teco tecomat®

PROGRAMMABLE CONTROLLERS

INSTRUCTION SET OF PLC TECOMAT
32 BIT MODEL

INSTRUCTION SET OF PLC TECOMAT
32 BIT MODEL

8th edition - January 2005

CONTENTS

INTRODUCTION

Principles of instruction description

In the following chapters individual PLC instructions are described. A great number of instructions allow operands of various types from various spaces or they can be without operand as well. Due to transparency of description, we will not be describing in detail all possible combinations, but only typical examples. For example the access to the operands X, Y, S, D, R is always analogical. Thus, if we describe the behaviour of the instruction LD %R12.3, we will assume that instruction LD %X1.7 will behave similarly.

An overview with permissible operands and operation times for individual types of central units are specified in an appendix.

In the header of each instruction there is its symbolical abbreviation and name specified, followed by a table showing the state of stack and scratchpad before and after the instruction. Then, permissible operands are specified (X, Y, S, D, R, #, T) and their type for individual series of central units, function description, flags being effected and typical examples of behaviour.

Since the central units with a stack of 32 bit width allow programming in a higher language according to the IEC 61131 standard, we will use the types of variables complying to this standard. They differ from the Teco variables (besides the name) mainly that they differentiate signed and unsigned variables. A list of variables types according to IEC 61131 and their equivalents according to Teco is given in Table 1.1.

Table 1.1: List of variables types acc. to IEC 61131 and their equivalents acc. to Teco

IEC 61131	Teco	Description
bool	bit	1 bit
byte	byte	8 bits
usint	byte	8 bits unsigned
sint	byte	8 bits signed
word	word	16 bits
uint	word	16 bits unsigned
int	word	16 bits signed
dword	long	32 bits
udint	long	32 bits unsigned
dint	long	32 bits signed
real	float	32 bits floating point
lreal	double	64 bits floating point

Absolute addresses are written beginning with %, which is obligatory when programming central units with a stack of 32 bit width. In the same way, the prefixes are notated in the form __*indx()* (see chapter 15.8).

Central unit series and stack model

PLC TECOMAT and TECOREG controllers central units are divided into the following series according to their characteristics:

Series B - NS950 CPM-1B, CPM-2B
Series C - TC700 CP-7001, CP-7002
Series D - TR050, TR200, TR300, TC400, TC500, TC600, NS950 CPM-1D
Series E - NS950 CPM-1E
Series M - NS950 CPM-1M

Series S - NS950 CPM-1S, CPM-2S

PLC TECOMAT have two stack models that differ from each other by the width of one layer. Series B, D, E, M and S have their individual stack layers 32 bits wide, while series C have the width of the stack layers 32 bits. This results in certain differences in the behaviour of individual models.

This manual deals exclusively with central units having their stack width of 32 bits. The instruction set for the central units with the stack width of 16 bits is described in the Instruction set of PLC TECOMAT - 16 bit model, TXV 001 05.02.

The differences in behaviour of both models and transfer of the user program between them are described in chapter 15 of this manual.

Principles of example illustrations

In the examples of some instructions memory spaces and PLC stack are shown graphically in compliance with principles corresponding to the format being used. Lower case letters are used for arbitrary unchanged values. The details on data formats in memory spaces and in the stack can be found in the Programmer's manual for PLC TECOMAT TXV 001 09.02.

When describing instructions, stack A is always used as the active one, but any other stack can be used instead of it.

A brief overview of instruction set

1. Data load and write instructions
Data write and load in all formats, indirect write and load, conditional write and write with alternation of the highest bit.

2. Logical instructions
Logical instructions AND, OR, XOR with direct as well as negated operands, negation, leading edge detection, detection of both edges, conditional setup of variables or setting variables to zero, rotate left, rotate right, shift left, shift right, logical swap of stack, interchange of bytes of the stack top, logical functions on the stack top.

3. Timers, shift registers, counters, step controllers
Ahead counter, back counter, bidirectional counter, shift register (left and right), retentive timer with on / off, integrating timer, defined length pulse, step sequencer.

4. Arithmetic instructions
Arithmetic instructions in fixed base point (8, 16, 32 bits) with sign, without sign, addition, subtraction, multiplication, division, incrementation, decrementation, comparison limit function, absolute value, sign change, conversion from binary system to BCD code and vice versa.

5. Stack operations
Stack shift, stack interchange, value transfer among stacks, system stack.

6. Jump and call instructions
Direct, indirect jumps, conditional jumps, direct subroutine calls, indirect calls, conditional calls, return from subroutine, conditional return from subroutine, labels.

7. Organizational instructions
Process start and end, conditional end of process, cycle end, no operation instruction, breakpoint, conditional process interrupt.

8. Table instructions

Load and write to the table or scratchpad field, search for value.

9. Block operations

Move data block, move table to scratchpad and vice versa, fill block with constant..

10. Operations with structured tables

Load and write items of structured tables, search for item, fill item with constant.

11. Floating point arithmetic instructions

Addition, subtraction, multiplication, division, comparison, rounding, absolute value, logarithmic, exponential and goniometric functions, conversion between formats with floating and fixed base point.

12. PID controller instructions

Conversion of measured analog values to standardized ones with diagnostics of edge states, PID controller, PID controller with automatic debugging. .

13. Terminal operation instructions and operations with ASCII characters

Operation of alphanumerical display, conversion of numbers to ASCII strings and vice versa.

14. System instructions

Scratchpad communication feedback control, access to real time circuit (RTC), load and write to additional DataBox memory, peripheral system control.

There is one more group of the instructions that is not described in this manual. These are the instructions used exclusively to support a higher language. They are not used in user programs created in PLC instructions.

1. DATA LOAD AND WRITE INSTRUCTIONS

LD, LDQ	**Load direct data**
LDC	**Load complement data**

Instruction	**Input parameters**									**Result**								
	stack								ope-rand	stack								ope-rand
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
LD									a	A6	A5	A4	A3	A2	A1	A0	a	a
LD [lreal]									a	A5	A4	A3	A2	A1	A0	a		a
LDQ									a	A5	A4	A3	A2	A1	A0	a		a
LDC									a	A6	A5	A4	A3	A2	A1	A0	$\bar{a}$	a

Operands

		bool	byte usint sint	word uint int	dword udint dint	real	lreal
LD	X Y S D R	C	C	C	C	C	C
LD	#				C	C	
LDQ	#						C
LDC	X Y S D R	C	C	C	C		

Function

LD - load data on the stack top
LDQ - load 64-bit constant on the stack top
LDC - load complement data on the stack top

Description

Instructions **LD** and **LDQ** load data from the addressed location and save it without any change on the stack top, instruction **LDC** negates the loaded data and saves it on the stack top. The value of the source address remains unchanged.

Instructions with **bool** type operands shift the stack one level ahead and set identically all 32 bits of A0 stack top.

Instructions with **byte, usint** and **sint** type operands shift the stack one level ahead and write to the lower byte of A0 stack top. The other bytes of the stack top are set to zero.

Instructions with **word, uint** and **int** type operands shift the stack one level ahead and write to the lower word of the A0 stack top. The upper word of the stack is set to zero.

Instructions with **dword**, **udint**, **dint** and **real** type operands move the stack one level ahead and write to the entire A0 stack top.

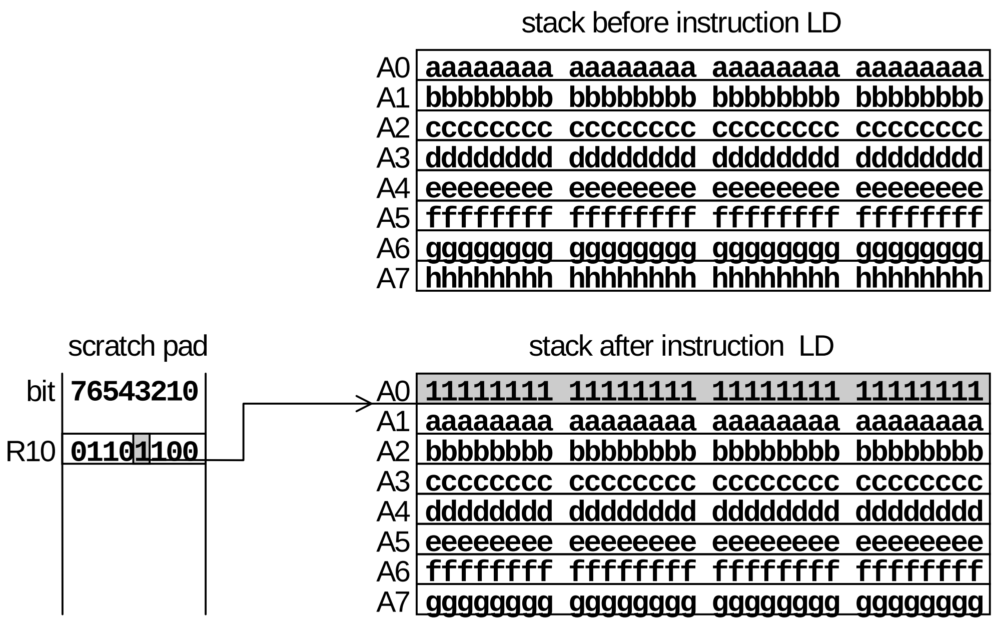
Instructions with **lreal** type operands move the stack two levels ahead and write to A01 stack top.

Example

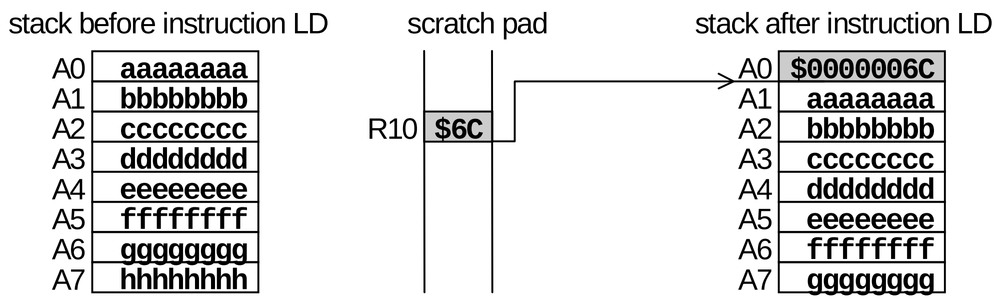
```
#reg bool  read, readc, write, writec
;
P 0
     LD    read
     WR    write
     LDC   readc
     WR    writec
E 0
```

Diagram

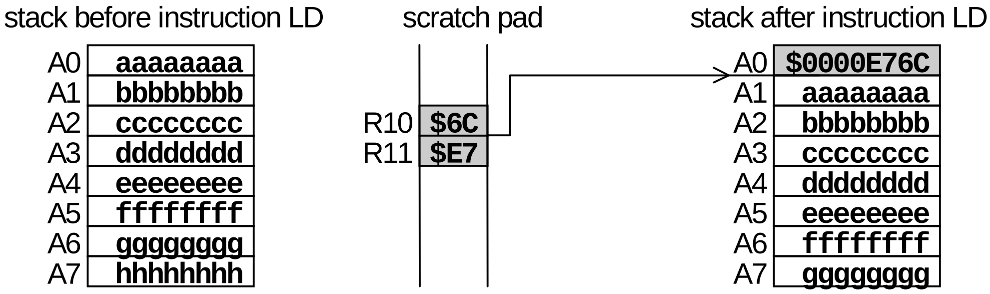
LD %R10.3



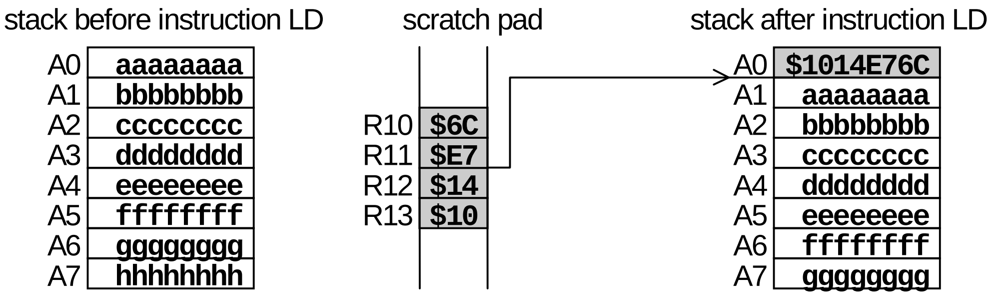
LD %R10



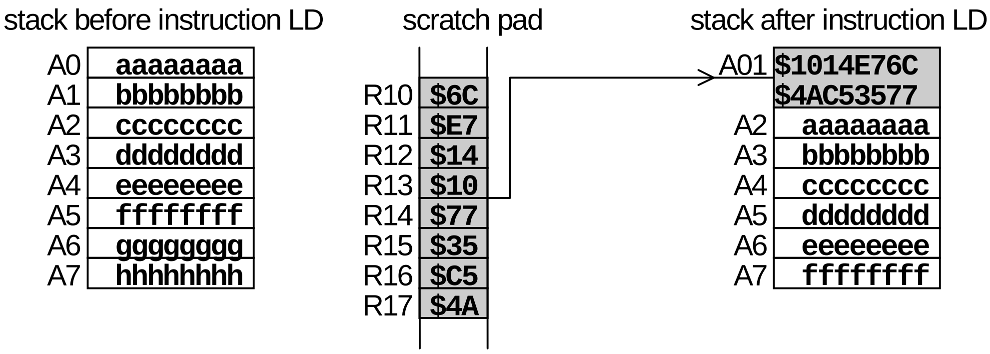
LD %RW10



LD %RL10



LD %RD10



LDIB, LDI, LDIW, LDIL, LDIQ Indirect data load

Instruction	Input parameters									Result								
	stack								ADR	stack								ADR
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
LDIB, LDI								ADR	*a*								*a*	*a*
LDIW, LDIL								ADR	*a*								*a*	*a*
LDIQ								ADR	*a*	A6	A5	A4	A3	A2	A1	*a*		*a*

ADR - address being loaded (type udint)

Operands

		bool	byte usint sint	word uint int	dword udint dint	real	lreal
LDIB	w/o operand	C					
LDI	w/o operand		C				
LDIW	w/o operand			C			
LDIL	w/o operand				C	C	
LDIQ	w/o operand						C

Function

LDIB - load data bit from the bit address saved at the stack top
LDI - load 8 bits of data from the address saved at the stack top
LDIW - load 16 bits of data from the address saved at the stack top
LDIL - load 32 bits of data from the address saved at the stack top
LDIQ - load 64 bits of data from the address saved at the stack top

Description

Instructions **LDIB**, **LDI**, **LDIW**, **LDIL** and **LDIQ** use the content of the stack top as the address. The content of this address is saved at the stack top without any change. The content of the source location remains unchanged. The instruction **LDIB** processes the stack top as so called bit address. The bit address from the byte address by multiplication by 8 and by addition the bit number where we want to read from. The other instructions use the byte address. To get the bit and byte base addresses, the **LEA** instruction is used.

The instruction **LDIB** sets identically all 32 bits of the A0 stack top to the value of the bit loaded.

The instruction **LDI** writes the value loaded to the lowest byte of the A0 stack top. The other bytes of the stack top are set to zero.

The instruction **LDIW** writes the value read to the lower word of the A0 stack top. The upper word of the stack top is set to zero.

The instruction **LDIL** writes the value loaded to the entire A0 stack top.

The instruction **LDIQ** shifts the stack one level ahead and writes the value read to the A01 stack top.

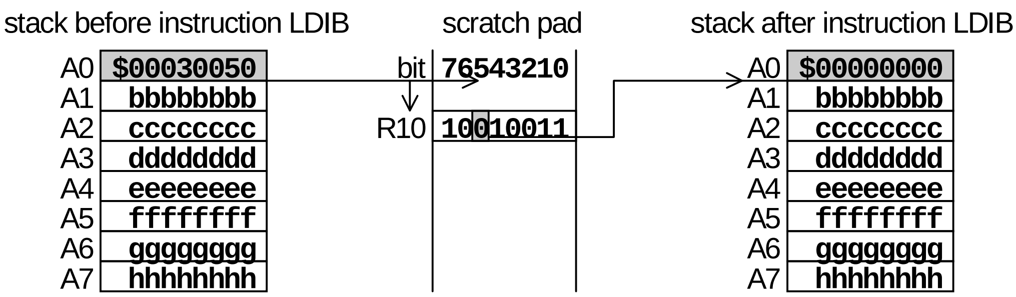
These instructions are useful for indirect data access, when the address is obtained by calculation.

Example

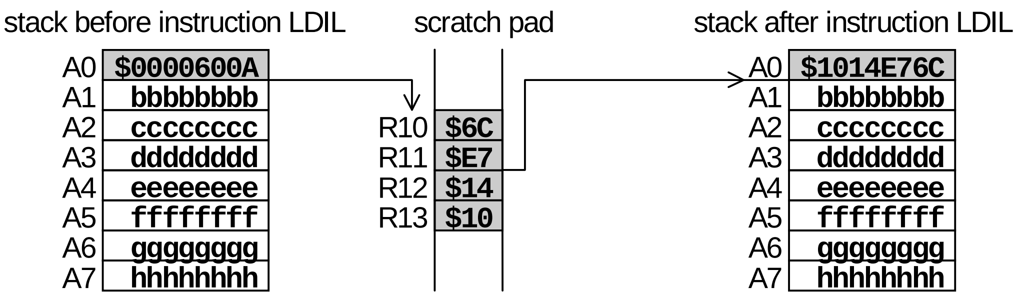
```
#reg usint array1[20], array2[20]     ;arrays of byte size items
#reg usint index1, index2             ;pointers to array item
;
P 0
     LEA   array1
     ADD   index1
     LDI
     LEA   array2
     ADD   index2
     WRI
E 0
```

Diagram

LEA %R10.5
LDIB



LEA %RL10
LDIL



LEA	**Load address**

Instruction	Input parameters									Result								
	stack								ope-rand	stack								
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
LEA									ADR	A6	A5	A4	A3	A2	A1	A0	ADR	

ADR - address being loaded (type udint)

Operands

		bool	byte usint sint	word uint int	dword udint dint	real	lreal
LEA	X Y S D R	C	C	C	C	C	C

Function

LEA - load the address contained in the operand

Description

The instruction **LEA** is used to create a base address for the instructions of indirect lode and write operations.

Instruction with **bool** type operand shifts the stack one level ahead and saves the bit address at the stack top, i.e. the octuple of the byte address increased by the bit number.

Instructions with other types of operand shift the stack one level ahead and saves the byte address at the stack top.

Example

```
#reg bool   array1[20], array2[20]    ;bit arrays
#reg usint  index1, index2            ;pointers to array items
;
P 0
     LEA  array1
     ADD  index1
     LDIB
     LEA  array2
     ADD  index2
     WRIB
E 0
```

WR	**Data write**
WRC	**Write data complement**

<table>
<tr><th rowspan="3">Instruction</th><th colspan="9">Input parameters</th><th colspan="9">Result</th></tr>
<tr><td colspan="8">stack</td><td rowspan="2">ope-rand</td><td colspan="8">stack</td><td rowspan="2">ope-rand</td></tr>
<tr><td>A7</td><td>A6</td><td>A5</td><td>A4</td><td>A3</td><td>A2</td><td>A1</td><td>A0</td><td>A7</td><td>A6</td><td>A5</td><td>A4</td><td>A3</td><td>A2</td><td>A1</td><td>A0</td></tr>
<tr><td>WR</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>a</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>a</td><td>a</td></tr>
<tr><td>WR [lreal]</td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="2">a</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="2">a</td><td>a</td></tr>
<tr><td>WRC</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>a</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>a</td><td>$\overline{a}$</td></tr>
</table>

Operands

		bool	byte usint sint	word uint int	dword udint dint	real	lreal
WR	X Y S R	C	C	C	C	C	C
WRC	X Y S R	C	C	C	C		

Function

WR - data write from the stack top
WRC - write data complement from the stack top

Description:

The instruction **WR** loads the value of the stack top and saves it to the addressed location without any change. The instruction **WRC** negates the loaded value and then it saves it to the addressed location. The content of the entire stack remains unchanged.

Instructions with **bool** type operands carry out the OR operation of all bits of the A0 stack top and saves its value to the addressed bit, the bit instruction **WRC** saves the negated value of this operation (NOR). If A0 = 0, then the instruction **WR** writes the value of log.0 and the **WRC** instruction writes the value of log.1, in the other cases (A0 ≠ 0) the instruction **WR** writes the value of log.1 and instruction **WRC** writes the value of log.0.

Important: Bit instruction **WRC** writes the negated value of the logic OR operation of all A0 bits, thus the NOR function. Its result is not identical to the result that we would get by logical addition of negated A0 bits.

Instructions with **byte, usint** and **sint** type operand work only with the lowest byte of the stack top. The other three bytes of the A0 stack top are not processed.

Instructions with **word, uint** and **int** type operand work only with the lower word of the stack top. The upper word of the A0 stack top is not processed.

Instructions with **dword, udint, dint** and **real** type operands work with the entire A0 stack top.

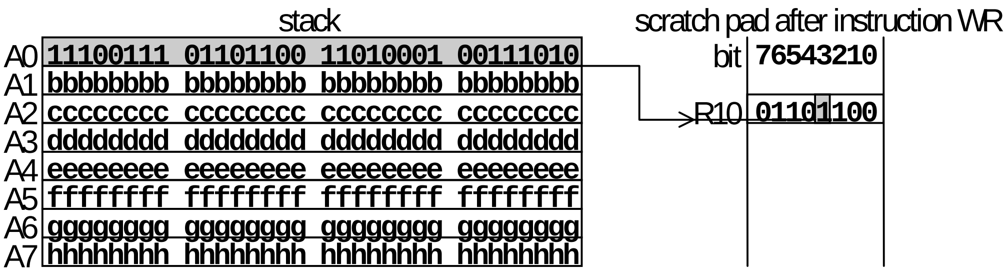
Instructions with **lreal** type operand work with the stack top consisting of the A01 double layer.

Example

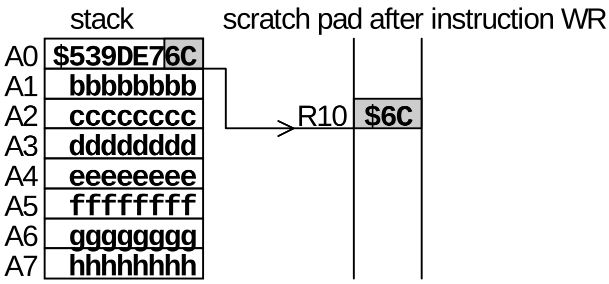
```
#reg bool  read, readc, write, writec
;
P 0
     LD    read
     WR    write
     LD    readc
     WRC   writec
E 0
```

Diagram

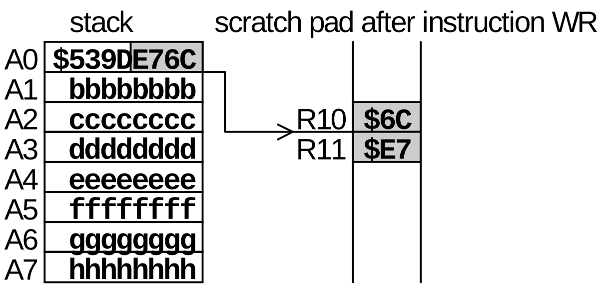
WR %R10.3



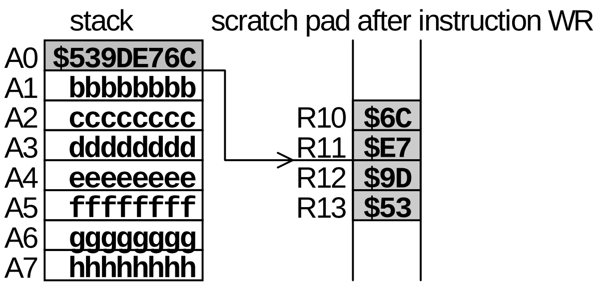
WR %R10



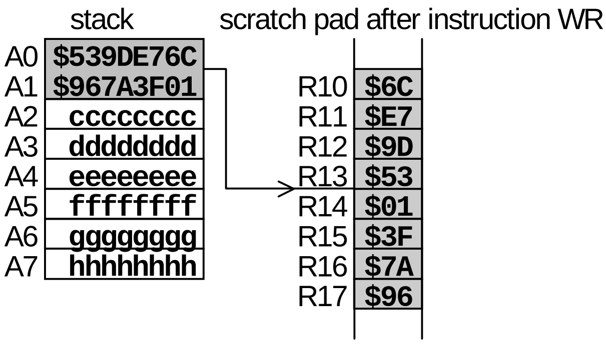
WR %RW10



WR %RL10



WR %RD10


stack
scratch pad after instruction WR
A0 $539DE76C
A1 $967A3F01
A2 cccccccc
A3 dddddddd
A4 eeeeeeee
A5 ffffffff
A6 gggggggg
A7 hhhhhhhh
R10 $6C
R11 $E7
R12 $9D
R13 $53
R14 $01
R15 $3F
R16 $7A
R17 $96

WRIB, WRI, WRIW, WRIL, WRIQ   Indirect data write

Instruction	Input parameters									Result								
	stack								ADR	stack								ADR
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
WRIB, WRI							*a*	ADR		ADR	A7	A6	A5	A4	A3	A2	*a*	*a*
WRIW, WRIL							*a*	ADR		ADR	A7	A6	A5	A4	A3	A2	*a*	*a*
WRIQ						*a*		ADR		*a*	A7	A6	A5	A4	A3	*a*		*a*

ADR   - address being written (type udint)

Operands

		bool	byte usint sint	word uint int	dword udint dint	real	lreal
WRIB	w/o operand	C					
WRI	w/o operand		C				
WRIW	w/o operand			C			
WRIL	w/o operand				C	C	
WRIQ	w/o operand						C

Function

WRIB - write data bit to the bit address saved at the stack top
WRI - write 8 bits of data to the address saved at the stack top
WRIW - write 16 bits of data to the address saved at the stack top
WRIL - write 32 bits of data to the address saved at the stack top
WRIQ - write 64 bits of data to the address saved at the stack top

Description

Instructions **WRIB**, **WRI**, **WRIW**, **WRIL** and **WRIQ** use the content of the stack top as the address. They shift the stack one level ahead and the content of the new stack top is saved without any change to this address. Instruction **WRIB** processes the stack top as so called "bit address". The bit address from the byte address by multiplication by 8 and adding the bit number to which we want to write. The other instructions use the byte address. To obtain the bit and byte base addresses the **LEA** instruction is used.

The instruction **WRIB** carries out logic OR operation of all bits of the new A0 stack top (former layer A1) and saves its value to the addressed bit. If A0 = 0, then the instruction writes the value of log.0, in the other cases (A0 ≠ 0) it writes the value of log.1.

The instruction **WRI** writes to the given address the lowest byte of the new stack top (former layer A1). The other three bytes of the A0 stack top are not processed.

The instruction **WRIW** writes to the given address the lower word of the new stack top (former layer A1). The upper word of the A0 stack top is not processed.

The instruction **WRIL** writes to the given address the content of the new stack top (former layer A1).

The instruction **WRIQ** writes to the given address the content of the new stack top formed by double layer A01 (former double layer A12).

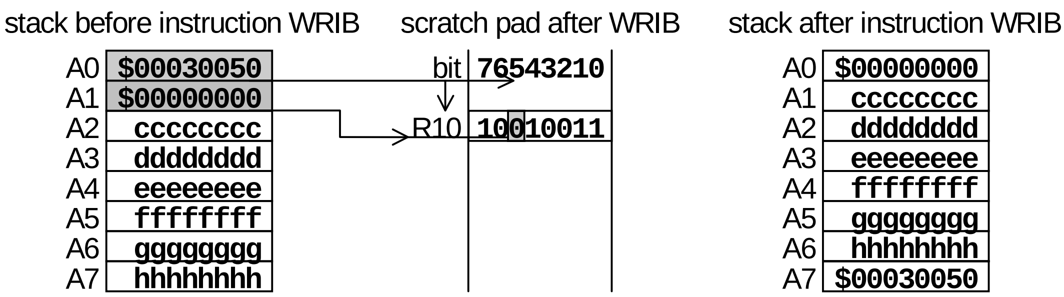
These instructions are useful for indirect data access, when the address is obtained by calculation.

Example

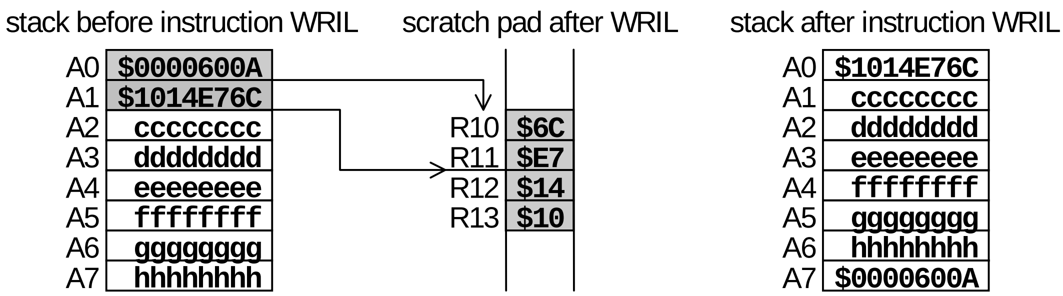
```
#reg usint array1[20], array2[20]     ;arrays of byte size items
#reg usint index1, index2             ;pointers to array items
;
P 0
     LEA   array1
     ADD   index1
     LDI
     LEA   array2
     ADD   index2
     WRI
E 0
```

Diagram

```
LEA %R10.5
WRIB
```




```
LEA %RL10
WRIL
```



WRA Write data with alternation

Instruction	Input parameters									Result								
	stack								ope-rand	stack								operand
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
WRA								a	b								a	$(\overline{b_{max}})a$

Operands

		byte usint	word uint	dword udint
WRA	X Y S R	C	C	C

Function

WRA - data write from the stack top with alternation of the highest bit

Description

The instruction **WRA** loads from the stack top, masks the highest bit and saves it at the addressed location. Then, it negates the currently highest bit of the addressed location (alternation). The content of the entire stack remains unchanged. This instruction can be very useful when controlling intelligent peripheral units requiring alternation of the highest bit during parameter passing (e.g. operation of the serial channel in **UNI** mode).

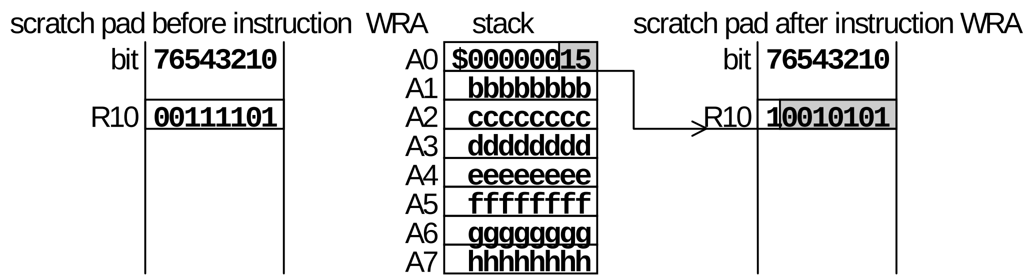
The instruction with the **byte** and **usint** types operand works only with the lowest byte of the A0 stack top. The other three bytes of stack top are not processed by the instruction.

The instruction with the **word** and **uint** types operand works only with the lower word of the A0 stack top. The upper word of the stack top is not processed.

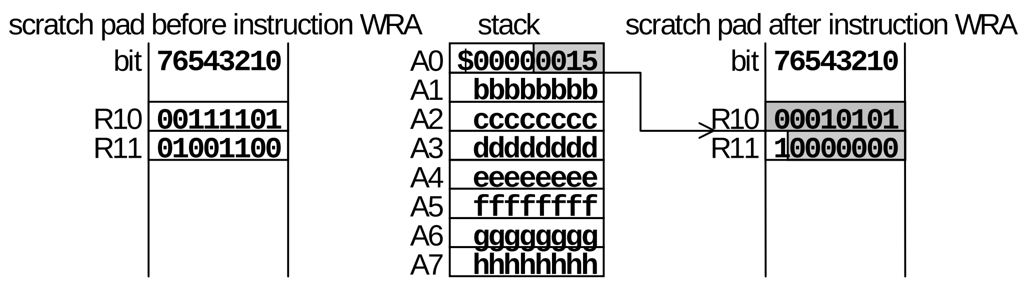
The instruction with **dword** and **udint** types operand works with the entire the A0 stack top.

Diagram

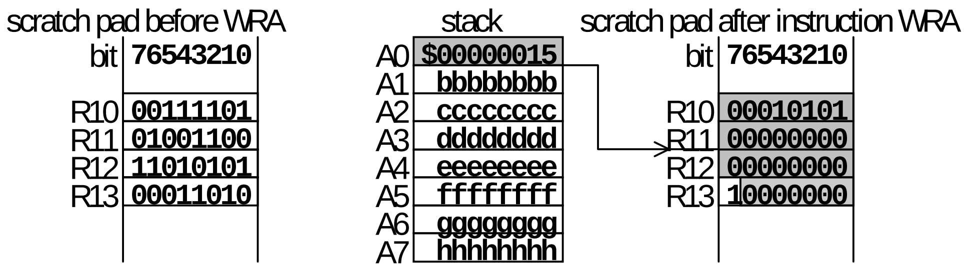
WRA %R10



WRA %RW10



WRA %RL10


scratch pad before WRA
bit 76543210
R10 00111101
R11 01001100
R12 11010101
R13 00011010
stack
A0 $00000015
A1 bbbbbbbb
A2 cccccccc
A3 dddddddd
A4 eeeeeeee
A5 ffffffff
A6 gggggggg
A7 hhhhhhhh
scratch pad after instruction WRA
bit 76543210
R10 00010101
R11 00000000
R12 00000000
R13 10000000

PUT Conditional data write

Instruction	Input parameters									Result								
	stack								S1.0	stack								ope-rand
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
PUT								*a*	1								*a*	*a*
								a	0								*a*	

Operands

		bool	byte usint sint	word uint int	dword udint dint	real
PUT	X Y S R	C	C	C	C	C

Function

PUT - data write from the stack top conditional on the value of log.1 of bit S1.0

Description

The instruction **PUT** is similar to instruction **WR** which is performed only if S1.0 = log.1. If S1.0 = log.0 it does not do anything. The instruction **PUT** tests the bit S1.0 and if it equals to log.1, it reads the value of the A0 stack top and saves it without any change to the addressed location. The content of the entire stack as well as flag registers remains unchanged.

The instruction with the **bool** type operand in case of S1.0 = log.1 performs the logic OR operation of all bits of the A0 stack top and saves its value to the addressed bit. If A0 = 0, then the instruction write the value of log.0, in the other cases (A0 ≠ 0) the instruction writes the value of log.1.

Instructions with the **byte**, **usint** and **sint** types operand work only with the lowest byte of the stack top. The other three bytes of the A0 stack top are not processed.

Instructions with the **word**, **uint** and **int** types operand work only with the lower word of the stack top. The upper word of the A0 stack top is not processed.

Instructions with **dword**, **udint**, **dint** and **real** types operands work with the entire the A0 stack top.

Flags

	.7	.6	.5	.4	.3	.2	.1	.0
S1	-	-	-	-	-	-	-	S

S1.0 (S) - input condition of the instruction
0 - instruction is not performed
1 - instruction is fully performed

Example

```
#reg bool  read, condition, write
;
P 0
     LD    condition
     WR    %S1.0
     LD    read
     PUT   write
E 0
```

```
P 0
 |condit»                                                               %S1.0 |
 +-[LD ]----------------------------------------------------------------( )---+
 |                                                                            |
 | read      write                                                            |
 +-[LD ]-----[PUT]------------------------------------------------------------+
 |                                                                            |
E 0
```

Diagram

If S1.0 has the value of log.1, the diagram of the instruction **PUT** is identical to the instruction **WR**. If S1.0 has the value of log.0, the instruction behaves as a do-nothing operation.

2. LOGICAL INSTRUCTIONS

AND	**Function AND**
ANC	**Function NAND**

Instruction	Input parameters									Result								
	stack								ope-rand	stack								ope-rand
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
AND								a	b								$a \cdot b$	b
AND w/o op.							a	b		b	A7	A6	A5	A4	A3	A2	$a \cdot b$	
ANC								a	b								$a \cdot \overline{b}$	b
ANC w/o op.							a	b		b	A7	A6	A5	A4	A3	A2	$a \cdot \overline{b}$	

Operands

		bool	byte usint sint	word uint int	dword udint dint
AND	X Y S D R	C	C	C	C
AND	#				C
AND	w/o operand				C
ANC	X Y S D R	C	C	C	C
ANC	#				C
ANC	w/o operand				C

Function

AND - logical AND of the stack top with operand
ANC - logical AND of the stack top with negated operand

Description

The function of the logical AND operation assumes the value of log.1, if both of their operands are log.1, otherwise it has the value of log.0. In Boolean algebra it represents "simultaneity" ("and", "as well as", "simultaneously"). In relay diagrams, serial arrangement of contacts corresponds to it. The function is clear from the truth table:

Input parameters		Result	
a	b	$a \cdot b$	$a \cdot \overline{b}$
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	0

Operand instructions **AND** take the content of the addressed location and perform its logical AND with the stack top. This is overwritten by the result of the operation. The instruction **ANC** performs logical AND of negation of the taken content of the addressed location with the stack top. The content of the source location remains unchanged.

Instructions with the **bool** type operand process the entire the A0 stack top in such a way that it performs the specified operation with each of its bits. The instruction saves the result of these 32 operations back on the A0 stack top.

Instructions with the **byte**, **usint** and **sint** types operand process the lowest byte of the A0 stack top as 8 bit operations among corresponding stack bits and the operand. The result is saved at the lowest byte of the A0 stack top. The other three bytes of the A0 stack top are set to zero (operation AND 0 is performed).

Instructions with the **word**, **uint** and **int** types operand process the lower word of the A0 stack top as 16 bit operations among corresponding stack bits and the operand. The result is saved on the lower word of the A0 stack top. The upper word is set to zero (operation AND 0 is performed).

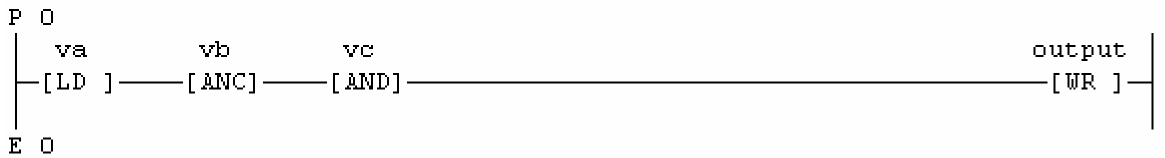
Instructions with the **dword**, **udint** and **dint** types operand process the A0 stack top as 32 bit operations among corresponding stack bits and the operand. The result is saved on the A0 stack top.

Instructions **AND**, **ANC** without operand perform 32 bit operations among corresponding bits of layers A0 and A1 of the stack. Then they shift the stack one level back and write the result of the operation on the new A0 stack top.

Examples

Logical AND $y = a \cdot \overline{b} \cdot c$

```
#reg bool  va, vb, vc, output
;
P 0
     LD   va
     ANC  vb
     AND  vc
     WR   output
E 0
```

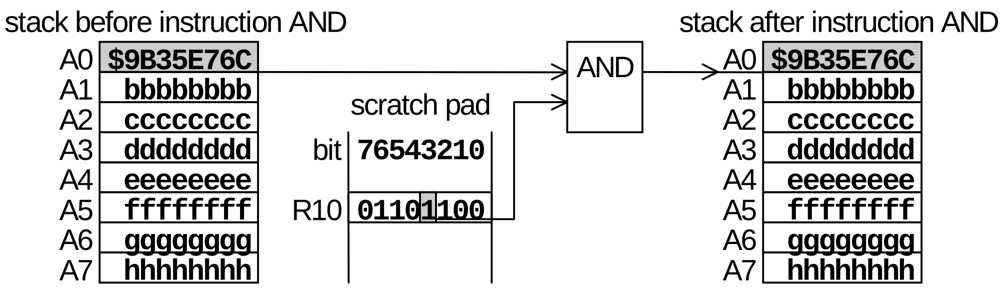


Logical AND $y = a \cdot b$

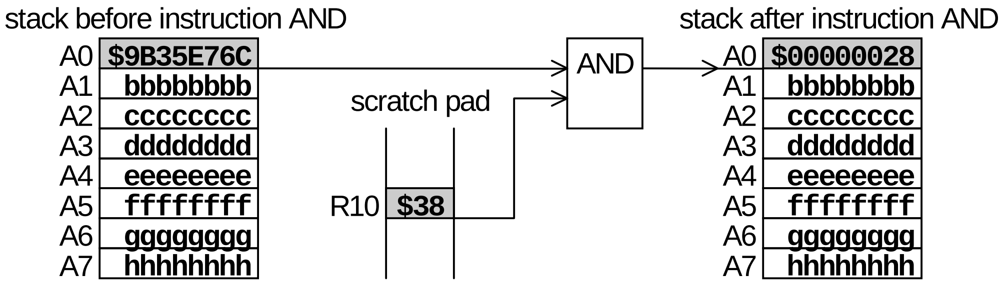
```
#reg bool  va, vb, output
;
P 0
     LD   va
     LD   vb
     AND
     WR   output
E 0
```

Diagram

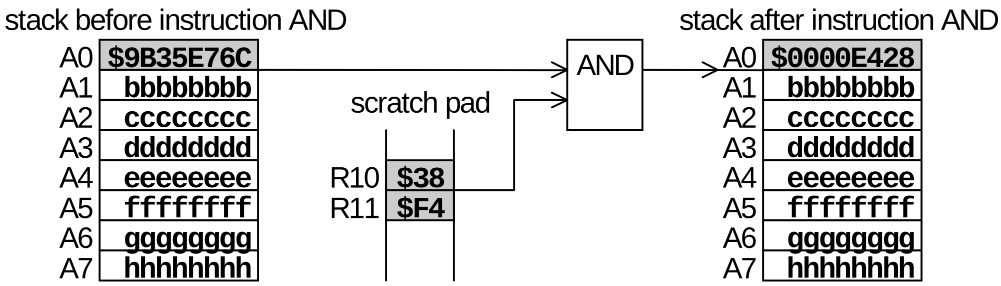
```
LD   $9B35E76C
AND %R10.3
```



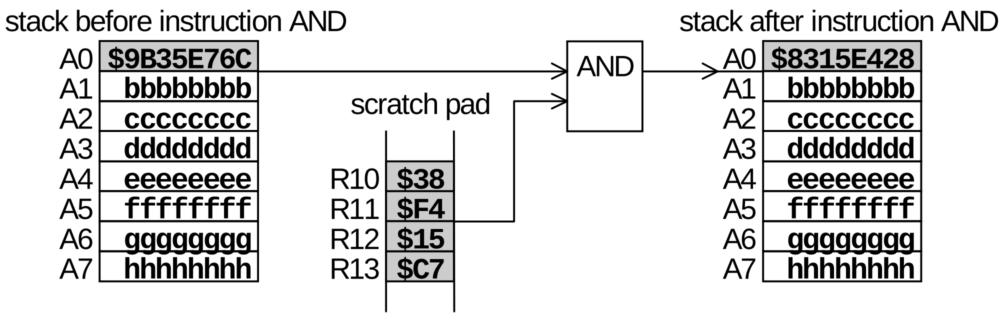

```
LD   $9B35E76C
AND %R10
```



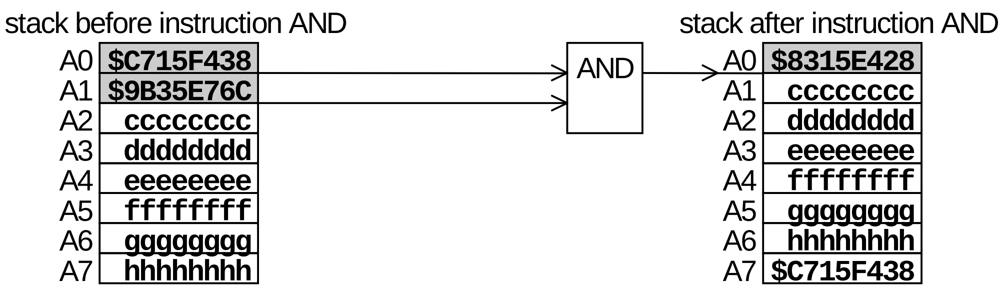

```
LD   $9B35E76C
AND %RW10
```




```
LD   $9B35E76C
AND %RL10
```




```
LD   $C715F438
LD   $9B35E76C
AND
```



OR	**Function OR**
ORC	**Function NOR**

Instruction	Input parameters									Result								
	stack								ope-rand	stack								ope-rand
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
OR								a	b								$a+b$	b
OR w/o op.							a	b		b	A7	A6	A5	A4	A3	A2	$a+b$	
ORC								a	b								$a+\overline{b}$	b
ORC w/o op.							a	b		b	A7	A6	A5	A4	A3	A2	$a+\overline{b}$	

Operands

		bool	byte usint sint	word uint int	dword udint dint
OR	X Y S D R	C	C	C	C
OR	#				C
OR	w/o operand				C
ORC	X Y S D R	C	C	C	C
ORC	#				C
ORC	w/o operand				C

Function

OR - logical OR of the stack top with operand
ORC - logical OR of the stack top with negated operand

Description

The function of the logical OR operation assumes the value of log.1, if at least one of their parameters is log. 1, otherwise it has the value of log. 0. In Boolean algebra it is represented by „or“. In relay diagrams, parallel arrangement of contacts corresponds to it. The function is clear from the truth table:

Input parameters		Result	
a	b	$a+b$	$a+\overline{b}$
0	0	0	1
0	1	1	0
1	0	1	1
1	1	1	1

Operand instructions **OR** take the content of the addressed location and perform its logical OR operation with the stack top. This is overwritten by the result of the operation. The instruction **ORC** performs logical OR of negation of the taken content of the addressed location with the stack top. The content of the source location remains unchanged.

Instructions with the **bool** type operand process the entire the A0 stack top in such a way that it performs the specified operation with each of its bits. The instruction saves the result of these 32 operations back on the A0 stack top.

Instructions with the **byte**, **usint** and **sint** types operand process the lowest byte of the A0 stack top as 8 bit operations among corresponding stack bits and the operand. The result is saved at the lowest byte of the A0 stack top. The other three bytes of the A0 stack top are left unchanged (operation OR 0 is performed).

Instructions with the **word**, **uint** and **int** types operand process the lower word of the A0 stack top as 16 bit operations among corresponding stack bits and the operand. They save the result on the lower word of the A0 stack top. The upper word is left unchanged (operation OR 0 is performed).

Instructions with the **dword**, **udint** and **dint** types operand process the A0 stack top as 32 bit operations among corresponding stack bits and the operand. They save the result on the A0 stack top.

Instructions **OR**, **ORC** without operand perform 32 bit operations among corresponding bits of layers A0 and A1 of the stack. They then shift the stack one level back and write the result of the operation on the new A0 stack top.

Examples

Logical OR $y = a + b + \bar{c}$

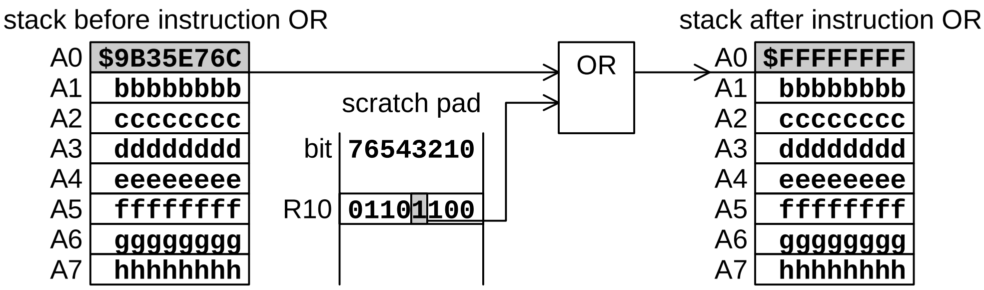
```
#reg bool  va, vb, vc, output
;
P 0
     LD   va
     OR   vb
     ORC  vc
     WR   output
E 0
```

Logical OR $y = a + b$

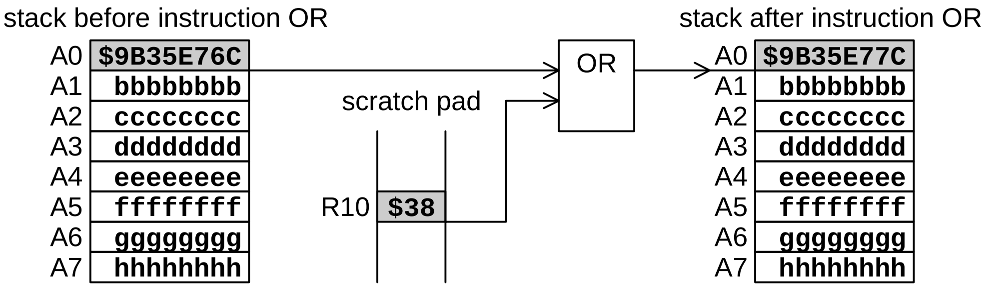
```
#reg bool  va, vb, output
;
P 0
     LD   va
     LD   vb
     OR
     WR   output
E 0
```

Diagram

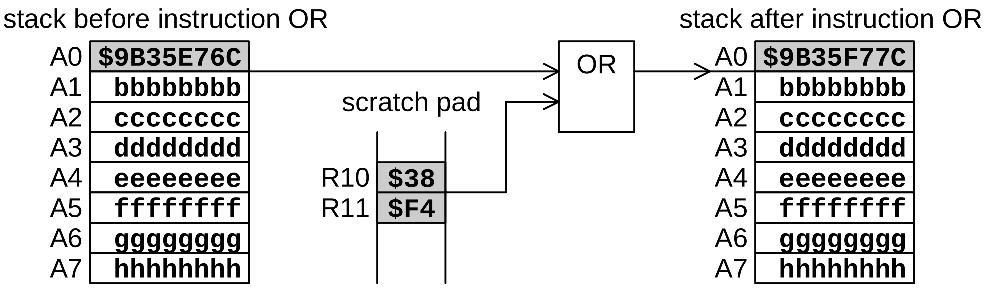
```
LD   $9B35E76C
OR   %R10.3
```



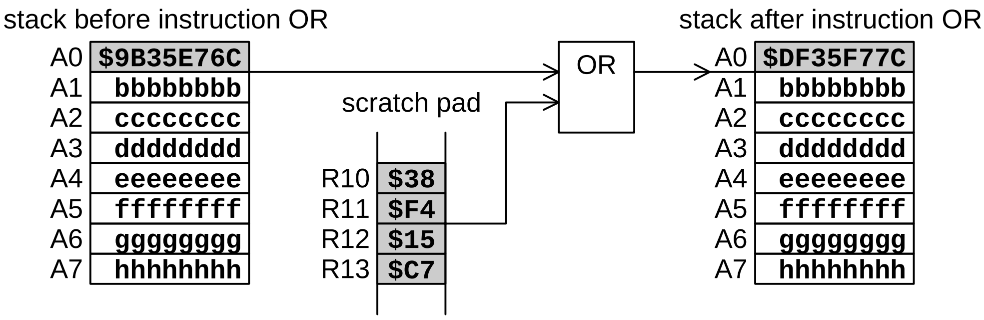

```
LD   $9B35E76C
OR   %R10
```



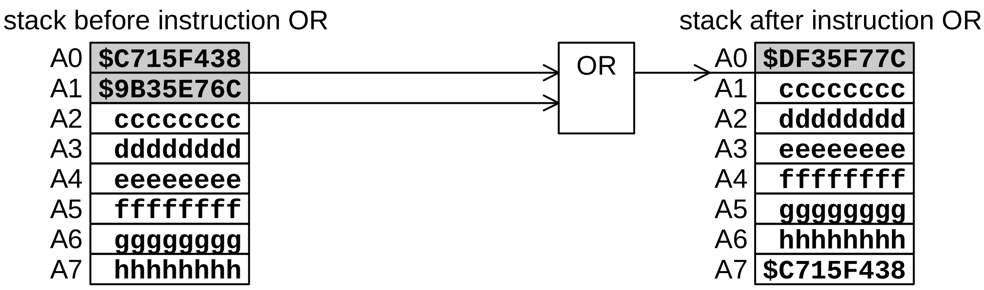

```
LD   $9B35E76C
OR   %RW10
```




```
LD   $9B35E76C
OR   %RL10
```




```
LD   $C715F438
LD   $9B35E76C
OR
```



XOR	**Function Exclusive OR**
XOC	**Function Exclusive NOR**

Instruction	Input parameters									Result								
	stack								ope-rand	stack								ope-rand
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
XOR								a	b								$a \oplus b$	b
XOR w/o op.							a	b		b	A7	A6	A5	A4	A3	A2	$a \oplus b$	
XOC								a	b								$a \oplus \overline{b}$	b
XOC w/o op.							a	b		b	A7	A6	A5	A4	A3	A2	$a \oplus \overline{b}$	

Operands

		bool	byte usint sint	word uint int	dword udint dint
XOR	X Y S D R	C	C	C	C
XOR	#				C
XOR	w/o operand				C
XOC	X Y S D R	C	C	C	C
XOC	#				C
XOC	w/o operand				C

Function

XOR - exclusive logical OR of the stack top with operand
XOC - exclusive logical OR of the stack top with negated operand

Description

The function of the exclusive OR operation (XOR) assumes the value of log.1, if just one of its operands is log.1, otherwise it has the value of log.0. In Boolean algebra it is represented by „either..., or“. For two variables is the function XOR identical to the functions of inequality, modulo 2 and odd parity. For a greater number of inputs this identity is not valid any more. The two-input function can be explained also as mismatch - it equals to log.1, if both operands differ from each other. The function is clear from the following truth table:

Input parameters		Result	
a	b	$a \oplus b$	$a \oplus \overline{b}$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

Operand instructions **XOR** take the content of the addressed location and perform its exclusive logical OR with the stack top. This is overwritten by the result of the operation. The instruction **XOC** performs exclusive logical OR of negation of the taken content of the addressed location with the stack top. The content of the source location remains unchanged.

Instructions with the **bool** type operand process the entire the A0 stack top in such a way that it performs the specified operation with each of its bits. The instruction saves the result of these 32 operations back on the A0 stack top.

Instructions with the **byte**, **usint** and **sint** types operand process the lowest byte of the A0 stack top as 8 bit operations among corresponding stack bits and the operand. The result is saved at the lowest byte of the A0 stack top. The other three bytes of the A0 stack top are left unchanged (operation XOR 0 is performed).

Instructions with the **word**, **uint** and **int** types operand process the lower word of the A0 stack top as 16 bit operations among corresponding stack bits and the operand. They save the result on the lower word of the A0 stack top. The upper word is left unchanged (operation XOR 0 performed).

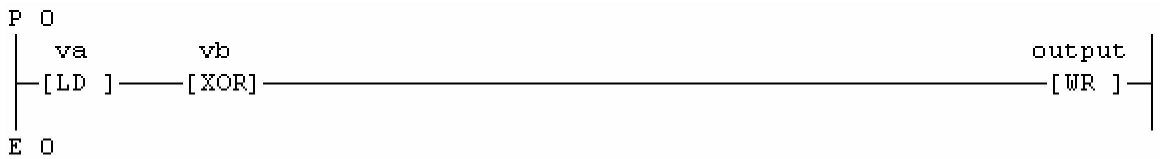
Instructions with the **dword**, **udint** and **dint** types operand process the A0 stack top as 32 bit operations among corresponding stack bits and the operand. They save the result on the A0 stack top.

Instructions **XOR**, **XOC** without operand perform 32 bit operations among corresponding bits of layers A0 and A1 of the stack. They then shift the stack one level back and write the result of the operation on the new A0 stack top.

Examples

Logical exclusive OR $y = a \cdot \overline{b} + \overline{a} \cdot b$

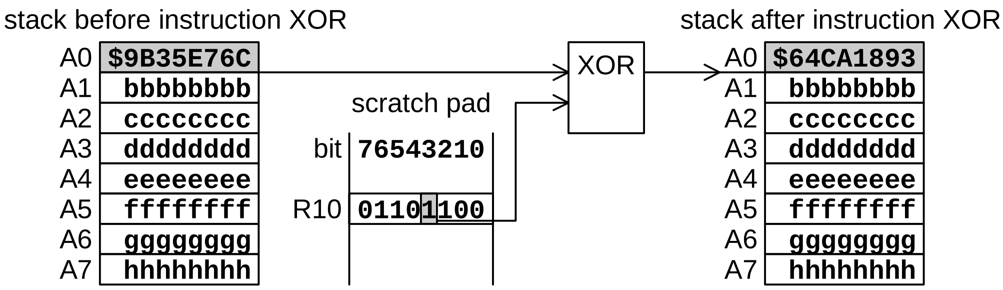
```
#reg bool  va, vb, output
;
P 0
      LD    va
      XOR   vb
      WR    output
E 0
```



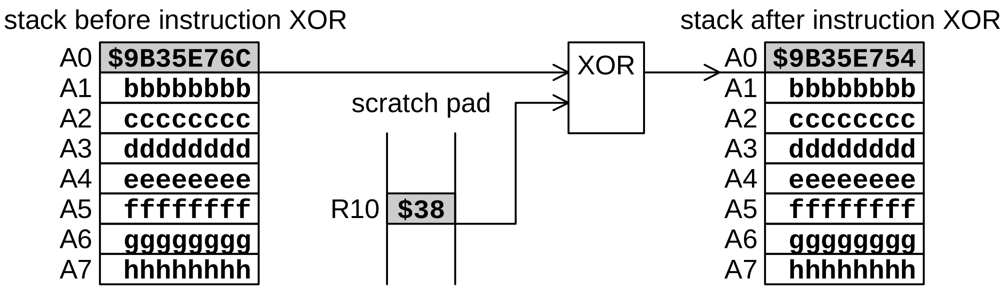

```
#reg bool  va, vb, output
;
P 0
      LD    va
      LD    vb
      XOR
      WR    output
E 0
```

Diagram

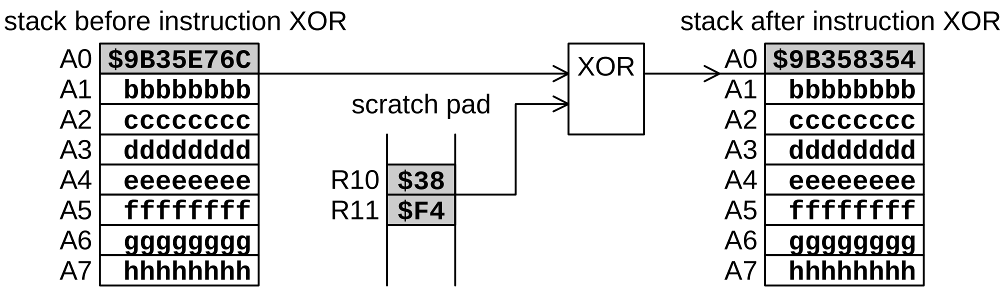
```
LD   $9B35E76C
XOR %R10.3
```



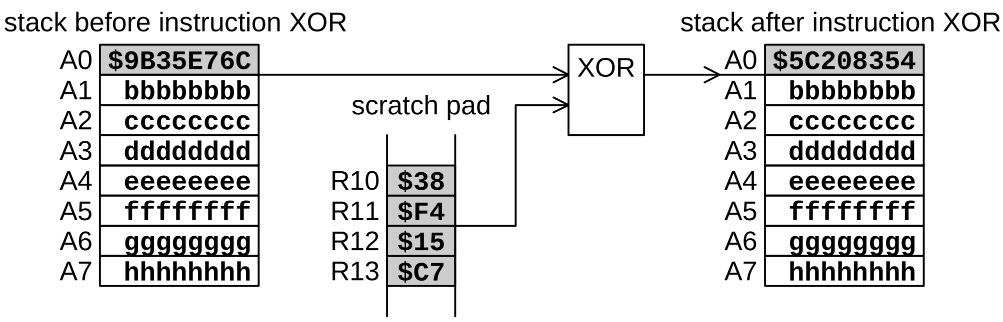

```
LD   $9B35E76C
XOR %R10
```



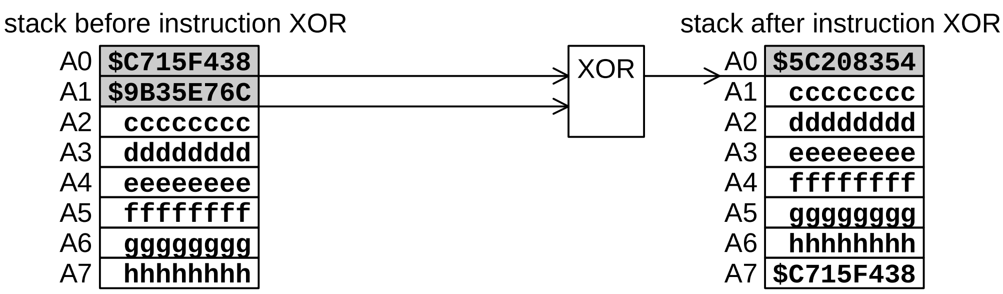

```
LD   $9B35E76C
XOR %RW10
```




```
LD   $9B35E76C
XOR %RL10
```




```
LD   $C715F438
LD   $9B35E76C
XOR
```



NEG Negation

Instruction	Input parameters									Result								
	stack									stack								
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
NEG								a									$\overline{a}$	

Operands

		dword udint
NEG	w/o operand	C

Function

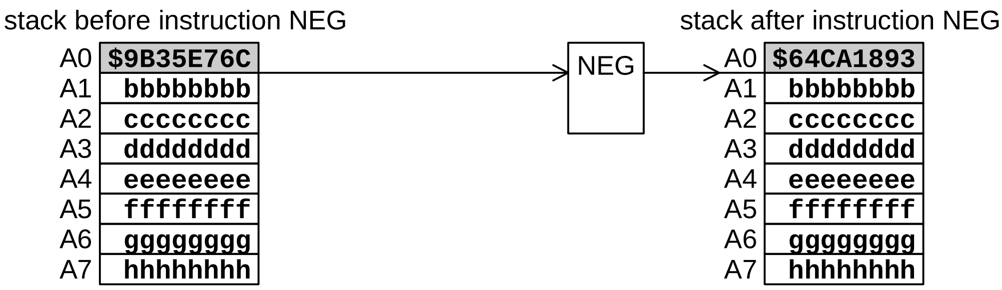
NEG - negation of the stack top

Description

The instruction **NEG** performs negation of all bits of the A0 stack top. The other stack levels remain unchanged.

Diagram

```
LD   $9B35E76C
NEG
```



SET	**Conditional set**
RES	**Conditional reset**

Instruction	Input parameters									Result								
	stack								op.	stack								operand
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
SET								a									a	$a+b$
RES								a									a	$\bar{a}\cdot b$

Operands

		bool	byte usint	word uint	dword udint
SET	X Y S R	C	C	C	C
RES	X Y S R	C	C	C	C

Function

SET - conditional write of log.1 to memory, R - S type flip-flop setting
RES - conditional write log.0 to memory, setting of the R - S type flip-flop to zero

Description

Instruction **SET** performs conditional write of log.1 to the addressed location, instruction **RES** performs conditional write of log.0. If both instructions work with the same memory location then we can understand its content as the analogy of the R - S type flip-flop or another flip-flop with asynchronous inputs R and S.
The instruction does not change the content of the stack.

Function SET sets the content of the addressed location to log.1 only if the control variable loaded from the stack top the value of log. 1, otherwise the content of the location remains unchanged. Function RES sets the content of the addressed location to zero only when the control variable has the value of log. 1, otherwise the content remains unchanged. It can be said that the function SET and RES are active (the content of the addressed location changes) only when the control variable has the value of log. 1 and in this case the function SET writes log.1 and function RES writes log.0. If the control variable has the value of log.0 then the content of the memory location remains unchanged neither after SET or RES (the previous content is retained). Functions SET and RES can be described with the following truth table:

Input parameters		Result	
a	b	$a+b$ (SET)	$\bar{a}\cdot b$ (RES)
0	0	0	0
0	1	1	1
1	0	1	0
1	1	1	0

For instructions with the **bool** type operand the control variable equals to logical OR of all 16 bits of the A0 stack top. If the content of A0 is non-zero (A0 ≠ 0) then the instruction **SET** sets the addressed bit to log.1 and instruction **RES** writes log. 0 under this condition. If the content of level A0 is zero (A0 = 0) then no instruction does not change the content of the addressed location.

Instructions with the **byte** and **usint** types operand perform at a stroke 8 bit operations for homothetic bits of the lowest byte of the A0 stack top (a file of 8 control variables a) and addressed location (a file of 8 status variables b).

Instructions with the **word** and **uint** types operand perform at a stroke 16 bit operations for homothetic bits of the lower word of the A0 stack top (a file 16 control variables a) and addressed location (a file of 16 status variables b).

Instructions with the **dword** and **udint** types operand perform at a stroke 32 bit operations for homothetic bits of the A0 stack top (a file of 32 control variables a) and addressed location (a file of 32 status variables b).

Note

From the technical point of view it is possible to address any location to which it can be written. From the functional point of view some possibilities make no sense since they do not guarantee the correct performance of the instructions (e.g. occupied inputs X, active system registers S).

During one cycle of user program, more **SET** or **RES** instructions can be activated, addressing the common state variable. When activating the same instructions (either **SET** only, or **RES** only), the result is identical after the last instruction as if we performed the only operation with the adding control variable (added OR function).

If the active instructions are performed above the common variable in the sequence of **SET** and **RES** (with the control variable having the value of log. 1), the state after instruction **RES** will be valid (memory with prevailing zero setting).

In the reverse order of the instructions (**RES** and then **SET**) the state after instruction **SET** will be valid (memory with prevailing one setting) - always the last active instruction prevails.

LET	**Pulse from the leading edge**
BET	**Pulse from any edge**

Instruction	Input parameters									Result								
	stack								operand	stack								operand
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
LET								a	b								$a \cdot \overline{b}$	a
BET								a	b								$a \oplus b$	a

Operands

		bool	byte usint	word uint	dword udint
LET	X Y S R	C	C	C	C
BET	X Y S R	C	C	C	C

Function

LET - generating of start pulse from the leading edge
BET - generating of start pulse from any edge

Description

The instructions set its addressed status variable according to same rules as the instruction **WR** or **WRX**. In addition to this they compare the original a newly written content of the status variables (before and after write).

Instruction **LET** sets the result at the stack top to log.1 only when the status variable changes from log.0 to log.1 (leading edge), otherwise they set it to zero.

Instruction **BET** sets the result at the stack top to log.1 only when the status variable changes from log.0 to log.1 or from log.1 to log.0 (any edge), otherwise they set it to zero.

The instruction does not change the content of the stack.

Logical functions LET and BET (value set on the stack top) can be defined with the following truth table:

Input parameters		Result	
a	b	$a \cdot \overline{b}$ (LET)	$a \oplus b$ (BET)
0	0	0	0
0	1	0	1
1	0	1	1
1	1	0	0

Instructions with the **bool** type operand perform the OR operation of all 16 bits of the A0 stack top and the value of this operation will be saved in the addressed bit after the stack top has been tested and set. The result of comparison at the stack top is identical in all 16 bits. Setting of the stack top to log.1 thus represented by the value of 65 535 (all ones).

Instructions with the **byte** and **usint** types operand perform at a stroke 8 bit operations for the homothetic bits of the lowest byte of the A0 stack top (a file of 8 control variables a) and addressed location (a file of 8 status variables b). The results of comparison are saved at the lowest byte of the A0 stack top (a file 8 results). The other three bytes A0 are set to zero.

Instructions with the **word** and **uint** types operand perform at a stroke 16 bit operations for the homothetic bits of the lower word of the A0 stack top (a file of 16 control variables a) and addressed location (a file of 16 status variables b). The results of comparison are saved in the lower word of the A0 stack top (a file of 16 results). The upper word A0 is set to zero.

Instructions with the **dword** and **udint** types operand perform at a stroke 32 bit operations for the homothetic bits of the A0 stack top (a file of 32 control variables a) and addressed location (a file of 32 status variables b). The results of comparison are saved at the A0 stack top (a file of 32 results).

Note

For correct function of the instructions **LET**, **BET** it is necessary that writing to the status variable performs only one instruction **LET**, **BET** (once at each cycle) and that the system program does not work on its content.

If we process the output of the instructions **LET**, **BET** only in internal variables, the pulse can be shorter. The time displacement between the leading edges must be reliably longer than the double of the cycle time is (during one cycle it is not possible to evaluate the leading and trailing edge, if we do not evaluate it in the interrupting process). But if the control variable derives from the internal variables of the user program (not from the inputs or system variables) then it is possible to evaluate also more leading edges.

During the first activation of the system program (after restart) the instructions **LET**, **BET** can accidentally generate false information. This can be avoided by either ignoring the results of the first cycle which will be understood as stabilization of a transient performance or before the first cycle during the process of restart handling the user sets all status variables do ones for instructions **LET** or to the state that corresponds to the idle steady state, for instructions **BET**.

FLG Logical flags of the stack top

Instruction	Input parameters									Result								
	stack									stack								S0
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
FLG								VAL		A6	A5	A4	A3	A2	A1	VAL	N4	NFLG

VAL processed value (type uint)
N4 - logical product AND of the lower word of the A0 stack top (see description)
NFLG - file of logical functions above the A0 stack top (see description)

Operands

		word uint
FLG	w/o operand	C

Function

FLG - logical AND and transverse functions of bytes of the lower word of A0 at S1

Description

The instruction **FLG** processes the content of A0, moves the stack ahead and performs the following operations:

- It determines the number one-bits at the lower word of the original top of A0. This number N assumes a value of 0 to 16, in the binary system it can be written on five bits. The four lower bits N3 to N0 are saved at the lower half of the system register S1. The highest bit N4 having the meaning of longitudinal logical AND operation that simultaneously has a meaning of the logical AND operation of all A0 bits, is saved in all bits of the new A0 stack top.
  The item N can be advantageously used for realization of symmetrical functions (parity, majority, threshold functions, etc.), for example:
  N > 0 (N ≠ 0) - logical OR
  N0 = S1.0 - odd parity, product above 2
  N4 = A0 - logical product AND of the lower word of the A0 stack top
  N3 = S1.3 - if the second lowest byte of the A0 stack top was zero, logical product AND of the A0 lowest byte
  N = 2 - threshold function F_{16}^{2} or F_{n}^{2}
  N = k - threshold function F_{16}^{k} or F_{n}^{k}
  N = 1 - just 1 of 16 (1 of n), function „either, or“, „exclusive or“
  N = {number file} - any symmetrical function defined by a number file

- It performs separately the functions of logical OR and logical AND for both lower bytes of the original A0 stack top and saves the results to register S1.

Flags

	.7	.6	.5	.4	.3	.2	.1	.0
S1	ORH	ORL	ANH	ANL	N3	N2	N1	N0

S1.3 to S1.0 (N3 to N0)
- Together with the value of bit N4, which is saved in all bits of the new A0 stack top it forms a five-bit number N specifying the number of one-bits in the original lower word of the A0 stack top.

S1.0 (N0) - odd parity of the lower word of the original stack top

S1.4 (ANL) - longitudinal logical product of all bits of the lowest byte of the original stack top
S1.5 (ANH) - longitudinal logical product of all bits of the second lowest byte of the original stack top
S1.6 (ORL) - longitudinal logical OR of all bits of the lowest byte of the original stack top
S1.7 (ORH) - longitudinal logical OR of all bits of the second lowest byte of the original stack top

Note

The instruction **FLG** is contained in the instruction file of the 32 bit model due to compatibility of the user programs transferred from a 16 bit model. We do not recommend using of this instruction for new user programs.

STK Transpose stack

Instruction	Input parameters									Result								
	stack									stack								
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
STK	*h*	*g*	*f*	*e*	*d*	*c*	*b*	*a*		*h*	*g*	*f*	*e*	*d*	*c*	*b*	NSTK	

NSTK - logical OR operations of all stack layers (usint - see description)

Operands

		dword udint
STK	w/o operand	C

Function

STK - transposing of logical values of 8 stack levels to A0

Description

The instruction **STK** performs for each layer of the stack A0 to A7 logical OR operation of all 16 bits of the level and then this "column" is "transposed" to the layer A0L according to the following scheme:

A0.7	A0.6	A0.5	A0.4	A0.3	A0.2	A0.1	A0.0
OR7	OR6	OR5	OR4	OR3	OR2	OR1	OR0

OR0 to OR7 are the values of the logical OR operations of the individual layers A0 to A7.

The other three bytes of the stack top are set to zero, the other stack levels remain unchanged.

ROL	**Rotate left**
ROR	**Rotate right**

Instruction	Input parameters									Result								
	stack								ope-rand	stack								
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
ROL n								a	n								$a\langle\langle n$	
ROL							a	n		n	A7	A6	A5	A4	A3	A2	$a\langle\langle n$	
ROR n								a	n								$a\rangle\rangle n$	
ROR							a	n		n	A7	A6	A5	A4	A3	A2	$a\rangle\rangle n$	

Operands

		word uint	dword udint
ROL	n	C	
ROL	w/o operand		C
ROR	n	C	
ROR	w/o operand		C

Function

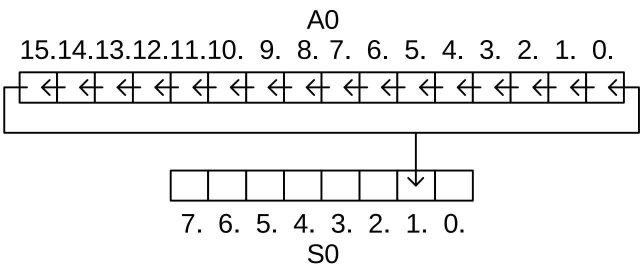
ROL - n-multiple rotation of the value of the A0 stack top to the left
ROR - n-multiple rotation of the value of the A0 stack top to the right

Description

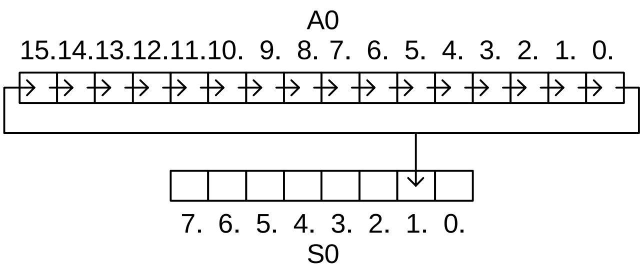
The instruction **ROL** with parameter n performs a circular shift of the lower 16 bits of the A0 stack top to the left. The instruction **ROR** with parameter n performs the same to the right. The parameter of the instruction specifies the number of unit shift, i.e. by how many bits the content is shifted. If the operand is greater than 15, modulo 16 is recalculated, so that there are never more than 15 shifts performed. When the parameter is zero, no shift is performed, only flags are set.

The upper 16 bits of the stack top (A0.16 to A0.31) are set to zero.

A schematic representation of the instruction **ROL** n:



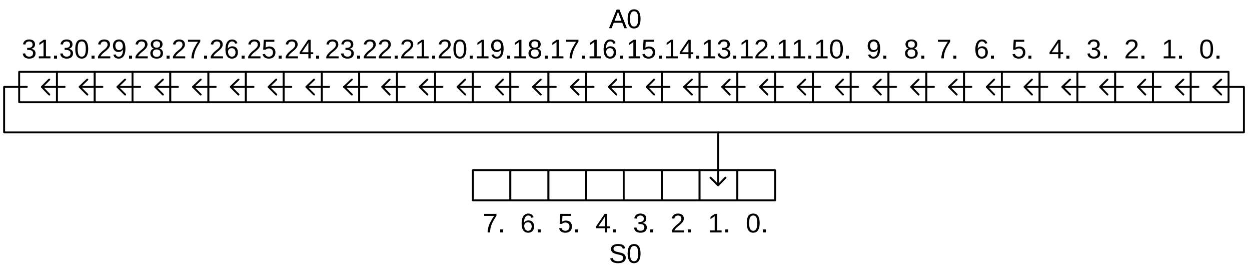
A schematic representation of the instruction **ROR** n:



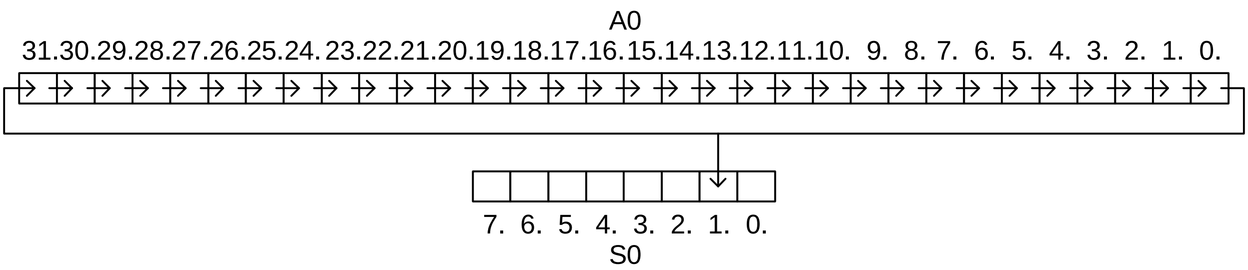
Instructions **ROL** and **ROR** without operand process the content of the stack top as the number of shifts. They then move the stack by one layer back and perform a circular shift of the new stack top (original layer A1) by the corresponding number of bits. If the number of shifts is greater than 31, modulo 32 is recalculated, so that there are never more than 31 shifts are performed. When the parameter is zero, no shift is performed, only flags are set.

The instruction **ROL** without operand performs a circular shift of all 32 bits of the A0 stack top to the left. The instruction **ROR** without operand performs the same to the right.

A schematic representation of the instruction **ROL**:



A schematic representation of the instruction **ROR**:



Flags

	.7	.6	.5	.4	.3	.2	.1	.0
S0	-	-	-	-	-	≤	CO	ZR

S0.0 (ZR) - zero value of result
1 - the result is 0

S0.1 (CO) - carry out
1 - the value of the last bit transferred in the circle (from the highest bit to the lowest at **ROL**, or from the lowest bit to the highest one at **ROR** log. 1 is transferred)

S0.2 (≤) - logical OR S0.0 OR S0.1

Note

Instructions **ROL** and **ROR** with parameter n are contained in the instruction set of the 32 bit model due to compatibility of the user programs transferred from a 16 bit model. We do not recommend using these instructions for new user programs but replace them with instructions **ROL** and **ROR** without operand.

SHL	**Shift number to the left**
SHR	**Shift number to the right**

Instruction	Input parameters									Result								
	stack									stack								
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
SHL							a	n		n	A7	A6	A5	A4	A3	A2	$a \langle\langle n$	
SHR							a	n		n	A7	A6	A5	A4	A3	A2	$a \rangle\rangle n$	

Operands

		dword udint
SHL	w/o operand	C
SHR	w/o operand	C

Function

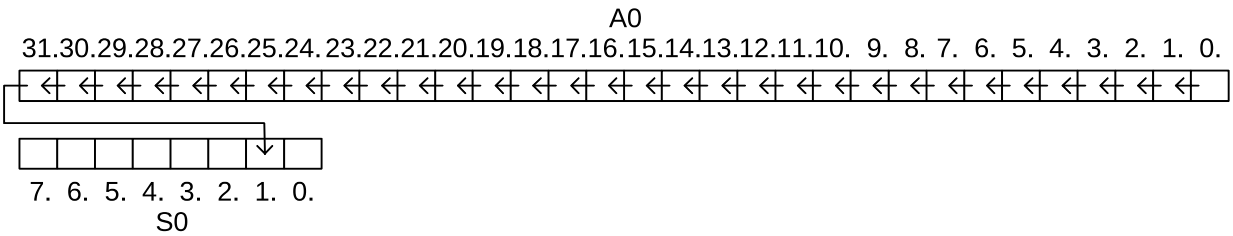
SHL - n-multiple shift of the content of the A0 stack top to the left
SHR - n-multiple shift of the content of the A0 stack top to the right

Description

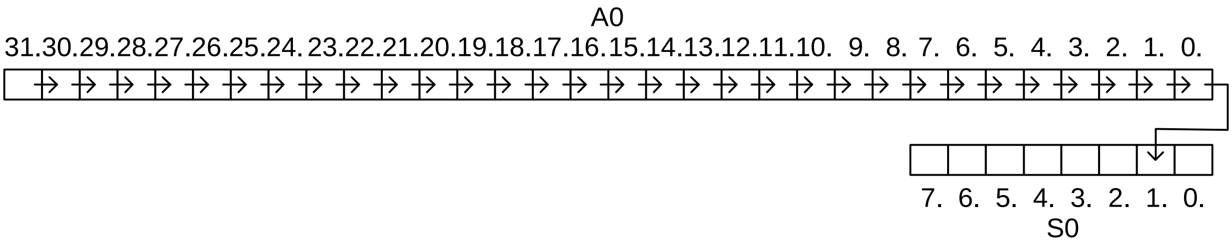
Instructions **SHL** and **SHR** process the content of the stack top as the number of shifts. They then shift the stack by one layer back and perform the shift of the new stack top (original layer A1) by the corresponding number of bits. If the number of shifts is greater than 31, modulo 32 is recalculated, so that there are never more than 31 shifts performed. When the parameter is zero, no shift is performed, only flags are set.

The instruction **SHL** performs the shift of all 32 bits of the A0 stack top to the left. The instruction **SHR** performs the same to the right. The empty bits are filled with log.0.

A schematic representation of the instruction **SHL**:



A schematic representation of the instruction **SHR**:



Flags

	.7	.6	.5	.4	.3	.2	.1	.0
S0	-	-	-	-	-	-	CO	-

S0.1 (CO) - carry out - the content of the last shifted bit

SWP	**Swap of two lower bytes of the stack top**
SWL	**Swap of words of the stack top**

Instruction	**Input parameters**									**Result**								
	stack									stack								
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
SWP								*abcd*									*abdc*	
SWL								*abcd*									*cdab*	

Operands

		word uint	dword udint
SWP	w/o operand	C	
SWL	w/o operand		C

Function

SWP - swap of two lower bytes of the stack top
SWL - swap of both words of the stack top

Description

The instruction **SWP** swaps the content of two lower bytes of the A0 stack top, the instruction **SWL** swaps the content of both words of the A0 stack top. The other stack levels remain unchanged.

3. COUNTERS, SHIFT REGISTERS, TIMERS, STEP SEQUENCER

CTU	**Upward counter**
CTD	**Downward counter**
CNT	**Bidirectional counter**

Instr.	**Input parameters**									**Result**								
	stack								ope-rand	stack								ope-rand
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
CTU							UP	RES	VAL0	A6	A5	A4	A3	A2	UPC	RES	VAL	VAL
CTD							DWN	SET	VAL0	A6	A5	A4	A3	A2	DNC	SET	VAL	VAL
CNT						UP	DWN	RES	VAL0	A6	A5	A4	A3	UPC	DNC	RES	VAL	VAL

UP - control variable for counting up (type bool)
DWN - control variable for counting down (type bool)
RES - zero-setting variable of the counter (type bool)
SET - setup variable of the counter (type bool)
VAL0 - numerical value of the counter before instruction (type according to the operand)
UPC - counting up transfer to a higher cascade (type bool)
DNC - counting down transfer to a higher cascade (type bool)
VAL - current numerical value of the counter (type according to the operand)

Operands

		uint	udint
CTU	R	C	C
CTD	R	C	C
CNT	R	C	C

Function

CTU - upward counter
CTD - downward counter
CNT - bidirectional counter

Description

The instruction **CTU** tests, whether the UP value has changed after activation of the instruction **CTU** or **CNT** from log.0 to log.1 (leading edge). If so, the counter value VAL addressed by the instruction is increased by one. If not, the content of the counter remains unchanged. At the same time, the instruction **CTU** moves the stack ahead and saves the current content of the counter on the new stack top. Provided carry was performed during counting (change of the content of the counter from the maximum value to 0), log.1 is saved to the variable UPC (all ones). Provided no carry was performed, UPC equals to log.0. The variable RES remains unchanged.

If the variable RES = log.1, the content of the counter is set to zero. Provided the leading edge is evaluated at the same time, setting to zero is of priority and the information on the edge is lost. But setting to zero does not effect the mechanism of the evaluation of leading edges, so that after RES is off, the first leading edge of the UP is processed normally.

The instruction **CTD** tests, whether the value of DWN has changed against the state of DWN after the last activated function **CTD** or **CNT** from log.0 to log.1 (leading edge), then the content of the counter which is addressed by the instruction, decreases by 1. If not, the content of the counter remains unchanged. At the same time, the instruction **CTD** moves the stack ahead and the content of the counter is saved on the new stack top. If carry takes place during counting, (change of the counter content from zero to the maximum value, log.1 is saved to the DNC (all ones). If carry did not take place, DNC = log.0. The variable SET remains unchanged.

If the variable SET = log.1, the counter content is set to the maximum value. If the leading edge is evaluated at the same time, SET is of higher priority and the leading edge data will be lost. But setting does not cancel the evaluation of leading edges. After the SET signal is off, the first leading edge of DWN is processed normally.

The instruction **CNT** tests inputs UP and DWN. If the value of UP changed against the state of UP after the last activated function **CTU** or **CNT** from log.0 to log.1 (leading edge), then the content of the counter which is addressed by the instruction, increases by 1. If the value of DWN changed against the state of DWN after the last activated function **CTD** or **CNT** from log.0 to log.1 (leading edge), then the content of the counter word, which is addressed by the instruction, decreases by 1. When both leading edges occur at the same time, the counter content remains unchanged (mutual elimination).

At the same time, the instruction **CNT** moves the stack ahead and the counter content is saved on the new stack top. Should carry up take place during counting (change of the counter content from the maximum value to 0), log.1 is saved to the variable UPC (all ones). Should carry down take place during counting (change of the counter content from 0 to the maximum value), log.1 is saved to the variable DNC (all ones). If carry does not take place, both variables will have zero value. The variable RES remains unchanged.

If the variable RES = log.1, the content of the counter is set to zero. If the leading edge is evaluated at the same time, setting to zero is of higher priority and the leading edge information will be lost. But setting to zero does not cancel the mechanism of leading edge evaluation, so after RES is off, the first leading edge of UP or DWN is processed normally.

Flags

	.7	.6	.5	.4	.3	.2	.1	.0
S0	-	-	-	-	-	≤	CO	ZR

S0.0 (ZR) - zero value of result
1 - counter value is zero
S0.1 (CO) - carry out
1 - counter valued exceeded the maximum
S0.2 (≤) - logical OR S0.0 OR S0.1

Note

During the first activation of the counter (after switching on or change of counter mode) log.1 is saved to the memory of the recent value of the control variable XT, so counting will begin by the first leading edge being really evaluated, not by a random transition.

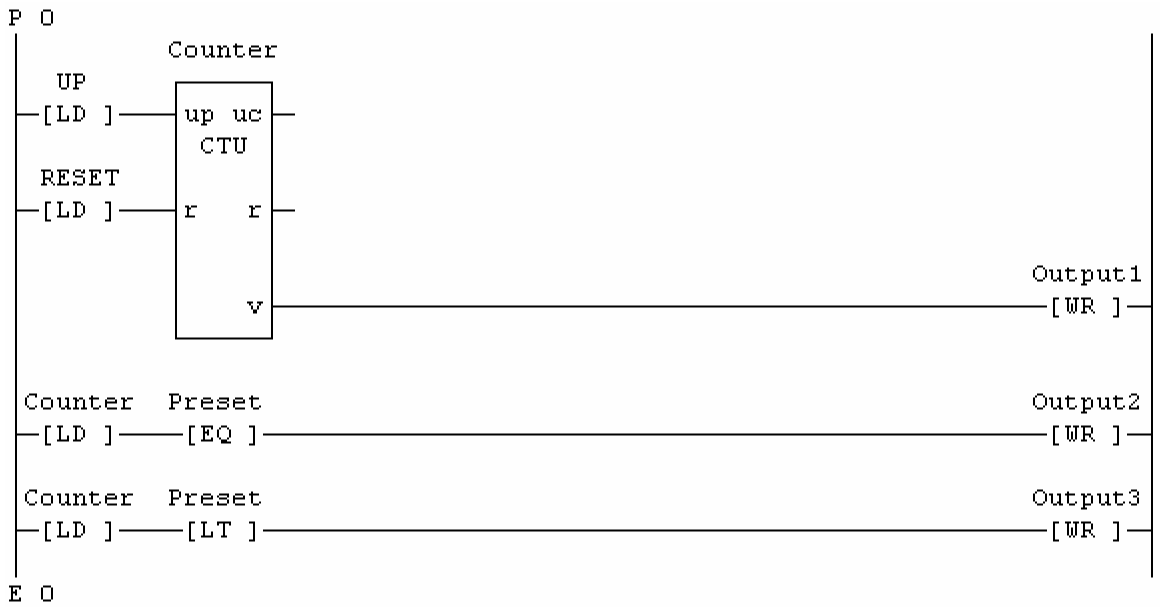
Any of the instructions **CTU**, **CTD**, **CNT**, **SFL** and **SFR** can work above one object, change of the instruction type does not activate initialization. But it is necessary to ensure that only one of these instructions for the same direction of counting is performed at one cycle (for example, it is not possible to use **CTU** or **CTD** and **CNT** twice at one cycle).

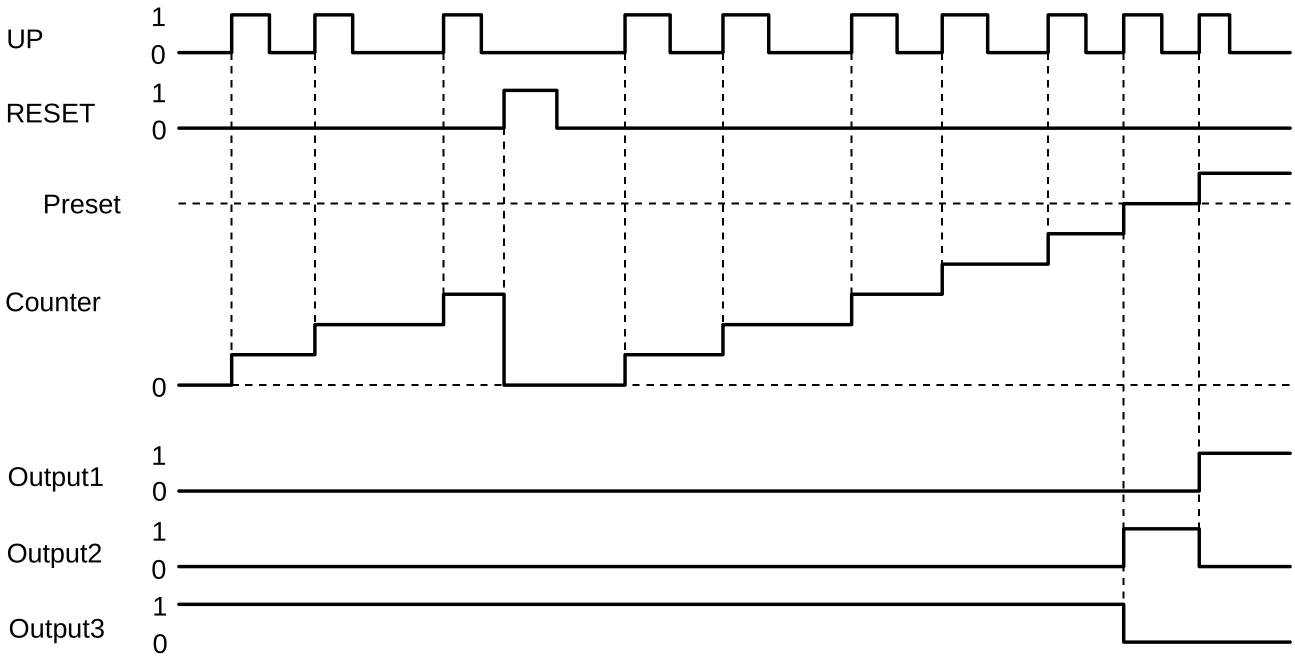
Examples

Let us realize an upward counter:

```
#reg uint Counter
#reg bool  UP, RESET, Output1, Output 2, Output 3
```

```
#def Preset  50
;
P 0
      LD    UP
      LD    RESET
      CTU   Counter
      WR    Output1
      LD    Counter
      EQ    Preset
      WR    Output2
      LD    Counter
      LT    Preset
      WR    Output3
E 0
```

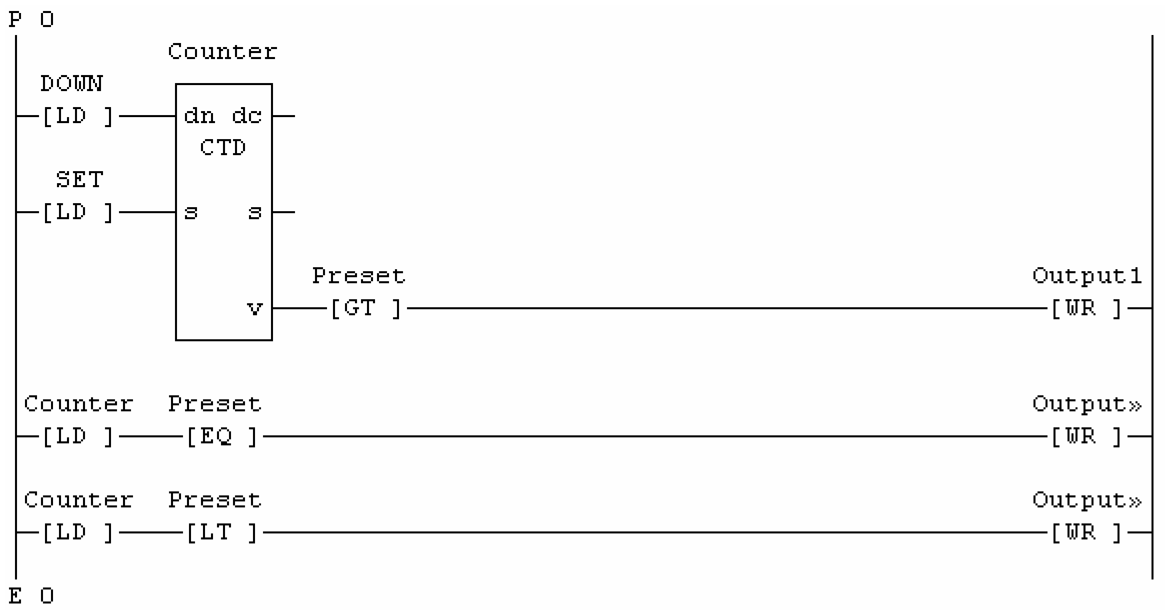


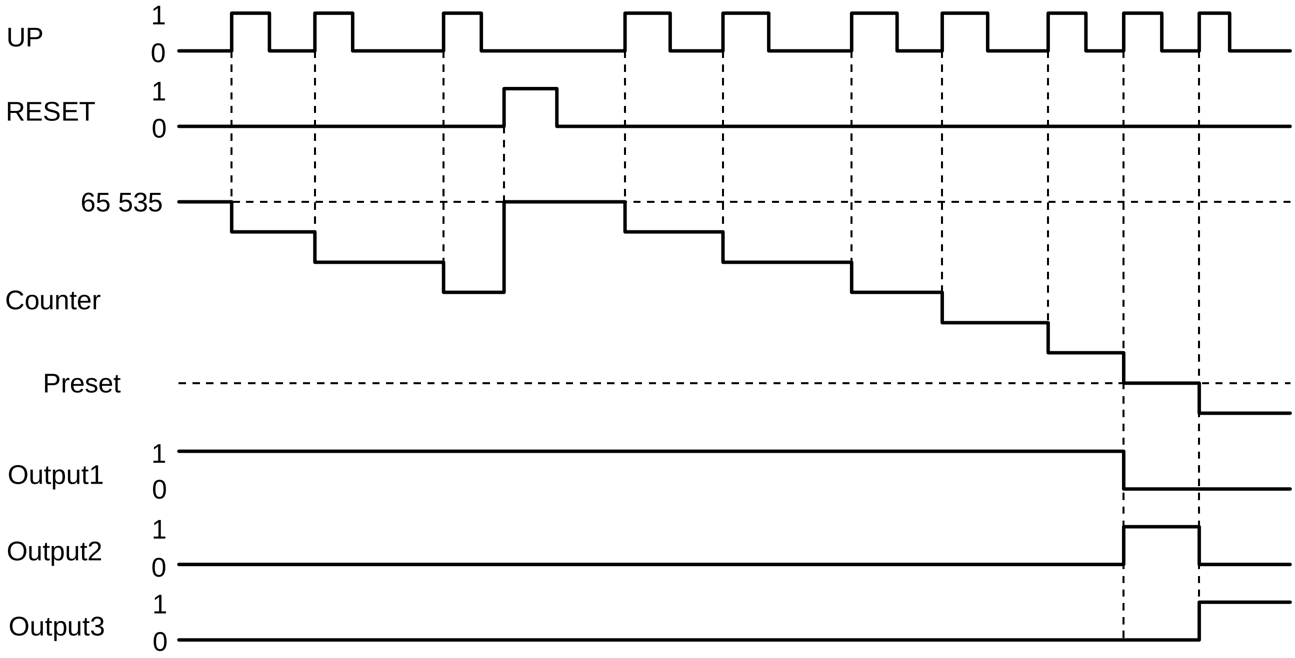


Behaviour of an upward counter according to an example

Let us realize a downward counter:

```
#reg uint Counter
#reg bool  DOWN, SET, Output1, Output2, Output3
#def Preset  65500
;
P 0
     LD    DOWN
     LD    SET
     CTD   Counter
     GT    Preset
     WR    Output1
     LD    Counter
     EQ    Preset
     WR    Output 2
     LD    Counter
     LT    Preset
     WR    Output 3
E 0
```

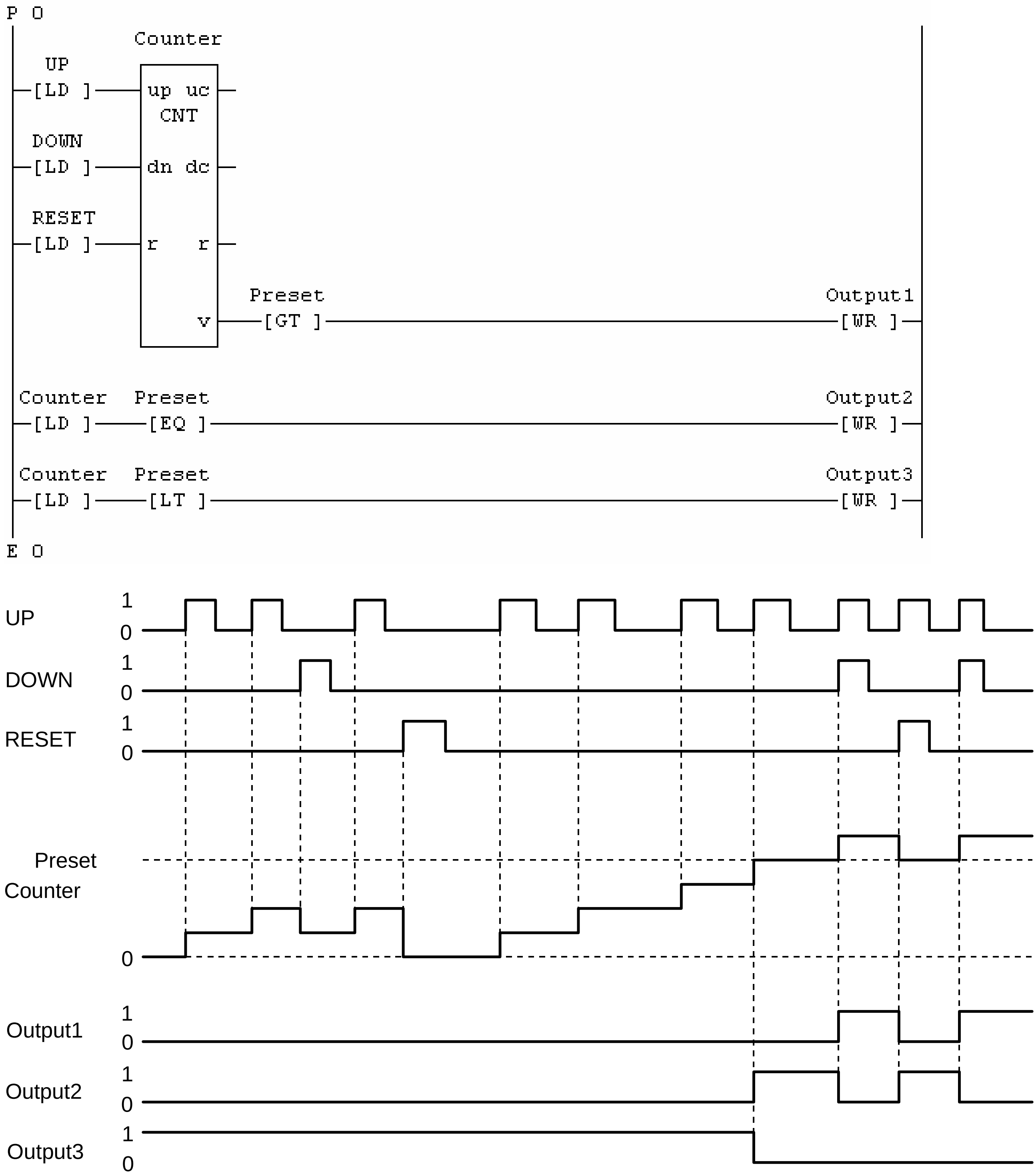




Behaviour of a downward counter according to an example

Let us realize a bidirectional counter:

```
#reg uint Counter
#reg bool  DOWN, UP, RESET, Output1, Output2, Output3
#def Preset 50
;
P 0
     LD   UP
     LD   DOWN
     LD   RESET
     CNT  Counter
     GT   Preset
     WR   Output1
     LD   Counter
     EQ   Preset
     WR   Output2
     LD   Counter
     LT   Preset
     WR   Output3
E 0
```



Behaviour of a bidirectional counter according to an example

SFL	**Shift register to the left**
SFR	**Shift register to the right**

Instr.	Input parameters									Result								
	stack								ope-rand	stack								ope-rand
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
SFL							CLC	DATAI	VAL0	A6	A5	A4	A3	A2	CLC	DATAO	VAL	VAL
SFR							CLC	DATAI	VAL0	A6	A5	A4	A3	A2	CLC	DATAO	VAL	VAL

CLC - control variable for shift (type bool)
DATAI - data input (type bool)
VAL0 - numerical value of the register before instruction (type according to the operand)
DATAO- data output (type bool)
VAL - current numerical value of the register (type according to the operand)

Operands

		word uint	dword udint
SFL	R	C	C
SFR	R	C	C

Function

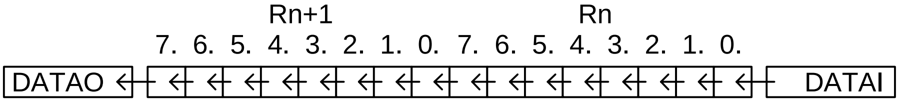
SFL - shift register value to the left
SFR - shift register value to the right

Description

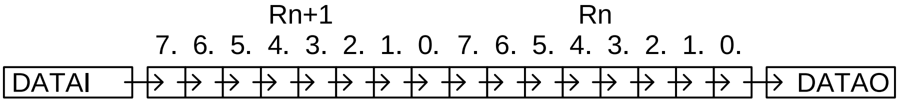
If the value of CLC changes against the state of CLC after the last activated function **SFL** or **SFR** from log.0 to log.1 (leading edge), then the entire content of the shift register shifts by one bit. After the instruction **SFL** the addressed register shifts by one bit to the left, the content of variable DATAI shifts to the position of the least significant bit and, from the position of the most significant bit, the content moves to the variable DATAO. After the instruction **SFR**, the addressed register shifts by one bit to the right, the content of variable DATAI shifts to the position of the most significant bit and, from the position of the least significant bit, the content shifts to the variable DATAO.

If the leading edge is not evaluated, the register content remains unchanged. At the same time, the instruction moves the stack ahead and saves the current content of the register to the new stack top. The variable CLC remains unchanged.

A schematic representation of the instruction **SFL** with word type operand (instruction **SFL** with long type operand behaves by analogy):



A schematic representation of the instruction **SFR** with **word** and **uint** types operand (instruction **SFR** with **dword** and **udint** types operand behaves by analogy):



Flags

	.7	.6	.5	.4	.3	.2	.1	.0
S0	-	-	-	-	-	≤	CO	ZR

S0.0 (ZR) - zero value of result
1 - register value is zero
S0.1 (CO) - carry
S0.2 (≤) - logical OR S0.0 OR S0.1

Note

Above one object, any of the instructions **CTU**, **CTD**, **CNT**, **SFL** and **SFR** can work, the change of the instruction type does not activate initialization. But it is necessary to ensure that only one of these instructions is performed at one cycle for the same direction of counting or shift (for example, it is not possible to use the **SFL** instruction twice, etc.)

TON	**Timer (on delay)**
TOF	**Timer (off delay)**

Instruction	Input parameters									Result								
	stack								ope-rand	stack								ope-rand
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
TON							XT	VAL	TIM							XT	YT	TIM
TOF							XT	VAL	TIM							XT	YT	TIM

XT - control variable (type bool)
VAL - numerical value of preset (type uint)
TIM - numerical value of timer (type uint) - units given by the parameter ***k***
YT - output variable, the result of comparison of the current value of timer to preset (type bool)

Operands

		uint
TON	R.k	C
TOF	R.k	C

k - time unit code (if it is not specified, then k = 0)
k = 0 - 10 ms, 1 - 100 ms, 2 - 1 s, 3 - 10 s

Function

TON - timing from start of input (leading edge is delayed)
TOF - timing from input OFF (trailing edge is delayed)

Description

The instruction **TON** tests the control variable XT. If XT = log.0, the timer is passive. If XT = log.1, it is active. The passive timer is set to zero and flags S0.4 and S0.5 are set to zero, too. If the preset is not zero, the entire register of S0 is set to zero. The active timer updates the time data and saves the result of the comparison with preset on the stack top. If preset is not reached, then YT = log.0. If the preset is reached or exceeded, then YT = log.1 (all ones). An overflow of timer range initiates setting of bits S0.4 and S0.5.

The instruction **TOF** tests the control variable XT. If XT = log.1, the timer is passive. If XT = log.0, it is active. The passive timer is set to zero and flags S0.4 and S0.5 are set to zero, too. If the preset is not zero, also flags S0.2 to S0.0 are set to zero and the timer output YT is set to the value of log.1 (A0 stack top). The active timer updates the time data and saves the result of the comparison with preset on the stack top. If the preset is not reached, then YT = log.1 (all ones). If the preset is reached or exceeded, then YT = log.0. An overflow of timer range initiates setting of bits S0.4 and S0.5.

Note

Above one object, just one type of timer instruction with the only time unit can be used. At any change of the instruction type or time unit, initialization is performed - the timer is set to zero.

Above one object, just one timer instruction can be active. Time-measuring system variables are updated only at the I/O scan (time is running by sudden changes). During the cycle, they have still the same value, so it is not important, in which location of the program the timer instruction is located. But if the timer instruction is omitted at one cycle, then it is not at the next I/O scan - the timer stops timing. It again starts timing, when the program passes through the timer instruction, but this value is disturbed by the corresponding time failure.

If the preset VAL is 0, the output of variable YT is identical to the control variable XT. The state of system flags S0 is not defined.

If the time unit **k** is approx. of the same value or less than the PLC cycle time, the function of flags S0.0 and S0.5 is not reliable (the timer value increases by bigger changes and the preset value or timer range is directly exceeded, so its teaching might not be detected). Flags S0.0 and S0.5 can be replaced by the test of the leading edge of flags S0.2 and S0.4.

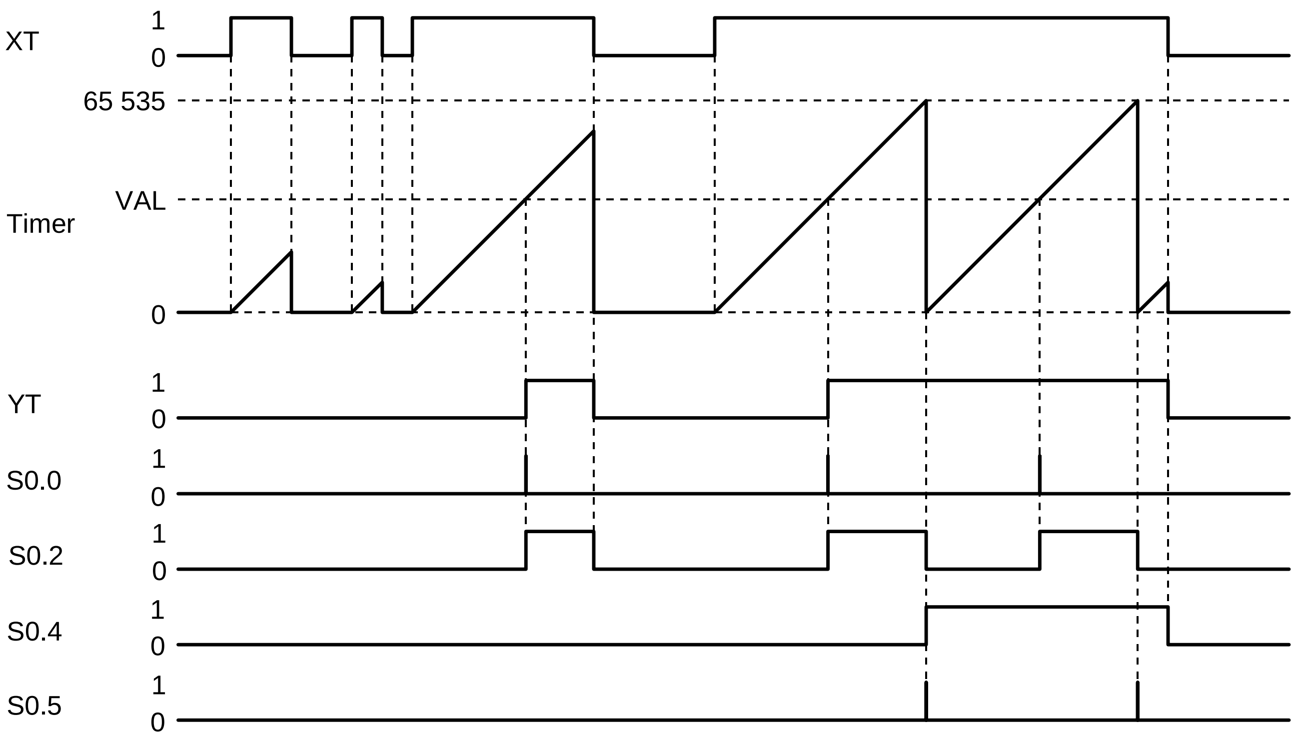
Flags

	.7	.6	.5	.4	.3	.2	.1	.0
S0	-	-	OC	OV	-	≤	CO	ZR

S0.0 (ZR) - preset reached
1 - preset reached at this cycle
S0.1 (CO) - preset exceeding
1 - preset exceeded
S0.2 (≤) - logical OR S0.0 OR S0.1
1 - preset reached or exceeded
S0.4 (OV) - exceeding of the maximum range of the timer
1 - the timer range was exceeded during the last activation
S0.5 (OC) - timer cascading
1 - the timer range was exceeded at this cycle

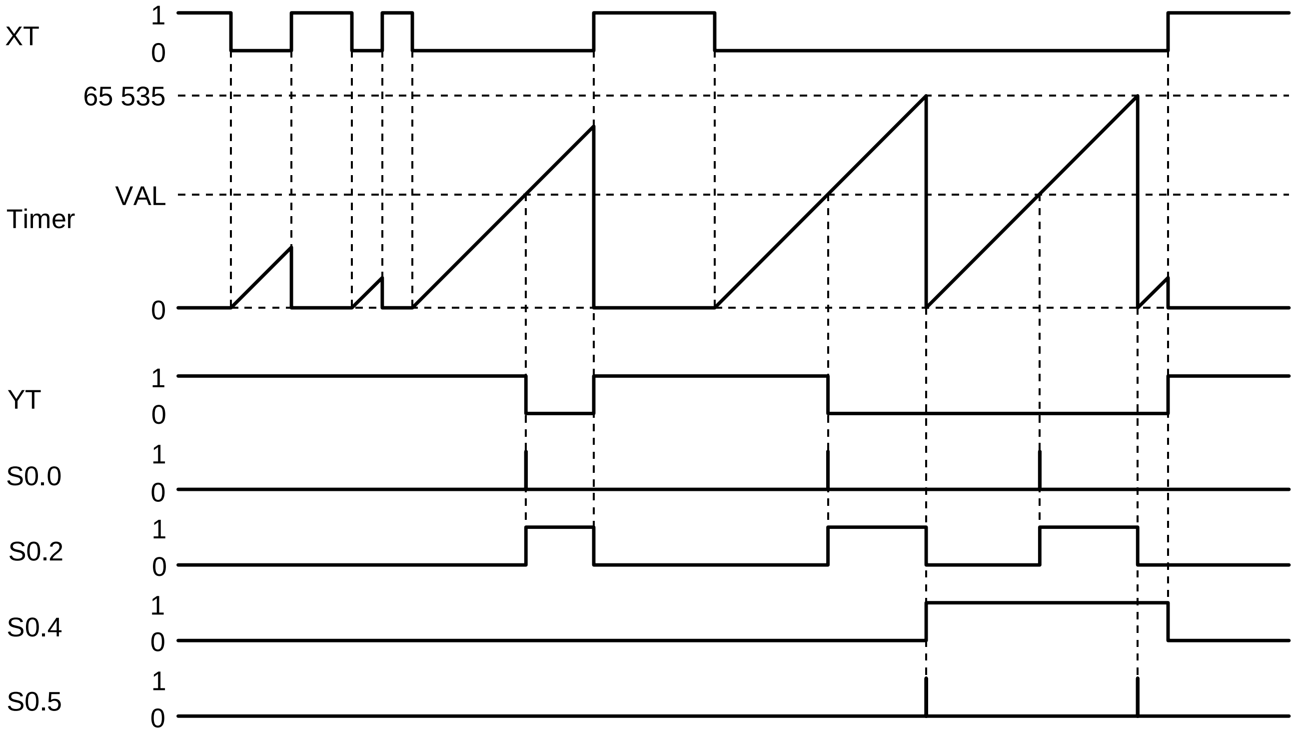
Examples

```
#reg uint Timer
#reg bool  XT, YT
#def VAL   5
#def sec   2
;
P 0
     LD    XT
     LD    VAL
     TON   Timer.sec
     WR    YT
E 0
```



Timing diagram of a TON timer

```
#reg uint Timer
#reg bool  XT, YT
#def VAL  5
#def sec  2
;
P 0
      LD    XT
      LD    VAL
      TOF   Timer.sec
      WR    YT
E 0
```



Timing diagram of a TOF timer

RTO Retentive timer on

Instruction	Input parameters									Result								
	stack								ope-rand	stack								ope-rand
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
RTO						XT	RT	VAL	TIM						YC	RT	YT	TIM

XT - control variable (type bool)
RT - zero setting variable (type bool)
VAL - preset numerical value (type uint)
TIM - numerical value of timer (type uint) - units given by the parameter ***k***
YC - maximum range overflow (type bool)
YT - output variable, the result of comparison of the current value of timer to preset (type bool)

Operands

		word uint
RTO	R.k	C

k - time unit code (if it is not specified, then k = 0)
k = 0 - 10 ms, 1 - 100 ms, 2 - 1 s, 3 - 10 s

Function

RTO - retentive timer on

Description

If the zero setting variable RT = log.1, the timer is passive. The passive timer sets the output of YT at the A0 stack top, carry from the timer YC and also the S0 flags are set to zero, too.

If the zero setting variable RT = log.0 and control variable XT = log.1, the timer is active. The active timer updates the time data and saves the result of the comparison with preset on the A0 stack top. If the preset value is not reached, then YT = log.0. If the preset value is reached or exceeded, then YT = log.1.

An overflow of timer range initiates setting of bits S0.4 and S0.5. The bit S0.5 is copied into all bits of the A2 layer (carry YC). It equals to log.1 only at the cycle where the overflow took place.

If the zero setting variable RT = log.0 and control variable XT = log.0, the timer is in the waiting state. The timer is not either timing in this state or setting to zero, but comparison with the preset value and setting the flags at S0 take place.

Flags

	.7	.6	.5	.4	.3	.2	.1	.0
S0	-	-	OC	OV	-	≤	CO	ZR

S0.0 (ZR) - preset value reached
1 - preset value reached at this cycle
S0.1 (CO) - exceeding of preset value
1 - preset value exceeded
S0.2 (≤) - logical OR S0.0 OR S0.1
1 - preset value reached or exceeded
S0.4 (OV) - exceeding of timer maximum range
1 - timer range exceeded during the last activation
S0.5 (OC) - timer cascading
1 - timer range exceeded at this cycle

Note

Above one object, just one type of timer instruction with the only time unit can be used. At any change of the instruction type or time unit, initialization is performed - the timer is set to zero.

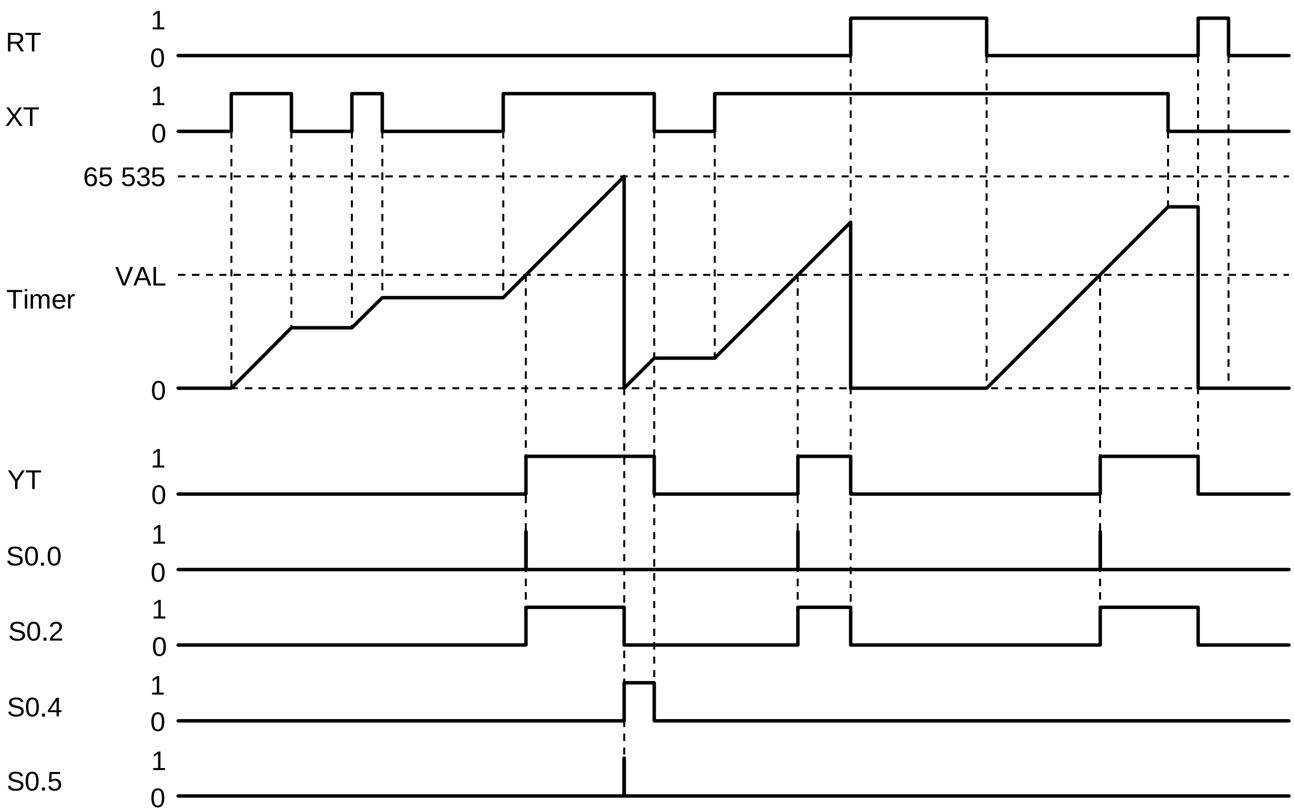
Above one object, just one timer instruction can be active. Time measuring system variables are updated only at the I/O scan (time is running by sudden changes). During the cycle, they have still the same value, so it is not important, in which location of the program the timer instruction is located. But if the timer instruction is omitted at one cycle, then it is not at the next I/O scan - the timer stops timing. It again starts timing, when the program passes through the timer instruction, but this value is disturbed by the corresponding time failure.

If the preset of value VAL is 0, the output variable YT is still log.1, only in case of RT = log.1, then YT = log.0. The state S0 system flags are not defined.

If the time unit **k** is approx. of the same value or less than the PLC cycle time, the function of flags S0.0 and S0.5 is not reliable (the timer value increases by bigger changes) and the preset value or timer range is directly exceeded, so its teaching might not be detected). Flags S0.0 and S0.5 can be replaced by the test of the leading edge of flags S0.2 and S0.4.

Example

```
#reg uint  Timer
#def bool  XT, RT, YT
#def VAL  5
#def sec 2
;
P 0
     LD   XT
     LD   RT
     LD   VAL
     RTO  Timer.sec
     WR   YT
E 0
```



Timing diagram of an RTO timer

IMP Pulse

Instruction	Input parameters									Result								
	stack								ope-rand	stack								ope-rand
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
IMP							XT	VAL	TIM							XT	YT	TIM

XT - control variable (type bool)
VAL - numerical value of preset (type uint)
*TIM - numerical value of timer (type uint) - units given by the parameter **k***
YT - output variable, the result of comparison of the current value of timer to preset (type bool)

Operands

		uint
IMP	R.k	C

k - time unit code (if it is not specified, then k = 0)
k = 0 - 10 ms, 1 - 100 ms, 2 - 1 s, 3 - 10 s

Function

IMP - pulse generating from the leading edge

Description

After initialization, the timer is passive. The passive timer is set to zero and also the flags S0.4 and S0.5 are set to zero. If the preset is not zero, flags S0.2 to S0.0 are set to zero.

The timer it is active after the first leading edge of the variable XT comes (transition from log.0 to log.1). The active timer updates the time data and saves the result of the comparison with the preset value on the A0 stack top. If the preset value is not reached, then YT = log.1 (all ones). If the preset value is reached, then YT = log.0, the timer becomes passive again and waits for a new leading edge of the variable XT.

The pulse length cannot be changed. The timer can be prematurely stopped only by initialization (system restart or change of timer mode - see note).

Flags

	.7	.6	.5	.4	.3	.2	.1	.0
S0	-	-	-	-	-	≤	-	ZR

S0.0 (ZR) - preset value reached
1 - preset value reached at this cycle
S0.2 (≤) - logical OR S0.0 OR S0.1

Note

Above one object, just one type of timer instruction with the only time unit can be used. At any change of the instruction type or time unit initialization is performed - the timer is set to zero.

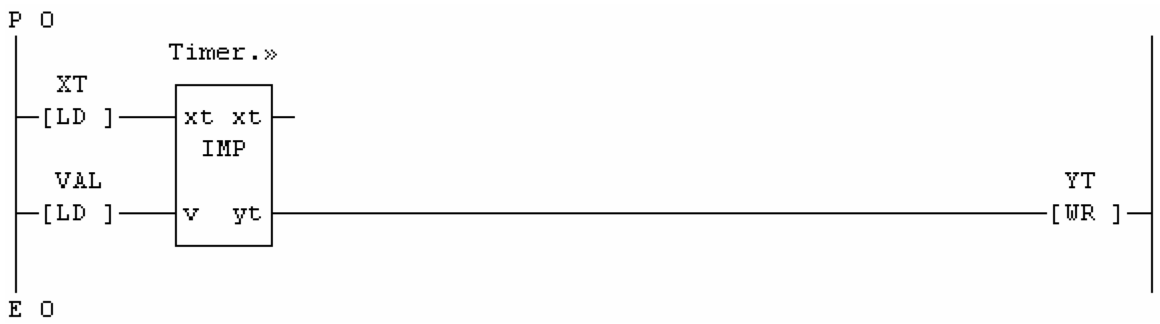
Above one object, just one timer instruction can be active. Time measuring systen variables are updated only at the I/O scan (time is running by sudden changes). During the cycle, they have still the same value, so it is not important, in which location of the program the timer instruction is located. But if the timer instruction is omitted at one cycle, then it is not at the next I/O scan - the timer stops timing. It again starts timing, when the program passes through the timer instruction, but this value is disturbed by the corresponding time failure.

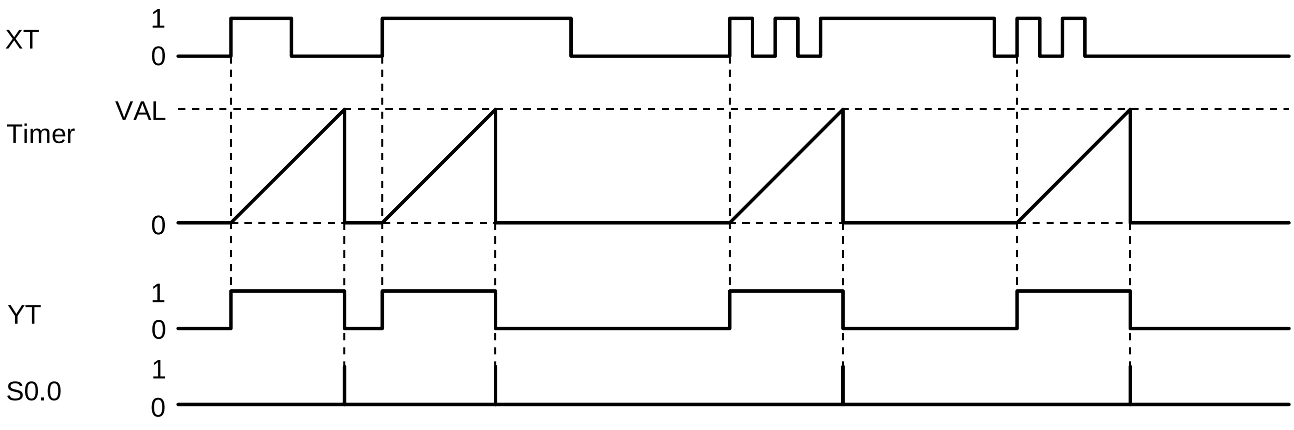
If the preset value of VAL is 0, the output variable YT is still log.0 (zero length pulse). The state of S0 system flags is not defined.

If the time unit **k** is approximately of the same value or less than the PLC cycle time, the S0.0 flag is not reliable (the timer value increases by bigger changes and the preset value is directly exceeded, so its reaching might not be detected). The S0.0 flag can be replaced by the S0.2 flag detecting also exceeding of the preset value.

Example

```
#reg uint  Timer
#def bool  XT, YT
#def VAL  5
#def sec 2
;
P 0
     LD    XT
     LD    VAL
     IMP   Timer.sec
     WR    YT
E 0
```





Timing diagram of an IMP timer

STE Step sequencer

Instruction	Input parameters									Result								
	stack								ope-rand	stack								ope-rand
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
STE								VEC	STP0								VAL	STP

VEC - conditional vector - a set of conditions for rotation of the state mask (type according to the operand)
STP0 - state of the step sequencer before instruction
VAL - resulting value of state mask (type according to the operand)
STP - current state of the step sequencer

Operands

		uint	udint
STE	R	C	C

Function

STE - step sequencer

Description

The instruction **STE** with **word** and **uint** type operand performs the following complex of activities. The lower byte of the addressed register (lower address) has the meaning of the state of the step sequencer. Only the lower 4 bits are of importance (values 0 to 15). The other 4 bits are ignored. The value of the lower 4 bits is converted to a mask 1 of 16 (status mask):

state (bits 3 - 0)	bit mask
0	00000000 00000001
1	00000000 00000010
:	:
14	01000000 00000000
15	10000000 00000000

After converting the state number to the state mask it is tested, if the condition vector has the one-condition bit at the corresponding position of the one in the mask. If so, the state number in the lower byte of the addressed register is increased by 1, the status mask shifts 1 bit to the left in a circle (the value of the highest bit goes to the lowest) and the S1.0 flag is set. If a carry took place (change of state from 15 to 0), the S1.1 flag is set, too. If the corresponding bit in the condition vector is zero, then neither the state number nor the status mask change and register S1 = 0. The updated value of the status mask is written on the A0 stack top. The updated state number is saved in the lower byte of the addressed register (lower address). The value is not corrected modulo 16, but assumes a value of up to 255. The content of the upper 4 bits from the lower byte (a value of 0 to 15) is saved to the lower byte of the addressed register (higher address) signifying a carry or overflow. Thus, the number of status mask "rotations" is saved here.

The instruction **STE** with **dword** and **udint** types operand performs the following complex of activities. The lower word of the addressed register (lower address) has the meaning of the state of the step sequencer. Only the lower five bits are of importance (values 0 to 31). The other 11 bits are ignored. The value of the lower five bits is converted to a mask 1 of 32 (status mask):

state (bits 4 - 0)	bit mask
0	00000000 00000000 00000000 00000001
1	00000000 00000000 00000000 00000010
:	:
30	01000000 00000000 00000000 00000000
31	10000000 00000000 00000000 00000000

After converting the state number to the status mask it is tested, if the condition vector has the one-condition bit at the corresponding position of the one in the mask. If so, the state number in the lower word of the addressed register is increased by 1, the status mask shifts 1 bit to the left in a circle (the value of the highest bit goes to the lowest) and the S1.0 flag is set. If a carry took place (change of state from 31 to 0), the S1.1 flag is set, too. If the corresponding bit of the condition vector is zero, then neither the state number nor the status mask change and the register S1 = 0. The updated value of the status mask is written on the A01 stack top. The updated state number is saved to the lower word of the addressed register (lower address). The value is not corrected modulo 32, but assumes a size of up to 65 535. The content of the upper 11 bits from the lower byte (a value of 0 to 15) is saved to the lower byte of the addressed register (higher address) signifying a carry or overflow. Thus, the number of status mask "rotations" is saved here.

Flags

	.7	.6	.5	.4	.3	.2	.1	.0
S1	-	-	-	-	-	-	OM	ST

S1.0 (ST) - transition in sequencer
1 - change of state and the status mask

S1.1 (OM) - carry in sequencer
1 - „rotation“ of the status mask (the one was transferred from the highest bit to bit 0)

Note

If the condition vector is still zero, the instruction **STE** works as a decoder number → mask „1 of n“.

If the condition vector contains all ones, the instruction **STE** performs mask rotation and incrementation of the number state at the same time.

4. ARITHMETIC INSTRUCTIONS

ADD	**Addition**
SUB	**Subtraction**

Instruction	Input parameters									Result								
	stack								ope-rand	stack								ope-rand
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
ADD								a	b								$a+b$	b
ADD w/o op.							a	b		b	A7	A6	A5	A4	A3	A2	$a+b$	
SUB								a	b								$a-b$	b
SUB w/o op.							a	b		b	A7	A6	A5	A4	A3	A2	$a-b$	

Operands

		usint sint	uint int	udint dint
ADD	X Y S D R	C	C	C
ADD	#			C
ADD	w/o operand			C
SUB	X Y S D R	C	C	C
SUB	#			C
SUB	w/o operand			C

Function

ADD - addition
SUB - subtraction

Description

The instruction **ADD** with operand adds the content of the given operand to the A0 stack top. The instruction **SUB** with operand subtracts the content of the given operand from the A0 stack top. The content of the other layers remains unchanged. The value on the stack is processed always as a 32-bit number regardless of the operand width.

The instruction **ADD** w/o operand moves the stack one level back and adds the original content of the A0 stack top to the stack top (originally A1).
The instruction **SUB** w/o operand moves the stack one level back and subtracts the original content of the A0 stack top from the stack top (originally A1).

Example

Realization of the expression $d = a + (b - c)$

```
#reg udint va, vb, vc, vd
;
P 0
     LD   vb
     SUB  vc          ;(b - c)
     ADD  va          ;a + ( )
     WR   vd
E 0
```

MUL	**Multiplication**
MULS	**Multiplication with sign**

Instruction	Input parameters									Result								
	stack								ope-rand	stack								ope-rand
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
MUL								a	b								$a \cdot b$	b
MUL w/o op.							a	b		b	A7	A6	A5	A4	A3	A2	$a \cdot b$	
MULS								a	b								$a \cdot b$	b
MULS w/o op.							a	b		b	A7	A6	A5	A4	A3	A2	$a \cdot b$	

Operands

		usint	sint	uint	int	udint	dint
MUL	X Y S D R	C		C		C	
MUL	#					C	
MUL	w/o operand					C	
MULS	X Y S D R		C		C		C
MULS	#						C
MULS	w/o operand						C

Function

MUL - multiplication
MULS - multiplication with sign

Description

Instruction **MUL** and **MULS** with operand multiplies the content of the A0 stack top by the content of the given operand. The result is saved on A0 the stack top. The content of the other stack levels remains unchanged.

The instructions **MUL** and **MULS** w/o operand multiply the content of the A1 layer by the content of the A0 layer. They then move the stack one level back and save the result on the new A0 stack top.

The instruction **MUL** considers the processed values as positive numbers, while the instruction **MULS** accepts the state of the highest bit of the value as a sign. The value on the stack is processed always as a 32-bit number regardless of the operand width.

Example

Realization of the expression $d = a + (b \cdot c)$

```
#reg udint  va, vb, vc, vd
;
P 0
     LD    vb
     MUL   vc          ;(b . c)
     ADD   va          ;a + ( )
     WR    vd
E 0
```

DIV, DID	**Division with reminder**
DIVL	**Division**
DIVS	**Division with sign**
MOD	**Division reminder**
MODS	**Division reminder with sign**

Instruction	**Input parameters**									**Result**								
	stack								ope-rand	stack								op.
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
DIV								a	b								M | a / b	b
DIV w/o op.							a	b		b	A7	A6	A5	A4	A3	A2	M | a / b	
DID								a	b	A6	A5	A4	A3	A2	A1	M	a / b	b
DID w/o op.							a	b								M	a / b	
DIVL								a	b								a / b	b
DIVL w/o op.							a	b		b	A7	A6	A5	A4	A3	A2	a / b	
DIVS								a	b								a / b	b
DIVS w/o op.							a	b		b	A7	A6	A5	A4	A3	A2	a / b	
MOD w/o op.							a	b		b	A7	A6	A5	A4	A3	A2	M	
MODS w/o op.							a	b		b	A7	A6	A5	A4	A3	A2	M	

M -division reminder (a % b)(type according to the result type)

Operands

		usint	sint	uint	int	udint	dint
DIV	X Y S D R	C					
DIV	#	C					
DIV	w/o operand	C					
DID	X Y S D R	C		C		C	
DID	#					C	
DID	w/o operand					C	
DIVL	X Y S D R	C		C		C	
DIVL	#					C	
DIVL	w/o operand					C	
DIVS	X Y S D R		C		C		C
DIVS	#						C
DIVS	w/o operand						C
MOD	w/o operand					C	
MODS	w/o operand						C

Function

DIV - division with reminder (usint / usint = usint)
DID - division with reminder
DIVL - division
DIVS - division with sign
MOD - division reminder
MODS- division reminder with sign

Description

The instruction **DIV** with operand divides the lowest byte of the A0 stack top by the content of the given operand. It saves the integral quotient in the lowest byte of the stack top, the reminder is saved in the second lowest byte. The content of the other stack levels remains unchanged.

The instruction **DID** with operand divides the content of the A0 stack top by the content of the given operand. It then moves the stack one level ahead and writes the integral quotient on the new A0 stack top, the reminder is saved to the A1 layer. The value on the stack is processed always as a 32-bit number regardless of the operand width.

The instructions **DIVL** and **DIVS** with operand divide the content of the A0 stack top by the content of the given operand. They write the integral quotient on the A0 stack top. The content of the other stack levels remains unchanged. The value on the stack is processed always as a 32-bit number regardless of the operand width.

The instruction **DIV** w/o operand divides the lowest byte of the A1 layer by the lowest byte of the A0 layer. It then moves the stack one level back and writes the integral quotient to the lowest byte of the stack top on the new stack top, the reminder is saved to the second lowest byte of the stack top.

The instruction **DID** w/o operand divides the content of the A1 layer by the content of the A0 layer. It writes the integral value on the A0 stack top, the reminder is saved at the A1 layer. The content of the other stack levels remains unchanged.

The instructions **DIVL** and **DIVS** w/o operand divide the content of the A1 layer by the content of the A0 layer. They then move the stack one level back and write the integral value on the new A0 stack top.

The instructions **MOD** and **MODS** w/o operand divide the content of the A1 layer by the content of the A0 layer. They then move the stack one level back and write the division reminder on the new A0 stack top.

The instructions **DIV**, **DID**, **DIVL**, **MOD** consider the values being processed as positive numbers, while the instructions **DIVS**, **MODS** accept the state of the highest bit of the value as a sign.

If division by zero is performed, the S0.0 bit is set to log.1 and error 16 is written to the S34 register. The stack top contains the value of $FFFFFFFF (all ones).

Flags

	.7	.6	.5	.4	.3	.2	.1	.0
S0	-	-	-	-	-	-	-	ZR

S0.0 (ZR) - division by zero
1 - division by zero performed, the result is not valid

S34 = 16 ($10) error of division by zero

Note

The instruction **DIV** is contained in the instruction file of the 32 bit model due to compatibility of the user programs transferred from a 16 bit model. We do not recommend using this instruction for new user programs.

The instruction **DID** computes the integral ration as well as the division reminder. If both results are not needed at the same time, we recommend the instruction **DIVL** or **MOD**, which are significantly faster.

Example

Realization of the expression $d = a + \frac{b}{c}$

```
#reg udint  va, vb, vc, vd
;
P 0
     LD    vb
     DIVL  vc         ;(b / c)
     ADD   va         ;a + ( )
     WR    vd
E 0
```

INR	**Incrementation**
DCR	**Decrementation**

Instruction	Input parameters									Result								
	stack								ope-rand	stack								ope-rand
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
INR									a									$a+1$
INR w/o op.								a									$a+1$	
DCR									a									$a-1$
DCR w/o op.								a									$a-1$	

Operands

		usint sint	uint int	udint dint
INR	X Y S R	C	C	C
INR	w/o operand			C
DCR	X Y S R	C	C	C
DCR	w/o operand			C

Function

INR - incrementation of the content by 1
DCR - decrementation of the content by 1

Description

The instruction **INR** with operand increments the content of the operand by 1. The content of the stack remains unchanged. The instruction does not set any flags.

The instruction **INR** w/o operand adds the value of 1 to the content of the stack top. The content of the other stack levels remains unchanged.

The instruction **DCR** with operand decrements the content of the operand by 1. The content of the stack remains unchanged. If the content of the operand after subtraction of 1 equals to 0, the S0.0 (ZR) flag is set. In connection with instructions **JZ** and **JNZ** program cycles can be realized very easy.

The instruction **DCR** w/o operand subtracts 1 from the content of the stack top. If the content of the stack top after subtraction of 1 equals to 0, the S0.0 (ZR) flag is set. The content of the other stack levels remains unchanged.

Flags

	.7	.6	.5	.4	.3	.2	.1	.0
S0	-	-	-	-	-	-	-	ZR

S0.0 (ZR) - zero value of result (they set instruction **DCR**)
1 - the result is 0

Example

Let us do 5x the same program sequence

```
#reg usint  Meter
;
P 0
     LD    5
     WR    Meter
Loop:                    ;cycle start
     :                   ;repeated program
     :
     DCR   Meter         ;test of number of repetitions
     JNZ   Loop
                         ;cycle end, Meter = 0
E 0
```

EQ	**Comparison (equality)**
LT	**Comparison (less than)**
LTS	**Comparison with sign (less than)**
GT	**Comparison (greater than)**
GTS	**Comparison with sign (greater than)**

Instruction	Input parameters									Result									
	stack								ope-rand	stack								op.
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
EQ								a	b								$a = b$?	b
EQ w/o op.							a	b		b	A7	A6	A5	A4	A3	A2	$a = b$?	
LT, LTS								a	b								$a < b$?	b
LT, LTS w/o op.							a	b		b	A7	A6	A5	A4	A3	A2	$a < b$?	
GT, GTS								a	b								$a > b$?	b
GT, GTS w/o op.							a	b		b	A7	A6	A5	A4	A3	A2	$a > b$?	

Operands

		usint	sint	uint	int	udint	dint
EQ	X Y S D R	C		C		C	
EQ	#					C	
EQ	w/o operand					C	
LT	X Y S D R	C		C		C	
LT	#					C	
LT	w/o operand					C	
LTS	X Y S D R		C		C		C
LTS	#						C
LTS	w/o operand						C
GT	X Y S D R	C		C		C	
GT	#					C	
GT	w/o operand					C	
GTS	X Y S D R		C		C		C
GTS	#						C
GTS	w/o operand						C

Function

EQ - value comparison with equality test
LT - value comparison with test less than ...
LTS - value comparison with sign with test less than ...
GT - value comparison with test greater than ...
GTS - value comparison with sign with test greater than ...

Description

The instructions **EQ**, **LT**, **LTS**, **GT**, **GTS** with operand are internally equal to each other. They compare the content of the stack top with operand, set the flags at S0 and they then write the truth result of the test - log.1 - on the stack top (all ones), if the test condition is fulfilled, or log.0, if the condition is not fulfilled.

The instructions **EQ**, **LT**, **LTS**, **GT**, **GTS** w/o operand are internally equal to each other. They compare the content of the A1 layer with the content of the A0 stack top, set the flags at S0, they move the stack one level back and then they write the truth result of the test -

log.1 (all ones) - on the new stack top, if the test condition is fulfilled, or log.0, if the condition is not fulfilled.

The instructions **EQ**, **LT**, **GT** consider the values being processed as positive numbers, while the instructions **LTS**, **GTS** accept the state of the highest bit of the value as a sign.

Flags

	.7	.6	.5	.4	.3	.2	.1	.0
S0	-	-	-	-	-	≤	CO	ZR

S0.0 (ZR) - comparison to match
0 - it is valid that $a \neq b$
1 - it is valid that $a = b$

S0.1 (CO) - carry out
0 - it is valid that $a \geq b$
1 - it is valid that $a < b$

S0.2 (≤) - logical OR S0.0 OR S0.1
0 - it is valid that $a > b$
1 - it is valid that $a \leq b$

CMP	**Comparison**
CMPS	**Comparison with sign**

Instruction	Input parameters									Result								
	stack								ope-rand	stack								ope-rand
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
CMP								*a*	*b*								*a*	*b*
CMPS w/o op.							*a*	*b*								*a*	*b*	

Operands

		usint	sint	uint	int	udint	dint
CMP	X Y S D R	C		C		C	
CMP	#					C	
CMP	w/o operand					C	
CMPS	X Y S D R		C		C		C
CMPS	#						C
CMPS	w/o operand						C

Function

CMP - comparison of values
CMPS - comparison of values with sign

Description

The instructions **CMP**, **CMPS** with operand compare the content of the stack top with operand and set the flags at S0.

The instructions **CMP**, **CMPS** w/o operand compare the content of the A1 layer with the content of the A0 stack top and set the flags at S0.

All these instructions do not change the content of the stack. For evaluation of the flags set at the S0 register, jump instructions **JZ**, **JNZ**, **JC**, **JNC**, **JB** and **JNB** can be advantageously used.

The instruction **CMP** considers the processed values as positive numbers, while the instruction **CMPS** accepts the state of the highest bit of the value as a sign.

Flags

	.7	.6	.5	.4	.3	.2	.1	.0
S0	-	-	-	-	-	≤	CO	ZR

S0.0 (ZR) - comparison to match
0 - it is valid that $a \neq b$
1 - it is valid that $a = b$

S0.1 (CO) - carry out
0 - it is valid that $a \geq b$
1 - it is valid that $a < b$

S0.2 (≤) - logical OR S0.0 OR S0.1
0 - it is valid that $a > b$
1 - it is valid that $a \leq b$

MAX	**Maximum**
MAXS	**Maximum with sign**
MIN	**Minimum**
MINS	**Minimum with sign**

Instruction	Input parameters									Result								
	stack									stack								
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
MAX							*a*	*b*		*b*	A7	A6	A5	A4	A3	A2	*MAX(a,b)*	
MAXS							*a*	*b*		*b*	A7	A6	A5	A4	A3	A2	*MAX(a,b)*	
MIN							*a*	*b*		*b*	A7	A6	A5	A4	A3	A2	*MIN(a,b)*	
MINS							*a*	*b*		*b*	A7	A6	A5	A4	A3	A2	*MIN(a,b)*	

Operands

		udint	dint
MAX	w/o operand	C	
MAXS	w/o operand		C
MIN	w/o operand	C	
MINS	w/o operand		C

Function

MAX - maximum from two values
MAXS- maximum from two values with sign
MIN - minimum from two values
MINS - minimum from two values with sign

Description

The instructions **MAX** and **MAXS** compare the content of the A1 layer with the content of the A0 layer. Then they move the stack one level back and write the value, which is greater, on the new A0 stack top.

The instructions **MIN** and **MINS** compare the content of the A1 layer with the content of the A0 layer. Then they move the stack one level back and write the value, which is less, on the new A0 stack top. The instructions **MAX** and **MIN** consider the values being processed as positive numbers, while instruction **MAXS** and **MINS** accept the state of the highest bit of the value as a sign.

Example

Value limitation within the range of –220 to +315

```
#def MINIMUM  -220
#def MAXIMUM  315
#reg udint  input,output
;
P0
      LD   MAXIMUM
      LD   input
      MIN                  ;top constraints
      LD   MINIMUM
      MAX                  ;bottom constraints
      WR   output
E0
```

ABSL	**Absolute value**
CSGL	**Sign change**
EXTB, EXTW	**Change of format with sign**

Instruction	Input parameters									Result								
	stack									stack								
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
ABSL								a									$\lvert a \rvert$	
CSGL								a									$-a$	
EXTB								a									a	
EXTW								a									a	

Operands

		dint
ABSL	w/o operand	C
CSGL	w/o operand	C
EXTB	w/o operand	C
EXTW	w/o operand	C

Function

ABSL - number absolute value
CSGL - sign change of a number
EXTB - change of number format from 8 bits to 32 bits with sign transfer
EXTW - change of number format from 16 bits to 32 bits with sign transfer

Description

The instruction **ABSL** performs the computation of the value of the stack top.

The instruction **CSGL** changes the sign of the value of the stack top.

The instruction **EXTB** performs the change of number format at the stack top from byte to long by copying the sign on the bit A0.7 to the bits A0.8 to A0.31.

The instruction **EXTW** performs the change of number format at the stack top from word to long by copying the sign on the bit A0.15 to the bits A0.16 to A0.31.

Example:

Absolute value of a number of byte width

```
#reg sint number      ;value with sign
;
P0
     LD   number      ;load number:  bytes A0 ... 0, 0, 0, number
     EXTB             ;conversion to long: A0 = number
     ABSL             ;absolute value
     WR   number      ;write number
E0
```

BIN , BIL Conversion from BCD format to binary
BCD, BCL Conversion from binary format to BCD

Instruction	Input parameters									Result								
	stack									stack								
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
BIN								NBCD									NBIN	
BIL								NBCD		-	A7	A6	A5	A4	A3	A2	NBIN	
BCD								NBIN									NBCD	
BCL								NBIN		A6	A5	A4	A3	A2	A1		NBCD	

NBCD - decimal number in BCD format
*(**BIN**, **BCD** - range 0 to 99 999 999; **BIL**, **BCL** - range 0 to 4 294 967 295)*
NBIN - number in binary format (type udint)

Operands

		udint
BIN	w/o operand	C
BIL	w/o operand	C
BCD	w/o operand	C
BCL	w/o operand	C

Function

BIN - conversion of decimal number in BCD code to binary format (8 BCD digits)
BIL - conversion of decimal number in BCD code to binary format (full range of udint)
BCD - conversion of number in binary format to decimal in BCD code (8 BCD digits)
BCL - conversion of number in binary format to decimal in BCD code (full range of udint)

Description

The instruction **BIN** processes the A0 stack top as an eight-digit decimal number in the BCD format (each digit is encoded in binary on four bits), it converts it to the binary system and saves back to A0. The values of the other stack layers remain unchanged. The numerical range of the converted numbers is 0 to 99 999 999.

The instruction **BIL** processes the A1 and A0 layers as a ten-digit decimal number in the BCD format, converts it to the binary system, moves the stack one level back and the result is saved on the A0 stack top. The numerical range of the converted numbers is 0 to 4 294 967 295.

The instruction **BCD** processes the A0 stack top as a binary number of 32 bit width, converts it to a decimal number in the BCD code and its 8 lower digits are saved on the A0 stack top. The value of the fifth digit is saved to the S0 register on the bits S0.6 to S0.4. The values of other stack levels remain unchanged. The range of converted numbers is 0 to 99 999 999.

The instruction **BCL** processes the A0 stack top as a binary number of 32 bit width, converts it to a decimal number in the BCD code, moves the stack one level ahead and the result is saved to the A1 and A0 layers. The range of converted numbers is 0 to 4 294 967 295.

Flags of BCD instruction

	.7	.6	.5	.4	.3	.2	.1	.0
S0	-	D5.2	D5.1	D5.0	-	-	-	-

S0.6 to S0.4 (D5.2 to D5.0) - the highest digit of the converted number in the BCD code (max. value of the number is 6)

Note

The instruction **BCD** set the flags at S0 due to compatibility of the user programs transferred from a 16 bit model. We do not recommend using of these flags in new user programs.

Examples

Conversion of a number in BCD to binary number

```
#reg udint Decim, Binar
;
P 0
     LD   Decim
     BIN
     WR   Binar
E 0

#reg usint DecimH          ;the highest digits (10. and 9.)
#reg udint DecimL          ;another 8 digits (8. to 1.)
#reg udint Binar
;
P 0
     LD   DecimH
     LD   DecimL
     BIL
     WR   Binar
E 0
```

Conversion of a binary number to BCD

```
#reg udint Decim, Binar
;
P 0
     LD   Binar
     BCD
     WR   Decim
E 0

#reg usint DecimH          ;the highest digits (10. and 9.)
#reg udint DecimL          ;another 8 digits (8. to 1.)
#reg udint Binar
;
P 0
     LD   Binar
     BCL
     WR   DecimL
     POP  1
     WR   DecimH
E 0
```

5. STACK OPERATIONS

POP	**Shift stack**

Instruction	**Input parameters**									**Result**								
	stack								ope-rand	stack								ope-rand
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
POP									*n*					*n x*				*n*

Operands

		dword udint dint
POP	n	C

n - number of stack shifts (–7 to 7)

Function

POP - n-multiple reverse shift stack

Description

The instruction **POP** shifts the stack back by the specified number of levels. The instruction performs the reverse rotation of the stack, so it is possible to return to the value shifted out from the A0 stack top at any time. If we need to shift the stack ahead, we will enter the number of rotations with sign –.

CHG, CHGS	**Change of active stack**
NXT	**Activation of next stack**
PRV	**Activation of previous stack**

Operands

		dword udint dint
CHG	n	C
CHGS	n	C
NXT	w/o operand	C
PRV	w/o operand	C

n - mark the chosen stack (0 to 7)

Function

CHG - activation of selected stack
CHGS - activation of selected stack with backing up S0 and S1
NXT - activation of next stack in the row with backing up S0 and S1
PRV - activation of previous stack in the row with backing up S0 and S1

Description

The instruction **CHG** activates the chosen stack specified by the parameter n, which assumes values 0 to 7, which represents stacks A to H. The instruction **CHGS** additionally performs also simultaneous saving and popping of the state of the system registers S0 and S1. The values of these registers are saved at the stack just left and the registers S0 and S1 in the scratchpad are overwritten by the values, which were saved at the activated stack just left.

The instructions **NXT** and **PRV** activate the stack according to the following table:

Active stack before instruction	Active stack after instruction NXT	Active stack after instruction PRV
A (0)	B (1)	H (7)
B (1)	C (2)	A (0)
C (2)	D (3)	B (1)
D (3)	E (4)	C (2)
E (4)	F (5)	D (3)
F (5)	G (6)	E (4)
G (6)	H (7)	F (5)
H (7)	A (0)	G (6)

The instruction **NXT** and **PRV** perform saving and popping of the state of the system registers S0 and S1.

Flags

The instructions **CHGS**, **NXT** and **PRV** save the values of S0 and S1 to the stack being just left and the registers S0 and S1 are overwritten by the values saved at the stack being just activated.

LAC	**Load value from the top of chosen stack**
WAC	**Write value on the top of chosen stack**

Instruction	**Input parameters**									**Result**								
	stack								ope-rand	stack								ope-rand
LAC	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
									n	A6	A5	A4	A3	A2	A1	A0	*a*	*n*
	m7	m6	m5	m4	m3	m2	m1	m0		m7	m6	m5	m4	m3	m2	m1	m0	
								a		*a*	m7	m6	m5	m4	m3	m2	m1	
WAC	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
								a	*n*								*a*	*n*
	m7	m6	m5	m4	m3	m2	m1	m0		m7	m6	m5	m4	m3	m2	m1	m0	
										m6	m5	m4	m3	m2	m1	m0	*a*	

n - stack number (0 to 7)
m - stack name (A to H)

Operands

		dword udint dint
LAC	n	C
WAC	n	C

n - mark the chosen stack (0 to 7)

Function

LAC - load values from the top of chosen stack and shift
WAC - write value on the top of chosen stack and shift

Description

The instruction **LAC** loads the value of the stack top specified by the parameter n, which assumes values 0 to 7, which represents stacks A to H, on the top of the active stack. Against the active stack, the instruction behaves in the same manner as the instruction **LD**, before writing the value on its top, it performs shift stack one level ahead. The chosen stack shifts after the operation one level back and a new value on its stack is ready to be loaded.

In connection with the **WAC** instruction, the chosen stack behaves as a swapping stack of type LIFO (last in, first out), this means that the value, which is written by the **WAC** instruction as the last, is loaded by the **LAC** instruction as the first.

The instruction **WAC** writes the value of the top of the active stack on the stack top specified by the parameter n, which assumes values 0 to 7, which represents stacks A to H. Against the active stack, the instruction behaves in the same manner as the instruction **WR**, it does not change its content. Against the chosen stack, the instruction behaves in the same manner as the instruction **LD**, before writing the value on its top, it performs shift stack one level ahead.

In connection with the **LAC** instruction, the chosen stack behaves as a swapping stack of type LIFO (last in, first out), this means that the value, which is written by the **WAC** instruction as the last, is loaded by the **LAC** instruction as the first.

The instruction **WAC** can be used also for preparing the parameters for instructions, which process more stack layers. If we acquire the values of these parameters within an extensive program from its various locations, we can put them gradually on the chosen stack and after that, the parameters are ready for processing by a simple switchover of the stacks.

PSHB, PSHW, PSHL, PSHQ	**Save value on system stack**
POPB, POPW, POPL, POPQ	**Pop value from system stack**

Instruction	**Input parameters**									**Result**								
	stack								stack	stack								stack
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
PSHB								*a*		*a*	A7	A6	A5	A4	A3	A2	A1	*a*
PSHW								*a*		*a*	A7	A6	A5	A4	A3	A2	A1	*a*
PSHL								*a*		*a*	A7	A6	A5	A4	A3	A2	A1	*a*
PSHQ							*a*			*a*		A7	A6	A5	A4	A3	A2	*a*
POPB									*a*	A6	A5	A4	A3	A2	A1	A0	*a*	
POPW									*a*	A6	A5	A4	A3	A2	A1	A0	*a*	
POPL									*a*	A6	A5	A4	A3	A2	A1	A0	*a*	
POPQ									*a*	A5	A4	A3	A2	A1	A0	*a*		

Operands

		byte usint sint	word uint int	dword udint dint	real	lreal
PSHB	w/o operand	C				
PSHW	w/o operand		C			
PSHL	w/o operand			C	C	
PSHQ	w/o operand					C
POPB	w/o operand	C				
POPW	w/o operand		C			
POPL	w/o operand			C	C	
POPQ	w/o operand					C

Function

PSHB- saving of data of byte, usint and sint type on the stack
PSHW- saving of data of word, uint and int type on the stack
PSHL - saving of data of dword, udint, dint and real type on the stack
PSHQ- saving of data of lreal type on the stack
POPB- popping of data of byte, usint and sint type from the stack
POPW- popping of data of word, uint and int type from the stack
POPL - popping of data of dword, udint, dint type from the stack
POPQ- popping of data of lreal type from the stack

Description

The instruction **PSHB** saves the content of the lowest byte of the A0 stack top on the stack. It then moves the stack one level back.

The instruction **PSHW** saves the content of the lower word of the A0 stack top on the stack. It then moves the stack one level back.

The instruction **PSHL** saves the content of the A0 stack top on the stack. It then moves the stack one level back.

The instruction **PSHQ** saves the content of the A01 double-layer on the stack. It then moves the stack two levels back.

The instruction **POPB** moves the stack one level ahead, pops the data of 8 bit width from the stack and saves them to the lowest byte of the A0 stack top. The other bytes of the top are set to zero.

The instruction **POPW** moves the stack one level ahead, pops the data of 16 bit width from the stack and saves them to the lower word of the A0 stack top. The upper word of the top is set to zero.

The instruction **POPL** moves the stack one level ahead, pops the data of 32 bit width from the stack and saves them on the A0 stack top.

The instruction **POPQ** moves the stack by two levels ahead, pops the data of 64 bit width from the stack and saves them to the A01 double-layer.

The instruction use higher languages for parameter preparation of function blocks. They can be used also for temporary deferring of the stack top content.

Attention! It is necessary, that the instructions are used in pairs, i.e. for one instruction **PSHB** one instruction **POPB**, for one instruction **PSHW** one instruction **POPW**, etc.

6. JUMP AND CALL INSTRUCTIONS

JMP	**Jump**
JMD	**Jump conditional to non-zero value of the stack top**
JMC	**Jump conditional to zero value of the stack top**
JMI	**Indirect jump**

Operands

JMP	Ln	C
JMD	Ln	C
JMC	Ln	C
JMI	Ln	C
JMI	w/o operand	C

Function

JMP - unconditional jump to the label L n
JMD - jump to label L n conditional to non-zero value of the A0 stack top
JMC - jump to label L n conditional to zero value of the A0 stack top
JMI - unconditional jump to label L n, the number n specifies the A0 stack top

Description

The instruction **JMP** unconditionally directs the program to the instruction **L** n.

The instruction **JMD** behaves as the instruction **JMP** only in such a case, that the A0 stack top is not 0 (logical OR of all 32 bits A0 is log.1). If this condition is not fulfilled, the instruction is ignored and the program continues performing the next immediately following instruction.

The instruction **JMC** behaves as the instruction **JMP** only in such a case, that the A0 stack top is 0 (logical OR of all 32 bits A0 is log.0). If this condition is not fulfilled, the instruction is ignored and the program continues performing the next immediately following instruction.

The instruction **JMI** unconditionally directs the program to the instruction **L** n, number of which (n) is contained at the A0 stack top. The instruction **JMI** with operand Ln performs jump to the label specified by this operand in such a case, that the label specified by the number at the stack top does not exist. In the user program, this can be treated as an error state, which otherwise would result in stopping the PLC due to the jump to a non-existing label.

JZ	**Jump conditional to non-zero value of flag ZR**
JNZ	**Jump conditional to zero value of flag ZR**
JC	**Jump conditional to non-zero value of flag CO**
JNC	**Jump conditional to zero value of flag CO**
JB	**Jump conditional to non-zero value of flag S0.2**
JNB	**Jump conditional to zero value of flag S0.2**
JS	**Jump conditional to non-zero value of flag S1.0**
JNS	**Jump conditional to zero value of flag S1.0**

Operands

JZ	Ln	C
JNZ	Ln	C
JC	Ln	C
JNC	Ln	C
JB	Ln	C
JNB	Ln	C
JS	Ln	C
JNS	Ln	C

Function

JZ - jump to label L n conditional to non-zero value of flag of equality ZR (S0.0)
JNZ - jump to label L n conditional to zero value of flag of equality ZR (S0.0)
JC - jump to label L n conditional to non-zero value of flag of carry CO (S0.1)
JNC - jump to label L n conditional to zero value of flag of carry CO (S0.1)
JB - jump to label L n conditional to non-zero value of flag of carry S0.2
JNB - jump to label L n conditional to zero value of flag of carry S0.2
JS - jump to label L n conditional to non-zero value of flag S1.0
JNS - jump to label L n conditional to zero value of flag S1.0

Description

The instructions **JZ**, **JNZ**, **JC**, **JNC**, **JB** and **JNB** are primarily used for easy evaluation of comparison results by the instructions **CMP**, **CMPS**, **CMF**, **CMDF**. The instructions **JS**, **JNS** are primarily used for easy evaluation of table instructions results and all other instructions, where the S1.0 flag is used by them as the flag of correctness of the performed operation.

The instruction **JZ** behaves as the instruction **JMP** only in such a case, that the flag of equality ZR (S0.0) is log.1.

The instruction **JNZ** behaves as the instruction **JMP** only in such a case, that the flag of equality ZR (S0.0) is log.0.

The instruction **JC** behaves as the instruction **JMP** only in such a case, that the flag of carry CO (S0.1) is log.1.

The instruction **JNC** behaves as the instruction **JMP** only in such a case, that the flag of carry CO (S0.1) is log.0.

The instruction **JB** behaves as the instruction **JMP** only in such a case, that the flag S0.2 is log.1.

The instruction **JNB** behaves as the instruction **JMP** only in such a case, that the flag S0.2 is log.0.

The instruction **JS** behaves as the instruction **JMP** only in such a case, that the S1.0 flag is log.1.

The instruction **JNS** behaves as the instruction **JMP** only in such a case, that the S1.0 flag is log.0.

If the corresponding condition is not fulfilled, the instruction is ignored and the program continues performing the next immediately following instruction.

Examples

Let us compare the content of registers *value1* and *value2* and let us realize a jump in a case, that the content of *value1* equals to the content of *value2*

```
     LD   value1
     CMP  value2
     JZ   jump
      :              ;value1 ≠ value2
jump:
      :              ;value1 = value2
```

Let us do 6x the same part of program

```
     LD   6
     WR   index      ;index = 6
jump:
      :              ;cycle body
     DCR  index      ;index = index - 1
     JNZ  jump       ;index = 0 ?
      :              ;yes, cycle ended
```

Let us compare the content of registers *value1* and *value2* and let us realize a jump in a case, that the content of *value1* is not greater than the content of *value2*

```
     LD   value1
     CMP  value2
     JC   jump
      :              ;value1 > value2
jump:
      :              ;value1 ≤ value2
```

Let us look for an item with the content of 4 in the table *Tab* and let us realize a jump in a case, that the item is found

```
     LD   4
     FTB  Tab        ;searching for item with content of 4
     JS   jump
      :              ;item with this content was not found
jump:
      :              ;item was found, index is at the stack top
```

CAL	**Subroutine call**
CAD	**Call conditional to non-zero value of the stack top**
CAC	**Call conditional to zero value of the stack top**
CAI	**Indirect subroutine call**

Operands

CAL	Ln	C
CAD	Ln	C
CAC	Ln	C
CAI	Ln	C
CAI	w/o operand	C

Function

CAL - unconditional subroutine call, specified by label L n
CAD - subroutine call specified by label L n conditional to non-zero value of the A0 stack top
CAC - subroutine call specified by label L n conditional to zero value of the A0 stack top
CAI - unconditional subroutine call specified by label L n, number of which (n) specifies the A0 stack top

Description

The instruction **CAL** unconditionally calls a subroutine beginning with instruction **L** n.

The instruction **CAD** behaves as the instruction **CAL** only in such a case, that the A0 stack top is not 0 (logical OR of all 32 bits A0 is log.1). If this condition is not fulfilled, the instruction is ignored and the program continues performing the next immediately following instruction.

The instruction **CAC** behaves as the instruction **CAL** only in such a case, that the A0 stack top is 0 (logical OR of all 32 bits A0 is log.0). If this condition is not fulfilled, the instruction is ignored and the program continues performing the next immediately following instruction.

The instruction **CAI** unconditionally calls a subroutine beginning with instruction **L** n, number of which (n) contains the A0 stack top. The instruction **CAI** with operand Ln performs jump to the label specified by this operand in such a case, that the label specified by the number at the stack top does not exists. In the user program, this can be treated as an error state, which otherwise would result in stopping the PLC due to the jump to a non-existing label.

Note

Each subroutine called must be ended with the instruction **RET**, which returns the program to the instruction immediately following after the instruction for subroutine call. In case this condition is not fulfilled, the PLC stops the program run and reports an error. The number of subroutine nesting (subroutine call within another subroutine) is 8 as maximum.

RET	**Return from subroutine**
RED	**Return conditional to non-zero value of the stack top**
REC	**Return conditional to zero value of the stack top**

Operands

RET	w/o operand	C
RED	w/o operand	C
REC	w/o operand	C

Function

RET - unconditional return from subroutine
RED - return from subroutine conditional to non-zero value of the A0 stack top
REC - return from subroutine conditional to zero value of the A0 stack top

Description

The instruction **RET** unconditionally ends the subroutine and returns the control to the instruction immediately following after the call instruction, by which the subroutine was called.

The instruction **RED** behaves as the instruction **RET** only in such a case, that the A0 stack top is not 0 (logical OR of all 32 bits A0 is log.1). If this condition is not fulfilled, the instruction is ignored and the program continues performing the next immediately following instruction.

The instruction **REC** behaves as the instruction **RET** only in such a case, that the A0 stack top is 0 (logical OR of all 32 bits A0 is log.0). If this condition is not fulfilled, the instruction is ignored and the program continues performing the next immediately following instruction.

L Label

Operands

L	n	C

Function

L - label number n

Description

The instruction **L** marks such a location in the program, which serves as a target to the jump and call instructions. Any location in the program can be labelled, if this should be necessary for a better lucidity when viewing, monitoring or debugging of the user program. From the programming point of view, the instruction **L** behaves as a do-nothing operation, no activities are performed.

Note

There cannot be more than one label of a specific parameter in the program. The numerical order of the label instructions within the program is not important.

7. OPERATING INSTRUCTIONS

P	**Process start**
E	**Process end**
ED	**Process end conditional to non-zero stack top value**
EC	**Process end conditional to zero stack top value**

Operands

P	n	C
E	n	C
ED	w/o operand	C
EC	w/o operand	C

n - process number (0 to 64)

Function

P - Pn process start
E - Pn process end
ED - end of active process conditional to non-zero value of the A0 stack top
EC - end of active process conditional to zero value of the A0 stack top

Description

The instruction **P** marks such a location in the program, where the corresponding process starts. It serves for its searching by the system program as the initial process stop latch.

The instruction **E** marks such a location in the program, where the appropriate process Pn ends. It serves for passing the control onto the system program, which decides on the activation of the next process, it also serves as the process stop latch.

The instruction **ED** behaves as the instruction **E** (but it does not serve as a stop latch) only in such a case, that the A0 stack top is not 0 (logical OR of all 32 bits A0 is log.1). If this condition is not fulfilled, the instruction is ignored and the program continues performing the next immediately following instruction.

The instruction **EC** behaves as the instruction **E** (but it does not serve as a stop latch) only in such a case, that the A0 stack top is 0 (logical OR of all 32 bits A0 is log.0). If this condition is not fulfilled, the instruction is ignored and the program continues performing the next immediately following instruction.

Note

The parameter n assumes values only within the range of numbers of permissible processes. The process beginning with the instruction **P** n must be ended by the instruction **E** n with the same parameter. This condition is formal and it is not an error, if there is a jump in the program in another process without return, even if this procedure is not "tidy" from the programmer's point of view.

Since the individual components of the MOSAIC development environment can generate instructions controlling some defined functions (e.g. communication with operator panels, PID controller, etc.) into a user program before its compilation, we recommend not to use the instructions **ED**, **EC**, since the function of these hidden functions could be eliminated. It is better to replace them by a jump to a label inserted at the end of the process.

Example

Permissible sequencing of processes

```
P 0
       :
       :
     JMD   jump
       :
       :
E 0
;
P 10
       :
       :
jump:
       :
       :
E 10       ;If JMD condition is fulfilled, this process end is valid
           ;also for process P0.
```

NOP	No-operation

Operands

NOP	n	C

Function

NOP - no-operation

Description

The instruction **NOP** does not perform any operation. From the user's point of view it has no meaning. It is usually generated by a compiler of a higher language to differentiate the start and end of the program modules or for saving the parameters of these modules.

BP Breakpoint

Operands

BP	n	C

n - number of activated process P5n (0 to 7)

Function

BP - breakpoint

Description

The instruction **BP** is used primarily for the debugging phase of the user program. It activates service processes based on the parameter value. The parameter n can assume only a value 0 to 7 and specifies the number of an activated process P50 to P57, in which it is possible to write the treatment of a situation corresponding to the location of the given instruction **BP** in the user program (e.g. delivering the status of the stack to the scratchpad, condition specification and definition of the status being searched, message printout).

The instruction **BP** n saves the active stack and passes the control to the process P5n. After this process is ended by the instruction **E**, **ED** or **EC** the active stack is refreshed and the program continues performing of the instruction following the instruction **BP** n. Thus, it is a special call instruction.

The instruction **BP** cannot be used within the processes P50 to P57.

Note

Unlike all other processes, the entire active stack is held after entering the processes P50 to P57. When these processes are ended, the state of the stack and the content of S0 and S1 flag registers is refreshed to the values, which were here before entering the process. If we use in the process P5n some of the instructions for stack switching (**NXT**, **PRV**, **CHG**, **CHGS**), the state of the stack will be refreshed after the process end, but only the stack will remain active, that was active at the P5n process end! By doing this, physical change of the stack takes place without changing its content. This can be used to create a copy of the stack. However, it is necessary to pay closer attention to these instructions.

SEQ Conditional process interrupt

Operands

SEQ	Ln	C

Function

SEQ - process interrupt conditional to zero value of the stack top, the process starts in the next cycle from label **L** n

Description

The instruction **SEQ** behaves as the instruction **E** (but it does not serve as a stop latch) in such a case, that the A0 stack top is 0 (logical OR of all 32 bits A0 is log.0). Additionally, it cases, that the process starts from label Ln next time. If the condition is not fulfilled, the instruction is ignored and the program continues performing the next immediately following instruction.

The instruction **SEQ** allows sequential programming within one process, when just a certain part of the process is performed and the transitions among these parts by means of conditions are carried out by the instruction **SEQ**.

Attention

The instruction **SEQ** is permissible only in processes P0 to P40.

Example

We require, that the activity 1 is performed, after setting the signal connected to X1.0 to log.1 to perform 2, after setting the signal connected to X1.1 to log.0 to perform activity 3 and after setting the signal connected to X1.2 to log.1 to perform activity 1 again and still in the circle.

```
P 10
      :              ;activity 1
label1
     LD   input1     ;condition 1
     SEQ  label1     ;as long as input1 = 0, process P10 ends here
                     ;and starts next time on label1
      :              ;input1 = 1 - activity 2
label2
     LDC  input2     ;condition 2
     SEQ  label2     ;as long as input2 = 1, process P10 ends here
                     ;and starts next time on label2
      :              ;input2 = 0 - activity 3
label3
     LD   input3     ;condition 3
     SEQ  label3     ;as long as input3 = 0, process P10 ends here
                     ;and starts next time on label3
E 10                 ;input3 = 1 - process end P10, process starts
                     ;again from the start
```

8. TABLE INSTRUCTIONS

LTB Load item

Instruction	Input parameters								Result							
	stack								stack							
	A7	A6	A5	A4	A3	A2	A1	A0	A7	A6	A5	A4	A3	A2	A1	A0
LTB XYSDR							LIMIT	INDEX	A6	A5	A4	A3	A2	LIMIT	INDEX	VAL
LTB T								INDEX	A6	A5	A4	A3	A2	LIMIT	INDEX	VAL

LIMIT - table limit value (index of last item of table) (udint type)
INDEX - index of required value (udint type)
VAL - content value (type corresponding to the type of operand)

Operands

		bool	byte usint sint	word uint int	dword udint dint	real
LTB	X Y S D R	C	C	C	C	C
LTB	T	C	C	C	C	C

Function

LTB - load item from table

Description

The instruction **LTB** is an indexed analogy to the instruction **LD**. First, it moves the stack ahead. If the specified index is within the table range (it is not greater than its limit), the content of the required value is passed on the A0 stack top and the S1.0 flag is set. If the required item is out of table range (the index is higher than its limit), the S1.0 flag is set to zero.

The instruction of **bool** type takes the item value and sets all 32 bits of the A0 stack top accordingly.

The instruction of **byte**, **usint** and **sint** types takes the item value and saves it without any change to the lowest byte of the A0 stack top, the other bytes are set to zero.

The instruction of **word**, **uint** and **int** types takes the item value and saves it without any change to the lower word of the A0 stack top. The upper word of the stack top is set to zero. The byte with the lowest address in the table within the item is saved to the lowest byte.

The instruction of **dword**, **udint**, **dint** and **real** types takes the item value and saves it without any change on the A0 stack top. The byte with the lowest address in the table within the item is saved to the lowest byte.

Note

If the bit field on the scratchpad is the operand, **this field must begin on bit 0** (by means of directive *#reg aligned*)!

Flags

	.7	.6	.5	.4	.3	.2	.1	.0
S1	-	-	-	-	-	-	-	IS

S1.0 (IS) - 0 - request for an item outside the table
1 - request for an item within the table

Examples

Load an item of bool format

```
#table bool  Tab = 0,1,0,1
#reg uint  INDEX
#reg bool  VAL
;
P 0
     LD   INDEX
     LTB  Tab        ;T table
     WR   VAL
E 0

#def LIMIT  3
#reg aligned bool Tab[LIMIT+1]
#reg uint  INDEX
#reg bool  VAL
;
P 0
     LD   LIMIT
     LD   INDEX
     LTB  Tab        ;table on the scratch pad
     WR   VAL
E 0
```

Load an item of usint type

```
#table usint  Tab = 0,1,2,3
#reg uint   INDEX
#reg usint  VAL
;
P 0
     LD   INDEX
     LTB  Tab        ;T table
     WR   VAL
E 0

#def LIMIT  3
#reg usint  Tab[LIMIT+1]
#reg uint   INDEX
#reg usint  VAL
;
P 0
     LD   LIMIT
     LD   INDEX
     LTB  Tab        ;table on the scratchpad
     WR   VAL
E 0
```

Load an item of uint type

```
#table uint  Tab = 0,1,2,3
#reg uint  INDEX
#reg uint  VAL
;
P 0
     LD    INDEX
     LTB   Tab        ;T table
     WR    VAL
E 0

#def LIMIT  3
#reg uint  Tab[LIMIT+1]
#reg uint  INDEX
#reg uint  VAL
;
P 0
     LD    LIMIT
     LD    INDEX
     LTB   Tab        ;table on the scratchpad
     WR    VAL
E 0
```

Load an item of udint type

```
#table udint  Tab = 0,1,2,3
#reg uint   INDEX
#reg udint  VAL
;
P 0
     LD    INDEX
     LTB   Tab        ;T table
     WR    VAL
E 0

#def LIMIT  3
#reg udint  Tab[LIMIT+1]
#reg uint   INDEX
#reg udint  VAL
;
P 0
     LD    LIMIT
     LD    INDEX
     LTB   Tab        ;table on the scratchpad
     WR    VAL
E 0
```

WTB Write item

Instruction	Input parameters								Result							
	stack								stack							
	A7	A6	A5	A4	A3	A2	A1	A0	A7	A6	A5	A4	A3	A2	A1	A0
WTB XYSR						LIMIT	INDEX	VAL						LIMIT	INDEX	VAL
WTB T							INDEX	VAL						LIMIT	INDEX	VAL

LIMIT - table limit value (index of last item of table) (udint type)
INDEX - index of required value (udint type)
VAL - written content (type corresponding to the type of operand)

Operands

		bool	byte usint sint	word uint int	dword udint dint	real
WTB	X Y S R	C	C	C	C	C
WTB	T	C	C	C	C	C

Function

WTB - write item to the table

Description

The instruction **WTB** is an indexed analogy of the instruction **WR**. The content of the stack remains unchanged. If the specified index is within the table range (it is not greater than its limit), the content of the A0 stack top is passed to the specified item and the S1.0 flag is set. If write to the item out of the table range is required (the index is higher than its limit), the S1.0 flag is set to zero.

The instruction of **bool** type writes to the item the value of logical OR of all 32 bits of the A0 stack top.

The instruction of **byte**, **usint** and **sint** types writes the lowest byte of the A0 stack top to the item.

The instruction of **word**, **uint** and **int** types writes the lower word of the A0 stack top to the item. The lowest byte of the stack top is saved to the byte with the lowest address in the table within the item.

The instruction of **dword**, **udint**, **dint** and **real** types writes the A0 stack top to the item. The lowest byte of the stack top is saved to the byte with the lowest address in the table within the item.

Note

If the bit field on the scratchpad is the operand, this field must begin on bit 0 (by means of directive *#reg aligned*)!

Flags

	.7	.6	.5	.4	.3	.2	.1	.0
S1	-	-	-	-	-	-	-	IS

S1.0 (IS) - 0 - request for an item outside the table
1 - request for an item in the table

Examples

Write an item of bool type

```
#table bool  Tab = 0,1,0,1
#reg uint  INDEX
#reg bool  VAL
;
P 0
     LD   INDEX
     LD   VAL
     WTB  Tab        ;T table
E 0

#def LIMIT  3
#reg aligned bool  Tab[LIMIT+1]
#reg uint  INDEX
#reg bool  VAL
;
P 0
     LD   LIMIT
     LD   INDEX
     LD   VAL
     WTB  Tab        ;table on the scratchpad
E 0
```

Write an item of usint type

```
#table usint  Tab = 0,1,2,3
#reg uint   INDEX
#reg usint  VAL
;
P 0
     LD   INDEX
     LD   VAL
     WTB  Tab        ;T table
E 0

#def LIMIT  3
#reg usint  Tab[LIMIT+1]
#reg uint   INDEX
#reg usint  VAL
;
P 0
     LD   LIMIT
     LD   INDEX
     LD   VAL
     WTB  Tab        ;table on the scratchpad
E 0
```

Write an item of uint type

```
#table uint  Tab = 0,1,2,3
#reg uint  INDEX
#reg uint  VAL
;
P 0
     LD    INDEX
     LD    VAL
     WTB   Tab        ;T table
E 0

#def LIMIT  3
#reg uint  Tab[LIMIT+1]
#reg uint  INDEX
#reg uint  VAL
;
P 0
     LD    LIMIT
     LD    INDEX
     LD    VAL
     WTB   Tab        ;table on the scratchpad
E 0
```

Write an item of udint type

```
#table udint Tab = 0,1,2,3
#reg uint   INDEX
#reg udint  VAL
;
P 0
     LD    INDEX
     LD    VAL
     WTB   Tab        ;T table
E 0

#def LIMIT  3
#reg udint  Tab[LIMIT+1]
#reg uint   INDEX
#reg udint  VAL
;
P 0
     LD    LIMIT
     LD    INDEX
     LD    VAL
     WTB   Tab        ;table on the scratchpad
E 0
```

FTB	**Find item**
FTBN	**Find next item**

Instruction	Input parameters								Result							
	stack								stack							
	A7	A6	A5	A4	A3	A2	A1	A0	A7	A6	A5	A4	A3	A2	A1	A0
FTB XYSDR							LIMIT	VAL							LIMIT	INDEX
FTB T								VAL							LIMIT	INDEX
FTBN XYSDR						INDX0	LIMIT	VAL						INDX0	LIMIT	INDEX
FTBN T							INDX0	VAL							LIMIT	INDEX

LIMIT - table limit value (index of last item of table) (udint type)
VAL - content to be found in the table (type corresponding to the type of operand)
INDX0 - item index, from which searching starts (udint type)
INDEX - index of item found (if the corresponding item is not found, the index has value LIMIT+1) (udint type)

Operands

		bool	byte usint sint	word uint int	dword udint dint	real
FTB	X Y S D R	C	C	C	C	C
FTB	T	C	C	C	C	C
FTBN	X Y S D R	C	C	C	C	C
FTBN	T	C	C	C	C	C

Function

FTB - find item in the table
FTBN - find next item in the table

Description

The instruction **FTB** progressively compares the data at the stack top with the content of table items, until it finds an identical item or until the entire table is read. If the item being searched is found, it writes its index on the A0 stack top and sets the S1.0 flag. If the item is not found, the S1.0 flag is set to zero and the limit at the A0 stack top is increased by 1. If the table contains more identical items, the function will find only the first one (with the lowest index).

The instruction **FTBN** behaves in the same manner, but additionally it contains the parameter INDX0, which contains the item index, from which searching starts. If there are more items of the same value in one table, the instruction **FTB** finds only the item with the lowest index, while by means of the instruction **FTBN** we will find progressively all these items in such a way, that after finding the first item with required content we will call the instruction **FTBN** again and to the parameter INDX0 we will write the value higher by 1, than the index of the found item was. The process is repeated, until the entire table is read.

The instruction of **bool** type compares the value of logical OR of all 32 bits of the A0 stack top with the table items. The bit instruction **FTB** can be used for example for testing of the bit field, where just one bit has a different value (keyboard).

The instruction of **byte**, **usint** and **sint** types compares the content of the lowest byte of the A0 stack top with the table items.

The instruction of **word**, **uint** and **int** types compares the content of the lower word of the A0 stack top with the table items. The lowest byte of the stack top is compared with the byte with the lowest address in the table within the item.

The instruction of **dword**, **udint** and **dint** types compares the content of the A0 stack top with the table items. The lowest byte of the stack top is compared with the byte with the lowest address in the table within the item.

Note

If the bit field on the scratchpad is the operand, this field must begin on bit 0 (by means of directive *#reg aligned*)!

Flags

	.7	.6	.5	.4	.3	.2	.1	.0
S1	-	-	-	-	-	-	-	IS

S1.0 (IS) - 0 - request for an item outside the table
1 - request for an item in the table

Examples

Find an item of bool type

```
#table bool  Tab = 1,1,0,1
#reg uint  INDEX
#reg bool  VAL
;
P 0
     LD   VAL
     FTB  Tab        ;T table
     WR   INDEX
E 0

#def LIMIT  3
#reg aligned bool  Tab[LIMIT+1]
#reg uint  INDEX
#reg bool  VAL
;
P 0
     LD   LIMIT
     LD   VAL
     FTB  Tab        ;table on the scratchpad
     WR   INDEX
E 0
```

Searching for all identical items of usint type

```
#table usint  Tab = 0,1,2,1,2,2,0
#reg uint count       ;number of findings of identical item
#reg usint VAL
;
P 0
     LD   0          ;initial index INDX0
     WR   count      ;count = 0
begin:
     LD   VAL
     FTBN Tab        ;T table
     JNS  end        ;test of finding the item
     INR  count      ;next item found
     INR             ;INDX0 = INDEX+1
     JMP  begin      ;search for next item
;
end:
E 0
```

FTM	**Find part of item**
FTMN	**Find part of next item**

Instruction	Input parameters								Result							
	stack								stack							
	A7	A6	A5	A4	A3	A2	A1	A0	A7	A6	A5	A4	A3	A2	A1	A0
FTM XYSDR							LIMIT	VAL							LIMIT	INDEX
FTM T								VAL							LIMIT	INDEX
FTMN XYSDR						INDX0	LIMIT	VAL						INDX0	LIMIT	INDEX
FTMN T							INDX0	VAL							LIMIT	INDEX

LIMIT - table limit value (index of last item of table) (udint type)
VAL - content to be found in the table (type corresponding to the type of operand)
INDX0 - item index, from which searching starts (udint type)
INDEX - index of item found (if the corresponding item is not found, the index has value LIMIT+1) (udint type)

Operands

		byte usint sint	word uint int	dword udint dint	real
FTM	X Y S D R	C	C	C	C
FTM	T	C	C	C	C
FTMN	X Y S D R	C	C	C	C
FTMN	T	C	C	C	C

Function

FTM - find part of item in the table
FTMN- find of next item part in the table

Description

The instruction **FTM** is generalizing of the instruction **FTB**, when the evaluated table has a double format. Each item contains two parts with a common index. The first part contains a value, the second part contains a selection mask.

	index n		index n+1		
...	item n	mask n	item n+1	mask n+1	...

The instruction **FTM** progressively compares the data at the stack top with the table items and the comparison results are masked with corresponding masks, that only such bits of comparison result are respected, to which the one in the bits at the selection mask corresponds, until an identical item is found or reads the entire table. If an identical item is found, its index is written on the A0 stack top and the S1.0 flag is set.
If an identical item is not found, the S1.0 flag is set to zero and the limit value at the A0 stack top is increased by 1.

The comparison function can be written by means of logic operators as follows

(VAL XOR item) AND mask = result

If this result is 0, the item is identical and its content is passed on the stack top. If the table contains more identical items, the function finds only the first one (with the lowest index).

The instruction **FTMN** behaves in the same manner, but additionally it contains the parameter INDX0, which contains the item index, from which searching starts. If there are more items of the same value in one table, instruction **FTM** finds only the item with the lowest index, while by means of the instruction **FTMN** we will find progressively all these items in such a way, that after finding the first item with required content we will call the instruction **FTMN** again and to the parameter INDX0 we will write the value higher by 1, than the index of the found item was. The process is repeated, until the entire table is read.

The instruction of **byte**, **usint** and **sint** types compares the content of the lowest byte of the A0 stack top with the table items.

The instruction of **word**, **uint** and **int** types compares the content of the lower word of the A0 stack top with the table items. The lowest byte of the stack top is compared with the byte with the lowest address in the table within the item.

The instruction of **dword**, **udint** and **dint** types compares the content of the A0 stack top with the table items. The lowest byte of the stack top is compared with the byte with the lowest address in the table within the item.

Flags

	.7	.6	.5	.4	.3	.2	.1	.0
S1	-	-	-	-	-	-	-	IS

S1.0 (IS) - 0 - request for an item outside the table
1 - request for an item in the table

Examples

Searching for an item of usint type

```
#table usint  Tab = 1,7,0,1
#reg udint  INDEX
#reg usint  VAL
;
P 0
     LD   VAL
     FTM  Tab        ;T table
     WR   INDEX
E 0

#def LIMIT  3
#reg aligned usint  Tab[LIMIT+1]
#reg udint  INDEX
#reg usint  VAL
;
P 0
     LD   LIMIT
     LD   VAL
     FTM  Tab        ;table on the scratchpad
     WR   INDEX
E 0
```

Searching for all identical items of uint type

```
#table uint Tab = 0,1,2,1,2,2,0,7
#reg uint count       ;number of findings of identical item
#reg uint VAL
;
P 0
     LD   0           ;initial index INDX0
     WR   count       ;count = 0
begin:
     LD   VAL
     FTMN Tab         ;T table
     JNS  end         ;test of finding the item
     INR  count       ;next item found
     INR              ;INDX0 = INDEX+1
     JMP  begin       ;search for next item
;
end:
E 0
```

FTS, FTSF Find with sorting
FTSS Find with sorting with sign

Instruction	Input parameters								Result							
	stack								stack							
	A7	A6	A5	A4	A3	A2	A1	A0	A7	A6	A5	A4	A3	A2	A1	A0
FTS XYSDR							LIMIT	VAL							LIMIT	INDEX
FTS T								VAL							LIMIT	INDEX
FTSF XYSDR							LIMIT	VAL							LIMIT	INDEX
FTSF T								VAL							LIMIT	INDEX
FTSS XYSDR							LIMIT	VAL							LIMIT	INDEX
FTSS T								VAL							LIMIT	INDEX

LIMIT - table limit value (index of last item of table) (udint type)
VAL - content to be found in the table (type corresponding to the type of operand)
INDEX - index of item found (if the corresponding item is not found, the index has value LIMIT+1) (udint type)

Operands

FTS	X Y S D R	C		C		C		
FTS	T	C		C		C		
FTSF	X Y S D R							C
FTSF	T							C
FTSS	X Y S D R		C		C		C	
FTSS	T		C		C		C	

Function

FTS - find with sorting according to the table
FTSF - find with sorting according to the table (floating point
FTSS - find with sorting with sign according to the table

Description

The instruction **FTS** is generalizing of the instruction **FTB** and performs multi-level comparison or sorting. To do this, it is necessary that the items in the table are ordered ascendingly according to the values since they represent limits separating particular classes to which the instruction puts the A0 content.

The instruction **FTS** does not shift the stack. It progressively compares the data at A0 with the content of table items, until it finds an item greater or equal to the compared value or until the entire table is read. If an identical item is found, it writes its index on the A0 stack top and the S1.0 flag is set. If an identical item is not found the flag S1.0 is set to zero and at the A0 stack top is the limit value increased by 1.

Categorizing into classes is as follows (k corresponds to the value LIMIT):

0 ≤ VAL ≤ item 0	class 0
item 0 < VAL ≤ item 1	class 1
item 1 < VAL ≤ item 2	class 2
item 2 < VAL ≤ item 3	class 3
:	:
item k–1 < VAL ≤ item k	class k
item k < VAL ≤ maximum	class k+1

The instruction **FTSS** behaves in the same manner, but it accepts the state of the highest bit as a sign at all values. The instruction **FTSF** processes all values in the floating point format (float).

The instruction of **usint, sint** types compares the content of the lowest byte of the A0 stack top with the table items.

The instruction of **uint**, **int** types compares the content of the lower word of the A0 stack top with the table items. The lowest byte of the stack top is compared with the byte with the lowest address in the table within the item.

The instruction of **udint**, **dint** and **real** types compares the content of the A0 stack top with the table items. The lowest byte of the stack top is compared with the byte with the lowest address in the table within the item.

Flags

	.7	.6	.5	.4	.3	.2	.1	.0
S1	-	-	-	-	-	-	-	IS

S1.0 (IS) - 0 - request for an item outside the table
1 - request for an item in the table

Examples

Find with sorting of usint type

```
#table usint  Tab = 1,4,8,15
#reg udint  INDEX
#reg usint  VAL
;
P 0
     LD   VAL
     FTS  Tab        ;T table
     WR   INDEX
E 0

#def LIMIT  3
#reg aligned usint  Tab[LIMIT+1]
#reg udint  INDEX
#reg usint  VAL
;
P 0
     LD   LIMIT
     LD   VAL
     FTS  Tab        ;table on the scratchpad
     WR   INDEX
E 0
```

9. BLOCK OPERATIONS

SRC	**Source of data to be moved**
MOV	**Move data block**

Instruction	Input parameters								Result							
	stack								stack							
	A7	A6	A5	A4	A3	A2	A1	A0	A7	A6	A5	A4	A3	A2	A1	A0
SRC								INDEX								INDEX
MOV							INDEX	LEN							INDEX	LEN

INDEX - index of first item in specified source / destination zone (udint type)
LEN - number of transferred byte items (udint type)

Operands

		byte usint sint
SRC	X Y S D R	C
SRC	T	C
MOV	X Y S R	C
MOV	T	C

Function

SRC - specification of source zone for move data block
MOV - move data block to the target zone

Description

The instruction **SRC** serves as a preparatory instruction before the instruction **MOV**. It saves the data on the initial address of source zone in the internal system memory. The address of the first zone is given by the instruction operand increased by the index saved at the stack top. By means of the index, the beginning of the zone can be dynamically changed.

The instruction **MOV** moves the content of the source zone specified by the instruction **SRC** to the target zone. The number of transferred byte items is loaded from the A0 stack top. The maximum number of items is limited only by the size of the scratchpad or the table, as the case may be. The address of the first item is given by the instruction operand increased by the index saved at the A1 layer.

Flags

	.7	.6	.5	.4	.3	.2	.1	.0
S1	-	-	-	-	-	-	-	IS

S1.0 (IS) - 0 - address of source zone given by the instruction **SRC** or target zone given by the instruction **MOV** is out of T table range or scratchpad, carry is not performed
1 - the address of the source and target zone are within the T table range or scratchpad, carry is performed

S34 = 20 ($14) source data block was defined out of range
S34 = 21 ($15) target data block was defined out of range

Note

Specification of the source zone remains saved in the memory, until it is overwritten by the new instruction **SRC**. Thus it can be used for more instructions **MOV**.

Example

Move data block

```
#def LEN  30
#reg uint   INDEX_SRC, INDEX_MOV
#reg usint  Source[LEN], Destination[LEN]
;
P 0
     LD    INDEX_SRC
     SRC   Source
      :
      :
     LD    INDEX_MOV
     LD    LEN
     MOV   Destination
E 0
```

MTN	**Move table to scratchpad**
MNT	**Fill table from the scratchpad**

Instruction	Input parameters									Result								
	stack									stack								
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
MTN							TAB	REG								TAB	LEN	
MNT							TAB	REG								TAB	LEN	

TAB - number of moved / filled table (udint type)
REG - index of first register R of specified zone in the scratchpad (udint type)
LEN - number of bytes being transferred (udint type)

Operands

		byte usint sint
MTN	w/o operand	C
MNT	w/o operand	C

Function

MTN - move table to scratchpad
MNT - fill table from the scratchpad

Description

The instruction **MTN** moves to the target zone in the scratchpad the entire content of the T table. The number of transferred byte items is given by the table size and the instruction shows it at the stack top after the move.

The instruction **MNT** fills from the source zone in the scratchpad the entire content of the T table. The number of transferred byte items is given by the table size and the instruction shows it at the stack top after the move.

Flags

	.7	.6	.5	.4	.3	.2	.1	.0
S1	-	-	-	-	-	-	-	IS

S1.0 (IS) - 0 - zone address in the scratchpad is out of range, carry is not performed
1 - the zone address in the scratchpad is within range, carry is performed

S34 = 20 ($14) source data block was defined out of range
S34 = 21 ($15) target data block was defined out of range

Examples

Move from the table

```
#def MaxLength  30
#table usint  Tab = 0,1,2,3
#reg usint  Destination[MaxLength]
;
P 0
     LD    __indx Tab           ;TAB
     LD    __indx Destination   ;REG
     MTN
E 0
```

Move to the table

```
#def MaxLength  30
#table byte Tab = 0,1,2,3
#reg byte Source[MaxLength]
;
P 0
     LD    __indx Tab      ;TAB
     LD    __indx Source   ;REG
     MNT
E 0
```

FIL Fill the block

Instruction	Input parameters									Result								
	stack									stack								
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
FIL							LEN	VAL								LEN	VAL	

LEN - length of filled zone (type udint)
VAL - written constant (type uint)

Operands

		byte usint sint
FIL	X Y S R	C

Function

FIL - fill the block with constant

Description

The number of byte items is loaded from the A1 layer of the stack. The address of the first item is given by the instruction operand, the items are filled alternately by the value from the lowest and the second lowest byte of the A0 stack top.

Flags

	.7	.6	.5	.4	.3	.2	.1	.0
S1	-	-	-	-	-	-	-	IS

S1.0 (IS) - 0 - address of filled zone is out of the scratchpad range
1 - address of filled zone is within the scratchpad range

S34 = 21 ($15) target data block was defined out of range

Example

Zone filling

```
#def LEN 30
#reg usint  Destination[LEN]
#reg uint   VAL
;
P 0
     LD   LEN
     LD   VAL
     FIL  Destination
E 0
```

BCMP Block comparison

Instruction	Input parameters									Result								
	Stack									Stack								
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
BCMP						AD1	AD2	LEN		AD2	LEN	A7	A6	A5	A4	A3	CMP	

AD1 - address of register, where the first block begins (type udint)
AD2 - address of register, where the second block begins (type udint)
LEN - length of compared arrays (type udint)
CMP - result of comparison (type bool)

Operands

		byte usint sint
BCMP	X Y S R	C

Function

BCMP - comparison of two data blocks

Description

The **BCMP** instruction performs comparison of two data blocks in the stack. The instruction expects on the layer A2 and A1 addresses of the start of compared blocks and the length of the compared blocks at the stack top. If the contents of the blocks are identical, then log. 1 (TRUE) is written by the instruction at the stack top (logical ones). If the contents of the blocks are not the same, log. 0 (FALSE) is written on the stack top.

Flags

	.7	.6	.5	.4	.3	.2	.1	.0
S1	-	-	-	-	-	-	-	IS

S1.0 (IS) - 0 - defined blocks out of stack range - invalid comparison
1 - defined blocks within stack range - valid comparison
S34 = 21 ($15) data block was defined out of range

Example

Comparison of two data blocks

```
#def Length 30
#reg usint  Zone1[Length], Zone2[Length]
#reg uint   VAL
#reg bool   Result
;
P 0
     LEA   Zone1
     LEA   Zone2
     LD    Length
     BCMP
     WR    Result
E 0
```

10. OPERATION WITH STRUCTURED TABLES

LDSR	**Load item from structured table in the scratchpad**
LDS	**Load item from structured table**

Instr.	**Input parameters**									**Result**								
	stack									stack								
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
LDSR					INDEX	SIZE	REGT	REG						INDEX	SIZE	REGT	REG	
LDS					INDEX	SIZE	TAB	REG						INDEX	SIZE	TAB	REG	

INDEX - item number of structured table (udint type)
SIZE - item size of structured table in bytes (usint type)
TAB - number of table read (udint type)
REGT - index of first register R of read table (udint type)
REG - index of first register R of designated target zone (udint type)

Operands

		byte usint sint
LDSR	w/o operand	C
LDS	w/o operand	C

Function

LDSR - load item from structured table in the scratchpad
LDS - load item from structured T table

Description

The designated table is structured into individual items of the size given by the parameter SIZE. The instruction **LDSR** moves to the scratchpad target zone one item of the table in the scratchpad beginning on the register REGT given by the parameter INDEX. The instruction **LDS** moves to the scratchpad target zone one item of the table TAB given by the parameter INDEX.

Flags

	.7	.6	.5	.4	.3	.2	.1	.0
S1	-	-	-	-	-	-	-	IS

S1.0 (IS) - 0 - required item is out of T table range, or target zone address is out of the scratchpad range, carry is not performed
1 - parameters are OK, carry is performed

Example

Load an item from structured table

```
#def MaxLength  30
#table usint  Tab = 0,1,2,3,4,5,6,7,8,9
#reg usint  Destination[MaxLength]
#reg usint  SIZE
#reg uint   INDEX
;
P 0
     LD   INDEX
     LD   SIZE
     LD   __indx Tab          ;TAB
     LD   __indx Destination  ;REG
     LDS
     JNS  jump
      :              ;operation is OK
jump:
      :              ;invalid operation
E 0
```

WRSR	**Write item to structured table in the scratchpad**
WRS	**Write item to structured table**

Instr.	Input parameters									Result								
	stack									stack								
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
WRSR					INDEX	SIZE	REGT	REG						INDEX	SIZE	REGT	REG	
WRS					INDEX	SIZE	TAB	REG						INDEX	SIZE	TAB	REG	

INDEX - item number of structured table (udint type)
SIZE - item size of structured table in bytes (usint type)
TAB - number of target table (udint type)
REGT - index of first register R of target table (udint type)
REG - index of first register R of designated source zone (udint type)

Operands

		byte usint sint
WRSR	w/o operand	C
WRS	w/o operand	C

Function

WRSR - write an item to structured table in the scratchpad
WRS - write an item to structured table T

Description

The designated table is structured into individual items of the size given by the parameter SIZE. The instruction **WRSR** fills from the source zone of the scratchpad one item of the table in the scratchpad beginning on the register REGT given by the parameter INDEX. The instruction **WRS** fills from the source zone of the scratchpad one item of the table TAB given by the parameter INDEX.

Flags

	.7	.6	.5	.4	.3	.2	.1	.0
S1	-	-	-	-	-	-	-	IS

S1.0 (IS) - 0 - required item is out of T table range, or the target zone address is out of the scratchpad range, carry is not performed
1 - parameters are OK, carry is performed

Example

Write an item to structured table

```
#def MaxLength  30
#table usint  Tab = 0,1,2,3,4,5,6,7,8,9
#reg usint  Source[MaxLength]
#reg usint  SIZE
#reg uint   INDEX
;
P 0
     LD   INDEX
     LD   SIZE
     LD   __indx Tab           ;TAB
     LD   __indx Source        ;REG
     WRS
     JNS  jump
      :              ;operation is OK
jump:
      :              ;invalid operation
E 0
```

FIS	**Fill item of structured table in the scratchpad**
FIT	**Fill item of structured table**

Instr.	Input parameters									Result								
	stack									stack								
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
FIS					INDEX	SIZE	REGT	VAL						INDEX	SIZE	REGT	VAL	
FIT					INDEX	SIZE	TAB	VAL						INDEX	SIZE	TAB	VAL	

INDEX - item number of structured table (udint type)
SIZE - item size of structured table in bytes (usint type)
TAB - number of table read (udint type)
REGT - index of first register R of read table (udint type)
VAL - filled value (usint type)

Operands

		byte usint sint
FIS	w/o operand	C
FIT	w/o operand	C

Function

FIS - fill an item of structured table in the scratchpad
FIT - fill an item of structured T table

Description

The designated part of scratchpad is structured into individual items of the size given by the parameter SIZE. The instruction **FIS** fills one item of the scratchpad by the specified value VAL given by the parameter INDEX.

The designated table is structured into individual items of the size given by the parameter SIZE. The instruction **FIT** fills one item of the T table by the specified value VAL given by the parameter INDEX.

Flags

	.7	.6	.5	.4	.3	.2	.1	.0
S1	-	-	-	-	-	-	-	IS

S1.0 (IS) - 0 - required item is out of the scratchpad range, or is out of T table range, carry is not performed
1 - parameters are OK, carry is performed

Examples

Fill an item of structured table

```
#def MaxLength 30
#reg usint  Array[MaxLength]
#reg usint  SIZE
#reg usint  VAL
#reg uint   INDEX
;
```

```
P 0
     LD    INDEX
     LD    SIZE
     LD    __indx Array    ;REG
     LD    VAL
     FIS
     JNS   jump
      :              ;operation is OK
jump:
      :              ;invalid operation
E 0

#table usint  Tab = 0,1,2,3,4,5,6,7,8,9
#reg usint  SIZE
#reg usint  VAL
#reg uint   INDEX
;
P 0
     LD    INDEX
     LD    SIZE
     LD    __indx Tab ;TAB
     LD    VAL
     FIT
     JNS   skok
      :              ;operation is OK
skok:
      :              ;invalid operation
E 0
```

FNS	**Find item of structured table in the scratchpad**
FNT	**Find item of structured table**

Instr.	Input parameters								Result							
	stack								stack							
	A7	A6	A5	A4	A3	A2	A1	A0	A7	A6	A5	A4	A3	A2	A1	A0
FNS				NUM	BYTE	SIZE	REGT	VAL				NUM	BYTE	SIZE	REGT	INDEX
FNT					BYTE	SIZE	TAB	VAL					BYTE	SIZE	TAB	INDEX

NUM - number of scanned items (type udint)
BYTE - index of scanned byte in the item (type usint)
SIZE - item size of structured table in bytes (type usint)
REGT - index of first register R of the table in the scratchpad (type udint)
TAB - number of scanned table (type udint)
VAL - value being searched (type udint)
INDEX - index of item found (type udint)

Operands

		byte usint sint
FNS	w/o operand	C
FNT	w/o operand	C

Function

FNS - find an item of structured table in the scratchpad
FNT - find an item of structured T table

Description

The designated part of scratchpad is structured into individual items of the size given by the parameter SIZE. The instruction **FNS** compares the specified value VAL with one byte of the item given by the parameter BYTE. The instruction scans the number of items specified by the parameter NUM. The index of the found item is shown at the stack top and the S1.0 flag is set. If more items have the same value of the byte being scanned, the item with the lowest index is always shown. If the item is not found, the value of the shown index is higher by 1 than the index of the last scanned item. Since indexing is performed from 0, the value equals to the value of the parameter NUM.

The designated table is structured into individual items of the size given by the parameter SIZE. The instruction **FNT** compares the specified value VAL with one byte of the item given by the parameter BYTE. The instruction scans all items of the table. The index of the found item is shown at the stack top and the S1.0 flag is set. If more items have the same value of the byte being scanned, the item with the lowest index is always shown. If the item is not found, the value of the shown index is higher by 1 than the index of the last scanned item. Since indexing is performed from 0, the value equals to the number of the items in the table.

Flags

	.7	.6	.5	.4	.3	.2	.1	.0
S1	-	-	-	-	-	-	-	IS

S1.0 (IS) - 0 - required item was not found
1 - required item was found

Examples

Find an item of structured table

```
#def NUM  30
#reg usint Array[NUM],SIZE,BYTE
#reg usint VAL
#reg uint INDEX
;
P 0
     LD   NUM
     LD   BYTE
     LD   SIZE
     LD   __indx Array    ;REG
     LD   VAL
     FNS
     JNS  jump
     WR   INDEX      ;item was found
jump:
      :              ;item was not found
E 0

#table byte Tab = 0,1,2,3,4,5,6,7,8,9
#reg usint SIZE,BYTE
#reg usint VAL
#reg uint INDEX
;
P 0
     LD   BYTE
     LD   SIZE
     LD   __indx Tab ;TAB
     LD   VAL
     FNT
     JNS  jump
     WR   INDEX      ;item was found
jump:
      :              ;item was not found
E 0
```

11. FLOATING POINT ARITHMETIC INSTRUCTIONS

ADF, ADDF Addition
SUF, SUDF Subtraction

Instruction	Input parameters									Result								
	stack								ope-rand	stack								ope-rand
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
ADF								a	b								$a+b$	b
ADF w/o op.							a	b		b	A7	A6	A5	A4	A3	A2	$a+b$	
ADDF w/o op.					a		b			b		A7	A6	A5	A4	$a+b$		
SUF								a	b								$a-b$	b
SUF w/o op.							a	b		b	A7	A6	A5	A4	A3	A2	$a-b$	
SUDF w/o op.					a		b			b		A7	A6	A5	A4	$a-b$		

Operands

		real	lreal
ADF	X Y S D R	C	
ADF	#	C	
ADF	w/o operand	C	
ADDF	w/o operand		C
SUF	X Y S D R	C	
SUF	#	C	
SUF	w/o operand	C	
SUDF	w/o operand		C

Function

ADF - addition (real)
ADDF - addition (lreal)
SUF - subtraction (real)
SUDF - subtraction (lreal)

Description

The instruction **ADF** with operand adds the content of the given operand to the A0 stack top. The instruction **SUF** with operand subtracts the content of the given operand from the A0 stack top. The result is written on the stack top. The content of the other layers remains unchanged. The instruction does not set any flags.

The instruction **ADF** w/o operand adds the content of the layers A1 and A0. The instruction **SUF** w/o operand subtracts the content of the A0 layer from the content of the A1 layer. After the operation is performed, the stack is moved one level back and the result is written on the A0 stack top. The instruction does not set any flags.

The instruction **ADDF** w/o operand adds the content of double-layers A23 and A01. The instruction **SUDF** w/o operand subtracts the content of the double-layer A01 from the content of the double-layer A23. After the operation is performed, the stack is moved two levels back and the result is written on the A0 stack top. The instruction does not set any flags.

Examples

Realization of the expression $d = a + (b - c)$

```
#reg real  va, vb, vc, vd
;
P 0
     LD   vb
     SUF  vc         ;(b - c)
     ADF  va         ;a + ( )
     WR   vd
E 0

#reg lreal  va, vb, vc, vd
;
P 0
     LD   vb
     LD   vc
     SUDF            ;(b - c)
     LD   va
     ADDF            ;a + ( )
     WR   vd
E 0
```

MUF, MUDF Multiplication
DIF, DIDF Division

Instruction	Input parameters									Result								
	stack								ope-rand	stack								ope-rand
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
MUF								a	b								$a \cdot b$	b
MUF w/o op.							a	b		b	A7	A6	A5	A4	A3	A2	$a \cdot b$	
MUDF w/o op.					a		b			b		A7	A6	A5	A4	$a \cdot b$		
DIF								a	b								a / b	b
DIF w/o op.							a	b		b	A7	A6	A5	A4	A3	A2	a / b	
DIDF w/o op.					a		b			b		A7	A6	A5	A4	a / b		

Operands

		real	lreal
MUF	X Y S D R	C	
MUF	#	C	
MUF	w/o operand	C	
MUDF	w/o operand		C
DIF	X Y S D R	C	
DIF	#	C	
DIF	w/o operand	C	
DIDF	w/o operand		C

Function

MUF - subtraction (real)
MUDF- subtraction (lreal)
DIF - division (real)
DIDF - division (lreal)

Description

The instruction **MUF** with operand multiplies the content of the A0 stack top by the content of the given operand. The result is written on the stack top. The content of the other layers remains unchanged. The instruction does not set any flags.

The instruction **MUF** w/o operand multiplies the content of the A1 and A0 layers. It then moves the stack one level back and writes the result on the A0 stack top. The instruction does not set any flags.

The instruction **MUDF** w/o operand multiplies the content of double-layers A23 and A01. It then moves the stack two levels back and writes the result on the new A0 stack top. The instruction does not set any flags.

The instruction **DIF** with operand divides the content of the A0 stack top by the content of the given operand. The result is written on the stack top. The content of the other layers remains unchanged.

The instruction **DIF** w/o operand divides the content of the A1 layer by the content of the A0 layer. It then moves the stack one level back and writes the result on the A0 stack top.

The instruction **DIDF** w/o operand divides the content of the A23 double-layer by the content of the double-layer A01. It then moves the stack two levels back and writes the result on the new A0 stack top.

If division by zero is performed, the S0.0 bit is set to log.1 and error 16 is written to the S34 register. The stack top contains all ones (invalid number according to the convention of float and double formats).

Flags

	.7	.6	.5	.4	.3	.2	.1	.0
S0	-	-	-	-	-	-	-	ZR

S0.0 (ZR) - division by zero
1 - division by zero performed, the result is not valid

S34 = 16 ($10) error of division by zero

Examples

Realization of the expression $d = a + (b \cdot c)$

```
#reg real va, vb, vc, vd
;
P 0
     LD    vb
     MUF   vc         ;(b . c)
     ADF   va         ;a + ( )
     WR    vd
E 0
```

Realization of the expression $d = a + \frac{b}{c}$

```
#reg real va, vb, vc, vd
;
P 0
     LD    vb
     DIF   vc         ;(b / c)
     ADF   va         ;a + ( )
     WR    vd
E 0
```

EQF, EQDF	**Comparison (equality)**
LTF, LTDF	**Comparison (less than)**
GTF, GTDF	**Comparison (greater than)**
CMF, CMDF	**Comparison**

Instruction	Input parameters									Result								
	stack								ope-rand	stack								op.
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
EQF								*a*	*b*								*a= b ?*	*b*
EQF w/o op.							*a*	*b*		*b*	A7	A6	A5	A4	A3	A2	*a= b ?*	
EQDF w/o op.					*a*		*b*			-	*b*		A7	A6	A5	A4	*a= b ?*	
LTF								*a*	*b*								*a< b ?*	*b*
LTF w/o op.							*a*	*b*		*b*	A7	A6	A5	A4	A3	A2	*a< b ?*	
LTDF w/o op.					*a*		*b*			-	*b*		A7	A6	A5	A4	*a< b ?*	
GTF								*a*	*b*								*a> b ?*	*b*
GTF w/o op.							*a*	*b*		*b*	A7	A6	A5	A4	A3	A2	*a> b ?*	
GTDF w/o op.					*a*		*b*			-	*b*		A7	A6	A5	A4	*a> b ?*	
CMF								*a*	*b*								*a*	*b*
CMF w/o op.							*a*	*b*								*a*	*b*	
CMDF w/o op.					*a*		*b*							*a*		*b*		

Operands

		real	lreal
EQF	X Y S D R	C	
EQF	#	C	
EQF	w/o operand	C	
EQDF	w/o operand		C
LTF	X Y S D R	C	
LTF	#	C	
LTF	w/o operand	C	
LTDF	w/o operand		C
GTF	X Y S D R	C	
GTF	#	C	
GTF	w/o operand	C	
GTDF	w/o operand		C
CMF	X Y S D R	C	
CMF	#	C	
CMF	w/o operand	C	
CMDF	w/o operand		C

Function

EQF - comparison of values with equality test (real)
EQDF- comparison of values with equality test (lreal)
LTF - comparison of values with test less than ... (real)
LTDF - comparison of values with test less than ... (lreal)
GTF - comparison of values with test greater than ... (real)
GTDF- comparison of values with test greater than ... (lreal)
CMF - comparison of values and setting of result flags (real)
CMDF- comparison of values and setting of result flags (lreal)

Description

The instructions **EQF**, **LTF**, **GTF** with operand are internally equal to each other. They compare the content of the stack top with operand, set the flags at S0 and then they write the truth result on the stack top - log.1 (all ones), if the test condition is fulfilled, or log.0, if the condition is not fulfilled.

The instructions **EQF**, **LTF**, **GTF** w/o operand are internally equal to each other. They compare the content of the A1 layer with the content of the A0 stack top, set the flags at S0, they move the stack one level back and then they write the truth result on the new A0 stack top - log.1 (all ones), if the test condition is fulfilled, or log.0, if the condition is not fulfilled.

The instructions **EQDF**, **LTDF**, **GTDF** w/o operand are internally equal to each other. They compare the content of the double layer A23 with the content of the A01 stack top, set the flags at S0, they move the stack three levels back and then they write the truth result on the new A0 stack top - log.1 (all ones), if the test condition is fulfilled, or log.0, if the condition is not fulfilled.

The instruction **CMF** with operand compares the content of the stack top with operand and sets the flags at S0. The content of the stack remains unchanged. The instruction **CMF** w/o operand compares the content of the A1 layer with the content of the A0 stack top and sets the flags at S0. The content of the stack remains unchanged.

The instruction **CMDF** w/o operand compares the content of double layer A23 with the content of the A01 stack top and sets the flags at S0. The content of the stack remains unchanged.

Flags

	.7	.6	.5	.4	.3	.2	.1	.0
S0	-	-	-	-	-	≤	CO	ZR

S0.0 (ZR) - comparison to match
0 - it is valid that $a \neq b$
1 - it is valid that $a = b$

S0.1 (CO) - carry out
0 - it is valid that $a \geq b$
1 - it is valid that $a < b$

S0.2 (≤) - logic OR S0.0 OR S0.1
0 - it is valid that $a > b$
1 - it is valid that $a \leq b$

MAXF, MAXD Maximum
MINF, MIND Minimum

Instruction	Input parameters									Result								
	stack									stack								
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
MAXF							*a*	*b*		*b*	A7	A6	A5	A4	A3	A2	*MAX(a,b)*	
MAXD					*a*		*b*			*b*		A7	A6	A5	A4	*MAX(a,b)*		
MINF							*a*	*b*		*b*	A7	A6	A5	A4	A3	A2	*MIN(a,b)*	
MIND					*a*		*b*			*b*		A7	A6	A5	A4	*MIN(a,b)*		

Operands

		real	lreal
MAXF	w/o operand	C	
MAXD	w/o operand		C
MINF	w/o operand	C	
MIND	w/o operand		C

Function

MAXF- maximum from two values (real)
MAXD- maximum from two values (lreal)
MINF - minimum from two values (real)
MIND - minimum from two values (lreal)

Description

The instruction **MAXF** compares the content of the A1 layer with the content of the A0 stack top. It then moves the stack one level back and writes the value, which is greater, on the new A0 stack top. The instruction **MAXD** compares the content of double layer A23 with the content of the A01 stack top. It then moves the stack two levels back and writes the value, which is greater, on the new A01 stack top.

The instruction **MINF** compares the content of the A1 layer with the content of the A0 stack top. It they moves the stack one level back and writes the, which is less, on the new A0 stack top. The instruction **MIND** compares the content of double layer A23 with the content of the A01 stack top. It then moves the stack two levels back and writes the value, which is less, on the new A01 stack top.

Example

Value limitation within a range of –2,2 to +3,15

```
#def MINIMUM  -2.2
#def MAXIMUM  3.15
#reg udint  input,output
;
P0
     LD   MAXIMUM
     LD   input
     MIN                ;top constraints
     LD   MINIMUM
     MAX                ;bottom constraints
     WR   output
E0
```

CEI, CEID	**Rounding up**
FLO, FLOD	**Rounding down**
RND, RNDD	**Arithmetic rounding**

Instruction	Input parameters									Result								
	stack									stack								
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
CEI								a									a ↗	
CEID							a									a ↗		
FLO								a									a ↘	
FLOD							a									a ↘		
RND								a									$\cong a$	
RNDD							a									$\cong a$		

Operands

		real	lreal
CEI	w/o operand	C	
CEID	w/o operand		C
FLO	w/o operand	C	
FLOD	w/o operand		C
RND	w/o operand	C	
RNDD	w/o operand		C

Function

CEI - rounding of a number in floating point to the closest higher integral number (real)
CEID - rounding of a number in floating point to the closest higher integral number (lreal)
FLO - rounding of a number in floating point to the closest lower integral number (real)
FLOD - rounding of a number in floating point to the closest lower integral number (lreal)
RND - arithmetical rounding of a number in floating point (real)
RNDD - arithmetical rounding of a number in floating point (lreal)

Description

The instructions **CEI** and **CEID** perform rounding of a number at the stack top to the closest higher integral number and save this number on the stack top. The content of the other layers remains unchanged.

The instructions **FLO** and **FLOD** perform rounding of a number at the stack top to the closest lower integral number and save this number on the stack top. The content of the other layers remains unchanged.

The instructions **RND** and **RNDD** perform arithmetical rounding of a number at the stack top to an integral number, i.e. the numbers with a value of their tenths 0 to 4 will be rounded down and the numbers with a value of their tenths 5 to 9 will be rounded up. The content of the other layers remains unchanged.

ABS, ABSD Absolute value
CSG, CSGD Sign change

<table>
<tr><th>Instruction</th><th colspan="9">Input parameters</th><th colspan="9">Result</th></tr>
<tr><td></td><td colspan="8">stack</td><td></td><td colspan="8">stack</td><td></td></tr>
<tr><td></td><td>A7</td><td>A6</td><td>A5</td><td>A4</td><td>A3</td><td>A2</td><td>A1</td><td>A0</td><td></td><td>A7</td><td>A6</td><td>A5</td><td>A4</td><td>A3</td><td>A2</td><td>A1</td><td>A0</td><td></td></tr>
<tr><td>ABS</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>a</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>$|a|$</td><td></td></tr>
<tr><td>ABSD</td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="2">a</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="2">$|a|$</td><td></td></tr>
<tr><td>CSG</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>a</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>$-a$</td><td></td></tr>
<tr><td>CSGD</td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="2">a</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="2">$-a$</td><td></td></tr>
</table>

Operands

		real	lreal
ABS	w/o operand	C	
ABSD	w/o operand		C
CSG	w/o operand	C	
CSGD	w/o operand		C

Function

ABS - computation of absolute value of a number (real)
ABSD- computation of absolute value of a number (lreal)
CSG - sign change (real)
CSGD- sign change (lreal)

Description

The instructions **ABS** and **ABSD** perform setting of the highest bit of the number to zero at the stack top, which carries the sign. The content of the other layers remains unchanged.

The instructions **CSG** and **CSGD** perform the change of the value of the highest bit of the number at the stack top, which carries the sign. The content of the other layers remains unchanged.

LOG, LOGD	**Decimal logarithm**
LN, LND	**Natural logarithm**
EXP, EXPD	**Exponential function**
POW, POWD	**Common power**
SQR, SQRD	**Square root**
HYP, HYPD	**Hypotenuse**

Instruction	Input parameters									Result								
	stack									stack								
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
LOG								a									$\log_{10} a$	
LOGD							a									$\log_{10} a$		
LN								a									$\ln a$	
LND							a									$\ln a$		
EXP								a									e^a	
EXPD							a									e^a		
POW							a	b		b	A7	A6	A5	A4	A3	A2	a^b	
POWD					a		b			b		A7	A6	A5	A4	a^b		
SQR								a									$\sqrt{a}$	
SQRD							a									$\sqrt{a}$		
HYP							a	b		b	A7	A6	A5	A4	A3	A2	$\sqrt{a^2+b^2}$	
HYPD					a		b			b		A7	A6	A5	A4	$\sqrt{a^2+b^2}$		

Operands

		real	lreal
LOG	w/o operand	C	
LOGD	w/o operand		C
LN	w/o operand	C	
LND	w/o operand		C
EXP	w/o operand	C	
EXPD	w/o operand		C
POW	w/o operand	C	
POWD	w/o operand		C
SQR	w/o operand	C	
SQRD	w/o operand		C
HYP	w/o operand	C	
HYPD	w/o operand		C

Function

LOG - computation of common logarithm (real)
LOGD- computation of common logarithm (lreal)
LN - computation of natural logarithm (real)
LND - computation of natural logarithm (lreal)
EXP - computation of exponential function (real)
EXPD - computation of exponential function (lreal)
POW - computation of square (real)
POWD-computation of square (lreal)
SQR - computation of square root (real)

SQRD- computation of square root (double)
HYP - computation of hypotenuse (float)
HYPD- computation of hypotenuse (double)

Description

The instructions **LOG** and **LOGD** perform computation of common logarithm and the instructions **LN** and **LND** computation of natural logarithm of the stack top content. The content of the stack top must be greater than 0. The result is saved on the stack top. The content of the other layers remains unchanged.

The instructions **EXP** and **EXPD** perform the computation of exponential function. The power exponent of the Euler's number is expected at the stack top. The result is saved on the stack top. The content of the other layers remains unchanged. The instruction does not set any flags.

The instructions **POW** and **POWD** perform computation of square.

The power exponent b is expected by the instruction **POW** at the A0 stack top, the base number a at the A1 layer. The stack is shifted one level back and the result is saved at the stack top.

The power exponent b is expected by the instruction **POWD** at the A01 stack top, the base number a at the double layer A23. The stack is shifted two levels back and the result is saved at the stack top.

The numbers passed to the instructions **POW** and **POWD** must not be zero at the same time. If the number being raised is negative, then the power exponent can have only an integral value.

The instructions **SQR** and **SQRD** perform computation of square root of the stack top content. The number being extracted must not be negative. The result is saved on the stack top. The content of the other layers remains unchanged.

The instructions **HYP** and **HYPD** perform computation of hypotenuse.

The parameters are expected by the instruction **HYP** at layers A0 and A1. The stack is shifted one level back and the result is saved at the stack top.

The parameters are expected by the instruction **HYPD** at double layers A01 and A23. The stack is shifted two levels back and the result is saved at the stack top.

The instruction does not set any flags.

Attention: The input parameters of the instructions **HYP** and **HYPD** must be such, that the expression $a^2 + b^2$ does not exceed the maximum range of the real or lreal format, as the case may be.

Flags

	.7	.6	.5	.4	.3	.2	.1	.0
S1	-	-	-	-	-	-	-	S

S1.0 (S) - 1 - the parameters are OK, the result is valid (they set instruction **LOG**, **LOGD**, **LN**, **LND**, **POW**, **POWD**, **SQR**, **SQRD**)
0 - invalid parameters, the result is not valid

SIN, SIND	**Sine**
COS, COSD	**Cosine**
TAN, TAND	**Tangent**
ASN, ASND	**Arc sine**
ACS, ACSD	**Arc cosine**
ATN, ATND	**Arc tangent**

<table>
<tr><th>Instruction</th><th colspan="9">Input parameters</th><th colspan="9">Result</th></tr>
<tr><td rowspan="2"></td><td colspan="8">stack</td><td rowspan="2"></td><td colspan="8">stack</td><td rowspan="2"></td></tr>
<tr><td>A7</td><td>A6</td><td>A5</td><td>A4</td><td>A3</td><td>A2</td><td>A1</td><td>A0</td><td>A7</td><td>A6</td><td>A5</td><td>A4</td><td>A3</td><td>A2</td><td>A1</td><td>A0</td></tr>
<tr><td>SIN</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>a</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>$\sin a$</td><td></td></tr>
<tr><td>SIND</td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="2">a</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="2">$\sin a$</td><td></td></tr>
<tr><td>COS</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>a</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>$\cos a$</td><td></td></tr>
<tr><td>COSD</td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="2">a</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="2">$\cos a$</td><td></td></tr>
<tr><td>TAN</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>a</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>$\tan a$</td><td></td></tr>
<tr><td>TAND</td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="2">a</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="2">$\tan a$</td><td></td></tr>
<tr><td>ASN</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>a</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>$\arcsin a$</td><td></td></tr>
<tr><td>ASND</td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="2">a</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="2">$\arcsin a$</td><td></td></tr>
<tr><td>ACS</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>a</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>$\arccos a$</td><td></td></tr>
<tr><td>ACSD</td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="2">a</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="2">$\arccos a$</td><td></td></tr>
<tr><td>ATN</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>a</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>$\arctan a$</td><td></td></tr>
<tr><td>ATND</td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="2">a</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="2">$\arctan a$</td><td></td></tr>
</table>

Operands

		real	lreal
SIN	w/o operand	C	
SIND	w/o operand		C
COS	w/o operand	C	
COSD	w/o operand		C
TAN	w/o operand	C	
TAND	w/o operand		C
ASN	w/o operand	C	
ASND	w/o operand		C
ACS	w/o operand	C	
ACSD	w/o operand		C
ATN	w/o operand	C	
ATND	w/o operand		C

Function

SIN - sine (real)
SIND - sine (lreal)
COS - cosine (real)
COSD- cosine (lreal)
TAN - tangent (real)
TAND- tangent (lreal)
ASN - function reverse to sine (real)
ASND- function reverse to sine (lreal)
ACS - function reverse to cosine (real)
ACSD- function reverse to cosine (lreal)

ATN - function reverse to tangent (real)
ATND- function reverse to tangent (lreal)

Description

The instructions **SIN** and **SIND** perform sine of the stack top content. The parameter is expected in radians in the range of <–65 536; +65 536>. The result is saved on the stack top. The content of the other layers remains unchanged.

The instruction **COS** and **COSD** perform cosine of the stack top content. The parameter is expected in radians in the range of <–65 536; +65 536>. The result is saved on the stack top. The content of the other layers remains unchanged.

The instructions **TAN** and **TAND** perform tangent of the stack top content. The parameter is expected in radians in the range of $<-\frac{\pi}{2};\ +\frac{\pi}{2}>$. The result is saved on the stack top. The content of the other layers remains unchanged.

The instructions **ASN** and **ASND** perform arc sine of the stack top content. The parameter is expected in the range of <–1; +1>. The result in the range of $<-\frac{\pi}{2};\ +\frac{\pi}{2}>$ is saved on the stack top. The content of the other layers remains unchanged.

The instructions **ACS** and **ACSD** perform arc cosine of the stack top content. The parameter is expected in the range of <–1; +1>. The result in the range of $<-\frac{\pi}{2};\ +\frac{\pi}{2}>$ is saved on the stack top. The content of the other layers remains unchanged.

The instructions **ATN** and **ATND** perform arc tangent of the stack top content. The result in the range of $<-\frac{\pi}{2};\ +\frac{\pi}{2}>$ is saved on the stack top. The content of the other layers remains unchanged. The flag at the register S1 is not set, the result is always valid.

Flags

	.7	.6	.5	.4	.3	.2	.1	.0
S1	-	-	-	-	-	-	-	S

S1.0 (S) - 1 - parameters are OK, the result is valid (they do not set instruction **ATN**, **ATND**)
0 - invalid parameters, the result is not valid

UWF	**Conversion of uint value to real**
IWF	**Conversion of int value to real**
ULF	**Conversion of udint value to real**
ILF	**Conversion of dint value to real**

Instruction	**Input parameters**									**Result**								
	stack									stack								
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
UWF								NUI									NR	
IWF								NI									NR	
ULF								NUD									NR	
ILF								ND									NR	

NUI - *value of uint type*
NI - *value of int type*
NUD - *value of udint type*
ND - *value of dint*
NR - *value converted to real type*

Operands

		uint	int	udint	dint
UWF	w/o operand	C			
IWF	w/o operand		C		
ULF	w/o operand			C	
ILF	w/o operand				C

Function

UWF - conversion of value of uint type to real type
IWF - conversion of value of int type to real type
ULF - conversion of value of udint type to real type
ILF - conversion of value of dint type to real type

Description

The instruction **UWF** processes the A0 stack top as a number of uint type in the range of <0; 65 535> and converts it to the real type. The result is saved on the A0 stack top. The content of the other layers remains unchanged.

The instruction **IWF** processes the A0 stack top as a number of int type in the range of <–32 768; +32 767> and converts it to the real type. The result is saved on the A0 stack top. The content of the other layers remains unchanged.

The instruction **ULF** processes the A0 stack top as a number of udint type in the range of <0; 4 294 967 295> and converts it to the real type. The result is saved on the A0 stack top. The content of the other layers remains unchanged.

The instruction **ILF** processes the A0 stack top as a number of dint type in the range of <–2 147 483 648; +2 147 483 647> and converts it to the real type. The result is saved on the A0 stack top. The content of the other layers remains unchanged.

ULDF	**Conversion of udint value to lreal**
ILDF	**Conversion of dint value to lreal**
FDF	**Conversion of real value to lreal**

Instruction	**Input parameters**									**Result**								
	stack									stack								
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
ULDF								NUD		A6	A5	A4	A3	A2	A1	NLR		
ILDF								ND		A6	A5	A4	A3	A2	A1	NLR		
FDF								NR		A6	A5	A4	A3	A2	A1	NLR		

NUL - value of udint type
ND - value of dint type
NR - value of real type
NLR - value converted to lreal type

Operands

		udint	dint	real
ULDF	w/o operand	C		
ILDF	w/o operand		C	
FDF	w/o operand			C

Function

ULDF - conversion of value of udint type to lreal type
ILDF - conversion of value of dint type to double type
FDF - conversion of value of float type to double type

Description

The instruction **ULDF** processes the A0 stack top as a number of udint type in the range of <0; 4 294 967 295> and converts it to the lreal type. The stack is shifted one level ahead and the result is saved at the A01 stack top.

The instruction **ILDF** processes the A0 stack top as a number of dint type in the range of <–2 147 483 648; +2 147 483 647> and converts it to the lreal type. The stack is shifted one level ahead and the result is saved at the A01 stack top.

The instruction **FDF** processes the A0 stack top as a number of real type and converts it to the lreal type. The stack is shifted one level ahead and the result is saved at the A01 stack top.

UFW	**Conversion of real value to uint**
IFW	**Conversion of real value to int**
UFL	**Conversion of real value to udint**
IFL	**Conversion of real value to dint**

Instruction	**Input parameters**									**Result**								
	stack									stack								
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
UFW								NR									NUI	
IFW								NR									NI	
UFL								NR									NUD	
IFL								NR									ND	

NF - value of real type
NUW - value converted to uint type
NIW - value converted to int type
NUL - value converted to udint type
NIL - value converted to dint type

Operands

		uint	int	udint	dint
UFW	w/o operand	C			
IFW	w/o operand		C		
UFL	w/o operand			C	
IFL	w/o operand				C

Function

UFW - conversion of value of real type to uint type
IFW - conversion of value of real type to int type
UFL - conversion of value of real type to udint type
IFL - conversion of value of real type to dint type

Description

The instruction **UFW** processes the A0 stack top as a number of real type and converts it to the uint type in the range of <0; 65 535>. The result is saved on the A0 stack top. The content of the other layers remains unchanged.

The instruction **IFW** processes the A0 stack top as a number of real type and converts it to the int type in the range of <–32 768; +32 767>. The result is saved on the A0 stack top. The content of the other layers remains unchanged.

The instruction **UFL** processes the A0 stack top as a number of real type and converts it to the udint type in the range of <0; 4 294 967 295>. The result is saved on the A0 stack top. The content of the other layers remains unchanged.

The instruction **IFL** processes the A0 stack top as a number of real type and converts it to the dint type in the range of <–2 147 483 648; +2 147 483 647>. The result is saved on the A0 stack top. The content of the other layers remains unchanged.

Flags

	.7	.6	.5	.4	.3	.2	.1	.0
S1	-	-	-	-	-	-	-	S

S1.0 (S) - 1 - the result is valid
0 - format range exceeded, invalid result

UDFL	**Conversion of lreal value to udint**
IDFL	**Conversion of lreal value to dint**
DFF	**Conversion of lreal value to real**

Instruction	**Input parameters**									**Result**								
	stack									stack								
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
UDFL							NLR			-	A7	A6	A5	A4	A3	A2	NUD	
IDFL							NLR			-	A7	A6	A5	A4	A3	A2	ND	
DFF							NLR			-	A7	A6	A5	A4	A3	A2	NF	

ND - value of lreal type
NUL - value converted to udint type
NIL - value converted to dint type
NF - value converted to real type

Operands

		udint	dint	real
UDFL	w/o operand	C		
IDFL	w/o operand		C	
DFF	w/o operand			C

Function

UDFL - conversion of value of lreal type to udint type
IDFL - conversion of value of lreal type to dint type
DFF - conversion of value of lreal type to real type

Description

The instruction **UDFL** processes the A01 stack top as a number of lreal type and converts it to the udint type in the range of <0; 4 294 967 295>. The stack is shifted one level back and the result is saved at the A0 stack top.

The instruction **IDFL** processes the A01 stack top as a number of lreal type and converts it to the dint type in the range of <–2 147 483 648; +2 147 483 647>. The stack is shifted one level back and the result is saved at the A0 stack top.

The instruction **DFF** processes the A01 stack top as a number of lreal type and converts it to the real type. The stack is shifted one level back and the result is saved at the A0 stack top.

Flags

	.7	.6	.5	.4	.3	.2	.1	.0
S1	-	-	-	-	-	-	-	S

S1.0 (S) - 1 - the result is valid
0 - format range exceeded, invalid result

12. PID CONTROLLER INSTRUCTIONS

CNV Data processing from analog inputs

Instr.	Input parameters								Result							
	stack								stack							
	A7	A6	A5	A4	A3	A2	A1	A0	A7	A6	A5	A4	A3	A2	A1	A0
CNV			INDF	TAU	INDM	FCE	AVAL	MODE			INDF	TAU	INDM	FCE	AVAL	VAL

INDF - register index of filter status variable - optional, see text bellow
TAU - filter time constant t - optional, see text bellow
INDM - register index of variable for scaling - optional, see text bellow
FCE - activated functions
AVAL - measured analog value (int/real type)
MODE - type of conversion
VAL - result of conversion (int/real type)

Operands

		int	real
CNV	w/o operand	C	C

Function

CNV - function for conversion of measured analog values

Description

The instruction **CNV** was designed primarily for conversions of values from current analog inputs at the older systems. The analog modules of TC700 PLC provide normalized values in the registers directly and they do not require this conversion.

The **CNV** instruction processes the values of type int or real. The selection of the type being processed is expected at the stack top in the MODE parameter. If MODE = 0, then the instruction **CNV** works with the values of type int. If MODE = $10000, the **CNV** instruction works with values of type real.

The instruction **CNV** further contains the following functions:

- scaling by linear interpolation
- first order filtering
- square root

The particular functions differ in their demands for the number of scratchpad registers used and the number of the parameters written at the stack (see the following descriptions of the particular functions). The functions are activated by means of the control bits of the value FCE at the A2 stack layer. The functions can be combined.

	.7	.6	.5	.4	.3	.2	.1	.0
FCE	-	-	-	-	SQ	FI	F1	F0

FCE.1,.0 - 00 – linear interpolation is off
11 - linear interpolation between two points

FCE.2 (FI) - filtering of values by linear filter of the first order
0 - off
1 - on

FCE.3 (SQ) - square root
0 - off
1 - on

Detailed information on the particular functions are given in the following text.
The instruction **CNV** does not shift the stack and writes the result of conversion on its top.

Flags

	.7	.6	.5	.4	.3	.2	.1	.0
S1	-	-	-	-	-	-	-	S

S1.0 (S) - 1 - instruction is performed
0 - data structure is out of scratchpad, instruction is not performed

S34 = 20 ($14) scratchpad range exceeded

Scaling by linear interpolation

Scaling by linear interpolation is performed at setting of the bits FCE.0 and FCE.1 to log.1 (layer A2 of the stack). The instruction **CNV** finds for input value its functional value on the line given by two points. The co-ordinates of the two points are in the instruction data structure, which is given by the index of the initial register INDM (the layer A3).

This function is useful for example for linear calibration of a measuring string, range change of resistance transmitters, generating of the range of the resistance transmitter, generating of required values, defined by means of linear sequences given by the table.

Data structure:
MinY - 1st co-ordinate of the 1st point of line (e.g. measured actual value) (int/real type)
MaxY - 1st co-ordinate of the 2nd point of line (e.g. measured actual value) (int/real type)
MinW - 2nd co-ordinate of the 1st point of line (e.g. set value) (int/real type)
MaxW- 2nd co-ordinate of the 2nd point of line (e.g. set value) (int/real type)

The layers A4 and A5 of the stack are not used. If the filtering function is not activated, which requires them, it is not necessary to specify them.

The ranges of standardized values are dependant on the type of PLC.

Note

The instruction **CNV** requires that the data structure does not begin closer than 8 bytes from the end of the scratchpad.

Example 1

We want to correct the measurement of a resistance transmitter with the range of 100 ÷ 1000 Ω to the range of 0 ÷ 10 000. The resistance transmitter is connected to the channel 0 of the analog module IT-7604. We use an input value at FS format (Full Scale – variable of integer type – 16 bit with sign).

```
;Input type for connection of resistance transmitter in the range of 0 ÷
1000 Ω at Full Scale format (FS) selected
;
#reg int  MinY,       ;1st co-ordinate of 1st line point (actual value)
          MaxY,       ;1st co-ordinate of 2nd line point (actual value)
          MinW,       ;2nd co-ordinate of 1st line point (set value)
          MaxW        ;2nd co-ordinate of 2nd line point (set value)
#reg int  resistance  ;resulting value
;
```

```
P 63
     LD   3000       ;actual value
     WR   MinY
     LD   30000
     WR   MaxY
     LD   0          ;set point
     WR   MinW
     LD   10000
     WR   MaxW       ;the equation of this line is
                     ;y = (10000(x-3000))/30000
E 63
;
P 0
     LD   __indx MinY     ;INDM – reg. index where data structure begins
     LD   3               ;FCE  - linear interpolation
     LD   r0_p5_AI0.FS    ;AVAL - load channel 0 at FS format
                          ;       of module in the rack 0 at position 5
     LD   0               ;MODE – int type
     CNV
     WR   resistance      ;VAL  - resulting value
E 0
```

Example 2

We want to calculate the tepmerature acquired by a sensor working within a range of 4 ÷ 20 mA connected to channel 1 of the analog module IT-7604. As the input value, a value in the ENG format shall be used (variable of type real - floating point).

```
;selected type of input for connection of current transmitter with range
;0 ÷ 20 mA in format ENG
;
#reg real  MinY,     ;1st coordinate of 1st line point (real value)
           MaxY,     ;1st coordinate of 2nd line point (real value)
           MinW,     ;2nd coordinate of 1st line point (required value)
           MaxW      ;2nd coordinate of 2nd line point (required value)
#reg real  temperature     ;resultant value
;
P 63
     ld   4.0             ;real value
     WR   MinY            ; 4 mA
     ld   20.0
     WR   MaxY            ; 20 mA
     ld   -30.0           ;required value
     WR   MinW            ; -30 st.C
     ld   70.0
     WR   MaxW            ; 70 st.C
E 63
;
P 0
     LD   __indx (MinY)   ;INDM - register index where data structured
                          ;begins
     LD   3               ;FCE  - linear interpolation
     LD   r0_p5_AI1.ENG   ;AVAL - read of channel 1 in format ENG
                          ;       of module in rack 0 position 5
     LD   $10000          ;MODE - type real
     CNV
     WR   temperature     ;VAL  - resultant value
E 0
```

First order filtering

This function can be combined with all previous functions. The resulting value after previous function is filtered by the numerical filter of the first order (averaging is performed). The sampling frequency is given by the cycle length of the controller.

The filter is defined by the following expression:

$$y_t = \frac{y_{t-1} \cdot \tau + x}{\tau + 1}$$

x - converted value of analog input
y_t - CNV output (value VAL)
y_{t-1} - recent output CNV (value VAL)
t - 1st order filter time constant

The value of the constant τ is specified in ms in the parameter TAU (the layer A4). If TAU = 0, the measured value is used in the filter status variable (filter initialization).

Multiplied use of the instruction **CNV** for filtering is conditional on the individual declaration of the filter status variable for each instruction **CNV** (the index of the initial register of the variable INDF is passed on the level A5). Two or more **CNV** instructions must not work above one status variable (except own initialization).

The analog modules of TC700 have usually implemented this function.

Note

The instruction **CNV** requires that the declared status variable does not begin closer than 4 bytes from the end of the scratchpad.

Example 1

The input signal must be filtered by a filter of approx. 0,2 s. The filtered value is the result at the variable *temperature*.

```
#reg int   adata,temperature
#reg usint AuxD[4]    ;auxiliary status variable
;
P 63
;filter initialization
     LD    __indx AuxD     ;INDF - index of filter status variable
     LD    0               ;TAU  - tau=0, load to the status variable
     LD    0               ;INDM - not used
     LD    4               ;FCE  - filtering only
     LD    adata           ;AVAL - load input
     LD    1531            ;MODE – int type
     CNV
E 63
;
P 0
     LD    __indx AuxD     ;INDF - index of filter status variable
     LD    200             ;TAU  - tau = 200 ms
     LD    0               ;INDM - not used
     LD    4               ;FCE  - filtering only
     LD    adata           ;AVAL - load input
     LD    0               ;MODE – int type
     CNV
     WR    temperature     ;VAL  - resulting value
E 0
```

Example 2

The input value of type real is necessary to filter by a filtr of approx. 0,2 s. The result is the filtered value in variable *teplota*.

```
#reg real   adata, temperature
#reg usint  AuxD[4]   ;auxiliary status variable
;
P 63
;inicializace filtru
     LD    __indx (AuxD)   ;INDF - index of status variable of filter
     LD    0               ;TAU  - tau=0, transcription to status
                           ;variables
     LD    0               ;INDM - not used
     LD    4               ;FCE  - only filtering
     LD    adata           ;AVAL - load input
     LD    $10000          ;MODE - typ real
     CNV
E 63
;
P 0
     LD    __indx (AuxD)   ;INDF - index of status variable of filter
     LD    200             ;TAU  - tau = 200 ms
     LD    0               ;INDM - not used
     LD    4               ;FCE  - only filtering
     LD    adata           ;AVAL - load input
     LD    $10000          ;MODE - type real
     CNV
     WR    temperature     ;VAL  - resultant value
E 0
```

Square root

This function can be combined with all previous functions. The resulting value after previous functions is extracted.

The layers A3, A4 and A5 of the stack are not used. If they are not required by some of the previous functions, it is not necessary to specify them.

Example

Let us do square root of a value.

```
#reg uint adata,squart
;
P 0
     LD    8          ;FCE  - square root
     LD    adata      ;AVAL - load input
     LD    0          ;MODE – int type
     CNV
     WR    squart     ;VAL  - resulting value
E 0
```

PID PID controller

Instr.	Input parameters								Result							
	stack								stack							
	A7	A6	A5	A4	A3	A2	A1	A0	A7	A6	A5	A4	A3	A2	A1	A0
PID						INPUT3	INPUT2	INDEX						INPUT3	INPUT2	DET

INPUT3- y_3 - servo-valve position - optional, see text bellow
INPUT2- y_2 - ratio control - optional, see text bellow
INDEX - register index where controller data structure begins
DET - error and action detection (usint type)

Operands

PID	w/o operand	C

Function

PID - PID controller

Description

By means of the instruction **PID** it is possible to control such systems at which the transient period is at least one digit place longer then controller sampling (e.g. during 1 s sampling, it is possible to control a system with a transient period taking tenths of seconds). Controller sampling must be set with regard to the PLC cycle time in such a way that a certain accuracy of sampling is ensured. It is useful when the controller sampling is set one digit place higher than the PLC cycle time (e.g. the cycle time of 30 ms allows to set the controller sampling to 300 ms).

The PIDMaker tool is designed for PID controller setting and debugging. The tool is an integrated part of the MOSAIC development software.

The instruction **PID** allows integration into the P41 interrupt process, which is inserted every 10 ms. This allows to set the controller sampling to 10 ms and so control the systems with a short transient period. The time of interrupt process execution must not exceed 5 ms!

The basic advantage of the controller realized by the instruction **PID** is the integration of all of its variables to the PLC system. So, the user has a possibility to define any conditions by means of PLC instructions for alarms, action devices control as well as for controller settings dependent on the status of the entire technology. The measurement of controlled values is ensured by PLC analog units.

Note: In the text, the word "controller" is often used as a synonym for a control algorithm. Further, the name "PID controller" established in practice is used instead of a more accurate abbreviation PSD for a digital version of a classical continuous algorithm.

The instruction **PID** ensures in optional multiplies of 10 ms the calculation of the value of the action according to the PID algorithm or PPID, to be more accurate. The algorithm control is ensured by means of the variable structure, which is defined on the PLC registers. In principle, the PID works according to the discrete version of the equation:

$$u(t) = K\left[e(t) + \frac{1}{T_I}\int_0^t e(\tau)d\tau + T_D \frac{de(t)}{dt} \right]$$

The control algorithm ensures the following functions:

1. **Smooth switching-over of the manual and automatic modes**, which is based on the estimation of the status of the system being controlled. **The parameters of the controller can be modified in the automatic mode, too.**
2. Switching-over of the second proportional band according to the sign of deviation. (For a faster suppression of the overshoot of the system being controlled.
3. Setting of action zone range of the I and D parts in % of the range of the measured value.
4. Setting of the controller dead band, i.e. enabling of new changes of actions from the given level of deviation in % of the range of the measured value.
5. Incremental control of positioning valves also without necessity of measuring of their position. If the position of the valve is measured, the action correction is performed according to the actual position of the valve.
6. Realization of ration control, filtering or linear interpolation of the required value (ramp).
7. Entry of action ranges. (This allows realizing for example an action range from 0 to 100%, or from –100% to +100%.) For the action, it is possible to enter the limitation of its increment, too.

The parameters of the control algorithm are entered to the reserved data space in the register zone. In the process P63 it is useful to enter all required controller parameters (they are zero by default!). In the course of the control process, it is necessary to enter the value of the controlled value or the auxiliary value of the position of the actuating mechanism during incremental control to the variable in question. After computation, it is enough to transcript the value form the variable *ConOut* to the analog output unit. At on / off control, the bits from the variable *Status* are transferred to the outputs of the binary unit.

Note: In the course of the control process, it is possible to modify also the parameters of the controller by assigning the values to the variables in question. The control algorithm ensures the right function also for a non-stationary controller (with time variant parameters).

The instruction **PID** uses a 72 byte data structure in which all of its variables are saved. Each controller must have its exclusive structure reserved!

The list of variables, which acquire the corresponding position in the register zone is carried out as follows:

```
#struct _PID
     int MinY,        ;minimum value measured
     int MaxY,        ;maximum value measured
     int Input1,      ;measured value (y, controlled)
     int gW,          ;set point (w), target
     int ConW,        ;actual set point
     int tiW,         ;filter time constant w or time interval
                      ;ramps in multiples of the output cycle
     int Dev,         ;deviation [%]
     int Output,      ;direct action (u) required by algorithm or
                      ;manually [%]
     int LastOut,     ;previous action, i.e. delayed by 1 step [%]
     int CurOut,      ;output really required [%]
     int ConOut,      ;output realized by the controller
     int DefOut,      ;default value of output at measurement error [%]
     uint MinU,       ;minimum permissible action [%]
     uint MaxU,       ;maximum permissible action [%]
     uint dMaxU,      ;maximum permissible action increment [%]
     uint OutCycle,   ;length of output cycle (sampling period)
```

```
uint PBnd,        ;proportional band [%]
uint RelCool,     ;auxiliary proportional band [%]
uint Ti,          ;integrating constant [s]
uint Td,          ;derivative constant [s]
uint EGap,        ;symmetrical dead band [%]
uint DGap,        ;symmetrical band of deviation, where
                  ;derivation part is active [%]
uint IGap,        ;symmetrical band of deviation, where
                  ;integrating part is active [%]
uint Control,     ;control word
usint Status,     ;status
usint[23] AuxD    ;auxiliary variables - write prohibited!
```

The parameters of the data structure are described in more detail in the following text.

The instruction **PID** moves back at the stack top the result of error detection and edge conditions.

	.7	.6	.5	.4	.3	.2	.1	.0
DET	EY3	EY2	EY1	UMX	UMN	ER2	ER1	ER0

DET.2,.1,.0 - parameter errors
000 - parameters are OK
001 - invalid time specification of the output cycle *OutCycle* (def. value 1)
010 - invalid value of action limitation (default value 0 ÷ 10000)
011 - invalid value of action increment (default value 10000)
100 - *MinY* ≥ *MaxY* (upper and bottom limit of the input value) (default value 0 ÷ 10000)
101 - proportional band *PBnd* is zero (default value 1000)
110 - second proportional band *RelCool* is zero, (default value 1000)
If some of the parameters *OutCycle*, *PBnd*, *RelCool* is wrong, the instruction **PID** sets the default value of the invalid parameter. A mistake is indicated only in the cycle, when it happened and when it was corrected by the instruction. In the next cycle, this is not indicated any more, i.e. log.0 is on the lower 3 bits.

DET.3 (UMN)- detection of minimum action
1 - action is less than *MinU*
DET.4 (UMX)- detection of maximum action
1 - action is greater than *MaxU*
DET.5 (EY1) - detection of measuring error y_1 (*Input1*)
1 - y_1 out of interval <*MinY*, *MaxY*>
DET.6 (EY2) - detection of measuring error y_2 (*Input2*)
1 - y_2 out of interval <*MinY*, *MaxY*>
DET.7 (EY3) - detection of measuring error y_3 (*Input3*)
1 - y_3 out of interval <*MinY*, *MaxY*>

Flags

	.7	.6	.5	.4	.3	.2	.1	.0
S1	-	-	-	-	-	-	-	S

S1.0 (S) - 1 - instruction is performed
0 - data structure is out of scratchpad, instruction is not performed

S34 = 20 ($14) scratchpad range exceeded

Description of parameters of the data structure

MinY	-	Minimum value measured. Used for deviation standardization.
MaxY	-	Maximum value measured. Used for deviation standardization.
Input1 (y1)	-	Measured (controlled) value.
gW (w)	-	Set point, within the interval of measured value <*MinY*, *MaxY*>.
tiW	-	Time constant for the filter of the first order or linear interpolation of required value in multiples of *OutCycle*.
ConW	-	Current set point.
Dev (e)	-	Deviation of actual value from set point [%].
Output	-	Output required by algorithm or manually. The action can be in the range of –10000 to +10000 (i.e. –100,00% to +100,00%) at the most. It is thus standardized in such a way that for amplification 1 (proportional band 100%) and deviation 100,00% the action is 100,00%. The range is always limited to the range of <*MinU*, *MaxU*>.
LastOut	-	Previous action, i.e. delayed by 1 step [%] or valve position (see cascade control).
CurOut	-	Output really required in the given step [%] or action increment.
ConOut	-	Output realized by the controller [%] or actual value realized by the output unit or by time-proportional on / off control in the absolute value.
DefOut	-	Default value of output at measurement error.
MinU	-	Minimum permissible action [%]. Direct action cannot be less than this value.
MaxU	-	Maximum permissible action [%]. Direct action cannot be greater than this value.
dMaxU	-	Maximum permissible action increment [%]. A new action cannot differ in the absolute value by more than *dMaxU* from the previous value.
OutCycle	-	Length of output cycle, sampling period [hundredths of s]. It specifies a period, within which the action does not change or the period of repetition frequency for time-proportional control. The minimum value is 1, i.e. 10 ms, and it can be set up to 65535, i.e. more than 10 minutes.
PBnd	-	Proportional band. It is set in the range of 1 to 30000 (0,1 to 3000,0%). The amplification is given as follows: $K = \frac{1000}{PBnd}$
RelCool	-	Auxiliary proportional band for negative deviation. It is set in the range of 1 to 30000 (0,1 to 3000,0%). The amplification is given as follows: $K = \frac{1000}{PBnd} \cdot \frac{1000}{RelCool}$ This results in the fact that for *RelCool* = 1000 (100,0%) this part is without any influence.
Ti	-	Integrating constant [tenths of s]. It is set in the range of 0 to 30000 (0 to 3000,0 s). For zero value, the integrating part is off.
Td	-	Derivation constant [tenths of s]. It is set in the range of 0 to 30000 (0 to 3000,0 s).
Egap	-	Symmetrical dead band. The range is from 0 to 10000 (0 to 100,00%). If the deviation is less than *EGap*, i.e. it is in the dead band, the action remains unchanged.
Dgap	-	Symmetrical band of deviation, where the derivation part is active. The range is from 0 to 10000 (0 to 100,00%). It means that the derivation part is still active for *DGAP* = 10000.

Igap - Symmetrical band of deviation, where integrating part is active. The range is from 0 to 10000 (0 to 100,00%). It means that the integrating part is still active for *IGAP* = 10000.

Control - Control word is used for setting of the controller function. The controller can be operated in automatic, manual or emergency mode. It can be used as a controller with direct or incremental algorithm. If a servo-valve is used as an actuator, it is possible to use for the correction of the action increment the measured value of its position, in this case we talk about cascade control. If the output cycle is longer, it is possible to realize the time-proportional control of the output on / off. The resolution is based on the controller cycle time. For example, if the controller cycle time is 100 ms and the output cycle 10 s, the resolution 1%.

.15	.14	.13	.12	.11	.10	.9	.8	.7	.6	.5	.4	.3	.2	.1	.0
FU2	FU1	FU0	-	-	P41	RIO	RF	HR	AM	IP	BU	KC	A12	AO	RC

RC - 1 - request for cold start of controller (instruction itself sets the bit to zero)
AO - 1 - shift of zero of controller output for range 4 ÷ 20 mA
A12 - 1 - output to 12 bit D/A converter
KC - 1 - cascade control
BU - 0 - unified output
1 - binary output (time-proportional, on / off control)
IP - 0 - direct control
1 - incremental control
AM - 0 - manual mode
1 - automatic mode
HR - 1 - more reliable measurement mode, two measured values are used
RF - 0 - modification of required value by the first order filter
1 - modification of required value by linear interpolation
RIO - 1 - ratio control
P41 - 1 - instruction **PID** is called in the process P41, i.e. in the raster of 10 ms (only for CPUs of series B and C)

Note:
Also in this case it is possible to use the setting of the period at the variable *OutCycle*. This has a practical meaning only for on / off control. For example, if *OutCycle* = 100, the period is 1 s and the resolution of the width of the output pulse is 10 ms, i.e. 1%. When 220 V output unit is used as an example, it is possible to realize the output of the cyclic control type, i.e. with resolution of one period of phase voltage.

FU2-FU0 - filtering of short actions

Generally it is valid that if *CurOut* < 32 ∗ FU, the action is not performed and the content of *CurOut* is set to zero.
0 - all actions permitted
1 - suppressed actions less than 32 (i.e. 0,32%)
2 - suppressed actions less than 64 (i.e. 0,64%)
3 - suppressed actions less than 96 (i.e. 0,96%)
4 - suppressed actions less than 128 (i.e. 1,28%)
5 - suppressed actions less than 160 (i.e. 1,6%)
6 - suppressed actions less than 192 (i.e. 1,92%)
7 - suppressed actions less than 224 (i.e. 2,24%)

Status - Used especially for bit value transfer for on / off control, this is to say if the action is designed as time-proportional control (pulse width). Further, it contains error bits of measurement.

.7	.6	.5	.4	.3	.2	.1	.0
-	EY3	EY2	EY1	DR	U–	UC	UH

UH - output for positive action, i.e. heating
UC - output for negative action, i.e. cooling
U– - action signalization
0 - positive action
1 - negative action
DR - detection of the course of linear interpolation of required value
1 - interpolation active
EY1 - detection of measuring error y_1 (*Input1*)
1 - y_1 out of interval <*MinY*, *MaxY*>
EY2 - detection of measuring error y_2 (*Input2*)
1 - y_2 out of interval <*MinY*, *MaxY*>
EY3 - detection of measuring error y_3 (*Input3*)
1 - y_3 out of interval <*MinY*, *MaxY*>

AuxD - Auxiliary variables of the controller. It is forbidden to write into this zone!!

Ranges and formats of measured values

Into the register *Input1* (y_1) the measured (controlled) value from the analog input is entered. The ranges of the measured value *Input1* are entered into the registers *MinY and MaxY*.

At the A1 layer of the stack, the value *Input2* (y_2) is used for ratio control or for more reliable measuring. In the first case, the bit RIO in the register *Control* must be set to log.1, in the second case, the bit HR to must be set to log.1 (see Control of required value). If these two modes are not used, the variable does not have to be specified.

At the layer A2 of the stack is the value *Input3* (y_3) used for measuring of the valve position. In this case we talk about cascade control and in the register Control the bit KC must be set to log.1 (see Cascade control). If this mode is not used, the variable does not have to be specified.

Control of required value and deviation computation

Control bits:	*Control*	- RF, RIO, RC
Diagnostic bits:	*Status*	- DR
Registers:	*gW, ConW, tiW*	

The set point is specified into the register *gW*. But the controller takes for deviation computation the required value from the register *ConW*!

Filtering (RF = log.0)

If the bit RF has the value of log.0, the filter of the 1st order of the required value is activated. The register *tiW* specifies the time constant for this filter.

If *tiW* = 0, after entering a new value into the register *gW,* the value is also received into the register *ConW*, this is to say *ConW* = *gW*.

If *tiW* > 1, then after changing of *gW* the value filtered with the time constant *tiW* in *ConW.* For example, for *tiW* = 5, *OutCyclus* = 10 the time constant w = 5 s.

If the value of the bit RC = log.1 after restarting the PLC, then in the first cycle *ConW* = *Input1*. The same value is received into the filter status variable. It this way, reaching of the required value can be realized with a minimum overshoot.

Ramp (RF = log.1)

If the bit RF has the value of log. 1, linear interpolation of the required value is activated. The register *tiW* specifies the time of interpolation of the required value.

If *tiW* = 0, it is possible to change the values of *ConW* and *gW* registers without any mutual influence.

After specification of the required time into *tiW* linear interpolation is performed from the value at *ConW* to *gW*. From the table of required values realized by the PLC instructions it is possible to realize for example any temperature cycles. The selection of a new value can be synchronized by means of the bit DR.

If the value of the bit RC = log.1 after restarting the PLC, the value of the register *ConW* is received into the status variable, but *ConW* is not changed.

Deviation computation

Deviation computation is performed according to the value of the bit RIO. Internally, the deviation is standardized within the range of –10000 ÷ +10000 (–100,00% ÷ +100,00%) as follows:

Tow control (RIO = log.0):

$$e = 10000 \cdot \frac{ConW - y_1}{MaxY - MinY}$$

Ratio control (RIO = log.1):

$$e = 10000 \cdot \frac{\frac{ConW}{100} \cdot y_2 - y_1}{MaxY - MinY}$$

In this case, *ConW* is specified in the range of 0 to 10000. For the same ration y_1 and y_2 *ConW* = 100. When the control is finished, i.e. e = 0, y1/y2 equals to the required ration *ConW*/100.

Mode of more reliable measurement

If the value of the bit HR = log.1, the controller uses two measurement input *Input1* (y_1) and *Input2* (y_2) depending on the occurrence of a measuring error:

both measurements are in order - for deviation computation, average from y_1, y_2 is used.

only one measurement is in order - for deviation computation is used that measurement, which is in order. The controller is not switched to the emergency mode! Error indication of the particular measurement is in the register *Status* and at the A0 stack top after the execution of the instruction PID.

error of both measurements - emergency state (see Emergency mode)

The diagnostic bits for the measurement are permanently active.

Attention! At the variable *Input1* of this mode, the controlled value used for control is saved after the execution of the instruction, i.e. either averages y_1, y_2, or the valid value from y_1, y_2, or the failure code ($±7FFF).

Controller modes

Control bits:	*Control*	- AM
Diagnostic bits:	*Status*	- EY1, EY2, EY3
	A0	- EY1, EY2, EY3

The controller can be operated in automatic, manual or emergency mode.

Manual mode (AM = log.0)

Switching into the **manual** mode (AM = log.0) from the automatic mode is based on the suspension of the calculation of the controller action. The controller permanently displays the changes of the deviation and diagnoses measurement errors. After specification of a new action into the register *Output* the action starts immediately with the effect of the limitation of action increment (speed)! Into the *Output* a direct output value is entered.

Automatic mode (AM = log.1)

Switching into the **automatic** mode (AM = log.1) from the manual mode is smooth varies according to whether the integrating part is in the controller.

- In case of a PD controller, the last manually entered value of the action is the offset value, which is added to the parts of the PD controller. This feature can be used for example for the control of the systems with shifted zero, where the failures have a character of a "noise" with the zero mean value and the I-part is not appropriate to be specified. For astatic systems, the last entered value in the manual mode must be 0 (for direct algorithm).
- In case of a controller with the I-part, the initial condition of the controller is given by the estimate of the steady state of the system being controlled.

In the close range of the steady state and after switching, the action is practically unchanged. Out of steady state, the integrating part is cleared. After switching AM = log.1 the first step of the control is performed immediately.

Emergency mode

If a measuring error occurs (first occurrence of error), the value of the parameter *DefOut* is put into to the variable *Output* and the controller is switched to the manual mode. In case of a permanent error condition this value remains unchanged, the controller is operated in the manual mode.

If the value *DefOut* is greater than 10000, then at the error condition, the last value of the action remains at *Output*. After switching into the manual mode the action is controlled by the value *Output*. The measuring error is indicated after completing the instruction at the A0 stack top in bits of the detection of measuring error EY1 and EY2. At the variable Status, the flag of the occurrence of the measuring error is saved in the bits EY1 or EY2. The bits are set at the measuring error and reset only after cold start of the controller (by setting the bit RC to log.1). The operation of the bits EY2 is active only in case of using of the auxiliary input *Input2*.

Cascade control (KC = log.1)

Input3 (y_3 - transmitted at the layer A2 of the stack) is used as the third measured value for measurement of the valve position. In this case, the action of the master loop is corrected according to this value, if the control bit KC is set (cascade control) in the control word Control. The measured value of the resistance transmitter being measured by the analog unit must directly express the opening of the valve in tenths of per mille, i.e. it must have to range 0 to 10000.

This can be easily done by means of the instruction **CNV**. In this case, the position of the valve is at the variable *LastOut*.

If the measured input y_3 is faulty, the controller is not switched into the emergency mode. It only cleats the bit KC in the control word and continues the control without measuring of the valve position. At the same time, the error bit EY3 is set in the register *Status*, or at the A0 stack top, as the case may be.

Controller algorithms

Control bit:	*Control*	- IP

The controller works as a direct (positional) or incremental.

Direct algorithm

Direct algorithm (IP = log.0) is a classical algorithm where simple non-linear bands were added, which can prevent from undesirable reactions of the controller under non-standard situations, such as after switching of a controlled circuit. It is suitable to set the dead band according to the estimate of the value of the dispersion variance (steady states can be significantly influenced).

At the variable *CurOut* the actually required output is given back.

Incremental algorithm

Incremental algorithms (IP = log.1) give back at the variable *CurOut* the increment of a new action. At the variable *Output* the direct value of the action is still kept. The incremental algorithm is designed for control of astatic systems (especially with a positional valve). Even in case of manual control, the direct value of the action is entered, which means that the controller monitors the action after the transmission with zero pole, which is considered to be part of the controller.

For example, if a positional valve, an estimate of the valve output is still available even without a resistance transmitter used for sensing of its position. In this case, it is really an estimate and in case of manual control it is always necessary to perform calibration of the output value according to actual valve position (see text bellow).

Switching between both control algorithms should be done the manual mode.

Controller outputs

Control bits:	*Control*	- BU, KC, A12, AO
Diagnostic bits:	*Status*	- U–, UH, UC
	A0	- UMX, UMN

The output can be unified, continuous, realized by means of an analog output unit or binary unit, time-proportional control on / off, realized by means of a binary output unit.

The continuous output of the controller (bit BU = log.0), without regard to the mode and the algorithm, gives back **the absolute value** of the action at the variable *ConOut*. According to the bit U–, indicating the minus sign of the action, it is possible to control two analog outputs, etc.

The parameter *OutCycle* here has the meaning of repetition frequency. The output value is always limited to the range of 0 ÷ 10000. In case of setting the bit A12 = log.1 at the variable *Control* the output value is standardized to the range of 4095 i.e. 12 bits. If the bit setting is AO = log.1, the range 0 ÷ 10000 is transformed to the range of 2000 ÷ 10000.

The result is saved to the variable *ConOut* in both cases In this way, it is possible to control the unified outputs of the analog units directly.
The binary output on / off (bit BU = log.1), time-proportional control, is used for direct control of the actuator. Here is *OutCycle* the value of the output cycle, i.e. repetition frequency period. In case of incremental control, the time of real switching at one output cycle and after its elapsing the value of the action change is adjusted automatically.

For a better resolution, during incremental control, it is possible to use limitations of action increment. This is to say it is possible that during the period of the output cycle the value of this change is maximized.

If cascade control of the servo-valve is used, the travel speed of the positional servo-valve is given by the variables *dMaxU* and *OutCycle*.

For example, *dMaxU* = 1000 i.e. 10% and *OutCycle* = 1000 i.e. 10,0 s.

In this case, the speed of the valve of 1% in a second is specified, i.e. the time of valve overtravel is 100 s. If the mean PLC cycle time is 100 ms, the resolution is 0,1%.

Calibration of servo-valve without position resistance transmitter

The variable *Control* = $10

A self-actuated controller with the range of –10000 ÷ +10000, *dMaxU* = 10000. In the manual mode we set the valve position by switching by means of manually specified values –10000 or +10000.

The variable *Control* = 0

To the variable *Output* we will enter a value of the valve position with the accuracy of a hundredth of per cent and we set *MinU* = 0. This step is necessary due to suppression of undesirable control pulses to the valve.

The variable *Control* = $30

An incremental controller with the range of 0 ÷ +10000 is set. It is now possible to enter manually the required valve positions to *Output*.

The changes are available after elapsing of *OutCycle*! From this reason it is necessary to set a low value at *OutCycle.*

Example

Let us assume we control temperature measured with IT-7604 module of TC700. The conversion and filtration is performed directly by the module. The actuating mechanism used is a servo-valve with position measuring, resistance transmitter (RT) 0 to 200 Ω. The RT data is filtered, too. The controller is set as a cascade one, incremental with binary control (the valve is arranged in a cascade).

```
#program Cascade
;
#reg bool  Output0, Output1
;
#struct _PID
     int   MinY,     ;minimum measured value
     int   MaxY,     ;maximum measured value
     int   Input1,   ;measured value (y, controlled)
     int   gW,       ;required value (w), target
     int   ConW,     ;current required value
     uint  tiW,      ;time constant of filter w or time interval
                     ;ramps in multiples of output cycle
     int   Dev,      ;deviation [%]
     int   Output,   ;direct intervention (u) required by algorithm or
                     ;manually [%]
```

```
     int   LastOut,  ;recent intervention, i.e. 1 step delayed [%]
     int   CurOut,   ;output really required %]
     int   ConOut,   ;output made by controller
     int   DefOut,   ;default value of output at measurement error[%]
     uint  MinU,     ;minimum permitted intervention[%]
     uint  MaxU,     ;maximum permitted intervention[%]
     uint  dMaxU,    ;maximum permitted increment of intervention[%]
     uint  OutCycle, ;output cycle length (sampling period)
     uint  PBnd,     ;proportionality band[%]
     uint  RelCool,  ;auxiliary proportionality band [%]
     uint  Ti,       ;integration constant [s]
     uint  Td,       ;derivation constant [s]
     uint  EGap,     ;symmetrical dead band [%]
     uint  DGap,     ;symmetrical deviation band where
                     ;derivation part is active [%]
     uint  IGap,     ;symmetrical deviation band, in which
                     ;integration part is active [%]
     uint  Control,  ;control word
     usint Status,   ;status
     usint [23] AuxD ;auxiliary variables - write prohibited!
;
#reg _PID  PID       ;Data pro PID
;
;Initialization of temperature control:
P 63
;initialization of PID
     LD    0
     WR    PID~MinY
     LD    10000
     WR    PID~MaxY
     LD    0         ;output values range
     WR    PID~MinU
     LD    10000
     WR    PID~MaxU
     ;
     LD    1000      ;Definition of valve overtravel speed
     WR    PID~dMaxU ;10,00% (maximum permissible
     LD    1000      ;increment of valve position) in 10 s,
     WR    PID~OutCycle    ;i.e. the time of valve overtravel is 100 s.
     ;
     LD    11000     ;DefOut>10000, i.e. after emergency
     WR    PID~DefOut ;the output value remains unchanged.
     ;               ;The controller is switched to manual mode.
     ;
     LD    500       ;temperature set point 50,0°C (in tenths)
     WR    PID~gW
     ;
;Setting of controller parameters:
     LD    1000      ;Proportional band 100,0%, i.e. amplification 1
     WR    PID~Pbnd
     LD    1000      ;Second proportional band 100,0%
     WR    PID~RelCool
     LD    0
     WR    PID~Ti    ;Without integration.
     LD    10
     WR    PID~Td    ;Derivation part 1,0 s
     LD    10
     WR    PID~EGap  ;Dead band 0,1%
```

```
      LD    10000
      WR    PID~DGap   ;Derivation permitted within the whole range
      LD    0          ;Without integration
      WR    PID~IGap
      ;
      LD    %01111001  ;Control word: incremental control of valve in
                       ;cascade
      WR    PID~Control
E 63
;
P 0
;temperature measurement
      LD    r0_p5_AI0.PCT   ;percentual value
      MUF   100
      IFW                   ;conversion real -> int
      WR    PID~Input1      ;write to PID structure
      ;
;measurement of the resistance transmitter
      LD    r0_p5_AI1.PCT   ;percentual value
      MUF   100
      IFW                   ;conversion real -> int
                            ;gives back at A0 the measured value
                            ;for PID Input3=[A2]
      LD    0               ;value not used Input2=[A1]
      LD    __indx (PID~MinY);data index structure PID=[A0]
      PID
      LD    PID~Status.0
      WR    output0         ;open valve more
      LD    PID~Status.1
      WR    output1         ;open valve less
E 0
```

13. INSTRUCTIONS OF TERMINAL OPERATION AND OPERATIONS WITH ASCII CHARACTERS

TER	**Terminal instruction**

Instruction	Input parameters									Result								
	stack									stack								
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
TER n							TXT	DISP								TXT	DISP	

TXT - register number, where control variables for the instruction are located (udint type)
DISP - register number, where videoRAM memory begins (udint type)

Operands

TER	n	C

n - type of operator panel (7 - ID-07, 8 - ID-08)

Function

TER - terminal instruction

Description

The instruction **TER** reduces the necessity of programming the operation of the operator panel to a minimum level. It processes text definition and according it creates the current text to be displayed on the display of the operator panel. The text definition consists of the text, which will be displayed on the operator pane, and of variables definition, which are in the given text. The entire definition is performed in the T table of the user program. Even at the easiest task it is necessary to work with more than one text, it is, of course, possible to define a number of texts. The type of the text, which will be displayed at the given moment, can de specified by means of one of the control variables, which are set before performing of the instruction **TER**.

From this short introduction, the basic principle being used is apparent. For processing of the task, which is connected with the operator panel, it is necessary to define the appearance of all displays, which will be displayed on the panel and through the program to set the control variable specifying, which text will be displayed. The instruction **TER** fills the memory with the corresponding text, in which the values of the variables were added, which are displayed within the given text. The content of such prepared memory is necessary to be transmitted through the serial channel to the connected operator panel. The operation of the serial channel is not part of the instruction **TER**. The serial channel is necessary to be operated through a user program, according to which serial channel and at which mode it is used for the connection of the operator panel. If the operator presses a key on the operator panel, the code of the pressed key is transmitted by the panel to the PLC and the programmer of the task must ensure saving of the code of the pressed key into one of the control variables, which are passed onto the instruction **TER**. Then the instruction processes the pressed key. In principle it is not necessary to process the codes of the pressed keys by the user program, the instruction TER ensures the operation of the keyboard, when entering or editing the displayed data.

The operand of the instruction **TER** is a number specifying the type of the operator panel, which is served by the instruction. The value 7 is valid for the panel ID-07 and the value 8 for the panel ID-08.

Note

The Mosaic development environment contains the PanelMaker tool, which easily allows defining of the screen of the panels ID-07 and ID-08. This tool automatically generates the program for the PLC, in which the instruction TER is used, and it is done in the way, which corresponds to the following description. At the same time, operation of the serial channel for communication with the panel is generated. This significantly facilitates the use of the operator panels ID-07 and ID-08.

If you use the PanelMaker tool for the operation of the operator panel, proceed according to the manual called The PanelMaker tool TXV 003 25.

If you want to use the instruction **TER** in the program design itself, proceed as follows.

Use of the instruction TER in the user program

The input of the instruction **TER** is a piece of information specifying, from which register starts the zone of the control variables for the instruction **TER** (passed at the A1 layer of the stack) and further from which register the instruction TER should save the generated text (passed at the A0 layer).

The output of the instruction **TER** is filling of the registers R with the text, which is ready to be sent to the operator panel on the display. This memory is called in the next text as *videoRAM*.

For correct functioning of the instruction **TER** <u>**it is necessary**</u> to initialize control variables in the program before calling the instruction **TER.** Either of the processes P63 or P62 treating the PLC restart can be advantageously used.

```
#reg  uint  NumText,              ;control variables for TER
            minText,
            maxText
#reg  usint enableBits,
            sizeDisp,
            keyb,
            inter[24]             ;end of control variables
#def lenDisp 32                   ;size of display used
#reg byte videoRam[lenDisp]       ;TER saves the text here
;
#table byte openingText =  ' First text for ',
                           '      ID-07     '
;
P 0
     LD __indx NumText            ;register number, where
                                  ;control variables for TER are located
     LD __indx videoRam           ;register number, where videoRam starts
     TER   7                      ;preparation of text for panel ID-07
E 0
;
P 63                              ;initialization of control variables
     LD    __indx (openingText)
     WR    NumText                ;what text to be displayed
     LD    lenDisp
     WR    sizeDisp               ;size of display used
E 63
```

Control variables of the instruction TER

The activity of the instruction **TER** is controlled by means of several variables, which must be present in the registers R in a continuous zone (they must be placed behind each other). By writing into these variables and consequential calling of the instruction **TER** it is then possible to change for example the text displayed, etc. The register number, in which the first control variable is located, is passed onto the instruction **TER** on the stack at the A1 layer when the instruction is called. The control variables must be defined in the sequence according to the following table:

Variable	Type	Description
NumText	uint	table number, where the text is defined, which will be filled to the register zone by the instruction **TER**, designated for sending on the panel display
minText	uint	bottom limit for listing in the texts If listing is enabled at control variables *enableBits*, pressing the key *UP ARROW* causes displaying of the previous text (the control variable *NumText* is reduced by 1), listing ends at *NumText = minText*
maxText	uint	top limit for listing in the texts If listing is enabled at control variables *enableBits*, pressing the key *DOWN ARROW* causes displaying of the next text (the control variable *NumText* is increased by 1), listing ends at *NumText = maxText*
enableBits	usint	bits enabling / disabling listing in the texts and editing of variables located in the texts and rolling of menu items, list a message *enableBits.0* 0 - editing disabled, 1 - editing enabled *enableBits.1* 0 - listing disabled, 1 - listing enabled *enableBits.2* 0 - cyclic rolling when editing in the items displayed in the format *dispMenu*, *dispList* and *dispMes* is disabled *enableBits.3* 1 - rolling enabled 0 - the keys *UP ARROW* and *DOWN ARROW* are enabled when editing items in the format *dispMenu* and *dispList* 1 - the keys *UP ARROW* and *DOWN ARROW* are disabled in the above mentioned case, item selection can be done only the key ENTER, selection cancellation can be done only by the key *C*
sizeDisp	usint	display size (number of characters)
keyb	usint	code of the key pressed on the terminal, which should be processed by the instruction **TER**
inter[24]	usint	24 bytes for internal use of the instruction **TER**, write to these bytes is forbidden

Number of text displayed

Number of text displayed corresponds to number of the T table, in which the text is defined (see the following section Definition of the text displayed). The instruction **TER** generates the text for the display according to the table, the number of which is saved at the control variable *NumText*. This variable can be arbitrarily set from the user program.

Listing of the texts

For listing in the texts by means of the keys *UP ARROW* and *DOWN ARROW* the way of definition of the particular texts in the T tables is of importance. To have a possibility of listing in the texts, the texts in which listing will be performed, must be defined in the T tables, numbers of which follow each other and the control variable *enableBits.1* must have the value of log.1. The variable *minText* and *maxText* then specify from which text to which text listing can be performed. During listing the value of the variable *NumText* is automatically changed, this specifies the current number of the text being displayed.

Disabling and enabling of variables editing

By means of the control variable *enableBits.0* it is possible to enable or disable globally the editing of displayed variables from the keyboard of the operator panel. This can be useful for example in such cases, when it is necessary to disable the change of variables that are set from the panel, for some states of the technology being controlled (e.g. in the automatic mode). If the control variable *enableBits.0* is set to log.0, it is possible only to view the variables, but their value cannot be changed.

Disabling and enabling of listing in the texts

Similarly, it is possible to disable or enable listing in the texts globally for all the texts. If the control variable *enableBits.1* has the value of log.0, it is not possible to list in the texts independently on how the control variables *minText* and *maxText* are set. This makes sense in such cases, when the text being displayed on the panel is given by the state of the technology (for example error messages) and it is not desirable that it is possible to initiate displaying of another text from the panel through listing.

Display size

The display size of the operator panel used must be written to the control variable *sizeDisp* before the first call of the instruction **TER**. The written value must not be less than 8 characters.

Processing of pressed keys

The instruction **TER** ensures processing of the keys pressed on the operator panel only in such a case that the codes of the pressed keys are continuously written to the control variable *keyb*. This variable is set to zero by the instruction **TER**. If the keys are to be processed also by the user program not only by the instruction **TER**, it is recommended to create another variable, into which the code of the pressed key is written in the same way as into *keyb* or to process the code of the pressed key before execution of the instruction **TER**. This situation can for example occur when operating function keys *F1*, ..., *F6* by the user program.

Internal variables of the instruction TER

The field of control variables *inter[24]* serves for internal needs of the instruction and the write into these variables is forbidden. The field has the obligatory length of 24 bytes. Only the first variable of this field *inter[0]* can be used by the user program when necessary, since its non-zero value signalizes that editing of a variable on the operator panel is just being performed. This can be useful for example in a situation when one or more operator panels are connected to one PLC TECOMAT, from which the same variables can be changed and the programmer PLC must treat collisions when accessing one variable from more panels at the same time.

Definition of control variables in the program

The control variables can be defined in the program symbolically:

```
#reg  uint  NumText, minText, maxText
#reg  usint enableBits, sizeDisp, keyb, inter[24]
```

For a practically the same result of the next definition, the advantage of it is primarily a simple possibility to create control variables for more operator panels.

```
#struct controlTer          ;structure name
  uint Num_Text,            ;number of text displayed
  uint min_Text,            ;bottom limit for listing in the texts
  uint max_Text,            ;top limit for listing in the texts
  usint enable_Bits,        ;enabling/disabling listing and editing
  usint size_Disp,          ;display size
  usint keyb_,              ;code of the key pressed on the terminal
  usint [24] inter_         ;place for TER internal variables
;
#reg controlTer panel1, panel2, panel3
```

Definition of the structure *controlTer* is part of the file *defter.mos*. In the program we then write the numbers of the displayed texts for the particular panels as follows:

```
     LD    1
     WR    panel1~Num_Text
     LD    2
     WR    panel2~Num_Text
     LD    3
     WR    panel3~Num_Text
```

Definition of text displayed

The particular texts are defined in the T tables. The definition contains text specification, which will be displayed on the terminal display and further optionally a specification or more specifications of the variables, which will be displayed within the text.

Specification of text displayed:

Text specification must have the same number of characters as the display used on the operator panel. For example for a 32-character display the text can be specified as follows:

```
#table byte text1 = 'This is text displayed at ID-08.'
```

The above mentioned definition uses the possibility to specify the content of the tables in the Mosaic development environment by means of so called apostrophe conversion, converting the text specified in apostrophes to ASCII codes, which are saved to the table. Text specification can be written in a better way to respect division of the text on the lines of the display used. The specification for a 32-character 2-line display will be as follows

```
#table byte text2 =
     '1st line of text',
     '2nd.line of text'
```

By analogy for an 80-character 4-line display the text will be specified as follows:

```
#table byte text3 =
     '  1st line of text  ',
```

```
    '  2nd line of text  ',
    '  3rd line of text  ',
    '  4th line of text  ',
```

Specification of displayed variables:

Specification of the variable has a fixed number of items, which describe the displayed variable and contain the information on the display mode. Specification of the variable in the table T is written immediately after the text specification. It contains from the following items:

Item	Type	Description
var	uint	register number, where the displayed variable is located
size	usint	size of the variable being displayed
pos	usint	position of the on the display (position number)
form	usint	display format of the variable
numDig	usint	number of displayed positions
tabLim	uint	table number with message definition, items of the menu or limits

var - register number, where the variable is located - in the texts it is possible to display only the variables located in the registers R, the variables from the areas X, Y, S must be first moved into the registers R and from here their content on the display of the operator panel can be displayed.

size - constant specifying the size and type of the variable being displayed, when the particular bits are displayed, this constant also specifies the bit number, which will be displayed

Variable size	Variable type	Value of constant size	Symbolic name
bit .0	bool	0	*Bit0*
bit .1	bool	1	*Bit1*
bit .2	bool	2	*Bit2*
bit .3	bool	3	*Bit3*
bit .4	bool	4	*Bit4*
bit .5	bool	5	*Bit5*
bit .6	bool	6	*Bit6*
bit .7	bool	7	*Bit7*
byte	byte, usint, sint	8	*sizeByte*
2 bytes	word, uint, int	9	*sizeWord*
4 bytes	dword, udint, dint	10	*sizeLong*
4 bytes	real	11	*sizeFloat*

pos - number of position on the display, from which the variable will be displayed, the positions are numbered from the upper left corner of the display from 0 as follows:

display of 32 characters

```
 0 1 2 3 4 5 6 7 8 9 . . . . . 15
 T E X T   A T   1 s t   l i n e
 * T E X T   A T   2 n d   l i n e
16                             31
```

display of 80 characters

```
      0 1 2 3 4 5 6 7 8 9 . . . . . . . . . 19
    ┌───────────────────────────────────────────┐
    │ 2 0   c h a r a c t e r s   t e x t !     │
 20 │ 2 n d   l i n e   o f   t h e   t e x t   │ 39
 40 │                                           │ 59
    │   l a s t   l i n e   o f   t e x t !     │
    └───────────────────────────────────────────┘
     60                                      79
```

form - constant specifying variable display format
- values 1 to 6 specify the format of variable display, the values in the next table added to these values make the possibilities of display wider

Variable display format	**Value of constant *form***	**Symbolic name**
decadically without sign	1	*dispDec*
decadically with sign	2	*dispSignDec*
hexadecimally	3	*dispHexa*
to each value a message will be assigned, which will be displayed on the display	4	*dispMes*
variable will be displayed as menu with all items	5	*dispMenu*
variable will be displayed as menu with one item	6	*dispList*

Variable display format extension (added to the values of the previous table)	**Value of constant *form***	**Symbolic name**
It is possible to edit the displayed variable from the keyboard of the operator panel, otherwise the variable is displayed only	$10 + *form*	*readWrite*
Justify the displayed variables left	$20 + *form*	*leftJust*
Display of leading zero of variable value	$40 + *form*	*leadZero*

- processing of keys when editing will be ensured by the instruction TER

numDig - constant specifying the number of positions on the display, on which the variables is displayed

Variable display format	**Description of constant *numDig***
dispDec, dispSignDec	lower 4 bits mean the total number of digits and upper 4 bits the number of digits behind the decimal point
dispHexa	number of digits displayed
dispMes	length of one displayed message
dispMenu, dispList	length of one item of the menu

tabLim - table number with definition of additional information for variable display

If this parameter has the value of 0, this means the corresponding table is not declared. When editing variables from the panel keyboard the instruction **TER** will assign default limits corresponding to the maximum range according to the selected size of the variable. For definition of the limits a table of the udint type (the table of type real is also for the display of the variable of type real) containing two numbers is used independently of the size of the variable being displayed - the two numbers are the minimum and maximum value that can be written to the variable when editing from the keyboard of the operator panel.

Variable display format	Description of constant *tabLim*
dispDec, *dispSignDec,* *dispHexa*	table number with definition of minimum and maximum value, which can be written when editing the variable from the keyboard panel
dispMes	table number with message definition
dispMenu, dispList	table number with menu item definition

An example of text definition, in which the content of the register R200 will be displayed:

```
#table usint text1 =       ;text specification
     ' Content of reg.',   ;first line of text
     ' R200 =         ',   ;second line of text
                           ;specification of variable
     200,                  ;location of variable at R (lower byte)
     0,                    ;location of variable at R (higher byte)
     8,                    ;size of variable (byte)
     24,                   ;position on display
     1,                    ;display as a number of 0..255,readOnly
     3,                    ;number of digits displayed
     0,                    ;tabLim (lower byte)
     0                     ;tabLim (higher byte)
```

Constants for variable specification

For more well-arranged text definitions it is useful to introduce the following declarations, which enable to use symbols in the specification of the variable. On a CD supplied with each PLC these declarations can be found on the file *defter.mos*.

```
;------------------------ ;display size definition
#def lenDisp 32            ;32 characters
;
;------------------------ ;structure for variable specification
#struct typeSpecif
  uint var,                ;register number, where the variable is
  usint size,              ;size of variable
  usint pos,               ;cursor position of the display
  usint form,              ;display format
  usint numDig,            ;number of digits displayed
  uint tabLim              ;table number defining limits
;
;-------------------------;structure for texts without variables
#struct textTable0
  uint[lenDisp] text0
;-------------------------;structure for texts with 1 variable
#struct textTable1
  uint[lenDisp] text1,
  typeSpecif Specif11
;-------------------------;structure for texts with 2 variables
#struct textTable2
  uint[lenDisp] text2,
  typeSpecif[2] Specif22
;
;.........................;constants for variable specification
; for Specif~size :
#def Bit0       0    ;display bit number 0
```

```
#def Bit1        1    ;display bit number 1
#def Bit2        2    ;display bit number 2
#def Bit3        3    ;display bit number 3
#def Bit4        4    ;display bit number 4
#def Bit5        5    ;display bit number 5
#def Bit6        6    ;display bit number 6
#def Bit7        7    ;display bit number 7
#def sizeByte    8    ;display 1 byte (type byte / usint / sint)
#def sizeWord    9    ;display 2 bytes(type word / uint / int)
#def sizeLong    10   ;display 4 bytes(type dword / udint / dint)
#def sizeFloat        11   ;display 4 bytes (type real)

;
; for Specif~form :
#def dispDec          1    ;display variable as a decimal number
#def dispSignDec      2    ;display decadically with sign
#def dispHexa         3    ;display hexadecimally
#def dispMes          4    ;instead of value display message
#def dispMenu         5    ;variable will be displayed as menu
#def dispList         6    ;variable will be displayed as menu
;
#def readOnly         0    ;display variable only
#def readWrite        $10  ;enable edition of variable from keyboard
#def leftJust         $20  ;align left
#def leadZero         $40  ;display leading zero
;
; for Specif~numDig
#def des +16*              ;const. for declaration of number of displayed
                           ;digits for numbers with decimal point
;
;write : 5 des 2 => total 5 digits, 2 decimal positions
;generated constant : 5+16*2 = 37 (=$25)
;this results in: lower 4 bits ... total number of digits
;               upper 4 bits  ... number of decimal positions
;!!! ATTENTION !!!
;for correct compilation space in symbolic write are necessary,
;i.e. 5 des 2           ... the only correct write
;5des2, 5 des2, 5des 2 ... incorrect
```

When using the above mentioned declarations, the text definition from the example in the previous chapter will be as follows:

```
#table textTable1 text1 =
     ' Content of reg.',   ;first line of text
     ' R200 =         ',   ;second line of text
     __indx R200,          ;display variable R200
     sizeByte,             ;size of variable (byte)
     24,                   ;position on display
     dispDec,              ;display the content R200
                           ;as a number of 0..255,
                           ;readOnly
     3,                    ;number of digits displayed
     0                     ;limits - default
```

Display range of the variables

The following table specifies the possible formats for the display of the variables and their range. The variables must be located in the register zone R. The variables from the areas X, Y, S or T must be for the purpose of display by the instruction **TER** copied into the registers R.

Size of variable	Display format	Display range
bit	*dispDec*	0,1
	dispSignDec	0,1
	dispHexa	0,1
	dispMes	one message max. 20 characters
	dispMenu	item Menu max. 16 characters
	dispList	item Menu max. 16 characters
usint	*dispDec*	0 to 255
sint	*dispSignDec*	–128 to 127
byte	*dispHexa*	0 to $FF
	dispMes	one message max. 20 characters
	dispMenu	item Menu max. 16 characters
	dispList	item Menu max. 16 characters
uint	*dispDec*	0 to 65 535
int	*dispSignDec*	–32768 to 32 767
word	*dispHexa*	0 to $FF FF
udint	*dispDec*	0 to 4 294 967 295
dint	*dispSignDec*	–2 147 483 648 to 2 147 483 647
dword	*dispHexa*	0 to $FF FF FF FF
real	*dispSignDec*	$\pm 1,175494 \times 10^{-38}$ to $\pm 3,402823 \times 10^{38}$

Display of negative numbers

From the table the number of positions on the display is obvious which is necessary for displaying the variables within the entire range. Displaying in the format *dispSignDec* works with the numbers, the most significant bit of which represents a sign (0 = plus, 1 = minus). The plus sign is not displayed.

Display of decimal numbers

The display formats *dispDec* and *dispSignDec* allow displaying and editing of decimal numbers in the fixed and floating point. When using this type of display it is necessary to include not only the sign into the number of displayed positions, but the decimal point must be displayed, too. It displays the decimal point on the LCD display, to which we are used from common mathematics. The display with decimal point is useful especially for PLC timer preset. The values of type real are displayed in a form without an exponent, this is to say as an ordinary decimal number. Their size is limited to 11 positions including the decimal point and a sign, if any.

Editing of displayed variables

The instruction **TER** allows not only to display the combinations of the texts and variables of various sizes and various output formats, but it also ensures editing (change of the displayed value) of the variable. The conditions for changing the variable from the terminal keyboard are as follows:

- in the specification of the variable being displayed the parameter *readWrite* must be specified
- the control variable *enableBits.0* (the zero bit at the variable *enableBits*) must have the value of 1

If these conditions are fulfilled, the variables displayed can be changed as described in the following chapters.

Editing on the panel ID-07

- Beginning of editing is performed by pressing the key *ENTER,* on the operator panel the first variable starts flashing, which has the parameter *readWrite* in the specification
- The first pressing of any of the keys *LEFT ARROW*, *RIGHT ARROW*, *PLUS* or *MINUS* sets the lowest order of the flashing number for editing, which is signalized by flashing of the particular digit
- By the keys *LEFT ARROW*, *RIGHT ARROW* the edited order is changed
- By the key *PLUS* the unit of the corresponding order is added to the edited value
- By the key *MINUS* the unit of the corresponding order is subtracted from the edited value
- The new value is written (confirmed) to the variable by the key *ENTER*, by the key *C* the editing of the value is cancelled and the original value of the variable will be displayed on the display.

Editing on the panel ID-08

- Beginning of editing is performed by pressing the key *ENTER,* on the operator panel the first variable starts flashing, which the parameter *readWrite* in the specification.
- At this moment you can start editing (if the variable, the number of which we want to change, is flashing) or you can select another displayed variable for editing (if more variables are displayed on one display).
- Selection of another variable that is not flashing is carried out by pressing the key *ENTER* again*,* then the next displayed variable with the defined parameter *readWrite* starts flashing.
- Specification of a new value is performed by the numerical keys, after pressing the first numerical key the original value of the variable disappears and only the last specified digit is flashing.
- The newly specified value is written to the variable after pressing the key *ENTER*, the edited digit stops flashing and if some other variable with the parameter *readWrite* is displayed on the display, it starts flashing and it is possible to enter a new value to this variable.
- If we want to cancel entering the values and return to the original value of the variable the key C must be pressed, editing will be cancelled the variable being displayed or some of its digits stops flashing and the original value will be displayed.
- When entering negative numbers the sign is entered after entering the numerical value before pressing the key *ENTER* (as with the most of calculators).

Correction of the value displayed

- Under correction the change of the displayed value by the units in the lowest displayed order is understood. The procedure of correction does not depend on the type of the panel.
- The correction starts by pressing the key *ENTER,* on the operator panel the first variable starts flashing, which has the parameter readWrite in the specification.

- By the key *PLUS* the displayed value is increased by 1 in the lowest displayed order, by the key *MINUS* the value is decreased by 1, only one digit that corresponds to the order being corrected is flashing.
- The modified value is written (confirmed) by the key *ENTER*, by the key *C* the mode value correction is cancelled and the original value is displayed on the display.

Variable editing in the format dispMes

- Beginning of editing is performed by pressing the key *ENTER*
- The message to be edited starts flashing.
- The change of value (i.e. displayed message) is performed by the keys *LEFT ARROW*, *RIGHT ARROW,* at this moment only the first and the last character of the displayed message flash alternately with the characters *<* and *>*
- During editing the state of the variable in the PLC scratchpad remains unchanged.
- The new value is written (confirmed) to the variable by the key *ENTER*, only at this moment the state of the variable is changed in the PLC scratchpad, characters < > stop flashing
- By the key *C* the editing mode of the message is cancelled and the original message is displayed on the display.

Variable editing in the format dispMenu

- In this case, it is not necessary to start editing, since in the case of the format *dispMenu* with the parameter *readWrite* it is started automatically.
- On the display all defined menu items are displayed.
- The item offered for selection from the menu is marked by characters < >, which are flashing on the first and the last position of the offered item.
- Selection from the offered items is done by the keys *LEFT ARROW*, *RIGHT ARROW.*
- During the selection from the menu the state of the variable in the PLC scratchpad also changes automatically depending on which item of the menu is active at this moment.
- Ending of the selection is performed by pressing the key *ENTER* or *DOWN ARROW*, in this case the instruction TER writes a new text number to the control variable *numText* according to the table defining the menu items **-** the text displayed is changed.
- If we do not want to select any of the offered items, the selection from the menu can be cancelled by the key *C* or *UP ARROW* and in this case the text number specified at the beginning of the table defining the menu items is written to the control variable *numText* and the corresponding text will be displayed.
- The keys *UP ARROW* and *DOWN ARROW* can be disabled during selection from the menu by setting *enableBits.3 = 1.*

Variable editing in the format dispList

- In this case, it is not necessary to start editing, since in the case of the format *dispList* with the parameter *readWrite* it is started automatically.
- On the display only one item menu is displayed, which corresponds to the value of the variable being displayed.
- For the item offered for selection from the menu, characters < > are flashing on the first and the last position.
- Next steps are identical with the selection of an item from the menu, also the way of definition of the menu items is identical with the format dispMenu.

Examples of text definitions

The following examples are created for displaying of the size of 32 characters.

Display of the bit in the text

The text definition, in which the value of the bit R200.1 is displayed. The bit cannot be changed from the keyboard of the operator panel.

```
#table textTable1 text1 =
     ' Content of reg.',   ;first line of text
     ' R200.1 =       ',   ;second line of text
     __indx R200,          ;display variable R200.1
     Bit1,                 ;size of variable (bit č.1)
     26,                   ;position on display
     dispDec + readOnly,   ;display the content R200.1 as a number of
     1,                    ;number of digits displayed
     0                     ;not used
```

The operator *indx* used in the specification of the variable in the item *var* sets the variable address, i.e. the write *indx R200* saves number 200 to the table.

Display on the operator panel (for R200.1=1):

```
 0 1 2 3 4 5 6 7 8 9 . . . . . 15
|  C o n t e n t   o f   r e g . |
|  R 2 0 0 . 1   =   1           |
 16                             31
```

Display of more bits in one text

Text definition, in which the values of the bit *LEFT* and *RIGHT* are displayed. The bits cannot be changed from the keyboard of the operator panel.

```
#reg bool LEFT, RIGHT      ;definition of bit variables
#table textTable2 text2 =
     ' Display of bit ',   ;first line of text
     ' LEFT=x RIGHT=x ',   ;second line of text
                           ;
                           ;specification of 1st variable
     __indx (LEFT),        ;display variable LEFT
     __bitpart (LEFT),     ;size of variable - bit number
     22,                   ;position on display
     dispDec,              ;display LEFT as a number of 0/1
     1,                    ;number of digits displayed
     0,                    ;not used
                           ;
                           ;specification of the second variable
     __indx (RIGHT),       ;display variable RIGHT
     __bitpart (RIGHT),    ;size of variable - bit number
     31,                   ;position on display
     dispDec,              ;display RIGHT as a number of 0/1
     1,                    ;number of digits displayed
     0                     ;not used
```

Please note the type of the table used (*textTable2*), which allows specification of two variables within one text. If we want to display 3 variables, the type of the table used will be *textTable3*, for 4 variables *textTable4*, etc. For the constant *size* the operator *bitpart* was used in this case, this operator computes the bit number from the symbolical declaration variables *LEFT*, *RIGHT*.

Compared to the previous example we can see that the attribute *readOnly* (for reading only) is assigned automatically and it is not necessary to specify it. Instead of characters "x" defined on the second line of the text, the values of the bits *LEFT* and *RIGHT* will be displayed on the display.

Display on the operator panel (for *LEFT* = 1, *RIGHT* = 0):

```
 Display of bit
 LEFT=1  RIGHT=0
```

The above mentioned exampled shows the display of two variables of the bit size. Similarly, also the variables of other sizes can be specified in the text. In one text, so many variables will be displayed, how many of them were specified in the text definition. The variables of various sizes can be combined and each of them can be displayed in a different output format.

Integral display of a variable of byte size

The text definition, in which the value of the variable of byte size is displayed, which has the symbolic name START in the program. The variable can be set from the keyboard of the operator panel.

```
#reg usint START            ;definition of variable START
#table textTable1 text3 =
     ' OBSAH BYTU     ',    ;first line of text
     ' START =        ',    ;second line of text
                            ;specification of variable
     __indx (START),        ;display variable START
     sizeByte,              ;size of variable is byte
     26,                    ;position on display
     dispDec+readWrite,     ;display START as a number of,
                            ;enable edition of variable START
     3,                     ;number of digits displayed
     0                      ;default limits 0...255
```

The above mentioned definition displays the variable *START* as a number in the range of 0 to 255, always 3 positions will be used on the display, leading zero will not be disturbed. The displayed value can be changed from the keyboard terminal. The limits for the specified value are default (for the variable of byte size 0...255), since a reference to the table of limits is not specified (the item *tabLim* in the text definition has the value of 0).

Display on the operator panel (for *START* = 32):

```
 CONTENT OF BYTE
 START =   32
```

Procedure for editing a number

The procedure for entering of a new value to the variable *START* is as follows:

The key *ENTER* (↵) starts editing of the variable *START*.
The entire variable on the display starts flashing.

```
 CONTENT OF BYTE
 START =   32
```

By means of the numerical keys a new value is entered. After pressing the key *1* only that decade is flashing, which will be changed.

```
CONTENT OF BYTE
START =     1
```

After pressing the key *8* the next decade flashes, which will be changed. The digit 1 is entered and shifted to the left.

```
CONTENT OF BYTE
START =    18
```

After pressing the key *5* the next decade flashes, which will be changed. The digits 18 are entered and shifted to the left.

```
CONTENT OF BYTE
START =   185
```

The key *ENTER* (↵) ends editing of the variable *START*. The entire variable on the display stops flashing.

```
CONTENT OF BYTE
START =   185
```

Display of a variable as a number with decimal point

The text definition, in which the value of the variable *LENGTH* is displayed. This variable should be displayed as a number with two decimal points with a possibility of editing from the keyboard of the operator panel. The limits for entering the value from the keyboard of the operator panel are <0.0, 99.99>.

```
#reg uint LENGTH                ;definition of variable LENGTH
#table udint tabLimits = 0, 9999;limits for specific. of variable LENGTH
;
#table textTable1 text4 =
     ' Variable value ',   ;first line of text
     ' LENGTH =       ',   ;second line of text
                           ;specification of variable
     __indx (LENGTH),      ;display variable LENGTH
     sizeWord,             ;size of variable is word
     26,                   ;position on display
     dispDec+readWrite,    ;display decadically, enable editing
     5 des 2,              ;5 positions in total, of which 2 are decimal
     __indx tabLimits      ;table number with limit definition
```

The above mentioned definition displays the variable *LENGTH* as a number in the range of 0 to 655.35, for displaying always 5 positions on the display will be used, the decimal point takes one position. A new value can be entered in the limits of 0 to 99.99. In the definition of the number of the displayed digits, first the total number of digits is displayed including the decimal point and then the number of digits behind the decimal point (*5 des 2*). In the definition of limits the udint or dint type of the table is obligatorily used.

Display on the operator panel (for *LENGTH* = 172):

```
Variable value
LENGTH =   1.72
```

Display of a message

The text definition, in which the variable *ERROR* in the format *dispMes* is displayed, i.e. instead of the variable value *ERROR* the text from the message table will be displayed.

```
#reg usint ERROR      ;definition of variable ERROR
;
#table byte tabErr =        ;definition of message table
 'Machine is OK   ',        ;message displayed at ERROR=0
 'Hydraulics error',        ;message displayed at ERROR=1
 'Motor 1 defect  ',        ;message displayed at ERROR=2
 'Motor 2 defect  ',        ;message displayed at ERROR=3
 'Lubricat. defect',        ;message displayed at ERROR=4
 'Cooling defect  '         ;message displayed at ERROR=5
;
#table textTable1 text5 =
     ' Error message: ',   ;first line of text
     '                ',   ;second line of text
                           ;specification of variable
     __indx (ERROR),       ;display variable ERROR
     sizeByte,             ;size of variable is byte
     16,                   ;position on display
     dispMes,              ;display of ERROR as message
     16,                   ;length of one message
     __indx (tabErr)       ;table number with messages
```

The above mentioned definition displays the variable *ERROR* in such a way that to each value of the variable it assigns one line from the message table. The reference to this table is specified at the end of variable specification (item *tabLim*). In the position of the number of displayed digits, the number of characters of one message is displayed. As the result of this is, that all the messages must be of the same length.

The variables displayed in the format *dispMes* can be of bool or byte size. Another sizes are not permissible.

Display on the operator panel (for *ERROR* = 4):

```
 Error message:
Lubricat. defect
```

In the given example, it is necessary to follow the sequence of declarations. First, the variable must be declared, then the table of messages followed by the table with text definition. The xPRO compiler requires that the object is declared first and after having done this, a reference to this object can be programmed.

The instruction **TER** will work as described only in such a case, if the texts in the table of messages are of the same length. To shorter texts it is necessary to add space characters. The length must correspond to the data that is in the specification of the variable being displayed.

Editing a message

The text definition, in which the variable *COLOUR* in the format *dispMes* is displayed. The variable can be edited from the keyboard of the operator panel in such a way, that after pressing the key *ENTER,* by means of the keys *LEFT ARROW* and *RIGHT ARROW* one of the defined colours is selected. After selection confirmation by the key *ENTER* the code, which corresponds to the chosen variant, is saved to the variable *COLOUR.*

```
#reg usint COLOUR          ;definition of variable COLOUR
;
#table byte tabColour =    ;definition of message table
 ' red            ',       ;message for COLOUR=0
 ' green          ',       ;message for COLOUR=1
 ' light blue     ',       ;message for COLOUR=2
 ' dark blue      ',       ;message for COLOUR=3
 ' azure blue     ',       ;message for COLOUR=4
 ' mottled        '        ;message for COLOUR=5
;
#table textTable1 text6 =
     ' Colour choice: ',   ;first line of text
     '                ',   ;second line of text
                           ;specification of variable
     __indx (COLOUR),      ;display variable COLOUR
     sizeByte,             ;size of variable is byte
     16,                   ;position on display
     dispMes + readWrite,  ;display COLOUR as a message, enable editing
     16,                   ;length of one message
     __indx (tabColour)    ;table number with messages
```

Display on the operator panel (for *COLOUR* = 3):

```
 Colour choice:
 dark blue
```

Procedure for editing a message

The key *ENTER* (↵) starts editing of the variable *COLOUR*. The entire variable on the display starts flashing.

```
 Colour choice:
 dark blue
```

The key *LEFT ARROW* (←) selects the previous item. On the position of the first and the last character of the message characters < > are flashing.

```
 Colour choice:
<light blue    >
```

The key *LEFT ARROW* (←) selects the previous item. On the position of the first and the last character of the message characters < > are flashing.

```
 Colour choice:
<green         >
```

The key *ENTER* (↵) ends editing of the variable *COLOUR*. Characters < > stop flashing, the code 1 is saved to the variable *COLOUR*.

```
 Colour choice:
 green
```

Note: If the control variable *enableBits.2* = 0, then after repeated pressing of the key *RIGHT ARROW* the item „mottled“ is offered as the last item. For selection of the previous items it will be necessary to use the key *LEFT ARROW*. If the control variable *enableBits.2* = 1, after the item „mottled“, the item "red" will follow after pressing the key *RIGHT ARROW*, which means that the particular items when editing the variable *COLOUR* will be rolling cyclically. The same is valid for the variables displayed in the formats *dispMenu* and *dispList*.

Selection of an item in the format dispMenu

The text definition, in which the variable *BATCH* in the format *dispMenu* (menu) is displayed. The variable can be edited from the keyboard of the operator panel by selection one of the variants. Compared to the previous example, when just one message being offered was displayed on the display, all defined menu items are displayed at the same time. The item offered to be selected is marked by characters <>, which are flashing on the first and the last position of the item. During selection, a code is continuously saved in the variable *BATCH*, which corresponds to the currently selected variant. When selecting an item by the key *ENTER* a text will b e automatically displayed, the number of which is specified in the table defining the particular menu items. When defining text numbers, which will be displayed during selection of an item from the menu, it must not be forgotten that the numbers of the texts are entered as word (this is to say 2 bytes, in the memory first the lower byte by significance is saved and then higher byte is saved on the address higher by one).

```
#reg usint BATCH           ;definition of variable BATCH
;
#table textTable0 noChoice =
      ' No variant     ',
      ' was chosen.    '
;
#table textTable0 choiceMini =
      ' MINI variant   ',
      ' was chosen.    '
;
#table textTable0 choiceMidi =
      ' MIDI variant   ',
      ' was chosen.    '
;
#table textTable0 choiceMaxi =
      ' MAXI variant   ',
      ' was chosen.    '
;
#table usint tabBatch =    ;definition of menu table
 __indx (noChoice),        ;text number (low, high) displayed at
 __indx (noChoice/256),    ;pressing the key C
 ' mini ',                 ;menu for BATCH=0
 __indx (choiceMini),      ;text number (low, high) displayed at
 __indx (choiceMini/256),  ;selection of the item mini
 ' midi ',                 ;menu for BATCH=1
 __indx (choiceMidi),      ;text number (low, high) displayed at
 __indx (choiceMidi/256),  ;selection of the item midi
 ' maxi ',                 ;menu for BATCH=2
 __indx (choiceMaxi),      ;text number (low, high) displayed at
 __indx (choiceMaxi/256),  ;selection of the item maxi
;
#table textTable1 text7 =
      ' Batch :        ',  ;first line of text
```

```
'                 ',   ;second line of text
                       ;specification of variable
__indx (BATCH),        ;display variable BATCH
sizeByte,              ;size of variable is byte
10,                    ;position on display
dispMenu+readWrite,    ;display BATCH as menu + editing
6,                     ;length of one item of menu
__indx (tabBatch)      ;table number with menu items
```

The table format with definition of menu items is obligatory and the following scheme must be followed:

- text number, which will be displayed when pressing the key C - 2 bytes !!!!
- text for the first item of the menu
- text number, which will be displayed when selecting the first item from the menu - 2 bytes
- text for second item of the menu
- text number, which will be displayed when selecting the second item from the menu - 2 bytes
- etc. for next items

The texts for the particular menu items must have the same length, which corresponds to the data in the specification of the variable being displayed. Since all the menu items are displayed on the display at the same time, they must fit on the display. Since they are displayed immediately behind each other, it is better, when the texts of the particular items begin and end with the space character. This recommendation takes in account the way of signalizing of the item being offered for selection (in the positions of the first and the last character characters <> are flashing).

Display on the operator panel (for *BATCH*= 1):

```
  Batch :    mini
 <midi>  maxi
```

Procedure for selecting of an item

The key *LEFT ARROW* (←) selects the previous item. On the position of the first and the last character of the selected menu item, characters < > are flashing.

```
  Batch :   >mini>
  midi   maxi
```

The key *RIGHT ARROW* (→) selects the next item. On the position of the first and the last character of the selected menu item, characters < > are flashing.

```
  Batch :    mini
 <midi>  maxi
```

The key *RIGHT ARROW* (→) selects the next item. On the position of the first and the last character of the selected menu item, characters < > are flashing.

```
  Batch :    mini
  midi  <maxi<
```

The key *ENTER* (↵) ends selecting from the menu. In the next step, the text defined for the item maxi is displayed.

```
MAXI variant
was chosen.
```

Display of an item selected from the menu

If we need to display only, which of the items was selected in the previous menu, it is possible to use reference to the table, which defined the menu. If we use the display format *dispMenu* with the parameter *readOnly*, only the selected item from the menu will be displayed on the display, not all the items as in the previous case. Editing of the item from the keyboard of the operator panel is not possible.

```
#table textTable1 text8 =
     ' Chosen batch : ',   ;first line of text
     '               ',    ;second line of text
                           ;specification of variable
     __indx (BATCH),       ;display variable BATCH
     sizeByte,             ;size of variable is byte
     20,                   ;position on display
     dispMenu + readOnly,  ;display selected item from the menu
     6,                    ;length of one item of menu
     __indx (tabBatch)     ;table number with menu items
```

Display on the operator panel (for *DAVKA* = 2):

```
 Chosen batch :
       MAXI
```

Selection of an item in the format dispList*:*

The format *dispList* is used in such cases, when all the menu items do not fit together on the display. When using this format of display only one menu item is displayed. The way of editing of the variable as well as the definition of the menu items is identical with the format *dispMenu*. With the definitions from the previous example, the text definition, in which the variable *BATCH* is displayed by means of the format *dispList* is as follows:

```
#table textTable1 text9 =
     ' Choose batch : ',   ;first line of text
     '               ',    ;second line of text
                           ;specification of variable
     __indx (BATCH),       ;display variable BATCH
     sizeByte,             ;size of variable is byte
     23,                   ;position on display
     dispList+readWrite,   ;display BATCH as menu + editing
     6,                    ;length of one item of menu
     __indx (tabBatch)     ;table number with menu items
```

Display on the operator panel (for *BATCH* = 1):

```
  Choose batch :
       <midi>
```

The key *LEFT ARROW* (←) selects the previous item. On the position of the first and the last character of the selected menu item, characters < > are flashing.

```
  Choose batch :
       >mini>
```

The key *RIGHT ARROW* (→) selects the next item. On the position of the first and the last character of the selected menu item, characters < > are flashing.

```
  Choose batch  :
        <midi>
```

The key *RIGHT ARROW* (→) selects the next item. On the position of the first and the last character of the selected menu item, characters < > are flashing.

```
  Choose batch  :
        <maxi<
```

The key *ENTER* (↵) ends selecting from the menu. As next, the text defined for the item maxi is displayed.

```
MAXI variant
was chosen.
```

Non-text definitions for the instruction TER

The instruction **TER** also allows processing of the tables with text definitions as well as tables containing another information. Above all, the tables with declarations of the values for presetting of control variables and further the table having information on subroutine call as a reaction to the display of the text of a certain number. These tables can be put among the definitions of the particular texts, so the particular actions are called simply by listing the text of a certain number. Further to this, the use of tables with non-text definitions reduces the need of programming in classical instructions of the PLC when realizing a dialog on the operator panel.

Definition of tables for text flow control

The instruction **TER** allows processing of tables, which contain the data for a new setup of the control variables *numText*, *minText*, *maxText* and *enableBits*. These table will be called in further text as the tables for text flow control of the type *textFlow*. They provide the PLC programmer a greater free play when defining text tables, since it is not necessary to follow the request for continuous table numbering defining the texts so strictly, if the possibility of listing in the texts is to be maintained. The tables for text flow control allow switching among the texts, the numbers of which do not follow each other. Further to this they allow setting of new values for the limits of listing and finally, the y allow setting of the control bits for enabling or disabling of editing, for example according to selection made from the menu.

The tables for text flow control must have a structure according to the following table:

Item	Type	Description
_numText	uint	constant, by which the control variable *numText* will be filled
_minText	uint	constant, by which the control variable *minText* will be filled (if 0 is specified, the content of the control variables *minText* remains unchanged)
_maxText	uint	constant, by which the control variable *maxText* will be filled (if 0 is specified, the content of the control variables *maxText* remains unchanged)
_enableBits	usint	constant, by which the control variable *enableBits* will be filled

For an easy declaration of the tables for text flow control it is sensible to specify the declaration of the structure *textFlow*.

```
#struct textFlow
  uint  _numText,
  uint  _minText,
  uint  _maxText,
  usint _enableBits
```

The following example shows the use of the tables *textFlow*. Let us assume that on the operator panel we want to select one of the activities offered - viewing of a group of variables or entering of a new value to the group of other variables. The tables *textFlow* allow filling the control variables for the instruction **TER** without any problem based o the fact that the corresponding variant was selected on the panel. Each of the selected variants has or can have a different setting of the control variables.

```
#reg  uint  numText,        ;control variables for TER
            minText,
            maxText
#reg  usint enableBits,
            sizeDisp,
            keyb,
            inter[24]            ;end of control variables
#def lenDisp 32                  ;size of display used
#reg byte videoRam[lenDisp]      ;TER saves the text here
;
#reg bool  CHOICE                ;variable for realization menu
#reg usint flow, temperature     ;of variable being viewed
#reg uint  heatingTime, numPieces     ;edited variables
;
#table textTable0 noText = ;text for pressing C in the menu
      ' No variant was ',
      ' chosen  !!!    '
;
#table textTable1 browse1 =
      ' Observed flow  ',
      '   xxx  m3/sec  ',
  __indx (flow), sizeByte, 19, dispDec, 3, 0
;
#table textTable1 browse2 =
      'Boiler temperat.',
      '    xxx °C      ',
  __indx (temperature), sizeByte, 19, dispDec, 3 , 0
;
#table textTable1 edition1 =
      'Set heating time',
      '    xx  min     ',
  __indx (heatingTime), sizeWord+readWrite, 20, dispDec, 2, 0
;
#table textTable1 edition2 =
      'Set no.of pieces',
      '   xxxx pcs     ',
  __indx (numPieces), sizeWord+readWrite, 19, dispDec, 4, 0
;
#table textFlow nothingChosen=
  __indx (noText), __indx (noText), __indx (noText), 0
;
#table textFlow chosenEdition =
```

```
  __indx (edition1), __indx (edition1), __indx (edition2), 3
;
#table textFlow chosenBrowse =
  __indx (browse1), __indx (browse1), __indx (browse2), 1
;
#table byte choiceItems=
  __indx (nothingChosen), __indx (nothingChosen /256),
  ' edit      ', __indx (chosenEdition), __indx (chosenEdition/256),
  ' browse    ', __indx (chosenBrowse), __indx (chosenBrowse/256)
;
#table textTable1 text11 =
     'What will you do',   ;first line of text
     '                ',   ;second line of text
                           ;specification of variable
     __indx (CHOICE),      ;display variable CHOICE
     __bitpart (CHOICE),   ;size of variable is bit
     17,                   ;position on display
     dispList+readWrite,   ;display CHOICE as menu + editing
     11,                   ;length of one item of menu
     __indx (choiceItems)  ;table number with menu items
;
P 63
     LD    __indx (text11)
     WR    numText         ;set initial text number
     WR    minText         ;limits for listing
     WR    maxText
     LD    lenDisp
     WR    sizeDisp        ;set display length
     LD    3
     WR    enableBits      ;enable editing and listing
E 63
;
P 0
      :
     LD __indx (numText)   ;register number, where
                                ;control variables for TER are located
     LD __indx (videoRam)       ;register number, where videoRam starts
     TER   7                    ;preparation of text for panel ID-07
      :
E 0
```

After starting the program the field *videoRam* will be filled with the text:

```
 What will you do
   >edit     >
```

By the keys *LEFT ARROW* and *RIGHT ARROW* any of the variants can be selected. After selecting a variant and confirmation of selection by the key *ENTER* the table *textFlow* is processed by the instruction **TER**, the reference of which is specified in the definition of the menu items. By doing this, new values are set to the control variables *numText*, *minText*, *maxText* and *enableBits*. Then the text is displayed according to the new content of the control variables *numText*.

For example, if we select the item *<edit>* then it will be possible to list by the keys *UP ARROW* and *DOWN ARROW* between the texts *edition1* and *edition2* and change the variables displayed on them. If we select the item *<browse>*, it will be possible to list by the keys *UP ARROW* and *DOWN ARROW* between texts *browse1* and *browse2*. The variables displayed in these texts cannot be changed. If we do not select any of the variants being offered, the text *noText* will appear on the display and even this text will not be possible to be edited.

As it can be seen from the previous example, programming of this task is reduced to the definition of the particular texts, initialization of control variables for the instruction **TER** and calling the instruction.

In the end it is useful to note down that the texts with the variables in the formats *dispMenu* and *dispList* can be tree-organized and so complex dialogs can be created. By means of the tables *textFlow* it is then possible to re-change the values of the control variables during the dialogs as needed.

Table definitions for subroutine calls

The next table, which can process the instructions TER, is the table containing the information enabling to call a subroutine after executing the instruction **TER** and pass one parameter onto it. These tables will be called as tables for subroutine calls of the type *makeSubr*. The tables for subroutine calls must have a structure according to the following table:

Item	Type	Description
_numLabel	uint	label number, which will be given back at the layer 0 of the active stack after executing the instruction TER
_parCall	uint	the parameter passed onto the called subroutine at the layer 1 of the active stack
_nextText	uint	constant, by which the control variable *numText* will be filled

For easy declaration of the tables for subroutine calls it is useful to specify the declaration of the structure *makeSubr*.

```
#struct makeSubr
  uint _numLabel,
  uint _parCall,
  uint _nextText
```

During processing of the table for subroutine calls, the instruction **TER** first fills the layer 0 of the active stack by the item *_numLabel*, and then it saves the item *_parCall* to the layer 1 of the active stack and finally it fills the control variable *numText* by the item *_nextText*. Then the text is processed according to this item. If we insert the instruction **CAI** behind calling the instruction TER in the program, this instruction will call a subroutine beginning with the label specified in the table *makeSubr* after the execution of the instruction **TER**. This mechanism allows calling subroutines based on the dialog running on the panel. When processing the tables with texts or tables for text flow control, the layers 0 and 1 of the active stack are set to zero.

The following example shows the use of the tables for subroutine calls when realizing a dialog on the operator panel.

```
#reg uint       numText,   ;control variables for TER
                minText,
                maxText
#reg usint      enableBits,
                sizeDisp,
                keyb,
                [24]            ;end of control variable
#def lenDisp 32                 ;size of display used
#reg byte videoRam[lenDisp]     ;TER saves the text here
;
#reg usint SELECTION            ;variable for realization menu
```

```
#reg uint VALUE
#reg bool auxText               ;auxiliary variables
#reg usint delay
;
#table textTable0 textAction =
     'Required action ',
     ' performed !!   '
;
#label 0,Empty
#label Nothing
#table makeSubr tabNothing = __indx (Nothing), 0, __indx (textAction)
#label Add
#table makeSubr tabAdd = __indx (Add), 0, __indx (textAction)
#label Subtract
#table makeSubr tabSubtract = __indx (Subtract), 0, __indx (textAction)
#label Multiply
#table makeSubr tabMultiply = __indx (Multiply), 0, __indx (textAction)
;
#table byte selectionItems =
  __indx (tabNothing), __indx (tabNothing/256),
  ' Add 5     ', __indx (tabAdd), __indx (tabAdd/256),
  ' Subtract 8', __indx (tabSubtract), __indx (tabSubtract/256),
  ' Multiply 7', __indx (tabMultiply), __indx (tabMultiply/256)
;
#table textTable2 text12 =
     ' value =        ',   ;first line of text
     '                ',   ;second line of text
                           ;specification of variable
     __indx (VALUE),       ;display variable VALUE
     sizeWord,             ;size of variable is word
     9,                    ;position on display
     dispDec,              ;display VALUE
     5,                    ;number of digits
     0,                    ;default limits
;
     __indx (SELECTION),   ;display variable SELECTION
     sizeByte,             ;size of variable is byte
     17,                   ;position on display
     dispList+readWrite,   ;display VYBER as menu + editing
     11,                   ;length of one item of menu
     __indx (selectionItems);table number with menu items
;
P 63
     LD   __indx (text12)
     WR   numText          ;set initial text number
     WR   minText          ;limits for listing
     WR   maxText
     LD   lenDisp
     WR   sizeDisp         ;set display length
     LD   3
     WR   enableBits       ;enable editing and listing
E 63
;
P 0
     LD __indx (numText)   ;register number, where
                                ;control variables for TER are located
     LD __indx (videoRam)       ;register number, where videoRam starts
     TER  7                     ;preparation of text for panel ID-07
```

```
      CAI                       ;call subroutine
;
      LD    auxText             ;temporary text flag
      LD    300                 ;preset 3 sec
      TON   delay               ;time
      WR    %S1.0
      LD    __indx (text12)
      PUT   numText        ;after 3 seconds display menu
E 0
;
P 60
Empty:
      RET
;
Nothing:
      LD    1
      WR    auxText
      RET
;
Add:
      LD    VALUE          ;load VALUE
      ADD   5              ;add
      WR    VALUE          ;write result
      LD    1
      WR    auxText        ;set flag of the temporary text
      RET
;
Subtract:
      LD    VALUE
      SUB   8
      WR    VALUE
      LD    1
      WR    auxText
      RET
;
Multiply:
      LD    VALUE
      MUL   7
      WR    VALUE
      LD    1
      WR    auxText
      RET
E 60
```

After starting the program the field *videoRam* will be filled with the following text:

```
  value =      0
 >add 5    >
```

By the keys *LEFT ARROW* and *RIGHT ARROW* one of the variants offered can be selected. After selecting a variant and confirmation of selection by the key *ENTER* the table *makeSubr* is processed by the instruction **TER**, the reference of which is in the definition of the corresponding menu item. So the subroutine number and a new value of the variable *numText* are set to the layer 0 of the stack. Then the text according to new content of control variables *numText* is displayed. In the end the corresponding subroutine is called by the instruction **CAI**.

For example, if we select the item *<add 5>* then the subroutine *Add* will be called and the text *textAction* will be displayed. The subroutine adds the value of 5 to the variable VALUE being displayed and sets the flag *auxText*, which causes that after 3 seconds the text number *text12* will be written to the control variable *numText.* The text12 will be displayed on the display and the procedure will be repeated in a circle.

Error messages of the instruction TER

The instruction **TER** sets the failure code to the system register S34 when incorrect control variables for the instruction are specified or when error in the specification of the variable occurs, which will be displayed. The following table specifies the failure codes written to the system register S34. If an error is not detected during the execution of the instruction TER, the register S34 is set to zero.

Table13.1 Table of errors reported by the instruction **TER**

S34	Error description
40	display range exceeded (position for the display on the display is greater than the display size at the control variables *sizeDisp*)
41	register range exceeded (register number, in which the displayed variable is saved, the number is specified out of range that can be used by the central unit)
42	display size is incorrect or not specified
43	total number of specified digits is too big (10 digits can be displayed at the most)
44	a large number of digits behind the decimal point (it is possible to display 9 digits behind the decimal point at the most)
45	greater variables than byte specified for format MESSAGE, LIST, MENU (the variables displayed in the specified formats must be of bit or byte size)
46	too long message was entered (the specified message length is inconsistent with message declaration in the corresponding table)

BAS Conversion from binary format to ASCII

Instruction	Input parameters									Result								
	stack									stack								
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
BAS								VAL									ASCII	

VAL - four-digit number in binary format or in BCD (type uint)
ASCII - four ASCII characters

Operands

		uint
BAS	w/o operand	C

Function

BAS - conversion of a number of 16 bit width in the binary format to 4 ASCII characters

Description

The instruction **BAS** processes the lower word of the A0 stack top as a four-digit number, which is converted to 4 ASCII characters. The instruction saves the ASCII characters on the A0 stack top in such a way that the highest digit is saved in the lowest byte of the stack and the lowest digit in the highest byte, which is conversely than in the binary format. This saving allows writing of four ASCII characters at a time by the instruction **WR** RLn to the scratchpad, from where this string will be for example displayed on the display.

Note

The instruction **BAS** works with numbers in the hexadecimal format 0 to F. If we want to display a number decadically, it must be first converted to the BCD format by the instruction **BCD** or **BCL**.

Example

Conversion of a binary number to BCD and to ASCII

```
#reg udint Binar
#reg udint BCDvar
#reg byte ASCII[8]
;
P 0
      LD    Binar
      BCD
      WR    BCDvar
      BAS
      WR    dword ASCII+4    ;4th to 1st digit
      LD    uint BCDvar+2
      BAS
      WR    dword ASCII      ;8th to 5th digit
E 0
```

ASB Conversion from ASCII to binary format

Instruction	Input parameters									Result								
	stack									stack								
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
ASB								ASCII									VAL	

ASCII - four ASCII characters
VAL - four-digit number in binary format or in BCD (type uint)

Operands

		uint
ASB	w/o operand	C

Function

ASB - conversion of a number from ASCII characters to binary format

Description

The instruction **ASB** processes the stack top as four ASCII characters, the highest digit is saved in the lowest byte of the stack top, the lowest digit in the highest byte. The instruction converts characters to the binary number and saves them on the A0 stack top.

Note

The instruction **ASB** works with numbers in the hexadecimal format 0 to F (values $30 - $39, $41 - $46). If we need to process less than four ASCII characters, we do nnot need to fill the not used higher orders with $30, but the value of $00 is permissible, too. Another values are not permissible. If illegal values occur in an ASCII string, the result is 0.

If a decimal number is saved in ASCII characters, we will get a number in BCD format, converted by the instruction **ASB**. If we want to process this number further, it must be first converted to the binary system by the instruction **BIN** or **BIL**.

Examples

Conversion of a decimal number in BCD code in ASCII characters to a binary number

```
#reg uint Binar
#reg byte ASCII[4]          ;4 digits
;
P 0
     LD   dword ASCII
     ASB
     BIN
     WR   Binar
E 0

#reg usint  Binar
#reg byte  ASCII[2]         ;2 digits
;
P 0
     LD   word ASCII
     ASB
     BIN
     WR   Binar
E 0
```

```
#reg uint Binar
#reg byte ASCII[8]          ;8 digits
;
P 0
     LD    dword ASCII      ;1st to 4th digit
     ASB
     SWL                    ;upper word
     LD    dword ASCII+4    ;5th to 8th digit
     ASB                    ;lower word
     OR                     ;combination of two words to uint
     BIN
     WR    Binar
E 0
```

STF	**Conversion of ASCII string to real**
STDF	**Conversion of ASCII string to lreal**

Instruction	**Input parameters**									**Result**								
	stack									stack								
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
STF							REG	LEN		LEN	A7	A6	A5	A4	A3	A2	VAL	
STDF							REG	LEN								VAL		

REG - register index R, in which the first character of the string is saved (type udint)
LEN - length of string of characters (number of filled R registers) (type usint)
VAL - converted numerical value (type real/lreal)

Operands

		real	lreal
STF	w/o operand	C	
STDF	w/o operand		C

Function

STF - conversion of a string of ASCII characters to a value of real type
STDF - conversion of a string of ASCII characters to a value of lreal type

Description

The instruction **STF** expects the R register number at the A1 layer of the stack , in which the ASCII string of the length specified at the A0 stack top begins and converts it to the real type. The stack is shifted one level back and the result is saved at the A0 stack top.

The instruction **STDF** expects the R register number at the A1 layer of the stack, in which the ASCII string of the length specified at the A0 stack top begins, and converts it to the lreal type. The result is saved on the A01 stack top. The other stack layers remain unchanged.

The instruction permits the ASCII string in the formats according to C language convention, for example:

1.15	- number 1.15
–45	- number –45
1.5e3	- number 1.5 x 10^{3}, or 1500
2.48e–4	- number 2.48 x 10^{-4}, or 0,000248

In the C language, each ASCII string obligatorily ends with 0 (value $00, not the ASCII code of the digit 0). If the ASCII string being processed has the length, which exactly corresponds to the parameter LEN, it is not necessary to specify this end zero. But when ASCII strings of various lengths are processed and the parameter LEN specifies only the maximum length, then adding the zero to the end of the string is desirable.

The instructions **STF** and **STDF** accept as the end of the string the value of 0 and the ASCII space character (value $20). Permissible ASCII characters in the string are ‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘9’, ‘+’, ‘–’, ‘.’, ‘e’, ‘E’.

Flags

	.7	.6	.5	.4	.3	.2	.1	.0
S1	-	-	-	-	-	-	-	S

S1.0 (S) - 1 - the result is valid
0 - string error, the result is not valid

Example

Conversion of a string to a number of real type

```
#def LEN 10
#reg byte Zone[LEN]
#reg real VAL
;
P 0
     LD    __indx (Zone)   ;REG
     LD    LEN
     STF
     WR    VAL
E 0
```

FST Conversion of real to ASCII string
DFST Conversion of lreal to ASCII string

Instruction	Input parameters									Result								
	stack									stack								
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
FST						VAL	REG	LEN		REG	LEN	A7	A6	A5	A4	A3	VAL	
DFST					VAL		REG	LEN		REG	LEN	A7	A6	A5	A4	VAL		

VAL - converted numerical value (type real/lreal)
REG - R register index, in which the first character of the string is saved (type udint)
LEN - length of string of characters (number of filled R registers) (type usint)

Operands

		real	lreal
FST	w/o operand	C	
DFST	w/o operand		C

Function

FST - conversion of a value in the real format to the string of ASCII characters
DFST - conversion of a value in the lreal format to the string of ASCII characters

Description

The instructions **FST** and **DFST** expect the R register number at the A1 layer of the stack, in which the field of the length specified at the A0 stack top begins. The instruction **FST** expects the value of real type at the A2 layer, the instruction **DFST** expects at the double-layer A23 the value of lreal type. This value is converted to the string of ASCII characters, which is saved to the R register field, parameters of which specify the A1 and A0 layers. The stack shifts two levels back, which results in returning of the converted value on the stack top and it thus can be used for further processing. So, the instructions allow displaying of various intermediate results, too.

The instruction permits the ASCII string in the formats according to C language convention, for example:

1.15	- number 1.15
–45	- number –45
1.5e+03	- number 1.5×10^{3}, or 1500
2.48e–04	- number 2.48×10^{-4}, or 0.000248

If the resulting string is shorter than specified by the parameter LEN, the ASCII codes of the space character ($20) are added. If such a small string length is specified by the parameter LEN that the resulting number cannot be displayed (the exponent does not fit), the field of registers designated for the write of the string is filled with the ASCII codes of the letter X ($58).

Example

Display of intermediate results

```
#def LEN 16          ;display line length
#reg byte Row1[LEN], Row2[LEN]
#reg real va, vb, vc
;
P 0
     LD    va
     MUF   vb              ;VAL = a.b
     LD    __indx (Row1)   ;REG
     LD    LEN
     FST
     DIF   vc              ;VAL = (a.b)/c
     LD    __indx (Row2)   ;REG
     LD    LEN
     FST
E 0
```

14. SYSTEM INSTRUCTIONS

RDT	**Read time from RTC**
WRT	**Write time into RTC**

Instruction	Input parameters									Result								
	stack									stack								
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
RDT								REG									REG	
WRT								REG									REG	

REG - index of the first register R of the time zone, in which time data is saved (see text bellow) (type udint)

Operands

RDT	w/o operand	C
WRT	w/o operand	C

Function

RDT - read current time directly from the real time circuit (RTC) of the central unit
WRT - write time into the real time circuit (RTC) of the central unit

Description

The instruction **RDT** serves for creating of accurate time markers to a certain event (usually processed in an interrupt process. While in the registers S5 to S12 the time is updated always at the I/O scan and it does not change during the processing of the user program, the instruction **RDT** reads the current time directly from the real time circuit of the central unit and performs synchronization correction (see warning).

The time zone in the scratchpad has the following structure:

Register index	Time data	Range
REG	year	0 - 99
REG+1	month	1 - 12
REG+2	day	1 - 28 / 29 / 30 / 31 (according to month and year)
REG+3	hour	0 - 23
REG+4	minute	0 - 59
REG+5	second	0 - 59
REG+6	day of the week	1 - 7
REG+7, +8	millisecond	0 - 999 (in the sequence of lower byte, upper byte)

All the time values are saved in the binary code.

Warning: The time value read by the instruction **RDT** is synchronized, which means that the time of time write into the RTC either by the instruction **WRT** or communication service through the serial line, the value of milliseconds is reset. In contrast to this, the time in the registers S5 to S12 is continuous, which means that the milliseconds are not reset. This time data is shifted against the time stamp read by the instruction **RDT** by 0 to 999 ms. Therefore, it is not useful to use both time values at the same time.

The instruction **WRT** serves for re-adjustment of the real time circuit of the central unit. This is important especially for changing the daylight saving time to winter time and vice versa, or for time synchronization by an external signal.

The time zone in the scratchpad has the following structure:

Register index	Time data	Range
REG	year	0 - 99
REG+1	month	1 - 12
REG+2	day	1 - 28 / 29 / 30 / 31 (according to month and year)
REG+3	hour	0 - 23
REG+4	minute	0 - 59
REG+5	second	0 - 59
REG+6	day of the week	1 - 7

All the time values are saved in the binary code.

Example

```
#struct time
     usint year, usint month, usint day, usint hours, usint min,
     usint sec, usint dayofweek, uint milisec
#reg cas mark
;
P 0
     :
E 0
;
P 42
     :               ;interrupt from periphery
     LD   __indx (mark)
     RDT             ;write accurate time marker to the variable mark
     :
E 42
```

RDB	**Read from DataBox**
WDB	**Write to DataBox**
IDB	**Identification from DataBox**

Instruction	Input parameters									Result								
	stack									stack								
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
RDB								REG									LEN	
WDB								REG									LEN	
IDB										A6	A5	A4	A3	A2	A1	A0	SIZE	

REG - index of first register R of the parametric zone (see text bellow) (type udint)
LEN - number of bytes being transferred (type udint)
SIZE - size of DataBox in kB (type udint)

Operands

RDB	w/o operand	C
WDB	w/o operand	C
IDB	w/o operand	C

Function

RDB - read data block from DataBox secondary memory
WDB - write data block into the DataBox secondary memory
IDB - identification of DataBox size

Description

The instruction **IDB** is used for identification of the size of DataBox being occupied. This instruction does not require any input parameters. After execution of this instruction, the user stack will be increased by 1 level and the size of DataBox, which is identified in kB will be written on the stack top, for example the value of 256. If the DataBox is not found, the instruction gives back the value of 0.

Before calling the instructions **RDB** and **WDB** it is necessary set several parameters, which are located in the registers R, they must be stored closely behind each other and their sequence must be necessarily observed. The register number, in which the first parameter is located, is passed onto the stack when calling the instructions **RDB** and **WDB** (see text bellow). The parameters have the following sequence:

Parameter name	Type	Description
adrDB	udint	address at DataBox memory
indR	uint	index of the initial register in the scratchpad
len	usint	number of bytes being transferred

The instructions **RDB** and **WDB** do not change the level of the user stack. At the stack top they give back the quantity of really transmitted data. At the same time, they set the content of the of the system register S1.0 to log.1, if the result is valid.

If S1.0 = log.0, no data transmission is performed and simultaneously with this error 14 or 15 is written into the register S34 (source or target data block was defined out of range).

According to the size of the DataBox memory used, the following address spaces are available:

DataBox memory size	Available address space
128 kB (standard CP-7001, CP-7002) 3 MB (optional CP-7002)	0 - $1FFFF 0 - $2FFFFF

When trying to read or write outside the available space, setting of S1.0 = log.0 takes place and at the same time, the relevant failure code is set at S34.

Flags

	.7	.6	.5	.4	.3	.2	.1	.0
S1	-	-	-	-	-	-	-	IS

S1.0 (IS) - 0 - address of source zone in DataBox (**RDB**), or in the scratchpad (**WDB**) or target zone in the scratchpad (**RDB**), or in DataBox (**WDB**), is out of range, carry is not performed

1 - address of source and target zone is within the range of DataBox or scratchpad, carry is performed

S34 = 20 ($14) source data block was defined out of range
S34 = 21 ($15) target data block was defined out of range

Example

```
#struct parDB                ;structure name
     udint adrDB,            ;address at DataBox
     uint indR,              ;index of the initial register in the
                             ;scratchpad
     usint len               ;number of bytes being transferred
;
#reg parDB parusi
#def lenDat 56
#reg usint blockDat[lenDat]
#reg bool  DataBoxOK         ;DataBox flag is in order
;
P 63
     :
     LD   32                 ;required size of DataBox for application
     IDB                     ;identification of DataBox size
     GT
     NEG                     ;DataBox of at least of required size?
     WR   DataBoxOK          ;set flag
     :
E 63
;
P 0
     :
     LD   DataBoxOK          ;DataBox OK ?
     JMC  endDBX             ;no
     LD   $FC00              ;address at DataBox
     WR   parusi~adrDB
     LD   __indx (blockDat);to which reg. data from DataBox are
                             ;transferred
     WR   parusi~indR
     LD   lenDat             ;number of bytes being transferred
     WR   parusi~len
     LD   __indx (parusi) ;register number, where the parameters are
```

```
     RDB                  ;read data block from DataBox to scratchpad
                          ;block of 56 bytes is read from address $FC00
                          ;and saved into the field blokDat
endDBX:
     :
E 0
```

STATM Status of peripheral module

Instruction	Input parameters									Result								
	stack									stack								
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
STATM							NRM	NPOS		NPOS	A7	A6	A5	A4	A3	A2	STAT	

NRM - rack number (type usint)
NPOS - position in the rack (type usint)
STAT - status of peripheral module (type udint)

Operands

STATM	w/o operand	C

Function

STATM - status of peripheral module

Description

The instruction **STATM** reads the status of the peripheral module specified by the rack number at the A1 layer and by the position in the rack at the A0 layer on the stack top. The stack shifts one level ahead. The status has the following structure:

	.15	.14	.13	.12	.11	.10	.9	.8	.7	.6	.5	.4	.3	.2	.1	.0
STAT	-	-	-	-	IE	IR	BT	BE	MM	SC	INI	-	BO	EH	EO	EF

EF - fatal error - active at 1
EO - other error - active at 1
EH - hardware error - active at 1
BO - input status
  0 - released
  1 - blocked
INI - module initialization
  0 - initialization not performed, data cannot be exchanged
  1 - valid initialization
SC - communication status
  0 - module communicates
  1 - module does not communicate
MM - module mode
  0 - HALT
  1 - RUN
BE - blocking of outputs when an error occurs
  0 - block
  1 - do not block
BT - bus control timer
  0 - off
  1 - on
IR - request for interrupt - active at 1
IE - interrupt enable
  0 - disabled
  1 - enabled

CHPAR Parameters of serial channel

Instruction	Input parameters									Result								
	stack									stack								
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
RESM							REG	CHN								REG	CHN	

REG - address of initial register of the zone of parameters of the serial channel in the scratchpad (type udint)
CHN - number of the serial channel (type usint)

Operands

CHPAR	w/o operand	C

Function

CHPAR - loading of parameters of the serial channel

Description

The instruction **CHPAR** loads the setting of the length of 8 bytes of the serial channel specified by the parameter CHN at the stack top to the scratchpad beginning with the register, the address of which contains the parameter REG at the A1 layer. The content of the stack remains unchanged.

The parameter CHN can assume the following values:
1 to 10 - the serial channel CH1 to CH10
209 - line USB
225, 226 - sítě Ethernet ETH1, ETH2

The parameters of the serial channel (CHN = 1 to 10) have the following sequence:

Parameter name	Type	Description
chMod	usint	mode of the serial channel
adr	usint	address of the serial channel
speed	usint	communication speed of the serial channel
timeOut	usint	response timeout in ms
pause	usint	delay time in 100 ms
segm	usint	address segment - expansion of address space
rez7	usint	spare
rez8	usint	spare

The parameter *chMod* has the following structure:

CTS	MT	PAR	CM4	CM3	CM2	CM1	CM0
.7	.6	.5	.4	.3	.2	.1	.0

CTS - detection of signal CTS
0 - do not detect signal CTS
1 - detect signal CTS
MT - token mode
0 - multimaster mode
1 - monomaster mode
PAR - parity mode
0 - even parity
1 - no parity

CM4-CM0 - channel mode

0 - channel off
2 - mode **PC**
3 - mode **PLC**
5 - mode **UNI**
6 - mode **MPC**
7 - mode **MDB**
8 - mode **PFB**
16 - mode **UPD**
17 - mode **DPS**
18 - mode **CAN**
19 - mode **CAS**
25 - mode **EIO**

The parameter *speed*, specifying the communication speed of the serial channel, can assume the following values:

speed	speed [Bd]	speed	speed [Bd]	speed	speed [Bd]
1	50	8	4 800	18	76 800
2	100	10	9 600	19	93 750
3	200	11	14 400	20	115 200
4	300	12	19 200	23	172 800
5	600	13	28 800	24	187 500
6	1 200	14	38 400	26	230 400
7	2 400	16	57 600	29	345 600

The parameters of the USB line (CHN = 209) have the following sequence:

Parameter name	Type	Description
chMod	usint	mode of the serial channel
rez2	usint	spare
rez3	usint	spare
rez4	usint	spare
rez5	usint	spare
rez6	usint	spare
rez7	usint	spare
rez8	usint	spare

The parameters of the Ethernet network (CHN = 225 or 226) have the following sequence:

Parameter name	Type	Description
IPAddr	udint	IP address
IPMask	udint	sub-network mask

Detailed information on serial communications can be found in the manual Serial communication of programmable logic controllers TECOMAT - 32 bit model TXV 004 03.01.

RFRM Refresh data of peripheral module

Instruction	Input parameters									Result								
	stack									stack								
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
RFRM						PAR	RM	POS		PAR	RM	POS						

POS - position of the peripheral module in the rack (type usint)
RM - rack number, where the peripheral module is fitted (type usint)
PAR - parameter of data refresh (1 - only inputs, 2 - only outputs, 3 - inputs and outputs)

Operands

RFRM	w/o operand	C

Function

RFRM - refresh data of peripheral module

Description

The instruction **RFRM** initiates immediate data exchange with the peripheral module, which is given by the rack number (RM) and by the position in the rack (POS). The value of the A2 layer (PAR) then specifies, whether the inputs or outputs or both will be refreshed.

The parameter PAR can assume the following values:
1 - input refresh
2 - output refresh
3 - input and output refresh

If we need to read the current state of the input data from the peripheral module on the position 4 in the rack number 1, we proceed as follows:

```
LD    1          ;PAR - load inputs
LD    1          ;RM  - rack number
LD    4          ;POS - module position in the rack
RFRM             ;read current data
```

The instruction **RFRM** reads the current values of the input data from the peripheral module (their structure is given by the initialization of the peripheral module) and saves them to their images in the scratchpad. Then we can work with the data by means of common instructions working with the scratchpad.

If we need to write the current state of the output data to the peripheral module on the position 5 in the rack number 1, we proceed as follows. We will write required values into the image of the output data in the scratchpad and call the instruction **RFRM**:

```
LD    2          ;PAR - load outputs
LD    1          ;RM  - rack number
LD    5          ;POS - module position in the rack
RFRM             ;write current data
```

The instruction **RFRM** writes the current values of the output data to the peripheral module (their structure is given by the initialization of the peripheral module) from their images in the scratchpad.

If we need to write the current state of the output data and load the current state of the input data from the peripheral module on the position 6 in the rack number 1 at the same time, we proceed as follows. We will write the required values into the image of the output data in the scratchpad and call the instruction **RFRM**:

```
LD   3          ;PAR - load outputs and load inputs
LD   1          ;RM  - rack number
LD   6          ;POS - module position in the rack
RFRM            ;current data exchange
```

The instruction **RFRM** loads the current values of the output data to the peripheral module from their images in the scratchpad and reads the current values of the input data from the peripheral module (the data structure is given by initialization of the peripheral module) and saves them to their images in the scratchpad. Then we can work with the data by means of common instructions working with the scratchpad.

IDTM Load peripheral module identification

Instruction	Input parameters									Result								
	stack									stack								
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
IDTM						REG	RM	POS		REG	RM	POS	A7	A6	A5	A4	A3	

POS - peripheral module position in the rack (usint type)
RM - number of the rack, where peripheral module is placed (usint type)
REG - the first register of array (udint type)

Operands

IDTM	w/o operand	C

Funkce

IDTM - load peripheral module identification

Description

The instruction **IDTM** saves the string containing a peripheral module identification to the array beginning at the register with the address REG. The peripheral module is defined by the rack number (RM) and its position in the rack (POS).

If we need load identification from the peripheral module at the position 4 in the rack 1, we proceed such way:

```
LD    __offset(array) ;REG – array address
LD    1               ;RM  - rack number
LD    4               ;POS – module position in the rack
IDTM                  ;identification loading
```

The length of identification string is always 32 bytes and the form is a string of ASCII characters ended by $00 character (C language syntax). The identification contains the module name, software and hardware version, serial number and name of producer. For example:

```
CP-7002 15H0100 R9 0012 TECO
```

TABM Peripheral module initialization table number

Instruction	Input parameters									Result								
	Stack									Stack								
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
TABM							RM	POS		POS	A7	A6	A5	A4	A3	A2	TAB	

RM - rack number (typ usint)
POS - position in rack (typ usint)
TAB - initialization table number (typ udint)

Operands

TABM	w/o operand	C

Function

TABM - peripheral module initialization table number detection

Description

The instruction **TABM** detects a number of an initialization table of a peripheral module given by the number of the rack at the layer A1 and by the position in the rack at the layer A0 and writes them on the stack top. The stack shifts one level forward. If the module being operated is not on the position specified, the value of $FFFFFFFF is written on the stack top.

CRCM CRC polynomial calculation

Instruction	Input parameters									Result								
	Stack									Stack								
	A7	A6	A5	A4	A3	A2	A1	A0		A7	A6	A5	A4	A3	A2	A1	A0	
CRCM							MES	LEN		A6	A5	A6	A5	A4	A3	A2	CRC	

MES - address of first register of array of protected data (type udint)
LEN - length of array of protected data (type udint)
CRC - calculated CRC character (type uint)

Operands

		uint
CRCM	w/o operand	C

Function

CRCM - CRC polynomial calculation

Description

The instruction **CRCM** calculates a CRC polynomial, sometimes called as CRC16. This polynomial uses for example IBM, DEC and protocol MODBUS RTU. The polynomial has a form

$x^{16} + x^{15} + x^2 + 1$

The property of the CRC value is that, if we calculate the value of the CRC again with the same data including their CRC value placed behind the data block, the result will be 0. Therefore, the CRC polynomial is used to protect transmitted data, when a station transmitting data calculates the CRC through the data being transmitted and a receiving station detect by a control, whether data protection fault does not occur.

The **CRCM** instruction shifts the stack forward and writes the CRC calculated value on its top.

Example

CRC calculation

```
#def length 6
#reg usint message[length+2]
;
P 0
     LD    __offset (message)        ;where message starts
     LD    length                    ;number of characters of message
     CRCM                            ;calculate CRC
     LD    __offset (message) + length
     WRIW                            ;write CRC at the end of message
E 0
```

Control of CRC

```
#def length 6
#reg usint message[length+2]
;
P 0
     LD    __offset (message)    ;where message starts
     LD    length+2              ;number of characters of message include
                                 ;CRC
     CRCM                        ;calculate CRC
     JMD   error
     :                           ;A0 = 0 -> CRC is O.K.
     JMP   end
;
error:
     :                           ;A0 ≠ 0 -> CRC is wrong
end:
E 0
```

15. TRANSFER OF THE USER PROGRAM AMONG VARIOUS MODELS OF INSTRUCTION SETS

At present time, there are two models of the instruction set in the PLC TECOMAT, which differ from each other by the width of the layer of the user stack. In the PLC TECOMAT TC400, TC500, TC600, NS950 the model with the width of the stack layer of 16 bits is used (central units of series A, B, D, E, M, S). In the PLC TECOMAT TC700 the model with the width of the stack layer of 32 bits is implemented (central units of series C).

The user program written for one model can be used into the second model without any changes or with minimum requirements provided certain conditions will have to be met, these conditions are specified in the following paragraphs.

15.1. Operations with variables

The main difference, from which the following principles are derived, is the width of the stack layer. The differences in the behaviour are dependant on the type of the values being processed.

Values of bool (bit) type

The bit values are expanded on all the layer bits on the stack. In both models, the bit instruction behave identically and are thus fully portable. The only exception is the case of the bit constant when we want to write "all ones" on the stack top, thus the value of log.1. If we want to do this by means of constant write, we will use the following instruction in the 16 bit model:

```
LD     $FFFF
```

while in the 32 bits model the instruction

```
LD     $FFFFFFFF
```

But if we realize, what "only ones" mean in another types of variables, then we can write in both models identically

```
LD     -1
```

Values of byte, usint and sint types

Byte, usint and sint types values are saved on the stack on the lowest byte of the layer. In both models instructions working with these types behave identically and thus they are fully portable. The only difference is that they are converted to the type corresponding to the layer width on the stack, i.e. uint (word) for the first case, udint (long) for the second case, and they are processed this way. But when we save the resultant value to the variable of byte, usint and sint types, the lowest byte of the stack layer will be saved in both cases.

Values of word, uint and int types

The values of word, uint and int types are saved on the entire layer on the stack of the width of 16 bits, and on the lower word of the layer on the stack of the width of 32 bits. In both models instructions working with these types behave identically and thus they are fully portable. The only difference is that they are converted to the type corresponding to the layer width on the stack, i.e. they stay of the same type for the first case, or they are converted to udint (long) type for the second case, and they are processed this way. When we save the resultant value to the variable of word, uint and int types, the entire layer will be saved in the first case and the lower word of the stack layer in the second case.

Values of dword, udint and dint (long) types

The values of long type are saved on the stack of the width of 16 bits in two layers, on the stack of the width of 32 bits in one layer. In both models the instructions working with these types behave identically - they keep the sequence of the values saved on the stack. The difference is, in which layer is which value, or its part saved. The portability of the instructions is thus limited in such cases, when we have to shift the stack by means of the instruction **POP**, since with the layer width of 16 bits we shift the stack by two layers, but with the layer width of 32 bits we shift the stack by one layer. We do not avoid conditional compilation in this case.

```
#if _PLCTYPE_ == CP7002
     POP  1
#else
     POP  2
#endif
```

The constructions, which process the value of dword, udint and dint types in parts, cannot be transferred, since while on the stack of the width of 16 bits we can consider the value of these types as two values of uint type in two layers, on the stack of the width of 32 bits this is not possible. The use of such constructions should be avoided and conditional compilation should be used again. A typical example is the use of the variable of uint type in operations of udint type:

```
#reg uint numberw
#reg udint numberl, result

     LD   0                ;conversion of cislow to udint type
     LD   numberw
     ADD  numberl
     WR   result
```

The best solution is to declare the variable *numberw* also as the udint type. But we can also reconcile with the fact that in the 32 bits model one more layer set to zero will rise (redundant instruction LD 0), or we can use conditional compilation.

A second typical example is a conditional jump according to the variable value of udint type:

```
#reg udint number

     LD   number
     ORL
     JMC  jump
```

This procedure can be used only for 16 bit models. The following algorithm can be transferred between both models without any change:

```
#reg udint number

     LD   number
     CML  0
     JZ   jump
```

For operations of 32 bit width with constants and w/o operand special instructions in the 16 bit model can be found, which have been cancelled in the 32 bit model. But the compiler for the 32 bit model accepts the names of these instructions and converts them to the equivalent instructions (see list chapter 15.3.).

Values of real type (float)

For the values of real type the same principles are valid as for the values of dword, udint and dint types.

15.2. Operations on the stack

The difference in processing of no-operand instructions is that both models work with the values of the different width, e.g. the 16 bit or 32 bit. In case of arithmetic operations, this is not usually a problem.

With the logic operations of the word, uint and int types we must not forget during processing on the upper 16 bits in case of the 32 bits model. If we for example write the following algorithm:

```
#reg uint number
#reg bool  flag

     LD    number
     NEG
     WR    flag       ;flag is 0, when negation of number is 0
```

then we find out that while for the 16 bit model this procedure works, for 32 bit model the variable *flag* will always be log.1, since the upper 16 bits after negation always contain the ones.

Here it concerns an incorrect use the conversion between formats, since when saving to the variable of bool type all layer bits are accepted. If we declare the variable *number* as the uint type, then we find out that that procedure does not work in both models.
A corresponding mask must be used.

```
#reg uint number
#reg bool  flag

     LD    number
     NEG
     AND   $FFFF           ;limitation to 16 bits
     WR    flag            ;flag is 0, when negation of number is 0
```

15.3. Reduction of the instructions and their equivalents

Several instructions existing in the 16 bit model have been replaced with their equivalents in the 32 bit model. The compiler for the 32 bit model accepts the names of the cancelled instructions and converts them to the equivalent instructions according to the table 15.1.

The instructions **LMS**, **WMS** and **EOC** have been cancelled without any replacement.

Also the instruction **LDC** # has been cancelled. The construction

```
  LDC      constant
```

can be replaced for both models by the construction

```
LD      constant
NEG
```

or it is more effective to specify a new constant, which is the negation of the original one and to use the instruction **LD**.

The third and most efficient possibility from both calculation and intensity of user program modifications points of view is to change constant definition by means of *#def* to constant declaration by means of *#data.* The construct

```
#def constant  $27
```

can be replaced by the following construct for both models

```
#data byte constant = $27
```

The symbolic name *constant* will be assigned to a byte of the scratchpad D area, which will be filled with the value of $27 at the restart of the user program. All instructions **LD constant** and **LDC constant** shall be compiled as **LD Dn** and **LDC Dn** and it is not necessary to change them. This construct assumes the use of symbolic names.

Table 15.1 List of cancelled instructions in the 16 bit model and their equivalents in the 32 bit model

Original instruction 16 bit model	Equivalent instruction 32 bit model	Function
ADX Z ADX ZW ADX ZL ADL # ADL	ADD Z ADD ZW ADD ZL ADD # ADD	Addition Addition Addition Addition Addition
ANL # ANL	AND # AND	AND with direct operand AND with direct operand
CML # CML	CMP # CMP	Comparison Comparison
LDL #	LD #	Load direct data
MUD ZW MUD # MUD	MUL ZW MUL # MUL	Subtraction Subtraction Subtraction
NGL	NEG	Negation of the stack top
ORL # ORL	OR # OR	OR with direct operand OR with direct operand
SUX Z SUX ZW SUX ZL SUL # SUL	SUB Z SUB ZW SUB ZL SUB # SUB	Subtraction Subtraction Subtraction Subtraction Subtraction
XOL # XOL	XOR # XOR	XOR with direct operand XOR with direct operand

15.4. The instruction SWL does not swap A0 and A1

The instruction **SWL** executes swapping of the upper and lower word of the value of the 32 bit width at the stack top. An additional effect for the 16 bit model is a mutual swap of the content of the A0 and A1 stack layers. But this does not work for the 32 bit model, since the instruction **SWL** works only with the A0 layer.

15.5. Cancelling of cascading of arithmetic operations

The 32 bit model does not support cascading of arithmetic operations, which contained instructions **ADD**, **SUB**, **INR**, **DCR**, **EQ**, **GT**, **LT** working with 16 bit width values. It is necessary to use standard types of variables.

15.6. Formats of subtraction and division

The instruction **MUL** is in the 32 bits model expanded to udint type and fully replaces the instructions **MUL** and **MUD** in the 16 bit model (the compiler of the 32 bits model accepts the name **MUD** and replaces it by the instruction **MUL** - see table 15.1).

In case of division, the situation is more complicated. The instruction **DIV** is in its original form also in the 32 bit model just due to compatibility. Nevertheless we recommend using rather the instruction **DID** for the new algorithms, which is in the 32 bits model expanded to the udint type and the remainder after division is saved to a separate layer. In the algorithms, which are designated only for the 32 bit model we then recommend using of the instructions **DIVL** and **MOD**, which work faster, since we need very rarely the quotient and the remainder at the same time.

15.7. Direct access to peripheral modules

In the 16 bit model the physical addresses marked with the letter U are used for direct access to the peripheral modules. They differ based on the PLC type being used.

In contrast, in the 32 bit model the system instruction **RFRM** has been introduced, which executes immediate data exchange between the scratchpad of the central unit and selected peripheral module (only inputs, only outputs or inputs and outputs). The current data is then accessed by means of the standard instructions **LD**, **WR** and another ones, which work with the scratchpad.

15.8. Higher language support

Central units with the stack width of 32 bits have support of higher languages incorporated (e.g. structured text according to IEC 61131-2 standard). Moreover, the instruction set contains several special instructions, which are not described in this manual, but a higher language is employed. These instructions are not dedicated for normal use by the user. Higher language support required modifications of notation of some constructs of user programs written in individual instructions. All the following modifications can be used also for writing algorithms for central units with the stack width of 16 bits.

Absolute operands

Absolute operands have to start with the character %.

Original notation	Current notation
X1.2	%X1.2
RW20	%RW20
T15	%T15

Use of prefixes

The prefixes *indx*, *bitpart*, *bitcnt*, *offset*, *sizeof* have to be written with two underline characters at the start and the operand has to be parenthesized.

Original notation	Current notation
indx memory	__indx (memory)
bitpart memory	__bitpart (memory)
bitcnt memory	__bitcnt (memory)
offset memory	__offset (memory)
sizeof memory	__sizeof (memory)

The compiler for central units with the stack width of 16 bits accepts both types of notations, but for central units with the stack width of 32 bits, notations in the right column have to be used. We recommend that you always use this type of notation due to code portability.

INSTRUCTION LIST

INSTRUCTION LIST WITH PERMISSIBLE OPERANDS

Operand symbols used:

Z - scratchpad X, Y, S, D, R
T - tables
A - w/o operand (works only on user stack)

n - numerical parameter
- constant
Ln - label with nr. n

Data load and write instructions

Mnemo code	Operand type						Instruction description	Page
	bool	byte usint sint	word uint int	dword udint dint	real	lreal		
LD	Z	Z	Z	Z #	Z #	Z	Load direct data	8
LDQ						#	Load direct data	8
LDC	Z	Z	Z	Z			Load complement data	8
LDIB	A						Indirect data load	11
LDI		A					Indirect data load	11
LDIW			A				Indirect data load	11
LDIL				A	A		Indirect data load	11
LDIQ						A	Indirect data load	11
LEA	Z	Z	Z	Z	Z	Z	Load address	13
WR	Z	Z	Z	Z	Z	Z	Write direct data	14
WRC	Z	Z	Z	Z			Write data complement	14
WRIB	A						Indirect data write	17
WRI		A					Indirect data write	17
WRIW			A				Indirect data write	17
WRIL				A	A		Indirect data write	17
WRIQ						A	Indirect data write	17
WRA		Z	Z	Z			Write direct data with alternation	19
PUT	Z	Z	Z	Z	Z		Conditional data write	21

Logical instructions

Mnemo code	Operand type				Instruction description	Page
	bool	byte usint	word uint	dword udint		
AND	Z	Z	Z	Z # A	AND with direct operand	23
ANC	Z	Z	Z	Z A	AND with negated operand	23
OR	Z	Z	Z	Z # A	OR with direct operand	26
ORC	Z	Z	Z	Z A	OR with negated operand	26
XOR	Z	Z	Z	Z # A	XOR with direct operand	30
XOC	Z	Z	Z	Z A	XOR with negated operand	30
NEG				A	Negation of the top of the user stack	33
SET	Z	Z	Z	Z	Conditional set	34
RES	Z	Z	Z	Z	Conditional reset	34
LET	Z	Z	Z	Z	Pulse from the leading edge	36
BET	Z	Z	Z	Z	Pulse from any edge	36
FLG			A		Logical AND of all bits and transverse functions of A0 bytes in S1	38
STK				A	Transposing of logical values of stack levels to A0	40
ROL n			A		Value rotation to the left n-times	41
ROL				A	Value rotation to the left n-times	41
ROR n			A		Value rotation to the right n-times	41
ROR				A	Value rotation to the right n-times	41
SHL				A	Shift of value to the left n-times	43
SHR				A	Shift of value to the right n-times	43
SWP			A		Swap of first and second A0 byte	44
SWL				A	Swap of the lower and upper word of A0	44

Counters, timers, shift registers, step sequencer

Mnemo code	Operand type word uint	Operand type dword udint	Instruction description	Page
CTU	R	R	Upward counter	44
CTD	R	R	Downward counter	44
CNT	R	R	Bidirectional counter	44
SFL	R	R	Shift register to the left	50
SFR	R	R	Shift register to the right	50
TON	R*		Timer (on delay)	52
TOF	R*		Timer (off delay)	52
RTO	R*		Integrating timer, time meter	56
IMP	R*		Timer - generating of pulse of specified length	59
STE	Z	Z	Step sequencer (stepper)	61

* Any timer can be programmed with an increment unit: .0 - 10 ms; .1 - 100 ms; .2 - 1s; .3 - 10s

Arithmetic instructions

Mnemo code	usint	sint	uint	int	udint	dint	Instruction description	Page
ADD	Z	Z	Z	Z	Z # A	Z # A	Addition	63
SUB	Z	Z	Z	Z	Z # A	Z # A	Subtraction	63
MUL	Z		Z		Z # A		Multiplication	64
MULS		Z		Z		Z # A	Multiplication with sign	64
DIV	Z # A						Division (usint / usint = usint)	65
DID	Z		Z		Z # A		Division with reminder	65
DIVL	Z		Z		Z # A		Division	65
DIVS		Z		Z		Z # A	Division with sign	65
MOD					A		Division reminder	65
MODS						A	Division reminder with sign	65
INR	Z	Z	Z	Z	Z A	Z A	Incrementation (+ 1)	68
DCR	Z	Z	Z	Z	Z A	Z A	Decrementation (– 1)	68
EQ	Z		Z		Z # A		Comparison (equality)	70
LT	Z		Z		Z # A		Comparison (less than)	70
LTS		Z		Z		Z # A	Comparison with sign (less than)	70
GT	Z		Z		Z # A		Comparison (greater than)	70
GTS		Z		Z		Z # A	Comparison with sign (greater than)	70
CMP	Z		Z		Z # A		Comparison	72
CMPS		Z		Z		Z # A	Comparison with sign	72
MAX					A		Maximum	73
MAXS						A	Maximum with sign	73
MIN					A		Minimum	73
MINS						A	Minimum with sign	73
ABSL						A	Absolute value	74
CSGL						A	Sign change	74
EXTB						A	Sign expansion from 8 to 32 bits	74
EXTW						A	Sign expansion from 16 to 32 bits	74
BIN					A		Conversion of number from BCD (8 BCD digits)	75
BIL					A		Conversion of number from BCD (10 BCD digits)	75
BCD					A		Conversion of number to BCD (8 BCD digits)	75
BCL					A		Conversion of number to BCD (10 BCD digits)	75

Operations with user stacks and system stack

Mnemo code	Operand	Instruction description	Page
POP	n	Shift (rotation of the user stack back by n levels	77
NXT		Activation of the next user stack in a row	78
PRV		Activation of the previous user stack in a row	78
CHG	n	Activation of selected user stack (n is 0 to 7)	78
CHGS	n	Activation of selected user stack (n is 0 to 7)	78
LAC	n	Load value from the top of the selected user stack (n is 0 to 7)	79
WAC	n	Write value on the top of selected user stack(n is 0 to 7)	79
PSHB		Saving of 8 bits of the top of the user stack to the stack according to SP	80
PSHW		Saving of 16 bits of the top of the user stack to the stack according to SP	80
PSHL		Saving of 32 bits of the top of the user stack to the stack according to SP	80
PSHQ		Saving of 64 bits of the top of the user stack to the stack according to SP	80
POPB		Filling of 8 bits of the top of the user stack from the stack according to SP	80
POPW		Filling of 16 bits of the top of the user stack from the stack according to SP	80
POPL		Filling of 32 bits of the top of the user stack from the stack according to SP	80
POPQ		Filling of 64 bits of the top of the user stack from the stack according to SP	80

Jump and call instructions

Mnemo code	Operand	Instruction description	Page
JMP	Ln	Unconditional jump	82
JMD	Ln	Jump conditional to non-zero value of result	82
JMC	Ln	Jump conditional to zero value of result	82
JMI	A Ln	Jump indirect	82
JZ	Ln	Jump conditional to non-zero value of equality flag ZR	83
JNZ	Ln	Jump conditional to zero value of equality flag ZR	83
JC	Ln	Jump conditional to non-zero value of carry flag CO	83
JNC	Ln	Jump conditional to zero value of carry flag CO	83
JB	Ln	Jump conditional to non-zero value of flag S0.2	83
JNB	Ln	Jump conditional to zero value of flag S0.2	83
JS	Ln	Jump conditional to non-zero value of flag S1.0	83
JNS	Ln	Jump conditional to zero value of flag S1.0	83
CAL	Ln	Unconditional subroutine call	85
CAD	Ln	Call conditional on non-zero stack top value	85
CAC	Ln	Call conditional on zero stack value	85
CAI	A Ln	Indirect subroutine call	85
RET		Unconditional return from subroutine	86
RED		Return from subroutine conditional to non-zero value of result	86
REC		Return from subroutine conditional to zero value of result	86
L	n	Label n (jump and call target)	87

Operating instructions

Mnemo code	Operand	Instruction description	Page
P	n	Process start	88
E	n	Unconditional process end	88
ED		Process end at non-zero value of result	88
EC		Process end at zero value of result	88
NOP	n	No-operation	90
BP	n	Breakpoint	91
SEQ	Ln	Conditional process interrupt	92

Table instructions

Mnemo code	Operand type bool	byte usint sint	word uint int	dword udint dint	real	Instruction description	Page
LTB	Z T	Z T	Z T	Z T	Z T	Load item from table	93
WTB	Z T	Z T	Z T	Z T	Z T	Write item to the table	96
FTB	Z T	Z T	Z T	Z T	Z T	Find item in the table	99
FTBN	Z T	Z T	Z T	Z T	Z T	Find next item in the table	99
FTM		Z T	Z T	Z T	Z T	Find part of item in the table	102
FTMN		Z T	Z T	Z T	Z T	Find next part of item in the table	102
FTS		Z T	Z T	Z T		Find with sorting according to the table	105
FTSF					Z T	Find with sorting according to the table	105
FTSS		Z T	Z T	Z T		Find with sorting with sign according to the table	105

Block operations

Mnemo code	Operand	Instruction description	Page
SRC	Z T	Source specification for data move	107
MOV	Z T	Move data block	107
MTN	A	Move table to scratchpad	109
MNT	A	Fill table from the scratchpad	109
FIL	Z	Fill the block with constant	111
BCMP	Z	Block comparison	112

Operations with structured tables

Mnemo code	Operand	Instruction description	Page
LDSR	A	Load item from structured table in the scratchpad	113
LDS	A	Load item from structured table T	113
WRSR	A	Write item to structured table in the scratchpad	115
WRS	A	Write item to structured table T	115
FIS	A	Fill item of structured table in the scratchpad	117
FIT	A	Fill item of structured table T	117
FNS	A	Find item of structured table in the scratchpad	119
FNT	A	Find item of structured table T	119

Floating point arithmetic instructions

Mnemo code	Operand type real	lreal	Instruction description	Page
ADF	Z # A		Addition	121
ADDF		A	Addition	121
SUF	Z # A		Subtraction	121
SUDF		A	Subtraction	121
MUF	Z # A		Multiplication	123
MUDF		A	Multiplication	123
DIF	Z # A		Division	123
DIDF		A	Division	123
EQF	Z # A		Comparison (equality)	125
EQDF		A	Comparison (equality)	125
LTF	Z # A		Comparison (less than)	125
LTDF		A	Comparison (less than)	125
GTF	Z # A		Comparison (greater than)	125
GTDF		A	Comparison (greater than)	125
CMF	Z # A		Comparison	125
CMDF		A	Comparison	125
MAXF	A		Maximum	127
MAXD		A	Maximum	127
MINF	A		Minimum	127
MIND		A	Minimum	127
CEI	A		Rounding up	128
CEID		A	Rounding up	128

Floating point arithmetic instructions

Mnemo code	Operand type real	Operand type lreal	Instruction description	Page
FLO	A		Rounding down	128
FLOD		A	Rounding down	128
RND	A		Arithmetical rounding	128
RNDD		A	Arithmetical rounding	128
ABS	A		Absolute value	129
ABSD		A	Absolute value	129
CSG	A		Sign change	129
CSGD		A	Sign change	129
LOG	A		Decimal logarithm	130
LOGD		A	Decimal logarithm	130
LN	A		Natural logarithm	130
LND		A	Natural logarithm	130
EXP	A		Exponential function	130
EXPD		A	Exponential function	130
POW	A		Common power	130
POWD		A	Common power	130
SQR	A		Square root	130
SQRD		A	Square root	130
HYP	A		Hypotenuse	130
HYPD		A	Hypotenuse	130
SIN	A		Sine	132
SIND		A	Sine	132
ASN	A		Arc Sine	132
ASND		A	Arc sinus	132
COS	A		Cosine	132
COSD		A	Cosine	132
ACS	A		Arc cosine	132
ACSD		A	Arc cosine	132
TAN	A		Tangent	132
TAND		A	Tangent	132
ATN	A		Arc tangent	132
ATND		A	Arc tangent	132
UWF	A		Conversion of uint type to real type value	134
IWF	A		Conversion of int type to real type value	134
ULF	A		Conversion of udint type to real type value	134
ILF	A		Conversion of dint type to real type value	134
ULDF		A	Conversion of udint type to lreal type value	135
ILDF		A	Conversion of dint type to lreal type value	135
FDF		A	Conversion of real type to lreal type value	135
UFW	A		Conversion of real type to uint type value	136
IFW	A		Conversion of real type to int type value	136
UFL	A		Conversion of real type to udint type value	136
IFL	A		Conversion of real type to dint type value	136
UDFL		A	Conversion of lreal type to udint type value	137
IDFL		A	Conversion of lreal type to dint type value	137
DFF		A	Conversion of lreal type to real type value	137

PID controller instructions

Mnemo code	Operand	Instruction description	Page
CNV	A	Data processing from analog units	138
PID	A	PID controller	145

Terminal operation instructions and operations with ASCII characters

Mnemo code	Operand type uint	real	lreal	Instruction description	Page
TER				Terminal instruction	155
BAS	A			Conversion from binary format to ASCII	182
ASB	A			Conversion from ASCII to binary format	183
STF		A		Conversion of ASCII string to real	185
STDF			A	Conversion of ASCII string to lreal	185
FST		A		Conversion of real to ASCII string	187
DFST			A	Conversion of lreal to ASCII string	187

System instructions

Mnemo code	Equivalent	Instruction description	Page
RDT	SYS 3	Read time from RTC	189
WRT	SYS 4	Write time into RTC	189
RDB	SYS 5	Read from DataBox	191
WDB	SYS 6	Write to DataBox	191
IDB	SYS 7	Identification from DataBox	191
STATM	SYS 9	Status of peripheral module	193
CHPAR	SYS 11	Parameters of serial channel	194
RFRM	SYS 12	Refresh data of peripheral module	197
IDTM	SYS 13	Load peripheral module identification	199
TABM	SYS 14	Peripheral module initialization table number detection	200
CRCM	SYS 16	CRC polynomial calculation	201

ALPHABETICAL LIST OF THE INSTRUCTIONS

For more information please contact:
Teco a. s. Havlíčkova 260, 280 58 Kolín 4, Czech Republic
tel.: +420 321 737 611, fax: +420 321 737 633, teco@tecomat.cz, www.tecomat.com

TXV 004 01.02
The manufacturer reserves the right of changes to this documentation. The latest edition of this document is available at www.tecomat.cz