bit_range2>
BITFLD.<size> 0x00   2.            14.

;Example 6          <bit_range1>  <bit_range2>
BITFLD.<size> 0x00   2.            14.--15.
```

Example for handling unused/reserved values:



BUTTON — Define command button

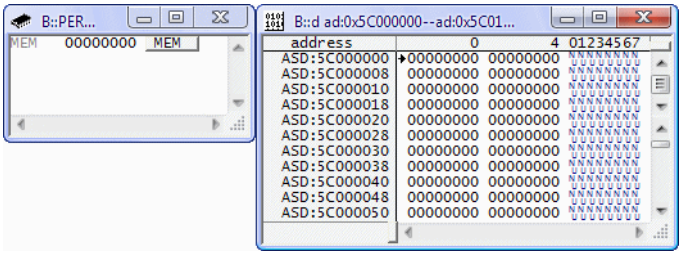
Format:	**BUTTON** "*<text>*" "*<cmdline>*"

Clicking an input field (button) executes the defined command line. This field can be used to execute input/output commands or open different views (e.g. memory dumps).

<text>	Name of the button.
<cmdline>	Contains command, address area and an access size.

Example 1: Button with single command.

```
GROUP.LONG 0x00++0x3
  LINE.LONG 0x00 "MEM,Memory Array"
  BUTTON "MEM " "Data.dump ad:0x5C000000--ad:0x5C01FFFF /Long"
```



Example 2: Button with multiple commands.

```
GROUP.LONG D:0x00++0xFF
  LINE.LONG 0x00 "RST_VEC,Reset Vector"
  BUTTON "Clear Vector Table"
  (
     Data.dump 0x00++0xFF /Long
     Data.set %Long ad:0x5C000000++01FFFF 0
  )
```

COPY — Copy GROUP

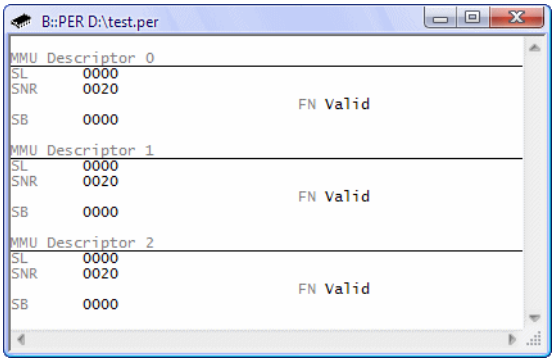
Format:	**COPY** [*<number>*]

Copies the last defined **GROUP** to the current **GROUP**. The optional argument defines which **GROUP** should be copied. Number of the **GROUP** is calculated backward form the current one. The command is used to duplicate the definition of **GROUP**s, e.g. for devices with many equal channels.

<number> Optional GROUP number.

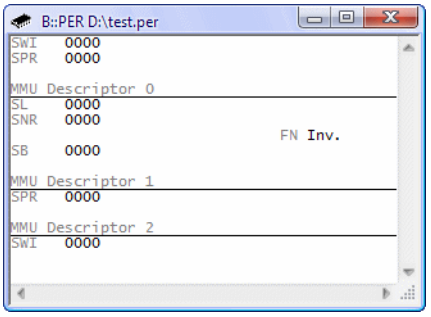
Example 1:

```
GROUP.WORD sd:0x80008038--0x8000803f "MMU Descriptor 0"
  LINE.WORD 0x0 "SL,Segment Length"
  LINE.WORD 0x2 "SNR,Segment Number"
       bit 5 " FN, Flush" "Inv.,Valid"
  LINE.WORD 0x4 "SB,Segment Base Address"
GROUP.WORD sd:0x80008048--0x8000804f "MMU Descriptor 1"
  copy
GROUP.WORD sd:0x80008050--0x80008057 "MMU Descriptor 2"
  COPY
```



Example 2:

```
GROUP.WORD sd:0x80008034--0x80008035
  LINE.WORD 0x0 "SWI,Segment Width"
GROUP.WORD sd:0x80008036--0x80008037
  LINE.WORD 0x0 "SPR,Segment Priority"
GROUP.WORD sd:0x80008038--0x8000803f "MMU Descriptor 0"
  LINE.WORD 0x0 "SL,Segment Length"
  LINE.WORD 0x2 "SNR,Segment Number"
       bit 5 " FN, Flush" "Inv.,Valid"
  LINE.WORD 0x4 "SB,Segment Base Address"
GROUP.WORD sd:0x80008048--0x8000804f "MMU Descriptor 1"
  COPY 2
GROUP.WORD sd:0x80008050--0x80008057 "MMU Descriptor 2"
  COPY 4
```



DECMASK

Define bits for decimal display

Format: **DECMASK.**<*access_size*>[**.**<*display_length*>] <*offset*> <*bit_range*> **<*scale*>** **[<*add*>]** "**<*display_name*>,<*tooltip*>**"

While the similar command **HEXMASK** displays bits as a hexadecimal value, **DECMASK** displays bits as decimal value.

DECMASK defines a set of bits, which should be displayed as decimal value. The bits are extracted from the current buffer at location defined in the bitrange. The result of this extract is multiplied by *<scale>* and increased by the optional *<add>* value.

<access_size>	Size of register access (**Byte**, **Word**, **TByte**, **Long**, **Quad**).
<display_length>	Length of displayed field (**Byte**, **Word**, **TByte**, **Long**, **PByte**, **HByte**, **SByte**, **Quad**).
<offset>	The DECMASK field offset refers to the start address of the GROUP command.
<bit_range>	Defines range of the DECMASK field. LSB is defined as the first, MSB as the second character.
<scale>	Multiplier value. May be a floating point value since build. 46110
<add>	Optional addend - increases value.
<display_name>	Short name (abbreviation) of corresponding DECMASK field.
<tooltip>	The sentence accurately describing a DECMASK field functionality.

FLOATMASK

Define bits for decimal floating point display

Format: **FLOATMASK.***<access_size>*[**.SIGNED**][**.***<display_length>*] *<offset>* *<bit_range>* *<scale>* [*<add>*] "*<display_name>*,*<tooltip>*"

While the similar command **DECMASK** displays bits only as a decimal value *without* positions after decimal point, **FLOATMASK** displays bits as decimal value *with* positions after decimal point.

FLOATMASK defines a set of bits, which should be displayed as decimal value. The bits are extracted from the current buffer at location defined in the bitrange. The result of this extract is multiplied by *<scale>* and increased by the optional *<add>* value.

<access_size>	Size of register access (**Byte**, **Word**, **TByte**, **Long**, **Quad**).
<display_length>	Length of displayed field (**Byte**, **Word**, **TByte**, **Long**, **PByte**, **HByte**, **SByte**, **Quad**).
<offset>	The DECMASK field offset refers to the start address of the GROUP command.
<bit_range>	Defines range of the DECMASK field. LSB is defined as the first, MSB as the second character.

<scale>	Multiplier value. May be a floating point value since build. 46110
<add>	Optional addend - increases value.
<display_name>	Short name (abbreviation) of corresponding DECMASK field.
<tooltip>	The sentence accurately describing a DECMASK field functionality.

<access_size>	Size of register access (byte, word, tbyte, long, quad).
<display_length>	Length of displayed field (byte, word, tbyte, long, quad).
<offset>	The FLOATMASK field offset refers to the start address of the GROUP command.
<bit_range>	Defines range of the FLOATMASK field. LSB is defined as the first, MSB as the second character.
<scale>	Multiplier value. Usually a floating point value.
<add>	Optional addend - increases value.
<display_name>	Short name (abbreviation) of corresponding FLOATMASK field.
<tooltip>	The sentence accurately describing a FLOATMASK field functionality.

Example:

```
GROUP D:0x80001204++3 "Timer"
  TEXTLINE ""
  DECMASK.LONG   0 0--31. 1     " milliseconds:  "
  TEXTLINE ""
  FLOATMASK.LONG 0 0--31. 0.001 " seconds:        "
  TEXTLINE ""
```

Format: **EVENTFLD.***<size> <offset> <bit_range>* "*<display_name>*,*<tooltip>*" "*<choices>*"

Defines an event bit display in a free format. An event bit can be cleared by writing a ’1’. Writing '0' does not affect event bit. The fields are chained together in a line. A new line can be created by a **TEXTLINE** command. The implementation format is the same as a **BITFLD** format.

<size>	Size of register (byte, word, tbyte, long, quad).
<offset>	The event bit offset refers to the start address of the GROUP command.
<bit_range>	Defines range of the bit field. LSB is defined as the first, MSB as the second character. Optionally the third character is bit (or bit range), used if two bit fields are conjuncted.
<display_name>	Short name (abbreviation) of corresponding event bit field.
<tooltip>	The sentence accurately describing a event bit field functionality.
<choices>	Indicates states with bit field may take. LSB is defined as the first, MSB as the last one. Each state is separated by a comma.

Example:

```
GROUP.WORD d:0x100--0x11f "TPU Channels"
  TEXTLINE ""
  TEXTLINE "CH FUNC PRIO HSF HSR IEF ISF LNK SGL CHS PRM0 PRM1"
  TEXTLINE " 0,Channel 0"
  BITFLD.WORD 0x1e 0.--1. " " " Off, Low, Mid,High"
  BITFLD.WORD 0x16 0.--1. " " " $0, $1, $2, $3"
  EVENTFLD.WORD 0x1a 0. " " "No,Yes"
```

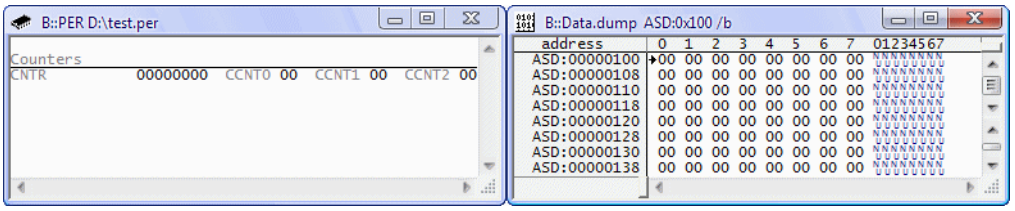
Format:	**HEXFLD.***<length>* *<offset>* "*<display_name>*,*<tooltip>*"

Defines HEX value in a free format. The fields are chained together in a line. A new line can be created using **TEXTLINE** command. If not the whole value should be displayed. The output size can be limited by the “length” parameter.

<length>	Length of HEX field (**Byte**, **Word**, **TByte**, **Long**, **Quad**).
<offset>	The HEX field offset refers to the start address of the GROUP command.
<display_name>	Short name (abbreviation) of corresponding HEX field.
<tooltip>	The sentence accurately describing a HEX field functionality.

Example:

```
GROUP 0x100++0x03 "Counters"
  LINE.LONG 0x00 "CNTR,Channel Counter Register"
       HEXFLD.BYTE 0x00 " CCNT0 ,Channel Counter 0"
       HEXFLD.BYTE 0x01 " CCNT1 ,Channel Counter 1"
       HEXFLD.BYTE 0x02 " CCNT2 ,Channel Counter 2"
```



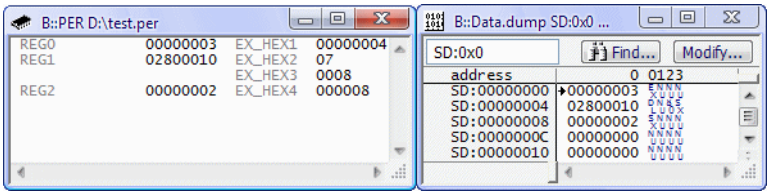
Format:	**HEXMASK.***<access_size>*[**.***<display_length>*] *<offset>* *<bit_range>* *<scale>* [*<add>*] "*<display_name>*,*<tooltip>*"

Defines set of bits using HEX value. The bits are extracted from the current buffer at location defined in the bitrange. The result of this extract is multiplied by scale. The *<add>* value is optional.

<access_size>	Size of register access (**Byte**, **Word**, **TByte**, **Long**, **Quad**).
<display_length>	Length of displayed field (**Byte**, **Word**, **TByte**, **Long**, **PByte**, **HByte**, **SByte**, **Quad**).
<offset>	The HEX mask field offset refers to the start address of the GROUP command.
<bit_range>	Defines range of the HEX mask field. LSB is defined as the first, MSB as the second character.
<scale>	Multiplier value. May be a floating point value since build. 46110.
<add>	Optional addend - increases Hex mask value.
<display_name>	Short name (abbreviation) of corresponding HEX mask field.
<tooltip>	The sentence accurately describing a HEX mask field functionality.

Example:

```
CONFIG 16. 8.

BASE 0x0
WIDTH 6.
GROUP.LONG 0x00++0xb
LINE.LONG 0x00 " REG0,register 0"
  HEXMASK.LONG 0x00 0.--29. 1. 1. " EX_HEX1  ,Example Hex mask 1"
LINE.LONG 0x04 " REG1,Register 1"
  HEXMASK.LONG.BYTE 0x04 23.--30. 1. 2. " EX_HEX2  ,Example Hex mask 2"
  TEXTLINE "              "
  HEXMASK.LONG.WORD 0x04 4.--15. 8. "  EX_HEX3  ,Example Hex mask 3"
LINE.LONG 0x8 " REG2,Register 2"
  HEXMASK.LONG.TBYTE 0x08 0.--23. 1. 6. " EX_HEX4  ,Example Hex mask 4"
```



HIDE — Define write-only line

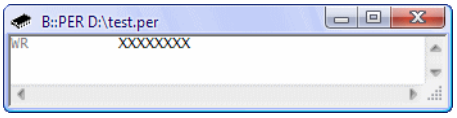
Format: **HIDE.**<*size*> <*offset*> "<*display_name*>,<*tooltip*>"

This field is used for write-only ports like USART transmitters data registers. **HIDE** command should be used together with **HGROUP** command.

<*size*>	Size of register (byte, word, tbyte, long, quad).
<*offset*>	The register offset refers to the start address of the HGROUP command.
<*display_name*>	Short name (abbreviation) of corresponding register.
<*tooltip*>	The sentence accurately describing a register functionality.

Example:

```
HGROUP.LONG 0x00++0x3
  HIDE.LONG 0x00 "WR,Write only Register"
```



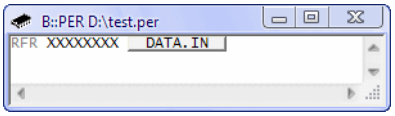
IN — Define input field

Format: **IN**

An input-field (key) is displayed for the previously defined byte. Clicking that field results in reading data from previously defined location. To execute a read cycle **IN** command must be used along with a **HIDE** definition. It is used for destructive-read ports (i.e. data port of serial interface).

Example:

```
BASE d:0xA00F0000
HGROUP.LONG 0x00++0x3
  HIDE.LONG 0x00 "RFR,Receive FIFO Register"
  IN
```



INDEX

Output a value

Format:	**INDEX** *<address>* [**%***<format>*] *<dataread>* *<datawrite>* **OUT** (deprecated)
<format>:	**Byte** | **Word** | **Long** | **Quad** | **TByte** | **HByte** **Float.** [**Ieee** | **IeeeDbl** | **IeeeeXt** | *<others>*] **BE** | **LE**

Sends specified data to the port. **INDEX** command must be placed after a **GROUP** definition. The data is sent to the port prior to the port access or modification. If two bytes are defined, the second byte is used for writing to the specified port (different indices for reading and writing). It is useful for ports which must be selected first.

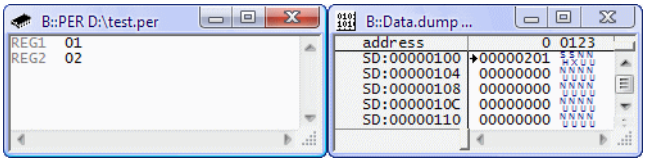
Please consider: As the display is refreshed permanently the index register is modified as well.

NOTE:	The **INDEX** command has no effect inside an **SGROUP** command.

<address>	Destination address.
<dataread>	Data send to the specified address before fetching the data shown by the group definition.
<datawrite>	Data send to the specified address before executing a write to a member of the group definition.

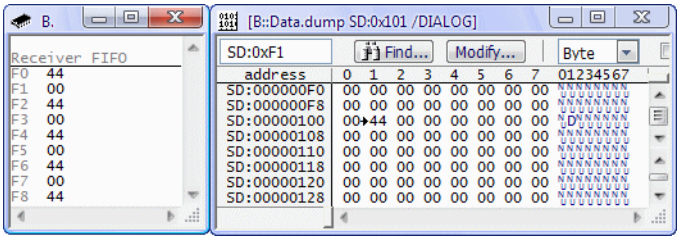
Example 1:

```
GROUP sd:0x100--0x100                          ; select register 1
  INDEX sd:0x100 0x01
  LINE.BYTE 0x0 "REG1,Register index 1"
GROUP sd:0x101--0x101                          ; select register 2
  INDEX sd:0x101 0x02
  LINE.BYTE 0x0 "REG2,Register index 2"
```



Example 2:

```
GROUP sd:0x101 0x10 "Receiver FIFO"
INDEX sd:0x100 0 0x80 0
     LINE.BYTE 0x0 "F0,FIFO position 0"
     LINE.BYTE 0x1 "F1,FIFO position 1"
     LINE.BYTE 0x2 "F2,FIFO position 2"
     LINE.BYTE 0x3 "F3,FIFO position 3"
     LINE.BYTE 0x4 "F4,FIFO position 4"
     LINE.BYTE 0x5 "F5,FIFO position 5"
     LINE.BYTE 0x6 "F6,FIFO position 6"
     LINE.BYTE 0x7 "F7,FIFO position 7"
     LINE.BYTE 0x8 "F8,FIFO position 8"
```



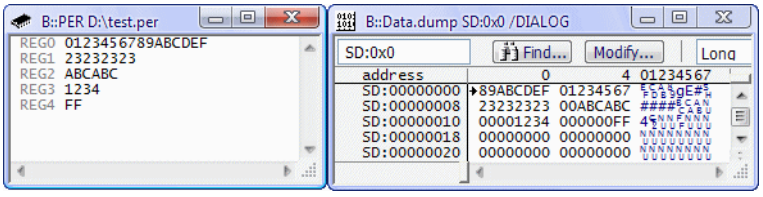
Format:	**LINE.[**<*size*> | **FLOAT**.<*format*>] <*offset*> "<*display_name*>,<*tooltip*>"

The **LINE** command defines registers short name and its long name. The value of the offset is added to the address defined in the previous **GROUP** command. The **CONFIG** command affects the displayed format of the **LINE** command.

<*size*>	Size of register (**Byte**, **Word**, **TByte**, **Long**, **Quad**).
<*format*>	Display register content as floating point number. Currently the following formats are supported: • IEEE: 32 bit IEEE-754 single • IEEEDBL: 64 bit IEEE-754 double
<*offset*>	The register offset refers to the start address of the GROUP command.
<*display_name*>	Short name (abbreviation) of corresponding register.
<*tooltip*>	Register long name (a sentence accurately describing the register functionality).

Example:

```
BASE 0x0
WIDTH 6.
GROUP.QUAD 0x00++0x7
  LINE.QUAD  0x00 " REG0,Register 0"
GROUP.LONG    0x08++0x3
  LINE.LONG  0x00 " REG1,Register 1"
GROUP.TBYTE   0x0c++0x2
  LINE.TBYTE 0x00 " REG2,Register 2"
GROUP.WORD    0x10++0x1
  LINE.WORD  0x00 " REG3,Register 3"
GROUP.BYTE    0x14++0x0
  LINE.BYTE  0x00 " REG4,Register 4"
```



Only available on TRACE32 for PowerPC

Format: **MUNGING** *<belle>*

Usually byte ordering is either little endian or big endian mode. For PPC additional munging little endian and munging big endian modes are provided. For a detailed description refer to PPC documentation.

Special address translation for PowerPC little endian mode.

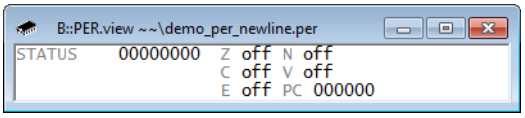
```
MUNGING.LE
```

NEWLINE Line break within detailed register description

Format: **NEWLINE**

Creates a line break for the detailed description of the fields of a peripheral register. The indentation of the new line can be configured with the first parameter of **WIDTH** and **CONFIG**.

```
CONFIG 32.
WIDTH  10.
GROUP.LONG D:0x100++3
LINE.LONG 0x00 "STATUS,Status Register"
    BITFLD.LONG 0x00 31. " Z ,Zero Flag"      "off,on"
    BITFLD.LONG 0x00 30. " N ,Negative Flag"  "off,on"
    NEWLINE
    BITFLD.LONG 0x00 29. " C ,Carry Flag"     "off,on"
    BITFLD.LONG 0x00 28. " V ,Overflow Flag"  "off,on"
    NEWLINE
    BITFLD.LONG 0x00 27. " E ,Interrupt Mask" "off,on"
    HEXMASK.LONG.TBYTE 0x00 0.--23. 4 " PC ,Program Counter"
```



Format: **RBITFLD.***<size> <offset> <bit_range>* "*<display_name>*,*<tooltip>*" "*<choices>*"

RBITFLD is identical to **BITFLD** with the difference that the defined bits are read-only. It can be used to visualize that certain settings within a read-write register are read-only.

<size>	Size of register (**Byte**, **Word**, **TByte**, **Long**, **Quad**).
<offset>	The bit field offset refers to the start address of the GROUP command.
<bit_range>	Defines range of the bit field. LSB is defined as the first, MSB as the second character. Optionally the third character is bit (or bit range), used if two bit fields are conjuncted.
<short_name>	Short name (abbreviation) of corresponding bit field.
<long_name>	The sentence accurately describing a bit field functionality.
<choices>	Defines the possible values (in words) which the bit field may take. LSB is defined as the first, MSB as the last one. Each state is separated by a comma.

```
BASE D:0xF0001234
GROUP 0x00++0x03
  LINE.LONG 0x00 "CSR,Control and Status Register"
    RBITFLD.LONG 0x00 1. " RSTST ,Reset status" "Reset inactive, Reset
active"
    BITFLD.LONG 0x00 0. " RST ,Reset" "No reset,Reset"
```

RHEXMASK Define bits for a hexadecimal display (read-only)

Format: **RHEXMASK.<access_size>[.***<display_length>] <offset>* <bit_range> <scale> [<add>] “<display_name>,<tooltip>”

Same as **HEXMASK** but bits are read-only.

Format:	**SAVEINDEX** *<address>* [**%***<format>*] *<dataread>* *<datawrite>* **SAVEOUT** (deprecated)
<format>:	**Byte** | **Word** | **Long** | **Quad** | **TByte** | **HByte** **Float.** [**Ieee** | **IeeeDbl** | **IeeeeXt** | *<others>*] **BE** | **LE**

Sends the specified data to the port. The current values at the port are read before the access is made and are restored after the access. The byte is sent to the port prior to the port access or modification. SAVEINDEX command must be placed after a GROUP definition. If two bytes are defined, the second byte will be used for writing to the specified port (different indices for reading and writing). This is useful for ports which are selected by another port when the index register can be read back.

<address>	Destination address.
<dataread>	Data send to the specified address before fetching the data shown by the group definition.
<datawrite>	Data send to the specified address before executing a write to a member of the group definition.

NOTE: **SAVEINDEX** command has no effect inside an **SGROUP** command.

```
GROUP d:0x11--0x11 "SERIAL CONTROL 80196"
  SAVEINDEX d:0x14 %byte 0x00 0x0f              ;index 0 for read,
                                                ;15 for write
  LINE.BYTE 0 "SCN,Serial Control Register"
```


Format:	**SAVETINDEX** *<address>* [**%***<format>*] *<dataread>* *<datawrite>*
<format>:	**Byte** | **Word** | **Long** | **Quad** | **TByte** | **HByte** **Float.** [**Ieee** | **IeeeDbl** | **IeeeeXt** | *<others>*] **BE** | **LE**

Similar to **SAVEINDEX**, uses however a different sequence for write accesses: the data value is first written to the address and the index is written to trigger/transfer the write operation.

Same as **DECMASK**, but values are interpreted as signed numbers.

SFLOATMASK Signed FLOATMASK

Same as **FLOATMASK**, but values are interpreted as signed numbers.

SETCLRFLD Define set/clear locations

Format: **SETCLRFLD.***<size> <offset1> <bit1> <offset2> <bit2> <offset3> <bit3>* "*<display_name>*,*<tooltip>*" "*<choices>*"

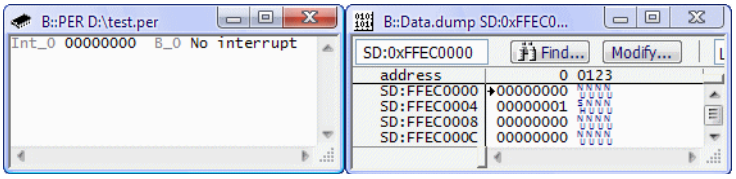
Defines a bit display in a free format. The fields are chained together in a line. A new line can be created by a **TEXTLINE** command.

<size>	Size of register (**Byte**, **Word**, **TByte**, **Long**, **Quad**).
<offset1><bit1>	Status register offset and corresponding bit number.
<offset2> <bit2>	Set register offset and corresponding bit number.
<offset3> <bit3>	Clear register offset and corresponding bit number.
<display_name>	Short name (abbreviation) of corresponding set/clear bits.
<tooltip>	The sentence accurately describing a set/clear bits functionality.
<choices>	Indicates states with bit field may take. The first state is responsible for clearing, the second one for setting corresponding set/clear bits. Each state is separated by a comma.

The command is an extension of the **BITFLD** command. Additionally to the BITFLD command two further locations must be entered. The first parameter pair offset1 - bit1 is the location where the data is read from. The second parameter pair offset2 - bit2 is the set location. The third parameter pair offset3 - bit3 is the clear location.

Usually the SETCLRFLD-command is used if the read location is a status register, which shows the status of I/O ports and other (not static) registers exist to enable and disable ports. If the port is enabled, the value of '1' is set to the corresponding bit in the register addressed by location 2 (other bits are cleared). If the port is disabled, the value of '1' is set at the corresponding bit position in the register addressed by location 3 (the other bits are cleared).

```
BASE sd:0xffec0000
GROUP.LONG 0x00++0x3
  LINE.LONG 0x00 "Int_0,Interrupt Register 0"
       SETCLRFLD.LONG 0x0 0. 0x4 0. 0x8 0. " B_0 ,Bit 0"
"No Interrupt,Interrupt"

;writing 1 sets the bit in the Set   Register
;writing 0 sets the bit in the Clear Register
;the result is read from the Status register
```



STRING

Display a string saved in memory

Format: **STRING** *<display_width> <offset> <string>*

Defines a field to display an ASCII encoded string, which is saved in target memory.

<width>	Number of bytes/characters.
<offset>	Offset to group start address.
<string>	Field name. Will prepend the ASCII string.

Example:

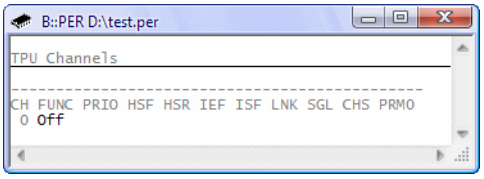
```
BASE sd:0xff000000
WIDTH 8.
GROUP.LONG 0x00++0x03
lINE.LONG 0x00 "KEYREG,"
STRING 4. 0. "KEY "
STRING 3. 0. " KEY "
STRING 3. 1. " KEY "
```

Format: **TEXTLINE** "*<text>*"

The text can either be used as general comment or as a header to **BITFLD** or **HEXFLD** fields. **TEXTLINE** creates a new line.

<text> Optional text.

```
GROUP d:0x0e00--0x0fff "TPU Channels"
  TEXTLINE ""
  TEXTLINE "-----------------------------------------"
  TEXTLINE "CH FUNC PRIO HSF HSR IEF ISF LNK SGL CHS PRM0"
  TEXTLINE " 0,Channel 0"
  BITFLD.WORD 0x1e 0.--1. " " "Off,Low,Mid,High"
```



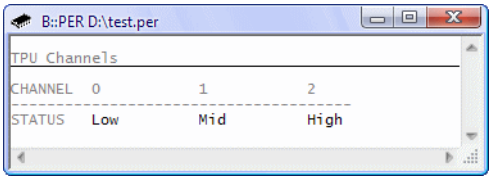
TEXTFLD Define text header

Format: **TEXTFLD** "*<text>*"

Defines text without creating a new line.

<text> Optional text.

```
GROUP d:0x80000000--0x80000fff "TPU Channels"
TEXTLINE ""
TEXTLINE "CHANNEL "
TEXTFLD " 0,Channel 0"
TEXTFLD " 1,Channel 1"
TEXTFLD " 2,Channel 2"
TEXTLINE "-------------------------------------"
TEXTLINE "STATUS  ,Status"
BITFLD.WORD 0x0 0.--1. " " "Off,Low,Mid,High"
BITFLD.WORD 0x0 2.--3. " " "Off,Low,Mid,High"
BITFLD.WORD 0x0 4.--5. " " "Off,Low,Mid,High"
```



TINDEX

Output a value

Format:	**TINDEX** *<address>* [**%***<format>*] *<dataread>* *<datawrite>*
<format>:	**Byte** | **Word** | **Long** | **Quad** | **TByte** | **HByte** **Float.** [**Ieee** | **IeeeDbl** | **IeeeeXt** | *<others>*] **BE** | **LE**

Similar to **INDEX**, uses however a different sequence for write accesses: the data value is first written to the address and the index is written to trigger/transfer the write operation.

Graphical User Interface

TRACE32 is able to generate peripheral files out of certain other file formats. A unified GUI is available through the **PER.IMPORT.view** command. But of course TRACE32 lets you convert all input files via command line or cmm script as well.

For a complete list of input formats see **PER.IMPORT.ForMaT**.

Rules file

NOTE:	A rules file is a XML based recipe which allows you to modify the appearance of a converted input file retrospectively. The according schema file can be found at */demo/tools/per_import/rules.xsd*.

Rules file description

The rule file consists of root tag <rules> and list of <rule> tags inside of them. Nesting of <rule> tags is not allowed.

Structure of <rules> tag:

```
<rules>
    <rule><!-- rule definition --></rule>
    <!-- other rules... -->
</rules>
```

Rule definition

A rule definition contains one or more select tags <select> followed by one or more <command> tags <command_name>:

- The <select> tag defines on which elements the <command> tags will be applied. The first <select> tag will search in all elements of the XML file. The next <select> tag will search on the results of the previous select tag. That way, selecting the desired elements can be achieved by reducing the search base step by step.
- The <command> tag defines a modification that will be executed on the selected elements. Several <command> tags can be specified to apply independent modifications on the same search results.

Structure of <rule> tag:

```
<rule verbose="yes|no">
    <select ... />
    <!-- other selects... -->
    <command_name ... ><!-- command definition --></command_name>
    <!-- other commands -->
</rule>
```

Available verbose values:

verbose	description
no	Default. Do not write debug messages into rules-logs.log file.
yes	Write debug messages into rules-log.log file.

For example to change the name of the module MODULE_EXAMLE to a new one, you can write follow rule:

```
<rules>
    <rule>
        <select element="module" property="name" regex="MODULE_EXAMPLE"
/>
        <modify property="name" value="BETTER_MODULE_NAME" />
    </rule>
</rules>
```

The above rule is a one-step search with a single command for modification. It can be explained as follows:

1. <search>: Search all elements of type module, where element.name is "MODULE_EXAMPLE".

2. <command>: Change name of element.name to "BETTER_MODULE_NAME" for all found elements.

Selecting defined elements using <select>

Selects targets elements to be processed by the commands. The selection is determined by the element type and its properties.

<select> tags usage:

```
<select element="element_name"
        property="property_name"
        regex="regular_expression"
        all_occurrences="no|yes"
        invert_regex="no|yes" />
```

element	Specifies the type of element to be searched for. See **list of all elements**.
property	Specifies the type of property to be matched.See list of all properties. Not all properties are allowed for a given element. **See this table**.
regex	Regular expression for matching the property's value.
all_occurrences	• no: Default. Search first occurrence only. • yes: Search all occurrences (can reduce performance).
invert_regex	• no: Default. regular expression is not inverted. • yes: Invert regular expression.

Elements

	element
`sif(cpuis("CORTE?R4*"))`	**sif**
`  repeat 2. (increment 0 1) (list ad:0x0 ad:0x10)`	**repeat**
`    tree.open "DAM$1"` `    base $2`	**module**
`      group.long 0x4++0x03` `        line.long 0x0 "ACCEN,Access Enable"`	**register**
`          bitfld.long 0x00 0. "EN0,Master0 Enable"`	**field**
`            "Enabled,Disabled"`	**state**

		module	register	field	state
name		x	x	x	x
description		x	x	x	x
value					x
access_type	• RW (read write) • RO (read only) • WO (write only) • W1C (write one to clear) • WS (write secured) • H (hidden)		x		
access_class		x			
offset			x		
size			x		
lower_range				x	
upper_range				x	
intrusive_read	• no: Reading of register is not intrusive. • yes: Reading of register is intrusive		x		
is_open	• no: Default. Tree is hidden. • yes: Tree is shown.	x			
sif		x	x	x	
view_name			x		
path		x	x	x	x
button_name			x		
button_command	For multiple button commands use
as line break.		x		

Commands

Commands are element modificators that process data sets extracted from data model by the <select> tag. One data set can be used by multiple commands. Command tags must follow the <select> tags.

Structure of command tag:

```
<command_name command_atributes... > <!-- definition --> </command_name>
<!-- or -->
<command_name command_atributes... />
```

Each command has its own set of attributes. The following are common to all commands:

element	Specifies target elements for command. Valid are elements chosen by preceeding <select> operations.
property	Property for filtering selected elements.
regex	Regular expression for filtering selected elements.

Above attributes allow to extend the search operation of <select> tags in the commands. This way the

number of used <select> tags can be reduced. For example below rule:

```
<select element="module" property="name" regex="MODULE_NAME" />
<select element="register" property="name" regex="REG_NAME" />
<modify property="name" value="NEW_NAME" />
```

is the long version of:

```
<select element="module" property="name" regex="MODULE_NAME" />
<modify element="register" property="name" regex="REG_NAME"
value="NEW_NAME" />
```

Selecting all subelements is possible, too. To do this, omit the regex attribute:

```
<select element="module" property="name" regex="MODULE_NAME" />
<modify element="register" property="name" value="NEW_NAME" />
```

<create_header>

Used to overwrite the default header at the beginning of each peripheral file.Only has an effect if **PER.<format>.Save** is used with the **/Header** option.

Supported elements: permodel

Structure of <create_header> tag:

```
<create_header title="title" props=”props” author=”author”
changelog=”changelog” manufacturer=”manufacturer” doc=”doc” core=“core”
chip=”chip” copyright=”copyright” include=”include”/>
```

title (optional)	Overwrites default @Title text.
props (mandatory)	Must be “Confidential” or “Released”.
author (mandatory	Set author.
changelog (optional)	Overwrites default @Changelog text.
manufacturer (optional)	Overwrites default @Manufacturer text.
doc (optional)	Overwrites default @Doc text.
core (optional)	Overwrites default @Core text.
chip (optional)	Overwrites default @Chip text.
copyright (optional)	Overwrites default @Copyright text.
include (optional)	Adds an %include after the copyright.

<derive_module>

Derives new trees by means of module or register names. A typical use case is an input file with no explicit hierarchy information. This command helps in creating trees instead of using lots of **<create_module>** and **<modify>** commands.

Supported elements: module , register

Structure of <derive_module> tag:

```
<derive_module separator="character" depth=”max_level” element="element"
preserve=”yes|no”/>
```

character	Any character that separates module levels in the name.
depth	Maximum number of tree levels to derive. Default=none.
element	register | module | all. Default = all.
preserve	Preserves original register name. Default=no.

Example:

```
<!-- Register name is MEM_FLASH_STATUS -->

<derive_module separator="_">

<!-- Tree "MEM"                -->
<!--    Tree "FLASH"           -->
<!--       Register "STATUS"   -->
```

<destroy_module>

Removes a tree but not its content. This is different from **<remove>**, which deletes the tree and all its subtrees and subcomponents.

Supported elements: module

Structure of <destroy_module> tag:

```
<select element="module" property="name" regex="MODULE_NAME" />
<destroy_module/>
```

<include>

Include another rules files at the current postion.

Structure of <include> tag:

```
<include path="file_path"/>
```

<include_module>

Adds an %include command to the generated .p/.ph/.per file.

Supported elements: module

Structure of <include_module> tag:

```
<include_module name="name_of_tree" view_name="view name of tree"
description="description_of_tree" path="file path" offset="address"
args="arguments" is_open="yes|no" position="pos_mode"/>
```

name	Name of the module. Used to be referenced by the **<select>** command.
view name	Surround %include command by a **TREE**.
description	Tooltip of new tree.
path	Filename and path of the file to include.
is_open	• no: Default. Created module will be expanded. • yes: Created module will be collapsed.
position	• sorted: If sorting of top/subtrees is enabled, the new module will be positioned accordingly. • top: Default. Place module at the top of the file. • bottom: Place module at the bottom of the file.
offset	Add a **BASE** command in front of the module.
args	Arguments to pass to %include file..

Typically %include commands are used to include CPU-specific module files. The example below demonstrates how to surround the included module by a conditional **SIF**:

```
<select element="include_module" property="name" regex="MyModule"/>
<modify element="module" property="condition" value="SIF CPUIS(MyCPU)"/>
```

<open_module>

By default all converters will create closed trees (**TREE**.close). Using this command you can create opened trees (**TREE**.OPEN).

Supported elements: module

Structure of <open_module> tag:

```
<open_module depth=<depth> element=<module|mixed|all>/>
```

depth(optional)	Start with selected module and iterate over all submodules until *depth* levels. Default: Unlimited
module	Apply rule only if (sub)module has no other submembers than modules.
mixed	Apply rule only if (sub)module has no other submembers than modules and registers.
all	Apply rule always ((Sub)modules can have registers as only submembers.)

<modify>

Changes chosen property of an element.

Supported elements: all

Structure of <modify> tag:

```
<modify element="element_type" property="property_name" regex="reg_expr"
value="NEW_VALUE" />
```

<replace>

Replaces all found elements to new ones defined in <replace>.

Supported elements: all, register

Structure of <replace> tag:

```
<replace>
    <!-- <state> or <field> or <register> or <module> or -->
    <!-- <states> or <fields> or <registers> or <modules> -->
</replace>
```

state	<replace> <state> <name>state_name</name> <value>number</value> </state> </replace>
states	<replace> <states> <state><!-- ... --></state> <!-- other states... --> </states> </replace>
field	<replace> <field> <name>field_name</name> <description>field_description</description> <access>access_type_value</access> <lower_range>number</lower_range> <upper_range>number</upper_range> <states><!-- ... --></states> </field> </replace>
fields	<replace> <fields> <field><!-- ... --></field> <!-- other fields... --> </fields> </replace>
register	<replace> <register> <name>register_name</name> <description>register_description</description> <access>access_type_value</access> <offset>hex_number</offset> <size>number</size> <intrusive_read>yes|no</intrusive_read> <fields><!-- ... --></fields> </register> </replace>
registers	<replace> <registers> <register><!-- ... --></register> <!-- other registers... --> </registers> </replace>

module	<replace> <module> <name>module_name</name> <description>module_description</description> <is_open>yes|no</is_open> <registers><!-- ... --></registers> </module> </replace>
modules	<replace> <modules> <module><!-- ... --></module> <!-- other modules... --> </modules> </replace>
if (register only)	<replace> <if> <condition value="condition_of_practices_if_statement"> <register><!-- ... --></register> </condition> <!-- other conditions... --> <default> <register><!-- ... --></register> </default> </if> </replace>

Above listing show several ways of using <replace>. Choice between element and subelement depends on how <select> had been used. Let's see what will happen with following register:

```
group.long 0x00++0x03
  line.long 0x00 "REG,Test Register"
    bitfld.long 0x00 1. "FLD1,Field 1" "0,1"
    bitfld.long 0x00 0. "FLD0,Field 0" "0,1"
```

If selected element/elements comes directly from <select>, then there must be a definition of a single element (the type must be the same with selected elements) in <replace>. Following listing shows this case:

```
<rule>
    <select element="register" property="name" regex="REG" />
    <select element="field" property="name" regex="FLD1" />
    <replace>
        <field>
            <name>FEATURE_EN</name>
            <description>Featue Enable</description>
            <access>RW</access>
            <lower_range>1</lower_range><upper_range>1</upper_range>
            <states>
                <state><name>Disabled</name><value>0</value></state>
                <state><name>Enabled</name><value>1</value></state>
            </states>
        </field>
    </replace>
</rule>
```

The register after applying first rule:

```
group.long 0x00++0x03
  line.long 0x00 "REG,Test Register"
    bitfld.long 0x00 1. "FEATURE_EN,Feature Enable" "Disabled,Enabled"
    bitfld.long 0x00 0. "FLD0,Field 0" "0,1"
```

However if subelements had been extruded from selected elements, then <replace> must contain the definition of the element's group (type must match). Look below for this case:

```
<rule>
    <select element="register" property="name" regex="REG" />
    <replace element="field">
        <fields>
            <field>
                <name>FEATURE_EN</name>
                <description>Featue Enable</description>
                <access>RW</access>
                <lower_range>1</lower_range>
                <upper_range>1</upper_range>
                <states>
                    <state><name>Disabled</name><value>0</value></state>
                    <state><name>Enabled</name><value>1</value></state>
                </states>
            </field>
            <field>
                <name>STATUS</name>
                <description>Status</description>
                <access>RO</access>
                <lower_range>0</lower_range>
                <upper_range>0</upper_range>
                <states>
                    <state><name>Normal</name><value>0</value></state>
                    <state><name>Error</name><value>1</value></state>
                </states>
            </field>
        </fields>
    </replace>
</rule>
```

The register after applying second rule:

```
group.long 0x00++0x03
  line.long 0x00 "REG,Test Register"
    bitfld.long 0x00 1. "FEATURE_EN,Feature Enable" "Disabled,Enabled"
    bitfld.long 0x00 0. "STATUS,Status" "Normal,Error"
```

When using the <if> tag, only <register> is allowed as subtag. This is a special case which creates view conditions for a given register. It may look similar to the example below::

```
<rule>
    <select element="register" property="name" regex="REG" />
    <replace>
    <if>
    <condition value="Data.Long(D:0x04)==0x01">
        <register>
        <name>REG</name>
        <description>Test Register</description>
        <access>RW</access>
        <offset>0x00</offset>
        <size>4</size>
        <intrusive_read>no</intrusive_read>
        <fields>
            <field>
                <name>FEATURE_EN</name>
                <description>Featue Enable</description>
                <access>RW</access>
                <lower_range>1</lower_range>
                <upper_range>1</upper_range>
                <states>
                    <state><name>Disabled</name><value>0</value></state>
                    <state><name>Enabled</name><value>1</value></state>
                </states>
            </field>
            <field>
                <name>STATUS</name>
                <description>Status</description>
                <access>RO</access>
                <lower_range>0</lower_range>
                <upper_range>0</upper_range>
                <states>
                    <state><name>Normal</name><value>0</value></state>
                    <state><name>Error</name><value>1</value></state>
                </states>
            </field>
        </fields>
        </register>
    </condition>
    </if>
    </replace>
</rule>
```

The register after applying third rule:

```
if (Data.Long(D:0x04)==0x01)
  group.long 0x00++0x03
    line.long 0x00 "REG,Test Register"
      bitfld.long 0x00 1. "FEATURE_EN,Feature Enable" "Disabled,Enabled"
      bitfld.long 0x00 0. "STATUS,Status" "Normal,Error"
endif
```

<protect>

In some registers there are bit fields that can only be changed when another bit is written '1' at the same time. Such bit fields are called "protected". To ease changing such bit fields by the peripheral file, one should keep protection bits connected together into one bit field, with their values being "write protect/write enable". Value descriptions should signal the state in which it is possible to alter the value of a secured register, for instance: "Set value".

Supported elements: register

Structure of <protect> tag (Creates protected field if the specified field and protector are found.):

```
<protect>
    <field regex="name_regex" />
    <protected_by regex="name_regex" />
</protect>
```

Structure of <protect> tag (Finds field with given prefix or suffix, then tries to find field that is protected by first one and then creates protected field from them.):

```
<protect>
    <common prefix="prefix_string" suffix="suffix_string" />
</protect>
```

<remove>

In some situations it is necessary to remove elements, e.g. confidential modules, registers or fields. Then the command below should be used.

Structure of <remove> tag:

```
<remove />
```

For a better useability, it is often required to place the registers of similar purpose in separate subtree. The <create_module> command creates a new subtree and moves enclosed trees/registers into it.

Supported elements: module

Structure of <create_module> tag:

```
<create_module name="name_of_tree" description="description_of_tree"
is_open="yes|no" mode="mode_name" position="pos_mode">
    <element property="name|description" regex="REGA_([0-9]*)" />
    <element property="name|description" regex="REGB_([0-9]*)" />
</create_module>
```

name	Name of new tree.
description	Tooltip of new tree.
is_open	• no: Default. Created module will be expanded. • yes: Created module will be collapsed.
mode	• single: Default. Create one tree for all matched groups of elements. • multi: Create separate trees for each matched groups of elements. Names will be numerated.
position	• inplace: Default. Place trees in position where elemet has been found. • top: Place trees on the top of module. • bottom: Place trees on the bottom of module.
property	Finds element by name or description.
regex	Regular expression.

<for>

In case when groups of registers occurrence in several channels then you have to create subtree for each channel separately (e.g. REG_0_A, REG_0_B, REG_1_A, REG_1_B, ...). To avoid redundant commands you may use the <for> tag. <for> tags can be nested.

Supported elements: module

Structure of <for> tag:

```
<for iter_name="name" min_value="value" max_value="value">
    <create_module><!-- command definition --></create_module>
    <for><!-- ... --></for>
    <!-- other <create_module> or <for> commands...
</for>
```

Index of the for in module's name attribute and element's regex attribute is allowed. Index must be placed in #{} brackets. Is possible to change format of index value. Syntax accepts all C-like format specifiers (d, x, etc.). Format must be placed after index name and ":" separator.

Example:

```
<for iter_name="i" min_value="0" max_value="15">
    <create_module name="Module #{i:u}">
        <element regex="Reg#{i:u}_*" />
    </create_module>
</for>
```

<create_view>

Some peripherals, e.g. an Ethernet Controller, may have different operating modes. Depending on the mode, registers and their bitfields may have different meanings and encodings. <create_view> creates a view with an alternative register element depending on the assigned condition. Created views can be selected with <select> command and changed with <modify> command. A view corresponds to the **IF** statement in peripheral files.

Supported elements: register, module

Structure of <create_view> tag:

```
<create_view view_name="view_name" if="practice_condition|default"
use="name_of_register" append=”name_of_other_view”/>
```

practice_condition	Practice condition.
default	Can be used to generate ‘else’ clause.
append	Optional attribute which appends this view to a previously defined view. The **IF** statement in the referenced will be turned into an **ELIF** as a result.
use	Optional attribute that allows to create a view from an existing register. Used register will be removed. Regular expression of register's short name must be place here.

Offset's param usage in <create_view>:

```
<create_view view_name="view1"
if="(per.long(D:#{offset:x})&amp;0x800)==0x800" />
<!-- &amp; expands to '&' -->
```

Reference of register's offset in if attribute is allowed. Index must be placed in #{} brackets. Is possible to change format of offset value. Syntax accepts all C-like format specifiers (d, x, etc.). Format must be placed after offset word and ":" separator. For example "#{offset:d}".

NOTE:	When applied to a module, <create_view> will not include the **TREE** statement in the **IF** condition.

<map_cpu>

In case to change cpu name in sif conditions you can do it by selecting component via condition and then use <modify> to change condition property. Instead of that you can create a cpu_map where all the conditions under selected component that uses regex value will be replaced.

Supported elements: sif

Structure of <map_cpu> tag:

```
<map_cpu regex="cpu_to_replace" value="new_cpus_separated_by_coma"/>
```

NOTE:	It is recommended to select permodel using <select> and its attribute element="permodel".

Example:

```
<select element="permodel" />
<map_cpu regex="ComputeCluster_*_*" value="CortexA15,CortexA15A7"/>
```

Variables can be used to save certain properties of elements and use them later in rules.

The syntax is nearly the same as for rules. Simply replace the <rule> tag by a <variable> tag, do the <select>ions and define the **property** you want to save by the <get> command:

```
<variable name="variable_name">
    <select element=... />
    <select element=.../>
    <get property="property"
</variable>
```

The variable can later be referenced in a <rule> via:

```
#{variable_name[position]:format}
```

variable_name	Name of the variable.
position	If multiple elements have been selected with all_occurences="yes", the variable will be created as an array. With the **position** you can select a single array entry.
format	C-like format: • s = string • i = decimal • d = decimal. • x = hexadecimal • X = hexadecimal

Example:

```
<variable name="bank_base">
    <select element="module" property="name" regex="bank0"/>
    <get property="address" />
</variable>

<rule>
    <select element="module" property="name" regex"bank1" />
    <modify property="address" value="#{bank_base:x}" />
</rule>
```

Properties	
Target Namespace	https://www.lauterbach.com/per-converter/rules
Element and Attribute Namespaces	

The annotation file concept implies the existence of peripheral data to which it will refer and update. The reference to components of data is obtained by using regular expressions.
The more general the expression, the more universal the rule, and the more it may also work for another peripheral data.
There are several ways to update data's components. All of them are listed in the Commands section.

Declared Namespaces	
Prefix	Namespace
Default namespace	https://www.lauterbach.com/per-converter/rules
xml	http://www.w3.org/XML/1998/namespace
xs	http://www.w3.org/2001/XMLSchema

Schema Component Representation

```
<xs:schemaelementFormDefault"qualified"
targetNamespace"https://www.lauterbach.com/per-converter/rules" >
...
</xs:schema>
```

Element: create_header

Properties	
Name	create_header
Type	Locally-defined complex type
Nillable	no
Abstract	no

Configures the header.

Special placeholders:

- `<chip>` - Is replaced with chip list.
- `<author>` - Is replaced with author name if specified.
- `<version>` - Is replaced with current TRACE32 version.
- `<date>` - Is replaced with current date.

XML Instance Representation

```
<create_header
  title"xs:string" [0..1]
  props"props_type" [1]
  author"xs:string" [1]
  changelog"xs:string" [0..1]
  manufacturer"xs:string" [0..1]
  doc"xs:string" [0..1]
  core"xs:string" [0..1]
  chip"xs:string" [0..1]
  copyright"xs:string" [0..1]
  include"xs:string" [0..1]
/>
```

Schema Component Representation

```
<xs:elementname"create_header" >
   <xs:complexType>
      <xs:attributename"title" type"xs:string" use"optional" />
      <xs:attributename"props" type"props_type" use"required" />
      <xs:attributename"author" type"xs:string" use"required" />
      <xs:attributename"changelog" type"xs:string" use"optional" />
      <xs:attributename"manufacturer" type"xs:string" use"optional" />
      <xs:attributename"doc" type"xs:string" use"optional" />
      <xs:attributename"core" type"xs:string" use"optional" />
      <xs:attributename"chip" type"xs:string" use"optional" />
      <xs:attributename"copyright" type"xs:string" use"optional" />
      <xs:attributename"include" type"xs:string" use"optional" />
   </xs:complexType>
</xs:element>
```

Element: create_module

Properties	
Name	create_module
Type	Locally-defined complex type
Nillable	no
Abstract	no

Creates new sub tree and moves trees/registers to them according to given template.

XML Instance Representation

```
<create_module
  name"xs:string" [0..1]
  description"xs:string" [0..1]
  is_open"bool" [0..1]
  mode"create_module_mode" [0..1]
  position"create_module_position" [0..1]
>
   <element
     property"property_type" [0..1]
     regex"xs:string" [0..1]
/>[1..*]
</create_module>
```

Schema Component Representation

```
<xs:elementname"create_module" >
   <xs:complexType>
      <xs:sequence>
         <xs:elementname"element" maxOccurs"unbounded" >
            <xs:complexType>
               <xs:attributename"property" type"property_type" />
               <xs:attributename"regex" type"xs:string" />
            </xs:complexType>
         </xs:element>
      </xs:sequence>
      <xs:attributename"name" type"xs:string" />
      <xs:attributename"description" type"xs:string" />
      <xs:attributename"is_open" type"bool" />
      <xs:attributename"mode" type"create_module_mode" />
      <xs:attributename"position" type"create_module_position" />
   </xs:complexType>
</xs:element>
```

Element: create_view

Properties	
Name	create_view
Type	Locally-defined complex type
Nillable	no
Abstract	no

Creates if statements to the selected register.

XML Instance Representation

```
<create_view
  view_name"xs:string" [1]
  if"if_type" [1]
  use"xs:string" [0..1]
  append"xs:string" [0..1]
/>
```

Schema Component Representation

```
<xs:elementname"create_view" >
   <xs:complexType>
      <xs:attributename"view_name" type"xs:string" use"required" />
      <xs:attributename"if" type"if_type" use"required" />
      <xs:attributename"use" type"xs:string" use"optional" />
      <xs:attributename"append" type"xs:string" use"optional" />
   </xs:complexType>
</xs:element>
```

Element: derive_module

Properties	
Name	derive_module
Type	Locally-defined complex type
Nillable	no
Abstract	no

Creates trees based on tree name and its separator.

XML Instance Representation

```
<derive_module
  separator"xs:string" [1]
  depth"number" [0..1]
  element"derive_module_element" [0..1]
  preserve"bool" [0..1]
/>
```

Schema Component Representation

```
<xs:elementname"derive_module" >
   <xs:complexType>
      <xs:attributename"separator" type"xs:string" use"required" />
      <xs:attributename"depth" type"number" use"optional" />
      <xs:attributename"element" type"derive_module_element" use"optional" />
      <xs:attributename"preserve" type"bool" use"optional" />
   </xs:complexType>
</xs:element>
```

Element: destroy_module

Properties	
Name	destroy_module
Type	Locally-defined complex type
Nillable	no
Abstract	no

XML Instance Representation

```
<destroy_module/>
```

Schema Component Representation

```
<xs:elementname"destroy_module" >
   <xs:complexType>
      <xs:sequence/>
   </xs:complexType>
</xs:element>
```

Element: field

Properties	
Name	field
Type	Locally-defined complex type
Nillable	no
Abstract	no

XML Instance Representation

```
<field>
   <name>xs:string </name>[1]
   <description>xs:string </description>[1]
   <access>access_type</access>[1]
   <lower_range>number</lower_range>[1]
   <upper_range>number</upper_range>[1]
   <states... </states[1]
</field>
```

Schema Component Representation

```
<xs:elementname"field" >
   <xs:complexType>
      <xs:sequence>
         <xs:elementname"name" type"xs:string" minOccurs"1" />
         <xs:elementname"description" type"xs:string" minOccurs"1" />
         <xs:elementname"access" type"access_type" minOccurs"1" />
         <xs:elementname"lower_range" type"number" minOccurs"1" />
         <xs:elementname"upper_range" type"number" minOccurs"1" />
         <xs:elementref"states" minOccurs"1" />
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

Element: fields

Properties	
Name	fields
Type	Locally-defined complex type
Nillable	no
Abstract	no

XML Instance Representation

```
<fields>
   <field... </field[0..*]
</fields>
```

Schema Component Representation

```
<xs:elementname"fields" >
   <xs:complexType>
      <xs:sequence>
         <xs:elementref"field" minOccurs"0" maxOccurs"unbounded" />
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

Element: for

Properties	
Name	for
Type	Locally-defined complex type
Nillable	no
Abstract	no

Creates sequence of <create_module> commands.

*Index of the for in module's `name` attribute and element's `regex` attribute is allowed.
Index must be placed in `#{}` brackets. Is possible to change format of index value.
Syntax C-like format specifiers (see available [format](#type_format_type)). Format must be placed after index name and ":" separator.
For example `"Module #{i:d}"`.*

XML Instance Representation

```
<for
  iter_name"xs:QName" [0..1]
  min_value"number" [0..1]
  max_value"number" [0..1]
  description"xs:string" [0..1]
>
   <create_module... </create_module[0..*]
   <for... </for[0..*]
</for>
```

Schema Component Representation

```
<xs:elementname"for" >
   <xs:complexType>
      <xs:sequence>
         <xs:elementref"create_module" minOccurs"0" maxOccurs"unbounded" />
         <xs:elementref"for" maxOccurs"unbounded" minOccurs"0" />
      </xs:sequence>
      <xs:attributename"iter_name" type"xs:QName" />
      <xs:attributename"min_value" type"number" />
      <xs:attributename"max_value" type"number" />
      <xs:attributename"description" type"xs:string" />
   </xs:complexType>
</xs:element>
```

Element: get

Properties	
Name	get
Type	Locally-defined complex type
Nillable	no
Abstract	no

Defines property that will be saved

XML Instance Representation

```
<get
  property"property_type" [1]
/>
```

Schema Component Representation

```
<xs:elementname"get" >
   <xs:complexType>
      <xs:attributename"property" type"property_type" use"required" />
   </xs:complexType>
</xs:element>
```

Properties	
Name	if
Type	Locally-defined complex type
Nillable	no
Abstract	no

XML Instance Representation

```
<if>
   <condition
     value"xs:string" [0..1]
   >[1..*]
      <register... </register[1..*]
   </condition>
   <default   >[0..1]
      <register... </register[1..*]
   </default>
</if>
```

Schema Component Representation

```
<xs:elementname"if" >
   <xs:complexType>
      <xs:sequence>
         <xs:elementname"condition" minOccurs"1" maxOccurs"unbounded" >
            <xs:complexType>
               <xs:sequence>
                 <xs:elementref"register" minOccurs"1" maxOccurs"unbounded"
/>
               </xs:sequence>
               <xs:attributename"value" type"xs:string" />
            </xs:complexType>
         </xs:element>
         <xs:elementname"default" minOccurs"0" >
            <xs:complexType>
               <xs:sequence>
                 <xs:elementref"register" minOccurs"1" maxOccurs"unbounded"
/>
               </xs:sequence>
            </xs:complexType>
         </xs:element>
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

Element: include

Properties	
Name	include
Type	Locally-defined complex type
Nillable	no
Abstract	no

Defines either an absolute or relative path to a file to be included.

XML Instance Representation

```
<include
  path"xs:string" [1]
/>
```

Schema Component Representation

```
<xs:elementname"include" >
   <xs:complexType>
      <xs:attributename"path" type"xs:string" use"required" />
   </xs:complexType>
</xs:element>
```

Element: include_module

Properties	
Name	include_module
Type	Locally-defined complex type
Nillable	no
Abstract	no

Creates %include command.

XML Instance Representation

```
<include_module
  name"xs:string" [1]
  path"xs:string" [1]
  offset"xs:string" [0..1]
  position"include_module_position" [0..1]
  view_name"xs:string" [0..1]
  args"xs:string" [0..1]
  description"xs:string" [0..1]
  is_open"bool" [0..1]
  type"include_type" [0..1]
/>
```

Schema Component Representation

```
<xs:elementname"include_module" >
   <xs:complexType>
      <xs:attributename"name" type"xs:string" use"required" />
      <xs:attributename"path" type"xs:string" use"required" />
      <xs:attributename"offset" type"xs:string" use"optional" />
      <xs:attributename"position" type"include_module_position" use"optional"
/>
      <xs:attributename"view_name" type"xs:string" use"optional" />
      <xs:attributename"args" type"xs:string" use"optional" />
      <xs:attributename"description" type"xs:string" use"optional" />
      <xs:attributename"is_open" type"bool" use"optional" />
      <xs:attributename"type" type"include_type" use"optional" />
   </xs:complexType>
</xs:element>
```

Element: map_cpu

Properties	
Name	map_cpu
Type	Locally-defined complex type
Nillable	no
Abstract	no

Changes cpu name in sif conditions to a new `value`.

XML Instance Representation

```
<map_cpu
  regex"xs:string" [1]
  value"xs:string" [1]
/>
```

Schema Component Representation

```
<xs:elementname"map_cpu" >
   <xs:complexType>
      <xs:attributename"regex" type"xs:string" use"required" />
      <xs:attributename"value" type"xs:string" use"required" />
   </xs:complexType>
</xs:element>
```

Element: modify

Properties	
Name	modify
Type	Locally-defined complex type
Nillable	no
Abstract	no

Changes choosen property of element to `NEW_VALUE`.

XML Instance Representation

```
<modify
  element"element_type" [0..1]
  property"property_type" [0..1]
  regex"xs:string" [0..1]
  value"xs:string" [0..1]
/>
```

Schema Component Representation

```
<xs:elementname"modify" >
   <xs:complexType>
      <xs:attributename"element" type"element_type" />
      <xs:attributename"property" type"property_type" />
      <xs:attributename"regex" type"xs:string" />
      <xs:attributename"value" type"xs:string" />
   </xs:complexType>
</xs:element>
```

Element: module

Properties	
Name	module
Type	Locally-defined complex type
Nillable	no
Abstract	no

XML Instance Representation

```
<module>
   <name>xs:string </name>[1]
   <description>xs:string </description>[1]
   <address>number</address>[1]
   <is_open>bool</is_open>[1]
   <registers... </registers[1]
</module>
```

Schema Component Representation

```
<xs:elementname"module" >
   <xs:complexType>
      <xs:sequence>
         <xs:elementname"name" type"xs:string" minOccurs"1" />
         <xs:elementname"description" type"xs:string" minOccurs"1" />
         <xs:elementname"address" type"number" minOccurs"1" />
         <xs:elementname"is_open" type"bool" minOccurs"1" />
         <xs:elementref"registers" minOccurs"1" />
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

Element: modules

Properties	
Name	modules

Type	Locally-defined complex type
Nillable	no
Abstract	no

XML Instance Representation

```
<modules>
   <module... </module[0..*]
</modules>
```

Schema Component Representation

```
<xs:elementname"modules" >
   <xs:complexType>
      <xs:sequence>
         <xs:elementref"module" minOccurs"0" maxOccurs"unbounded" />
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

Element: open_module

Properties	
Name	open_module
Type	Locally-defined complex type
Nillable	no
Abstract	no

Adds `.open` option to selected tree

XML Instance Representation

```
<open_module
  depth"number" [0..1]
  element"open_module_element" [1]
/>
```

Schema Component Representation

```
<xs:elementname"open_module" >
   <xs:complexType>
      <xs:attributename"depth" type"number" use"optional" />
      <xs:attributename"element" type"open_module_element" use"required" />
   </xs:complexType>
</xs:element>
```

Element: protect

Properties	
Name	protect
Type	Locally-defined complex type
Nillable	no
Abstract	no

Can be used in two different sequences:

- using `field` and `protected_by` - Creates protected field if the specified field and protector are found.
- using `common` - Finds field with given prefix or suffix, then tries to find field that is protected by first one and then creates protected field from them.

XML Instance Representation

```
<protect>
   Start Choice[1]
      <field>protect_field_type</field>[0..1]
      <protected_by>protect_field_type</protected_by>[0..1]
      <common>protect_common_type</common>[0..1]
   End Choice
</protect>
```

Schema Component Representation

```
<xs:elementname"protect" >
   <xs:complexType>
      <xs:choice>
         <xs:sequence>
            <xs:elementname"field" type"protect_field_type" minOccurs"0"
maxOccurs"1" />
          <xs:elementname"protected_by" type"protect_field_type" minOccurs"0"
maxOccurs"1" />
         </xs:sequence>
         <xs:sequence>
            <xs:elementname"common" type"protect_common_type" minOccurs"0"
maxOccurs"1" />
         </xs:sequence>
      </xs:choice>
   </xs:complexType>
</xs:element>
```

Element: register

Properties	
Name	register
Type	Locally-defined complex type
Nillable	no
Abstract	no

XML Instance Representation

```
<register>
   <name>xs:string </name>[1]
   <description>xs:string </description>[1]
   <access>access_type</access>[1]
   <offset>number</offset>[1]
   <size>number</size>[1]
   <intrusive_read>bool</intrusive_read>[1]
   <fields... </fields[1]
   <special>xs:string </special>[0..1]
   <port>number</port>[0..1]
   <dataread>number</dataread>[0..1]
   <datawrite>number</datawrite>[0..1]
</register>
```

Schema Component Representation

```
<xs:elementname"register" >
   <xs:complexType>
      <xs:sequence>
         <xs:elementname"name" type"xs:string" minOccurs"1" />
         <xs:elementname"description" type"xs:string" minOccurs"1" />
         <xs:elementname"access" type"access_type" minOccurs"1" />
         <xs:elementname"offset" type"number" minOccurs"1" />
         <xs:elementname"size" type"number" minOccurs"1" />
         <xs:elementname"intrusive_read" type"bool" minOccurs"1" />
         <xs:elementref"fields" minOccurs"1" />
         <xs:elementname"special" type"xs:string" minOccurs"0" />
         <xs:elementname"port" type"number" minOccurs"0" />
         <xs:elementname"dataread" type"number" minOccurs"0" />
         <xs:elementname"datawrite" type"number" minOccurs"0" />
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

Element: registers

Properties	
Name	registers
Type	Locally-defined complex type
Nillable	no
Abstract	no

XML Instance Representation

```
<registers>
   <register... </register[0..*]
</registers>
```

Schema Component Representation

```
<xs:elementname"registers" >
   <xs:complexType>
      <xs:sequence>
         <xs:elementref"register" minOccurs"0" maxOccurs"unbounded" />
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

Element: remove

Properties	
Name	remove
Type	Locally-defined complex type
Nillable	no
Abstract	no

XML Instance Representation

```
<remove/>
```

Schema Component Representation

```
<xs:elementname"remove" >
   <xs:complexType>
      <xs:sequence/>
   </xs:complexType>
</xs:element>
```

Element: replace

Properties	
Name	replace

Type	Locally-defined complex type
Nillable	no
Abstract	no

Replaces all found elements to new ones defined in `<replace>`.

XML Instance Representation

```
<replace
  element"element_type" [0..1]
>
   Start Group: replace_element_type[1..*]
      Start Choice[1]
         <module... </module[1]
         <modules... </modules[1]
         <register... </register[1]
         <registers... </registers[1]
         <field... </field[1]
         <fields... </fields[1]
         <state... </state[1]
         <states... </states[1]
         <if... </if[1]
      End Choice
   End Group: replace_element_type
</replace>
```

Schema Component Representation

```
<xs:elementname"replace" >
   <xs:complexType>
      <xs:sequence>
        <xs:groupref"replace_element_type" minOccurs"1" maxOccurs"unbounded"
/>
      </xs:sequence>
      <xs:attributename"element" type"element_type" />
   </xs:complexType>
</xs:element>
```

Element: rule

Properties	
Name	rule

Type	Locally-defined complex type
Nillable	no
Abstract	no

A rule definition contains one or more select tags `<select>` followed by one or more `<command>` tags `<command_name>`:

- The `<select>` tag defines on which elements the `<command>` tags will be applied. The first `<select>` tag will search in all elements of the XML file. The next `<select>` tag will search on the results of the previous select tag. That way, selecting the desired elements can be achieved by reducing the search base step by step.
- The `<command>` tag defines a modification that will be executed on the selected elements. Several `<command>` tags can be specified to apply independent modifications on the same search results.

XML Instance Representation

```
<rule>
   <select... </select[1..*]
   Start Group: commands[1..*]
      Start Choice[1]
         <modify... </modify[1]
         <replace... </replace[1]
         <protect... </protect[1]
         <remove... </remove[1]
         <create_module... </create_module[1]
         <for... </for[1]
         <create_view... </create_view[1]
         <map_cpu... </map_cpu[1]
         <destroy_module... </destroy_module[1]
         <include_module... </include_module[1]
         <derive_module... </derive_module[1]
         <open_module... </open_module[1]
         <create_header... </create_header[1]
      End Choice
   End Group: commands
</rule>
```

Schema Component Representation

```
<xs:elementname"rule" >
   <xs:complexType>
      <xs:sequence>
         <xs:elementref"select" minOccurs"1" maxOccurs"unbounded" />
         <xs:groupref"commands" minOccurs"1" maxOccurs"unbounded" />
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

Element: rules

Properties	
Name	rules
Type	Locally-defined complex type
Nillable	no
Abstract	no

The rule file consists of root tag `<rules>` and list of `<rule>` or `<variable>` tags inside of them. Nesting of tags is not allowed.

XML Instance Representation

```
<rules
  verbose"bool" [0..1]
>
   Start Choice[1..*]
      <rule... </rule[1]
      <variable... </variable[1]
      <include... </include[1]
   End Choice
</rules>
```

Schema Component Representation

```
<xs:elementname"rules" >
   <xs:complexType>
      <xs:choicemaxOccurs"unbounded" >
         <xs:elementref"rule" />
         <xs:elementref"variable" />
         <xs:elementref"include" />
      </xs:choice>
      <xs:attributename"verbose" type"bool" use"optional" />
   </xs:complexType>
</xs:element>
```

Element: select

Properties	
Name	select
Type	Locally-defined complex type
Nillable	no
Abstract	no

Selects targets elements to be processed by the commands. The selection is determined by the element type and its properties.

XML Instance Representation

```
<select
  element"element_type" [1]
  property"property_type" [0..1]
  regex"xs:string" [0..1]
  num_equal"xs:string" [0..1]
  all_occurrences"bool" [0..1]
  invert_regex"bool" [0..1]
  on_error"on_error_type" [0..1]
/>
```

Schema Component Representation

```
<xs:elementname"select" >
   <xs:complexType>
      <xs:attributename"element" type"element_type" use"required" />
      <xs:attributename"property" type"property_type" use"optional" />
      <xs:attributename"regex" type"xs:string" use"optional" />
      <xs:attributename"num_equal" type"xs:string" use"optional" />
      <xs:attributename"all_occurrences" type"bool" use"optional" />
      <xs:attributename"invert_regex" type"bool" use"optional" />
      <xs:attributename"on_error" type"on_error_type" use"optional" />
   </xs:complexType>
</xs:element>
```

Element: state

Properties	
Name	state
Type	Locally-defined complex type
Nillable	no
Abstract	no

XML Instance Representation

```
<state>
   <name>xs:string </name>[1]
   <value>number</value>[1]
</state>
```

Schema Component Representation

```
<xs:elementname"state" >
   <xs:complexType>
      <xs:sequence>
         <xs:elementname"name" type"xs:string" minOccurs"1" />
         <xs:elementname"value" type"number" minOccurs"1" />
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

Element: states

Properties	
Name	states
Type	Locally-defined complex type
Nillable	no
Abstract	no

XML Instance Representation

```
<states>
   <state... </state[0..*]
</states>
```

Schema Component Representation

```
<xs:elementname"states" >
   <xs:complexType>
      <xs:sequence>
         <xs:elementref"state" minOccurs"0" maxOccurs"unbounded" />
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

Element: variable

Properties	
Name	variable
Type	Locally-defined complex type
Nillable	no
Abstract	no

A variable definition contains one or more select tags `<select>` followed by one or more `<get>` tags:

- The `<select>` tag defines on which elements the `<command>` tags will be applied. The first `<select>` tag will search in all elements of the XML file. The next `<select>` tag will search on the results of the previous select tag. That way, selecting the desired elements can be achieved by reducing the search base step by step.
- The `<get>` tag defines a property value that will be saved under variable name.

NOTE: to refer to the variable use `#{variable_name[position]:format}`, where:

- variable_name - name of the variable
- [format](#type_format_type) - available C-like formats
- position (optional) - in_case of using all_occurrences while searching the values are stored in vector, use this to refer to proper value.

XML Instance Representation

```
<variable
  name"xs:string" [1]
>
   <select... </select[1..*]
   <get... </get[1..*]
</variable>
```

Schema Component Representation

```
<xs:elementname"variable" >
   <xs:complexType>
      <xs:sequence>
         <xs:elementref"select" minOccurs"1" maxOccurs"unbounded" />
         <xs:elementref"get" minOccurs"1" maxOccurs"unbounded" />
      </xs:sequence>
      <xs:attributename"name" type"xs:string" use"required" />
   </xs:complexType>
</xs:element>
```

Complex Type: protect_common_type

Type hierarchy	
Super-types:	None
Sub-types:	None

Properties	
Name	protect_common_type
Abstract	no

Finds field with given prefix or suffix, then tries to find field that is protected by first one and then creates protected field from them.

XML Instance Representation

```
<...
  prefix"xs:string" [1]
  suffix"xs:string" [1]
/>
```

Schema Component Representation

```
<xs:complexTypename"protect_common_type" >
   <xs:attributename"prefix" type"xs:string" use"required" />
   <xs:attributename"suffix" type"xs:string" use"required" />
</xs:complexType>
```

Complex Type: protect_field_type

Type hierarchy	
Super-types:	None
Sub-types:	None

Properties	
Name	protect_field_type
Abstract	no

Creates protected field if the specified field and protector are found.

XML Instance Representation

```
<...
  regex"xs:string" [1]
/>
```

Schema Component Representation

```
<xs:complexTypename"protect_field_type" >
   <xs:attributename"regex" type"xs:string" use"required" />
</xs:complexType>
```

Model Group: commands

Properties	
Name	commands

Commands are element modificators that process data sets extracted from data model by the `<select>` tag. One data set can be used by multiple commands. Command tags must follow the `<select>` tags.

XML Instance Representation

```
Start Choice[1]
   <modify... </modify[1]
   <replace... </replace[1]
   <protect... </protect[1]
   <remove... </remove[1]
   <create_module... </create_module[1]
   <for... </for[1]
   <create_view... </create_view[1]
   <map_cpu... </map_cpu[1]
   <destroy_module... </destroy_module[1]
   <include_module... </include_module[1]
   <derive_module... </derive_module[1]
   <open_module... </open_module[1]
   <create_header... </create_header[1]
End Choice
```

Schema Component Representation

```
<xs:groupname"commands" >
   <xs:choice>
      <xs:elementref"modify" />
      <xs:elementref"replace" />
      <xs:elementref"protect" />
      <xs:elementref"remove" />
      <xs:elementref"create_module" />
      <xs:elementref"for" />
      <xs:elementref"create_view" />
      <xs:elementref"map_cpu" />
      <xs:elementref"destroy_module" />
      <xs:elementref"include_module" />
      <xs:elementref"derive_module" />
      <xs:elementref"open_module" />
      <xs:elementref"create_header" />
   </xs:choice>
</xs:group>
```

Model Group: replace_element_type

Properties	
Name	replace_element_type

XML Instance Representation

```
Start Choice[1]
   <module... </module[1]
   <modules... </modules[1]
   <register... </register[1]
   <registers... </registers[1]
   <field... </field[1]
   <fields... </fields[1]
   <state... </state[1]
   <states... </states[1]
   <if... </if[1]
End Choice
```

Schema Component Representation

```
<xs:groupname"replace_element_type" >
   <xs:choice>
      <xs:elementref"module" />
      <xs:elementref"modules" />
      <xs:elementref"register" />
      <xs:elementref"registers" />
      <xs:elementref"field" />
      <xs:elementref"fields" />
      <xs:elementref"state" />
      <xs:elementref"states" />
      <xs:elementref"if" />
   </xs:choice>
</xs:group>
```

Simple Type: access_type

Type hierarchy	
Super-types:	xs:token < **access_type**(by restriction)
Sub-types:	None

Properties	
Name	access_type
Content	Base XSD Type: token, *value* comes from list: {'RW'|'RO'|'WO'|'W1C'|'WS'|'H'}

Available `access_type` values:

- `RW`: Read write (group, bitfld)
- `RO`: Read only (rgroup, rbitfld)
- `WO`: Write only (wgroup)
- `W1C`: Write one to clear (eventfld)
- `WS`: Write secured
- `H`: Hidden (hgroup)

Schema Component Representation

```
<xs:simpleTypename"access_type" >
   <xs:restrictionbase"xs:token" >
      <xs:enumerationvalue"RW" />
      <xs:enumerationvalue"RO" />
      <xs:enumerationvalue"WO" />
      <xs:enumerationvalue"W1C" />
      <xs:enumerationvalue"WS" />
      <xs:enumerationvalue"H" />
   </xs:restriction>
</xs:simpleType>
```

Simple Type: bool

Type hierarchy	
Super-types:	xs:string < **bool**(by restriction)
Sub-types:	None

Properties	
Name	bool
Content	Base XSD Type: string, *value* comes from list: {'no'|'yes'}

Similar to xs:boolean but with a 'no/yes' representation

Schema Component Representation

```
<xs:simpleTypename"bool" >
   <xs:restrictionbase"xs:string" >
      <xs:enumerationvalue"no" />
      <xs:enumerationvalue"yes" />
   </xs:restriction>
</xs:simpleType>
```

Simple Type: create_module_mode

Type hierarchy	
Super-types:	xs:token < **create_module_mode**(by restriction)
Sub-types:	None

Properties	
Name	create_module_mode
Content	Base XSD Type: token, *value* comes from list: {'single'|'multi'}

- `single` - Default. Create one tree for all matched groups of elements.
- `multi` - Create separate trees for each matched groups of elements. Names will be numerated.

Schema Component Representation

```
<xs:simpleTypename"create_module_mode" >
   <xs:restrictionbase"xs:token" >
      <xs:enumerationvalue"single" />
      <xs:enumerationvalue"multi" />
   </xs:restriction>
</xs:simpleType>
```

Simple Type: create_module_position

Type hierarchy	
Super-types:	xs:token < **create_module_position**(by restriction)
Sub-types:	None

Properties	
Name	create_module_position
Content	Base XSD Type: token, *value* comes from list: {'inplace'|'top'|'bottom'}

- `inplace` - Default. Place trees in position where elemet has been found.
- `top` - Place trees on the top of module.
- `bottom` - Place trees on the bottom of module.

Schema Component Representation

```
<xs:simpleTypename"create_module_position" >
   <xs:restrictionbase"xs:token" >
      <xs:enumerationvalue"inplace" />
      <xs:enumerationvalue"top" />
      <xs:enumerationvalue"bottom" />
   </xs:restriction>
</xs:simpleType>
```

Simple Type: derive_module_element

Type hierarchy	
Super-types:	xs:token < **derive_module_element**(by restriction)
Sub-types:	None

Properties	
Name	derive_module_element
Content	Base XSD Type: token, *value* comes from list: {'register'|'module'|'all'}

- `register` - Module contains registers only.
- `module` - Module contains submodules only.
- `all` - Module contains both modules and registers.

Schema Component Representation

```
<xs:simpleTypename"derive_module_element" >
   <xs:restrictionbase"xs:token" >
      <xs:enumerationvalue"register" />
      <xs:enumerationvalue"module" />
      <xs:enumerationvalue"all" />
   </xs:restriction>
</xs:simpleType>
```

Simple Type: element_type

Type hierarchy	
Super-types:	xs:token < **element_type**(by restriction)
Sub-types:	None

Properties	
Name	element_type
Content	Base XSD Type: token, *value* comes from list: {'permodel'|'module'|'register'|'field'|'state'|'include_module'}

- `permodel` - Top level element.
- `module` - Can be nested in other module.
- `register` -Can be nested in module.
- `field` -Can be nested in register.
- `state` -Can be nested in field.
- `include_module` - Can be nested in permodel.

Schema Component Representation

```
<xs:simpleTypename"element_type" >
   <xs:restrictionbase"xs:token" >
      <xs:enumerationvalue"permodel" />
      <xs:enumerationvalue"module" />
      <xs:enumerationvalue"register" />
      <xs:enumerationvalue"field" />
      <xs:enumerationvalue"state" />
      <xs:enumerationvalue"include_module" />
   </xs:restriction>
</xs:simpleType>
```

Type hierarchy	
Super-types:	xs:token < **format_type**(by restriction)
Sub-types:	None

Properties	
Name	format_type
Content	Base XSD Type: token, *value* comes from list: {'s'|'i'|'d'|'x'|'X'}

- `s` - saves string
- `i` - saves decimal
- `d` - saves decimal
- `x` - saves hexadecimal
- `X` - saves hexadecimal

Schema Component Representation

```
<xs:simpleTypename"format_type" >
   <xs:restrictionbase"xs:token" >
      <xs:enumerationvalue"s" />
      <xs:enumerationvalue"i" />
      <xs:enumerationvalue"d" />
      <xs:enumerationvalue"x" />
      <xs:enumerationvalue"X" />
   </xs:restriction>
</xs:simpleType>
```

Simple Type: if_type

Type hierarchy	
Super-types:	None
Sub-types:	None

Properties	
Name	if_type
Content	Union of following types: xs:token Locally defined type: Base XSD Type: token, *value* comes from list: {'default'}

- `practice_condition` - PRACTICE condition syntax.
- `default` - Can be used to generate else clause.

Schema Component Representation

```
<xs:simpleTypename"if_type" >
   <xs:unionmemberTypes"xs:token" >
      <xs:simpleType>
         <xs:restrictionbase"xs:token" >
            <xs:enumerationvalue"default" />
         </xs:restriction>
      </xs:simpleType>
   </xs:union>
</xs:simpleType>
```

Simple Type: include_module_position

Type hierarchy	
Super-types:	xs:token < **include_module_position**(by restriction)
Sub-types:	None

Properties	
Name	include_module_position
Content	Base XSD Type: token, *value* comes from list: {'top'|'bottom'|'sorted'}

- `top` - Place %include on the top of module. (default)
- `bottom` - Place trees on the bottom of module.
- `sorted` - %include command will be sorted with `SortTopTrees` option.

Schema Component Representation

```
<xs:simpleTypename"include_module_position" >
   <xs:restrictionbase"xs:token" >
      <xs:enumerationvalue"top" />
      <xs:enumerationvalue"bottom" />
      <xs:enumerationvalue"sorted" />
   </xs:restriction>
</xs:simpleType>
```

Simple Type: include_type

Type hierarchy	
Super-types:	xs:token < **include_type**(by restriction)
Sub-types:	None

Properties	
Name	include_type
Content	Base XSD Type: token, *value* comes from list: {'%include'|'INCLUDE'}

Outputs proper include command

Schema Component Representation

```
<xs:simpleTypename"include_type" >
   <xs:restrictionbase"xs:token" >
      <xs:enumerationvalue"%include" />
      <xs:enumerationvalue"INCLUDE" />
   </xs:restriction>
</xs:simpleType>
```

Simple Type: number

Type hierarchy	
Super-types:	xs:string < **number**(by restriction)
Sub-types:	None

Properties	
Name	number
Content	Base XSD Type: string, *pattern* = 0x[0-9A-Fa-f]+|[0-9]+.?|0[b|y][01]+

The `number` type accepts either hexadecimal or decimal values.

Schema Component Representation

```
<xs:simpleTypename"number" >
   <xs:restrictionbase"xs:string" >
      <xs:patternvalue"0x[0-9A-Fa-f]+|[0-9]+.?|0[b|y][01]+" />
   </xs:restriction>
</xs:simpleType>
```

Simple Type: on_error_type

Type hierarchy	
Super-types:	xs:token < **on_error_type**(by restriction)
Sub-types:	None

Properties	
Name	on_error_type
Content	Base XSD Type: token, *value* comes from list: {'error'|'ignore'|'logfile'}

- `error (default)`: Current behavior, conversion process is aborted with an error message.
- `ignore`: Ignore error.
- `logfile`: Conversion process continues but error is printed to logfile. Same as ignore option if logfile is disabled.

Schema Component Representation

```
<xs:simpleTypename"on_error_type" >
   <xs:restrictionbase"xs:token" >
      <xs:enumerationvalue"error" />
      <xs:enumerationvalue"ignore" />
      <xs:enumerationvalue"logfile" />
   </xs:restriction>
</xs:simpleType>
```

Simple Type: open_module_element

Type hierarchy	
Super-types:	xs:token < **open_module_element**(by restriction)
Sub-types:	None

Properties	
Name	open_module_element
Content	Base XSD Type: token, *value* comes from list: {'module'|'mixed'|'all'}

-`module` - Only open tree if module does not have registers (only other submodules)*
-`mixed` - Only open tree if module has registers and submodules.*
-`all` - Open tree unconditionally*

The specified attribute supersedes another attribute. For instance, when using "element"="module" and "depth"="3" as an example, it means,
if there are registers at the top level, the sublevels at depths 2 and 3 will not be opened.

Schema Component Representation

```
<xs:simpleTypename"open_module_element" >
   <xs:restrictionbase"xs:token" >
      <xs:enumerationvalue"module" />
      <xs:enumerationvalue"mixed" />
      <xs:enumerationvalue"all" />
   </xs:restriction>
</xs:simpleType>
```

Type hierarchy	
Super-types:	xs:token < **property_type**(by restriction)
Sub-types:	None

Properties	
Name	property_type
Content	Base XSD Type: token, *value* comes from list: {'name'|'description'|'value'|'access_type'|'offset'|'size'|'lower_range'|'upper_range'|'intrusive_read'|'condition'|'view_name'|'path'|'button_name'|'button_command'|'address'|'access_class'|'special'|'port'|'dataread'|'datawrite'}

- `name` - Name of the element.
- `description` - Description of the element.
- `value` - State code value.
- `access_type` - Access type of the given element (e.g., read/write/...).
- `offset` - Address offset from the base address in hexadecimal.
- `size` - Size of the register in bytes.
- `lower_range` - Field's lower boundary.
- `upper_range` - Field's upper boundary.
- `intrusive_read` - Indicates whether reading the given register causes data loss or not.
- `condition` - Indicates the display condition of the given element.
- `view_name` - Indicates the reference name for the conditional view of the given element. It is used for view creation or selection for making changes.
- `path` - Path of the element based on element names.
- `button_name` - Button name of the given element.
- `button_command` - Button commands of the given element.
- `address` - Base address of the module.
- `access_class` - Access classification for the base address.
- `special` - Is register part of group saveindex, savetindex, index, tindex.
- `port` - Is the address of the reg addr register for saveindex, savetindex, index, and tindex commands.
- `dataread` - Index of the register within special group.
- `datawrite` - If unspecified, value is same as dataread.

Schema Component Representation

```
<xs:simpleTypename"property_type" >
   <xs:restrictionbase"xs:token" >
      <xs:enumerationvalue"name" />
      <xs:enumerationvalue"description" />
      <xs:enumerationvalue"value" />
      <xs:enumerationvalue"access_type" />
      <xs:enumerationvalue"offset" />
      <xs:enumerationvalue"size" />
      <xs:enumerationvalue"lower_range" />
      <xs:enumerationvalue"upper_range" />
      <xs:enumerationvalue"intrusive_read" />
      <xs:enumerationvalue"condition" />
      <xs:enumerationvalue"view_name" />
      <xs:enumerationvalue"path" />
      <xs:enumerationvalue"button_name" />
      <xs:enumerationvalue"button_command" />
      <xs:enumerationvalue"address" />
      <xs:enumerationvalue"access_class" />
      <xs:enumerationvalue"special" />
      <xs:enumerationvalue"port" />
      <xs:enumerationvalue"dataread" />
      <xs:enumerationvalue"datawrite" />
   </xs:restriction>
</xs:simpleType>
```

Simple Type: props_type

Type hierarchy	
Super-types:	xs:token < **props_type**(by restriction)
Sub-types:	None

Properties	
Name	props_type
Content	Base XSD Type: token, *value* comes from list: {'Confidential'|'Released'|'Strictly-confidential'}

Schema Component Representation

```
<xs:simpleTypename"props_type" >
   <xs:restrictionbase"xs:token" >
      <xs:enumerationvalue"Confidential" />
      <xs:enumerationvalue"Released" />
      <xs:enumerationvalue"Strictly-confidential" />
   </xs:restriction>
</xs:simpleType>
```

Abstract(Applies to complex type definitions and element declarations). An abstract element or complex type cannot used to validate an element instance. If there is a reference to an abstract element, only element declarations that can substitute the abstract element can be used to validate the instance. For references to abstract type definitions, only derived types can be used.

All Model GroupChild elements can be provided *in any order*in instances. See: http://www.w3.org/TR/xmlschema-1/#element-all.

Choice Model Group*Only one*from the list of child elements and model groups can be provided in instances. See: http://www.w3.org/TR/xmlschema-1/#element-choice.

Collapse Whitespace PolicyReplace tab, line feed, and carriage return characters with space character (Unicode character 32). Then, collapse contiguous sequences of space characters into single space character, and remove leading and trailing space characters.

Disallowed Substitutions(Applies to element declarations). If *substitution* is specified, then substitution groupmembers cannot be used in place of the given element declaration to validate element instances. If *derivation methods* , e.g. extension, restriction, are specified, then the given element declaration will not validate element instances that have types derived from the element declaration's type using the specified derivation methods. Normally, element instances can override their declaration's type by specifying an `xsi:type`attribute.

Key ConstraintLike Uniqueness Constraint, but additionally requires that the specified value(s) must be provided. See: http://www.w3.org/TR/xmlschema-1/#cIdentity-constraint_Definitions.

Key Reference ConstraintEnsures that the specified value(s) must match value(s) from a Key Constraintor Uniqueness ConstraintSee: http://www.w3.org/TR/xmlschema-1/#cIdentity-constraint_Definitions.

Model GroupGroups together element content, specifying the order in which the element content can occur and the number of times the group of element content may be repeated. See: http://www.w3.org/TR/xmlschema-1/#Model_Groups.

Nillable(Applies to element declarations). If an element declaration is nillable, instances can use the `xsi:nil`attribute. The `xsi:nil`attribute is the boolean attribute, *nil* , from the *http://www.w3.org/2001/XMLSchema-instance* namespace. If an element instance has an `xsi:nil`attribute set to true, it can be left empty, even though its element declaration may have required content.

NotationA notation is used to identify the format of a piece of data. Values of elements and attributes that are of type, NOTATION, must come from the names of declared notations. See: http://www.w3.org/TR/xmlschema-1/#cNotation_Declarations.

Preserve Whitespace PolicyPreserve whitespaces exactly as they appear in instances.

Prohibited Derivations(Applies to type definitions). Derivation methods that cannot be used to create sub-types from a given type definition.

Prohibited Substitutions(Applies to complex type definitions). Prevents sub-types that have been derived using the specified derivation methods from validating element instances in place of the given type definition.

Replace Whitespace PolicyReplace tab, line feed, and carriage return characters with space character (Unicode character 32).

Sequence Model GroupChild elements and model groups must be provided *in the specified order* in instances. See: http://www.w3.org/TR/xmlschema-1/#element-sequence.

Substitution GroupElements that are *members* of a substitution group can be used wherever the *head* element of the substitution group is referenced.

Substitution Group Exclusions(Applies to element declarations). Prohibits element declarations from nominating themselves as being able to substitute a given element declaration, if they have types that are derived from the original element's type using the specified derivation methods.

Target NamespaceThe target namespace identifies the namespace that components in this schema belongs to. If no target namespace is provided, then the schema components do not belong to any namespace.

Uniqueness ConstraintEnsures uniqueness of an element/attribute value, or a combination of values, within a specified scope. See: http://www.w3.org/TR/xmlschema-1/#cIdentity-constraint_Definitions.

<location> Invalid attribute *<attribute>* in tag *<name>*

Unknown attribute occured for parent tag.

Example:

```
<rules xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="rules.xsd">
    <rule>
        <select unknown="unknown" element="module" property="name"
regex="MODULE" all_occurrences="yes" invert_regex="yes" />
        <remove />
    </rule>
</rules>
```

test.xml:5:120 Invalid attribute unknown in tag <select>

Fix:
Remove the attribute from file or report the problem to developer.

<location> Invalid node *<node>* in tag *<name>*

Unknown node occurred for parent tag.

Example:

```
<rules xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="rules.xsd">
    <rule>
        <select element="module" property="name" regex="CSCU" />
        <unknown_command name="ACCEN" position="bottom">
            <element regex="CSCU_ACCEN_.*" />
       </unknown_command>
    </rule>
</rules>
```

test.xml:5:47 Invalid node 'unknown_command' in tag '<rule>'

Fix:
Check spelling or report the problem to developer.

<location> Invalid value *<value>* in tag *<name>*

Unknown value occurred in node or attribute.

Example:

```
...
<enumeratedValue>
    <name>DISABLED</name>
    <description>N/A</description>
    <value>UnknownValue</value>
</enumeratedValue>
...
```

cyt2b7.svd:3208:26 Invalid value 'UnknownValue' of 'value'

Fix:
Check spelling or report the problem to developer.

<location> *<name>* from *<name>* must occur only once

Node or attribute has occured more than once.

Example:

```
<rules xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="rules.xsd">
    <rule>
        <select element="module" element="register" property="name"
regex="CSCU"/>
    </rule>
</rules>
```

test.xml:4:50 'element' from 'select' must occur only once

Fix:
Remove duplicates or report the problem to developer.

<location> Missing *<name>* in tag *<name>*

Node or attribute is missing in parent node.

Example:

```
...
<device schemaVersion="1.3" xmlns:xs="http://www.w3.org/2001/XMLSchema-
instance"
xs:noNamespaceSchemaLocation="CMSIS-SVD.xsd">
    <vendor>Cypress Semiconductor</vendor>
    <vendorID>Cypress</vendorID>
    <name>cyt2b7</name>
    ...
    <peripherals>
    </peripherals>
</device>
...
```

cyt2b7.svd:29:12 Missing 'peripheral' in tag '<peripherals>'

Fix:
Add the missing element or report the problem to developer.

Invalid value *<value>* for property “property”

The right side of a property represents an invalid value.

Example:

```
<rules>
    <rule>
        <select element="module" property="name" regex="PERI" />
       <select element="register" property="name" regex="TIMEOUT_CTL" />
        <select element="field" property="name" regex="TIMEOUT" />
        <modify property="lower_range" value="unknownValue" />
    </rule>
</rules>
```

Invalid value "unknownValue" for property "lower_range"

Fix:
Change the value to proper decimal format.

<location> Invalid property attribute for selected component

Property does not match with selected component.

Example:

```
<rules>
    <rule>
        <select element="module" property="lower_range" regex="PERI" />
        <modify property="lower_range" value="unknownValue" />
    </rule>
</rules>
```

test.xml:4:38 Invalid property attribute for selected component

Fix:
Set proper property according to availability of properties.

<location> missing <select> for *<name>* tag

Command does not has a selected item defined.

Example:

```
<rules>
    <rule>
        <modify property="lower_range" value="unknownValue" />
    </rule>
</rules>
```

test.xml:4:53 missing <select> for <modify> tag

Fix:
Define <select> before <modify>.

<location> <select> can not be used after *<command>*

<select> node can not be written after using a <command>.

Example:

```
<rules>
    <rule>
        <create_module name="ACCEN" position="bottom">
            <element regex="CSCU_ACCEN_.*" />
        </create_module>
        <select element="module" property="name" regex="PERI" />
    </rule>
</rules>
```

test.xml:7:55 <select> can not be used after <command>

Fix:
Define <select> before <create_module>.

<location> <select> with ‘property=path’ can be used only once for single <rule>

<select> with property=path can be used only once for each rule <command>.

Example:

```
<rules>
    <rule>
        <select element="module" property="path" regex="PERI,PERI" />
        <select element="register" property="path" regex="PERI,TIMEOUT"
/>
    </rule>
</rules>
```

test.xml:5:65 <select> with 'property=path' can be used only once for single <rule>

Fix:
Define whole path in single <select>.

<location> <name> tag requires subtags

Node was found that should contain subtags but is empty.

Example:

```
<rules>
    <rule>
        <select element="register" property="name" regex="\w+_OCS"
all_occurrences="yes" />
        <protect>
        </protect>
    </rule>
</rules>
```

test.xml:5:8 <protect> tag requires subtags

Fix:
Define subtags.

None of component *<name>* elements match *<value>*

No components were found by regex=<value>.

Example:

```
<rules>
    <rule>
        <select element="register" property="path" regex="PERI,TR_CMD"
all_occurrences="yes" />
        <protect>
            <field regex="TR_SEL_WRONG" />
            <protected_by regex="GROUP_SEL" />
        </protect>
    </rule>
</rules>
```

None of component "TR_CMD" elements match "TR_SEL_WRONG"

Fix:
Check spelling in <field> regex.

<location> <name> can't be used with *<name>*

Both tags can't be combined.

Example:

```
<rules>
    <rule>
        <select element="register" property="path" regex="PERI,TR_CMD"
all_occurrences="yes" />
        <protect>
            <field regex="TR_SEL" />
            <common prefix="prefix" suffix="suffix" />
        </protect>
    </rule>
</rules>
```

test.xml:5:8 <field> can't be used with <common>.

Fix:
Choose either <field> or <common> for single rule or split them to different rules.

Invalid min_value

Both tags can't be combined.

Example:

```
<rules>
    <rule>
        <select element="register" property="path" regex="PERI"
all_occurrences="yes" />
        <for iter_name="i" min_value="UNKNOWN" max_value="20">
            <create_module name="CH#{i:u}" position="bottom">
                <element regex="CSS\d_CH#{i:u}_.*" />
            </create_module>
        </for>
    </rule>
</rules>
```

Invalid min_value

Fix:
Fix the min value to be either the decimal value or it refers to another <for>.

Invalid iter_name

Variables with same iter_name can not be nested.

Example:

```
<rules>
    <rule>
        <select element="register" property="path" regex="PERI"
all_occurrences="yes" />
        <for iter_name="i" min_value="0" max_value="20">
            <for iter_name="i" min_value="#{o:u}" max_value="20">
                <create_module name="CH#{i:u}" position="bottom">
                    <element regex="CSS\d_CH#{i:u}_.*" />
                </create_module>
            </for>
        </for>
    </rule>
</rules>
```

Invalid iter_name

Fix:
Fix the min_value fixing the 'o' to 'i' as there is no 'o' named for above.

The *<value>* register could not be found

Register defined in use attribute could not be found in selected component.

Example:

```
<rules>
    <rule>
        <select element="register" property="path"
regex="PERI,TIMEOUT_CTL" all_occurrences="yes" />
        <create_view view_name="view1" if="(per.long(D:#
{offset:x})&amp;0x800)==0x800" use="UNKNOWN" />
    </rule>
</rules>
```

The "UNKNOWN" register could not be found

Fix:
Check spelling in register and make sure it belongs to selected components parent.

ELSE command can not be created for *<value>* without if command

Register defined in use attribute could not be found in selected component.

Example:

```
<rules>
    <rule>
        <select element="register" property="path"
regex="PERI,TIMEOUT_CTL" all_occurrences="yes" />
        <create_view view_name="view1" if="default"/>
    </rule>
</rules>
```

ELSE command can not be created for "TIMEOUT_CTL" without if command

Fix:
Use create_view with condition before "default".

<location> Root tag *<name>* not found.

Its thrown if any of leading node from xml is not found in xml.

Example:

```
<rule>
    <select element="module" property="name" regex="PERI" />
</rule>
```

test.xml::0:0 Root tag <rules> not found.

Fix:
Insert proper root tag to the xml.

<location> duplicated element.

Duplicated element in create_module was found.

Example:

```
<rules>
    <rule>
        <select element="register" property="path"
regex="PERI,TIMEOUT_CTL" all_occurrences="yes" />
        <create_module name="ACCENCS" position="bottom">
            <element regex="CSS\d_ACCENCS_.*" />
            <element regex="CSS\d_ACCENCS_.*" />
        </create_module>
    </rule>
</rules>
```

test.xml:7:35 duplicated element

Fix:
Use create_view with condition before "default".

Wrong input file specified for *<name>* format.

Unknown file was asked to be converted using wrong converter Type.

Example:

SVD converter was asked to be convert a file without <device> node

Wrong input file specified for SVD format

This inputs are not supported by our converter

Unknown input file is being converted using AUTO mode.

Functions

The table below shows an extract of functions useful for writing PER files.

For a complete list of available functions please see:

- **PowerView Function Reference**
- **General Function Reference**
- **Stimuli Generator Function Reference**

<int>	**CONVert.BOOLTOINT(***<bool>***)**	Converts a boolean value to an integer. TRUE becomes 1, FALSE becomes 0 This function allows you to write conditional base statements e.g.: `base VM:(0x1010*conv.booltoint(d.l(vm:0)==42)|0x1070*conv.booltoint(d.l(vm:0)!=42)`
<int>	**PER.ARG(***<index>***)** and **PER.ARG.ADDRESS()** **(deprecated)**	We recommend that you no longer use these two deprecated functions. Instead, use the method described in **"Passing Arguments"**, page 8. Returns the (optional) argument of the **Per.view** command. The parameter is currently not used. Only useful inside peripheral definition files.
<int>	**PER.Buffer.Byte(***<index>***)**	Returns a byte from the SGROUP buffer. Only useful within a **SGROUP** of a PER file.
<int>	**PER.Buffer.Word(***<index>***)**	Returns a 16 bit word from the SGROUP buffer. Only useful within a **SGROUP** of a PER file.
<int>	**PER.Buffer.Long(***<index>***)**	Returns a 32 bit word from the SGROUP buffer. Only useful within a **SGROUP** of a PER file.
<int>	**PER.Buffer.Quad(***<index>***)**	Returns a 64 bit from the SGROUP buffer. Only useful within a **SGROUP** of a PER file.
<address>	**PER.EVAL(***<index>***)**	Returns the value of a expression (defined with **BASE**) inside a peripheral definition file (PER file), which was defined after **BASE**, **IF**, **ELIF** or **ELSE** command. The parameter defines which expression is returned (0=first one). **Note 1:** The function returns only the last evaluated value of the expression. It will not evaluated the expression again. Expressions after **BASE**, will be evaluated by a **GROUP** command after the **BASE** command in a PER file. **Note 2:**The function must only be used in the context of **IF** or **ELIF**.