3

3. PLC 836 Basic Instruction Set

A language PLC836 is destined for effective programming of system CNC836 interface. It has following advantages against assembler programming: A coding time as well as coding error probability is significantly decreased, utilization of microprocessor time is optimized and it conducts to keep recommended program structure. A language uses exclusively symbolic addresses even for single memory bit addressing. A source program code has a similar structure as a language used for programmable controllers family NS900 manufactured by TESLA Kolin.

3.1 Writing a Source Code

The source code can be written in any ASCII text editor. This manual is not a textbook for interface programming. Hence only necessary information for source code writing are introduced. Programming is provided in mnemonic code. For code writing are valid similar rules as for writing an assembler source code

Source code format:

a) Label
It is optional and determines a symbolic memory location where a variable is stored, or to which a program jump is provided. The label can content max 31 alphanumerical characters including 3 special characters:

_ Underline character
? Question mark
@ "At sign"

A first character must not be numeric. The label must end with colon. The label can be placed on separate line and relates to closest next line which contents instruction code, variable, or memory allocation.

b) Instructions
It is a mnemonic symbol of appropriate Operational Code. As instructions only instructions mentioned next in this manual are allowed. Except these it is possible to use 8086-microprocessor instruction set but using it is not necessary for writing an interface program.

c) Operand
The instructions can be without operand (e.g. instruction BCD), or mostly with single operand (e.g. LDR ALFA). Some instructions can have double operands (e.g.. INP port,addr.) separated by commas.

d) Comment
A text introduced by semicolon is considered as commentary and is ignored. The comment can be on separate line or together with instruction code.

Example:

```
                                        ; This is a comment
PAM12:      LDR         ALFA            ; this is a comment
```

3.2 Working Registers of PLC836 Language

Instructions of PLC836 language use following registers:

a) The one bit register called **Register of Logical Operations**, (**RLO)**. Actually it is bit with weight 40Hex of microprocessor AH register.

b) The sixteen-bit register called Data **Register** , (**DR)**. Some instructions can work with extended 32 bit DR.
Actually it is microprocessor CX register. I extended mode (32-bit) it is register CX and BP.

c) Stack - 8 cells type WORD.

If only instructions of PLC 836 language are used the actual representation of the registers is not important for programmer. Working with actual registers is necessary only when some parts of the program are programmed in 8086 assembler.

Changing value in register RLO does not affect content of register DR and vice versa. The PLC836 Language automatically recognizes if Byte or Word approach to DR register or memory must be used.

3.3 Memory Declaration

The PLC836 language can work with any address in its symbolic form. The symbolic addressing can be used either for bit, eight-bit (Byte type) or sixteen-bit (Word type) RAM memory cells. The symbolic address is defined as a label with maximum 31 alphanumeric characters. (A length of the label is not restricted but only first 31 characters are significant). A first character must be a letter. The label must be different from predefined keywords. A list of keywords is in Appendix.
To assign the label to certain memory location can be provided in two ways. In case of address declaration in program memory location (EPROM) or declaration eight-bit or sixteen-bit word in RAM memory it is enough to end the label with a colon. Then the address relates to instruction next to colon.

Example:
In the program, which consists of instructions 1 to 5, assign the address of instructions 2 and 3 by labels ALFA and BETA.

```
            INSTRUCTION     1
ALFA:       INSTRUCTION     2
BETA:       INSTRUCTION     3
            INSTRUCTION     4
            INSTRUCTION     5
```

3.4 Bit definition, Byte Word and Constant

Instruction	**DFM**

operation **bit definition in memory**

syntax **DFM** **[bit0],[bit1],,,,,,[bit7]**

For symbolic definition of each bit in the RAM memory is destined a special instruction DFM. A one DFM instruction defines always in the place of its record one memory byte (eight bits). A whole byte can be named via prefix symbol, which must end with a colon. Such defined byte permits bite as well as byte access. The symbols, which are after statement DFM, declare individual bits of a defined byte. A first symbol belongs to bit d0, second to bit d1 and eventually the eight symbol belongs to bit d7. Commas must separate the symbols. In case that we do not want define some bits an appropriate symbol can be omitted. The commas can not be omitted in any case!

Each inputs and outputs of peripheral circuits are scanned and loaded via octuples. To facilitate PLC program we shall consider this inputs and outputs only this way. At the beginning of the program all RAM inputs are read via special instruction and at the end of the program dedicated RAM location shall be load to outputs. All outputs res. inputs can be declared and treated in a similar way as the RAM memory bits.

Instruction	**DS**	**1**
	DS	**2**
	DS	**n**

operation	**DS**	**1**	**memory declaration length 1 byte**
	DS	**2**	**memory declaration length 1 word**
	DS	**n**	**memory declaration length n byte**
syntax	**DS**	**1**	
	DS	**2**	

The instruction DS is destined for variable definition and memory initialization. By operand a length of dedicated memory type BYTE is declared. The instruction DS 1 with a label of variable declares this variable as BYTE and instruction DS2 declares this variable as WORD.

Instruction	**EQUI**

Operation **constant definition**

syntax **EQUI** **const, value**

Instruction EQUI defines constants used in a program. The first parameter is name of constant (symbolic); the second one is its value the value can be in following forms:

1. Decimal e.g. 123, 54213
2. Hexadecimal e.g. 0F8H, 15H
3. Character e.g. 'A', '8'

Example:
In the RAM memory define byte GAMA and its single bits d0 to d7 as symbols GAMA0 to GAMA7. On address DELTA define bit d= DELTAM, bit d2= DELTAG and bit d7=DELTAT.

```
GAMA:     DFM     G0, G1, G2, G3, G4, G5, G6, and G7
DELTA:    DFM     DELTAM,DELTAG,,,,,DELTAT
```

Example:
Define length 1 byte for variable ALFA
Define length 2 byte for variable BETA
Define field of length 30byte for variable GAMA
Define a constant DELTA with value 1000

```
ALFA:     DS      1
BETA:     DS      2
GAMA:     DS      30
          EQUI    DELTA, 1000
```

3.5 Logical operations with memory bits and RLO

Instruction	**LDR**

Operation **loading memory bit to RLO /LOAD RLO/**

syntax

syntax	**LDR**	**[-]bit**	
	LDR	**[-][adr.]bit**	**more complex addressing**
	LDR	**[-][$+xx.]bit**	

Instruction **LDR** loads memory bit to register RLO

An operand in instruction LDR is obligatory. This operand is a symbolic name of RAM memory bit, which is defined by instruction DFM. If symbolic name is without sign or with "+" sign, the appropriate bit is directly loaded. If a minus signs "-" prefixed the name, the negative value of appropriate bit is loaded.

The more complex bit addressing: (compiler version 5.039 and higher)
In the case of loading bit from other memory location then is defined for this bit a memory location separated by comma is prefixed to bit's name. Instead of memory address is possible to use a name of there defined bit, or is possible to use indexing (via BX) or structure items. It gives us a possibility to work with a large bit field. Also is possible to use an offset for loading bit from an address other then is defined for this bit. For example loading a bit from defined address + 12 can be written as $+ ($ + 12), where character $ is used instead of bit's name. The complex way of bit addressing is possible to use for instructions LDR, LA, LO, LX and for instructions WR, FL1 a FL, if VERINSTRU is set at least to V1.
This type of addressing is used when is necessary to load bit with appropriate weight from any memory location. In the PLC program the universal bit's names are declared (e.g. B0,B1,...,B7,) which are then used for an access to any memory location.

Examples of bit addressing:

```
DFM   B0,B1,B2,B3,B4,B5,B6,B7 ; formal bit definition

LDR   ALFA             ;loading  bit  ALFA  from  memory,  where  is
defined
```

```
      LDR   BUN5.ALFA         ;from a cell BUN5 bit is loaded from the same
position as original bit ALFA
                              LDR   BUN5.B4          ;loading   4.bit
(weight 10h) from  cell  BUN5
      LDR   -(BUN5+3).ALFA    ;loading bit negation  from cell BUN5+3
      LDR   BUN5[BX].ALFA     ;using indexing (see chapter18)
      LDR   -(BUN5[BX]-6).ALFA; other combination
      LDR   (BUN5[BX]+PRVNI+2).ALFA ;PRVNI is  scripture item
      LDR   $+3.ALFA          ;loading bit ALFA with memory offset 3
```

3.6 Stack and End-instructions

If two or more **LDR** instructions are written consecutively in the program without affecting RLO (memory writing or conditional jump), a content of RLO is pushed into Stack before executing next **LDR**. It is possible to work with those values via instructions **LA** , **LO** or **LX** without operand specifying. Using the Stack a bracket operations can be easily programmed.

Equability Vyrovnanost of Stack is checked at the end of each module (see next) and at the end of each Sequential Logical Unit. Sekvenční logický celek. The unequability of the Stack results in an error report in initial compilation phase of TECHNOL compiler. The equability checking during program debugging can be done via instruction **CHECK** (see description of auxiliary instructions).

Each logical expression that is programmed via instructions LDR, LA, LO and **LX** must end with so called **end-instruction**. By this instruction a stack pointer is cleared and thus in next following LDR instruction RLO is not stored in stack.

The end-instructions of PLC836 language are:

- WR
- JL0, JL1
- STO1
- FL1
- CONRD
- EX, EX0, EX1, BEX
- TEX0, TEX1
- TIM

Instructions	**LA** **LO** **LX**

Operation	**LA**	**logical AND with RLO /AND/**	
	LO	**logical OR with RLO /OR/**	
	LX	**exclusive OR with RLO /XOR/**	
Syntax	**LA**		
	LO		
	LX		
Syntax2	**LA**	**[-bit]**	
	LO	**[-bit]**	
	LX	**[-bit]**	
	LA,LO,LX	**[-][adr.]bit**	**more complex addressing**
	LA,LO,LX	**[-][$+xx.]bit**	

The operand in **LA**, **LO** a **LX** instruction is optional. If used this operand is a symbolic name of RAM memory bit defined by instruction DFM. If symbolic name is without sign or with sign “+” the direct value of appropriate bit is loaded, when a sign” - “precedes symbolic name a negation of appropriate bit is loaded.

If two or more **LDR** instructions are written consecutively in the program without affecting RLO (memory writing or conditional jump), a content of RLO is stored in Stack before executing next **LDR**. It is possible to work with those values via instructions **LA** , **LO** or **LX** without operand specifying. Instruction LA provides logical AND with last value stored in stack with RLO and stores result in RLO. In this same way instruction LO provides logical OR and instruction LX logical XOR. Equability Vyrovnanost of Stack is checked at the end of each module (see next) and at the end of each Sequential Logical Unit. Sekvenční logický celek. The unequability of the Stack results in an error report in initial compilation phase of TECHNOL compiler.
The more complex addressing is described in instruction LDR.

Instruction CA

Operation **negation of RLO**

syntax **CA**

Instruction **CA** is without operand and provides a negation of RLO contents. If content of RLO is logical 1 by instruction **CA** is changed to 0 and vice versa.

Example 3:
Symbolically define memory bits A1, A2. Depending on value of these bits load RLO according to expression:

RLO = A1 + A2

```
Solution a):    LDR     A1
                LDR     A2
                LO
```

If instructions LA, LO or LX have an operand (symbolically assigned memory bit), the meaning of instruction is following:

Instruction LA with operand provides logical AND of appropriate memory bit with RLO and the result is stored in RLO. Similarly instruction LO provides logical OR and instruction LX logical XOR. Back to example 3 we have other solution:

```
Solution b):    LDR     A1
                LO      A2
```

Solution b) is equal to solution a) but program length is shortened.

With above mentioned instructions (LDR, LA, LO, LX, CA) is possible to program any logical expression. A programming approach is showed in examples 4, 5, and 6.

Example:
Program logical expression (notice. <> is XOR function):

RLO = (ALFA <> BETA) + (GAMA <> DELTA)

```
        LDR     ALFA
```

```
LX      BETA
LDR     GAMA        ;store RLO into stack and load bit GAMA
LX      DELTA
LO                  ;Logical OR RLO with stack
```

Example:
Program logical expression:

```
         ----    ----   ----          ----
RLO = [(A1.A2.A3) + (B1.B2)] . (C2 + C3)
```

```
LDR     -A1
LA      A2
LA      -A3
LDR     -B1     ;Store RLO to Stack and load negation of bit B1
LA      B2
LO              ;Logical OR RLO with Stack
LDR     C2      ;Store temporary result to Stack and load bit C2
LO      -C3
LA              ;Logical AND RLO with Stack
```

Example:
Program logical expression :

$RLO = \overline{A1.A2.A3} + \overline{B1.B2.B3}$

```
LDR     A1
LA      A2
LA      A3
CA
LDR     B1
LA      B2
LA      B3
CA
LO
```

3.7 Writing Bits to Memory

Instruction	**WR**

Operation **writing contents of RLO to memory**

syntax1	**WR**	**bit**	
Syntax2	**WR**	**<bit1 AND bit2 [AND bit3 ...]>**	
Syntax3	**WR**	**bit1 [,bit2 ...]**	**for VERINSTRU WR_V1**
	WR	**[adr.]bit**	**more complex addressing**
	WR	**[adr.]bit [,[adr2.]bit2 ...]**	
	WR	**[$+xx.]bit[,[$+yy.]bit2 ...]**	

Instruction WR writes content of RLO to dedicated bits of memory byte. The content of RLO a DR remains unaffected. Instruction WR is **End- instruction** for logical expression.

An original syntax of **WR** instruction can have one or more (max. 8) operands (left for compatibility). These operands are symbolic names of RAM memory bits that are defined per instruction DFM. If two or more operands are used they must assign bits of the same memory byte.

The bit's names are separated by word AND and its list is in closed in sharp brackets " < >". In the case of one symbolic name the brackets can be omitted.

The instruction **WR** exists also in a new modified version. The modification is provided per instruction **VERINSTRU**. (See description of VERINSTRU instruction in 3.17 Auxiliary statements). The modification permits using in instruction parameters more bit operands which on the contrary of original instruction may not be located in the same byte. The operands are not closed in brackets but are separated by commas.

The more complex addressing is described in instruction LDR.

Instruction FL

Operation **setting memory bits**

syntax1	**FL**	**0,bit**	
	FL	**1,bit**	
Syntax2	**FL**	**0,<bit1 AND bit2 [AND bit3 ...]>**	
	FL	**1,<bit1 AND bit2 [AND bit3 ...]>**	
Syntax3	**FL**	**0,bit1 [,bit2 ...]**	**pro VERINSTRU FL_V1**
	FL	**1,bit1 [,bit2 ...]**	**pro VERINSTRU FL_V1**
	FL	**0,[adr.]bit**	**more complex addressing**
	FL	**1,[adr.]bit [,[adr2.]bit2 ...]**	
	FL	**0,[$+xx.]bit[,[$+yy.]bit2 ...]**	

Instruction FL fills bits of dedicated byte with zeroes or ones independently on RLO content. The first operand of FL instruction is value 0 or 1. The second operand contents a list of bits to be filled. The operands are obligatory as an option is possible to set via third and fourth operand an opposite constant and bits to be loaded. A content of RLO and DR remains unaffected.

The instruction **WR** exists also in a new modified version. The modification is provided per instruction **VERINSTRU**. (See description of VERINSTRU instruction in 3.17 Auxiliary statements). The modification permits using in instruction parameters more bit operands which on the contrary of original instruction may not be located in the same byte. The operands are not closed in brackets but are separated by commas.

The more complex addressing is described in instruction LDR.

instruction FL1

Operation **conditional setting of memory bits**

Syntax1	**FL1**	**0,bit**	
	FL1	**1,bit**	
Syntax2	**FL1**	**0, <bit1 AND bit2 [AND bit3 ...]>**	
	FL1	**1,<bit1 AND bit2 [AND bit3 ...]>**	
Syntax3	**FL1**	**0,bit1 [,bit2 ...]**	**for VERINSTRU FL1_V1**
	FL1	**1,bit1 [,bit2 ...]**	**for VERINSTRU FL1_V1**
	FL1	**0,[adr.]bit**	**more complex addressing**
	FL1	**1,[adr.]bit [,[adr2.]bit2 ...]**	
	FL1	**0,[$+xx.]bit[,[$+yy.]bit2 ...]**	

Instruction **FL1** fills bits of dedicated byte by zeroes or ones only when value of RLO is **1.** The first operand of FL1 instruction is value 0 or 1. The second operand contents a list of bits to be filled. The operands are

obligatory A content of RLO and DR remains unaffected. Instruction FL1 is the end-instruction for logical expressions.

The instruction **FL1** exists also in a new modified version. The modification is provided per instruction **VERINSTRU**. (See description of VERINSTRU instruction in 3.17 Auxiliary statements). Starting from TECHNOL 2.3 compiler version it is possible to use similar instruction **FL0** The modification permits using in instruction parameters more bit operands which on the contrary of original instruction may not be located in the same byte. The operands are not closed in brackets but are separated by commas.

The more complex addressing is described in instruction LDR.

.

Two instructions and one label can compensate instruction FL1.

```
        JL0     OBSKOK
        FL      1,BIT           is equivalent of:  FL1     1,BIT
OBSKOK:
```

Example:
Load symbolic named memory bits PETR, IVAN, and JANA dependently to memory locations PAVEL and EVA according to following rule:

PETR = IVAN = PAVEL + EVA
JANA = PAVEL . EVA

```
JMENA1:     DFM         PAVEL,EVA, , , , , ,
JMENA2:     DFM         PETR,IVANA,JANA, , , , , ,
            ...
            LDR         PAVEL
            LO          EVA
            WR          <PETR AND IVAN>
            LDR         PAVEL
            LA          EVA
            WR          JANA
```

Example:
Define following memory bits as follows:

A1, A2
B1, B2, B3, B4 in byte B

Fill the memory bits as follows:

0 into A1, B3, B4
1 into B1, B2, A2

```
        FL          0,A1
        FL          1,A2
        FL          1,<B1 AND B2>,0,<B3 AND B4>
```

3.8 The Program Branching

instructions	**JUM** **JL0** **JL1**

operation	**JUM**	**unconditional jump**
	JL0	**jump if RLO = 0**
	JL1	**jump if RLO = 1**
syntax	**JUM**	**adr**
	JL0	**adr**
	JL1	**adr**

All branching instructions, in other words a program jump, must have an address of next executed instruction (if the jump conditions are performed) as an operand. If condition is not fulfilled the processor executes next instruction (jump is not performed). This address is set by symbolic form, when dedicated symbol is defined via colon in any program location.

Instruction **JUM** is an unconditional jump so the jump is always performed. Instruction **JL0** is performed only when RLO = 0. Instruction **JL1** is performed only when RLO = 1. Instructions JL0 a JL1 are end-instructions for logical expression.

Example:
Write via PLC 836 instructions following logical algorithm:

If A1 = A2, load 0 to B1 and B2 defined in the same memory byte.
If A1 . A2 = 1, load 1 to B3.

```
            LDR     A1
            LX      A2
            JL1     NAV1
            FL      0,<B1 AND B2>

 NAV1:      LDR     A1
            LA      A2
            JL0     NAV2
            FL      1,B3
NAV2:       ...
            ...
            ...
```

3.9 Ways of Type Redefining for Data Variables

The PLC836 language works with operands in the data operations automatically according to their definition. For example instruction LOD loads variable type BYTE, WORD or constant according to operand definition :

DEFINITION:				**OPERATION:**		
BUNKA1:	DS	1	....	LOD	BUNKA1	load BYTE to DR register upper part of DR register is zero
BUNKA2:	DS	2	....	LOD	BUNKA2	load WORD to DR register
EQUI	KONST, 124		.... LOD		KONST	load constant 124 to DR register

Some instructions that work with DR register have possibility to redefine the type of data variable. The following description deals with instructions LOD, STO, STO1, AD, SU, EQ, EQ1, LT, GT, LE, GE, RR, RL, TM, TIM, TEX0, TEX1, MULB a DIVB.

The instructions can have optional prefix before variable name (TYPE.), which can redefine or amend of variable type to :

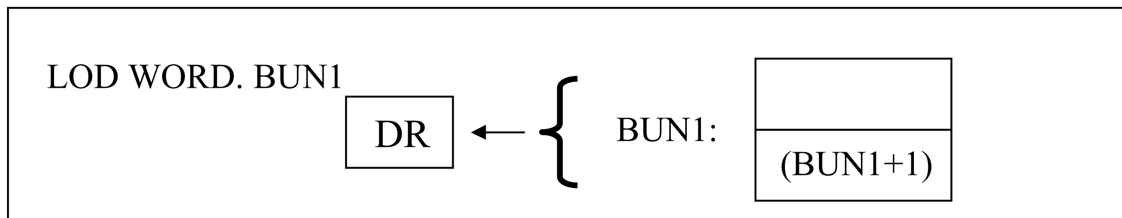
- **CNST**
- **BYTE**
- **WORD**
- **HIGH**
- **DWRD**

Prefix is connected to operand via dot.

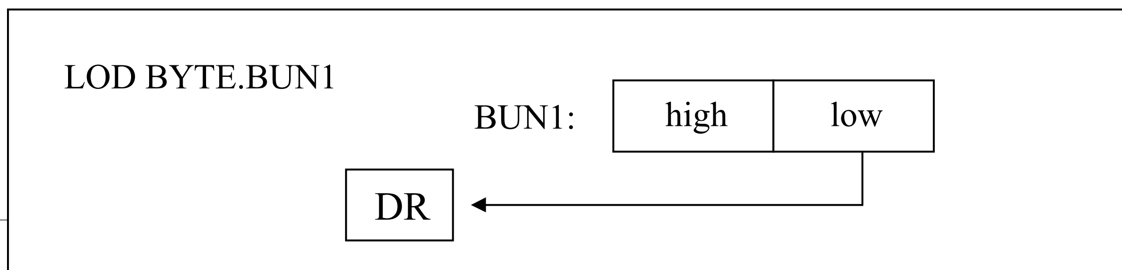
Instructions LOD, AD, SU, EQ, EQ1, LT, GT, LE, GE, RR, RL, MULB, DIVB can have a constant as an operand which is not defined via instruction **EQUI**. In this case is placed before constant a value type: **"CNST."**. It is supposed that constant has its length maximally 16 bits. The only exception is using prefix CNST together with LOD instruction when from practical reasons the upper 16 bits of extended 32 bit DR register are cleared.

Next are examples of using prefixes with instructions LOD. With other instruction prefixes work similarly but acts according instruction operation.

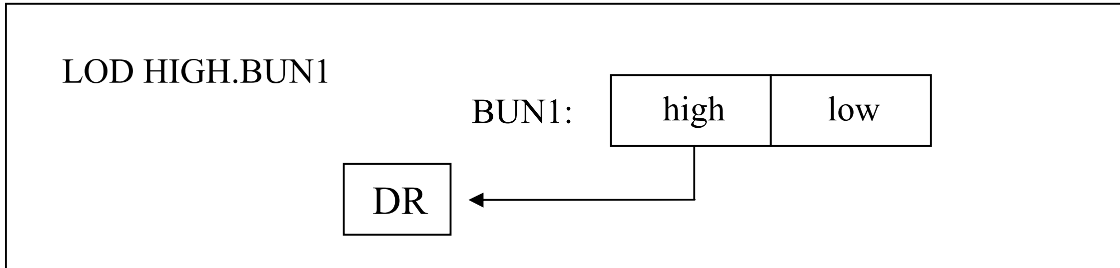
Prefix **"WORD."** loads 2 sequential bytes to DR register regardless their definition:



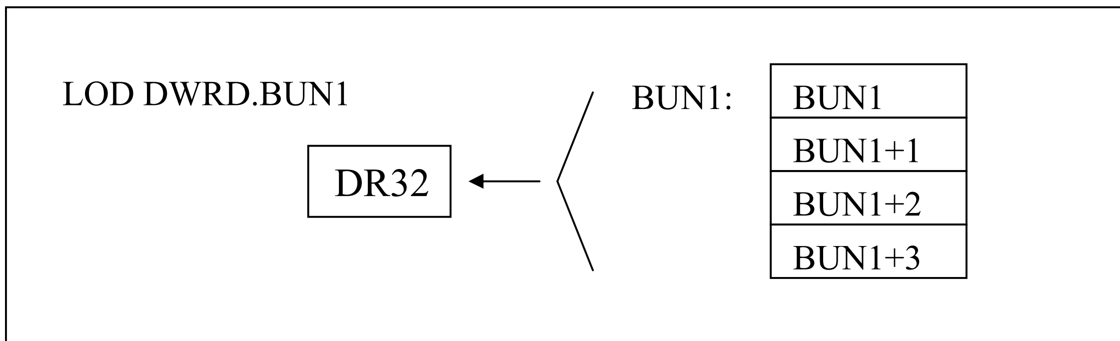
Prefix **"BYTE."** loads one byte to DR register regardless its definition. In the case that a cell is defined as type WORD, the lower byte is loaded to DR register:



Prefix **"HIGH."** loads 1 byte to DR register regardless how is defined from the next memory location: In case that a cell is defined as type WORD a high byte is loaded to DR register:



Prefix **"DWRD."** loads 4 sequential bytes to extended 32 bits DR register regardless of their definition. With instructions that can work with double word (four byte) operands is prefix DWRD obligatory.



Example:
Write to X-axis measurement cell:

```
BUNKA:          DS      4
```

```
CLI                          ;interrupt disabled
LOD    DWRD.B_POL            ;load to extended DR register B_POL
AD     DWRD.B_INK            ;add two words B_INK
STI                          ;interrupt enabled
STO    DWRD.BUNKA            ;write to BUNKA 32 bits
```

Example:
Using prefixes:

```
LOD    CNST.123              ;loads constant
AD     CNST.50h              ;add 50 hexadecimal
AD     BYTE.ALFA             ;add lower byte ALFA
STO    WORD.BETA             ;write to BETA as WORD
```

Possibility of declaration and type redefining of data variables:

	DECLARATION			***possible type redefining***				
	BYTE	**WORD**	**CNST**	**BYTE.**	**WORD.**	**HIGH.**	**CNST.**	**DWRD.**
LOD	✓	✓	✓	✓	✓	✓	✓	✓
STO, STO1	✓	✓	.	✓	✓	✓	.	✓
TM	✓	✓	.	✓	✓	✓	.	.
CU, CD	✓	✓	.	.	.	.	.	.
CUBCD	✓	✓	✓	.	.	.	.	.
AD, SU	✓	✓	✓	✓	✓	✓	✓	✓
MULB	✓	.	✓	✓	✓	✓	✓	✓
DIVB	✓	.	✓	✓	✓	✓	✓	✓
RR, RL	✓	.	✓	✓	.	✓	✓	.
EQ, EQ1	✓	✓	✓	✓	✓	✓	✓	✓
LT, GT	✓	✓	✓	✓	✓	✓	✓	✓
LE, GE	✓	✓	✓	✓	✓	✓	✓	✓
TIM	✓	✓	.	✓	✓	✓	.	.
TEX0, TEX1	✓	✓	.	✓	✓	✓	.	.

3.10 Memory Writing and Reading from Data Register DR

instruction	**LOD**

operation **LOD** **loading byte, word, constant (or DWORD) to DR**

syntax

syntax	**LOD**	**[-]adr**
	LOD	**[-][TYPE.]adr**
	LOD	**[-]TYPE.(adr+n)**
	LOD	**[-]CNTS.konst**
		TYPE = BYTE, WORD, HIGH, DWRD

Instruction **LOD** loads memory content of length byte or word (DWORD) or constant to data registers DR. The memory address is set via instruction’s operand in its symbolic form. A constant must be defined via instruction EQUI, or via prefix "**CNST.**". Using instruction with "CNST." prefix results loading also into extended part of DR register Assigning of “-“sign loading a complementary value to register DR.

The type redefining is described in chapter 3.9.

instruction STO

operation STO write DR in memory byte, word (or DWORD)

syntax STO adr
STO [TYPE.]adr
STO TYPE.(adr+n)

Instruction **STO** permits writing of DR content to dedicated memory address with byte word (or DWORD using prefix) length. The memory address is again set via instruction's operand in its symbolic form.

The type redefining is described in chapter 3.9.

instruction STO1

operation STO1 conditional writing DR to memory byte, word (or DWORD)

syntax Stoh adr
STO1 [TYPE.]adr
STO1 TYPE.(adr+n)
TYPE = BYTE. WORD. HIGH. DWRD.

Instruction **STO1** is analogous to instruction **STO**, but writing DR to memory is provided only when content of RLO is logical 1 Instruction STO1 is introduced to maintain compatibility with automaton NS915. The important difference is that STO1 is an end-instruction for logical expressions.
Starting from compiler's TECHNOL 2.3 version it possible to use similar instruction **STO0**, when writing DR to memory is provided only when content of RLO is logical 0

Instruction **STO1** can be substituted via two instructions and one label:

```
                JL0     OBSKOK
STO     BUNKA equal:    STO1    BUNKA
OBSKOK:
```

Example:
Have symbolically defined RAM memory bytes:

ALFA, BETA, GAMA

A content of ALFA writes to BETA, constant 123H write to GAMA.

```
        EQUI        K123, 123H

        LOD         ALFA
        STO         BETA
        LOD         K123
        STO         GAMA
```

The way of writing is the same also for length type Word.

Example:
Have following symbolically defined RAM memory bytes: ALFA type WORD and BETA type BYTE. Negative contents of byte BETA write to lower byte of variable ALFA.

```
LOD     -BETA
STO     BYTE.ALFA
```

Example:
Rewrite value of cell NASTAVENI to cell BUNKA, when expression ALFA*BETA is equal to 1:

```
LDR     ALFA
LA      BETA
LOD     NASTAVENI
STO1    BUNKA
```

3.11 Realization of Time Dependent Functions

instruction	**DFTM01** **DFTM1** **DFTM10** **DFTM100**

operation	**DFTM01**	**program segment activated after 0,1 s**
	DFTM1	**program segment activated after1 s**
	DFTM10	**program segment activated after 10 s**
	DFTM100	**program segment activated after 100 s**
syntax	**DFTM01**	**konec**
	DFTM1	**konec**
	DFTM10	**konec**
	DFTM100	**konec**

To get a most optimal processor's run time and to define time for TM and TIM timers are the time functions realized in special program segments. This program segment starts with instructions DFTM01, DFTM1, DFTM10 or DFTM100 and ends with label "Konica".

Unlike the other PLC program parts which are activated every 20mS the special program segments are activated in longer time intervals. Program segment defined by instruction DFTM01, is activated ("triggered by processor") in intervals 0,1 sec. Program segment defined by instruction DFTM1, is activated in intervals 1 sec. Program segment defined by instruction DFTM10, is activated every 10 sec and program segment defined by instruction DFTM100 is activated every 100 sec.

In a standard system version the instructions DFTM01, DFTM1, DFTM10 and DFTM100can be used ones only. System CNC8x9 – DUAL allow multiple using of DFTM instruction in all program segments.

When a system is powered on all counters (which are used for creating 0.1,1,10 and 100s intervals) are cleared.

instruction	**TM**

operation		**timer dependent on DR, RLO a DFTM block**
syntax	**TM**	**citac**
	TM	**[TYPE.]citac**
	TM	**TYPE.(citac+n)**
	TM	

TYPE = BYTE. WORD.

Instruction **TM** works with register DR, and with register RLO. As an operand is necessary to set a symbolic address of memory byte, which is destined for time counting (type BYTE or WORD).
For systems CNC8x9 –DUAL is this parameter optional. In that case a compiler defines itself automatically a variable for dedicated counter. The time counting in instruction TM depends on the current program block DFTM. If for example the instruction TM is in a program block which is defined via instruction DFTM10 (is activated every 10 sec.) the setting time is in tens of seconds.

Instruction TM works as follows:

1) If RLO = 0, the dedicated counter is cleared. This ends instruction execution.

2) If RLO = 1, the content of dedicated counter is compared with content of data register DR.
 a) If /ČÍTAČ/ >= DR, RLO is set to 1 and instruction ends.
 b) If /ČÍTAČ/ < DR, RLO is set to 0 and counter is incremented by 1 (In block DFTM01 it means time 0,1 sec. in block DFTM1 time 1 sec.).

Except instruction TM itself the counter block must content instructions for setting initial conditions for registers DR and of course instructions for setting one or more memory cells according to result of instruction execution (RLO).

Example:
Realize following time dependent function:
If a value of bit ALFA is logical 1 for time longer than 0,4 sec. set to logical 1 bit GAMA. As a counter use memory byte CITACA.

```
        EQUI        DOBA,4

        DFTM01      NAV30
        ...
        LDR         ALFA
        LOD         DOBA
        TM          CITACA
        FL1         1,GAMA
        ...
NAV30:
```

Example:
If logical AND of A1 a A2 is equal to zero for time longer than 5 sec. clear bit GAMA and clear memory cell BYTE. As a counter use memory byte CITACB.

```
        DFTM1       NAV50
        ...
        LDR         A1
        LA          A2
        CA
        LOD         CNST.5
        TM          CITACB
        FL1         0,GAMA
        LOD         CNTS.0
```

```
        STO1        BYTE
        ...
NAV50:
```

Example:
If signal TLAK is for reconfigured time (REKONFIG+20) equal logical 1 set bit HAVAR and a memory cell POCET to 67h:

```
        LDR         TLAK
        LOD         WORD.(REKONFIG+20)
        TM          WORD.(CITAC+10)
        FL1         1,HAVAR
        LOD         CNTS.67H
        STO1        POCET
```

instruction	**CU** **CD** **CUBCD**

operation	**CU**	**Up Counter dependent on DR, RLO and block DFTM**
	CD	**Down Counter dependent on DR, RLO and block DFTM**
	CUBCD	**BCD Up Counter dependent on DR, RLO and block DFTM**
syntax	**CU**	**citac**
	CD	**citac**
	CUBCD	**citac**

Counter instructions are destined for realization of counting functions. If counting conditions are fulfilled (clear input of a counter is on) the instruction Counter Up CU increments a memory cell defined in address part of instruction CU. The counter is incremented when an input bit goes from logical 0 to logical 1 A counter stage is concurrently compared with a value of data register DR (counter preset) . When a value of the counter and a register DR is equal RLO register is set to log. 1, in other case to log.0. After finishing of instruction CU a content of addressed counter is loaded to data register DR.

Instruction CU works with binary coded data in a range of byte or word. When reaching of preset value a content and declaration of counters remain unaffected.

Instruction counter down **CD** works similarly but instead increment a decrement of a counter is provided. Instruction **CUBCD works** as CU but in BCD code.

Example:
Count zero to one changes of input signal ALFA when disabling input BETA is in log. 1.

```
        LDR         ALFA        ;VSTUP
```

```
        LA      BETA    ;BLOKOVÁNÍ
        LOD     GAMA    ;NAČTENÍ PŘEDVOLBY
        CU      DELTA   ;ČÍTAČ (BYTE, WORD)
        JL0     NAV1
        ...
NAV1:
```

Example:
Clear counter when reaches a preset value GAMA

```
        LDR     ALFA    ;VSTUP
        LOD     GAMA    ;NAČTENÍ PŘEDVOLBY
        CU      DELTA   ;ČÍTAČ (BYTE, WORD)
        LOD     CNTS.0
        STO1    DELTA   ;VYNULOVÁNÍ
```

3.12 Arithmetical instructions with operand and DR register

Instruction performs logical or arithmetical operation mostly between DR register and a memory location or a constant. The address is an instruction parameter except instructions INR, DCR, BIN, BCD, ABS, INV, RR and RL that are without an operand.

instruction	**AD** **SU**

operation	**AD**	**Add byte, word or constant to DR**
	SU	**Subtract byte, word or constant to DR**
syntax	**AD (SU)**	**adr**
	AD (SU)	**[TYPE.]adr**
	AD (SU)	**TYPE.(adr+n)**
	AD (SU)	**CNTS.konst**
		TYPE = BYTE. WORD. HIGH. DWRD.

The instruction AD adds content of DR register to content of memory location defined by operand or to constant. A result of this operation is stored in DR register. The instruction can work with operand type BYTE, WORD or DWORD .

The instruction SU subtracts from content of DR register content of memory location defined by an operand or a constant. The result of this operation is stored in DR register. The instruction can work with operand type BYTE or WORD.

A Type redefining is described in Chapter.

instruction	**MULB** **DIVB**

operation	**MULB**	**multiplying DR register with an operand**
	DIVB	**division of DR register with an operand**

The instruction **MULB** multiplies register DR with a content of memory location specified by its operand. The result is stored in DR register. Instruction cans 8-bit (byte) operands and a result is type WORD. When the operand is redefined via prefix **"WORD."** instruction works with 16-bit (word) operands and the result is 32-bits (word) type in 32- bit extended DR register.

The instruction **DIVB** divides register DR (generally type word) by memory content specified by 8-bit (byte) operand. The result (integer number) is stored in DR register. The result can have 8 bits as maximum.

When an operand type is redefined by prefix **"WORD."** instruction divides content of extended 32-bit DR register by 16-bit (word) operand and the result is stored in DR register with length maximum 16 bit In this case is necessary before using this instruction load 32 bit DR register.

If CPU04 processor with statement P386 (see next) is used multiplying and dividing four bytes (double word) operands is allowed. For multiplying a redefinition of operand type to **"DWRD"**. In this case the instruction works with 32-bit (double-word) operands and the result is 64 bits (8 byte = QWORD). Lower 32 bits is stored in extended 32-bit DR register. It is supposed than multiplying type DWRD shall be immediately followed by division with redefining type **"DWRD".** The instruction divides content of 64 bit after multiplying by 32 bit (double-word) operand and the result remains in extended 32 bit DR register.

Example:
Value of cell ALFA (BYTE) diminished by 23h multiply by value of cell BETA (BYTE) result should be stored in cell GAMA (WORD).

```
LOD      ALFA              ;reads ALFA to DR
SU       CNST.23H          ;subtract 23h
MULB     BETA              ;multiply with BETA (result 16 bits)
STO      GAMA              ;write to GAMA
```

Example:
Add cells ALFA (WORD) and BETA (BYTE) result write to cell VYSLEDEK1, subtract constant 123h and multiply by cell GAMA (WORD). Result 32 bits write to cell VYSLEDEK2 (DWORD).

```
LOD      ALFA              ;read ALFA to DR
AD       BETA              ;Add DR  with BETA
STO      VYSLEDEK1         ;write to VYSLEDEK1
SU       CNST.123H         ;subtract 123h
MULB     WORD.GAMA         ; multiply with  GAMA (result 32 bits)
STO      DWRD.VYSLEDEK2    ;write result 32 bits to VYSLEDEK2
```

Example:
A division result of 32 bit cell DELENEC (DWORD) and 16 bit cell DELITEL (WORD) write to cell PODIL (WORD).

```
LOD      DWRD.DELENEC      ;red dividend 32 bits to DR
DIVB     WORD.DELITEL      ;dividing of 32 bits DR by factor 16 bits
STO      PODIL             ;store result to PODIL 16 bits
```

3.13 Non- operand Instructions for Working with DR Register

Some non-operand instructions for operations with data register DR can work with an extended 32-bit DR register (DWORD). These instructions are **INR, DCR, INV, ABS, RR**, **RL, CONDR, and CONRD.** In this case is necessary modify non-operand instruction by parameter **DWRD**.

instruction	**INR** **DCR** **INRBCD**

operation	**INR**	**increment DR register**
	DCR	**decrement DR register**
	INRBCD	**increment DR register in BCD code**
syntax	**INR (DCR)**	
	INR (DCR)	**[DWRD]**
	INRBCD	

Instruction **INR** without an operand increments DR register by 1. If overflow occurs increment starts with zero.
Instruction **DCR** without an operand decrements register DR by 1.
Instruction **INRBCD** increments register DR by 1 in BCD code. A working range is 0.. 9999.
Instructions INR a DCR with parameter **DWRD** increment or decrement an extended 32 bit DR register.

instruction	**BIN** **BCD**

operation	**BIN**	**conversion BCD → BIN code**
	BCD	**conversion BIN → BCD code**
syntax	**BIN**	
	BCD	
	BIN	**[DWRD]**
	BCD	**[DWRD]**

Instruction **BIN** without an operand converts a number in BCD code stored in a register DR (maximally 16 bits) to binary number. The result is stored in DR register.
Instruction **BIN** with parameter **DWRD** converts number in extended 32-bit DR register. A negative BCD number has before the conversion set to 1 bit with log weight 31.

Instruction **BCD** without an operand converts number in binary form stored in DR register to BCD coded number. The result is stored in DR register. A number in DR register before the conversion must not be greater then 9999d = 270Fh.
Instruction **BCD** with parameter **DWRD** converts number in an extended 32-bit DR register. The number in DR register must not be greater then 99999999d = 5F5E0FFh.

instruction	**RL** **RR**

operation	**RL**	**logical left shift of DR register**
	RR	**logical right shift of DR register**

syntax	**RL (RR)**	**n**
	RL (RR)	**[TYPE.]n**

Instruction **RL n** provides a logical left shift of DR register content by "n" bits. An operand is a constant or a value in a cell (8 bits as maximum) which assign a number of rotations.Tak je to rotace nebo posun ?
Instruction **RR n** provides a logical right shift of DR register content by "n" bits. An operand is a constant or a value in a cell (8 bits as maximum) which assign a number of rotations.
Instructions can have a second parameter **DWRD**, which means that a shift of an extended 32-bit DR register occurs

instruction	**INV** **ABS**

operation	**INV**	**negation of DR register (2nd complement)**
	ABS	**absolute value of DR register**
syntax	**INV (ABS)**	
	INV (ABS)	**[DWRD]**

Instruction **INV** performs a negation of DR register. In modulo 2 implementation it is 2nd complement.
Instruction **ABS** returns an absolute value of DR register.
Instruction can have an optional parameter **DWRD**, which modifies the instructions in such a way that negation or absolute value is provided with an extended 32 bit DR register.

Example:
Various operations:

```
LOD     ALFA                ;reading  ALFA (type according to declaration)
INR                         ;increment DR (16 bits)
ABS                         ;absolute value of DR (16 bits)
RL      POSUN               ;left shift of DR  by value stored in POSUN
STO     VYSLEDEK1           ;write to VYSLEDEK1
INV                         ;2nd complement of DR (16 bits)
RR      CNST.3              ;right shift of DR  by 3 bits
STO     VYSLEDEK2           ;write to VYSLEDEK2

LOD     DWRD.BETA           ;reading 32 bits from BETA to DR
ABS     DWRD                ;absolute value of DR 32 bits
INR     DWRD                ;decrement DR 32 bits
RR      CNST.18,DWRD        ; right shift by 18 bits  of DR 32 bits
STO     DWRD.VYSLEDEK3      ;write to VYSLEDEK3 32 bits
```

3.14 Logical instruction with operands and DR register

This group provides comparison of DR register with memory content or constant with memory content or constant and as a result sets RLO register.

instructions	**EQ**
	EQ1
	LT
	GT
	LE
	GE

operation	**EQ**	**comparison of DR register with operand**
	EQ1	**conditional comparison of DR with operand, when RLO = 1**
	LT	**DR is smaller then operand**
	GT	**DR is greater then operand**
	LE	**DR is smaller or equal then operand**
	GE	**DR is greater or equal then operand**

syntax	**EQ (EQ1,LT,GT,LE,GE)**	**adr**
	EQ (EQ1,LT,GT,LE,GE)	**[TYPE.]adr**
	EQ (EQ1,LT,GT,LE,GE)	**TYPE.(adr+n)**
	EQ (EQ1,LT,GT,LE,GE)	**CNTS.konst**
		TYPE = BYTE. HIGH. WORD. DWRD.

Instruction **EQ** is logical instruction and provides a comparison of DR register with a memory content assigned via operand or with constant. If DR is equal to memory content or with a constant the RLO register is set to 1 otherwise is set to zero. The operands can be of type Byte, Word, constant or DWORD.

Instructions **LT** res. **LE** are logical instructions and provide a logical comparison of DR register with a memory content assigned via operand or with a constant. If DR is smaller or equal then a memory content or constant the RLO is set to 1 otherwise is set to zero

Instructions **GT** res. **GE** are logical instructions and provide a logical comparison of DR register with a memory content assigned via operand or with a constant. If DR is greater or equal then a memory content or constant the RLO is set to 1 otherwise is set to zero

.

Instruction **EQ1** is introduced to maintain a compatibility with automaton NS915. A comparison is provided only when value of RLO is **1**, otherwise in RLO remains value 0. The instruction EQ1 can be used for comparison of larger memory areas because instruction EQ1 can be in multiple mode. Two instructions and one label can substitute instruction EQ1:

```
                    JL0     OBSKOK
EQ      BUNKA equal:              EQ1     BUNKA
OBSKOK:
```

When a prefix "**DWRD.**" is used before an operand then comparison of an extended 32-bit DR register with 32-bit operand (DWORD) is performed.

A Type redefining is described in Chapter.

Example:
Jump to label MENSI, when a cell BUNKA1 is smaller then BUNKA2

```
        LOD         BUNKA1          ;read BUNKA1 to DR
        LT          BUNKA2          ;if DR < BUNKA2 then RLO=1, otherwise 0
        JL1         MENSI           ;jump to MENSI if RLO=1
```

Example:
Compare value of cell NASTAVENI with cell BUNKA, when expression ALFA*NOT(BETA) is equal 1. Write a result to GAMA:

```
        LDR         ALFA            ;read bit ALFA to RLO
LA              -BETA           ;logical AND of RLO with a negation of BETA
```

```
LOD     NASTAVENI       ;read cell NASTAVENI to DR
EQ1     BUNKA           ;conditional comparison when RLO=1
WR      GAMA            ;write RLO to GAMA
```

Example:
Compare two memory areas PAM1 and PAM2 each has a length 8 BYTE.

```
LOD     DWRD.PAM1       ;read 4 Bytes from PAM1 to DR 32 bits
EQ      DWRD.PAM2       ;compare DR 32 bits with 4 BYTE PAM2
LOD     DWRD.(PAM1+4)   ;reads next 4 BYTE fromPAM1 to DR 32 bits
EQ1     DWRD.(PAM2+4)   ;perform next comparison only if
                        ;first one got equality RLO=1
JL1     ROVNO           ; jump if not equal
```

Example:
Set bit AKCE to log 1 when a cell MATTL is equal to code ´W´

```
LOD     MATTL           ;read cell MATTL to DR
EQ      CNST.'W'        ;compare DR with a constant an setting of RLO
FL1     1,AKCE          ;if RLO=1 then set bit AKCE to log 1
```

Example:
If a difference counter DIFCIT_X (DWORD) exceeds value LIMIT jump to ERROR.

```
CLI                     ;interrupt disabled (ASM86)
LOD     DWRD.DIFCIT_X   ;read difference counter 32 bits to DR
STI                     ;interrupt enabled (ASM86)
ABS     DWRD            ;absolute value of 32 bits DR reg.
GE      DWRD.LIMIT      ;if DR(32 bits) >= LIMIT (DWORD),then
                        ;RLO=1, otherwise RLO=0
JL1     ERROR           ;jump to error
```

3.15 Conversion and Moving Register and Memory

instruction	**CONDR** **CONRD**

operation	**CONDR**	**conversion DR → RLO**
	CONRD	**conversion RLO → DR**
syntax	**CONDR (CONRD)**	
	CONDR (CONRD)	**[DWRD]**

Instruction **CONDR** performs a conversion of register DR to bit register RLO. If register DR is zero, the RLO register is set to 0. If register DR is nonzero the RLO register is set to 1. The DR register remains unaffected.

Instruction **CONRD** performs a conversion of a bit register RLO into data register DR . If register RLO is zero then DR register is set to 0. If register RLO is 1 then DR register is set to value FFFFh. The RLO register remains unaffected. Instruction CONRD is the end- instruction *for logical expression*.
If a parameter **DWRD** is used the conversion is performed with an extended 32-bit DR register.

instruction	**MV**

operation	**MV**	**memory move**
syntax	**MV**	**source, destination, number**

Instruction **MV** performs a move of memory area. Parameter "source" is a source address - from where a move starts, parameter "destination" is the address of destination - to where the memory area shall be moved and parameter "number" is a number of bytes to be moved.

instruction	**CLEAR**

operation	**CLEAR**	**memory clear**
syntax	**CLEAR**	**start, end**
	CLEAR	**GLOBAL, ALL**
	CLEAR	**INTERNAL, ALL**

Instruction **CLEAR** performs a memory clear. Parameter "start" is a starting address of memory area to be cleared and parameter "end" is the end address of this memory area. The instruction in systems family CNC8x9 clears memory of global variables and memory of local variables as well. An appropriate area is assigned according to variable address in instruction parameter.

Instruction “**CLEAR GLOBAL, ALL**“ clears a global data including initial variable mechanism and timers.

Then after this instruction a new initialization of all mechanisms must occur. The instruction is used for example in a module PIS_CLEAR (MODULE_CLEAR), to ensure that clearing (start and stop) of PLC program was equivalent with a run of module PIS_INIT (MODULE_INIT), which runs when a machine is switched ON only.

Instruction "**CLEAR INTERNAL, ALL**" clears all local data defined by PLC programmer including "automatic" variables defined in development of a PLC836 language. Instruction is used for example in a module PIS_CLEAR (MODULE_CLEAR).

3.16 Procedures

Defining and calling of the procedures is valid only for family CNC8x9 – DUAL. In all files of PLC program can be defined procedures (subroutines) and also in all files can be all procedure calls defined in other files.

statement	**PROC_BEGIN** **PROC_END** **PROC_CALL**

operation	**PROC_BEGIN**	**beginning of procedure**
	PROC_END	**end of procedure**
	PROC_CALL	**calling of procedure**
Syntax	**PROC_BEGIN**	**name**
	PROC_END	**name**
	PROC_CALL	**name**

Statements PROC_BEGIN and PROC_END provide definition of a procedure, which have a name of procedure as parameter. The procedure location during its definition can be at any place in module PROVOZ_VYSTUP (MODULE_MAIN). Execution of instruction skip memory area, which is assigned by procedure definition. The procedure is called by statement **PROC_CALL** with appropriate parameter (the name of procedure). When procedure ends the program continues to execute instruction placed immediately next to procedure calling.

The event procedures:

In the PLC836 language some procedure names are exclusively destined for a special purpose and are triggered automatically when a particular event occurs. Those names are as follows:

_ON_ESET	The procedure is automatically called in case of PLC error. A location of the procedure can be at any file which content a PLC program. The TECHNOL compiler investigates the presence of such a procedure and if exists triggers it via statement z rozvoje instrukce ESET. (see Chapter „Error Codes and Messages of PLC program".
_ON_MSET	The procedure is automatically called when a program's information message occurs. A location of the procedure can be at any file which content a PLC program. The TECHNOL compiler investigates the presence of such a procedure and if exists triggers it via statement z rozvoje instrukce MSET. (see Chapter „Error Codes and Messages of PLC program").
_ON_REK	The procedure is automatically called if changing of machine constants occurs. A location of the procedure can be at any file which content a PLC program. The TECHNOL compiler investigates the presence of such a procedure and if exists triggers it after new

	reconfiguration of the system. At this place can be a new BCD to binary conversions for machine constants used in PLC program.

3.17 Auxiliary Statements

The PLC836 language works with auxiliary statements, which take place during PLC program compilation only. All statements described bellow are non-operand type.

Statement	**SYMBOLTAB** **INTERSTACK** **CHECK** **P386** **CPU04** **VERINSTRU**

Statement **SYMBOLTAB** causes shortening of symbol table during PLC program compilation. It is recommended to use this statement only when compiler reports an overflow of the symbol table. When during the PLC program compilation products of company BORLAND are used the SYMBOLTAB is without any effect.

Instruction **INTERSTACK** uses its own (internal] processor stack during evaluation of logical expression. Using instruction INTERSTACK speeds up a PLC program execution but problems may occur with stack unequability even if compiler TECHNOL checks this enequability. For instance setting of BREAK-POINTER in the program area is inhibited.

Instruction **CHECK** checks stack equability. The stack equability is checked always at the end of appropriate module. During program debugging it is possible to use an instruction CHECK, which checks stack equability at assigned place and thus helps to localize program errors.

Instruction **P386** modifies instructions of PLC836 language to fit in "assembler 386" for 80486DX processor used in CPU04 processor card. Data operation in double word notation use for inner notation register ECX instead of registers BP, CX. This instruction provides all actions, which are necessary for transition to processor CPU04. Instruction P386 must precede instruction DATA (see chapter Structure of PLC program). **In systems type DUAL may not be used.**

Instruction **CPU04** provides all actions which are necessary for transition to card CPU04 (with processor 80486DX), but do not modify instructions of PLC 386 language. This instruction is may be used in case that PLC program has a lot assembler 86 instructions. (e.g. work with double word notation in registers BP, CX). Instruction P386 must precede instruction DATA (see chapter Structure of PLC program). **In systems type DUAL may not be used.**

Instruction **VERINSTRU** modifies the instructions according to defined version. A parameter of this instruction contains a name of instruction to be modified. To the instruction's name is via character "underline" ('_') connected a version type. In TECHNOL compiler staring with version 2.3 are valid following combinations:

FL_V1 **FL1_V1** **WR_V1**

All described modifications give as a possibility to use in parameters of instructions FL, FL1 and WR more bit operands which may not be located in the same byte The operands are not bracketed in a " less than" and " greater then" signs but are separated by commas. **In the systems type DUAL is this modification performed automatically.**

Starting with software version 4.036 is possible to use a modification of all instructions that work with analog inputs and with measurement actions in dependence of coordinate unit type. **In systems type DUAL may not be used.**

SU02 **SU04**

Example:

```
PAM1:  DFM    , , ,ALFA,BETA, , ,
PAM2:  DFM    ,GAMA, , , , , ,
PAM3:  DFM    , , , , ,DELTA, ,

       VERINSTRU    FL_V1,  FL1_V1,  WR_V1

       FL     1, ALFA, GAMA, DELTA
       WR     GAMA, DELTA
       FL1    0, DELTA, BETA
```