



tecomat®

PROGRAMMABLE CONTROLLERS

#program
#unit #reg
byte float
high #table
indx

**PLC TECOMAT
Programmer's Manual**

PLC TECOMAT Programmer's Manual

10th edition - January 2005

Table of contents

1. INTRODUCTION	6
2. PLC AND USER PROGRAM	8
2.1. Activation sequence	9
2.2. PLC operating modes.....	9
2.3. User program restarts.....	11
3. USER PROGRAM STRUCTURE	13
4. INSTRUCTION AND OPERAND STRUCTURE.....	17
4.1. Immediate operand	18
4.1.1. Number system	18
4.1.2. Direct data formats.....	19
4.2. Address operand.....	21
4.2.1. Bool type operand	23
4.2.2. Byte / usint / sint operands.....	24
4.2.3. Word / uint / int operands.....	25
4.2.4. Dword / udint / dint operands	27
4.2.5. Real type operand.....	28
4.2.6. Lreal type operand	30
4.3. Transition destination	32
4.4. Instruction parameter.....	32
5. SCRATCHPAD MEMORY STRUCTURE.....	34
5.1. Input images X.....	35
5.2. Output images Y.....	35
5.3. System registers S	35
5.4. User registers R.....	44
6. DIRECT INPUT/OUTPUT ACCESS	45
6.1. Direct input/output access - 16 bit model	45
6.1.1. Physical addresses in PLC TECOMAT NS950	46
6.1.2. Physical addresses in PLC TECOMAT TC400, TC500, TC600.....	47
6.2. Direct input/output access - 32 bit model	47
7. OTHER ADDRESS SPACES	49
7.1. Data D	49
7.2. Tables T	49
7.3. DataBox additional data memory	51
8. RESULT STACK	56
8.1. Stack structure.....	57
8.2. Data interpretation at the stack - 16 bit model.....	58
8.2.1. Data of bool type - 16 bit model	58

8.2.2. Data of byte / usint / sint type - 16 bit model	59
8.2.3. Data of word / uint / int type - 16 bit model.....	59
8.2.4. Data of dword / uint / dint type - 16 bit model.....	59
8.2.5. Data of real type - 16 bit model	60
8.3. Data interpretation at the stack - 32 bit model.....	60
8.3.1. Data of bool type - 32 bit model	60
8.3.2. Data of byte / usint / sint type - 32 bit model	61
8.3.3. Data of word / uint / int type - 32 bit model.....	61
8.3.4. Data of dword / uint / dint type - 32 bit model.....	62
8.3.5. Data of real type - 32 bit model	62
8.3.6. Data of lreal type- 32 bit model	63
8.4. Switching among stacks	63
9. COMPILER DIRECTIVES	65
9.1. #program	65
9.2. #unit, #module	66
9.3. #include, #usefile.....	67
9.4. #def	67
9.5. #reg, #rem	68
9.6. #struct.....	70
9.7. #data, #table.....	74
9.8. #if, #elif, #else, #endif.....	75
9.9. #ifdef, #ifndef, #else, #endif	76
9.10. #usi.....	77
9.11. #label.....	78
9.12. #macro, #endm	79
9.13. #mnemo, #mnemoend	80
9.14. #useoption.....	81
10. USER PROCESSES	83
10.1. General principles of activation	83
10.2. I/O scan	85
10.3. Restart treatment - processes P62, P63	86
10.4. Loop processes	87
10.4.1. Basic process P0.....	87
10.4.2. Four-phase activated processes P1, P2, P3, P4.....	88
10.4.3. Time-activated processes P5, P6, P7, P8, P9	89
10.4.4. User-activated processes P10 to P40	90
10.4.5. P64 cycle final process.....	91
10.5. Interrupt processes.....	91
10.5.1. Time - activated interrupt P41	93
10.5.2. Input - activated interrupt P42	93
10.5.3. Error - activated interrupt P43	95
10.5.4. HW counter - activated interrupt or incremental encoder activated interrupt P44.....	96
10.5.5. Serial channel - activated CH2 P45	96
10.6. Breakpoint treatment - processes P50 to P57	96
10.7. P60 Subroutine package	97
11. INSTRUCTION SET	98

12. USER INSTRUCTIONS	100
12.1. Application of USI in a user program.....	100
12.2. USI for particular series of central units.....	100
12.3. Creating a user-defined USI	101
12.4. C language compilers used	102
12.5. Example of creation of a user-defined instruction USI	103
12.6. Application example for USI instruction.....	104
12.7. Comments	104
A. APPENDIX	106
A.1. Instruction execution time for central unit CPM-1E TECOMAT NS950.....	106
A.2. Instruction execution time for central unit CPM-1M TECOMAT NS950	108
A.3. Instruction execution time for central unit CPM-2S TECOMAT NS950.....	111
A.4. Instruction execution time for central units CPM-1D TECOMAT NS950 and TECOMAT TC400, TC500, TC600.....	114
A.5. Instruction execution time for central units CPM-1B, CPM-2B TECOMAT NS950	119
A.6. Instruction execution time for central units CP-7001, CP-7002 TECOMAT TC700 and TECOMAT TC650	124

1. INTRODUCTION

The objective of the manual is to give information on the TECOMAT programmable logic controllers (PLCs) and facilitate their programming. In the next text the central units are described by their series (see text), not by PLC types.

Central unit (CPU) series

Every PLC TECOMAT type has several central processor units (CPUs) available that are differentiated by marking them by means of a letter representing their series.

Every series of these CPUs has given their size of memory spaces, the range of the instruction file and operand, stack structure and is the main parameter for the compiler of a user program. The following central units PLC TECOMAT are divided by their properties into the following series:

B series	- NS950 CPM-1B, CPM-2B
C series	- TC650, TC700 CP-7001, CP-7002
D series	- TC400, TC500, TC600, NS950 CPM-1D
E series	- NS950 CPM-1E
M series	- NS950 CPM-1M
S series	- NS950 CPM-1S, CPM-2S

Manual division

- ◆ The 2nd chapter describes general principles for processing a user program in a PLC.
- ◆ The 3rd chapter describes the basic structure of a user program in the Mosaic development environment.
- ◆ In 4th chapter the structure of instructions and their operands are described.
- ◆ The 5th chapter describes the structure of the scratchpad memory including a detailed overview of system services contained in system registers S.
- ◆ The 6th chapter is aimed at the basic principles of the physical addressing of the PLC units.
- ◆ The 7th chapter describes other address spaces for data - D data, T tables and DataBox additional memory.
- ◆ The 8th chapter describes the structure and behaviour of result stackers
- ◆ The 9th chapter contains an overview of the directives that can be used in the Mosaic development environment including samples for their applications.
- ◆ The 10th chapter deals with user processes
- ◆ The 11th chapter contains an overview of instructions and permissible operands.
- ◆ In the 12th chapter user instructions are described

The examples in the manual are in their mnemocode only due to space. To display the examples in a relay line diagram please use the Mosaic development environment.

Related manuals

Detailed information on instructions are contained in the PLC TECOMAT Instruction set manual - 16 bit model (TXV 001 05.02 - for CPU of series B, D, E, M, S) and PLC TECOMAT Instruction set manual - 32 bit model (TXV 004 01.02 - for CPU of series C).

The examples for the solution of various partial problems are contained in the PLC TECOMAT Programming examples manual – 16 bit model (TXV 001 07.02 – for CPUs of

B, D, E, M, S series) and in the PLC TECOMAT Programming examples manual – 32 bit model (TXV 004 04.02 – for CPUs of C series).

PLC programming

The programming of control algorithms and testing of the correctness of created programs for the TECOMAT PLCs is carried out on standard PCs. For connection with the PLC standard serial channel of these PCs is used. Some types of the central units are additionally equipped with Ethernet interface and USB.

With each PLC a CD-ROM with examples and the Mosaic development environment for Windows in the Mosaic Lite version is provided.

The examples of the PLC programs contain instructions for the operation of various PLC units and also the examples from manuals TXV 001 07.02 and TXV 004 04.02.

The Mosaic development environment

The Mosaic development environment is a complex development tool for programming PLC TECOMAT applications and TECOREG controllers that provides a user-friendly application for program creation and debugging. It is a product running under Windows 2000 / XP platforms where a great number of modern technologies are employed. The modular structure of the Mosaic development environment enables the users to create such an environment from its parts that he will need. The following versions are available:

- | | |
|----------------|---|
| Mosaic Lite | non-keyed version with a possibility to program a PLC with two peripheral units |
| Mosaic Compact | enables programming compact PLC TECOMAT of series TC400, TC500, TC600 and TECOREG controllers without limitations |
| Mosaic Profi | designated for all systems of company Teco without limitations |

The basic environment contains the components necessary for creation of a program: text editor xPRO mnemocode compiler, debugger, a module for communication with the PLC, a PLC configuration module and help. A simulator of operation panels ID-07 / ID-08 and built-in TC500 panel are part of the basic environment.

The expansion of the environment is done by means of plug-ins - modules that are initialised in connection with the basic environment, so the Mosaic development environment can be expanded to have a possibility of further programming - structured text according to standard EN 61131 (Mosaic ST plugin), language of ladder diagrams (Mosaic LD plugin - under development) or function blocks (Mosaic FBD plugin - under development) and other support tools for designing operator panel screens (PanelMaker), a tool for working with PID controllers (PIDMaker), graphic on-line analysis of variables being monitored or off-line analysis of archived data (GraphMaker).

2. PLC AND USER PROGRAM

What is a Programmable Logic Controller

A programmable logic controller (further in the text as PLC) is a numerical control electronic system designed to control the processes in industrial environments. It uses a programmable memory for internal saving of user-oriented instructions that are used for the execution of specific functions to control various types of machines or processes through digital or analog inputs and outputs.

The company Teco a. s. manufactures PLC systems under the TECOMAT trademark.

Principles of user program execution

The control algorithm of a programmable logic controller is written as a sequence of instructions in the memory of a user program. The central unit stepwise reads the memory instructions, executes the operations assigned to them with the data from the scratchpad memory and stack or executes transitions in instruction sequences if the instruction is from the group of organizational instructions. When all instructions of the algorithm required are executed, the central unit updates the output variables into output peripheral units and updates the states from the input peripheral units into scratchpad memory. This is continuously repeated and the process is called a "program cycle" (figure 2.1).

Cyclic execution of user program

The one-time state update of input variables during the program cycle avoids the possibilities of hazardous states during solution of control algorithm (the input variables cannot be changed during computation).

Before creating a user program for the PLC it is necessary to realize this. In some cases this can facilitate the solution of the problems, sometimes it makes the situation more complicated.

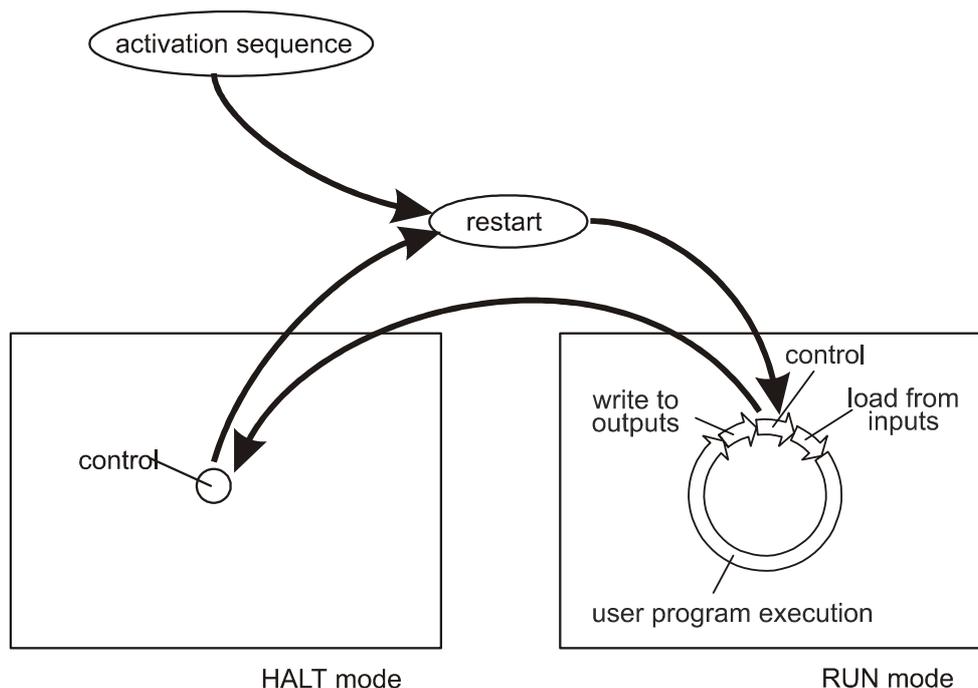


Figure 2.1 Designing of a user program in PLC

2. PLC and user program

On figure 2.1 a simplified layout for a user program in the PLC is shown with the following meanings:

- ◆ activation sequence is the activity of the PLC after switching the supply (see chapter 2.1.)
- ◆ restart is the activity of the PLC immediately before executing the user program (see chapter 2.3.)
- ◆ the RUN and HALT modes represent the operating modes of the PLC (see chapter 2.2.)
- ◆ read out from the inputs represents the transcription of values from the PLC input units into zone X in the scratchpad memory
- ◆ the execution of the user program is done with the values in the scratchpad memory
- ◆ write to outputs represents the transcription of values as calculated by the user program from zone Y into the PLC output units
- ◆ control includes the preparation of the PLC central unit for the solution of next program cycle

The activities into outputs, control and read out from the inputs are together called "I/O scan".

2.1. ACTIVATION SEQUENCE

The activation sequence represents the activity of the PLC immediately after switching power supply. It includes HW as well as SW testing of the PLC and setting the PLC to a defined initial state. In the central units equipped with setting buttons or in the PLC equipped with a keyboard panel (TC500) it is possible to call the setup mode for setting up the parameters after switching on power supply.

After the termination of the activation sequence, restart is carried out, the PLC is switched to the RUN mode and the user program is started. If the PLC diagnostics evaluates a critical error during the activation sequence, the PLC remains in the HALT mode and the error is signalised.

If the setup mode is called and the activation sequence is carried out after its termination, but the PLC then switches to the HALT mode, the user program is not executed, the PLC outputs remain locked and the PLC is expecting the commands from the superior system. The user program can be run either by means of the superior system or by switching the power supply off and on. This can be used in such cases, when there is a program in the PLC that significantly violates its basic functions. The details on behaviour of particular PLC types are given in corresponding manuals.

2.2. PLC OPERATING MODES

The TECOMAT PLC can run in two basic modes. The two modes are called RUN and HALT.

RUN Mode

In the RUN mode, the PLC reads the values of the input signals from the input units, executes the instructions of the user program and writes the calculated values of output signals into the output units. These activities represent a program cycle.

As you can see in figure 2.1, for the PLC the inputs are evaluated discontinuously (a general property that differs digital systems from fully analog ones), the sampling

sequence of the PLC is given mainly by the size and structure of the user program. Based on the capacity of the central unit, the cycle time ranges from milliseconds up to hundreds of milliseconds.

HALT Mode

The HALT mode is mainly used for the activities connected with user program editing. In this mode, neither the user program nor data transmission between the central unit and peripheral units are executed.

PLC behaviour during a critical error

An exception from the above mentioned is, if a critical error occurs in the PLC that inhibits control activities. In this case, a mechanism for critical error treatment is started and treats the critical error from the point of control security and **always** switches the PLC to the HALT mode.

Operating mode change

The change of the PLC modes can be done through a superior system (master computer) that is connected to the communication channel supporting system services (serial channel, Ethernet or USB) or by means of service inputs. Typically, a standard PC represents this superior system serving as a programming means, monitoring or visualization workplace for the operation of the object being controlled.

When changing the PLC operating modes, some activities are executed standardly and some are optional.

It is possible to say in general that the change of the PLC operating mode is an activity requiring higher concentration of the operators, since in many cases it very significantly influences the state of the object being controlled. The change from the RUN to the HALT mode can be an example when the PLC stops executing of the user program and the object connected is not controlled. Therefore we recommend reading the following text very carefully.

When the change of the PLC mode is done through the Mosaic development environment, the optional activities for the mode change are part of the Project manager in folder *Environment / Control PLC*.

Switching from HALT to RUN

When switching from the HALT to the RUN mode the following activities are executed:

- ◆ user program integrity test
- ◆ software configuration test for the peripheral units stated in the user program
- ◆ user program start

The following items are optional:

- ◆ PLC error reset
- ◆ warm or cold restart
- ◆ output locking during user program execution

Switching from RUN to HALT

When switching from the RUN the HALT mode the following activities are executed:

- ◆ the of the user program stops
- ◆ locking (disconnection) of PLC outputs

The following items are optional:

- ◆ PLC error reset
- ◆ PLC output reset

If during the change from one operating mode to the other one a critical error occurs, the PLC sets the HALT mode, indicates the error and waits for clearing the error cause.

Warning: Control stop through the HALT mode is designated only for PLC program debugging purposes. This feature does not replace the CENTRAL STOP feature in any way. The CENTRAL STOP circuits must be connected in such a way so that their function is independent on PLC work!

Details on behaviour and possibilities of particular PLC types are given in corresponding manuals.

2.3. USER PROGRAM RESTARTS

Restart is an activity of the PLC, the task of which is to prepare the PLC to execute the user program. Under normal conditions restart is executed after a successful termination of the activation sequence and at every change of user program.

The TECOMAT PLCs differ two types of restart, warm and cold. The warm restart enables to holdback the values in the registers also when power supply is off (remanent zone). The cold restart always performs full memory initialisation.

Activities during restart

During restart, the following activities are performed:

- ◆ user program integrity test
- ◆ reset of the entire PLC scratchpad
- ◆ remanent zone reset (cold restart only)
- ◆ setting of backed-up registers (warm restart only)
- ◆ initialisation of system registers S
- ◆ initialisation and check of PLC peripheral system

User program startup without restart

The user program is also possible to be run without restarting, in this case only the user program integrity test and PLC peripheral system check are performed (not initialisation).

User processes during restart

Dependent on the type of restart being executed functions also the scheduler of user processes P. If warm restart is executed within HALT → RUN switching, user process P62 is executed as the first one after switching into the RUN mode (if it is programmed). In case of cold restart, user process P62 is executed as the first one. If no restart during switching into the RUN mode is executed, process P0 is executed as the first one after switching.

Program change during PLC run

The Mosaic development environment also enables a program change when the PLC is running. In this case it is important to realize that for the period of loading of a new

program, the execution of the program is topped without locking the outputs. This can also take a couple of seconds!

Restart type preset

The type of restart after starting a user program from the Mosaic development environment can be set in the Manager project in folder *Environment / Control PLC*. For the type of restart after switching the power supply on for the PLC (after a successful activation sequence) is given the option in folder *Sw / Cpm*.

This setting will automatically be transferred during compilation into the user program by means of directive *#useoption*.

3. USER PROGRAM STRUCTURE

Programming language of xPRO compiler

The programming language of the xPRO compiler within the Mosaic development environment is based on the mnemonic language of the PLC as described in chapter 4. The extension is given by a possibility to use symbolic names of the variables (registers, labels, etc.) and directives for the compiler. The xPRO compiler can be incorporated with compilers of higher language, e.g. according to IEC 61131-3.

Rules for program write

The user program write is governed by several simple rules:

- ◆ Each line of the source text of the program can have one instruction at most. This means the line can also be empty or can contain just a comment. Attention - a label is also an instruction!
- ◆ Symbolic names can begin with letters 'a' to 'z', 'A' to 'Z' or '_', and can also contain characters 'a' to 'ž', 'A' to 'Ž', '0' to '9' and '_' (underline character). A symbolic name must not begin neither with a digit nor a character with a diacritical mark (wedges and acute accents).

Attention! A symbolic name must not be identical with any of the reserved symbols of the compiler (see Table 3.1).

- ◆ Comments start after the semi-colon character ';'. The entire text beyond this character is considered as a comment and ignored.
- ◆ Lower-case and upper-case letters can be used arbitrarily, the compiler internally converts all letters to upper-case ones, thus it is not case sensitive.

When using these rules it is possible to write the following simple program:

Example 3.1

```
#def  StartStop  %X0.0      ;declaration of inputs, outputs and constants
#define Output   %Y0.0
#define Value     21
#reg  bool  Reset          ;declaration of registers
#reg  uint  Timer, Counter
;
P 0                                ;program start
    LD    StartStop        ;timer control bit
    LD    Reset            ;setting timer to zero
    LD    Value            ;timer preset
    RTO   Timer.3         ;second timer function
    WR    Output           ;output
    JMC   Jump            ;conditional jump
    INR   Counter         ;number of cycles, when Output is = 0
Jump:                                ;label
E 0                                ;program end
```

The program as per example 3.1 represents the basic components of the user program in the Mosaic development environment.

The Mosaic development environment automatically creates the program header in the control file *xxx.mak*, where *xxx* is the name of the project within the project group. In the

Mosaic development environment the *#program* directive is not written into the source file, as opposed to the xPRO environment where the directive is used.

After the program header is created, declarations of inputs, outputs, constants and registers follow. If we do not use these declarations, the user program then can be written by means of absolute operands (i.e. X, Y, S, D, R registers). But we do not recommend fundamentally to do this since this way is very confused and makes prospective program modifications very difficult. In addition, some program designs using structures and symbolic references are almost impracticable when using absolute addresses.

Problems also can occur when transferring the program algorithm in absolute representation among particular types of central units.

The central units with the 32 bit stack width supporting a higher language require so called percent convention when writing absolute operands, this is to say that the % character has to be written before an absolute operand character (see example 3.1). All prefixes (*indx*, *bitpart*, *bitcnt*, *offset*, *sizeof*) are written with two underline characters (*__indx*, *__bitpart*, *__bitcnt*, *__offset*, *__sizeof*) and the following object is parenthesised. These measures are necessary to eliminate collision with the symbolic names of a higher language. In case of the central units with the 16 bit stack width, the use of the percent character and underline characters before an absolute operand is not obligatory, but we recommend to use them due to code transferability.

Software configuration is the description of the peripheral units used by means of the *#unit* directive, or *#module* (TC650, TC700) respectively. These directive determine the interconnection of the inputs and outputs with the scratchpad memory and enable the data transfer between the user program and environment. The Mosaic development environment generates a list of the *#unit* / *#module* directives, related initialisation tables and necessary declarations automatically based on filled-in tables in the configuration section (Project manager, folder *Hw* | *HW Configuration*) in an individual file *xxx.hwc*, where *xxx* is the name of the project.

If we check the option *Suppress directive #UNIT* in the configuration section, then the PLC will execute the user program only in the scratchpad memory. The inputs will not be downloaded into the scratchpad memory and there will be no write into the outputs in the scratchpad memory. The outputs will remain locked. This state can be useful when debugging an algorithm in the PLC without connected technology.

After this the user program follows. Since the Mosaic development environment contains the xPRO compiler, the user program instruction write in this environment is identical to the recent xPRO development environment. Each user program must have the P0 process even if it should remain empty. The P0 and E0 instructions are obligatory ones.

Using comments is not obligatory, but the rule is, the better the program is commented with more details, the easier to do some modifications later on is.

The indentation of the instruction by means of tabulators, as you can see from the example, is not necessary, it is enough to use the space character to separate the operand and all instructions could be written from the line margin but we try to do our best to have the program as well-ordered as possible.

In table 3.1 there is a list of reserved compiler symbols that cannot be used as symbolic names since they are used by the compiler to mark preset object. It is necessary to pay attention to this list since the mistakes caused by using the reserved symbols for another purpose could result in an unpredictable behaviour.

3. User program structure

Table 3.1 List of reserved compiler symbols that cannot be used as a symbolic name
(valid for upper-case and lower-case letters)

A*	CHGS	DIGITOUT8	FTSS	LDQ	OPTION
ABS	CHPAR	DIGIT_050	G*	LDS	OR
ABSD	CMDF	DIGIT_200	GS*	LDSR	ORC
ABSL	CMF	DIGIT_300	GT	LDU	ORL
ACS	CML	DIGIT_400	GTDF	LDX	P
ACSD	CMP	DIGIT_500	GTF	LDY	PERIOD_500
ADD	CMPS	DIGIT_600	GTS	LEA	PERIOD_600
ADDF	CNT	DIGIT_633	H*	LEAX	PID
ADF	CNV	DIGIT_63X	HIGH	LEAY	PIDA
ADL	COLD	DIGOUT16	HPD	LET	PLC
ADX	COS	DIGOUT8	HPE	LETX	POP
ALIGNED	COSD	DIG_10IN_10OUT	HS*	LINK	POPB
ANALOG_050	COUNT_500	DIG_5IN_6OUT	HYP	LINT	POPL
ANALOG_200	COUNT_600	DINT	HYPD	LMS	POPQ
ANALOG_300	CP7001	DISPASCII	IC_04	LN	POPW
ANALOG_400	CP7002	DISPHEX	IDB	LND	POW
ANALOG_500	CPM1A	DIV	IDFL	LOG	POWD
ANALOG_600	CPM1B	DIVL	IDFW	LOGD	PROGRAM
ANC	CPM2B	DIVS	IF	LONG	PRV
AND	CPM1D	DL0, DL1...*	IFDEF	LOW	PSHB
ANL	CPM1E	DQ0, DQ1...*	IFL	LREAL	PSHL
AN_4IN	CPM1M	DS*	IFNDEF	LT	PSHQ
AN_4IN_4OUT	CPM1S	DST	IFW	LTB	PSHW
AN_8IN	CPM2S	DT	ILDF	LTDF	PUBLIC
AN_8IN_4OUT	CS*	DW0, DW1...*	ILF	LTF	PUT
AS*	CSG	DWORD	IMP	LTS	PUTX
ASB	CSGD	E	INCLUDE	LWORD	R*
ASN	CSGL	EC	INDX**	M*	R0, R1...*
ASND	CTD	ED	INR	M0, M1...*	RCHK
ATN	CTU	EOC	INT	MACRO	RD0, RD1...*
ATND	D*	EQ	INTIN_500	MAX	RDB
B*	D0, D1...*	EQDF	INTIN_600	MAXD	RDT
BAS	DATA	EQF	IRC_500	MAXF	REAL
BCD	DATE	ENDIF	IRC_600	MAXS	REC
BCL	DCR	ENDM	IWDF	MD0, MD1...*	RED
BCMP	DD0, DD1...*	ES*	IWF	MF0, MF1...*	REG
BET	DEF	ETH1, ETH2...	JB	MIN	REI
BETX	DF0, DF1...*	EXP	JC	MIND	RES
BIL	DFF	EXPD	JMC	MINF	RESM
BIN	DFST	EXTB	JMD	MINS	RESX
BIT	DID	EXTW	JMI	ML0, ML1...*	RET
BITCNT**	DIDF	F*	JMP	MNT	RF0, RF1...*
BITPART**	DIF	FDf	JNB	MOD	RFRM
BOOL	DIFCNT100MS	FIL	JNC	MODS	RL0, RL1...*
BOX	DIG2	FIS	JNS	MOV	RND
BP	DIG4	FIT	JNZ	MQ0, MQ1...*	RNDD
BRC	DIG8	FLG	JS	MTN	ROL
BRD	DIGIN16	FLO	JZ	MUD	ROR
BRE	DIGIN8	FLOAT	L	MUDF	RQ0, RQ1...*
BS*	DIGIN8OUT8	FLOD	L0, L1...*	MUF	RTO
BYTE	DIGIT2	FNS	LABEL	MUL	RW0, RW1...*
C*	DIGIT4	FNT	LAC	MULS	S*
CAC	DIGIT8	FS*	LD	MW0, MW1...*	S0, S1...*
CAD	DIGITIN16	FST	LDC	NEG	SCH2
CAI	DIGITIN32	FTB	LDI	NGL	SCMP
CAL	DIGITIN64	FTBN	LDIB	NOP	SCNV
CEI	DIGITIN8	FTM	LDIL	NXT	SCON
CEID	DIGITOUT16	FTMN	LDIQ	OFF	SD0, SD1...*
CH1, CH2...	DIGITOUT32	FTS	LDIW	OFFSET**	SDEL
CHG	DIGITOUT64	FTSF	LDL	ON	SEQ

Table 3.1 List of reserved compiler symbols that cannot be used as a symbolic name (valid for upper-case and lower-case letters) - continued

SET	STATM	TER	UNLK	WRX	_ANALOG_
SETX	STDF	TF0, TF1...*	USB1, USB2...	WRY	_CHX
SF0, SF1...*	STE	TIME	USEOPTION	WSTRING	_GT_40_
SFL	STF	TL0, TL1...*	USI	WTB	_GT_40A_
SFND	STK	TOD	USINT	X*	_IC_12_
SFR	STRING	TOF	UWDF	X0, X1...*	_IC_13_
SHL	STRUCT	TON	UWF	XD0, XD1...*	_IM_61_
SHR	SUB	TQ0, TQ1...*	UW0,UW1...	XF0, XF1...*	_INTELG
SIN	SUDF	TR050	UX_52	XH_04	_INTELLIGENT
SIND	SUF	TR200	VIRTMUX	XL0, XL1...*	_IR_11_
SINS	SUL	TR300	WAC	XOC	_IT_04_
SINT	SUX	TW0, TW1...*	WARM	XOL	_IT_06_
SIZEOF**	SW0, SW1...*	U*	WDB	XOR	_IT_12_
SL0, SL1...*	SWL	U0, U1...*	WMS	XQ0, XQ1...*	_IT_15_
SLEN	SWP	UD0, UD1...*	WORD	XW0, XW1...*	_KEYDISP_200_
SLFT	SYNC	UDFL	WR	X_OFF	_KEYDISP_500_
SMID	SYS	UDFW	WRA	X_ON	_OT_04_
SND	T*	UDINT	WRC	Y*	_OT_04X_
SPEC	T0, T1...*	UF0, UF1...*	WRI	Y_OFF	_OT_05_
SPECIAL	TABLE	UFL	WRIB	Y_ON	_OT_05X
SQ0, SQ1...*	TAN	UFW	WRIL	Y0, Y1...*	_PLCTYPE_
SQR	TAND	UINT	WRIQ	YD0, YD1...*	_SC_11_
SQRD	TC400	UL0, UL1...*	WRIW	YF0, YF1...*	_SPECIALTAB_
SRC	TC500	ULDF	WRS	YL0, YL1...*	_SPECTAB_
SRD	TC600	ULF	WRSR	YQ0, YQ1...*	
SREP	TC700	ULINT	WRT	YW0, YW1...*	
SRGT	TD0, TD1...*	UNIT	WRU	_ANAL_	

* Symbols marked * are in the 32 bit stack width central units obligatorily used with the % character and then it is possible to use them as a symbolic name. For example:

```
%SW12                ;absolute marking of register SW12
#def SW12 %X0        ;symbolic marking of input
```

** Symbols marked ** are in the 32 bit stack width central units obligatorily used with the __ characters and then it is possible to use them as a symbolic name. For example:

```
__indx (item)        ;use of prefix __indx
#def indx 20         ;symbolic marking of constant
```

Note: Due to continuous development and enhancement of the xPRO compiler we do not recommend using one to four-letter symbolic names (mainly originating from English names or abbreviations) separately or in combination with a numeric index. When really necessary to use such a name we recommend using the underline character before it (_A as an example).

4. INSTRUCTION AND OPERAND STRUCTURE

Instruction

An instruction is the smallest element of the user program. It consists of a mnemonic code and an operand. Formally, we distinguish between operand-free instructions from instructions with one operand.

Mnemonic code

A mnemonic code is a group of one or three letters having a meaning of an abbreviation usually derived from the English name of the instructions (for example AND, OR, XOR, NEG, FLG, RET, ED, EC).

Operand-free instruction

The operand-free instruction usually processes the content of the stack top, or also other stack layers of the stack pertinently or executes an unequivocally specified activity (for example return from subroutine). The operand-free instruction consists of the mnemonic code only and its activity is not necessary to specify in more detail.

Instruction with one operand

The mnemonic code of the operand instruction is followed with a group of characters that specify one of the operands as agreed and with this operand the instruction works or which specify the behaviour of the instruction (for example instruction parameter, number of repetitions of the basic operation, jump position, etc.). The second operand for logical and arithmetic operations is the stack top. The operand is separated from the mnemonic code at least by one space character.

Absolute addresses are written beginning with the % character, which is not obligatory when programming 16 bit stack width central units, but we recommend to use them with regard to the transferability of the user programs to the central units with the 32 bit stack width.

The operands instructions can have the following structure, for example:

```
AND  %X0
AND  %Y2
OR   %RW4
TON  %RW16.1
JMP  %L15
P    0
NOP  17
LTB  %T5
POP  3
LD   123
LD   %10110110
```

Sort of operands

According to the meaning we can specify four types of operands:

immediate operand - the instruction operand is directly a numeric value (written in the numeric system selected), with which the required instruction is executed
address operand - determines the address of the position, from where the operation result is read or where it is saved

transition destination - the operand is the number of the label (a labelled position in the program), where the programmed transition (a jump or a call) is directed
 instruction parameter - a single numeric or letter parameter marking the instruction in question (for example process number, label number, null instruction number, stack) or specifies its behaviour (for example number of stack shifts).

From the format point of view there are no other instructions. If some instructions or the subroutine require more parameters (function blocks, table or block instructions), they are passed in several stack levels where the sequences of instructions LD or LDC are saved.

Operand types

According to data width several operand types are distinguished. These types are marked in two ways - one way is marked as TECOMAT which is the marking used with previous compiler versions, the second way of marking corresponds to IEC 61131. Both ways of identification can be normally used. An overview is given in Table 4.1.

Tab.4.1 Operand types

TECOMAT	IEC 61131	Width	Numeric range
bit	bool	1 bit	Boolean information 0 - 1
byte	byte	8 bits	0 to 255
	usint	8 bits	0 to 255
	sint	8 bits	-128 to +127
word	word	16 bits	0 to 65 535
	uint	16 bits	0 to 65 535
	int	16 bits	-32 768 to +32 767
long	dword	32 bits	0 to 4 294 967 295
	udint	32 bits	0 to 4 294 967 295
	dint	32 bits	-2 147 483 648 to +2 147 483 647
float	real	32 bits	$\pm 1,175494 \times 10^{-38}$ to $\pm 3,402823 \times 10^{38}$
double	lreal	64 bits	$\pm 2,2 \times 10^{-308}$ to $\pm 1,8 \times 10^{308}$

Types float and double, or real and lreal as the case may be contain numeric formats with floating point according to IEEE-754.

In the following text we will be using the types according to IEC 61131. The information is valid also for equivalent types of TECOMAT.

4.1. IMMEDIATE OPERAND

In this case a number is written as the operand of the instruction that is directly processed by the instruction - data carried directly in the instruction. The immediate operand thus has the meaning of a numeric constant.

4.1.1. Number systems

Constant entry in common number systems

The numeric constant can be entered in any numeric system in the following form:

#n#cccc

where n stands for the base of the numeric system,

4. Instruction and operand structure

cccc is the number itself in the numeric system selected

If the base of the numeric system is greater than 10, then the particular digits can be written decimally (in the form of a two-digit number) and are separated with a dot.

```
LD    #8#360                ;octal number system
WR    #60#15.28.35          ;write time data in hours,
                               ;minutes and seconds by means of
                               ;sexagesimal system
```

Shortened entry in most frequently used number systems

The most frequently numeric systems used are the decimal systems used mainly for arithmetic instructions, the binary and hexadecimal systems used preferably for logic instructions. These systems allow a shortened notation.

If the number is written without specifying a numeric system to be used it is considered to be a decimal number. The binary system is marked with the % character before the number. The hexadecimal system is marked with the \$ character before the number where digits 0 to 9, A to F are used:

```
LD    152                    ;decimal system
AND    %0111110100110100    ;binary system
AND    $7D34                 ;hexadecimal system
```

Negative numbers

For the decimal system the write of a negative number is permissible. The negative sign causes that instead of the binary equivalent of the number the binary complement is saved as the instruction operand of this 8 bit number (sint), 16 bits (int) or 32 bits (dint)(according to the type of instruction).

For example, the value of -1 will have the value of 255 in the sint format, the value of 65 535 in the int format and the value of 4 294 967 295 in the dint format. The real and lreal format already contains the information on the sign.

4.1.2. Immediate data formats

The format of the numeric constant is unequivocally determined by the type of instruction. The instructions expect the constants of type corresponding to the description of appropriate instruction.

If to the instruction requiring a numeric constant of 32 bit width a number of 8 bit width is written, then the compiler adds zeros to that number to 32 bit width. This way is thus permissible.

```
LDL    $1F                    ;32 bit width expected
LDL    $0000001F              ;identical notation
AND    %11                    ;16 bit width expected
AND    %0000000000000011     ;identical notation
```

The numeric constant of the real and lreal types is permissible only for the decimal system. Since the real type has the same length as the dword, uint and dint types but entirely different data interpretation, the numeric constant entry of the real type marks out that it contains the decimal point even if it is an integral number. The presence of the decimal point is thus the information important for the xPRO compiler to compile this constant into the correct format.

```
LDL    1                      ;type dint
```

```
LDL  1.0          ;type real
LDQ  1.0          ;type lreal
```

Note: Instead of the LDL instruction, the LD instruction is used in the user programs for the central units with the stack width of 32 bits (CPU series C). Nevertheless, the compiler accepts also the LDL instruction and converts it automatically to the LD one. By doing this, program portability is ensured (see chapter 8 Result stack).

Table 4.2 Immediate operand ranges

byte / sint	byte / usint	word / int	word / uint
-128 to +127	0 to 255	-32768 to +32 767	0 to 65535
	%0 to %11111111		%0 to %1111111111111111
-\$80 to \$7F	\$0 to \$FF	-\$8000 to \$FFFF	\$0 to \$FFFF
	#60#0 to #60#8.15		#60#0 to #60#18.12.15

dword / dint	long / uint
-2147483648 to +2147483647	0 to 4294967295
	%0 to %11111111111111111111111111111111
-\$80000000 to \$FFFFFFFF	\$0 to \$FFFFFFFF
	#60#0 to #60#59.59.59 *

real	lreal
$\pm 1,175494 \times 10^{-38}$ to $\pm 3,402823 \times 10^{38}$	$\pm 2,2 \times 10^{-308}$ to $\pm 1,8 \times 10^{308}$

* Sexagesimal system is valid for hours, minutes and seconds, not for days.

Symbolic name used as operand

As a immediate operand it is also possible to use a symbolic name defined by the *#def* directive. A mathematical expression can be used, too.

```
#def number      10
LD  number      ;LD 10
LD  number *4   ;LD 40
```

Object number as operand – prefix *__indx*, *__bitpart*, *__bitcnt*

If we need to use a register number, table number or label number as an operand, then the *__indx* prefix will be used.

```
#reg uint register1,register2 ;RW0,RW2
LD  __indx (register1) ;LD 0
LD  __indx (register2) ;LD 2
```

If we need to use a register bit number as an operand, then the *__bitpart* prefix will be used.

```
#reg bool register1,register2 ;R10.0,R10.1
LD  __bitpart (register1) ;LD 0
LD  __bitpart (register2) ;LD 1
```

If we need to use the bit ordinal number as an operand within the entire register zone, then the *__bitcnt* will be used.

```
#reg bool register1,register2 ;R10.0,R10.1
LD  __bitcnt register1 ;LD 80
LD  __bitcnt register2 ;LD 81
```

Object address as operand - prefix `__offset`

By means of prefix `__offset` we can work with an object address as it is required by some special functions as the pointer for data positioning.

```
#reg usint work[20]          ;R150
    LD  __offset (list) ;LD 790 - for CPUs of B, D, E, M, S series
                                ;LD 24726 - for CPU of C series
```

Length of object as operand - prefix `__sizeof`

By means of the `__sizeof` prefix we can work with the object length, mostly a structure.

```
#struct list usint first, uint second, real third
    LD  __sizeof (list)      ;LD 7
```

4.2. ADDRESS OPERAND

Address operand space type

The address operand has the meaning of the position address, from where the information being processed is read or where the operation result is saved. The type of operand space is marked with the first character in the operand:

- X - scratchpad input image
- Y - scratchpad output image
- S - scratchpad system registers
- R - scratchpad user registers
- U - physical addresses
- D - user program data constant zone
- T - user program table zone

This first character is sometimes called as "operand space specifier".

In the user programs for the central units with the 32 bit stack width it is obligatory to use the beginning character of `%` before the specifier. By this it is determined unequivocally that this is an absolute address and not a symbolic name. With the central units with 16 bit stack width this way of notation is not obligatory, but we recommend to use it with regard to user program transferability.

Symbolic name used as operand

Usually, we use the symbolic name assigned by means of the `#def`, `#reg`, `#rem`, `#data` or `#table` directives as the address operand.

```
#def input %X0
#rem  bool  register1          ;R0.0 remanent
#reg  bool  register2          ;R1.0
#data  usint record = 1,2,3,4  ;D0, D1, D2, D3
#table uint  10,tab = 1,2,3,4  ;TW10
```

Table 4.3 Central unit address operand ranges for series E and M

bool	byte / usint / sint	word / uint / int
X0.0 - X15.7	X0 - X15	XW0 - XW14
Y0.0 - Y15.7	Y0 - Y15	YW0 - YW14
S0.0 - S63.7	S0 - S63	SW0 - SW62
R0.0 - R255.7	R0 - R255	RW0 - RW254
-	U\$0000 - U\$FFFF *	UW\$0000 - UW\$FFFE *
D0.0 - D255.7	D0 - D255	DW0 - DW254
T0.0 - T255.0 *	T0 - T255 *	TW0 - TW255 *

* Not implemented in central units of series E.

Table 4.4 Central unit address operand ranges for series S

bool	byte / usint / sint	word / uint / int
X0.0 - X127.7	X0 - X127	XW0 - XW126
Y0.0 - Y127.7	Y0 - Y127	YW0 - YW126
S0.0 - S63.7	S0 - S63	SW0 - SW62
R0.0 - R511.7	R0 - R511	RW0 - RW510
-	U\$0000 - U\$FFFF	UW\$0000 - UW\$FFFE
D0.0 - D255.7	D0 - D255	DW0 - DW254
T0.0 - T255.0	T0 - T255	TW0 - TW255

Table 4.5 Central unit address operand ranges for series B and D

bool	byte / usint / sint	word / uint / int
X0.0 - X127.7	X0 - X127	XW0 - XW126
Y0.0 - Y127.7	Y0 - Y127	YW0 - YW126
S0.0 - S63.7	S0 - S63	SW0 - SW62
R0.0 - R8191.7	R0 - R8191	RW0 - RW8190
-	U\$0000 - U\$FFFF	UW\$0000 - UW\$FFFE
D0.0 - D255.0	D0 - D255	DW0 - DW254
T0.0 - Tmax	T0 - T255	TW0 - TW255

dword / udint / dint	real
XL0 - XL124	XF0 - XF124
YL0 - YL124	YF0 - YF124
SL0 - SL60	SF0 - SF60
RL0 - RL188	RF0 - RF188
-	-
DL0 - DL252	DF0 - DF252
-	-

Table 4.6 Central unit address operand ranges for series C

bool	byte / usint / sint	word / uint / int
X0.0 - X8191.7	X0 - X8191	XW0 - XW8190
Y0.0 - Y8191.7	Y0 - Y8191	YW0 - YW8190
S0.0 - S6143.7	S0 - S6143	SW0 - SW6142
R0.0 - R40955.7	R0 - R40955	RW0 - RW40954
D0.0 - D2047.7	D0 - D2047	DW0 - DW2046
T0.0 - Tmax.0	T0 - Tmax	TW0 - TWmax

4. Instruction and operand structure

dword / uint / dint	real	lreal
XL0 - XL8188	XF0 - XF8188	XD0 - XD8184
YL0 - YL8188	YF0 - YF8188	YD0 - YD8184
SL0 - SL6140	SF0 - SF6140	SD0 - SD6136
RL0 - RL40952	RF0 - RF40952	RD0 - RF40948
DL0 - DL2044	DF0 - DF2044	DD0 - DD2040
TL0 - TLmax	TF0 - TFmax	-

* In the central units of series C the U-operands are replaced with the RFRM system instructions which performs immediate update of dedicated peripheral module data.

Note: Apart from the data, each T-table additionally occupies 4 bytes for service information in the memory.
 A list of T-table addresses created by the compiler is part of the user program. This list has n+1 items, where n stands for the highest table number declared in the user program. In the list, addresses of all tables from T0 to Tn are listed including those not being declared (having zero address). As the result of this, if we use only the tables with high numbers in the user program, the table address will occupy unnecessarily a great part of the memory determined for the user program.
 Therefore, we recommend to number the tables ascendingly from 0 (the xPRO compiler supports this principle when using symbolic names of the tables).

4.2.1. Bool type operand

Data of the bool type in the scratchpad represents one concrete bit in the byte given by the address of the byte and the number of the bit.

Data assumes values 0 and 1 (due to a different interpretation on the stack we call them log.0 and log.1).

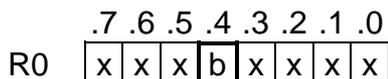


Figure 4.1 Bool type data saving in the scratchpad
b - logical value of the bit (log.0 or log.1)

The bool type operand is addressed by the byte address in the operand space and the bit number inside this byte. In absolute expression, the byte address is unequivocally given by the number in the address operand. The bit number has a value 0 to 7 and is entered after the point, for example:

```
LD   %X1.2
WR   %Y1.7
LD   %S53.4
WR   %R123.6
LD   %D25.6
```

The lowest (low) bit of the byte correspond to 0, the highest (upper) bit of the byte corresponds to 7. If the contents of the byte is displayed as a binary number or as a sequence of bit values on the line, then the lowest bit is rightmost and the top one leftmost.

The U-operands do not enable bit access.

The T-operands determining the bit access to the tables always have the bit number equal to 0. So the bit access is distinguished from the byte one, otherwise the bit number does not have any meaning in this case.

```
LTB %T4.0 ;load item from bit table T4 (the item number
           ;is determined by the index saved on the stack top)
```

Symbolic expression

In symbolic expression the operand is determined by the *#def*, *#reg*, *#rem*, *#data* and *#table* directives.

```
#def input %X2.1
#rem  bool register1          ;R0.0 remanent
#reg  bool register2          ;R1.0
#reg  bool register3          ;R1.1
#data  bool record = 1,0,1,0  ;D0.0, D0.1, D0.2, D0.3
#table bool tab = 1,0,1,1     ;T0.0
```

Casting to bool

If we need to cast an operand locally in our user program (we need to use another operand type than it is defined by the above mentioned directives) it is possible to cast it locally to the bool type by adding the bit number. The bits can be numbered within the entire operand width, thus from 0 to 7 for 8 bit width, 0 to 15 for 16 bit width and 0 to 31 for 32 bit width.

```
#reg  usint register1          ;R0
#reg  uint  register2          ;RW1
#reg  udint register3          ;RL3
#table usint tab = 1,2,3,4     ;T0
;
LD    register1.0             ;LD %R0.0
LD    register2.5             ;LD %R1.5
LD    register3.31            ;LD %R6.7
LTB   tab.0                   ;LTB %T0.0
```

4.2.2. Byte / usint / sint operand

The byte, usint and sint types data in the scratchpad represent one concrete byte given by the address.

The data assumes values of 0 to 255 (byte, usint) or, when using the highest (upper) bit as the sign, -128 to +127 (sint).

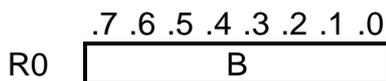


Figure 4.2 Byte, usint, sint data saving in the scratchpad
B - byte value

In absolute expression the operand is addressed by the number in the address operand.

```
LD    %X1
WR    %Y0
LD    %S53
WR    %R123
LD    %D25
```

4. Instruction and operand structure

```
LTB %T4
```

The U-operand is addressed by the physical hexadecimal address of 16 bit width (\$ stands for the hexadecimal number system).

```
LD %U$9101 ;address hexadecimal value
```

Symbolic expression

In symbolic expression the operand is determined by the *#def*, *#reg*, *#rem*, *#data* and *#table* directives.

```
#def input %X2
#rem usint register1 ;R0 remanent
#reg byte register2 ;R1
#reg usint registr3 ;R2
#reg sint registr4 ;R3
#data usint record = 1,2,3,4 ;D0, D1, D2, D3
#table usint tab = 1,2,3,4 ;T0
```

Casting to byte, usint, sint

If we need to cast an operand locally in our user program (we need to use another operand type than it is defined by the above mentioned directives) it is possible to cast it locally by adding the *byte*, *usint*, *sint* prefixes.

```
#reg bool register1 ;R0.0
#reg uint register2 ;RW1
#reg uint register3 ;RL3
#table uint tab = 1,2,3,4 ;TW0
;
LD usint register1 ;LD %R0
LD sint register2 ;LD %R1
LD usint register3+3 ;LD %R6
LTB usint tab ;LTB %T0
```

The variable of the word, uint, int types can be cast to two variables of the byte, uint, int types also by means of the *__high* and *__low* prefixes.

```
LD __high (register2) ;LD R2
LD __low (register2) ;LD R1
```

4.2.3. Word / uint / int operand

The word, uint and int types data in the scratchpad represent two concrete bytes given by the address of the first of them. The data is saved in such a way that low-significance byte has the low address that the high-significance byte (Intel convention).

The data can assume values of 0 to 65 535 (word, uint) or, when using the highest (upper) bit as the sign, -32 768 to +32 767 (int).

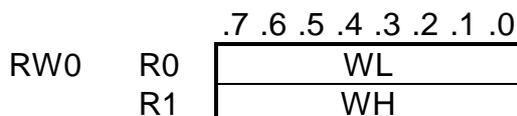


Figure 4.3 Word, uint, int data saving in the scratchpad

WH - higher byte value
WL - lower byte value

In absolute expression, the address of operand is written in such a way that behind the specifier the W character is written as the symbol of the operand type followed by a numeric value addressing the lower word byte. If, for example, register R14 has the value of \$21 and register R15 a value of \$43, then by means of instruction

```
LD    %RW14
```

we read a value of \$4321. Similarly, other operands are written.

```
LD    %XW1
WR    %YW0
LD    %SW53
WR    %RW123
LD    %DW25
LTB   %TW4
```

Similarly, using the U-operand means working with two physical addresses following each other. For example, if we have a sixteen-input binary unit with address 2 in PLC NS950, the state of the input signals of byte 0 is % 1101 0001 in binary notation and the state of the input signals of byte 1 is %0010 1100 in binary notation, then by means of the following instruction

```
LD    %UW$9200
```

we load value \$2CD1.

Time unit coding in timer instructions

Timer instructions (TON, TOF, RTO, IMP) work only with the uint type operand containing the current value of the timer. Moreover, it is necessary to enter a time unit, which the instruction should use and in which the value of the timer is measured out. Four time units are possible that are entered by means of a code number 0 to 3 specified in the instruction after the address number. Both items are separated by the full stop. The time units are encoded as follows:

- .0 - unit of 10 ms
- .1 - unit of 100 ms
- .2 - unit of 1 s
- .3 - unit of 10 s

If the code of the time unit is not specified, it is understood as zero one, i.e. the unit of 10 ms. Let us specify some examples of timer instructions:

```
TON   %RW10.0      ;timer is at RW10, unit of 10 ms
TON   %RW10        ;identical write
TOF   %RW24.1      ;timer is at RW24, unit of 100 ms
RTO   %RW32.2      ;timer is at RW32, unit of 1 s
IMP   %RW34.3      ;timer is at RW34, unit of 10 s
```

Symbolic expression

In symbolic expression, the operand is determined by the *#def*, *#reg*, *#rem*, *#data* and *#table* directives.

```
#def input %XW2
#rem   uint register1      ;RW0 remanent
#reg   word register2     ;RW2
#reg   uint registr3      ;RW4
#reg   int  registr4      ;RW6
```

4. Instruction and operand structure

```
#data uint record = 1,2,3,4 ;DW0, DW2, DW4, DW6
#table uint tab = 1,2,3,4 ;TW0
```

Casting to word, uint, int

If we need to cast an operand locally in our user program (we need to use another operand type than it is defined by the above mentioned directives) it is possible to cast it locally by adding the *word*, *uint*, *int* prefixes.

```
#reg bool register1 ;R0.0
#reg uint register2 ;R1
#reg uint register3 ;RL2
#table uint tab = 1,2,3,4 ;T0
;
LD uint register1 ;LD %RW0
LD int register2 ;LD %RW1
LD uint register3+2 ;LD %RW4
LTB uint tab ;LTB %TW0
```

4.2.4. Dword / uint / dint operand

The dword, uint and dint types data in the scratchpad represents four concrete bytes determined by the address of the first one of them. The data is saved in such a way that the low-significance byte has the lowest address (Intel convention).

The data assumes values 0 to 4 294 967 295 or, when using the highest (upper) bit as the sign, -2 147 483 648 to +2 147 483 647.

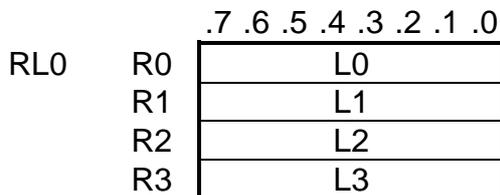


Figure 4.4 Dword, uint, dint data saving in the scratchpad

L0 - lowest byte value

:

L3 - highest byte value

In absolute expression, the address of the operand is written in such a way that behind the space specifier the L character is written as the symbol of the operand type followed by a numeric value addressing the lowest byte. If, for example, register R14 has the value of \$21, register R15 the value of \$43, register R16 the value of \$65 and register R17 the value of \$87, then by means of instruction

```
LD %RL14
```

we load \$87654321. Similarly, other operands are written.

```
LD %XL1
WR %YL0
LD %SL53
WR %RL123
LD %DL25
LTB %TL4
```

The U operand does not enable access in this format.

Symbolic expression

In symbolic expression, the operand is determined by the *#def*, *#reg*, *#rem*, *#data* and *#table* directives.

```
#def input %XL4
#rem   uint register1           ;RL0 remanent
#reg   dword register2         ;RL4
#reg   uint  register3         ;RL8
#reg   dint  register4         ;RL12
#data  uint record = 1,2,3,4   ;DL0, DL4, DL8, DL12
#table uint tab = 1,2,3,4      ;TL0
```

Casting to dword, uint, dint

If we need to cast an operand locally in our user program (we need to use another operand type than it is defined by the above mentioned directives) it is possible to cast it locally by adding the *dword*, *uint*, *dint* prefixes.

```
#reg   bool  register1         ;R0.0
#reg   usint register2         ;R1
#reg   uint  register3         ;RW2
#table usint tab = 1,2,3,4     ;T0
;
      LD   uint register1       ;LD  %RL0
      LD   dint register2       ;LD  %RL1
      LD   uint register3       ;LD  %RL2
      LTB  uint tab             ;LTB %TL0
```

4.2.5. Real operand

The real type data in the scratchpad represent four concrete bytes given by the address of the first one of them. The data is saved in such a way that the lowest-significance byte has the lowest address (Intel convention).

According to IEEE-754 (Institute of Electrical and Electronics Engineers) the data for the number with single precision assumes a value within a range of approx. $\pm 1,175494 \times 10^{-38}$ to $\pm 3,402823 \times 10^{38}$ with accuracy to approx. 7 valid decimal digits. Further, four values specifying the following states are defined:

\$7FFFFFFF	- invalid number (NaN - not a number)
\$FFFFFFFF	- invalid number (NaN - not a number)
\$7F800000	- exceeding the range of positive numbers (+INF)
\$FF800000	- exceeding the range of negative numbers (-INF)

Real type structure

The data is divided into three parts. The highest (upper) bit marked with **s** on figure 4.5 determines the sign of the whole number. If $s = 0$, the number is positive, if $s = 1$, the number is negative. Next 8 bits marked with the **e** character are called exponent, which carries information on the number size. The other 23 bits marked with the **m** character are called mantissa and carry significant digits (i.e. without insignificant zeros on the left side).

The mantissa has not the control bit expressed which means that it represents the binary number in the form of 1,mmmmmmmm. The exponent then specifies the number of binary orders, by which we have to move the decimal point imaginarily to get the number


```
#def input %XF4
#rem   real register1           ;RF0 remanent
#reg   real register2         ;RF4
#data  real record = 1.0,2.0,3.0,4.0 ;DF0, DF4, DF8, DF12
#table real tab = 1.0,2.0,3.0,4.0   ;TF0
```

Casting to real

If we need to cast an operand locally in our user program (we need to use another operand type than it is defined by the above mentioned directives) it is possible to cast it locally by adding the *real* prefix.

```
#reg   bool  register1         ;R0.0
#reg   usint register2         ;R1
#reg   uint  register3         ;RW2
#table usint tab = 1,2,3,4     ;T0
;
      LD   real register1       ;LD  %RF0
      LD   real register2       ;LD  %RF1
      LD   real register3       ;LD  %RF2
      LTB  real tab             ;LTB %TF0
```

Caution! By simple casting of the operand to the real type it is not possible to convert the content of the operand to the real type. It is necessary to use the corresponding conversion instruction (also for conversions between real and lreal types).

4.2.6. Lreal operand

The lreal type data in the scratchpad represent eight concrete bytes given by the address of the first one of them. The data is saved in such a way that the lowest-significance byte has the lowest address (Intel convention).

According to IEEE-754 (Institute of Electrical and Electronics Engineers) the data for the number with double precision assumes a value within a range of approx. $\pm 2,2 \times 10^{-308}$ to $\pm 1,8 \times 10^{308}$ with accuracy to approx. 16 valid decimal digits. Further, four values determining the following states are defined:

- \$FFFFFFFF FFFFFFFF - invalid number (NaN - not a number)
- \$FFFFFFFF FFFFFFFF - invalid number (NaN - not a number)
- \$7FF00000 00000000 - exceeding the range of positive numbers (+INF)
- \$FFF00000 00000000 - exceeding the range of negative numbers (-INF)

Lreal type structure

The data is divided into three parts. The highest (upper) bit marked with **s** on figure 4.6 determines the sign of the whole number. If $s = 0$, the number is positive, if $s = 1$, the number is negative. Next 11 bits marked with the **e** character are called exponent, which carries information on the number size. The other 52 bits marked with the **m** character are called mantissa and carry significant digits (i.e. without insignificant zeros on the left side).

The mantissa has not the control bit expressed which means that it represents the binary number in the form of 1,mmmmmmmm. The exponent then specifies the number of binary orders, by which we have to move the decimal point imaginarily to get the number required. If we move the decimal point to the left, the exponent will be negative, if we move it to the right, the exponent will be positive.

4. Instruction and operand structure

The exponent is not expressed in the binary complementary code, but in the code with shifted zero, which means that value \$3FF (1023) will be added to the actual value of the exponent. Zero is represented by \$3FF, number one by \$400, etc. When decoding the exponent the value of \$3FF must be subtracted.

The value of the number can be expressed as follows:

$$val = (-1)^s \times 2^{(e-1023)} \times 1,m$$

		.7 .6 .5 .4 .3 .2 .1 .0
RD0	R0	m m m m m m m m
	R1	m m m m m m m m
	R2	m m m m m m m m
	R3	m m m m m m m m
	R4	m m m m m m m m
	R5	m m m m m m m m
	R6	e e e e m m m m
	R7	s e e e e e e e

Figure 4.6 Double and lreal format data saving in the scratchpad
s - sign(1 bit)
e - exponent (11 bits)
m - mantissa (52 bits)

In absolute expression, the operand address is written in such a way that after the specifier the D character is written as the operand type symbol followed by a number addressing the lowest byte.

```
LD    %XD1
WR    %YD0
LD    %SD53
WR    %RD123
LD    %DD25
```

The U and T operands do not enable any access in this type.

Symbolic expression

In symbolic expression, the operand is determined by the *#def*, *#reg*, *#rem* a *#data* directives.

```
#def input    %XD4
#rem  lreal register1          ;RD0 remanent
#reg  lreal register2        ;RD8
#data lreal record = 1.0,2.0,3.0,4.0 ;DD0, DD8, DD16, DD24
```

Casting to lreal

If we need to cast an operand locally in our user program (we need to use another operand type than it is defined by the above mentioned directives) it is possible to cast it locally by adding the *lreal* prefix.

```
#reg uint register1          ;RL0
#reg uint register2        ;RW4
;
LD  lreal register1        ;LD  %RD0
LD  lreal register2        ;LD  %RD4
```

Caution! By simple casting of the operand to the lreal type it is not possible to convert the content of the operand to the lreal type. It is necessary to use the corresponding conversion instruction (also for conversions between real and lreal types).

4.3. TRANSITION DESTINATION

For the jump and call instructions, the operand is the label to which the transition will be executed. The instruction has the following format, as an example:

```
JMP %L15
```

and passes the program execution to label L15. In the program the label is written as instruction L with the corresponding numeric parameter and serves exclusively as the jump and call target. During execution, the L instruction is interpreted as no operation instruction.

Using symbolic labels is much more effective, where instead of the L instruction with the label number we write a symbolic name after which the compiler itself inserts a number. By doing this, the user program is optimised for a list of label addresses to be as short as possible, being part of the user program and having information for the PLC during jump instruction execution.

```
jump1:                ;L 0
:
JMC jump2             ;JMC %L1
JMP jump1             ;JMP %L0
:
jump2:                ;L 1
```

Application of #label directive

The *#label* directive is used only when we want to assign a concrete number to a label or when we need to ensure the order of labels for indirect jumps by instructions LMI and CAI. The *#label* directive is necessary also when we use the number of the label as operand (with *indx* prefix) or as a table item before the label is used. For the compiler it would be an unknown name.

Caution: If we use the L label with the highest value of the n parameter in the user program, the compiler will produce a list of label addresses for all labels from L0 to Ln including those that will not be used (they will have zero address). As a consequence of this, if we use in the program the L1023 label only, the label list will occupy unnecessarily 2 KB of the memory, with the addresses of the other labels being zero! Therefore, we recommend to number the tables ascendingly from 0 (the xPRO compiler supports this principle when using symbolic names of the tables).

4.4. INSTRUCTION PARAMETER

Some instructions require entering a numeric parameter. With some instructions the parameter is saved passively and serves only for identification of the instruction (for example with instruction L, NOP, P, E). With some instructions, the parameter effects the

4. Instruction and operand structure

way of instruction execution, for example, with instructions ROL, ROR, POP it specifies the number of steps (shifts).

The instructions working with more stacks (CHG, WAC, LAC) use the character parameter marking the stack being in use.

The instruction parameters is thus entered as a numeric value or one or two-letter string. The parameter range is determined by the type of instruction.

5. SCRATCHPAD MEMORY STRUCTURE

Scratchpad function

Scratchpad or scratchpad memory is a section of the PLC memory space that is accessible for reading as well as writing user data. The PLC instructions enable access to any part of the scratchpad in all supported formats (based on CPU series). This memory is pre-divided into several sections with designated meaning. A schematic diagram of the scratchpad memory is on figure 5.1.

Scratchpad division

The scratchpad memory is divide into the following sections:

- ◆ images of input signals X
- ◆ images of output signals Y
- ◆ system registers S
- ◆ user registers R

Central unit series	E	M	S	D	B	C
Input signal images X	X0 ⋮ X15	X0 ⋮ X15	X0 ⋮ X127	X0 ⋮ X127	X0 ⋮ X127	X0 ⋮ X8191
Output signal images Y	Y0 ⋮ Y15	Y0 ⋮ Y15	Y0 ⋮ Y127	Y0 ⋮ Y127	Y0 ⋮ Y127	Y0 ⋮ Y8191
System registers S	S0 ⋮ S63	S0 ⋮ S63	S0 ⋮ S63	S0 ⋮ S63	S0 ⋮ S63	S0 ⋮ S6143
User registers R	R0 ⋮ R255	R0 ⋮ R255	R0 ⋮ R511	R0 ⋮ R8191	R0 ⋮ R8191	R0 ⋮ R40955

Figure 5.1 Scratchpad memory structure including operand ranges for particular series of central units

Access to scratchpad

Generally, the principle is followed that the access of the system program to the scratchpad memory is realized exclusively at the phase of the I/O scan of the user program cycle. This concerns not only scanning physical inputs into area X and setting values from area Y to physical outputs, but it is valid also for the changes of the values of system variables S. This means that for the cycle time of the user program, the scratchpad data is frozen and updated after the forthcoming cycle I/O scan. In this way, a possibility of occurrence of various hazardous states as the result of asynchronism of change moments of particular variables is reduced in the user program. Within the cycle only such variables are changed that are influenced by the user program (direct write to the scratchpad - WR, WRC, WRA, PUT, LET, BET, SET, RES), or effects of some functions (for example setting of result flags, counter and timer status updating, shift register updating, etc.).

Notice: It is necessary to realize that the moments of user interrupt are asynchronous against the idle cycle of the user program and through unsystematic management over stack variables the user himself can cause enough hazardous states. Thus it is necessary to pay attention to

5. Scratchpad memory structure

the assignment of the variables, establishing rules for co-operation between the processes of the idle cycle and interrupt processes. For this, the Mosaic development environment provides a significant support.

Data backup during power failure

During supply voltage failure, a part of the scratchpad content is backed up from a back up power supply (so called remanent zone in user registers R). During restart these backed up data can be used for further control - depending on the type of start-up and other circumstances (scratchpad integrity, unchanged content of the user program, etc.). When selecting configuration, the user also can select the size of the remanent zone.

5.1. INPUT IMAGES X

Before each program cycle start the central unit ensures the update of this area of the scratchpad memory from the input peripheral units based on the declaration table specified in the user program and describing the assignment between input images X and physical addresses of particular units.

In case of insufficient memory in the X area it is possible to use without limitations the area of user registers R for the same purpose.

5.2. OUTPUT IMAGES Y

After each cycle termination of the program the central unit assures the transfer of the results from this area of the scratchpad memory to the outputs of the peripheral units based on the declaration table specified in the user program and describing the assignment between output images Y and physical addresses of particular units.

In case of insufficient memory in the Y area it is possible to use without limitations the area of user registers R for the same purpose.

5.3. SYSTEM REGISTERS S

This area of the scratchpad memory is reserved for specific use by the system program of the controller and is not recommended to use it for other purposes. Some bits and bytes are regularly set by the system program at the I/O scan and are suitable for reading only. Conversely, some bits by their setting modify the behaviour of the system program.

Table 5.1 System register overview

Registers	Application	CPU Series
S0	arithmetic operation result flag	E M S D B C
S1	logical operation result flag	E M S D B C
S2	system state flags	E M S D B C
S3	time of recent cycle in 10 ms	E M S D B C
S4	cycle counter	E M S D B C
S5	counter of tens of milliseconds of system time	M S D B C
S6	counter of seconds of system time	M S D B C
S7	counter of minutes of system time	M S D B C
S8	counter of hours of system time	M S D B C
S9	counter of days in a week	M S D B C
S10	counter of days in a month	M S D B C
S11	month counter	M S D B C
S12	year counter	M S D B C
S13	units of time	M S D B C
S14 - S15	counter in 100 ms increments	M S D B C
S16 - S17	counter in 1 s increments	M S D B C
S18 - S19	counter in 10 s increments	M S D B C
S20	time unit leading edges from S13	M S D B C
S21	time unit falling edges from S13	M S D B C
S22 - S23	time of recent cycle in 100 µs	S D B C
S24 - S29	process control masks	M S D B C
S30 - S33	spare	
S34	error internal code	E M S D B C
S35	hardware state flags	E M S D B C
S36	processor board temperature	C
S37	spare	
S38	user program edition number	E M S D B C
S39	contents change number	E M S D B C
S40 - S41	PLC system program version code	E M S D B C
S42 - S43	spare	
S44 - S45	compiler type	E M S D B C
S46 - S47	spare	
S48 - S51	error full code	S D B C
S52 - S55	counter 1 ms	C
S56 - S57	interrupting module address	C
S58 - S63	spare	
S64 - S75	system registers of high-level programming language	C
S76 - S2047	spare	
S100 - S227	peripheral system state zone	C
S228 - S2047	spare	
S2048 - S6143	system stack	C

Attention! The not occupied system registers (in table 5.1 marked as reserve) must not be used at all as user memory! These registers can be used by the system for testing and diagnostics purposes. Any write to them can result in unpredictable consequences!

5. Scratchpad memory structure

The meaning of the system registers are as follows:

S0 - flags of arithmetic operation results

It is influenced only by some arithmetical instructions, comparison instructions, timer instructions and counters. The other instructions do not change it.

S0.7	S0.6	S0.5	S0.4	S0.3	S0.2	S0.1	S0.0
0	D5.2	D5.1	D5.0	CI	≤	<(CO)	=(ZR)

- S0.0 (=) - equality of both operands
 (ZR) - result zero value
 - division by zero during division instructions
- S0.1 (<) - first operand < second operand
 (CO) - output carry (carry out), during the operation, carry was executed to a higher order
- S0.2 (≤) - first operand ≤ second operand
 - logical add of bits S0.0 OR S0.1
- S0.3 (CI) - input carry (carry in)
 - serves for cascading of arithmetical operations ADD, SUB, INR, DCR, EQ, LT, GT (16 bit model only). If we want to respect the carry from a lower order in some of these operations, it is necessary to set CI to the value of the carry before this operation. At every setting of S0 (after the instructions specified) the CI bit is set to zero, so that the next arithmetic instruction is executed without carry from below. Arithmetic instructions are thus cascaded only when the user enters the value of the carry in bit CI. In the 32 bit model, this flag is not functional.
- S0.4 to S0.6 (D5) - after instruction BCD they contain the highest digit of the result (the maximum value is 6), after other instructions the bits are set to zero
- S0.4 (OV) - exceeding the maximum timer range (at this cycle or any time before during simultaneous activation of the timer)
- S0.5 (OC) - exceeding the maximum timer range just at this cycle
- S0.7 - spare

Detailed information is given in instruction descriptions effecting register S0.

S1 - flags of logic operation results

Table instructions set the S1.0 bit with the following meaning:

- S1.0 = 1 - item within table range, item found
- S1.0 = 0 - item out of table range, item not found

Arithmetic instructions for applications with the floating point. block operations and operations with structured tables set the S1.0 bit with the following meanings:

- S1.0 = 1 - input parameters are OK, the result is valid
- S1.0 = 0 - input parameters out of range, the result is invalid

Instruction FLG sets register S1 according to the content of the stack top.

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

- S1.0 to S1.3 (N) - binary number (N4, N3, N2, N1, N0) assuming values 00000 to 10000, having the meaning of the number of logic one bits of the stack top (bit N4 is released in all bits of the stack top)
- S1.4 (ANL) - logical product of bits of low byte of A0 stack top
- S1.5 (ANH) - logical product of bits of upper byte of A0 stack top
- S1.6 (ORL) - logical add of bits of low byte of A0 stack top
- S1.7 (ORH) - logical add of bits of upper byte of A0 stack top

The STE instruction sets bits S1.0 and S1.1 in the following meaning:

- S1.0 = 1 - sequencer status changed
- S1.0 = 0 - sequencer status not changed
- S1.1 = 1 - I/O scan (sequencer carry - status changed from 15 to 0)
- S1.1 = 0 - other cases

Detailed information is given in instruction descriptions that effect register S1.

S2 - system state flags

Set by the system program according to its state at the cycle I/O scan.

S2.7	S2.6	S2.5	S2.4	S2.3	S2.2	S2.1	S2.0
LIM	NRS	ON	RST	HOT	RUN	MS	SP

- S2.0 (SP) - state of service input SP - external start of the PLC
- S2.1 (MS) - state of service input MS - external blocking of outputs
- S2.2 (RUN) - 1 - cycle start, the program is executed (RUN mode)
0 - cycle stop, the program is not executed, the PLC dwells at the cycle I/O scan (HALT mode)
- S2.3 (HOT) - 1 - first pass through the cycle after warm restart
- S2.4 (RST) - 1 - first pass through the cycle after cold restart
- S2.5 (ON) - 1 - active outputs
0 - locked outputs
- S2.6 (NRS) - 1 - first pass through the cycle without restart
- S2.7 (LIM) - 1 - cycle time first limit exceeded (alarm)

Bits S2.3, S2.4 and S2.6 are useful for variable initialisation.

Notice: The S2 register is designated for indication only, not for write.

S3 - time of recent cycle in tens of milliseconds

The binary data with the unit of 10 ms (range 0 - 2,55 s) informs on the duration of the recent cycle of the user program.

S4 - cycle counter

Binary data that is set to zero at every system restart and after each I/O scan it is increased by one. It enables more complex distribution of the control algorithm into particular cycles.

S5 to S12 - system time and date

A binary data file, the data has the meaning of time in units of time. In the user program it enables to use hour and date data without complicated recalculations. The time data is received from the real time circuit and is updated at every cycle

5. Scratchpad memory structure

I/O scan. During power supply failure time does not stop, since this circuit has a backup battery.

- S5 - counter of tens of milliseconds (0 - 990 ms)
- S6 - counter of seconds (0 - 59 s)
- S7 - counter of minutes (0 - 59 minutes)
- S8 - counter of hours (0 - 23 hours)
- S9 - counter of days in a week (1 - 7)
- S10 - counter of days in a month (1st to the last day in the current month, leap year respected)
- S11 - counter of months (1 to 12)
- S12 - counter of years - the last two digits of year (0 - 99)

S13 - time units

A file of bit variables that change their status once a unit specified. The course of these time signals has repeating of approx. 1:1 and is derived from states S5 to S8 (figure 5.2 to 5.5). They can be used as a source of time pulses for user counters, for realization of time functions (flashing, D, T and JK circuits). The time-measuring variables can be used under the assumption that the cycle time of the user program is reliably shorter than the half of the time unit being used.

The particular bits have the following meaning:

S13.7	S13.6	S13.5	S13.4	S13.3	S13.2	S13.1	S13.0
1 day	1 hour	10 min	1 min	10 s	1 s	500 ms	100 ms

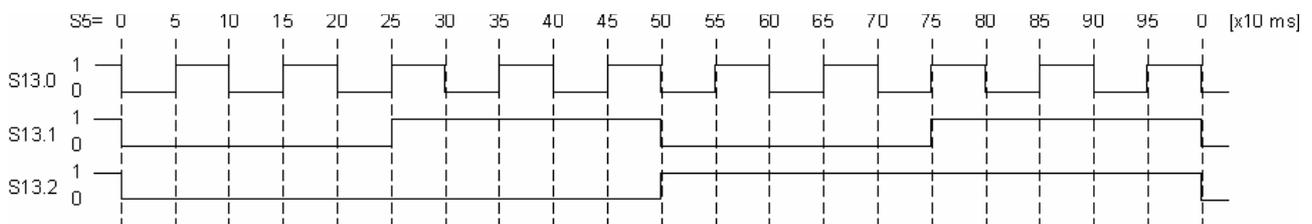


Figure 5.2 State of bit variables S13 in relation to the counter of tens of milliseconds S5

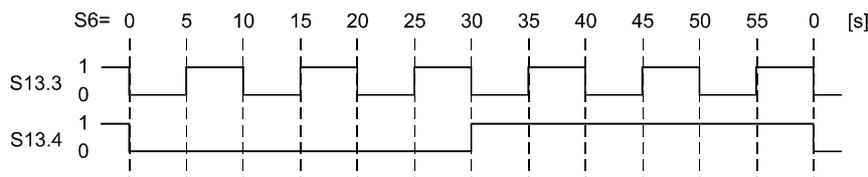


Figure 5.3 State of bit variables S13 in relation to the counter of seconds S6

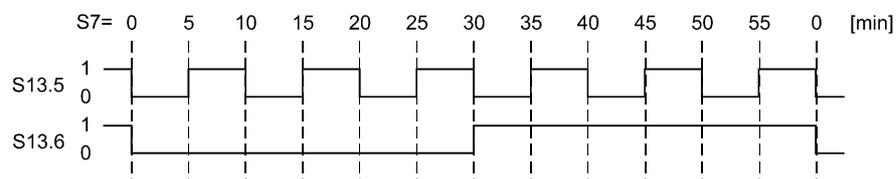


Figure 5.4 State of bit variables S13 in relation to the counter of minutes S7

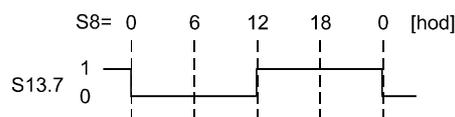


Figure 5.5 State of bit variables S13 in relation to the counter of hours S8

S14, S15 - counter in 100 ms increments

S16, S17 - counter in 1 s increments

S18, S19 - counter in 10 s increments

Each word SW14, SW16, SW18 contains a binary data in a range of 65 535 unit of time. The changes of this data are synchronous with changes S5 to S13. The main use for them is replacement of timers, especially in cases of realization of time interval sequences (sequential and time control). The particular bits can be used as the sources of the time units. Based on our needs, only the low or upper byte can be used.

S20 - leading edges time units from S13

S21 - falling edges time units from S13

On the positions with the same meaning as at S13, the change signals are set with the change of homogenous content of S13 from log.0 to log.1 (S20) or from log.1 to log.0 (S21). The changes are evaluated against the state from the recent cycle.

S20.7	S20.6	S20.5	S20.4	S20.3	S20.2	S20.1	S20.0
S21.7	S21.6	S21.5	S21.4	S21.3	S21.2	S21.1	S21.0
1 day	1 hour	10 min	1 min	10 s	1 s	500 ms	100 ms

S22, S23 - time of recent cycle in 100 µs

Binary data with 100 µs units (range 0 - 6,5535 s) informs on duration of the recent cycle of the user program. It is specified data of the S3 register.

S24 to S29 - process control masks

Control masks for control and indication of activated processes P1 to P48 (chapter 10.). The assignment is as follows:

	.7	.6	.5	.4	.3	.2	.1	.0
S24	P8	P7	P6	P5	P4	P3	P2	P1
S25	P16	P15	P14	P13	P12	P11	P10	P9
S26	P24	P23	P22	P21	P20	P19	P18	P17
S27	P32	P31	P30	P29	P28	P27	P26	P25
S28	P40	P39	P38	P37	P36	P35	P34	P33
S29	P48	P47	P46	P45	P44	P43	P42	P41

The logic ones correspond to the active processes, the logic zeros to the passive processes. Bits S24.0 to S25.0 are set by the process scheduler at the cycle I/O scan and decides on the activation of the P1 to P9 processes. If the bits are overwritten by the user, the process planned is yet executed.

Bits S25.1 to S28.7 are available to the user to control processes P10 to P40. Setting the bit to log.1 results in inclusion of the process in question into the next cycle. After restart these bits are always set to zero.

Bits S29.0 to S29.7 are available to the user to forbid or permit the execution of interrupt processes P41 to P48. These bits are not implemented of the central unit of series M, so it is not possible to forbid the execution of the interrupt process. After restart, the bits corresponding to the interrupt processes used in the user program are set to log.1.

S30 to S33 - spare

5. Scratchpad memory structure

S34 - error internal code

The first byte of the code for the last error occurred.

S34.7 = 1 (code \geq 128) - fatal error, user program execution is stopped, the PLC is switched to the HALT mode and locks the outputs

S34.7 = 0 (code $<$ 128) - other errors significantly not effecting the control itself, the user program continues, these errors can be user-treated by means of this system register

- S34 = 0 - error-free state
- 2 - serial communication error
- 7 - remanent zone check error
- 8 - exceeding first limit of cycle time watch
- 9 - wrong system time of RTC circuit
- 16 - division by zero
- 17 - initial index for WMS instruction is out of table T
- 18 - initial index for LMS instruction is out of table T
- 19 - table instruction over the scratchpad exceeded its range
- 20 - data source block was defined out of range of the scratchpad, data or table
- 21 - data destination block was defined out of range of the scratchpad or table
- 32 - user program failure detection during continuous check
- 80 - central unit does not support the service for user instructions
- 81 - central unit does not support the service required for user instructions
- 128 - user program error
- 129 - peripheral system error
- 130 - communication error with expansion modules
- 131 - errors in serial channels
- 160 to 175 - errors in peripheral system

S35 - flags of hardware state

S35.7	S35.6	S35.5	S35.4	S35.3	S35.2	S35.1	S35.0
STEN	STIN	0	0	0	0	ERO	BAT

- S35.0 (BAT) - state of backup battery of RAM and RTC
 - 1 - voltage of backup battery is lower than 2,1 V
 - 0 - backup battery OK
- S35.1 (ERO) - 1 - binary output error (output short circuit, unloaded output) (contained in TC500, TC600)
- S35.6 (STIN) - current time indication (TC700)
 - 0 - normal time
 - 1 - daylight saving time
- S35.7 (STEN) - automatic transition between daylight saving time and normal time (TC700)
 - 0 - off, time the whole year without shifts
 - 1 - on, time is automatically changed

S36 - processor board temperature (TC700)

Temperature measured on the central unit board in °C. It is possible to control for example fans in control boards.

S37 - spare

S38 - user program edition number

In the activation sequence, it is copied from the configuration constant of the user program.

The value of the constant can be entered through the Mosaic environment.

S39 - content change number

In the activation sequence, it is copied from the configuration constant of the user program.

The value of the constant can be entered through the Mosaic environment. It is determined mainly as the differentiation of user program versions.

S40, S41 - central unit system program version code

For example, for version 4.1 the code will be S40 = 4 a S41 = 1.

S42, S43 - spare

S44, S45 - compiler type

Compiler type, by means of which the user program was created.

S46, S47 - spare

S48 to S51 - full error code

Full code of the last error occurred determined for easy reading of the last error occurred by the superior system. The error code is saved in the udint format, i.e. the first byte is saved in register S51 (identical with the content of register S34), the last byte is saved in register S48. An overview of error codes is listed in the manual of the PLC type in question.

S52 to S55 - counter with step 1 ms

A ms unit counter of type udint SL52. It allows a more precise time control. Particular bits can be used as time unit sources. Any byte or word can be used as required.

S56, S57 - interrupting module address

When entering the interrupting process P42 (interrupt from a peripheral module) the register S56 contains the rack position and the register S57 the rack number, where the module is fitted that initiated this interrupt. This information is used to distinguish interrupt from more modules and they can be also used to acquire parameters for the RFRM and STATM instructions. Details are given in chapter 10.5.

S58 to S63 - spare

S64 to S75 - system registers of higher programming language

System registers SL64, SL68, SL72, used by the system resources of the PLC supporting a higher programming language. The value must not be changed through direct access in any case.

5. Scratchpad memory structure

S76 to S2047 – spare

S100 to S227 - peripheral system status zone

Registers S100 to S227 contain the peripheral system status zone providing information on immediate status of each peripheral module. This is important especially in such cases, when it is allowed to take out a peripheral module under run and the user program requires a piece of information whether the data read from the module are valid. In other respects this zone can be used for a detailed PLC diagnostics realized by a superior system. To each position in the rack of the modular PLC or in the assembly of a compact PLC (see the relevant manual describing the particular PLC) corresponds one register, the index of which can be deduced from the following formula:

$$n = (r * 16) + p + 100$$

where n stands for register resultant index

r stands for rack number

p stands for rack position number

As the result of this is that the module fitted in rack 0 on position 0 has assigned register S100, module on position 1 register S101, ..., module in rack 1 on position 0 register S116, etc. All the registers of the status zone have the following structure:

Sn.7	Sn.6	Sn.5	Sn.4	Sn.3	Sn.2	Sn.1	Sn.0
POS	OTH	DEC	ERR	0	0	DATA	ECOM

- Sn.0 (ECOM) - communication status with module
0 - communication OK
1 - module stopped communicating
- Sn.1 (DATA) - validity of transmitted data
0 - data in scratchpad are not current, no data exchange
1 - current data in scratchpad, data exchange takes place
- Sn.4 (ERR) - an error reported by module
0 - module is without error
1 - module reports a serious error not allowing data exchange
- Sn.5 (DEC) - module operation is declared
0 - module is not operated by the user program
1 - module is operated by the user program
- Sn.6 (OTH) - wrong module type
0 - in position fitted module required by declaration
1 - module type fitted in position is different than required
- Sn.7 (POS) - position occupied
0 - position is not occupied
1 - module found on position

A detailed behaviour of the status zone depending on the PLC type is described in the manual of the particular PLC type.

S228 to S2047 - spare

S2048 to S6143 - system stack

Areas S2048 to S6143 are used as the system stack of the PLC, used by the system resources of the PLC supporting a higher programming language. The values saved here must not be changed through direct access in any case.

5.4. USER REGISTERS R

Memory area designated for user program variables, counters, timers, shift registers, step sequencers and dynamic tables. In the activation sequence of the user program, all R registers are set to zero after cold restart. After warm restart, the remanent zone of the R registers is kept unchanged, the other zones are reset.

Remanent zone

The remanent zone is the area of the R registers, the content of which remains unchanged when the PLC power supply is off. The size of this area is selected by means of the compiler of the user program (in the Mosaic development environment in the Project manager folder *Sw / Cpm*, item *Remanent register zone*, the start is always on register R0. The maximum size of the remanent zone is determined by the central unit series (see table 5.2).

Attention! Fundamentally, we do not place the input and output images of the peripheral units into the remanent zone. In case of intelligent peripheral units the beginning of data exchange can be violated which could result in their non-functionality.

It is up to the user how he will use the R registers, that will be used for working variables or tables and which will be used for realization of functional blocks.

The compiler integrated in the Mosaic development environment enables directly to declare the remanent variable by means of the *#rem* directive that is equivalent to the *#reg* directive, but in addition, it places the variable into the remanent zone (see chap. 9.5).

Table 5.2 Remanent zone maximum sizes for particular central unit series

CPU Series	Maximum remanent zone length	Backup registers
C	16384	R0 - R16383
B	4096	R0 - R4095
D	512	R0 - R511
E, M, S	256	R0 - R255

Use of the R registers for counters and timers

The system does not decide how many counters, timers etc. can be used. The only limitation is the total number of the R registers divided by two (uint type) or by four (udint type). To each pair of the R registers internal flags for timer and counter functions are assigned in the system. The system enables to use for the counter and timers both the pairs beginning with an even register (RW0, RW2, RW4, ...) and an odd register (RW1, RW3, RW5, ...). Needless to say, it is not possible to use registers RW0 and RW1 at a time. For one thing a collision in register R1 comes up and the same flags are assigned to these registers. If we want to use symbolic declarations of the variables by means of the *#reg* directives, such collision cannot occur. The same is valid also for the udint type counters.

The decision whether the object becomes remanent or not is made by the user in which area the object will be put. If the user program uses time-limited objects that eliminate each other in the course of time, then they can be realized with the same R registers. The registers reserved for a counter or shift register can be operated through various instructions for counting and shifting without error occurrence (not valid for timers). The R registers assigned to an object are freely accessible for other instructions, so, as an example, the value of the counter or timer can be compared with some data or it can be changed pertinently.

6. DIRECT INPUT/OUTPUT ACCESS

6.1. DIRECT INPUT/OUTPUT ACCESS - 16 BIT MODEL

Systems TECOMAT TC400, TC500, TC600 and NS950 allow direct access to peripheral units by means of the U-operands containing so called physical address of an input or output.

Physical address

The term "physical address" denominates the effective address that the inputs and outputs occupy on the bus. For standard peripheral unit operation, an image is assigned to the physical addresses of the unit in the scratchpad (area X, Y, R) by means of the *#unit* directive. The data is updated always at the cycle I/O scan.

U operand

But there are some situations when due to time it is necessary to load the immediate state or to write a value in the unit immediately. For this purpose a physical address marked by the U operand is available. The U operand thus provides an alternative to the X and Y areas in the scratchpad enabling direct contact with the peripheral units at the time as necessary without waiting for the cycle I/O scan.

The U operand enables the access in 8 bit and 16 bit formats by means of instructions LD and WR. It is useful to use this operand to support time-critical reactions. An excessive use results in reducing program performance, since the direct access to the peripheral units is more time-consuming than the operations with the scratchpad memory.

The physical address is written immediately after the U operand (or UW, as the case may be) always in hexadecimal notation. Particular physical addresses are differentiated according to the type of PLC, which is given by its design. Detailed information is given in the following chapters.

Restrictions for the use of the U operand

It is necessary to realize that working with physical address infringes the principle of constancy of the input and output data during the cycle and increases the risk of hazardous states. The use of the U operands should be limited solely to such cases when an immediate response must be ensured, for example when treating emergency situations, in the processes of interrupt handling, etc.

Important notes

By writing to the physical address or by reading from the physical address of the peripheral unit the corresponding value change in the image of this unit in the scratchpad memory **does not take place!**

In case of physical reading the value in the image of the cycle I/O scan is corrected but this will not do any harm (but it is necessary to take this into account).

But in case of physical writing, the correction must be ensured by the user program, otherwise the original value from the scratchpad will be written in the unit at the cycle I/O scan.

If it is necessary to use the physical write to the unit, it is better to switch off the service of the unit outputs in the software configuration during the compilation of the user program in the compiler (*#unit* directive item) and operate the outputs solely by the physical write through the U operand.

6.1.1 Physical addresses in PLC TECOMAT NS950

The physical addresses of the peripheral units are permissible only in the basic frame (frame 0 operated directly by the central unit). The peripheral units in expansion frames are accessible only through their images in the scratchpad memory.

Physical address structure

Peripheral unit physical address has the following structure:

address upper byte								address low byte							
A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0

- A15 - A12 - unit type (fixed)
 - 0xxx (0-7) - system and virtual units
(special cases described in related manuals)
 - 1000 (8) - 8 binary inputs or outputs
(units OR-14, OS-28, OS-32, OS-34, XH-04)
 - 1001 (9) - 16 binary inputs or outputs
(units IB-36 to IB-47, IB-50, OR-15, OS-29, OS-30, OS-31, OS-33, OS-35)
 - 1010 (A)- 32 binary inputs or outputs
(units IB-48, IB-49, OS-26, OS-27, UX-52)
 - 1100 (C)- special units without initialisation table
(units IC-04)
 - 1101 (D)- analog units without own processor
(units IT-04, IT-12, IT-15, OT-04)
 - 1111 (F) -units with own processor
(units IT-06, OT-05, GT-40, IC-12 to IC-15, SC-11, CD-01 to CD-04, UP-01, UP-02)
- A11 - A8 - unit address in the frame (optional through the jumpers accessible on the side wall of the peripheral unit box)
- A7 - 0 - input address
1 - output address
unit with own processor (type F) do not have input and output addresses differentiated through this bit, their input and output zones are assigned dynamically according to the size required
- A6 - A0 - byte number within unit

Example of application for physical address

```
LD    %U$8100    ;direct load of states of eight inputs of unit XH-04
                    ;with address 1 in the frame
LD    %UW$9200   ;direct load of state of 16 inputs of unit of types
                    ;IB-36 to IB-47, IB-50 with address 2 in the frame
WR    %U$A083    ;direct write of value into 8 outputs of the third byte
                    ;of the unit of types OS-26, OS-27 with address 0 in
                    ;the frame
WR    %UW$9380   ;direct write of value into 16 outputs of units of
                    ;types OR-15, OS-29, OS-30, OS-31, OS-33, OS-35
                    ;with address 3 in the frame
```

Concrete addresses occupied by the peripheral units, if they are available, are given in the manuals for these units.

6. Direct input / output access

Attention! The C, D, E, F unit types usually require a defined way of operation that very often contradicts the use of the physical address. Therefore, **we do not recommend at all** to use the access to physical addresses of these units without prior careful reading of their functions! If the physical addresses in the description of these units are not expressly identified, then the units do not allow direct access!

6.1.2 Physical addresses in PLC TECOMAT TC400, TC500, TC600

Physical address structure

The physical address of binary and analog inputs and outputs has the following structure:

address upper byte								address low byte							
A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0

- A15 - A12 - unit type (fixed)
0xxx (0-7) - spare
1000 (8) - board with max. 8 binary inputs or outputs
1001 (9) - board with max. 16 binary inputs or outputs
1010 (A) - board with max. 24 binary inputs or outputs
1101 (D) - board with analog inputs and outputs
- A11 - A8 - always 0
- A7 - 0 - input address
1 - output address
- A6 - A0 - byte number within unit

Examples of application for physical addresses

```
LD    %UW$9000    ;direct load of input state
WR    %UW$9080    ;value direct write into outputs
```

Concrete addresses occupied by the peripheral units are listed in the manuals:
TECOMAT TC400 TXV 138 12.02 programmable controller handbook,
TECOMAT TC500 TXV 138 07.02 programmable controller handbook and
TECOMAT TC600 TXV 138 08.02 programmable controller handbook.

6.2. DIRECT ACCESS TO INPUTS AND OUTPUTS - MODEL 32 BITS

Systems TECOMAT TC650 and TC700 do not use physical addresses, but for a fast access to peripheral modules they have a specific RFRM instruction carrying out immediate data exchange between the central unit scratchpad and the corresponding peripheral module. Data is then handled by means of standard instructions working with the scratchpad. In this case automatic link to the scratchpad is ensured unlike the physical address. A peripheral module, to which we realize direct access, has to have the operation on (which is an essential difference against the systems using the U-operands).

It is true also in this case that the use of direct access is suitable only with time-critical reactions. An excessive use results in slowing the program performance down, since direct access to particular peripheral modules is time-consuming. It is necessary to realize that immediate data update of a peripheral module breaches the principle of the constancy of input and output data during the cycle and increases hazard risks. The use

should be restricted exclusively for such cases when immediate response has to be ensured, for example for treatment of emergency situations, in the processes of interrupt operation, etc.

7. OTHER ADDRESS SPACES

7.1. D DATA

The D data has the meaning of the constants of the user program. It is part of the user program and for the user program is accessible only for reading. The D data can be entered and changed only when editing the user program.

The D data can be advantageously used as parameters modifying the user program. For a certain class of control algorithms, just one user program can be created and debugged which is adopted to the real conditions by entering the corresponding parameters in the D zone before particular use.

Similarly, the user program can be adopted to the changing requirements of the user, technology changes, assortment changes, etc. In the D data continuous data sequences can also be saved, such as tables or patterns for setting the scratchpad and outputs in key situations.

7.2. T TABLES

T tables are part of the user program. They can be used advantageously as parameters modifying the user program. For a certain class of control algorithms, just one user program can be created and debugged which is adopted to the real conditions by entering the corresponding parameters in the T table. Similarly, the user program can be adopted to the changing requirements of the user, technology changes, assortment changes, etc.

The T table are accessible only through special instructions that refer to the T address space (table instructions, block transmission instructions). They have a prescribed structure - this is always a sequence of values of the same width (1, 8, 16, 32 bits) with additional data on the length of the sequences in bytes. Each item (each value of the sequence) is assigned an ordinal number - index. The lowest item has zero index, the index of the last item is called "limit" (number of items = limit + 1). The T tables thus can be processed as a sequence of the values of arbitrary type excepting lreal (figure 7.1, figure 7.2, figure 7.3, figure 7.4). These types can be directly determined by the type of instruction used, they do not depend on the table. Physically the T tables are saved as a sequence of values with the information on their length in bytes. For bit processing of the table the last bits not used are always added zeros to the whole byte. The bit table must start at bit position 0 (it cannot start in the middle of byte).

The following operations can be carried out over the T tables:

- ◆ item selection to the index specified (LTB)
- ◆ item write based on index specified (WTB)
- ◆ find index to the value of the item specified (FTB, FTBN)
- ◆ find index to the part of item specified (FTM, FTMN)
- ◆ find class item - classification of the value specified into one of the classes (groups) that are determined by the table by the limit in the value sequence (FTS, FTSF, FTSS)
- ◆ data block transmission between the table and scratchpad (SRC, MOV, MTN, MNT)
- ◆ working with structured table (LDS, WRS, FIT, FNT)

The structure of table instructions enables easily to realize also very complex functions with the solution being more economical and faster than traditional solutions (sometimes

very significantly), but it always is more flexible. The resulting program is understandable and well-ordered.

Table quantity and size limitations

The central units of series E use the T tables solely for initialisation of peripheral units. The central units of series M can have 256 tables at most with a length of 255 bytes. As a consequence of this, 127 items of size word (254 bytes), 255 items of size byte and 256 items of size bit (32 bytes) are processible as maximum through the table instructions.

The central units of series S can have 256 tables at most. The length of the tables is not limited.

For the central units of series B, C and D only the memory size of the user program is the only limiting factor.

Table examples

index =	7	6	5	4	3	2	1	0	
	↓	↓	↓	↓	↓	↓	↓	↓	
	1	1	1	1	1	0	0	1	byte nr.0
15 →	1	0	1	0	1	1	1	0	← 8 byte nr.1
23 →	1	1	0	0	0	1	1	1	← 16 byte nr.2
31 →	0	1	1	1	0	0	0	0	← 24 byte nr.3

	0	0	0	0	1	0	0	0	byte nr.7
					↑	↑	↑	↑	
bit limit = 55					51	50	49	48	
	.7	.6	.5	.4	.3	.2	.1	.0	numbers of bit in byte

Figure 7.1 Example of bool type table (the last four bits are added zeros to the whole byte)

index = 0	156	byte nr.0
1	255	byte nr.1
2	147	byte nr.2
3	64	byte nr.3
:
21	0	byte nr.21
limit = 22	235	byte nr.22

Figure 7.2 Example of table of byte, usint, sint types

	item upper byte	item low byte	
index = 0	\$00	\$55	byte nr.1 and 0
1	\$12	\$FF	byte nr.3 and 2
2	\$01	\$47	byte nr.5 and 4
3	\$15	\$40	byte nr.7 and 6
:	
14	\$00	\$00	byte nr.29 and 28
limit = 15	\$00	\$35	byte nr.31 and 30

Figure 7.3 Example of table of word, uint, int types

7. Other address spaces

	highest byte of the item		lowest byte of the item		
index = 0	\$15	\$12	\$FF	\$55	byte nr.3 to 0
1	\$47	\$11	\$08	\$00	byte nr.7 to 4
2	\$00	\$40	\$56	\$00	byte nr.11 to 8
3	\$00	AB	\$0C	\$40	byte nr.15 to 12
:
8	\$48	\$D8	\$15	\$08	byte nr.31 to 28
limit = 9	\$07	\$35	\$20	\$04	byte nr.35 to 32

Figure 7.4 Example of table of dword, udint, dint and real types

7.3. DATABOX ADDITIONAL DATA MEMORY

DataBox is a additional memory designed to work higher amount of data, such as archiving of data on the controlled process for a longer period of time, etc. In central units it is realized either through an additional sub-module, or is part of standard equipment. DataBox is a CMOS RAM memory, backed up through a battery from the central unit. Data can be written to the memory or read either by the user program of the PLC or via serial line. In the Mosaic development environment the DataBox memory is accessible in the same way as the scratchpad memory, thus it can be edited also manually.

Serial communication with DataBox memory

For serial communication, any serial channel working in PC mode can be employed. The program enabling to load the data from the DataBox memory into a file or write the data from a file into the DataBox memory is called *complc32.exe* (Windows 2000, XP), or *complc.exe* (DOS, Windows 95, 98). It also provides a possibility to test the size of the memory accessible as DataBox. Programs *complc32.exe* and *complc.exe* are part of the installation of the Mosaic development environment.

DataBox instructions

The user has three instructions available to work with the DataBox memory. Instruction RDB for loading data from DataBox into registers R, instruction WDB to write data from registers R into DataBox and instruction IDB to identify the size of DataBox memory.

IDB instruction function

To detect the size of the occupied DataBox memory, the IDB instruction is used. This instruction does not require any input parameters. After execution it increases the user stack by one level and the size of the DataBox memory in kB is written on the stack top, i.e. the value of 128, 512 etc. If DataBox memory is not found, the instruction resets the value of 0.

During simulation in the Mosaic development environment, the IDB instruction creates file *databox.**** in the project folder (if it already exists). The size of the file we want to create is passed in layer A0 of the active stack in KB. The newly created file is the binary one and filled with zeros. Instruction IDB increases the user stack by one level and gives back the required size of DataBox memory in the following situations:

- ◆ file *databox.**** was successfully created

- ◆ file databox.\$\$\$ already exists and has required size (or is even bigger)

In the following cases, the IDB instruction will return the value of 0 at A0:

- ◆ file databox.\$\$\$ was not created (e.g. not enough space on the disk)
- ◆ file databox.\$\$\$ exists, but an error occurred during file opening (e.g. file structure failure, a defective sector reported by the disk controller, etc.)
- ◆ file databox.\$\$\$ exists, but its size is smaller than required, - in this case the content of the databox.\$\$\$ file remains unchanged. To create a larger databox.\$\$\$ file for simulation, it is necessary to delete the original file or move it into another directory.

As a consequence of the above is that during simulation, instruction IDB works with file databox.\$\$\$ in the project directory. The size of the created file corresponds to the request that will be filled into layer A0 before instruction call. In a real PLC, the size of the DataBox memory is determined by the size of the memory being fitted, which means it can be bigger than the request. The example at the end of this chapter resolves this situation through unsharp inequality so, it can be used both for simulation and a real PLC. Detection of the DataBox memory size is carried out typically in the process for warm or cold restart.

Parameter zone structure for RDB and WDB

Before calling the RDB a WDB instructions it is necessary to set a couple of parameters. These parameters are located in the R registers and must be saved closely behind each other and their order must be observed. The register number where the first parameter is saved is passed on the stack when instructions RDB and WDB are called (see the following text). The parameters are ordered as follows:

Parameter name	Type	Description
adrDB	udint	address in DataBox memory
indR	uint	initial register index in scratchpad
len	usint	number of bytes transferred

The parameters in the program can be defined symbolically with automatic register assignment

```
#reg udint   adrDB
#reg uint    indR
#reg usint   len
```

For programming, the best definition is through the *#struct* directive, since it automatically ensures requirements for succession of parameters:

```
#struct parDB           ;structure name
    udint adrDB,        ;address in DataBox
    uint  indR,         ;initial register index in scratchpad
    usint len           ;number of bytes transferred
;
#reg parDB parusi
```

Function of RDB and WDB instructions

By execution of instructions RDB and WDB, the level of the user stack does not change. On the stack top it gives back the quantity of really transmitted data. At the same time, the instructions set the content of system register S1.0 with the following meaning:

S1.0 = log.1 - input parameters OK, the result is valid
 S1.0 = log.0 - input parameters out of range, the result is invalid

7. Other address spaces

If S1.0 = log.0, no data is transmitted and at the same time the content of register S34 is set with the following meaning:

S34 = \$14 - data source block was defined out of range

S34 = \$15 - data target block was defined out of range

Based on the size of DataBox memory, the address space is available, see table 7.1.

Table 7.1 Address space available of particular DataBox memories

DataBox memory size	Address space available
128 kB (CPU series D and S)	0 - \$01FFFB
128 kB (CPU series B and C)	0 - \$01FFFF
512 kB (CPU series D and S)	0 - \$07FFEF
1.5 MB (CPU series B)	0 - \$17FFFF
3.0 MB (CPU series C)	0 - \$2FFFFFF

When attempting to load or write out of this available space S1.0 is set to log.0 and at the same time, corresponding error code is set at S34. During simulation in the user program, instructions RDB and WDB execute loading or writing as the case may be into file databox.\$\$. If it is not possible to open this file or an error occurs during loading or writing into this file, the instruction set S1.0 = log.0 as well as A0 = 0 during the simulation.

Example of application

As an example, instructions RDB, WDB and IDB can be used in the user program as follows:

```
#reg uint 100,adrDB      ;variables controlling RDB activity
#reg uint  indR
#reg uint  len
#reg bool  DataBoxOK     ;DataBox flag OK
;
P 63
:
LD 32                    ;required size of DataBox in application
IDB                               ;DataBox size identification
GT
NEG                               ;DataBox of size required at least?
WR DataBoxOK              ;set flag
:
E 63
;
P 0
:
LD DataBoxOK              ;DataBox OK ?
JMC endDBX                ;no
LD $FC00                  ;address in DataBox (32 bit model)
;!! LDL $FC00              ;address in DataBox (16 bit model - long !!!)
WR adrDB
LD 200                    ;to which register
WR indR                   ;data from DataBox are transferred
LD 56                    ;number of bytes transferred
WR len
LD 100                    ;register number where parameters are stored
RDB                       ;reading data block from DataBox
                           ;to the scratchpad, block
                           ;56 bytes long is read from address $FC00
                           ;and saved from R200
```

```

endDBX:
    :
E 0
or better
#struct parDB          ;structure name
    udint adrDB,      ;address in DataBox
    uint  indR,      ;initial register index in scratchpad
    usint len        ;number of bytes transferred
;
#reg parDB parusi
#def lenDat 56
#reg usint  blockDat[lenDat]
#reg bool   DataBoxOK      ;DataBox flag OK
;
P 63
    :
    LD 32          ;required size of DataBox in application
    IDB          ;DataBox size identification
    GT
    NEG          ;DataBox of size required at least?
    WR DataBoxOK      ;set flag
    :
E 63
;
P 0
    :
    LD DataBoxOK      ;DataBox OK?
    JMC endDBX        ;no
    LD $FC00          ;address in DataBox (32 bit model)
;!! LDL $FC00        ;address in DataBox (16 bit model - long !!!)
    WR parusi~adrDB
    LD __indx (blockDat);to which register data
                        ;from DataBox
    WR parusi~indR
    LD lenDat         ;number of bytes transferred
    WR parusi~len
    LD __indx (parusi) ;register number where parameters are stored
    RDB              ;reading data block from DataBox to the
                    ;scratchpad block
                    ;56 bytes long is read from address $FC00
                    ;and saved in the blockDat field

endDBX:
    :
E 0

```

User instructions

If older types of central units are available that do not have instructions RDB, WDB and IDB implemented, user instructions READDDBX, WRITEDBX and SIZEDBX can be used (user instruction definition see chapter 12). Their function is identical to the corresponding instructions including simulation in the Mosaic development environment.

Definition of user instructions

The user instructions are necessary to be defined at the beginning of the user program:

7. Other address spaces

```
#usi u_readdbx = readdbx      ;file path and name
#usi u_writedbxx = writedbxx  ;file path and name
#usi u_sizedbx = sizedbx     ;file path and name
#def RDB usi u_readdbx       ;USI naming
#def WDB usi u_writedbxx     ;USI naming
#def IDB usi u_sizedbx      ;USI naming
```

These 6 lines will be inserted into the user program immediately at the beginning of the program. By doing this we have added the missing information to the PLC. The rest of the user program remains unchanged.

8. RESULT STACK

Stack model

During execution of the user program the PLC works with a stack having 8 levels numbered A0 to A7 (an accumulator). Active level A0 also called the stack top is used for the most of instructions. TECOMAT PLCs have two stack models that differ from each other by the width of one layer. Series B, D, E, M and S have particular stack layers 16 bits wide (see figure 8.1), while series C has the stack layers 32 bits wide (see figure 8.2). This results in some differences in behaviour of particular models.

The stack is used by logic operations, arithmetic operations, transmission operations, and also logic and numeric parameters of more complex instructions and subroutines are passed on.

The stack is cyclic (see figure 8.3), we can imagine it as a drum memory as you can see on figure 8.4.

A set of eight switchable stacks

In total 8 stacks are available (except central units of series E, that has only one stack) marked A to H. Only one stack is active at a time and it is possible to switch among them (see figure 8.4). This gives a lot of opportunities in the field of parameter transmission among the functions in the user program without necessity to use a worse arranged saving of intermediate parameters in the scratchpad.

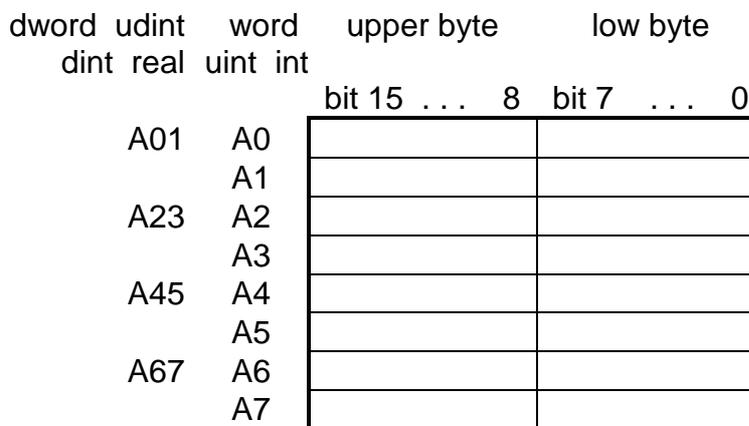


Figure 8.1 Schematic diagram of A stack structure of 16 bit model

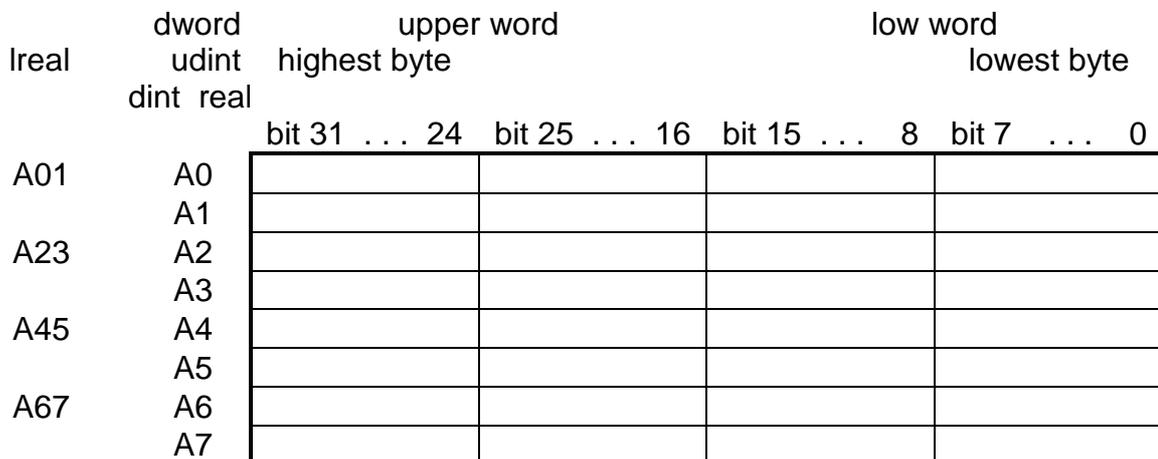


Figure 8.2 Schematic diagram of A stack structure of 32 bit model

8. Result stack

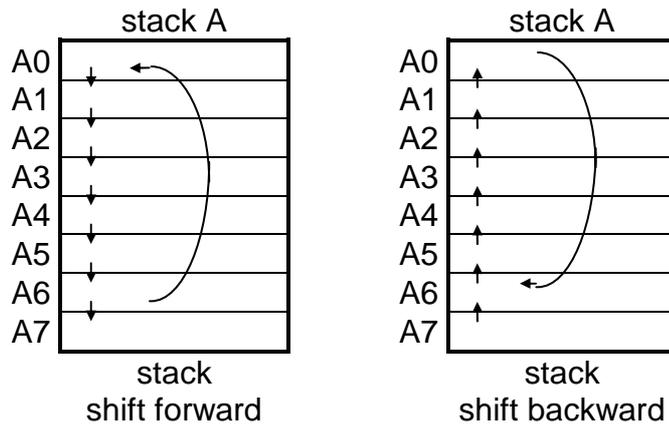


Figure 8.3 Function of cyclic stack A

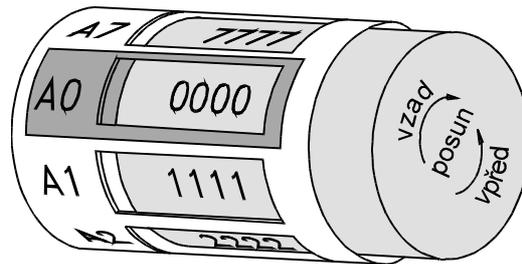


Figure 8.4 Stack A as a drum memory

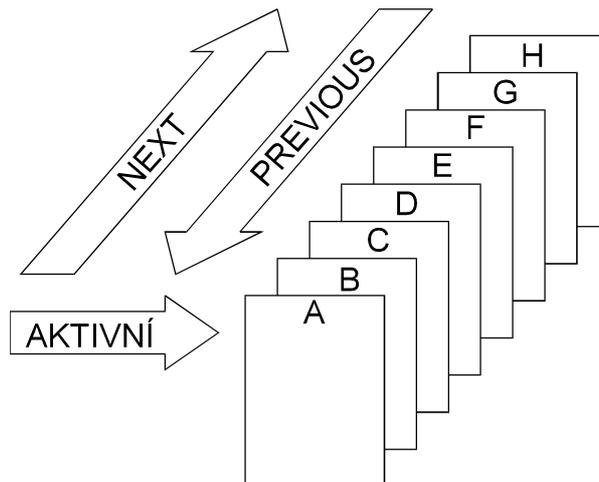


Figure 8.5 Stack switching

8.1. STACK STRUCTURE

Stack top

The stack top (A0 or A01 as the case may be) is the active layer having the meaning of an accumulator. Other layers (A1 to A7, or A23 to A67 as the case may be) contain gradually the sequence of the previous values of the stack top.

Stack shift forward

The stack shift forward is executed the read instructions (LD, LDC, etc.) and some more complex instructions. After each stack shift forward by one level, the values off all of its layers A0 to A6 are moved to the layers with numbers higher by one and the A0 stack top is occupied by a new value according to the following procedure:

A0 ← new data
 A1 ← original content A0
 A2 ← original content A1

 A7 ← original content A6
 Original content A7 is lost irrecoverably (it is overwritten by the new content of A0).

Stack shift backward

The stack shift backward is executed by the POP instruction and operand-free instructions of arithmetic and logic operations. The backward shift is executed by the following procedure:

A0 ← original content A1 or operation result between A0 and A1
 A1 ← original content A2
 A2 ← original content A3

 A7 ← original content A0

8.2. DATA INTERPRETATION ON THE STACK - 16 BIT MODEL

Each stack layer has the width of 16 bits (2 bytes). The entire layer is marked A0, A1, ... A7 (see figure 8.1). The data of dword, udint, dint and real types occupy two layers. In this case we talk about a double-layer numbered A01, A23, A45, A67.

8.2.1. Data of bool type - 16 bit model

If we load bool type data on the stack top (LD %R0.1, for example), then the bit value (0 or 1) will be saved on all sixteen bits of layer A0 (A0 = 0 or A0 = 65 535).

If we save bool type data (WR %R8.5, for example), then in case of the zero value of A0 we write the zero value of the bit, in case of any non-zero value of A0 we write one. Thus we save longitudinal logical add (OR operation) of all sixteen bits of stack A0.

This principle enables an easy format conversion, it is possible to combine instructions with operands of bool, byte, usint, sint, word, uint and int types almost without any limitation.

When talking about content of the stack top as the bool type value, we will use the following specification:

log.0 logic zero, A0 = 0
 log.1 logic one, A0 ≠ 0

The stack top is understood as the entire layer A0.

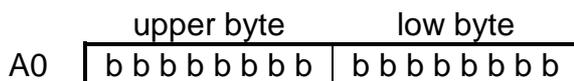


Figure 8.6 Bool type data saving on the stack top
 b - bit logic value (log.0 or log.1)

8.2.2. Data of byte, usint, sint types - 16 bit model

If we load byte, usint or sint types data on the stack top (LD %R0, for example), then the value of this byte will be saved in the low byte of layer A0, the upper byte of layer A0 will be set to zero.

If we write byte, usint or sint types data (WR %R8, for example), then only the low byte of layer A0 will be saved in the target address. The upper byte will be ignored.

This principle enables an easy format conversion for positive values, it is possible to combine instructions with operands of bool, byte, usint, word and uint types almost without any limitation. For negative values (types sint, int) it is necessary to treat sign transmission.

The data assumes values of 0 to 255 (byte, usint) or -128 to 127 (sint) when using the highest bit as sign.

The stack top is understood the entire layer A0.



Figure 8.7 Byte, usint, sint types data saving on the stack top
B - byte value

8.2.3. Data of word, uint, int types - 16 bit model

Data of word, uint, int types occupy 2 bytes.

If we load data of word, uint, int types on the stack top (LD %RW0, for example), the entire layer A0 will be filled with the data. The content of the register with the lower number will be saved in the low byte of layer A0, the content of the register with the higher number will be saved in the upper byte of layer A0.

If we write data of word, uint, int types (WR %RW8, for example), the entire content of layer A0 will be saved in the target address. The content of the low byte of layer A0 will be saved in the register with the lower number, the constant of the upper byte of layer A0 will be saved in the register with the higher number.

The data assumes values of 0 to 65 535 (word, uint) or -32 768 až +32 767 (int) when using the highest bit as sign.

The stack top is understood the entire layer A0.

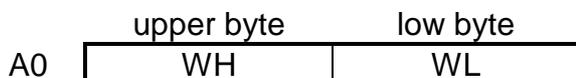


Figure 8.8 Word, uint, int types data saving on the stack top
WH - higher byte value
WL - lower byte value

8.2.4. Data of dword, udint, dint types - 16 bit model

Data of dword, udint, dint types occupy 4 bytes.

If we load data of dword, udint, dint types on the stack top (LD %RL0, for example), then the entire layers A0 and A1 are filled with the data that together form double-layer A01. The content of the register with the lowest number will be saved in the low byte of layer A0, the content of the register with the highest number will be saved in the upper byte of layer A1.

If we write data of dword, udint, dint types (WR %RL8, for example), the entire content of double-layer A01 will be saved in the target address. The content of the low byte of layer A0 will be saved in the register with the lowest number, the content of the upper byte of layer A1 will be saved in the register with the highest number.

The data can assume values of 0 to 4 294 967 295 (dword, udint) or -2 147 483 648 to +2 147 483 647 (dint) when using the highest bit as sign.

The stack top is the entire double-layer A01.

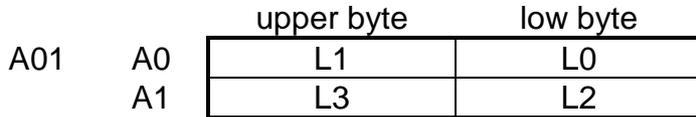


Figure 8.9 Dword, udint, dint types data saving on the stack top

L0 - lowest byte value

:

L3 - highest byte value

8.2.5. Data of real type - 16 bit model

Data of real type is 4 bytes wide, which is the same width as the dword type data has. The same principles as mentioned above are valid for them.

According to IEEE-754 real type data can assume values of approx. $\pm 1,175494 \times 10^{-38}$ to $\pm 3,402823 \times 10^{38}$. The following four values specify their state:

- \$7FFFFFFF - invalid number (NaN - not a number)
- \$FFFFFFFF - invalid number (NaN - not a number)
- \$7F800000 - exceeding the range of positive numbers (+INF)
- \$FF800000 - exceeding the range of negative numbers (-INF)

The stack top is the entire double-layer A01.

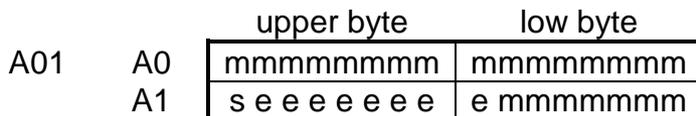


Figure 8.10 Real type data saving at the stack top

s - sign (1 bit)

e - exponent (8 bits)

m - mantissa (23 bits)

8.3. DATA INTERPRETATION AT THE STACK - 32 BIT MODEL

Each stack layer is 32 bits wide (4 bytes). The entire layer is labelled A0, A1, ... A7 (see figure 8.2). Lreal types data occupy two layers. Then we talk about so called "double-layer" labelled A01, A23, A45, A67.

8.3.1. Data of bool type - 32 bit model

If we load data of bool type on the stack top (LD %R0.1, for example), then the bit value (0 or 1) will be saved in all 32 bits of layer A0 (A0 = 0 or A0 = 4 294 967 295).

8. Result stack

If we write data of bool type (WR %R8.5, for example), then in case of zero value of A0 zero value of the bit will be saved, in case of any non-zero value of A0, number one will be saved. Thus we write longitudinal logical add (OR) of all 32 bits of stack top A0.

This principle enables an easy format conversion, it is possible to combine instructions with operands of bool, byte, usint, sint, word, uint, int, dword, udint and dint types almost without any limitation.

When talking about content of the stack top as the bool type value, we will use the following specification:

log.0 logic zero, A0 = 0
log.1 logic one, A0 ≠ 0

The stack top is understood the entire layer A0.

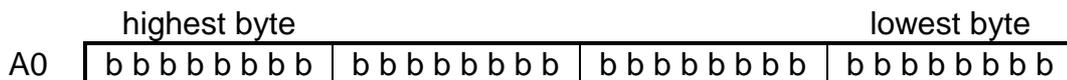


Figure 8.11 Bool type data saving on the stack top
b - bit logic value (log.0 or log.1)

8.3.2. Data of byte, usint, sint types - 32 bit model

If we load data of byte, usint, sint types on the stack top (LD %R0, for example), then the value of this type will be saved in the lowest byte of layer A0, other three bytes of layer A0 will be set to zero.

If we write data of byte, usint, sint types (WR %R8, for example), then the lowest byte of layer A0 will be saved to the target address. Other bytes will be ignored.

This principle enables an easy format conversion for positive values, it is possible to combine instructions with operands of bool, byte, usint, word, uint, dword and udint types almost without any limitation. For negative values (types sint, int, dint) it is necessary to treat sign transmission.

The data assumes values of 0 to 255 (byte, usint) or -128 to 127 (sint) when using the highest bit as sign.

The stack top is understood the entire layer A0.

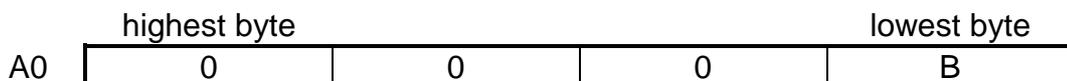


Figure 8.12 Byte, usint and sint types data saving on the stack top
B - byte value

8.3.3. Data of word, uint, int types - 32 bit model

Data of word, uint, int types occupies 2 bytes.

If we load data of word, uint, int types on the stack top (LD %RW0, for example), then the value of this word will be saved in the lower word of layer A0, the upper word of layer A0 will be set to zero. The content of the register with the lower number will be saved in the lowest byte of layer A0, the content of the register with the higher number will be saved in the second lowest byte of layer A0.

If we write data of word, uint, int types (WR %R8, for example), then only the low word of layer A0 will be saved in the target address. The upper word will be ignored. The content of the lowest byte of layer A0 will be saved in the register with the lower number,

the content of the second lowest byte of layer A0 will be saved in the register with the higher number.

This principle enables an easy format conversion for positive values, it is possible to combine instructions with operands of bool, byte, usint, word, uint, dword a uint types almost without any limitation. For negative values (types sint, int, dint) it is necessary to treat sign transmission.

The data assumes values of 0 to 65 535 (word, uint) or -32 768 až +32 767 (int) when using the highest bit as sign.

The stack top is understood the entire layer A0.

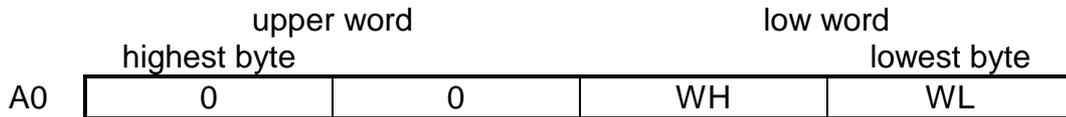


Figure 8.13 Word, uint and int types data saving on the stack top
 WH - higher byte value
 WL - lower byte value

8.3.4. Data of dword, uint, dint types - 32 bit model

Data of dword, uint, dint types occupy 4 bytes.

If we load data of dword, uint, dint types on the stack top (LD %RL0, for example), the entire layer A0 will be filled with the data. The content of the register with the lowest number will be saved in the lowest byte of layer A0, the content of the register with the highest number will be saved in the highest byte of layer A0.

If we write data of dword, uint, dint types (WR %RL8, for example), then the entire content of layer A0 will be saved in the target address. The content of the lowest byte of layer A0 will be saved in the register with the lowest number, the content of the highest byte of layer A0 will be saved in the register with the highest number.

The data can assume values of 0 to 4 294 967 295 (dword, uint) or -2 147 483 648 to +2 147 483 647 (dint) when using the highest bit as sign.

The stack top is understood the entire layer A0.

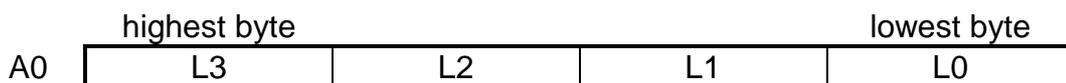


Figure 8.14 Dword, uint, dint types data saving on the stack top
 L0 - lowest byte value
 :
 L3 - highest byte value

8.3.5. Data of real type - 32 bit model

The data of real type is 4 bytes wide, having the same length as dword type data. The same principles as mentioned above are valid for them.

According to IEEE-754 real type data can assume values of approx. $\pm 1,175494 \times 10^{-38}$ to $\pm 3,402823 \times 10^{38}$. The following four values specify their state:

- \$7FFFFFFF - invalid number (NaN - not a number)
- \$FFFFFFFF - invalid number (NaN - not a number)
- \$7F800000 - exceeding the range of positive numbers (+INF)
- \$FF800000 - exceeding the range of negative numbers (-INF)

8. Result stack

The stack top is understood the entire layer A0.

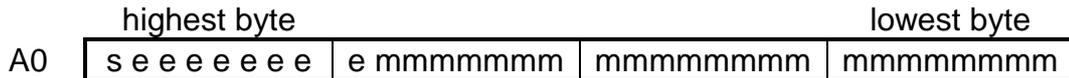


Figure 8.15 Real types data saving at the stack top

s - sign (1 bit)
e - exponent (8 bits)
m - mantissa (23 bits)

8.3.6. Data of lreal type - 32 bit model

The data of lreal type is 8 bytes wide.

If we load data of lreal type on the stack top (LD %RD0, for example), then two layers A0 and A1 are fully filled with the data. These two layers form a double-layer A01. The content of the register with the lowest number will be saved in the low byte of layer A0, the content of register with the highest number will be saved in the upper byte of layer A1.

If we write data of lreal type (WR %RD8, for example), the entire content of double-layer A01 will be saved in the target address. The content of the low byte of layer A0 will be saved in the register with the lowest number, the content of the upper byte of layer A1 will be saved in the register with the highest number.

According to IEEE-754, data of lreal type can assume values of approx. $\pm 2,2 \times 10^{-308}$ to $\pm 1,8 \times 10^{308}$. The following four values specify their state:

\$7FFFFFFF FFFFFFFF	- invalid number (NaN - not a number)
\$FFFFFFFF FFFFFFFF	- invalid number (NaN - not a number)
\$7FF00000 00000000	- exceeding the range of positive numbers (+INF)
\$FFF00000 00000000	- exceeding the range of negative numbers (-INF)

The stack top is the entire double-layer A01.

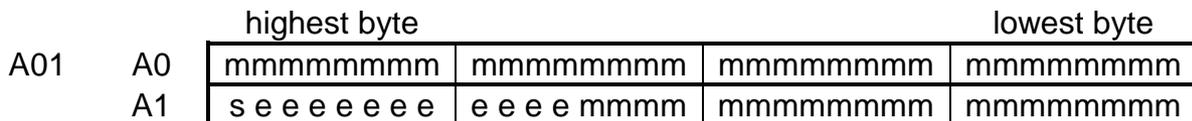


Figure 8.16 Lreal type data saving on the stack top

s - sign (1 bit)
e - exponent (11 bits)
m - mantissa (52 bits)

8.4. SWITCHING AMONG STACKS

There are 8 stacks available, marked A, B, C, D, E, F, G, H, each of them having such a structure as described for stack A in chapter 8.1. Only one stack is active at a time, the other stacks are hold at the state at which they were when leaving the user program and no state of the user program can effect them. For example, when opening a new user process, only the active stack will be set to zero (see chapter 10).

After switching on or restarting the PLC, all stack are set to zero and only the A stack is active. After cycle I/O scan, the A stack is always set to zero and set as the active one.

Switching among the stack is possible at any time, which means that the user program leaves the current active stack and works with the newly selected stack. At the same time, also the state of flag registers S0 and S1 will be backed up.

Instructions for stack switching

For switching the stacks, the following four instructions are used:

- NXT - Activation of next stack in the queue (figure 8.5)
for example if stack C was active, stack D will be activated now
- PRV - Activation of previous stack in the queue (figure 8.5)
for example if stack C was active, stack B will be activated now
- CHG - activation of any stack
for example, CHG 7 activates stack F, the state of flag registers S0 and S1 is not changed,
- CHGS- activation of any stack with backing-up S0 and S1 registers
for example CHGS 7 writes the state of system registers S0 and S1 to the stack being still active and activates stack F, in system registers S0 and S1 it refreshes the state corresponding to their state at the moment, when the last active stack F was left by means of instruction CHGS n, where n is 0 to 7 (stands for A to H), or by means of instructions NXT and PRV.

9. COMPILER DIRECTIVES

Directives are commands for the compiler enabling to control compilation optimally. All directives begin with character #. At the xPRO compiler containing the Mosaic development environment, the following directives are available:

#program
#unit, #module
#include, #usefile
#def
#reg, #rem
#data, #table
#struct
#if, #elif, #else, #endif
#ifdef, #ifndef
#macro, #endm
#label
#usi
#mnemo, #mnemoend
#useoption

9.1. #program

The *#program* directive specifies program start.

The syntax is as follows:

#program name [, [V][xyz]]

<i>name</i>	- program name, maximum 16 characters (a longer name will be shorted accordingly)
<i>V</i>	- optional prefix program version
<i>xyz</i>	- three optional characters characterizing program version

In the Mosaic development environment this directive is not entered manually by the user, but the user can use a form in folder *Sw / Program* where he specifies the program name, its version and other information on application and history of changes to the user program. This information is part of the project and are handy especially when you come to this user program later on. If the user does not fill this form, the default name will correspond to the name of the project within the project group and the version is 1.0. Based on these information, the compiler itself then generates a program heading that is in control file *xxx.mak*, where *xxx* is the project name within the project group.

The information on the name and version of the user program being executed by the PLC can be read any time, if we select option *PLC / Get PLC info...* For that reason, we recommend to use the program version in such a way that with each change of the program its version will also be changed. A very frequent problem in applications is to find out which program version is loaded in the PLC, therefore conscionable numbering of version avoids problems during later program modifications.

9.2. #unit, #module

By means of directives *#unit* or *#module* a list of peripheral units is created necessary for a proper function of the user program. Based on configuration identified PLC configuration is created by the compiler (see PLC manuals).

The syntax is as follows:

```
#unit rack,address,type,image_X,image_Y, activation[,tab]
#module TModule1 version, rack, address, location, length_X, length_Y,
__offset (image_X), __offset (image_Y), __indx (tab)
```

<i>version</i>	- number of description variant (always 1)
<i>rack</i>	- rack number in which the module is installed
<i>address</i>	- module address in the frame (can be set through jumpers or determined by position)
<i>location</i>	- numeric code denoting location of described module part
<i>type</i>	- numeric code characterizing module type This numeric code is not used directly, but through symbolic names defined standardly in file <i>xpro.sys</i> .
<i>length_X</i>	- length of defined module input data
<i>length_Y</i>	- length of defined module output data
<i>image_X</i>	- operand (does not have to be necessarily in area X) specifying to which registers data is transmitted from the inputs of the module defined
<i>image_Y</i>	- operand (does not have to be necessarily in area Y) specifying from which registers data is transmitted to the outputs of the module defined
<i>activation</i>	- code (again replaced by the symbolic name specified in file <i>xpro.sys</i>) informing on mode It can assume the following values: <i>X_on</i> = inputs permitted (copied to <i>image_X</i>) <i>Y_on</i> = outputs permitted (copied from <i>image_Y</i>) <i>On</i> = inputs and outputs permitted <i>Off</i> = inputs and outputs prohibited
<i>tab</i>	- optional parameter specifying the name of table (see <i>#table</i>) with data necessary for unit initialisation This initialisation is used by some special types of peripheral modules, system serial channels, etc.

In the Mosaic development environment, the user does not define peripheral units through manual entry of the *#unit* directive, but by filing in the tables in the configuration section (in the Project manager, item *Hw | HW Configuration*). Based on these instructions, the compiler then creates an individual file *xxx.hwc*, where *xxx* is the name of the project containing both the *#unit* directive and corresponding initialisation tables. This file is automatically attached to the user program by means of the *#usefile* directive (see following text) in the control file *xxx.mak*. If you check option *Suppress directive #UNIT* in the configuration section, the *#unit* directives will be only commentaries in the file (beginning with semi-colon character).

PLC TECOMAT TC650 and TC700 use *#module* directive instead of *#unit* directive.

In the Mosaic environment it is possible to read information from a connected PLC according to which configuration is automatically filled. This data can be modified by the user as needed and based on this data, configuration file *xxx.hwc* will be created during compilation.

9.3. #include, #usefile

When writing long programs, it is advantageous to arrange the entire program into smaller, logically compact files. This so-called file threading enables directive *#include*.

The syntax is as follows:

```
#include file_name
```

file_name - identifies the file that is necessary to connect to the program being compiled. If it is not found in the current working directory, it is necessary to specify the path to the file.

When the compiler finds the *#include* directive in the program source text, it interrupts compilation in the current file and continues compilation of the file specified after the *#include* directive. After compilation of this entire file it continues compilation of the text from the original file. The compiler has the capability of resolving such situations when a file inserted by the *#include* directive contains other *#include* directives.

In the Mosaic development environment it is more advantageous to use file threading by means of the project. Each file contained in the project is inserted into the control file *xxx.mak* in the given order by means of the *#usefile* directive. This directive is automatically generated by the compiler and is not intended to be used by the user. If we add another file into the project it will be automatically connected in the control file during compilation. The sequence of file compilation is determined by their sequence in the list and this can be changed by means of move-arrows.

The syntax is as follows:

```
#usefile file_name
```

file_name - identifies a file that is inserted into the project being compiled

In the Mosaic development environment, file threading is facilitated because the compiler supports so called

In the Mosaic development environment, file threading is facilitated by the compiler supporting so called "process assembling". This means, if we write instructions within the frame of process P 0 (beginning with instruction P 0 and ending with instruction E 0) in one file, and in another file we again write instructions within the frame of process P 0 (we again begin with instruction P 0 and end with instruction E 0), the compiler creates one process P 0 in which both parts are merged (it does not use excess instructions E 0 and P 0) in the sequence as they were ordered in the project.

But for this reason we do not recommend using of the EC and ED instructions, which cause premature process termination, this is to say that such parts of the program will not be executed that are connected by the compiler (this can include also important compilers generated by the Mosaic environment). A jump to the end of a process within one file is solved by a jump to a label.

9.4. #def

By means of directive *#def* symbolic names are defined, that can be used in the program instead of absolute operands, for example.

The syntax is as follows:

```
#def symbolic_name substitute_name
```

symbolic_name - a sequence consisting of arbitrary number of permitted characters (see directive *#program*)

substitute_name - another arbitrary string consisting of permitted characters (an absolute operand, a constant, etc.)

Directive *#def* can be used anywhere in the program after its header, but always before first use of the symbolic name defined. A general principle is applied that says to define first and then it is possible to use the object defined.

Example 9.1

```
#def Start_Bit      %R5.0      ;bit definition
#def CONSTANT      80         ;delay
#def PAUSE          CONSTANT+20
#def sec           2          ;type of timer 1 s
#def Timer_T1      %RW0.sec   ;timer 1 s
#def output_word   %YW0      ;output
```

9.5. #reg, #rem

Directive *#reg* is used for automatic definition of variables from area R. By means of this directive we assign a symbolic name to the variable and at the same time, we inform the compiler to reserve this variable in registers R of the PLC scratchpad memory. The number of registers reserved for the variable is determined by the data type required.

Directive *#rem* is used for automatic definition of variables from remanent zone R. By means of this directive, we assign a symbolic name to the variable and at the same time, we inform the compiler to reserve this variable in remanent registers R of the PLC scratchpad memory. The number of registers reserved for the variable is determined by the data type required.

If the remanent zone is too small for the variable, the compiler will warn you using an error report and offers you automatic change of the size of the remanent zone. If the size of the remanent zone still was set to its maximum, then the repeated compilation will be in order.

The syntax is as follows:

```
#reg [public] [aligned] type [index,]
                                var_name[[len_array]][,next_var...]
#rem [public] [aligned] type [index,]
                                var_name[[len_array]][,next_var...]
```

public - an optional command to generate into the list of public variables for visualizations in the project manager in folder *Sw / Compiler* option *Generate file / PUBLIC*

aligned - an optional command to assign the variables to the even register or to the bit 0

type - specifies data type of the variables assigned to the symbolic name *var_name*
Permitted types of assigned variables are listed in table 9.1.

var_name, next_var - symbolic names of defined registers

len_array - an optional parameter closed in square brackets [], it specifies the size of the field in the format corresponding to parameter *typ*

index - an optional parameter, it specifies the register number assigned to symbolic name *var_name* in area R

9. Compiler directives

Table 9.1 Permitted types of variables assigned

Type	Range	Example
bool	R0.0 - Rmax.7*	R10.2, R2.15
byte, usint, sint	R0 - Rmax*	R1, R2001
word, uint, int	R0 - Rmax-1*	RW2, RW2002
dword, udint, dint	R0 - Rmax-4*	RL4, RL2004
real	R0 - Rmax-4*	RF8, RF2008
lreal	R0 - Rmax-8*	RD12, RD2012
structure**		

* Rmax is determined by the series of the central unit used.

** Structure is the name (tag) of the structure defined by means of directive *#struct*.

The compiler automatically assigns indexes continuously ascending. When specifying optional parameter *index* the internal counter is set to its value which is then assigned as the index to the first symbolic name of the list (*var_name*). For other names of variables (*next_var*) in the list, assignment continues from this set index. The most appropriate thing is to put the manually indexed operands at the beginning of register definitions.

For one use of *#reg* and *#rem* arbitrary number of operands can be created, the names are separated by commas. If it not possible to put all names on one line, you can continue on the next line. When using directives *#reg* and *#rem*, the programmer does not have to assign the indexes to the variables with a risk to make mistakes.

Actually assigned values can be checked after program compilation by means of command *View | Symbols* or from a file of program dump including also a table with symbolic names.

If we check item *Generate file | Register map* in folder *Sw | Compiler* in the project manager of the Mosaic development environment, then after program compilation we find register assignment in file *xxx.map*, where *xxx* is the name of the project.

Example 9.2

```
#reg bool  Rbit00, Rbit01, Rbit02 ;Rbit00 ... %R0.0,
                                   ;Rbit01 ... %R0.1,
                                   ;Rbit02 ... %R0.2
#reg usint Rbyte0, Rbyte1         ;Rbyte0 ... %R1,
                                   ;Rbyte1 ... %R2
#reg uint  10, Rword0, Rword1     ;Rword0 ... %RW10,
                                   ;Rword1 ... %RW12
#reg udint Rlong0, Rlong1        ;Rlong0 ... %RL14,
                                   ;Rlong1 ... %RL18
#reg real  Rfloat0, Rfloat1       ;Rfloat0 ... %RF22,
                                   ;Rfloat1 ... %RF26
#reg lreal Rdouble0, Rdouble1     ;Rdouble0 ... %RD30,
                                   ;Rdouble1 ... %RD38
```

The *#reg* and *#rem* directives enable to declare one-dimensional fields as well. In this case, the number of field elements will be specified in square brackets after the name of the variable.

Example 9.3

Let us define variable *pole* having 20 elements of *word* type, the first element has index 0 and the last element has index 19. As you can see the field indexes, they begin with zero as all other at TECOMAT do.

```
#reg word array[20]           ;array[0] ... %RW14
                               ;array[1] ... %RW16
                               ;          ...
                               ;array[19] ... %RW52
```

If we want to load the value of the field element, for example with index 15, add one and write it to index 16, we can use the following procedure.

```
LD    array[15]
INR
WR    array[16]
```

If we want to use a bit field in the program to which we will access through table instructions, this field **must begin on bit 0!!** For this we will use command *aligned*. If the number of the bit items of the field is not a multiple of 8, we recommend not to use the bits behind this field remaining to the whole byte, for other bit variables.

```
#reg bool  var1, var2           ;%R0.0, %R0.1
#reg aligned bool  array[12]    ;%R2.0 - %R3.3
#reg aligned bool  var3, var4    ;%R4.0, %R5.0
#reg bool  var5, var6           ;%R5.1, %R5.2
:
LTB  array
:
```

9.6. #struct

Directive *#struct* is used for declaration of data structures, which are substantially new data types derived from basic data types or from previously declared structures. By declaring a structure there are no requirements to occupy the PLC memory, a new data type is just loaded that can be used for automatic assignment of variables in directive *#reg*, for definition of tables, etc.

Declaration of the structure consists of a name (tag) of the structure and types and names of particular structure members.

The syntax is as follows:

```
#struct name_str [aligned] type0[[repeat0] member0[,...]....]
                [aligned] typen[[repeatn] membern]
```

name_str - structure name (tag)
aligned - optional rounding of structure member index to even index
type0 - typen - types of structure members (byte, word...) or tag of another structure
member0 - membern - names of structure members
repeat0 - repeatn - optional repeating of structure members obligatorily closed in square brackets, whole numbers > 0

For definition of the structure, the following principles are valid:

- ◆ if the definition is longer than one line, it is possible to continue on the next line
- ◆ definitions of particular members are separated with comma (',')
- ◆ each structure member must have a type and a name assigned that must be unique within one structure
- ◆ a one-dimensional field of given type can be also a structure member

9. Compiler directives

- ◆ structures in T tables and D data are initialized, structure members are initialized from a listing of values in the order as they are defined, missing initialisation values are replaced by zeros.

Example 9.4

An easy structure can be defined as follows, for example:

```
;easy structure declaration
#struct typTime          ;structure name
    usint Hour,          ;first member of structure
    usint Minute,       ;second member of structure
    usint Second         ;the last member of structure
```

The above mentioned definition created a new data type called *typTime*. This type is 3 bytes long and consists of items *Hour*, *Minute* and *Second*. All items are of *usint* type. Now, we can define variables by means of directive *#reg*, these directives will be of type *typTime*.

```
#reg typTime StartTime
```

By means of directive *#reg* we created variable *StartTime*, which is of type structure *typTime* and has items *Hour*, *Minute* and *Second*. The items are of *usint* type. The access to the variable from the program show the following instructions.

```
;access to the items of variables StartTime
    LD    StartTime~Hour
    LD    StartTime~Minute
    LD    StartTime~Second
```

For structure member references in the program is valid the following:

- ◆ a structure member is separated by tilde character '~'
- ◆ multidimensional members can be indexed with indexes 0 to *defSize-1*, where *defSize* is the member dimension
- ◆ if expansion does not succeed to the final element of the structure, the closest structure member is considered as operand

In the program we can create an arbitrary number of variables of the new type declared by means of directive *#struct typTime*.

```
;declaration of several variables of type typTime
#reg typTime    Leaving, Coming, Time_For_Lunch
```

The following line shows the declaration of the variable *field*, the type of which was defined by the *#struct* directive in the previous text.

```
;variable field declaration of type typTime
#reg typTime    TimeArray[10]          ;variable field
```

For example, particular field elements can be accessed from the program as follows:

```
    LD    TimeArray[0]~Hour
    LD    TimeArray[0]~Minute
    LD    TimeArray[0]~Second
```

In directives *#struct*, the types defined by previous *#struct* directives can also be used when defining types of particular items. In other words structures can be mutually nested which enables to create also complex derived types.

Example 9.5

```

;definition of simple structure
#struct typTime          ;structure name
  usint Hour             ;first member of structure
  usint Minute           ;second member of structure
  usint Second           ;the last member of structure
;
;definition of next simple structure
#struct typDate          ;structure name
  usint Day,             ;first member of structure
  usint Month,           ;second member of structure
  uint Year              ;the last member of structure
;
;definition of structure, members of which are other structures
(structure nesting)
#struct TimeStamp        ;structure name
  typTime Time,         ;first member of structure
  typDate Date,         ;second member of structure
  uint NumberPcs        ;the last member of structure
;
#reg TimeStamp Batch     ;variable of type TimeStamp
#reg TimeStamp Record[10] ;variable field of type TimeStamp
;
P 0
  LD   Batch~Time~Hour
  LD   Batch~Date~Year
  LD   Batch~NumberPcs
  :
  LD   Record[1]~Time~Hour
  LD   Record[9]~Date~Year
  LD   Record[0]~NumberPcs
  :
E 0

```

Example 9.6

In some cases it is necessary that a field is an structure item. A definition of such structure can be for example:

```

#struct typTime          ;structure name
  usint Hour,           ;first member of structure
  usint Minute,         ;second member of structure
  usint Second           ;the last member of structure
;
#struct typDate          ;structure name
  usint Day,            ;first member of structure
  usint Month,          ;second member of structure
  uint Year              ;the last member of structure
;
#struct complexRecord
  typTime[2] Times,
  typDate[2] Dates,
  uint Pieces
;
#reg complexRecord Workpiece
;

```

9. Compiler directives

```
P 0
    LD  Workpiece~Times[0]~Hour
    LD  Workpiece~Dates[1]~Year
    LD  Workpiece~Pieces
    :
E 0
```

It might not be a surprise, that also from such defined structures a variable field can be created.

```
#reg complexRecord Workpieces[3]
;
P 0
    LD  Workpieces[0]~Times[0]~Hour
    LD  Workpieces[0]~Dates[1]~Year
    LD  Workpieces[0]~Pieces
    :
E 0
```

In conclusion of the examples let us specify concrete occupation of registers in the PLC scratchpad memory for this example. The below specified list of register occupation can be obtained in the file with *.map* extension after program compilation:

```
'R' register area map
;* Main file'C:\XPRO\PRAC\STRUCT.950
;
R 0 : 47 =COMPLEXRECORD WORKPIECES[3] ;
R0  >> WORKPIECES[0]~TIMES[0]~HOUR
R1  >> WORKPIECES[0]~TIMES[0]~MINUTE
R2  >> WORKPIECES[0]~TIMES[0]~SECOND
R3  >> WORKPIECES[0]~TIMES[1]~HOUR
R4  >> WORKPIECES[0]~TIMES[1]~MINUTE
R5  >> WORKPIECES[0]~TIMES[1]~SECOND
R6  >> WORKPIECES[0]~DATES[0]~DAY
R7  >> WORKPIECES[0]~DATES[0]~MONTH
RW8 >> WORKPIECES[0]~DATES[0]~YEAR
R10 >> WORKPIECES[0]~DATES[1]~DAY
R11 >> WORKPIECES[0]~DATES[1]~MONTH
RW12 >> WORKPIECES[0]~DATES[1]~YEAR
RW14 > WORKPIECES[0]~PIECES
R16 >> WORKPIECES[1]~TIMES[0]~HOUR
R17 >> WORKPIECES[1]~TIMES[0]~MINUTE
R18 >> WORKPIECES[1]~TIMES[0]~SECOND
R19 >> WORKPIECES[1]~TIMES[1]~HOUR
R20 >> WORKPIECES[1]~TIMES[1]~MINUTE
R21 >> WORKPIECES[1]~TIMES[1]~SECOND
R22 >> WORKPIECES[1]~DATES[0]~DAY
R23 >> WORKPIECES[1]~DATES[0]~MONTH
RW24 >> WORKPIECES[1]~DATES[0]~YEAR
R26 >> WORKPIECES[1]~DATES[1]~DAY
R27 >> WORKPIECES[1]~DATES[1]~MONTH
RW28 >> WORKPIECES[1]~DATES[1]~YEAR
RW30 > WORKPIECES[1]~PIECES
R32 >> WORKPIECES[2]~TIMES[0]~HOUR
R33 >> WORKPIECES[2]~TIMES[0]~MINUTE
R34 >> WORKPIECES[2]~TIMES[0]~SECOND
R35 >> WORKPIECES[2]~TIMES[1]~HOUR
R36 >> WORKPIECES[2]~TIMES[1]~MINUTE
R37 >> WORKPIECES[2]~TIMES[1]~SECOND
```

```
R38 >> WORKPIECES[2]~DATES[0]~DAY
R39 >> WORKPIECES[2]~DATES[0]~MONTH
RW40 >> WORKPIECES[2]~DATES[0]~YEAR
R42 >> WORKPIECES[2]~DATES[1]~DAY
R43 >> WORKPIECES[2]~DATES[1]~MONTH
RW44 >> WORKPIECES[2]~DATES[1]~YEAR
RW46 > WORKPIECES[2]~PIECES
```

Map complete.

Directives *#struct* thus enable to declare new data types. These defined types can be used not only for variable definition, but also for defining T tables. See also the description of directive *#table*.

9.7. #data, #table

Working with tables is strength of TECOMAT PLCs. Therefore a great attention has also been paid to the tools for their creation in the program. Data area D is also defined in the same way as the T tables are.

The general structure for table and data definition are as follows:

```
#data type [index,]name0 = [[val0[repeat0]],]
    [name1 = ]val1[[repeat1]], ...,
    [namen-1 = ]val-1[[repeat-1]],
    [namen = ]val[[repeatn]]
#table type [index,]TabName = [[val0[repeat0]],] val1[[repeat1]],
    ..., valn-1[[repeatn-1]], valn[[repeatn]]
```

- type* - table item type, it can be the same as for *#reg* (table9.1)
- index* - optional number setting the index of the table being created
After imposing of the table index by specifying number *index* other defined tables are assigned indexes ascendingly, beginning with index *index+1*. If *index* is not used, the other tables are automatically assigned the index ascendingly (which is the same as for *#reg*).
- TabName* - T table symbolic name
- Name0 - Namen* - D data symbolic names
- Val0 - Valn* - table items
They can be specified by means of using an arbitrary and permitted numeric system. If the value of the element overflows the maximum value as specified by the type of table (for byte type it is 255, for example) the number is shortened, in case of definition of a bit table is its element non-zero, if its value *valx* is non-zero, too.
- repeat0 - repeatn* - optional repeaters
A number closed with a pair of square brackets (characters '[' and ']') and they enable to repeat in the table the previous value *valx* repeatx-times before them, if the value *valx* is a text string, then only the last character is repeated.

Example 9.7

```
#def On          1
#def Off         0
#def Auto        1
#def Manual      0
#struct typRecord
    bool  OnOff,
    bool  AutMan,
    usint MachineNo,
    uint  TimePreset,
    udint CycleCounter
;
#table typRecord    Record1 = On, Manual, 20, $3009, 1236789
#table typRecord[2] Record2 = On, Manual, 20, $3009, 1236789,
                                Off, Auto, 21, 19029, 1236789
#table bool    BitTable = 0,1,1,0,1,1,1,0,1[8],0,0
#table usint 10,ByteTable = $12,34,%01010110,60#56
#data  uint    WordData = 1,2,3,
                                NextData = 4,$4567[12],8,9,10,0[11],3
;
P 0
:
E 0
```

9.8. #if, #elif, #else, #endif

These directives are used for conditional compilation.

Their syntax is as follows:

```
#if condition_if          ;initial condition and beginning of the first
                           ;block
                           ;of conditional compilation
:                          ; a part of program is being translated, if the arithmetic or
:                          ;logic expression condition_if is non-zero
#elif condition_elif_1    ;beginning of the next block of conditional
                           ;compilation
:                          ;a part of program is being translated, if the expression
                           ;condition_if is zero
:                          ;and condition_elif_1 is non-zero
#elif condition_elif_n    ;beginning of penultimate block of
                           ;conditional compilation
:                          ; a part of program is being translated, if the expression
                           ;condition_if is zero
:                          ;and at the same time the expressions condition_elif_1 to
                           ;condition_elif_n-1
:                          ;are zero and condition_elif_n is non-zero
#else                     ;beginning of the last block of conditional compilation
:                          ; a part of program is being translated, if no conditions
:                          ;from the previous conditions are fulfilled
#endif                   ;end of conditional compilation
```

condition_if, condition_elif_1, ..., condition_elif_n
- relational or mathematical expressions

Always just one branch is translated at a time, which fulfils its condition as the first one. Directives `#if ... #endif` can be mutually nested. The following principles are valid:

- ◆ each `#if` must have its `#endif`
- ◆ `#elif` nebo `#else` is related to the closest previous `#if`
- ◆ `#elif` and `#else` are optional

Example 9.8

Conditional compilations are very frequently used in such situations, when the programmer simulates the feedbacks of a real machine, when debugging the program under simulation. These program parts are almost always compiled conditionally only in case of debugging under simulation.

```
#def DEBUG 1          ;1 ... debugging in simulation
                    ;0 ... real machine

;
#if DEBUG == 1
#include simmachine.950 ;compile simmachine.950 during debugging
#endif
```

Example 9.9

Conditional compilations can be advantageously used also in such situations when the user program is necessary to be translated for the central units of different series. For this purpose, internal variable `_PLCTYPE_` is used the value of which determines the PLC series for which the user program is compiled. In the Mosaic development environment, setting of the compiler is determined by selection of the PLC type in the project manager in folder *Hw / Select type of PLC series*.

```
#if _PLCTYPE_ == CPM1D
    INR  variable
#else
    LD   variable
    INR
    WR   variable
#endif
```

9.9. #ifdef, #ifndef, #else, #endif

These directives are used for conditional compilation.

The condition for compilation is the existence of the symbolic name in the previous program. Their syntax is as follows:

```
#ifdef symbolic_name ;condition for compilation
:      ;a part of program is translated if the given symbolic name
      ;is defined in the previous part program
#else
:      ;switch of block of conditional compilation
:      ;a part of program is translated, if the given symbolic name
      ;is not defined in the previous part program
#endif ;end of conditional compilation
```

9. Compiler directives

Directive *#ifndef* enables the compilation, when the symbolic name as specified in the directive does not exist yet. Its application is the same as for directive *#ifdef*.

```
#ifndef NUMBER
    #def NUMBER    12
    #reg byte array[NUMBER]
#endif
```

9.10. #usi

In the user program, the xPRO compiler enables to use user instructions USI written for TECOMAT PLCs. Actually, the USI instruction is a function written in language C, its compiled code is linked by the compiler to the machine code of the user program. In this way, the instruction file of the PLC central unit can be expanded by the functions that are not part of the standard instruction file. For definition of the instructions USI in the program, directive *#usi* is used. Its syntax is as follows:

```
#usi [index,]instruction_name = file_name
```

- index* - an optional number setting the index of the instruction being created
After imposing of the instruction index by specifying number *index* other defined instructions are assigned indexes ascendingly, beginning with index *index+1*. If *index* is not used, the other instructions are automatically assigned the index ascendingly.
- instruction_name* - symbolic name of user instruction index, in case the index is specified, it is optional, otherwise obligatory.
- file_name* - obligatory name of the binary disk file containing executive part of user instructions

Together with the Mosaic development environment, a great number of USI instructions is supplied which are typically placed in directory *Mosaic1\USI* after environment installation. Each USI instruction is compiled several times for various types of the PLC central units. This means that for each USI instruction usually exist several files in the USI directory.

The type of central unit for which the file with the USI instruction code is designated differs by the extension of the file. If we do not specify the extension of the file in directive *#usi*, the compiler automatically selects the file with the correct extension based on the type of central unit for which it is being compiled. For simulations, files with extension *.dll* are used. These file are used automatically by the environment in the mode of the PLC being simulated independently of for which series of central units compilation was carried out. Files with the *.dll* extension are used only for simulation of the USI instruction. The file is saved in the machine code of the compiled program with the code corresponding to the selected type of the central unit. So, during switching from the mode of PLC simulation into debugging with a real PLC, it is not necessary to do any treatment in connection with the USI instructions used.

Creation of user instructions is described in chapter 12.

File extension with USI code	For central unit of series
.uia	A
.uib	B
.uic	C
.uid	D
.uim	M
.uis	S
.dll	for PLC simulations under Mosaic development environment

Example 9.10

```
#usi operatorPanel = ter_id04 ;definition
;
P 0
:
  USI operatorPanel ;application in the program
:
E 0

Or

#usi operatorPanel = ter_id04 ;definition
#def TERM USI operatorPanel
;
P 0
:
  TERM ;application in the program
:
E 0
```

9.11. #label

By means of directive *#label*, the initial index for automatic label assignment is set and symbolic labels for application for example in indexed or relative jumps are reserved. Thus this directive is applied only in such cases when the programmer of a task needs to reserve concrete label numbers. If we need to use a label in a PLC program, for example as the task of a conditional jump, it is not necessary to declare this label in directive *#label*. It is enough to write the symbolic name of the label on an individual program line and terminate it through colon character (see chapter 4.3). The *#label* directive has the following form:

```
#label [index,][name_L0[[repeat0]][,name_L1[[repeat1]],
...name_Ln[[repeatn]]]
```

- index* - an optional positive number specifying the first index for automatic assignment of label
- name_L0 - name_Ln* - arbitrary label symbolic names to which absolute labels are assigned, the list of label names can continue on more lines
- repeat0 - repeatn* - optional repeaters obligatorily closed in square brackets determining how many label indexes can be left out after the label defined

Attention! If absolute labels are used in the program, it is necessary to ensure that they are not assigned to another (symbolic) label. In the programs written in the Mosaic development environment we recommend not to use absolute labels. This avoids collisions, such as double declarations, etc.

Example 9.11

```
;space form L 0 to L 9 is not occupied
#label    10, FirstLabel [3],          ;FirstLabel = L 10
          Lab, Label[10],             ;Lab = L 13, Label = L 14
          NextLabel                    ;NextLabel = L 24
#label    100, LabelHundred           ;LabelHundred = L 100
```

9.12. #macro, #endm

The xPRO compiler disposes also with such a powerful tool as macroinstructions. A skilled programmer is able to arrange well a long and very hard-to-read program by means of macroinstructions.

Macroinstructions (shortly macros) are used in applications, where there are identical parts in the program, but which use different operands (such as controlling several motors). If the part of driver which is identical everywhere (the same instruction sequence is used) is written as a macroinstruction, then each operation of the motor will be written in the program as a macroinstruction. Only the parameters passed to this macroinstruction will be different. The macroinstructions can be mutually nested, which means in the body of a macroinstruction, another macroinstruction can be used.

Example 9.12

Let us define a short macroinstruction that logically adds two bits and the result will be written into another bit:

```
#reg bool input, output, va, vb, vc
;
;macroinstruction definition
#macro first_macro (first, second, third)
    LD    first
    OR    second    ;logical add
    WR    third     ;result
#endm          ;end of macroinstruction definition
;
P 0
:
LDC  input
first_macro (va, vb, vc)
WRC  output
:
E 0
```

It is obvious that the following principles must be followed when defining macroinstructions:

- ◆ The definition of a macroinstruction begins with directive *#macro* and ends with directive *#endm*.
- ◆ After the name of the macroinstruction, there is a list of so called formal macro parameters and is closed in parenthesis. Even if the macroinstruction does not use any parameters, the parenthesis must be used.

- ◆ If it not possible to put the list of formal parameters on one line, it is possible to continue on the next line, for example:

```
#macro long_macroinstruction (first_parameter,  
                             second_parameter)
```

- ◆ The body of a macroinstruction consists of a sequence of instructions. Inside the macroinstruction, key words *#def*, *#reg*, *#label*, *#table* and *#data* can be used. Such defined symbolic names are local ones, which means they are known only in the body of the macroinstruction. If a local symbolic name is identical to the one defined in the main program, then the local one is valid.

The application of the macroinstruction in the text is as follows:

```
first_macro (Block_M1, Starter_M1, Output_M1)
```

When using a macroinstruction, the name of the macroinstruction is written down and the passed, i.e. real parameters are put in brackets. These parameters replace the formal parameters of the macroinstruction during macro expansion. The number of parameters in the definition of the macroinstruction (i.e. the number of formal parameters) must be identical to the number of the parameters passed (i.e. real parameters).

How are macroinstructions processed?

If the compiler encounters a macroinstruction name, it will expand it. This means, the name of the macroinstruction will be replaced with a sequence of instructions forming the body of the macroinstruction. The formal parameters will be replaced with the real ones.

In case of previously defined macro *first_macro*

```
LDC  input  
first_macro (va, vb, vc)  
WRC  output
```

the program will look after expansion of macroinstruction *first_macro* by the compiler as follows:

```
LDC  input  
LD   va           ;macro expansion starts here  
OR   vb  
WR   vc           ;last instruction of macro expansion  
WRC  output
```

9.13. #mnemo, #mnemoend

Directive *#mnemo* informs the editor of relay diagrams that all instructions following for this directive be written in their text form, not in the form of a relay diagram. Return to the relay diagrams will be carried out after using directive *#mnemoend*.

Their syntax is as follows:

```
#mnemo  
      :           ;program section, which is written in instructions  
#mnemoend
```

When programming relay diagrams it is sometimes necessary to create a sequence of instructions, interpretation of which is not logical in relay symbols. This part of the program can be enclosed by the pair of *#mnemo* / *#mnemoend* and it will always be displayed in its text form.

9. Compiler directives

Note: Directives should be used in pairs which means that directive *#mnemo* must have its directive *#mnemoend* in pair to end.

9.14. #useoption

In the Mosaic development environment, directives *#useoption* are generated, based on the setting of the compiler, into control file *xxx.mak*, where *xxx* is the name of the project, during compilation. By doing this, setting of the compiler is implemented in the project and it is not necessary to change it manually. Changes to the setting of the compiler become effective in the next compilation. Thanks to the generation of directive *#useoption* into the control file *xxx.mak* we have additionally visual check of compiler parameters available.

The syntax of directive *#useoption* is as follows:

```
#useoption mod = n ;comment
```

where *mod* is one of the following parameters:

- | | |
|---------------------|---|
| <i>CPM</i> | - PLC central unit series
0 ... series A
1 ... series S
2 ... series M
3 ... series E
4 ... series D
5 ... series B
6 ... series C
This value is saved into internal variable <i>_PLCTYPE_</i> , which can be used anywhere in the user program, for example for conditional compilation (see example 9.11). |
| <i>BlockOut</i> | - external locking of PLC outputs
0 ... off
1 ... active at log.0
2 ... active at log.1 |
| <i>EnableRun</i> | - external permission RUN
0 ... off
1 ... active at log.0
2 ... active at log.1 |
| <i>AlarmTime</i> | - cycle length for warning [ms] |
| <i>MaxCycleTime</i> | - maximum cycle length [ms] |
| <i>RemZone</i> | - remanent zone length
0 ... no register backed up
≠0 ... number of backed up registers R beginning from R0 |
| <i>PlcStart</i> | - type of start of PLC after switching power supply on
0 ... warm (the content of backed up registers is hold)
1 ... cold (all registers R set to zero after switching on) |
| <i>ProtTable</i> | - protected T tables - is significant only in the case that user program is backed up in the EEPROM memory
0 ... off (after switching on PLC supply, the whole user program including T tables will be loaded from the EEPROM memory)
1 ... on (after switching on PLC supply, the whole user program including T tables will be loaded from the EEPROM memory) |

Example 9.13

; Warning: This file is managed by Mosaic development environment.
 ; It is not recommended to change it manually!

```
#program Plc1 , V1.0
;*****
;<ActionName/>
;<Programmer/>
;<FirmName/>
;<Copyright/>
;*****
;<History>
;</History>
;*****
#useoption CPM = 5           ;CPM type: B
#useoption RemZone = 2      ;the remanent zone length
#useoption AlarmTime = 150  ;first alarm [ms]
#useoption MaxCycleTime = 250 ;maximum cycle ms]
#useoption PLCstart = 1     ;cold start
#useoption BlockOut = 0     ;external outputs blocking off
#useoption EnableRun = 0    ;external program execution blocking
                             ; is off
#useoption ProtTable = 0    ;tables are not saved while PLC is
                             ; restarting
;*****
#usefile "Plc1.hwc"
#usefile "Plc1.mos"
#usefile "..\action.sym"
#usefile "Plc1.sym"
```

The above-mentioned example shows the control file *Plc1.mak* in the Mosaic development environment for the central unit of series B and cold restart after switching on PLC supply with backing up the contents of registers R0 and R1. After switching on, the registers have the same content as before switching off, the other registers are set to zero. The name of project group is *action* and the name of the project is *Plc1*.

10. USER PROCESSES

10.1. GENERAL PRINCIPLES OF ACTIVATION

The user program consists of user processes. Theoretically, there can be 65 of these processes (P0 to P64), but practically there are significantly fewer of them used. Contrary to traditional operation systems of real time for computers, the user does not have so many possibilities to control the processes.

The processes are activated according to pre-defined rules. Within these rules we can additionally effect the activation of the most of the processes in the user program when it is running.

Table 10.1 Overview of user program processes and their assignment

Processes	Assignment
P0	basic process
P1 to P4	four-phase activated processes
P5 to P9	time-activated processes
P10 to P40	user-activated processes
P41 to P48	interrupt processes
P49	system process – do not use!
P50 to P57	breakpoint treatment
P58, P59	system process – do not use!
P60	subroutine package
P61	system process – do not use!
P62	warm restart
P63	cold restart
P64	final process of cycle

In central units of series E, only process P0 can be programmed. A list of process activations is in figure 10.1. System actions are thin-framed, user processes are thick

Process elimination

The user does not have to use all processes. If the user is satisfied with conventional one-loop control, he can specify only process P0. The processes can be eliminated as follows:

- ◆ The process is not programmed, i.e. bracket instructions P and E of the particular process are not used. Process P0 cannot be eliminated in this way, but it can be empty.
- ◆ The process is empty, i.e. no other instruction is between bracket instructions P and E. Its activation will appear as no operation.
- ◆ The activation mask of the process is set to zero. Activation masks of processes P10 to P48 are contained in system registers S25 to S29 (see chapter 5.3.). The process with the activation mask set to zero will be suppressed in the next cycle or immediately as the case may be, when talking about processes P41 to P48. The processes P0 to P9 managed by the system cannot be eliminated in this way.

Caution: When entering into of any of processes P0 to P40, P62, P63, P64 being solved, the active user stack is set to zero (when entering into process P0 being solved, stack A is always active).

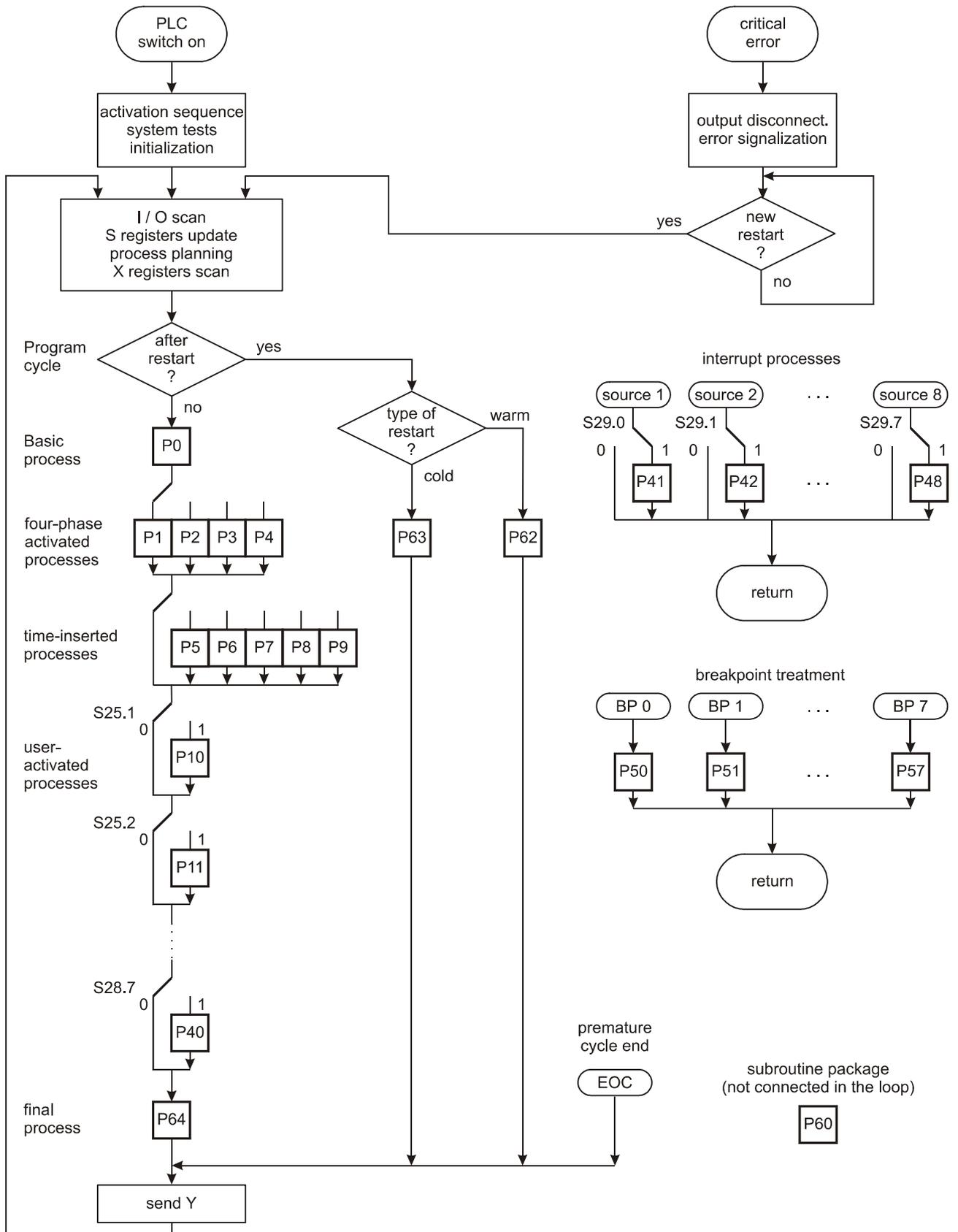


Figure 10.1 Process activation flow chart

Jump among processes

It is not excluded (even if it is not recommended directly) to convert control from one process to another one by means jump or call instructions. But the closest instruction E or ED, EC ordinarily ends the activated process - the user can switch among programs of any processes, the system program does not register these changes as the changes of process activation and it understands the evaluation of the process end as return from the process activated by the system program (not by user transitions). If, for example, we jump from process P0 to the label positioned in the last process P64, then E64 does not cause the I/O scan but closing P0 (the same as E0) and its return to the system program that calls the following user process. In this case, the stack will be set to zero after closing process P0, thus the transition between the processes of jump or call instructions does not result in any change of the stack values.

Premature cycle end

Only the EOC instruction is able to violate the sequence of the processes planned for the actual loop and directly (and unconditionally) carry out the I/O scan. This can be advantageously used for example for treating interrupts or ensuring a fast feedback to critical situations. When planning processes for the next cycle at such I/O scan, it is not respected that some of the originally planned processes were not activated - the planning starts again.

Interrupt processes

Any of the loop processes can be interrupted by some of the interrupt processes P41 to P48. The instruction being processed at the moment of interrupt will be finished and after that, the first instruction of the interrupt process will be executed. After its instruction E (or, ED, EC), the interrupted process will continue. The interrupt process does not change the state of any level of the active stack.

Cycle time of interrupt processes

The cycle time is increased by the sum of the times of interrupt processes that were activated when passing the loop. Therefore, utility programs should contain only necessary instructions to ensure a fast feedback to the situation that caused the interrupt. Otherwise, the cycle time could be significantly longer. The extreme situation could be that the execution of other processes could be stopped and the system would operate only this interrupt. This state is limited by a system condition that limits duration of the interrupt process to 5 ms. For the same reason, it is not possible to use interrupt nesting (interruption of the interrupt process).

The mechanism of interrupt is a very effective system enabling to shorten significantly the system feedback to critical situations, but it must be used carefully.

10.2. I/O SCAN

After the activation sequence (see chapter 2.1), after finishing the last process planned for the active cycle or after instruction EOC, so called I/O scan is carried out. The duration of the I/O scan depends on system configuration, remanent zone length and range of other system services (see appendix).

At the cycle I/O scan, the state of registers Y is sent to the outputs, registers X are updated according to the state of the inputs, system time is updated, processes for the activation of the next cycle are planned, leading and trailing edges are generated, and it is

decided on the mode of the system (mode RUN - HALT, output blocking, type of restart if necessary). Further to this, the valid state of system registers S is set and stack A is activated and set to zero.

10.3. RESTART TREATMENT - PROCESSES P62, P63

P62 - warm restart

P63 - cold restart

At the first cycle after restarting, one of the processes P62 or P63 can be activated. These processes serve to initiate variables. If one of the processes P62 or P63 is executed, no other process is executed at this cycle (not the P0 process). Central units of series S, D, B and C have the activation of interrupt processes hold during the execution of processes P62 or P63 to avoid hazards connected with working with non-initiated data structures.

Note: For distinguishing the first cycle after restart (SP = 1 or change of mode HALT → RUN without restart), bit S2.6 is used, that is set to log.1, when the system is started by this cycle without restart.

Warm restart – 16 bit model

If warm restart is set, process P62 is executed and in the following cycle process P0 as well as other programmed processes will be started. If process P62 is not programmed, process P63 will be executed instead of it. If no of the processes P62, P63 is programmed, process P0 as well as other processes will be directly executed.

Warm restart – 32 bit model

If warm restart is set, process P62 is executed and in the following cycle process P0 as well as other programmed processes will be started. If process P62 is not programmed, process P0 as well as other processes will be directly executed.

The central units with the 32 bit stack width **do not have** automatic restart treatment process doubling. If we want to run the same algorithm at cold as well as warm restart, we will write it as a subroutine into process P60 and call it from both processes P62 and P63.

Cold restart – 16 bit model

If cold restart is set, process P63 will be executed and in the following cycle process P0 as well as other programmed processes will be started. If process P63 is not programmed, process P62 will be executed instead of it. If no of the processes P62, P63 is programmed, process P0 as well as other processes will be directly executed.

Cold restart – 32 bit model

If cold restart is set, process P63 will be executed and in the following cycle process P0 as well as other programmed processes will be started. If process P63 is not programmed, process P0 as well as other processes will be directly executed.

The central units with the 32 bit stack width **do not have** automatic restart treatment process doubling. If we want to run the same algorithm at cold as well as warm restart, we will write it as a subroutine into process P60 and call it from both processes P62 and P63.

No restart

If a mode is set without restart, no of the processes P62, P63 will be executed, but process P0 as well as other processes will be directly executed.

Example 10.1

```
#reg byte register
;
P 0
:
E 0
;
P 62          ;warm restart
    LD    $10
    WR    register ;initial value
E 62
;
P 63          ;cold restart
    LD    $20
    WR    register ;initial value
E 63
```

10.4. LOOP PROCESSES

As you can see on picture 10.1 it is obvious that only the processes P0 and P64 are activated at every cycle, processes P1 to P9 are activated in selected cycles, processes P10 P40 are activated by the user through control masks.

As the result of this is that these processes seem to be different loops of the user program each of them having a different cycle time. Therefore it is possible to call this way of activation as multi-loop control.

10.4.1. Basic process P0

P0 - initial process of every cycle

Basic process P0 is obligatory part of the basic structure of the user program. Even if we do not want to use process P0, we have to program it (obligatory instructions P 0, E 0).

Basic process P0 is activated at every cycle as the first one excluding restart, when one of the processes P62 or P63 is activated. It is useful to implement here all initial operations and the user scheduler of processes. Since it is always activated, only such tasks should be implemented in it, where a short feedback time is required. It should not be filled with tasks of a lower priority.

Example 10.2

```
#reg byte input,output
;
P 0
    LD    input
    :
    WR    output
E 0
```

10.4.2. Four-phase activated processes P1, P2, P3, P4

- P1** - process implemented at every first cycle of four
- P2** - process implemented at every second cycle of four
- P3** - process implemented at every third cycle of four
- P4** - process implemented at every fourth cycle of four

The processes change cyclically in the order P1, P2, P3, P4, P1, ... (figure 10.2). Their basic property is that at each cycle just one of these processes is active. This enables to use these processes to program collision actions that must not be executed at the same cycle or for distribution of one long algorithm into four parts to reduce the cycle time of the user program and accelerate PLC feedback. With these simple means it is possible to carry out consequent algorithm synchronization and avoid hazardous synchronous situations, faulty transitions and undesirable transition processes.

Processes P1 to P4 are activated always after the basic process P0 is finished. The activation of processes P1 to P4 is derived from the state of the cycle counter at S4. As a consequence of this is, if we use, for example, processes P1, P2 and P3 only, then at the cycle that would belong to process P4, none of the processes of this group will be activated. At the first cycle, process P1 will always be activated after system restart.

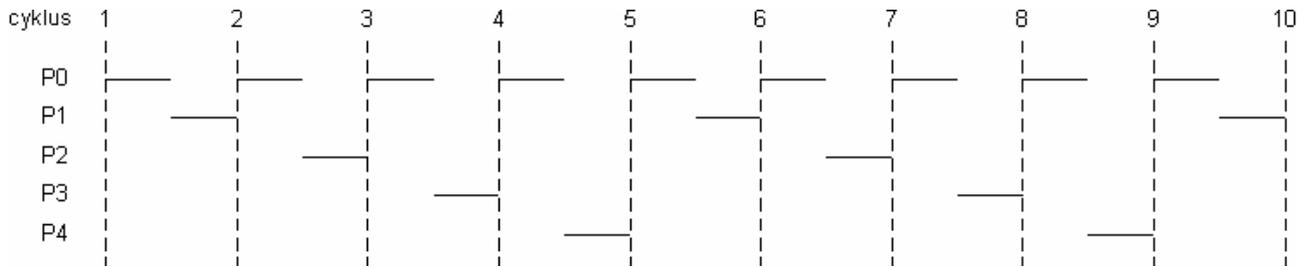


Figure 10.2 Time course of activations of P0 to P4 processes

Example 10.3

```
#reg byte input,output
;
P 0
    LD    input
    :
    WR    output           ;executed in every cycle
E 0
;
P 1
    :
    :           ;executed in 1., 5., 9., ... cycle
E 1
;
P 2
    :
    :           ;executed in 2., 6., 10., ... cycle
E 2
;
P 3
    :
    :           ;executed in 3., 7., 11., ... cycle
E 3
;
P 4
    :
    :           ;executed in 4., 8., 12., ... cycle
E 4
```

10.4.3. Time-activated processes P5, P6, P7, P8, P9

- P5** - process implemented every 400 ms
- P6** - process implemented every 3,2 s (shift by 200 ms against P5)
- P7** - process implemented every 25,6 s (shift by 400 ms against P6)
- P8** - process implemented every 204,8 s, i.e. 3,4 min. (shift by 800 ms against P7)
- P9** - process implemented every 1638,4 s, i.e. 27,2 min. (shift by 1,6 s against P8)

Processes P5, P6, P7, P8, P9 are activated always after a certain period of time has elapsed. The accuracy of this time slice is determined by thy cycle time.

Additionally, the activations of the particular processes are shifted against each other in such a way that maximally one of these processes is activated in one cycle (see Fig. 10.3).

Processes P6 to P9 are activated after four-phase activated processes P1 to P4. The activations of the processes are derived from the counter of time units SW14. The frequency of the activation of the higher number process is always eight times less than the frequency of the activation of the process with the number smaller by one (the interval is eight times longer). The condition for correct activations of processes P5 to P9 is a cycle time shorter than 200 ms. After exceeding this cycle time, more processes of this group could be activated in one cycle and the activations of the P5 process could fail.

Processes P5 to P9 are advantageously used especially in the following cases:

- “execute several times a second...”,
- “execute after several seconds...”,
- “several times a minute ...”,
- “after several minutes...”,
- “approximately in half an hour...”.

Then, it is not necessary to work with timers or time-measuring registers and it is enough to implement the task i the appropriate process.

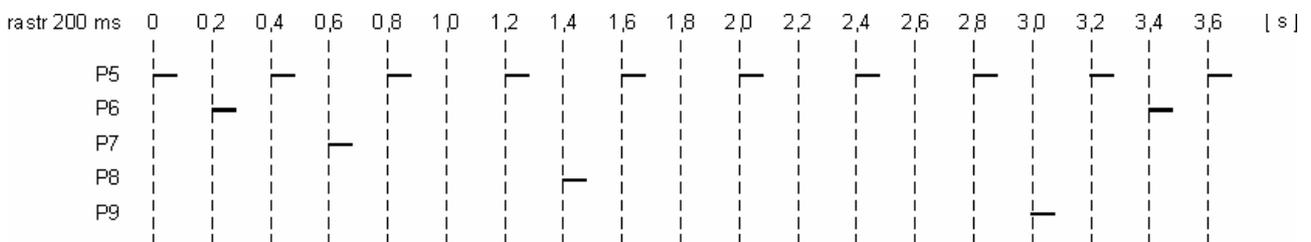


Figure 10.3 Possible moments of activation of processes P5 to P9

Example 10.4

```
#reg byte input,output
;
P 0
    LD    input
    :
    WR    input
    ;executed in every cycle
E 0
;
P 5
    :
    ;executed every 400 ms
E 5
;
P 6
    :
    ;executed every 3,2 s
E 6
```

```

;
P 7
:           ;executed every 25,6 s
E 7
;
P 8
:           ;executed every 3,4 min
E 8
;
P 9
:           ;executed every 27,2 min
E 9

```

10.4.4. User-activated processes P10 to P40

P10 to P40 - user-inserted processes

Processes P10 to P40 are activated by the user by setting the control bits in system registers S25 to S28. By setting the particular bit to log.1, the given user program is activated in the following cycle. The assignment of the processes to the masks is ordinal: P10 - S25.1, P11 - S25.2, ..., P40 - S28.7. The values in positions S25.1 to S28.7 are influenced only by the user, the system program does not influence them. Processes P10 to P40 are thus active from the moment when they were inserted by the user, till the moment when they were disabled. The processes are activated in the order according to figure 10.1 (ascendingly) and this order cannot be changed neither forward nor backward. After restarting the PC, all bits are set to zero and processes P10 to P40 are not activated.

	.7	.6	.5	.4	.3	.2	.1	.0
S24	P8	P7	P6	P5	P4	P3	P2	P1
S25	P16	P15	P14	P13	P12	P11	P10	P9
S26	P24	P23	P22	P21	P20	P19	P18	P17
S27	P32	P31	P30	P29	P28	P27	P26	P25
S28	P40	P39	P38	P37	P36	P35	P34	P33

Masks S24.0 to S25.0 are assigned to processes P1 to P9 and are managed by the system program at the I/O scan (it does not change them during the cycle). The mask values are designated to indicate the processes planned to activate this cycle. By rewriting the masks by the user program the activations of processes P1 to P9 cannot be influenced. Processes P10 to P40 can be used widely. They can be used to realize conditional execution of various activities when the individual activities are programmed in particular processes, in the P0 basic process are evaluated conditions and based on them particular processes are activated. It is the user's choice what meaning he assigns to processes P10 to P40 and according to which rules he will activate them and how long they will stay activated (from one-shot activations to a long-term mode selection).

Processes P10 to P40 are activated after time-inserted processes P5 to P9 ascendingly. The change of the particular control bit will become visible always in the following cycle.

Example 10.5

```

#reg bool input0, input1, input2, input3, test1, test2, test3
;
P 0
    LD    input0

```

10. User processes

```
WR  %S25.1      ;P10 active at input0 = log.1
LD  input1
LET  test1      ;P11 activated once at leading edge
SET  %S25.2     ;input1
LD  input2
LET  test2      ;P12 activated at leading edge
SET  %S25.3     ;input2
LD  input3
LET  test3      ;P12 deactivated at leading edge
RES  %S25.3     ;input3
E 0
;
P 10
:
E 10
;
P 11
:
LD  0
WR  %S25.2     ;P11 activated once, it will be auto-cancelled
E 11
;
P 12
:
E 12
```

10.4.5. Cycle final process P64

P64 - process always inserted at the end of cycle

Process P64 is executed always as the last user process of the cycle. It is suitable for programming of such algorithms that are necessary to be executed after processes P1 to P40 have been executed. Programming of this process is not obligatory.

Example 10.6

```
P 0
:      ;executed at the beginning of every cycle
E 0
;
P 5
:      ;executed in the middle of the cycle every 400 ms
E 5
;
P 64
:      ;executed at the end of every cycle
E 64
```

10.5. INTERRUPT PROCESSES

Behaviour of interrupt processes

Any of the loop processes can be interrupted at any instruction. The system program ensures that the instruction being processed will be finished, it postpones the state of the

active stack, system registers S0, S1 and passes control to the beginning of the interrupt process. Its last instruction (E or ED, EC with fulfilled condition) returns the undamaged state of the stack and returns control to the interrupted process behind the position of interruption. The interrupt process does not execute neither the start nor end phase of the cycle. As a consequence of this is that it is not possible to carry the parameters from the process being interrupted to the process interrupting at the active stack.

Duration limitation of interrupt processes

Interrupt processes should be as short as possible, they should not exceed 5 ms, otherwise the PLC will stop due to a critical error 80 31 PC PC. Generally speaking, the interrupt processes are used to treat critical situations and fast events. Here, it is necessary to realize the difference between the image of inputs and output in the scratchpad, which is updated only at the I/O scan, and direct access to inputs and outputs, which allows reading of the current state and react immediately. An excessive use of direct access results in dragging out of a cycle and a higher risk of time hazards (see chapter 6).

Interrupt cannot be nested

An interrupting process cannot be interrupted any more (it is not possible to nest another interrupt). If another request for interrupt comes during the interrupting process, it is not lost and the particular interrupting process will be activated consecutively. If more requests come, they are logically added (not as for quantity) and the following priorities for the implementation of interrupt processes are respected:

Interrupt priorities

- 1st priority - error-activated interrupt (P43)
- 2nd priority - input-activated interrupt (P42)
- 3rd priority - time-activated interrupt (P41)
- 4th and next priorities - other interrupt sources ascendingly according to numbers (P44 to P48)

Forbidden activation of the interrupt process

The central units of series S, D, B and C allow a temporary interrupt process disable. After restart, the bits in register S29 related to the processes programmed set to log.1. By setting of any bit to zero, the applicable interrupt process will be stopped (it is not executed till the I/O scan). By resetting of this bit, the process in question will be called immediately after a request. When an interrupt request comes at the moment, when the process in question is disconnected, this request is lost, thus it is not possible to realize this possibility to defer the interrupt process. After restart, during the execution of processes P62 and P63, the interrupt processes are temporarily disabled to avoid the risk due to non-initialised values.

The system releases them before the first entry to the P0 process. If we do not want the interrupt to take place, we disable the interrupt process by setting the S29 bit in question to zero already in process P62, or P63 as the case may be.

	.7	.6	.5	.4	.3	.2	.1	.0
S29	P48	P47	P46	P45	P44	P43	P42	P41

10.5.1. Time-activated interrupt P41

Process P41 is activated regularly every 10 ms. It is suitable to treat events that require shorter reaction times than the cycle time is. When working with inputs and outputs, we have to use the direct access, not images in the scratchpad, that do not change during the cycle (see chapter 6).

Process P41 has the third highest priority of interrupt process sequencing that is applied when there are more requests for more interrupt processes.

Example 10.7

16 bit model

```
#reg bool      state
#reg udint     counter
;
P 0
:
E 0
;
P 41
    LD  %U$9000    ;X0 input physical address
    AND %1000      ;input X0.3
    LET state      ;leading edge test
    EC
    INR counter    ;pulse incrementation
E 41
```

32 bit model

```
#reg bool      state
#reg udint     counter
;
P 0
:
E 0
;
P 41
    LD  1          ;PAR - read inputs
    LD  1          ;RM - rack number
    LD  4          ;POS - position of the module in the rack
    RFRM          ;actual data loading to the structure r1_p4_DI
    LD  r1_p4_DI~DI3 ;input 3
    LET state      ;leading edge test
    EC
    INR counter    ;increase number of pulses
E 41
```

10.5.2. Input-activated interrupt P42

Process P42 is activated at the change of interrupt input. The concrete solution depends on the type of PLC.

Process P42 can be advantageously used to treat events that require shorter reaction times than the cycle time is. When working with inputs and outputs, we have to use direct access, not images in the scratchpad, that do not change within the cycle (see chapter 6).

Process P42 has the second highest priority of interrupt process sequencing that is applied when there are more requests for more interrupt processes.

Activation of P42 in NS950

In PLC NS950 the P42 process is activated at any change of inputs 0.0 of those input units that have the interrupt activated through a jumper (units IB-36 to IB-47). If the active interrupt has more input units than one, this differentiation must be done by the user as follows:

In the user program and at each cycle, X images of those units are copied, from which we can expect interrupt. In the P42 interrupt process, we will find out by comparison of the direct inputs U with the copies of corresponding X images, in which unit the change of value and the lowest bit took place which resulted in the interrupt.

The reason for copying the X images is that if any either of the units initiates the interrupt during the I/O scan, then it is more than likely that the new state of the inputs of this unit will be written in the X images in the scratchpad, but the interrupt process will be called after a new cycle is started. This "time risk" is eliminated just through the comparison of the direct inputs with X images being one cycle older.

The minimum time between the interrupts from the same unit must be corresponding with this and it must be a little bit longer than the time of the longest cycle.

Example 10.8a

16 bit model

```
#def byte output
#reg bool state,aux
;
P 0
:
E 0
;
P 42
    LD    %U$9000      ;X0 input physical address
    AND   1             ;input X0.0
    LET   state        ;leading edge test
    SET   aux          ;output value
    LD    aux
    AND   %100         ;output bit 2
    LD    output
    AND   %11111011    ;clearing old value of bit 2
    OR
    WR    output       ;adding new value of bit 2
    WR    %U$9100      ;copy to scratchpad
    WR    %U$9100      ;write to physical address
E 42
```

Activation of P42 in TC500 and TC600

In PLCs TC500 and TC600 the P42 process is activated at any change of the logic level of inputs DI0 to DI3. The activation of process P42 allow the following versions: TC503 to TC507, TC513 to TC517, TC603 to TC607.

Detailed information for the operation of the P42 interrupt process can be found in the following manuals:

TECOMAT TC500 programmable logic controllers, order number TXV 138 07.02 and TECOMAT TC600 programmable logic controllers, order number TXV 138 08.02.

Activation of P42 in TC700

In the TC700 PLC, the P42 process can be initiated by a module having in the panel *Module Setup* accessible from the project manager in folder *Hw | HW Configuration* through icon , with enabled option *Module can initiate interrupt*. If the option is disabled, this module does not initiate the interrupt. If this option is not in the panel, this means that this module type can not initiate the interrupt at all.

If the interrupt is enabled for more than one module, system registers S56 and S57 are used to distinguish these interrupts. After passing through the instruction P42, the register S56 contains the module position in the rack that initiated the interrupt, and the register S57 contains the number of this rack (set by the switch on the rack). By this, the interrupting module is unequivocally determined. It is up to the user to make a decision by means of these registers at the beginning of the process how to treat the interrupt.

If the interrupt is initiated by two peripheral modules at the same time, the P42 process is initiated twice after each other, individually for each module. This means that during each of interrupt initiation, we always operate one module only.

To speed up the operation, those inputs or the whole module objects are updated before passing through the P42 instruction, which are allowed to initiate interrupt by means of a detailed setup, while another inputs are frozen, so as they do not change their value during the cycle. If, for example, we have on an input peripheral module the initiation of interrupt set from the leading edge of the input DI0 and from the trailing edge DI6, only the values of these two inputs and interrupt flags will be updated at the moment of interrupt initiation from this module, while the values of another inputs not.

Thanks to this function, it is possible to work with other inputs in interrupted processes without any risk of time hazard.

Details relating to the interrupt initiation for particular modules are given in the relevant manuals.

Example 10.8b

Model 32 bits

```
P 0
    :
E 0
;
P 42
    LD    r1_p4_INT~INT0    ;flag of input 0 leading edge interrupt
    WR    r1_p5_DO~DO2      ;output 2
    LD    2                  ;PAR - write outputs
    LD    1                  ;RM - rack number
    LD    5                  ;POS - position of the module in the rack
    RFRM                                ;actual data writing from structure r1_p5_DO
E 42
```

10.5.3. Error-activated interrupt P43

Process P43 is activated when an error occurs that does not stop running the PLC (it writes into register S34).

Process P43 can be advantageously used for bulk treatment of error states.

Process P43 has the highest priority of interrupt process sequencing that is applied when there are more requests for more interrupt processes.

Example 10.9

```
#reg bool  error
#reg uint  va,vb,vc
;
P 0
    LD    va
    LD    vb
    DID           ;when dividing by zero, process P43 is called
    WR    vc
E 0
;
P 43
    LD    1
    WR    error   ;setting error flag
E 43
```

10.5.4. HW counter-activated interrupt or incremental encoder-activated interrupt P44

In PLC TC500 and TC600 the P44 process is activated when reaching preset or when the range of the hw counter is overflowed. The activation of process P44 allow the following versions: TC503 to TC507, TC513 to TC517, TC603 to TC607.

Detailed information for the operation of the P44 interrupt process can be found in the following manuals:

TECOMAT TC500 programmable logic controller hardware, order number TXV 138 07.02 and TECOMAT TC600 programmable logic controller hardware, order number TXV 138 08.02.

Process P44 has the fourth highest priority of interrupt process sequencing that is applied when there are more requests for more interrupt processes.

10.5.5. CH2 serial channel-activated interrupt P45

If serial channel CH2 in PLC TC400, TC500, TC600 and NS950 CPM-2S, CPM-1D set to **UNI** mode, the reception of a message calls the P45 process. Immediately before starting this process, the data received are written into the receiving zone. The data is changed here during the user program cycle, and so it is necessary to treat a possible manipulation with the data in the loop processes.

Process P45 serves especially to process a smaller data volume immediately. In case of larger data volumes or when it is not necessary to process the data immediately, we will leave the operation of the serial channel in the loop processes. The data received will always be passed at the cycle I/O scan.

Process P45 has the fifth. highest priority of interrupt process sequencing that is applied when there are more requests for more interrupt processes.

10.6. BREAKPOINT TREATMENT - PROCESSES P50 TO P57

After instructions BP 0 to BP 7 are executed, control is passed to processes P50 to P57 (the last digit in the number is identical to the operand of the BP instruction). In this process, it is possible to treat receiving of information from this position in the program.

10. User processes

When entering the P5x process, the stack is kept. After the process is finished, the stack and registers S0 to S1 are restored to their original values. This function facilitates debugging of the user program without confusing interventions in the program body.

Example 10.10

```
#reg uint debug[128],aux, index
;
P 0
    :
    BP    0           ;debugging instruction inserted
    :
    BP    0           ;debugging instruction inserted
    :
E 0
;
P 50
    WR    aux         ;postponing of stack top value
    LD    127         ;limit
    LD    index
    LD    aux
    WTB   debug
E 50
```

10.7. P60 SUBROUTINE PACKAGE

This process is not activated from the system program and serves only for storage of the file of subroutines called from various processes.

If the subroutines are positioned inside the active process, then they have to be skipped, since they can be neither at the beginning nor at the end of the process. This additionally requires instruction JMP and label L and the program becomes worse arranged.

Example 10.11

```
P 0
    :
    CAL   subroutine_name      ;subroutine call
    :
E 0
;
P 10
    :
    CAL   subroutine_name      ;subroutine call
    :
E 10
;
P 60
subroutine_name:
    :
    RET
E 60
```

11. INSTRUCTION SET

The TECOMAT PLC central units of a particular series have an instruction set of various ranges depending on the performance of the central unit.

Instruction set of E series

The central units of series E have stack layers 16 bits wide and contain an instruction set with the following features:

- ◆ bit logical operations
- ◆ basic operations of counters and timers
- ◆ basic operating instructions and transitions in the program
- ◆ comparison within 16 bit operands
- ◆ one-loop control

Instruction set of M and S series

Central units of series M and S have stack layers 16 bits wide and contain an instruction set, having additionally the following features as compared to series E:

- ◆ logic operations in 8 and 16 bit format
- ◆ advanced counter, timer and shift register operations
- ◆ arithmetic instructions, conversion and comparisons in 16 bit format
- ◆ expanded operating instructions, transitions in programs
- ◆ table instructions over the T tables and scratchpad memory
- ◆ sequencer instructions
- ◆ instructions realizing the set of logic operations, including counting of one-bits in the operand of 16 bit width
- ◆ the system contains 8 user stacks and the instructions for their switching - this is suitable to pass more parameters among functions not following each other, saving of the immediate state of the stack
- ◆ automatic conversion of operand length and intermediate results when combining instructions of various types or when combining logic instructions with arithmetic ones
- ◆ a set of system variables, in which the system time, system time units and their edges, communication variables, flag and command variables and system messages are realized
- ◆ Multiprogramming (multi-loop control) including interrupt processes contribute to reducing the feedback time and making programming easier
- ◆ the USI instructions optimally realize complex and special tasks (on the level of microprocessor instructions)

Instruction set of D and B series

Central units of series D and B have stack layers 16 bits wide and contain an instruction set, having additionally the following features as compared to series M and S:

- ◆ logic operations in 32 bit format
- ◆ arithmetic instructions, conversions and comparisons in 32 bit format
- ◆ conditional jumps according to comparison flags
- ◆ single precision floating point arithmetic instructions (real type)
- ◆ expanded table instructions with table of great range

11. Instruction set

- ◆ table instructions with structured access
- ◆ PID controller instructions
- ◆ system instructions for optimising the central units performance and supporting special services

Instruction set of C series

Central units of series C have stack layers 32 bits wide and contain an instruction set having additionally the following features, compared to the D and B series:

- ◆ load and write instructions with indirect addressing
- ◆ arithmetic instructions, conversions and comparisons with sign (negative numbers in binary complement)
- ◆ double precision floating point arithmetic instructions (lreal type)
- ◆ table instructions with tables of 32 bit format
- ◆ counters, shift registers and step sequencer in 32 bit format
- ◆ limit functions, value shift
- ◆ operator instructions of the operator panel

A detailed description of the instruction set is in the manuals TECOMAT PLC instruction set – 16 bit model (B, D, E, M, S series), order number TXV 001 05.02, and TECOMAT PLC instruction set – 32 bit model (C series), order number TXV 004 01.02.

12. USER INSTRUCTIONS

User instructions USI is a PLC instruction defined by the user that can be used for such operations, realization of which by means of other PLC instructions would be very difficult or impossible. This means the TECOMAT PLC instruction file is not closed and new instructions can be added when necessary without changing system software of central units.

User instructions support all central units except series E.

12.1. APPLICATION OF USI IN A USER PROGRAM

Deklarace #usi

Before application in the user program, instructions USI require a definition containing information in which directory and in which file the binary code of the USI instruction is saved. For this purposes, directive *#usi* is used. Its syntax is as follows:

```
#usi [index,] instruction_name = file_name
```

index - an optional number setting the index of the instruction being created
 After imposing of the index of the instruction by specifying this number, the other defined instructions are assigned indexes ascendingly beginning with number *index+1*. If item *index* is not used, the following instructions are automatically assigned the index ascendingly.

instruction_name - an optional symbolic name of the user instruction

file_name - the name of the binary disk file containing the executive part of the user instruction (it can contain the whole path)

Other applications are identical to the PLC standard instructions.

Example 12.1

```
#usi UserInstr = instfile          ;definition of USI
;
P 0
    :
    USI UserInstr          ;application of USI in the program
    :
E 0
```

12.2. USI FOR PARTICULAR SERIES OF CENTRAL UNITS

The machine codes of the USI instructions are not transferable among the series of the central units due to various processors used and different mapping of system memory.

Part of each machine code of the USI instructions is the marking of the series of central unit for which the code is designated. If we initiate by mistake a USI instruction on a wrong central unit, for which the instruction is not designated, then the unit reports an error of false user instruction (\$80 17 PC PC) and the program stops. Part of the installation of the Mosaic development environment is a series of the USI instructions already created and

12. User instructions

that can be used in your own programs. The description of these functions is part of installation.

The files containing the machine codes of the USI instructions have extensions *ui-*, where the last letter of the extension specifies the series of central unit for which the code is designated (see table 12.1). The files with extension **.dll* are the machine codes of the USI instructions for the PLC simulator in the Mosaic development environment.

Table 12.1 List of file extensions with USI instruction code

Central unit series	File extension
A	.uia
B	.uib
C	.uic
D	.uid
E	not supported
M	.uim
S	.uis
PLC simulator in Mosaic environment	.dll

The work with the USI instructions is supported by a compiler that additionally supports the assignment of the extension of the binary file in *#usi* declarations. In practice it means, that if the extension in the name of the file is not specified in the *#usi* declaration, it will be assigned automatically based on the central unit for which the code is generated.

12.3. CREATING A USER-DEFINED USI

The USI declaration is carried out in the C language. When programming it is recommended to come out from the publication *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie published by Prentice-Hall Inc. To enter the function, there is a header available describing and making accessible the stack and scratchpad of the PLC. The input parameters as well as the USI function outputs can be positioned anywhere in these structures. The USI function must be programmed by the user as follows:

```
#include "usi.h"    /* file pertaining to used CPU */

void NameUSI( p1, p2)
struct notePLC *p1;
struct accPLC *p2;
{
    :
    USI function body ;
    :
    return;
};
```

From the above mentioned declaration of the USI instruction it is obvious, that pointers to structures *notePLC* (PLC scratchpad) and *accPLC* (PLC stack) are passed to the USI function by the PLC system program. The declaration of these structures contains the *usi.h* file, a listing of which is shown in chapter 12.5. In case the program for the USI instruction comprises of more functions, it is necessary to declare the main function first, as it can be seen in the following example:

```
#include "usi.h"
int aux_function1(); /* prototype of auxiliary function for USI */
```

```

int aux_function2(); /* prototype of other auxiliary function */

void functionUSI(p1,p2)
struct notePLC *p1;
struct accPLC *p2;
{
    p2->a[0]=aux_function1() + aux_function2();
    return();
}
int aux_function1()
{
    /* particular enumeration for main function */
}
int aux_function2()
{
    /* other auxiliary function */
}

```

12.4. C LANGUAGE COMPILERS USED

For compilation of the USI instructions, the following compilers can be used:

For CPU of series A, B	Microware OS-9 / 68000 C Compiler Microware Systems Corporation
For CPU of series C	GNU 68K C Compiler Free Software Foundation, Inc.
For CPU of series D, M, S	Keil C-Compiler-51 Keil Elektronik GmbH

These compilers use the data types as mentioned in table 12.2. As it can be seen from the table, in case of the **int** type, the compilers for various processor types use a different size of the object being created which results in different ranges of number representation. If we want to write functions that could be used for all series of the PLC central units, it is therefore better to use in the functions the types declared in the *usi.h* file by means of the *#typedef* directive.

Table 12.2 Data types used by the compilers of the C language

Data type	Byte number		Internal representation
	Microware GNU	Keil	
char	1	1	two's complement binary
unsigned char	1	1	unsigned binary
short	2	2	two's complement binary
unsigned short	2	2	unsigned binary
int	4	2	two's complement binary
unsigned int	4	2	unsigned binary
long	4	4	two's complement binary
float	4	4	binary floating point
pointer to ...	4	3	address

The user instructions can also be written in the environments for application development for PCs in the C language (C++ Builder, as an example), that can be advantageously used for its debugging capability and only the final compilation will be executed with particular compilers.

12. User instructions

Attention! For user instructions written for central units of series M, S and D the following limitations are valid: it is not possible to use initialisation of local variables when declaring a variable, for example:

```
char array[4] = {{1,2,3,4}};
```

The variables must be initiated by their assignment during the function calculation, for example:

```
array[0]=1; array[1]=2; array[2]=3; array[3]=4;
```

This inconvenience is connected with the relocation of the USI instruction when being incorporated into the user program. Disrespecting of this procedure can result in unpredictable consequences!!!

12.5. EXAMPLE OF CREATION OF A USER-DEFINED USI INSTRUCTION

```
/* usi.h file for CPU of D series */
typedef unsigned long long_word;
typedef unsigned short word;
typedef signed short signed_word;
typedef unsigned char byte;
typedef signed char signed_byte;

/* structure declaration for access to PLC stack */
struct accPLC { /* PLC stack structure */
    word a[8]; /* stack levels */
};

/* constants for PLC stack zone size definition */
#define MAXX 128 /* number of X bytes in the stack */
#define MAXY 128 /* number of Y bytes in the stack */
#define MAXS 64 /* number of S bytes in the stack */
#define REZS 64 /* reserve in S zone */
#define MAXD 256 /* number of D bytes in the stack */
/* copied from program code */
#define MAXR 8192 /* number of R bytes in the stack */
struct notePLC { /* PLC scratchpad memory structure */
    u_char x[MAXX], /* X inputs image */
           y[MAXY], /* Y outputs image */
           s[MAXS], /* S system registers */
           rs[REZS], /* reserve for system zone */
           d[MAXD], /* copy of data D from user program */
           r[MAXR]; /* R user registers */
};
/* end of usi.h file for CPU of D series */
```

The MUL 16 instruction multiplies the value in A0 of the PLC stack by the value from the A1 layer, the result is saved into the A01 double-layer.

The source text of the MUL 16 instruction:

```
#include "usi.h"

void Mul16(p1, p2) /* binary multiplication A0 * A1 = A01 */
    struct notePLC *p1;
    struct accPLC *p2;
```

```

{
  union {
    long_word l;
    word      w[2];
  } result;

  result.l=((long_word)p2->a[0])*((long_word)p2->a[1]);
  p2->a[0]=result.w[1];          /* A0 lower bytes of result */
  p2->a[1]=result.w[0];          /* A1 higher bytes of result */
  return;
}

```

The program of the MUL 16 function is necessary to compile through the compiler of the language C into the machine code of the particular processor.

12.6 APPLICATION EXAMPLE FOR USI INSTRUCTION

The application of the user instruction into the PLC user program will be realized in the xPRO program by directive *#usi* that assigns a file with a binary instruction code to the symbolic name of the instruction.

Example 12.2

```

#usi MUL16 = mul16      ;definition of USI with automatic assignment
                       ;of file extension with instruction code according
                       ;to CPU series

#reg uint  va, vb
#reg udint vc
;
P 0
    LD    va          ;load first multiplicand
    LD    vb          ;load second multiplicand
    USI   MUL16       ;A01 = a.b
    WR    vc          ;write (save) result
E 0

```

12.7. COMMENTS

When writing user instructions it is necessary to respect a couple of rules resulting from the way of PLC operation. Especially, it is necessary to realize the following restrictions:

USI computation time - any USI instruction defined by user will consume the PLC processor time when it is called, as the other instructions do. This time is added to the execution time of one PLC cycle. The function that need a great number of iterations to get results, there is a risk of a disproportionate prolongation of the PLC cycle time and significant slowdown of its reaction for a unit change on the input which can result in stopping the PLC activity due to the exceeding of the maximum cycle time. These functions can be programmed in such a way that during each USI call, only the defined number of iterations is executed so that the processor time consumed is acceptable. Such programmed USI

12. User instructions

instruction then produces the result once a several PLC cycles, which is an acceptable solution in a great number of cases.

Memory requirements - the machine code of the USI instruction is part of the user program as well as the structures are to make the user program accessible for the PLC processor. From this point of view it is advisable to avoid such algorithms during programming that result in large machine codes. It concerns especially when using the libraries of the C language. A skilled programmer and well optimising compiler are a great contribution in this.

PLC processor type - the processor used brings certain limitations concerning, for example, main memory size, stack space, utilization of processor hardware, etc. From this point of view we recommend to consult the USI instructions with the developers of company TECO a.s.

A. APPENDIX

A.1. INSTRUCTION EXECUTION TIME FOR CENTRAL UNIT CPM-1E TECOMAT NS950

The instruction execution times do not include the effects of system processes that interrupt the execution of the user program. These processes are serial communications and services of some peripheral units. Due to their effect the cycle time becomes longer.

Overview of operand symbols used:

Z - scratchpad X, Y, S, R A - no operand (works only on the stack)
 # - constant n - numeric parameter

Data load and write instructions

Mnemo code	Execution time [μ s] (Code length [B])				Instruction description
	bool	Z byte usint sint	word uint int	# word uint int	
LD	63 (3)	53 (3)	61 (3)	50 (3)	Load direct data
LDC	64 (3)	54 (3)	63 (3)	52 (3)	Load complement data
WR	67 (3)	50 (3)	57 (3)	-	Write direct data
WRC	67 (3)	51 (3)	59 (3)	-	Write data complement
PUT	77 (3)	60 (3)	66 (3)	-	Conditional data write - condition fulfilled
	51	46	46	-	- condition not fulfilled

Logical instructions

Mnemo code	Execution time [μ s] (Code length [B])		Instruction description
	Z bool	A word uint int	
AND	66 (3)	50 (1)	AND with immediate operand
ANC	67 (3)	-	AND with negated operand
OR	66 (3)	50 (1)	OR with immediate operand
ORC	67 (3)	-	OR with negated operand
XOR	66 (3)	50 (1)	XOR with immediate operand
XOC	67 (3)	-	XOR with negated operand
SET	66 (3)	-	Conditional set
RES	67 (3)	-	Conditional reset
LET	79 (3)	-	Leading edge pulse

Counters, timers

Mnemo code	Execution time [μ s] (Code length [B])	Instruction description
CTU	186 (3)	Forward counter
CTD	186 (3)	Backward counter
TON	220 (3)	Timer on delay

Arithmetic instructions

Mnemo code	Execution time [μ s] (Code length [B])	Instruction description
	A word uint int	
EQ	92 (1)	Comparison (equality)

Appendix

Stack operations

Mnemo code	Execution time [μ s] (Code length [B])	Instruction description
POP n	$42 + 7n$ (3)	Stack shift (rotation) backwards by n levels

Jump and call instructions

Mnemo code	Execution time [μ s] (Code length [B])		Instruction description
	Transit	Jump	
JMP	-	114 (3)	Unconditional jump
JMD	42	120 (3)	Jump conditional on non-zero stack top value
L	36 (3)	-	Label n (jump and call target)

Operating instructions

Mnemo code	Execution time [μ s] (Code length [B])					Instruction description
	Transit	Jump	P41 - P49	P50 - P57	P62 - P64	
P	151 (3)	-	76	73	143	Process start
E	-	54 (3)	229	77	54	Unconditional process end
NOP	36 (3)	-	-	-	-	No operation

Appendix

Arithmetic instructions

Mnemo code	Execution time [μs] (Code length [B])						Instruction description
	Z		#		A		
	byte usint	word uint	byte usint	word uint	byte usint	word uint	
ADD	-	99 (3)	-	88 (3)	-	88 (1)	Addition with carry
SUB	-	101 (3)	-	90 (3)	-	90 (1)	Subtraction with carry
MUL	59 (3)	-	50 (3)	-	51 (1)	-	Multiplication
DIV	74 (3)	-	65 (3)	-	66 (1)	-	Division
INR	-	-	-	-	-	79 (1)	Incrementation (+ 1)
DCR	-	-	-	-	-	79 (1)	Decrementation (- 1)
EQ	-	103 (3)	-	92 (3)	-	92 (1)	Comparison (equality)
LT	-	103 (3)	-	92 (3)	-	92 (1)	Comparison (less than)
GT	-	104 (3)	-	93 (3)	-	93 (1)	Comparison (greater than)
BIN	-	-	-	-	-	80 (1)	Conversion of number to binary format
BCD	-	-	-	-	-	226 (1)	Conversion of number to BCD

Stack operations

Mnemo code	Execution time [μs] (Code length [B])	Instruction description
POP n	42 + 7n (3)	Stack shift (rotation) backwards by n levels
NXT	375 (3)	Activation of next stack in the queue
PRV	375 (3)	Activation of previous stack in the queue
CHG	356 (3)	Change active stack without backing S1 and S1
CHGS	373 (3)	Change active stack with backing up S1 and S1

Jump and call instructions

Mnemo code	Execution time [μs] (Code length [B])		Instruction description
	Transit	Jump	
JMP	-	114 (3)	Unconditional jump
JMD	42	120 (3)	Jump conditional on non-zero stack top value
JMC	42	120 (3)	Jump conditional on zero stack top value
JMI	-	115 (1)	Indirect jump
CAL	-	138 (3)	Unconditional subroutine call
CAD	42	144 (3)	Call conditional on non-zero stack top value
CAC	42	144 (3)	Call conditional on zero stack top value
CAI	-	137 (1)	Indirect subroutine call
RET	-	48 (1)	Unconditional return from subroutine
RED	40	54 (1)	Return from subroutine conditional on non-zero stack top value
REC	40	54 (1)	Return from subroutine conditional on zero stack top value
L	36 (3)	-	Label n (jump and call target)

Operating instructions

Mnemo code	Execution time [μs] (Code length [B])					Instruction description
	Transit	Jump	P41 - P49	P50 - P57	P62 - P64	
P	151 (3)	-	76	73	143	Process start
E	-	54 (3)	229	77	54	Unconditional process end
ED	42 (1)	65	240	88	65	Process end conditional on non-zero result
EC	42 (1)	65	240	88	65	Process end conditional on zero result
EOC	-	35 (1)	-	-	-	End of cycle (exceptional cycle end)
NOP	36 (3)	-	-	-	-	No operation
BP	-	189 (3)	-	-	-	Breakpoint

Table instructions

Mnemo code	Execution time [μs] (Code length [B])						Instruction description
	Z			T			
	bool	byte usint sint	word uint int	bool	byte usint sint	word uint int	
LTB	136 (3)	118 (3)	131 (3)	258 (3)	225 (3)	250 (3)	Load item from table
WTB	151 (3)	126 (3)	140 (3)	-	-	-	Write item to table
LMS	-	-	-	-	-	248 (3)	Item sequential load
WMS	-	-	-	-	-	362 (3)	Item sequential write
						+11	- time addition for 1 table item
FTB	-	68 (3) +49	75 (3) +55	-	182 (3) +39	203 (3) +46	Find item in table - time addition for 1 item being searched
FTM	-	66 (3) +61	66 (3) +90	-	192 (3) +51	192 (3) +80	Find part of item in table - time addition for 1 item being searched
FTS	-	71 (3) +51	64 (3) +74	-	184 (3) +41	191 (3) +64	Find item with sorting - time addition for 1 item being searched

Block operations

Mnemo code	Execution time [μs] (Code length [B])		Instruction description
	Z	T	
SRC	99 (3)	195 (3)	Specification of data source for transfer
MOV	188 (3) +17	392 (3) +28	Data block move - time addition for 1 block item
FIL	69 (3) +48	-	Fill block with constant - time addition for 1 block item

Arithmetic instructions

Mnemo code	Execution time [μs] (Code length [B])						Instruction description
	Z		#		A		
	byte usint	word uint	byte usint	word uint	byte usint	word uint	
ADD	-	19,5 (3)	-	18,1 (3)	-	16,8 (1)	Addition with carry
SUB	-	20,1 (3)	-	18,6 (3)	-	17,4 (1)	Subtraction with carry
MUL	13,6 (3)	-	11,4 (3)	-	11,0 (1)	-	Multiplication
DIV	15,9 (3)	-	14,5 (3)	-	13,4 (1)	-	Division
INR	-	-	-	-	-	14,6 (1)	Incrementation (+ 1)
DCR	-	-	-	-	-	14,6 (1)	Decrementation (- 1)
EQ	-	21,0 (3)	-	19,5 (3)	-	18,3 (1)	Comparison (equality)
LT	-	21,0 (3)	-	19,5 (3)	-	18,3 (1)	Comparison (less than)
GT	-	20,8 (3)	-	19,4 (3)	-	18,1 (1)	Comparison (greater than)
BIN	-	-	-	-	-	17,5 (1)	Conversion of number to binary format
BCD	-	-	-	-	-	47,6 (1)	Conversion of number to BCD

Stack operations

Mnemo code	Execution time [μs] (Code length [B])	Instruction description
POP n	10,7 + 1,4n (3)	Stack shift (rotation) backwards by n levels
NXT	92,4 (3)	Activation of next stack in the queue
PRV	92,4 (3)	Activation of previous stack in the queue
CHG	87,9 (3)	Change active stack without backing S1 and S1
CHGS	92,1 (3)	Change active stack with backing up S1 and S1

Jump and call instructions

Mnemo code	Execution time [μs] (Code length [B])		Instruction description
	Transit	Jump	
JMP	-	22,2 (3)	Unconditional jump
JMD	10,1	23,5 (3)	Jump conditional on non-zero stack top value
JMC	10,1	23,5 (3)	Jump conditional on zero stack top value
JMI	-	21,7 (1)	Indirect jump
CAL	-	25,3 (3)	Unconditional subroutine call
CAD	10,1	26,6 (3)	Call conditional on non-zero stack top value
CAC	10,1	26,6 (3)	Call conditional on zero stack top value
CAI	-	24,2 (1)	Indirect subroutine call
RET	-	10,3 (1)	Unconditional return from subroutine
RED	9,0	11,6 (1)	Return from subroutine conditional on non-zero stack top value
REC	9,0	11,6 (1)	Return from subroutine conditional on zero stack top value
L	8,9 (3)	-	Label n (jump and call target)

Operating instructions

Mnemo code	Execution time [μs] (Code length [B])					Instruction description
	Transit	Jump	P41 - P49	P50 - P57	P62 - P64	
P	30,6 (3)	-	14,8	14,1	28,2	Process start
E	-	11,2 (3)	53,2	51,4	11,2	Unconditional process end
ED	9,0 (1)	12,8	54,8	53,0	12,8	Process end conditional on non-zero result
EC	9,0 (1)	12,8	54,8	53,0	12,8	Process end conditional on zero result
EOC	-	8,1 (1)	-	-	-	End of cycle (exceptional cycle end)
NOP	8,9 (3)	-	-	-	-	No operation
BP	-	72,0 (3)	-	-	-	Breakpoint

Appendix

Table instructions

Mnemo code	Execution time [μ s] (Code length [B])						Instruction description
	Z			T			
	bool	byte usint sint	word uint int	bool	byte usint sint	word uint int	
LTB	30,6 (3)	25,1 (3)	27,5 (3)	46,8 (3)	37,6 (3)	40,5 (3)	Load item from table
WTB	33,1 (3)	26,0 (3)	28,8 (3)	-	-	-	Write item to table
LMS	-	-	-	-	-	47,9 (3)	Item sequential load
WMS	-	-	-	-	-	61,7 (3)	Item sequential write - time addition for 1 table item
FTB	-	19,7 (3) +3,1	22,1 (3) +3,6	-	32,2 (3) +3,1	34,7 (3) +3,6	Find item in table - time addition for 1 item being searched
FTM	-	20,3 (3) +4,3	21,9 (3) +7,4	-	32,9 (3) +4,3	35,1 (3) +7,4	Find part of item in table - time addition for 1 item being searched
FTS	-	19,7 (3) +3,6	21,0 (3) +6,3	-	32,7 (3) +3,6	33,6 (3) +6,3	Find item with sorting - time addition for 1 item being searched

Block operations

Mnemo code	Execution time [μ s] (Code length [B])		Instruction description
	Z	T	
SRC	21,3 (3)	35,4 (3)	Specification of data source for transfer
MOV	49,0 (3) +3,1	74,3 (3) +5,4	Data block move - time addition for 1 block item
FIL	20,4 (3) +2,2	-	Fill block with constant - time addition for 1 block item

Appendix

Timers, shift registers, counters, step controller

Mnemo code	Execution time [μs] (Code length [B])	Instruction description
CTU	40,3 (3)	Forward counter
CTD	40,3 (3)	Backward counter
CNT	48,3 (3)	Bidirectional counter
SFL	41,4 (3)	Shift register left
SFR	41,4 (3)	Shift register right
TON	47,0 (3)	Timer on delay
TOF	47,2 (3)	Timer off delay
RTO	50,3 (3)	Integrating timer, time measurement
IMP	46,7 (3)	Timer - specified pulse length generator
STE	29,7 (3)	Step sequencer (stepper) - state change
	23,7	- state not changed

Arithmetic instructions

Mnemo code	Execution time [μs] (Code length [B])									Instruction description
	Z			#			A			
	byte usint	word uint	dword uint	byte usint	word uint	dword uint	byte usint	word uint	dword uint	
ADD	-	19,5 (3)	-	-	18,1 (3)	-	-	16,8 (1)	-	Addition with carry
ADX	13,6 (4)	14,3 (4)	18,6 (4)	-	-	-	-	-	-	Addition
ADL	-	-	-	-	-	17,0 (6)	-	-	16,3 (2)	Addition
SUB	-	20,1 (3)	-	-	18,6 (3)	-	-	17,4 (1)	-	Subtraction with carry
SUX	14,1 (4)	14,8 (4)	19,5 (4)	-	-	-	-	-	-	Subtraction
SUL	-	-	-	-	-	17,4 (6)	-	-	16,5 (2)	Subtraction
MUL	13,6 (3)	-	-	11,4 (3)	-	-	11,0 (1)	-	-	Multiplication
MUD	-	54 (4)	-	-	52 (4)	-	-	51 (2)	-	Multiplication
DIV	15,9 (3)	-	-	14,5 (3)	-	-	13,4 (1)	-	-	Division
DID	-	314 (4)	-	-	313 (4)	-	-	311 (2)	-	Division
INR	12,8 (4)	13,6 (4)	13,6 (4)	-	-	-	-	14,6 (1)	-	Incrementation (+ 1)
DCR	14,5 (4)	15,4 (4)	15,4 (4)	-	-	-	-	14,6 (1)	-	Decrementation (- 1)
EQ	-	21,0 (3)	-	-	19,5 (3)	-	-	18,3 (1)	-	Comparison (equality)
LT	-	21,0 (3)	-	-	19,5 (3)	-	-	18,3 (1)	-	Comparison (less than)
GT	-	20,8 (3)	-	-	19,4 (3)	-	-	18,1 (1)	-	Comparison (greater than)
CMP	17,5 (4)	18,8 (4)	24,1 (4)	-	17,5 (4)	-	-	17,5 (2)	-	Comparison
CML	-	-	-	-	-	23,3 (6)	-	-	21,2 (2)	Comparison
BIN	-	-	-	-	-	-	-	17,5 (1)	-	Conv. of number to bin. f.
BIL	-	-	-	-	-	-	-	-	173 (2)	Conv. of number to bin. f.
BCD	-	-	-	-	-	-	-	47,6 (1)	-	Conversion of number to BCD
BCL	-	-	-	-	-	-	-	-	314 (2)	Conversion of number to BCD

Stack operations

Mnemo code	Execution time [μs] (Code length [B])	Instruction description
POP n	10,7 + 1,4n (3)	Stack shift (rotation) backwards by n levels
NXT	92,4 (3)	Activation of next stack in the queue
PRV	92,4 (3)	Activation of previous stack in the queue
CHG	87,9 (3)	Change active stack without backing S0 and S1
CHGS	92,1 (3)	Change active stack with backing up S0 and S1
LAC	23,9 (4)	Load values from top of chosen stack
WAC	22,6 (4)	Write value on top of chosen stack

Jump and call instructions

Mnemo code	Execution time [μs] (Code length [B])		Instruction description
	Transit	Jump	
JMP	-	22,2 (3)	Unconditional jump
JMD	10,1	23,5 (3)	Jump conditional on non-zero stack top value
JMC	10,1	23,5 (3)	Jump conditional on zero stack top value
JMI	-	21,7 (1)	Indirect jump
JZ	11,4	24,8 (4)	Jump conditional on non-zero value of flag ZR
JNZ	11,4	24,8 (4)	Jump conditional on zero value of flag ZR
JC	11,4	24,8 (4)	Jump conditional on non-zero value of flag CO
JNC	11,4	24,8 (4)	Jump conditional on zero value of flag CO
JS	11,4	24,8 (4)	Jump conditional on non-zero value of flag S1.0
JNS	11,4	24,8 (4)	Jump conditional on zero value of flag S1.0
CAL	-	25,3 (3)	Unconditional subroutine call
CAD	10,1	26,6 (3)	Call conditional on non-zero stack top value
CAC	10,1	26,6 (3)	Call conditional on zero stack top value
CAI	-	24,2 (1)	Indirect subroutine call
RET	-	10,3 (1)	Unconditional return from subroutine
RED	9,0	11,6 (1)	Return from subroutine conditional on non-zero stack top value
REC	9,0	11,6 (1)	Return from subroutine conditional on zero stack top value
L	8,9 (3)	-	Label n (jump and call target)

Operational instructions

Mnemo code	Execution time [μs] (Code length [B])					Instruction description
	Transit	Jump	P41 - P49	P50 - P57	P62 - P64	
P	30,6 (3)	49,0 *	14,8	14,1	28,2	Process start (*jump to label of given SEQ)
E	-	11,2 (3)	53,2	51,4	11,2	Unconditional process end
ED	9,0 (1)	12,8	54,8	53,0	12,8	Process end conditional on non-zero result
EC	9,0 (1)	12,8	54,8	53,0	12,8	Process end conditional on zero result
EOC	-	8,1 (1)	-	-	-	End of cycle (exceptional cycle end)
NOP	8,9 (3)	-	-	-	-	No operation
BP	-	72,0 (3)	-	-	-	Breakpoint
SEQ	14,5 (3)	18,8	-	-	-	Conditional process interrupt

Table instructions

Mnemo code	Execution time [μs] (Code length [B])						Instruction description
	Z			T			
	bool	byte usint sint	word uint int	bool	byte usint sint	word uint int	
LTB	30,6 (3)	25,1 (3)	27,5 (3)	46,8 (3)	37,6 (3)	40,5 (3)	Load item from table
WTB	33,1 (3)	26,0 (3)	28,8 (3)	52,3 (4)	45,0 (4)	50,1 (4)	Write item to table
LMS	-	-	-	-	-	47,9 (3)	Item sequential load
WMS	-	-	-	-	-	61,7 (3)	Item sequential write
FTB	23,0 (4)	19,7 (3)	22,1 (3)	38,7 (4)	32,2 (3)	34,7 (3)	Find item in table
	+4,1	+3,1	+3,6	+4,1	+3,1	+3,6	- time addition for 1 item being searched
FTM	-	20,3 (3)	21,9 (3)	-	32,9 (3)	35,1 (3)	Find part of item in table
	-	+4,3	+7,4	-	+4,3	+7,4	- time addition for 1 item being searched
FTS	-	19,7 (3)	21,0 (3)	-	32,7 (3)	33,6 (3)	Find item with sorting
	-	+3,6	+6,3	-	+3,6	+6,3	- time addition for 1 item being searched

Appendix

Block operations

Mnemo code	Execution time [μs] (Code length [B])			Instruction description
	Z	T	A	
SRC	21,3 (3)	35,4 (3)	-	Specification of data source for transfer
MOV	49,0 (3) +3,1	74,3 (3) +5,4	-	Data block move - time addition for 1 block item
MTN	-	-	35,6 (2) +3,1	Move table to scratchpad - time addition for 1 table item
MNT	-	-	40,5 (2) +4,0	Fill table from scratchpad - time addition for 1 table item
FIL	20,4 (3) +2,2	-	-	Fill block with constant - time addition for 1 block item

Operations with structured tables

Mnemo code	Execution time [μs] (Code length [B])		Instruction description
LDS	47,6 (2) +3,1		Load item from T structured table - time addition for 1 byte of item
WRS	57,1 (2) +5,2		Write item to T structured table - time addition for 1 byte of item
FIS	28,0 (2) +2,2		Fill item of structured table in scratchpad - time addition for 1 byte of item
FIT	51,4 (2) +4,3		Fill item of T structured table - time addition for 1 byte of item
FNS	23,9 (2) +4,3		Find item of structured table in scratchpad - time addition for 1 item
FNT	36,2 (2) +6,9		Find item of T structured table - time addition for 1 item

Floating point arithmetic instructions (time data is orientational only, depending on input value)

Mnemo code	Execution time [μs] (Code length [B])			Instruction description
	Z	#	A	
ADF	83 (4)	81 (6)	79 (2)	Addition
SUF	83 (4)	81 (6)	79 (2)	Subtraction
MUF	125 (4)	123 (6)	121 (2)	Multiplication
DIF	332 (4)	329 (6)	328 (2)	Division
CMF	54 (4)	52 (6)	50 (2)	Comparison
CEI	-	-	550 (2)	Round up
FLO	-	-	520 (2)	Round down
ABS	-	-	11,6 (2)	Absolute value
LOG	-	-	1580 (2)	Decimal logarithm
LN	-	-	1580 (2)	Natural logarithm
EXP	-	-	4200 (2)	Exponential function
POW	-	-	4200 (2)	Power
SQR	-	-	710 (2)	Square root
HYP	-	-	980 (2)	Euclidean distance
SIN	-	-	1320 (2)	Sine
ASN	-	-	2590 (2)	Arcsine
COS	-	-	1650 (2)	Cosine
ACS	-	-	2770 (2)	Arccosine
TAN	-	-	2410 (2)	Tangent
ATN	-	-	1650 (2)	Arctangent
UWF	-	-	40 ÷ 170 (2)	Conversion of uint to real
IWF	-	-	40 ÷ 170 (2)	Conversion of int to real
ULF	-	-	40 ÷ 170 (2)	Conversion of uint to real
ILF	-	-	40 ÷ 170 (2)	Conversion of dint to real
UFW	-	-	110 ÷ 200 (2)	Conversion of real to uint
IFW	-	-	110 ÷ 200 (2)	Conversion of real to int
UFL	-	-	110 ÷ 200 (2)	Conversion of real to uint
IFL	-	-	110 ÷ 200 (2)	Conversion of real to dint

PID controller instructions (time data is orientational, they depend on selected functions and input values)

Mnemo code	Execution time [μs] (Code length [B]) A	Instruction description
CNV	900 (2)	Data conversion and processing from analog units
PID	2000 ÷ 10000 (2)	PID controller

Operations with ASCII characters

Mnemo code	Execution time [μs] (Code length [B]) A		Instruction description
	word	float	
BAS	21,7 (2)	-	Conversion of number to ASCII
ASB	22,4 (2)	-	Conversion of number from ASCII
STF		2000 (2)	Conversion of ASCII string to float format
FST		1000 (2)	Conversion of float format to ASCII string

A.5. INSTRUCTION EXECUTION TIME FOR CENTRAL UNITS CPM-1B, CPM-2B TECOMAT NS950

The instruction execution times do not include the effects of system processes that interrupt the execution of the user program. These processes are serial communications and services of some peripheral units. Due to their effect the cycle time becomes longer.

Overview of operand symbols used:

Z - scratchpad X, Y, S, D, R T - tables
 # - constant A - no operand (works only on the stack)
 U - peripheral unit physical address n - numeric parameter

Data load and write instructions

Mnemo code	Execution time [µs] (Code length [B])								Instruction description
	Z				#		U		
	bool	byte usint sint	word uint int	dword udint dint real	word uint int	dword udint dint real	byte usint sint	word uint int	
LD	4,0 (4)	3,2 (4)	4,2 (4)	5,0 (4)	2,8 (4)	-	200 (4)	210 (4)	Load direct data
LDL	-	-	-	-	-	3,6 (6)	-	-	Load direct data
LDC	4,0 (4)	3,4 (4)	4,4 (4)	5,2 (4)	2,9 (4)	-	-	-	Load complement data
WR	3,1 (4)	2,1 (4)	3,7 (4)	4,2 (4)	-	-	130 (4)	130 (4)	Write direct data
WRC	3,1 (4)	2,3 (4)	3,9 (4)	4,4 (4)	-	-	-	-	Write data complement
WRA	-	4,1 (4)	4,4 (4)	6,0 (4)	-	-	-	-	Write direct data with alternation
PUT									Conditional data write
	4,2 (4)	3,2 (4)	4,8 (4)	5,3 (4)	-	-	-	-	- condition fulfilled
	2,8	2,8	2,8	2,8	-	-	-	-	- condition not fulfilled

Logical instructions

Mnemo code	Execution time [µs] (Code length [B])							Instruction description
	Z		#			A		
	bool	byte usint sint	word uint int	word uint int	dword udint dint	word uint int	dword udint dint	
AND	3,2 (4)	3,1 (4)	3,5 (4)	2,6 (4)	-	3,2 (2)	-	AND with immediate operand
ANL	-	-	-	-	3,7 (6)	-	4,9 (2)	AND with immediate operand
ANC	3,6 (4)	3,3 (4)	3,5 (4)	-	-	-	-	AND with negated operand
OR	3,2 (4)	3,1 (4)	3,5 (4)	2,6 (4)	-	3,2 (2)	-	OR with immediate operand
ORL	-	-	-	-	3,7 (6)	-	4,9 (2)	OR with immediate operand
ORC	3,6 (4)	3,3 (4)	3,5 (4)	-	-	-	-	OR with negated operand
XOR	3,2 (4)	3,1 (4)	3,5 (4)	2,6 (4)	-	3,2 (2)	-	XOR with immediate operand
XOL	-	-	-	-	3,7 (6)	-	4,9 (2)	XOR with immediate operand
XOC	3,6 (4)	3,3 (4)	3,5 (4)	-	-	-	-	XOR with negated operand
NEG	-	-	-	-	-	2,6 (2)	-	Stack top negation
NGL	-	-	-	-	-	-	3,6 (2)	Stack top negation
SET	3,3 (4)	2,9 (4)	3,3 (4)	-	-	-	-	Conditional set
RES	3,3 (4)	2,9 (4)	3,3 (4)	-	-	-	-	Conditional reset
LET	5,0 (4)	3,5 (4)	4,3 (4)	-	-	-	-	Leading edge pulse
BET	4,8 (4)	3,9 (4)	4,3 (4)	-	-	-	-	Pulse from any edge
FLG	-	-	-	-	-	16,1 (2)	-	Logical functions A0
STK	-	-	-	-	-	16,4 (2)	-	Swap of stack levels to A0
ROL n	-	-	-	-	-	4,2 (4)	-	Number rotation left
ROR n	-	-	-	-	-	4,2 (4)	-	Rotation of number right
SWP	-	-	-	-	-	3,4 (2)	-	Swap of top and low byte in A0
SWL	-	-	-	-	-	-	4,3 (2)	Swap of layers A0 and A1

Timers, shift registers, counters, step controller

Mnemo code	Execution time [μs] (Code length [B])	Instruction description
CTU	12,7 (4)	Forward counter
CTD	13,1 (4)	Backward counter
CNT	16,4 (4)	Bidirectional counter
SFL	12,6 (4)	Shift register left
SFR	12,6 (4)	Shift register right
TON	15,6 (4)	Timer on delay
TOF	17,1 (4)	Timer off delay
RTO	17,6 (4)	Integrating timer, time measurement
IMP	16,8 (4)	Timer - specified pulse length generator
STE	4,8 (4)	Step sequencer (stepper)

Arithmetic instructions

Mnemo code	Execution time [μs] (Code length [B])									Instruction description
	Z			#			A			
	byte usint	word uint	dword udint	byte usint	word uint	dword udint	byte usint	word uint	dword udint	
ADD	-	6,2 (4)	-	-	5,8 (4)	-	-	6,5 (2)	-	Addition with carry
ADX	3,1 (4)	3,4 (4)	6,4 (4)	-	-	-	-	-	-	Addition
ADL	-	-	-	-	-	5,5 (6)	-	-	6,7 (2)	Addition
SUB	-	6,0 (4)	-	-	5,7 (3)	-	-	6,6 (2)	-	Subtraction with carry
SUX	3,1 (4)	3,2 (4)	5,9 (4)	-	-	-	-	-	-	Subtraction
SUL	-	-	-	-	-	5,5 (6)	-	-	6,7 (2)	Subtraction
MUL	4,6 (4)	-	-	4,4 (4)	-	-	5,2 (2)	-	-	Multiplication
MUD	-	5,7 (4)	-	-	5,5 (4)	-	-	5,5 (2)	-	Multiplication
DIV	6,2 (4)	-	-	6,0 (4)	-	-	6,8 (2)	-	-	Division
DID	-	32,1 (4)	-	-	31,9 (4)	-	-	31,9 (2)	-	Division
INR	3,0 (4)	3,8 (4)	4,5 (4)	-	-	-	-	5,8 (2)	-	Incrementation (+ 1)
DCR	4,2 (4)	4,7 (4)	5,3 (4)	-	-	-	-	5,8 (2)	-	Decrementation (- 1)
EQ	-	6,5 (4)	-	-	6,2 (4)	-	-	6,9 (2)	-	Comparison (equality)
LT	-	6,5 (4)	-	-	6,2 (4)	-	-	6,9 (2)	-	Comparison (less than)
GT	-	6,5 (4)	-	-	6,2 (4)	-	-	6,9 (2)	-	Comparison (greater than)
CMP	5,1 (4)	5,1 (4)	6,5 (4)	-	4,7 (4)	-	-	4,8 (2)	-	Comparison
CML	-	-	-	-	-	5,9 (6)	-	-	7,6 (2)	Comparison
BIN	-	-	-	-	-	-	-	6,7 (2)	-	Conv. of number to bin. f.
BIL	-	-	-	-	-	-	-	-	17,8 (2)	Conv. of number to bin. f.
BCD	-	-	-	-	-	-	-	21,3 (2)	-	Conversion of number to BCD
BCL	-	-	-	-	-	-	-	-	42,9 (2)	Conversion of number to BCD

Stack operations

Mnemo code	Execution time [μs] (Code length [B])	Instruction description
POP n	3,1 (4)	Stack shift (rotation) backwards by n levels
NXT	11,0 (4)	Activation of next stack in the queue
PRV	10,9 (4)	Activation of previous stack in the queue
CHG	10,1 (4)	Change active stack without backing S0 and S1
CHGS	10,5 (4)	Change active stack with backing up S0 and S1
LAC	6,4 (4)	Load values from top of chosen stack
WAC	5,6 (4)	Write value on top of chosen stack

Appendix

Jump and call instructions

Mnemo code	Execution time [μs] (Code length [B])		Instruction description
	Transit	Jump	
JMP	-	4,7 (4)	Unconditional jump
JMD	2,0	5,2 (4)	Jump conditional on non-zero stack top value
JMC	2,0	5,2 (4)	Jump conditional on zero stack top value
JMI	-	5,0 (2)	Indirect jump
JZ	2,0	5,2 (4)	Jump conditional on non-zero value of flag ZR
JNZ	2,0	5,2 (4)	Jump conditional on zero value of flag ZR
JC	2,0	5,2 (4)	Jump conditional on non-zero value of flag CO
JNC	2,0	5,2 (4)	Jump conditional on zero value of flag CO
JS	2,0	5,2 (4)	Jump conditional on non-zero value of flag S1.0
JNS	2,0	5,2 (4)	Jump conditional on zero value of flag S1.0
CAL	-	6,4 (4)	Unconditional subroutine call
CAD	2,0	7,9 (4)	Call conditional on non-zero stack top value
CAC	2,0	7,9 (4)	Call conditional on zero stack top value
CAI	-	7,7 (2)	Indirect subroutine call
RET	-	3,0 (2)	Unconditional return from subroutine
RED	1,8	3,5 (2)	Return from subroutine conditional on non-zero stack top value
REC	1,8	3,5 (2)	Return from subroutine conditional on zero stack top value
L	1,5 (4)	-	Label n (jump and call target)

Operating instructions

Mnemo code	Execution time [μs] (Code length [B])					Instruction description
	Transit	Jump	P41 - P49	P50 - P57	P62 - P64	
P	1,5 (4)	5,5*	4,5	4,5	1,5	Process start (*jump to label of given SEQ)
E	-	2,2 (4)	16,9	16,9	2,2	Unconditional process end
ED	1,8 (2)	3,7	18,5	18,5	3,7	Process end conditional on non-zero result
EC	1,8 (2)	3,7	18,5	18,5	3,7	Process end conditional on zero result
EOC	-	3,4 (2)	-	-	-	End of cycle (exceptional cycle end)
NOP	1,5 (4)	-	-	-	-	No operation
BP	-	19,5 (4)	-	-	-	Breakpoint
SEQ	4,5 (4)	6,0	-	-	-	Conditional process interrupt

Table instructions

Mnemo code	Execution time [μs] (Code length [B])						Instruction description
	Z			T			
	bool	byte usint sint	word uint int	bool	byte usint sint	word uint int	
LTB	11,2 (4)	9,5 (4)	10,6 (4)	10,7 (4)	8,9 (4)	10,3 (4)	Load item from table
WTB	10,5 (4)	8,7 (4)	9,9 (4)	14,5 (4)	12,5 (4)	15,3 (4)	Write item to table
FTB	10,9 (4)	7,8 (4)	9,1 (4)	9,4 (4)	8,6 (4)	10,2 (4)	Find item in table
	+1,0	+1,0	+1,1	+1,0	+1,0	+1,1	- time addition for 1 item being searched
FTM	-	8,3 (4)	9,0 (4)	-	9,4 (4)	10,0 (4)	Find part of item in table
		+1,5	+3,5		+1,5	+3,5	- time addition for 1 item being searched
FTS	-	7,8 (4)	8,3 (4)	-	8,6 (4)	9,4 (4)	Find item with sorting
		+1,0	+2,1		+1,0	+2,1	- time addition for 1 item being searched

Block operations

Mnemo code	Execution time [µs] (Code length [B])			Instruction description
	Z	T	A	
SRC	5,1 (4)	6,0 (4)	-	Specification of data source for transfer
MOV	7,8 (4) +1,0	10,4 (4) +1,0	-	Data block move - time addition for 1 block item
MTN	-	-	7,5 (2) +1,0	Move table to scratchpad - time addition for 1 table item
MNT	-	-	7,5 (2) +1,5	Fill table from scratchpad - time addition for 1 table item
FIL	7,5 (4) +1,0	-	-	Fill block with constant - time addition for 1 block item

Operations with structured tables

Mnemo code	Execution time [µs] (Code length [B])			Instruction description
	Z	T	A	
LDS	-	12,5 (2) +1,0	-	Load item from T structured table - time addition for 1 byte of item
WRS	-	14,0 (2) +1,5	-	Write item to T structured table - time addition for 1 byte of item
FIS	-	8,5 (2) +1,0	-	Fill item of structured table in scratchpad - time addition for 1 byte of item
FIT	-	8,9 (2) +1,2	-	Fill item of T structured table - time addition for 1 byte of item
FNS	-	8,5 (2) +1,2	-	Find item of structured table in scratchpad - time addition for 1 item
FNT	-	8,9 (2) +1,4	-	Find item of T structured table - time addition for 1 item

Floating point arithmetic instructions (time data is orientational only, depending on input value)

Mnemo code	Execution time [µs] (Code length [B])			Instruction description
	Z	#	A	
ADF	93 (4)	97 (6)	97 (2)	Addition
SUF	94 (4)	98 (6)	98 (2)	Subtraction
MUF	79 (4)	83 (6)	83 (2)	Multiplication
DIF	80 (4)	84 (6)	84 (2)	Division
CMF	57 (4)	61 (6)	61 (2)	Comparison
CEI	-	-	83 (2)	Round up
FLO	-	-	78 (2)	Round down
ABS	-	-	5,1 (2)	Absolute value
LOG	-	-	220 (2)	Decimal logarithm
LN	-	-	220 (2)	Natural logarithm
EXP	-	-	4500 (2)	Exponential function
POW	-	-	8000 (2)	Power
SQR	-	-	1050 (2)	Square root
HYP	-	-	1200 (2)	Euclidean distance
SIN	-	-	3300 (2)	Sine
ASN	-	-	4200 (2)	Arcsine
COS	-	-	3000 (2)	Cosine
ACS	-	-	4200 (2)	Arccosine
TAN	-	-	5800 (2)	Tangent
ATN	-	-	2800 (2)	Arctangent
UWF	-	-	40 ÷ 100 (2)	Conversion of uint to real
IWF	-	-	40 ÷ 100 (2)	Conversion of int to real
ULF	-	-	40 ÷ 100 (2)	Conversion of uint to real
ILF	-	-	40 ÷ 100 (2)	Conversion of dint to real
UFW	-	-	50 ÷ 130 (2)	Conversion of real to uint
IFW	-	-	50 ÷ 130 (2)	Conversion of real to int
UFL	-	-	50 ÷ 130 (2)	Conversion of real to uint
IFL	-	-	50 ÷ 130 (2)	Conversion of real to dint

Appendix

PID controller instructions (time data is orientational, it depends on selected functions and input values)

Mnemo code	Execution time [μ s] (Code length [B]) A	Instruction description
CNV	500 (2)	Data conversion and processing from analog units
PID	1000 ÷ 2000 (2)	PID controller

Operations with ASCII characters

Mnemo code	Execution time [μ s] (Code length [B]) A			Instruction description
	word	uint int	real	
BAS	8,9	(2)	-	Conversion of number to ASCII
ASB	3,6	(2)	-	Conversion of number from ASCII
STF			650 (2)	Conversion of ASCII string to float format
FST			340 (2)	Conversion of float format to ASCII string

**A.6. INSTRUCTION EXECUTION TIME FOR CENTRAL UNITS CP-7001, CP-7002
TECOMAT TC700 AND TECOMAT TC650**

The instruction execution times are orientational and do not include the effects of system processes that interrupt the execution of the user program. These processes are serial communications and operation of cash memories. Due to their effect the cycle time becomes longer.

Overview of operand symbols used:

Z - scratchpad X, Y, S, D, R
- constant
n - numeric parameter

T - tables
A - no operand (works only on the stack)

Data load and write instructions

Mnemo code	Execution time [µs] (Code length [B])							Instruction description
	Z					#		
	bool	byte usint sint	word uint int	dword udint dint real	lreal	dword udint dint real	lreal	
LD	0,9 (4)	0,8 (4)	1,0 (4)	1,3 (4)	2,1 (4)	0,9 (6)	-	Load direct data
LDQ	-	-	-	-	-	-	1,3 (10)	Load direct data
LDC	0,9 (4)	0,9 (4)	1,0 (4)	1,3 (4)	-	-	-	Load complement data
WR	0,8 (4)	0,6 (4)	0,7 (4)	1,1 (4)	1,9 (4)	-	-	Write direct data
WRC	0,9 (4)	0,6 (4)	0,7 (4)	1,1 (4)	-	-	-	Write data complement
WRA	-	0,9 (4)	1,1 (4)	1,8 (4)	-	-	-	Write direct data with alternation
PUT								Conditional data write
	1,0 (4)	0,8 (4)	0,9 (4)	1,3 (4)	-	-	-	- condition fulfilled
	0,7	0,7	0,7	0,7	-	-	-	- condition not fulfilled
LEA	0,8 (6)	0,7 (6)	0,7 (6)	0,7 (6)	0,7 (6)	-	-	Load address

Mnemo code	Execution time [µs] (Code length [B])					Instruction description
	A					
	bool	byte usint sint	word uint int	dword udint dint real	lreal	
LDIB	1,0 (2)	-	-	-	-	Load data of 1 bit width from address at A0
LDI	-	0,9 (2)	-	-	-	Load data of 8 bit width from address at A0
LDIW	-	-	1,0 (2)	-	-	Load data of 16 bit width from address at A0
LDIL	-	-	-	1,3 (2)	-	Load data of 32 bit width from address at A0
LDIQ	-	-	-	-	2,2 (2)	Load data of 64 bit width from address at A0
WRIB	1,1 (2)	-	-	-	-	Write data of 1 bit width to address at A0
WRI	-	0,8 (2)	-	-	-	Write data of 8 bit width to address at A0
WRIW	-	-	0,9 (2)	-	-	Write data of 16 bit width to address at A0
WRIL	-	-	-	1,3 (2)	-	Write data of 32 bit width to address at A0
WRIQ	-	-	-	-	2,2 (2)	Write data of 64 bit width to address at A0

Appendix

Logical instructions

Mnemo code	Execution time [μ s] (Code length [B])							Instruction description		
	Z				#				A	
	bool	byte usint sint	word uint int	dword udint dint	dword udint dint	word uint int	dword udint dint			
AND	0,9 (4)	0,8 (4)	0,9 (4)	1,3 (4)	0,9 (6)	-	0,8 (2)	AND with immediate operand		
ANC	0,9 (4)	0,9 (4)	1,0 (4)	1,4 (4)	-	-	0,9 (2)	AND with negated operand		
OR	0,9 (4)	0,8 (4)	0,9 (4)	1,3 (4)	0,9 (6)	-	0,8 (2)	OR with immediate operand		
ORC	0,9 (4)	0,9 (4)	1,0 (4)	1,4 (4)	-	-	0,9 (2)	OR with negated operand		
XOR	0,9 (4)	0,8 (4)	0,9 (4)	1,3 (4)	0,9 (6)	-	0,8 (2)	XOR with immediate operand		
XOC	0,9 (4)	0,9 (4)	1,0 (4)	1,4 (4)	-	-	0,9 (2)	XOR with negated operand		
NEG	-	-	-	-	-	-	0,7 (2)	Stack top negation		
SET	0,6 (4)	0,6 (4)	0,6 (4)	0,6 (4)	-	-	-	Conditional set		
RES	0,6 (4)	0,6 (4)	0,6 (4)	0,6 (4)	-	-	-	Conditional reset		
LET	1,2 (4)	1,0 (4)	1,1 (4)	2,0 (4)	-	-	-	Leading edge pulse		
BET	1,2 (4)	1,0 (4)	1,1 (4)	2,0 (4)	-	-	-	Pulse from any edge		
FLG	-	-	-	-	-	2,9 (2)	-	Logical functions A0		
STK	-	-	-	-	-	-	3,6 (2)	Swap of stack levels to A0		
ROL n	-	-	-	-	-	1,1 (4)	-	Rotation of number left n-times		
ROL	-	-	-	-	-	-	1,4 (2)	Rotation left		
ROR n	-	-	-	-	-	1,1 (4)	-	Rotation of number right n-times		
ROR	-	-	-	-	-	-	1,4 (2)	Rotation right		
SHL	-	-	-	-	-	-	1,2 (2)	Shift of number left n-times		
SHR	-	-	-	-	-	-	1,2 (2)	Shift of number right n-times		
SWP	-	-	-	-	-	0,9 (2)	-	Swap of top and low byte in A0		
SWL	-	-	-	-	-	-	0,7 (2)	Swap of layers A0 and A1		

Timers, shift registers, counters, step controller

Mnemo code	Execution time [μ s] (Code length [B])		Instruction description
	#		
	word uint	dword udint	
CTU	3,2 (4)	3,4 (4)	Forward counter
CTD	3,3 (4)	3,5 (4)	Backward counter
CNT	4,1 (4)	4,3 (4)	Bi-directional counter
SFL	3,2 (4)	3,4 (4)	Shift register left
SFR	3,2 (4)	3,4 (4)	Shift register right
TON	3,9 (4)	-	Timer on delay
TOF	4,3 (4)	-	Timer off delay
RTO	4,4 (4)	-	Integrating timer, time measurement
IMP	4,2 (4)	-	Timer - specified pulse length generator
STE	1,2 (4)	1,4 (4)	Step sequencer (stepper)

Arithmetic instructions

Mnemo code	Execution time [μs] (Code length [B])						Instruction description	
	Z			#		A		
	byte usint	word uint	dword udint	byte usint	dword udint	byte usint		dword udint
ADD	1,0 (4)	1,1 (4)	1,5 (4)	-	0,9 (6)	-	0,9 (2)	Addition
SUB	1,0 (4)	1,1 (4)	1,5 (4)	-	0,9 (6)	-	0,9 (2)	Subtraction
MUL	1,2 (4)	1,3 (4)	1,7 (4)	-	1,0 (6)	-	1,0 (2)	Multiplication
MULS	1,2 (4)	1,3 (4)	1,7 (4)	-	1,0 (6)	-	1,0 (2)	Multiplication with sign
DIV	1,8 (4)	-	-	1,4 (4)	-	1,6 (2)	-	Division (byte / byte = byte)
DID	2,6 (4)	2,7 (4)	3,1 (4)	-	2,4 (6)	-	2,1 (2)	Division with remainder
DIVL	2,0 (4)	2,1 (4)	2,5 (4)	-	1,8 (6)	-	1,9 (2)	Division
DIVS	2,0 (4)	2,1 (4)	2,5 (4)	-	1,8 (6)	-	1,9 (2)	Division with sign
MOD	-	-	-	-	-	-	1,8 (2)	Division remainder
MODS	-	-	-	-	-	-	1,8 (2)	Division remainder with sign
INR	1,0 (4)	1,2 (4)	2,0 (4)	-	-	-	0,8 (2)	Incrementation (+ 1)
DCR	1,0 (4)	1,2 (4)	2,0 (4)	-	-	-	0,8 (2)	Decrementation (- 1)
EQ	1,6 (4)	1,7 (4)	2,1 (4)	-	1,6 (6)	-	1,5 (2)	Comparison (equality)
LT	1,6 (4)	1,7 (4)	2,1 (4)	-	1,6 (6)	-	1,5 (2)	Comparison (less than)
LTS	1,6 (4)	1,7 (4)	2,1 (4)	-	1,6 (6)	-	1,5 (2)	Comparison with sign (less than)
GT	1,6 (4)	1,7 (4)	2,1 (4)	-	1,6 (6)	-	1,5 (2)	Comparison (greater than)
GTS	1,6 (4)	1,7 (4)	2,1 (4)	-	1,6 (6)	-	1,5 (2)	Comparison with sign (greater than)
CMP	1,4 (4)	1,5 (4)	1,9 (4)	-	1,4 (6)	-	1,4 (2)	Comparison
CMPS	1,4 (4)	1,5 (4)	1,9 (4)	-	1,4 (6)	-	1,4 (2)	Comparison with sign
MAX	-	-	-	-	-	-	1,0 (2)	Maximum
MAXS	-	-	-	-	-	-	1,0 (2)	Maximum with sign
MIN	-	-	-	-	-	-	1,0 (2)	Minimum
MINS	-	-	-	-	-	-	1,0 (2)	Minimum with sign
ABSL	-	-	-	-	-	-	0,8 (2)	Absolute value
CSGL	-	-	-	-	-	-	0,8 (2)	Change sign
EXTB	-	-	-	-	-	-	0,9 (2)	Extend sign from 8 bits to 32 bits
EXTW	-	-	-	-	-	-	0,9 (2)	Extend sign from 16 bits to 32 bits
BIN	-	-	-	-	-	-	2,1 (2)	Conversion of number to bin. form (8 digits BCD)
BIL	-	-	-	-	-	-	3,1 (2)	Conversion of number to bin. form (10 digits BCD)
BCD	-	-	-	-	-	-	12,6 (2)	Conversion of number to BCD (8 digits BCD)
BCL	-	-	-	-	-	-	14,7 (2)	Conversion of number to BCD (10 digits BCD)

Stack operations

Mnemo code	Execution time [μs] (Code length [B])	Instruction description
POP n	0,8 (4)	Stack shift (rotation) backwards by n levels
NXT	2,8 (4)	Activation of next stack in the queue
PRV	2,8 (4)	Activation of previous stack in the queue
CHG	2,7 (4)	Activation of selected stack without backing S0 and S1
CHGS	2,6 (4)	Activation of selected stack with backing S0 and S1
LAC	1,6 (4)	Load values from top of chosen stack
WAC	1,4 (4)	Write value on top of chosen stack
PSHB	1,4 (2)	Saving of data of 8 bit width on the stack acc. to SP
PSHW	1,4 (2)	Saving of data of 16 bit width on the stack acc. to SP
PSHL	1,4 (2)	Saving of data of 32 bit width on the stack acc. to SP
PSHQ	1,4 (2)	Saving of data of 64 bit width on the stack acc. to SP
POPB	1,6 (2)	Popping of data of 8 bit width to the stack acc. to SP
POPW	1,6 (2)	Popping of data of 16 bit width to the stack acc. to SP
POPL	1,6 (2)	Popping of data of 32 bit width to the stack acc. to SP
POPQ	1,6 (2)	Popping of data of 64 bit width to the stack acc. to SP

Appendix

Jump and call instructions

Mnemonic code	Execution time [μs] (Code length [B])		Instruction description
	Transit	Jump	
JMP Ln	-	1,2 (4)	Unconditional jump
JMD Ln	0,5	1,3 (4)	Jump conditional on non-zero stack top value
JMC Ln	0,5	1,3 (4)	Jump conditional on zero stack top value
JMI Ln	-	1,3 (4)	Indirect jump
JMI	-	1,3 (2)	Indirect jump
JZ Ln	0,5	1,3 (4)	Jump conditional on non-zero value of flag ZR
JNZ Ln	0,5	1,3 (4)	Jump conditional on zero value of flag ZR
JC Ln	0,5	1,3 (4)	Jump conditional on non-zero value of flag CO
JNC Ln	0,5	1,3 (4)	Jump conditional on zero value of flag CO
JB Ln	0,5	1,3 (4)	Jump conditional on non-zero value of equality flag S0.2
JNB Ln	0,5	1,3 (4)	Jump conditional on zero value of equality flag S0.2
JS Ln	0,5	1,3 (4)	Jump conditional on non-zero value of flag S1.0
JNS Ln	0,5	1,3 (4)	Jump conditional on zero value of flag S1.0
CAL Ln	-	1,7 (4)	Unconditional subroutine call
CAD Ln	0,5	2,0 (4)	Call conditional on non-zero stack top value
CAC Ln	0,5	2,0 (4)	Call conditional on zero stack top value
CAI Ln	-	2,0 (4)	Indirect subroutine call
CAI	-	2,0 (2)	Indirect subroutine call
RET	-	1,8 (2)	Unconditional return from subroutine
RED	0,5	1,9 (2)	Return from subroutine conditional on non-zero result
REC	0,5	1,9 (2)	Return from subroutine conditional on zero result
L n	0,4 (4)	-	Label n (jump and call target)

Operating instructions

Mnemonic code	Execution time [μs] (Code length [B])					Instruction description
	Transit	Jump	P41 - P49	P50 - P57	P62 - P64	
P	0,4 (4)	1,4*	1,2	1,2	0,4	Process start (*jump to label of given SEQ)
E	-	0,6 (4)	4,3	4,3	0,6	Unconditional process end
ED	0,5 (2)	0,9	4,7	4,7	0,9	Process end conditional on non-zero result
EC	0,5 (2)	0,9	4,7	4,7	0,9	Process end conditional on zero result
NOP	0,4 (4)	-	-	-	-	No operation
BP	-	4,9 (4)	-	-	-	Breakpoint
SEQ	1,2 (4)	1,5	-	-	-	Conditional process interrupt

Table instructions

Mnemonic code	Execution time [μs] (Code length [B])					Instruction description
	Z					
	bool	byte usint sint	word uint int	dword udint dint	real	
LTB	3,8 (4)	2,4 (4)	2,7 (4)	3,0 (4)	3,0 (4)	Load item from table
WTB	2,7 (4)	2,2 (4)	2,5 (4)	2,8 (4)	2,8 (4)	Write item to table
FTB	2,8 (4)	2,0 (4)	2,3 (4)	2,6 (4)	2,6 (4)	Find item in table
	+0,2	+0,2	+0,3	+0,4	+0,4	- time addition for 1 item being searched
FTBN	2,8 (4)	2,0 (4)	2,3 (4)	2,6 (4)	2,6 (4)	Find next item in table
	+0,2	+0,2	+0,3	+0,4	+0,4	- time addition for 1 item being searched
FTM	-	2,1 (4)	2,3 (4)	2,6 (4)	2,6 (4)	Find part of item in table
		+0,4	+0,8	+1,2	+1,2	- time addition for 1 item being searched
FTMN	-	2,1 (4)	2,3 (4)	2,6 (4)	2,6 (4)	Find next part of item in table
		+0,4	+0,8	+1,2	+1,2	- time addition for 1 item being searched
FTS	-	2,0 (4)	2,1 (4)	2,2 (4)	-	Find item with sorting
		+0,2	+0,5	+0,8		- time addition for 1 item being searched
FTSF	-	-	-	-	2,2 (4)	Find item with sorting
					+1,0	- time addition for 1 item being searched
FTSS	-	2,0 (4)	2,1 (4)	2,2 (4)	-	Find item with sorting
		+0,2	+0,5	+0,8		- time addition for 1 item being searched

Mnemo code	Execution time [µs] (Code length [B])					Instruction description
	bool	byte usint sint	word uint int	dword udint dint	real	
LTB	3,8 (4)	2,4 (4)	2,7 (4)	3,0 (4)	3,0 (4)	Load item from table
WTB	3,7 (4)	3,2 (4)	3,6 (4)	4,0 (4)	4,0 (4)	Write item to table
FTB	2,6 (4)	2,3 (4)	2,7 (4)	3,1 (4)	3,1 (4)	Find item in table
	+0,2	+0,2	+0,3	+0,4	+0,4	- time addition for 1 item being searched
FTBN	2,6 (4)	2,3 (4)	2,7 (4)	3,1 (4)	3,1 (4)	Find next item in table
	+0,2	+0,2	+0,3	+0,4	+0,4	- time addition for 1 item being searched
FTM	-	2,4 (4)	2,8 (4)	3,2 (4)	3,2 (4)	Find part of item in table
		+0,4	+0,8	+1,2	+1,2	- time addition for 1 item being searched
FTMN	-	2,4 (4)	2,8 (4)	3,2 (4)	3,2 (4)	Find next part of item in table
		+0,4	+0,8	+1,2	+1,2	- time addition for 1 item being searched
FTS	-	2,3 (4)	2,5 (4)	2,7 (4)	-	Find item with sorting
		+0,2	+0,5	+0,8		- time addition for 1 item being searched
FTSF	-	-	-	-	2,7 (4)	Find item with sorting
					+1,0	- time addition for 1 item being searched
FTSS	-	2,3 (4)	2,5 (4)	2,7 (4)	-	Find item with sorting
		+0,2	+0,5	+0,8		- time addition for 1 item being searched

Block operations

Mnemo code	Execution time [µs] (Code length [B])			Instruction description
	Z	T	A	
SRC	1,3 (4)	1,5 (4)	-	Specification of data source for transfer
MOV	1,9 (4)	2,6 (4)	-	Data block move
	+0,2	+0,2		- time addition for 1 block item
MTN	-	-	1,9 (2)	Move table to scratchpad
			+0,2	- time addition for 1 table item
MNT	-	-	1,9 (2)	Fill table from scratchpad
			+0,4	- time addition for 1 table item
FIL	1,9 (4)	-	-	Fill block with constant
	+0,2			- time addition for 1 block item
BCMP	1,9 (2)	-	-	Comparison of blocks
	+0,4			- time addition for 1 block item

Operations with structured tables

Mnemo code	Execution time [µs] (Code length [B])	Instruction description
LDSR	3,1 (2)	Load item from structured table in scratchpad
	+0,2	- time addition for 1 byte of item
LDS	3,2 (2)	Load item from T structured table
	+0,2	- time addition for 1 byte of item
WRSR	3,4 (2)	Write item to structured table in scratchpad
	+0,3	- time addition for 1 byte of item
WRS	3,5 (2)	Write item to T structured table
	+0,3	- time addition for 1 byte of item
FIS	2,2 (2)	Fill item of structured table in scratchpad
	+0,2	- time addition for 1 byte of item
FIT	2,3 (2)	Fill item of T structured table
	+0,3	- time addition for 1 byte of item
FNS	2,2 (2)	Find item of structured table in scratchpad
	+0,3	- time addition for 1 item
FNT	2,3 (2)	Find item of T structured table
	+0,3	- time addition for 1 item

Appendix

Floating point arithmetic instructions (time data is orientational only, depending on input value)

Mnemo code	Execution time [μs] (Code length [B])				Instruction description
	Z	#	A		
	real	real	real	lreal	
ADF	23 (4)	24 (6)	24 (2)	-	Addition
ADDF	-	-	-	24 (2)	Addition
SUF	23 (4)	24 (6)	24 (2)	-	Subtraction
SUDF	-	-	-	24 (2)	Subtraction
MUF	20 (4)	21 (6)	21 (2)	-	Multiplication
MUDF	-	-	-	21 (2)	Multiplication
DIF	20 (4)	21 (6)	21 (2)	-	Division
DIDF	-	-	-	21 (2)	Division
EQF	18 (4)	19 (6)	19 (2)	-	Comparison (equality)
EQDF	-	-	-	19 (2)	Comparison (equality)
LTF	18 (4)	19 (6)	19 (2)	-	Comparison (less then)
LTDF	-	-	-	19 (2)	Comparison (less then)
GTF	18 (4)	19 (6)	19 (2)	-	Comparison (greater then)
GTDF	-	-	-	19 (2)	Comparison (greater then)
CMF	15 (4)	16 (6)	16 (2)	-	Comparison
CMDF	-	-	-	16 (2)	Comparison
MAXF	-	-	24 (2)	-	Maximum
MAXD	-	-	-	24 (2)	Maximum
MINF	-	-	24 (2)	-	Minimum
MIND	-	-	-	24 (2)	Minimum
CEI	-	-	21 (2)	-	Round up
CEID	-	-	-	21 (2)	Round up
FLO	-	-	19 (2)	-	Round down
FLOD	-	-	-	19 (2)	Round down
RND	-	-	23 (2)	-	Arithmetic round
RNDD	-	-	-	23 (2)	Arithmetic round
ABS	-	-	0,5 (2)	-	Absolute value
ABSD	-	-	-	0,5 (2)	Absolute value
CSG	-	-	0,5 (2)	-	Sign change
CSGD	-	-	-	0,5 (2)	Sign change
LOG	-	-	55 (2)	-	Decimal logarithm
LOGD	-	-	-	55 (2)	Decimal logarithm
LN	-	-	55 (2)	-	Natural logarithm
LND	-	-	-	55 (2)	Natural logarithm
EXP	-	-	1100 (2)	-	Exponential function
EXPD	-	-	-	1100 (2)	Exponential function
POW	-	-	2000 (2)	-	Power
POWD	-	-	-	2000 (2)	Power
SQR	-	-	260 (2)	-	Square root
SQRD	-	-	-	260 (2)	Square root
HYP	-	-	300 (2)	-	Euclidean distance
HYPD	-	-	-	300 (2)	Euclidean distance
SIN	-	-	800 (2)	-	Sine
SIND	-	-	-	800 (2)	Sine
ASN	-	-	1000 (2)	-	Arcsine
ASND	-	-	-	1000 (2)	Arcsine
COS	-	-	800 (2)	-	Cosine
COSD	-	-	-	800 (2)	Cosine
ACS	-	-	1000 (2)	-	Arccosine
ACSD	-	-	-	1000 (2)	Arccosine
TAN	-	-	1500 (2)	-	Tangent
TAND	-	-	-	1500 (2)	Tangent
ATN	-	-	700 (2)	-	Arctangent
ATND	-	-	-	700 (2)	Arctangent
UWF	-	-	10 ÷ 25 (2)	-	Conversion of uint to real
IWF	-	-	10 ÷ 25 (2)	-	Conversion of int to real
ULF	-	-	10 ÷ 25 (2)	-	Conversion of uint to real
ILF	-	-	10 ÷ 25 (2)	-	Conversion of dint to real

Mnemo code	Execution time [μs] (Code length [B])				Instruction description
	Z real	# real	A real lreal		
ULDF	-	-	-	10 ÷ 25 (2)	Conversion of uint to lreal
ILDF	-	-	-	10 ÷ 25 (2)	Conversion of dint to lreal
FDF	-	-	-	10 ÷ 25 (2)	Conversion of real to lreal
UFW	-	-	10 ÷ 30 (2)	-	Conversion of real to uint
IFW	-	-	10 ÷ 30 (2)	-	Conversion of real to int
UFL	-	-	10 ÷ 30 (2)	-	Conversion of real to uint
IFL	-	-	10 ÷ 30 (2)	-	Conversion of real to dint
UDFL	-	-	-	10 ÷ 30 (2)	Conversion of lreal to uint
IDFL	-	-	-	10 ÷ 30 (2)	Conversion of lreal to dint
DFF	-	-	-	10 ÷ 30 (2)	Conversion of lreal to real

PID controller instructions (time data is orientational, it depends on selected functions and input values)

Mnemo code	Execution time [μs] (Code length [B]) A	Instruction description
CNV	120 (2)	Data conversion and processing from analog units
PID	250 ÷ 500 (2)	PID controller

Operations with ASCII characters

Mnemo code	Execution time [μs] (Code length [B]) A				Instruction description
	word uint	dword uint	real	lreal	
TER	-	200 (2)	-	-	Terminal instruction
BAS	2,3 (2)	-	-	-	Conversion of number in binary format to ASCII
ASB	0,9 (2)	-	-	-	Conversion of number in binary format from ASCII
STF	-	-	170 (2)	-	Conversion of ASCII string to float format
STDF	-	-	-	170 (2)	Conversion of ASCII string to double format
FST	-	-	90 (2)	-	Conversion of float format to ASCII string
DFST	-	-	-	90 (2)	Conversion of double format to ASCII string



teco

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 001 09.02

The manufacturer reserves the right of changes to this documentation. The latest edition of this document is available at www.tecomat.cz