3

3. ZÁKLADNÍ INSTRUKCE JAZYKA TECHNOL

Jazyk TECHNOL je určen pro efektivní programování interfejsu pro systémy CNC8x9 a CNC872. Jazyk používá výhradně symbolických adres a to i při práci s jednotlivými bity paměti.

3.1 Zápis zdrojového programu

Zdrojové programy interfejsu lze napsat v prostředí ladícího programu Wintechnol nebo v libovolném textovém ASCII editoru. Programování je prováděno v mnemonickém kódu. Pro zápis programu platí podobná pravidla jako pro zápis zdrojového programu assembleru.

Programový řádek obsahuje:

a) Návěští
Je nepovinné a určuje symbolicky adresu místa v paměti pro proměnnou nebo na kterou má být např. proveden skok v programu. Návěští může obsahovat max. 31 znaků, t.j. písmen, číslic a tří speciálních znaků:

_ podtrhovátko
? otazník
@ "zavináč"

Prvním znakem nesmí být číslice. Návěští musí být ukončeno dvojtečkou. Návěští může být uvedeno na samostatném řádku, přičemž se vztahuje k nejbližšímu následně uvedenému řádku s operačním kódem, konstantou nebo alokací paměti.

b) Operační kód
Je mnemonický zápis kódu příslušné instrukce. Jako operační kód jsou povoleny instrukce, uvedené v další části návodu. Kromě toho je umožněno používat i instrukce assembleru 80386 i když pro zápis programu interfejsu to není nutné.

c) Operand
Jednotlivé instrukce mohou být bez operandu (např. instrukce BCD) nebo většinou s jedním operandem (např. `LDR ALFA`), případně některé instrukce mohou mít víc operandů oddělených čárkami.

d) Komentář
Text, uvedený za středníkem se považuje za komentář a ignoruje se. Komentář může být uveden na samostatném řádku nebo na řádku společně s kódem.

Příklad:

```
                              ;toto je komentar
PAM12:      LDR   ALFA        ;toto je komentar
```

3.2 Pracovní registry jazyka TECHNOL

Instrukce jazyka TECHNOL využívají tyto registry:

a) Jednobitový takzvaný **registr logických operací**, který se bude v dalším textu označovat zkratkou **RLO**. (Fyzicky se jedná o bit s váhou 40h - registru AH mikroprocesoru.)

b) **Datový registr** , který se bude v dalším textu označovat zkratkou **DR**. Instrukce mohou pracovat s datovým registrem se šířkou slova 8, 16, 32, nebo 64 bitů. Pro operce v reálných číslech se používá 64 bitový registr `DR_REAL`.
(Fyzicky se jedná o registr ECX mikroprocesoru. V případě rozšířeného 64 bitového DR registru jsou to registry ECX a EDX).

c) Zásobník - jedná se o 8 buňek typu WORD.

Pokud se používají pouze instrukce jazyka TECHNOL, není fyzická reprezentace registrů pro programátora důležitá. Registry se musí respektovat pouze při eventuelním programování částí programu v assembleru 80386.

Změna hodnoty v registru RLO neovlivní datový registr DR a obráceně, změna v datovém registru DR neovlivní registr RLO. Jazyk TECHNOL automaticky rozpoznává, kdy se má použít bajtový, wordový nebo double-wordový přístup jak k registru DR, tak k paměti.

3.3 Deklarace paměti

Jazyk TECHNOL umožňuje pracovat s libovolnými adresami v symbolické formě. Jako symbolické adresy lze definovat adresy v programu i adresy bitových, 8-bitových (typu BYTE), 16-bitových (typu WORD) a 32-bitových (typu DWORD) buněk paměti RAM. Symbolická adresa je definována jako slovo o maximálně 31 znacích (délka není omezená, ale významných je prvních 31 znaků), které může obsahovat jak písmena, tak čísla, přičemž jako první musí být vždy uvedeno písmeno. Toto slovo však nesmí být uvedeno v seznamu klíčových slov, která mají již předem daný význam. Seznam klíčových slov je uveden v příloze.
Přiřazení symbolické adresy určité fyzické adrese lze provést dvěma způsoby. Jedná-li se o deklaraci adresy (místa) v programu nebo deklaraci slova v paměti RAM, postačí napsat za příslušný symbol dvojtečku a adresa je pak dána polohou tohoto symbolu nebo-li adresa se vztahuje na instrukci následující za dvojtečkou.

Příklad:
V programu, který je vyjádřen instrukcemi 1 až 5, chceme definovat adresu instrukce 2 a 3 symboly ALFA a BETA.

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

3.4 Definice bitových a datový proměnných a konstant

instrukce	**DFM**

funkce **definice bitu v paměti**

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

parametry	**„bit0,..bit7"**	**symbolické názvy bitových proměnných**
	„name"	**název slabiky BYTE**

Chceme-li symbolicky definovat jednotlivé bity paměti RAM, použijeme speciální instrukce **DFM**. Jedna instrukce DFM definuje vždy v místě svého zápisu jednu slabiku paměti (8 bitů). Celou slabiku můžeme symbolicky pojmenovat pomocí symbolu, zapsaného před tuto instrukci a zakončeného dvojtečkou. Takto definovaná slabika umožňuje kromě bitového i bajtový přístup. Symboly, které jsou uvedeny za příkazem DFM, deklarují pak jednotlivé bity této slabiky paměti. První symbol náleží bitu d0, druhý bitu d1 atd., až osmý symbol náleží bitu d7. Symboly musí být odděleny čárkami. V případě, že některý bit definovat nechceme, můžeme příslušný symbol vynechat. Čárky však uvedeny být musí!

Jednotlivé vstupy a výstupy jednotek periferních obvodů jsou snímány a plněny po osmicích. Abychom si tvorbu programu PLC usnadnili, budeme pracovat s těmito vstupy a výstupy výhradně tímto způsobem. Na začátku programu přečteme pomocí speciální instrukce všechny vstupy do paměti RAM a na konci programu PLC naopak vymezené paměti RAM zapíšeme do výstupů. Jednotlivé vstupy či výstupy pak můžeme deklarovat a obsluhovat stejně jako bity paměti RAM.

instrukce	**DS 1** **DS 2** **DS 4** **DS n**

funkce	**DS 1**	**vymezení paměti o délce 1 BYTE**
	DS 2	**vymezení paměti o délce 1 WORD**
	DS 4	**vymezení paměti o délce 1 DWORD**
	DS n	**vymezení paměti o délce n bajtů**

syntax
name: DS 1
name: DS 2
name: DS 4
name: DS n

parametry	**„1,2,4,n"**	**počet bajtů paměti**
	„name"	**název datové proměnné**

Instrukce **DS** se používá pro definování proměnných a inicializaci paměti. Operandem se deklaruje délka vymezené paměti v bajtech. Instrukce DS 1 s návěštím proměnné deklaruje tuto proměnnou jako BYTE, instrukce DS 2 deklaruje proměnnou jako WORD a DS 4 deklaruje proměnnou jako DWORD.

instrukce	**EQUI**

funkce **definice konstanty**

syntax **EQUI const, immed**

1.parametr **„const"** **název konstanty**
2.parametr **„immed"** **hodnota konstanty**

Instrukce **EQUI** definuje konstanty použité v programu. První parametr je symbolický název konstanty a druhý parametr je jeho hodnota. Hodnota konstanty může být zapsána :

```
1. dekadicky            např. 123, 54213
2. hexadecimálně        např. 0F8H, 15H
3. znakově              např. 'A', '8'
```

Příklad:
V paměti RAM chceme definovat slabiku GAMA a její jednotlivé bity d0 až d7 jako symboly GAMA0 až GAMA7. Na adrese DELTA chceme symbolicky definovat bit d0 = DELTAM, bit d2 = DELTAG a bit d7 = DELTAT.

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

Příklad:
Pro bajtovou proměnnou ALFA vymezte délku 1 byte.
Pro double-wordovou proměnnou BETA vymezte délku 4 byte.
Pro proměnnou GAMA vymezte pole o délce 30 byte.
Definujte konstantu DELTA s hodnotou 1000.

```
ALFA:       DS          1
BETA:       DS          4
GAMA:       DS          30
EQUI        DELTA, 1000
```

3.5 Logické operace s bity paměti a RLO

instrukce	**LDR**

funkce **čtení bitu paměti do RLO /LOAD RLO/**

syntax
```
LDR     [-] bit
LDR     [-] [adr.]  bit   složitější způsoby adresace
LDR     [-] [$+xx.] bit
```

parameter **„bit"** **symbolický název bitu**

Instrukce **LDR** provádí plnění registru logických operací RLO bitem zadané paměti.

Instrukce LDR má povinný operand. Tímto operandem je symbolický název bitu paměti RAM, definovaný pomocí instrukce DFM. Je-li tento symbolický název bitu paměti uveden bez znaménka nebo se znaménkem plus, je příslušný bit čten přímo. Je-li před symbolickým názvem bitu znaménko -, čte se příslušný bit paměti negovaný.

Složitější způsoby adresace bitu:

V případě, že je potřeba načíst bit z jiného místa paměti, než je daný bit definovaný, uvede se adresa paměti před názvem bitu a oddělí se tečkou. Místo adresy místa je možno použít i název bitu, který je tam definovaný, způsoby indexace pomocí [BX] a prvky struktury. Tímto je například umožněna práce s rozsáhlejším bitovým polem. Když potřebujeme načíst bit z místa o několik bajtů dál něž je bit definován (například o 12), můžeme místo vlastního názvu použít znak $+ ($ + 12). Složitější způsoby adresace bitu je možno použít u instrukcí LDR, LA, LO, LX a pro instrukce WR, FL1 a FL.
Složitější způsoby adresace bitu se používají i v případě, když je potřeba načíst bit s příslušnou váhou s libovolného místa paměti. V PLC programu se formálně nadeklarují univerzální názvy bitů, například B0,B1,...,B7, které se pak používají pro přístup k libovolnému místu paměti.

Příklady adresace bitu:
```
DFM   B0,B1,B2,B3,B4,B5,B6,B7 ;formální definice bitů

LDR   ALFA                    ;načtení bitu ALFA z paměti, kde je definován
LDR   BUN5.ALFA               ;z buňky BUN5 se načte bit ze stejné pozice,
                              ;jak je původně definován bit ALFA.
LDR   BUN5.B4                 ;načtení 4.bitu (váha 10h) z buňky BUN5
LDR   -(BUN5+3).ALFA          ;načte se negace bitu z buňky BUN5+3
LDR   BUN5[BX].ALFA           ;použití indexace (viz kapitola.18)
LDR   -(BUN5[BX]-6).ALFA      ;další kombinace
LDR   (BUN5[BX]+PRVNI+2).ALFA ;PRVNI je prvek struktury
LDR   $+3.ALFA                ;načtení bitu ALFA z pozice o 3 bajty dál
```

3.6 Princip zásobníku a koncových instrukcí

Jsou-li dvě nebo více instrukcí LDR zapsány v programu za sebou, aniž byl mezi nimi registr log. operací nějak využit (pro zápis do paměti či podmíněný skok), ukládá se před další instrukcí LDR předchozí stav RLO do zásobníku. S takto uloženými hodnotami lze potom pracovat pomocí instrukcí LA, LO nebo LX bez udání operandu. Pomocí zásobníku je takto umožněno programovat závorkové operace.

Vyrovnanost zásobníku se kontroluje na konci každého modulu (viz dále) a na konci každého sekvenčního logického celku. Nevyrovnanost zásobníku ohlásí chybu v úvodní fázi kompilace překladače TECHNOL. Kontrolu vyrovnanosti zásobníku ve fázi odlaďování programu možno vnutit pomocnou instrukcí **CHECK** (viz popis pomocných instrukcí).

Každá logická rovnice naprogramovaná pomocí instrukcí **LDR, LA, LO** a **LX** musí být ukončena tzv. **koncovou instrukcí**, která ukončuje logickou rovnici. Koncová instrukce nuluje vnitřní příznak vnořování do zásobníku, takže u první následující instrukce **LDR** nedojde k uložení RLO do zásobníku.

Koncové instrukce v jazyku TECHNOL jsou:

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

instrukce	**LA** **LO** **LX**

funkce	**LA**	**logický součin s RLO /AND/**
	LO	**logický součin s RLO /OR/**
	LX	**nonekvivalence s RLO /XOR/**
syntax 1	**LA**	
	LO	
	LX	
syntax 2	**LA**	**[-] bit**
	LO	**[-] bit**
	LX	**[-] bit**
	LA,LO,LX	**[-] [adr.] bit složitější způsoby adresace**
	LA,LO,LX	**[-] [$+xx.] bit**
parametr	**„bit"**	**symbolický název bitu**

Instrukce **LA**, **LO** a **LX** mají operand volitelný, t.j. může být zadán, ale nemusí. Tímto operandem je symbolický název bitu paměti RAM, definovaný pomocí instrukce DFM. Je-li tento symbolický název bitu paměti uveden bez znaménka nebo se znaménkem plus, je příslušný bit čten přímo. Je-li před symbolickým názvem bitu znaménko -, čte se příslušný bit paměti negovaný.

Jsou-li dvě nebo více instrukcí LDR zapsány v programu za sebou, aniž byl mezi nimi registr log. operací nějak

využit (pro zápis do paměti či podmíněný skok), ukládá se před další instrukcí LDR předchozí stav RLO do zásobníku. S takto uloženými hodnotami lze potom pracovat pomocí instrukcí LA, LO nebo LX bez udání operandu. Instrukce LA provede logický součin naposledy uložené hodnoty zásobníku s RLO a výsledek uloží do RLO. Instrukce LO provede obdobně logický součet a instrukce LX nonekvivalenci. Vyrovnanost zásobníku se kontroluje na konci každého modulu (viz dále) a na konci každého sekvenčně logického celku. Nevyrovnanost zásobníku ohlásí chybu v úvodní fázi kompilace překladače TECHNOL.

Složitější způsoby adresace jsou popsány u instrukce LDR.

instrukce	**CA**
funkce	**negace RLO**
syntax	**CA**

Instrukce **CA** je bez operandu a provádí negaci obsahu registru logických operací RLO. Má-li před touto instrukcí registr log. operací hodnotu 1, po této instrukci bude mít hodnotu 0 a naopak.

instrukce	**EDGE_H** **EDGE_L**	
funkce	**EDGE_H**	**detekce nástupné hrany**
	EDGE_L	**detekce sestupné hrany**
syntax	**EDGE_H**	**bit**
	EDGE_L	**bit**
parametr	**„bit"**	**symbolický název bitu**

Instrukce **EDGE_H** a **EDGE_L** testují příchod hrany u bitové proměnné. V případě výskytu příslušné hrany nastaví registr RLO na hodnotu log.1. Když není hrana bitové proměnné detekována, nastaví registr RLO na hodnotu log.0.
Instrukce nedetekují výskyt první hrany po startu PLC, nebo po stopu PLC a opětovném startu. V tomto případě se jedná jen o první nastavení hodnot proměnných, například podle aktuálního stavu vstupů.
Instrukce je možno použít i v rychlém modulu MODULE_FAST a také v časových blocích DFTM.

Příklady:
Mějme symbolicky definovat bity paměti A1, A2. Naplňte registr logických operací v závislosti na stavu těchto pamětí dle logického výrazu:

```
RLO = A1 + A2

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

Je-li za instrukcí `LA`, `LO` nebo `LX` uveden operand (symbolicky označený bit paměti), má instrukce následující význam:

Instrukce `LA` s operandem provede logický součin zadaného bitu paměti s registrem logických operací a výsledek opět uloží do RLO. Instrukce `LO` provede obdobně logický součet instrukce `LX` nonekvivalenci. Podíváme-li se zpět na příklad 3, zjistíme,že jeho další možné řešení je následující:

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

Řešení 3b je co do funkce naprosto rovnocenné řešení 3a, délka programu je však v tomto případě kratší.

Pomocí výše uvedených instrukcí (`LDR`, `LA`, `LO`, `LX`, `CA`) lze tedy naprogramovat libovolnou logickou podmínku (logický výraz). Postup programování názorně ukazují následující příklady 4, 5, 6.

Příklad:
Naprogramujte logický výraz (pozn.: <> je NONEKVIVALENCE):

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

      LDR         ALFA
      LX          BETA
      LDR         GAMA        ;ULOŽÍ RLO DO ZÁSOBNÍKU A NAČTE BIT GAMA
      LX          DELTA
      LO                      ;LOG.SOUČET RLO SE ZÁSOBNÍKEM
```

Příklad:
Naprogramujte logický výraz:

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

      LDR         -A1
      LA          A2
      LA          -A3
      LDR         -B1   ;ULOŽÍ RLO DO ZÁSOBNÍKU A NAČTE NEGACI BITU B1
      LA          B2
      LO                ;LOG.SOUČET RLO SE ZÁSOBNÍKEM
      LDR         C2    ;ULOŽÍ MEZIVÝSLEDEK DO ZÁSOBNÍKU A NAČTE BIT C2
      LO          -C3
      LA                ;LOG.SOUČIN RLO SE ZÁSOBNÍKEM
```

Příklad:
Naprogramujte logický výraz:

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

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

Příklad:
Když signál ALFA má nástupní hranu, nastavte do bitu BETA log.1, jinak log.0:

```
      EDGE_H      ALFA
      WR          BETA
```

3.7 Zápis bitů do paměti

instrukce	WR

```
funkce      zápis obsahu RLO do paměti

syntax 1    WR          bit

syntax 2    WR          bit1 [,bit2 ...]
            WR          [adr.] bit  složitější způsoby adresace
            WR          [adr.] bit  [,[adr2.]bit2 ...]
            WR          [$+xx.]bit  [,[$+yy.]bit2 ...]

parametry   „bit"       symbolický název bitů
```

Instrukce **WR** provádí zápis obsahu RLO do vyjmenovaných bitů jedné slabiky paměti. Obsah RLO a DR se po vykonání této instrukce nemění. Instrukce WR je ***koncová instrukce*** pro logické rovnice.
Instrukce umožňuje v parametrech instrukce použít více bitových operandů, které nemusí být umístěny v jednom bajtu. Operandy se oddělí čárkou.
Složitější způsoby adresace jsou popsány u instrukce LDR.

instrukce	FL

```
funkce      nastavování bitů v paměti

syntax 1    FL    0, bit
            FL    1, bit

syntax 2    FL    0, bit1 [,bit2 ...]
            FL    1, bit1 [,bit2 ...]
            FL    0, [adr.] bit          složitější způsoby adresace
            FL    1, [adr.]  bit [,[adr2.]bit2 ...]
            FL    0, [$+xx.] bit [,[$+yy.]bit2 ...]

1.parametr  „0, 1"      logická hodnota pro plnění bitů
2.parametr  „bit"       symbolický název bitů
```

Instrukce **FL** zajistí, nezávisle na obsahu RLO, plnění bitů jedné slabiky paměti nulou nebo jedničkou. Za příkazem FL, jako první operand, následuje hodnota 0 nebo 1. Jako druhý operand je nutné uvést seznam bitů, do kterých se bude uvedená hodnota plnit. Bity nemusí být umístěny v jednom bajtu. Obsah RLO a DR se po vykonání této instrukce nemění.
Instrukce umožňuje v parametrech instrukce použít více bitových operandů, které nemusí být umístěny v jednom bajtu. Operandy se oddělí čárkou.

Složitější způsoby adresace jsou popsány u instrukce LDR.

instrukce	FL1

```
funkce      podmíněné nastavování bitů v paměti

syntax 1    FL1   0,bit
            FL1   1,bit

syntax 2    FL1   0, bit1 [,bit2 ...]
            FL1   1, bit1 [,bit2 ...]
            FL1   0, [adr.] bit          složitější způsoby adresace
            FL1   1, [adr.]  bit [,[adr2.]bit2 ...]
            FL1   0, [$+xx.] bit [,[$+yy.]bit2 ...]

1.parametr  „0, 1"      logická hodnota pro plnění bitů
2.parametr  „bit"       symbolický název bitů
```

Instrukce **FL1** zajistí plnění bitů jedné slabiky paměti nulou nebo jedničkou jenom tehdy, když je v registru RLO hodnota **1**. Za příkazem FL1, jako první operand, následuje hodnota 0 nebo 1. Jako druhý operand je nutné uvést seznam bitů, do kterých se bude uvedená hodnota plnit. Bity nemusí být umístěny v jednom bajtu. Obsah RLO a DR se po vykonání této instrukce nemění. Instrukce FL1 je ***koncová instrukce*** pro logické rovnice. Složitější způsoby adresace jsou popsány u instrukce LDR.

Instrukce FL1 se dá nahradit pomocí dvou instrukcí a jednoho návěští:

```
      JL0   OBSKOK
      FL    1,BIT       ekvivalent: FL1   1,BIT
OBSKOK
```

instrukce	MOVR MOVR1

```
funkce      přímý přesun bitů v paměti

syntax      MOVR (MOVR1)      dst, src
            MOVR (MOVR1)      [adr.]dst, [adr.]src

1.parametr  „dst" symbolický název cílového bitu
2.parametr  „src" symbolický název zdrojového bitu
```

Instrukce **MOVR** zajistí přímý přesun bitu z jedné slabiky paměti (2.operand) do druhé slabiky paměti (1.operand). Obsah RLO a DR se po vykonání této instrukce nemění. Instrukce **MOVR1** provede přesun bitu podmíněně, jenom když je v registru RLO hodnota **1**. Instrukce MOVR1 je ***koncová instrukce*** pro logické rovnice. Instrukce umožňuje použít složitější způsoby adresace, které jsou popsány u instrukce LDR.

Příklad:
Symbolicky označené bity paměti PETR, IVAN, JANA naplňte v závislosti na pamětech PAVEL a EVA takto:

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

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

Příklad:
Mějme definované bity paměti těmito symboly:

B1, B2, A2
Naplňte tyto paměti na hodnotu log.1:

```
      FL          1,A2,B1,B2
```

3.8 Větvení programu

instrukce	**JUM** **JL0** **JL1**

funkce	**JUM**	**nepodmíněný skok**
	JL0	**skok, je-li RLO = 0**
	JL1	**skok, je-li RLO = 1**
syntax	**JUM**	**adr**
	JL0	**adr**
	JL1	**adr**
parametr	**„adr"**	**adresa programu**

Všechny tyto instrukce, které umožňují větvení programu, nebo-li programový skok, musí mít jako operand uvedenu adresu instrukce, která má být vykonávána dál v případě splnění příslušné podmínky skoku. Není-li tato podmínka splněna, pokračuje procesor ve zpracování instrukce následující (skok se neprovede). Tato adresa se zadává symbolickou formou, přičemž příslušný symbol lze definovat pomocí dvojtečky v kterémkoli místě programu.
Instrukce **JUM** nevyžaduje splnění žádné podmínky, skok se tedy provede vždy. Instrukce **JL0** zajistí skok na zadanou adresu pouze v případě, že RLO = 0. Instrukce **JL1** zajistí skok na zadanou adresu pouze v případě, že RLO = 1. Instrukce JL0 a JL1 jsou ***koncové instrukce*** pro logické rovnice.

Příklad:
Zapište pomocí instrukcí jazyka TECHNOL následující logický algoritmus:

```
Je-li A1 <> A2, naplň 0 do B1 a B2.
Je-li A1 . A2 = 1, naplň 1 do B3.

            LDR         A1
            LX          A2
            JL1         NAV1
            FL          0,B1,B2
 NAV1:      LDR         A1
            LA          A2
            FL1         1,B3
```

3.9 Způsoby předefinování typu u datových proměnných

Jazyk TECHNOL v datových operacích přistupuje k operandům automaticky podle toho, jaká byla jejich definice. Například instrukce LOD načte hodnotu typu BYTE, WORD, DWORD nebo konstantu podle způsobu definice operandu:

```
DEFINICE:                       OPERACE:

BUNKA1:     DS 1          ..... LOD   BUNKA1      načte BYTE do DR registru

BUNKA2:     DS 2          ..... LOD   BUNKA2      načte WORD do DR registru

BUNKA3:     DS 4          ..... LOD   BUNKA3      načte DWORD do DR registru

EQUI        CON,124       ..... LOD   CON         načte konstantu 124 do DR
```

Některé instrukce, které pracují s DR registrem, mají možnost při svém vykonávání předefinovat typ datové proměnné. Následující popis se bude týkat instrukcí `LOD, STO, STO1, AD, SU, EQ, EQ1, LT, GT, LE, GE, RR, RL, TM, TIM, TEX0, TEX1, MULB a DIVB.`

Instrukce můžou mít nepovinný prefix před názvem proměnné (TYPE.), který může předefinovat nebo dodefinovat typ proměnné na :

- **CNST**
- **BYTE**
- **WORD**
- **HIGH**
- **DWRD**
- **QWRD**
- **REAL**

Prefix se připojí k operandu instrukce pomocí tečky.
Instrukce `LOD, AD, SU, EQ, EQ1, LT, GT, LE, GE, RR, RL, MULB, DIVB` mohou mít jako operand konstantu, která není definovaná pomocí instrukce **`EQUI`**. V tomto případě se uvede před hodnotou konstanty typ: **`"CNST."`** Použití prefixu CNST. způsobí, že instrukce budou pracovat s 32-bitovým DR registrem. Pokud přímo zadaná hodnota obsahuje desetinnou tečku, považuje se to za zadání čísla v reálném tvaru..

Příklad:
Zapište do buňky odměřování systému v ose X:

```
BUNKA:      DS    4

            CLI                     ;zákaz přerušení
            LOD   DWRD.B_POL        ;načte do rozšířeného DR registru B_POL
            AD    DWRD.B_INK        ;připočte dvě slova B_INK
            STI                     ;povolení přerušení
            STO   BUNKA             ;zapíše do BUNKA 32 bitů
```

Příklad:
Použití prefixů:

```
            LOD   CNST.123          ;přímé naplnění konstantou
            AD    CNST.50h          ;připočte 50 hexadecimálně
```

```
AD    BYTE.ALFA          ;připočte spodní byte ALFA
STO   WORD.BETA          ;zapíše do BETA jako WORD
```

Příklad:

```
LOD   CNST.1.8           ;přímé naplnění reálnou konstantou
AD    REAL.BUN_R         ;připočte reálnou buňku
```

Možnosti deklarace a předefinování typu u datových proměnných:

	DEKLARACE				***MOŽNOST PŘEDEFINOVÁNÍ TYPU***						
	BYTE	**WORD**	**CNST**	**DWRD**	**BYTE.**	**WORD.**	**HIGH.**	**CNST.**	**DWRD.**	**QWRD.**	**REAL.**
LOD	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
STO	✓	✓	.	✓	✓	✓	✓	.	✓	✓	✓
STO1	✓	✓	.	✓	✓	✓	✓	.	✓	✓	✓
TM	✓	✓	.	.	✓	✓	✓	.	.	.	.
CU,CD	✓	✓	.	.	.	.	.	.	.	.	.
CUBCD	✓	✓	✓	.	.	.	.	.	.	.	.
AD,SU	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
MULB	✓	✓	✓	✓	✓	✓	✓	✓	✓	.	✓
DIVB	✓	✓.	✓	✓	✓	✓	✓	✓	✓	.	✓
RR,RL	✓	.	✓	.	✓	.	✓	✓	.	.	.
EQ	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
EQ1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
LT,GT	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
LE,GE	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
TIM	✓	✓	.	.	✓	✓	✓	.	.	.	.
TEX0,1	✓	✓	.	.	✓	✓	✓	.	.	.	.

3.10 Zápis a čtení do paměti z datového registru DR

instrukce	**LOD**

funkce **čtení paměti typu BYTE,WORD,DWORD,QWORD,REAL, konstanty do DR**

syntax
```
LOD   [-] val
LOD   [-] [TYPE.] val
LOD   [-] TYPE.(val+n)
LOD   [-] CNTS.immed
LOD       CNTS.ireal
               TYPE = BYTE. WORD. HIGH. DWRD. REAL. QWRD.
```

parametr **„val"** **symbolický název datové proměnné**

Instrukce **LOD** zajistí načtení obsahu paměti o délce BYTE, WORD, DWORD, QWORD, REAL nebo konstanty do datového registru DR (nebo DR_REAL). Adresa paměti se zadává formou operandu instrukce v symbolickém tvaru. Konstanta musí být definována pomocí instrukce **EQUI**, nebo pomocí prefixu "**CNST.**". Použití instrukce s prefixem "CNST." způsobí naplnění DR registru velkosti DWORD. Pokud přímo zadaná hodnota obsahuje desetinnou tečku, považuje se to za zadání čísla v reálném tvaru. Instrukce před načtením vynuluje obsah DR registru a po načtení znaménkově rozšíří DR registr až na 64 bitů.

Zadání znaménka - způsobí, že hodnota bude vzata do DR registru záporně. Ve skutečnosti se jedná o dvojkový doplněk z načtené hodnoty.

instrukce LOD	typ operace (výsledek DR64)	deklarace proměnné	popis
LOD Value	BYTE -> DR8	Value: DS 1	přístup podle deklarace proměnné
LOD Value	WORD -> DR16	Value: DS 2	přístup podle deklarace proměnné
LOD Value	DWORD -> DR32	Value: DS 4	přístup podle deklarace proměnné
LOD Value	Immed -> DR32	EQUI Value,Immed	přístup podle deklarace konstanty
LOD BYTE.Value	BYTE -> DR8		přetypování na BYTE
LOD WORD.Value	WORD -> DR16		přetypování na WORD
LOD DWRD.Value	DWORD -> DR32		přetypování na DWORD
LOD QWRD.Value	QWORD -> DR64		přetypování na QWORD
LOD Immed	Immed -> DR32		přímá hodnota celočíselná
LOD CNST.Immed	Immed -> DR32		přímá hodnota
LOD BYTE.Immed	Immed -> DR8		přímá hodnota + přetypování
LOD WORD.Immed	Immed -> DR16		přímá hodnota + přetypování
LOD DWRD.Immed	Immed -> DR32		přímá hodnota + přetypování
LOD REAL.Value	REAL -> DRr,DR64	Value: DS 8	reálná hodnota proměnné
LOD CNST.Ireal	Ireal -> DRr,DR64		přímá hodnota reálná

Instrukce pro práci s reálnými čísly musí mít povinně prefix „REAL" nebo „CNST" s číslem s desetinnou tečkou. Pro všechny instrukce s reálnými čísly platí, že po operaci zůstává výsledek jak v reálném registru DR_REAL, tak v celočíselném registru DR64. Proto je možno kdykoli pokračovat standardními celočíselnými operacemi. Nelze ale kombinovat celočíselné a reálné operace.

Popis s reálnými čísly bude popsán dále.

Popis:

`Value ..`	název datové proměnné
`Immed ...`	přímá hodnota celočíselná (bez desetinné tečky, může být i hexadecimálně)
`Ireal ...`	přímá hodnota reálná (číslo obsahuje desetinnou tečku)
`REAL...`	reálná hodnota proměnné (64 bitů)
`DR8 ...`	datový registr DR 8 bitů
`DR16 ...`	datový registr DR 16 bitů
`DR32...`	datový registr DR 32 bitů
`DR64...`	datový registr DR 64 bitů celočíselný
`DRr...`	datový registr DR 64 bitů reálný

instrukce	**STO**

`funkce` **zápis DR do paměti typu BYTE,WORD,DWORD,QWORD,REAL**

`syntax`

```
STO          val
STO          [TYPE.] val
STO          TYPE.(val+n)
                   TYPE = BYTE. WORD. HIGH. DWRD. REAL. QWRD.
```

`parametr` **„val“** **symbolický název datové proměnné**

Instrukce **STO** umožňuje zapsat obsah datového registru DR do zadané paměti o délce BYTE, WORD, DWORD, QWORD nebo REAL. Adresa paměti se opět zadává formou operandu instrukce v symbolickém tvaru.

Možnosti operandu instrukce a přetypování je popsáno v kapitole "Způsoby předefinování typu u datových proměnných".

instrukce STO, STO1	typ operace	deklarace proměnné	popis
`STO Value`	`DR8 -> BYTE`	`Value: DS 1`	přístup podle deklarace proměnné
`STO Value`	`DR16 -> WORD`	`Value: DS 2`	přístup podle deklarace proměnné
`STO Value`	`DR32 -> DWORD`	`Value: DS 4`	přístup podle deklarace proměnné
`STO BYTE.Value`	`DR8 -> BYTE`		přetypování na BYTE
`STO WORD.Value`	`DR16 -> WORD`		přetypování na WORD
`STO DWRD.Value`	`DR32 -> DWORD`		přetypování na DWORD
`STO QWRD.Value`	`DR64 -> QWORD`		přetypování na QWORD
`STO REAL.Value`	`DRr -> REAL`	`Value: DS 8`	reálná hodnota proměnné

instrukce	**STO1**

funkce **podmíněný zápis DR do paměti typu BYTE,WORD,DWORD,QWORD,REAL**

```
syntax      STO1        val
            STO1        [TYPE.] val
            STO1        TYPE.(val+n)
                              TYPE = BYTE. WORD. HIGH. DWRD. REAL. QWRD.

parametr    „val"       symbolický název datové proměnné
```

Instrukce **STO1** je podobná instrukci **STO**, ale zápis DR do paměti se provede jenom tehdy, když je v registru RLO hodnota 1. Instrukce STO1 je zavedena pro kompatibilitu s automatem NS915. Důležitý rozdíl vzhledem k instrukci STO je, že instrukce **STO1** je ***koncová instrukce*** vzhledem k logickým operacím.

Je možno použít i obdobnou instrukci **STO0**, která provede podmíněný zápis DR do paměti tehdy, když je v registru RLO hodnota 0.

Možnosti operandu instrukce a přetypování je popsáno v kapitole "Způsoby předefinování typu u datových proměnných".

Instrukce STO1 se dá nahradit pomocí dvou instrukcí a jednoho návěští:

```
            JL0   OBSKOK
            STO   BUNKA       ekvivalent:  STO1  BUNKA
OBSKOK:
```

instrukce	**MOVE** **MOVE1**

funkce **přímý přesun dat v paměti typu BYTE,WORD,DWORD,QWORD,REAL**

```
syntax      MOVE (MOVE1)      dst, src
            MOVE (MOVE1)      [TYPE.]dst, [TYPE.]src
            MOVE (MOVE1)      TYPE.(dst+n), TYPE.(src+n)
                              TYPE = BYTE. WORD. HIGH. DWRD. REAL. QWRD.

1.parametr  „dst"       symbolický název cílové datové proměnné
2.parametr  „src"       symbolický název zdrojové datové proměnné
```

Instrukce **MOVE** provede přímý přesun ze zdrojové buňky z paměti o délce podle typu (2.operand) do druhé cílové buňky paměti (1.operand). Zdrojová a cílová buňka musí mít stejný typ dat. Obsah RLO a DR se po vykonání této instrukce nemění.

Možnosti operandu instrukce a přetypování je popsáno v kapitole "Způsoby předefinování typu u datových proměnných".

Instrukce **MOVE1** je podobná instrukci **MOVE**, ale přesun se provede jenom tehdy, když je v registru RLO hodnota 1. Důležitý rozdíl vzhledem k instrukci MOVE je, že instrukce **MOVE1** je ***koncová instrukce*** vzhledem k logickým operacím.

Příklady:
Mějme symbolicky definované slabiky paměti RAM těmito symboly:

ALFA, BETA, GAMA

Obsah slabiky ALFA zapište do slabiky BETA, konstantu 123H do GAMA.

```
EQUI        K123, 123H

LOD         ALFA
STO         BETA
LOD         K123
STO         GAMA
```

Způsob zápisu je stejný i pokud se jedná o délky typu WORD nebo DWORD.

Příklad:
Mějme symbolicky definované slabiky paměti RAM těmito symboly: ALFA typu WORD a BETA typu BYTE. Záporný obsah slabiky BETA zapište do spodního byte proměnné ALFA.

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

Příklad:
Přepište hodnotu z buňky NASTAVENI do buňky BUNKA, když výraz ALFA*BETA je roven jedné:

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

Příklad:
Zapíše reálnou hodnotu -10.2 do buňky rBun.

```
LOD         CNST.-10.2
STO         REAL.rBun
```

3.11 Realizace časově závislých funkcí

instrukce	DFTM01 DFTM1 DFTM10 DFTM100

funkce	**DFTM01**	**úsek programu aktivován po 0,1 s**
	DFTM1	**úsek programu aktivován po 1 s**
	DFTM10	**úsek programu aktivován po 10 s**
	DFTM100	**úsek programu aktivován po 100 s**
syntax	**DFTM01**	**end**
	DFTM1	**end**
	DFTM10	**end**
	DFTM100	**end**
parametr	**„end"**	**adresa konce úseku programu DFTM**

Aby bylo dosaženo co nejoptimálnějšího využití strojového času procesoru a pro definici nastavované doby časovačů **TM** a **TIM,** jsou časově závislé funkce realizovány ve zvláštních blocích. Takovým programovým blokem je úsek programu, který je na začátku ohraničen instrukcí **DFTM01**, **DFTM1**, **DFTM10** či **DFTM100** a na konci návěštím "end".

Zvláštností těchto programových bloků je to, že jsou aktivovány v delších časových intervalech než ostatní části řídicího programu PLC, které jsou aktivovány po 20ms. Programový blok, definovaný instrukcí DFTM01, je aktivován ("procházen procesorem") po intervalech 0,1 sec. Programový blok, definovaný instrukcí DFTM1, je aktivován po intervalech 1 sec. Programový blok, definovaný instrukcí DFTM10, je aktivován po intervalech 10 sec a programový blok definovaný instrukcí DFTM100 je aktivován po 100 sec.
Instrukce DFTM mohou být použity ve všech souborech i vícekrát.

instrukce	TM

funkce	**časovač závislý na DR, RLO a bloku DFTM**	
syntax	**TM**	**count**
	TM	**[TYPE.] count**
	TM	**TYPE.(count+n)**
	TM	**-**
		TYPE = BYTE. WORD.
parametr	**„count"**	**název čítače časovače**

Instrukce **TM** pracuje jak s registrem DR, tak s registrem RLO. Jako operand této instrukce je nutno zadat symbolickou adresu slabiky paměti, která bude sloužit pro čítání příslušného času (typ BYTE nebo WORD).
Parametr je nepovinný. V tomto případě si překladač definuje "automatickou" proměnnou pro tento čítač. Čítání času v instrukci TM je závislé od toho, v jakém časovém bloku DFTM se instrukce TM nachází. Když se například

instrukce TM nachází v úseku programu definovaném instrukcí DFTM10 (je aktivován po 10 sec.), nastavená doba je v desítkách vteřin.

Instrukce TM pracuje takto:

1) Je-li RLO = 0, obsah zadaného čítače se nuluje a instrukce tím končí.

2) Je-li RLO = 1, provede se porovnání obsahu zadaného čítače s obsahem datového registru DR.
   a) Je-li /count/ >= DR, nastaví se RLO do stavu 1 a instrukce tím končí.
   b) Je-li /count/ < DR, nastaví se RLO do stavu 0 a provede se zvětšení zadaného čítače o jedničku (V bloku DFTM01 to znamená čas 0,1 sec. a v bloku DFTM1 čas 1 sec.).

Kromě vlastních instrukcí TM musí být v bloku časovačů ještě instrukce pro nastavení registru DR a RLO počátečními podmínkami a samozřejmě také instrukce pro nastavení jedné nebo více paměťových buněk dle výsledku této instrukce (RLO).

Příklad:
Realizujte tuto časově závislou funkci:
Je-li bit ALFA po dobu delší než 0,4 sec. ve stavu 1, nastavte do stavu 1 též bit GAMA. K čítání času použijte slabiku paměti CITACA.

```
             EQUI        DOBA,4

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

Příklad:
Je-li logický součin A1 a A2 po dobu delší než 5 sec. roven nule, vynulujte bit GAMA a vynulujte buňku BYTE. K čítání času se použije automatická definice čítače.

```
             DFTM1       NAV50
             ...
             LDR         A1
             LA          A2
             CA
             LOD         CNST.5
             TM          -
             FL1         0,GAMA
             LOD         CNTS.0
             STO1        BYTE
             ...
NAV50:
```

instrukce	CU CD CUBCD

funkce	**CU**	**čítač nahoru závislý na DR, RLO a bloku DFTM**
	CD	**čítač dolů závislý na DR, RLO a bloku DFTM**
	CUBCD	**BCD čítač nahoru závislý na DR, RLO a bloku DFTM**
syntax	**CU**	**count**
	CD	**count**
	CUBCD	**count**
parametr	**„count"**	**název čítače**

Čítačové instrukce jsou určeny k realizaci čítacích funkcí. Instrukcí čítač nahoru **CU** se provádí v případě, že je splněna podmínka pro čítání (nulovací vstup čítače je sepnut), inkrementování buňky, určené adresovou částí instrukce CU. Ke změně stavu čítače dojde pouze při změně stavu bitu, určeného jako vstup čítače ze stavu log.0 do stavu log.1. Zároveň je porovnáván stav datového registru DR, ve kterém je uložena předvolba čítače se stavem adresovaného čítače. V případě rovnosti čítače a DR je nastaven RLO do log. 1, v opačném případě do log.0. Po skončení instrukce CU je obsah adresovaného čítače uložen v datovém registru DR.

Instrukce čítače nahoru CU pracuje s daty v binárním kódu v rozsahu BYTE nebo WORD. Po dosažení nastavené hodnoty se čítače nenulují a není žádné přednastavení čítačů.

Instrukce čítač dolů **CD** pracuje podobně s tím rozdílem, že provádí dekrementaci čítače. Instrukce **CUBCD** pracuje podobně jako instrukce CU, ale v BCD kódu.

Příklad:
Změny vstupního signálu ALFA z nuly do jedničky jsou čítány, pokud blokovací vstup BETA je v jedničce.

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

Příklad:
Vynulování čítače po dočítání na zadanou předvolbu GAMA

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

3.12 Aritmetické a logické instrukce s operandy a DR registrem

Výše uvedené instrukce jsou určeny k provádění aritmetických nebo logických operací, přičemž většinou platí pravidlo, že operace se provádí mezi DR a obsahem paměti nebo konstantou. Adresa paměti se uvádí jako parametr instrukce, vyjma instrukcí `INR, DCR, BIN, BCD, ABS, INV, RR a RL` které jsou bez operandu.

instrukce	**AD** **SU**

funkce	**AD**	**sčítání BYTE,WORD,DWORD,QWORD,REAL nebo konstanty do DR**
	SU	**odčítání BYTE,WORD,DWORD,QWORD,REAL nebo konstanty od DR**
syntax	**AD (SU)**	**val**
	AD (SU)	**[TYPE.] val**
	AD (SU)	**TYPE.(val+a)**
	AD (SU)	**CNTS.immed**
	AD (SU)	**CNTS.ireal**
		TYPE = BYTE. WORD. HIGH. DWRD. REAL. QWRD.
parametr	**„val"**	**název datové proměnné**

Instrukce **AD** sečte obsah DR registru (nebo DR_REAL) s obsahem paměti, na kterou ukazuje operand nebo konstantou. Pokud přímo zadaná hodnota obsahuje desetinnou tečku, považuje se to za zadání čísla v reálném tvaru. Výsledek operace zůstane v DR registru (nebo DR_REAL). Instrukce může pracovat s operandy typu BYTE, WORD, DWORD,QWORD nebo REAL .

Instrukce **SU** odečte od obsahu DR registru (nebo DR_REAL) obsah paměti, na kterou ukazuje operand nebo konstantu. Výsledek operace zůstane v DR registru (nebo DR_REAL).

Instrukce pracují s registrem DR se stejnou šířkou slova, jaká je určena operandem instrukce. Využívají tak z DR registru 8, 16 nebo 32 bitů. Nevyužitá část DR registru zůstává nezměněná. U reálných operací zůstává výsledek jak v reálném registru DR_REAL, tak v celočíselném registru DR64.

instrukce AD (SU)	**typ operace**	**deklarace proměnné**	**popis**
`AD Value`	`DR8 + BYTE`	`Value: DS 1`	přístup podle deklarace proměnné
`AD Value`	`DR16 + WORD`	`Value: DS 2`	přístup podle deklarace proměnné
`AD Value`	`DR32 + DWORD`	`Value: DS 4`	přístup podle deklarace proměnné
`AD Value`	`DR32 + Immed`	`EQUI Value,Immed`	přístup podle deklarace konstanty
`AD BYTE.Value`	`DR8 + BYTE`		přetypování na BYTE
`AD WORD.Value`	`DR16 + WORD`		přetypování na WORD
`AD DWRD.Value`	`DR32 + DWORD`		přetypování na DWORD
`AD QWRD.Value`	`DR64 + QWORD`		přetypování na QWORD
`AD Immed`	`DR32 + Immed`		přímá hodnota
`AD CNST.Immed`	`DR32 + Immed`		přímá hodnota
`AD BYTE.Immed`	`DR8 + Immed`		přímá hodnota + přetypování
`AD WORD.Immed`	`DR16 + Immed`		přímá hodnota + přetypování
`AD DWRD.Immed`	`DR32 + Immed`		přímá hodnota + přetypování
`AD REAL.Value`	`DRr + REAL`	`Value: DS 8`	reálná hodnota proměnné
`AD CNST.Ireal`	`DRr + Ireal`		přímá hodnota reálná

instrukce	**MULB** **DIVB**

funkce **MULB** **násobení DR registru a operandu**
DIVB **celočíselné dělení DR registru s operandem**

syntax **MULB(DIVB) val**
MULB(DIVB) [TYPE.] val
MULB(DIVB) CNST. immed
MULB(DIVB) CNST. ireal
TYPE = BYTE. CNTS. HIGH. WORD. DWRD. REAL.

parametr **„val"** **název datové proměnné**

Instrukce **MULB** vynásobí registr DR (nebo DR_REAL) s obsahem paměti, na kterou ukazuje operand. Jedná se o znaménkové násobení. Výsledek operace zůstane v DR registru (nebo DR_REAL). Užitečná šířka slova v DR registru je po vynásobení dvojnásobná, ale instrukce vždy znaménkově rozšíří DR registr až na 64 bitů (je to pro případ, že po násobení bezprostředně následuje dělení).
Pokud přímo zadaná hodnota obsahuje desetinnou tečku, považuje se to za zadání čísla v reálném tvaru.
U reálných operací zůstává výsledek jak v reálném registru DR_REAL, tak v celočíselném registru DR64.

instrukce MULB	typ operace (výsledek DR64)	deklarace proměnné	popis (přístup)
MULB Value	DR8 * BYTE	Value: DS 1	podle deklarace proměnné
MULB Value	DR16 * WORD	Value: DS 2	podle deklarace proměnné
MULB Value	DR32 * DWORD	Value: DS 4	podle deklarace proměnné
MULB Value	DR32 * Immed	EQUI Value,Immed	podle deklarace konstanty
MULB BYTE.Value	DR8 * BYTE		přetypování
MULB WORD.Value	DR16 * WORD		přetypování
MULB DWRD.Value	DR32 * DWORD		přetypování
MULB Immed	DR32 * Immed		přímá hodnota
MULB CNST.Immed	DR32 * Immed		přímá hodnota
MULB BYTE.Immed	DR8 * Immed		přímá hodnota + přetypování
MULB WORD.Immed	DR16 * Immed		přímá hodnota + přetypování
MULB DWRD.Immed	DR32 * Immed		přímá hodnota + přetypování
MULB REAL.Value	DRr * REAL	Value: DS 8	reálná hodnota proměnné
MULB CNST.Ireal	DRr * Ireal		přímá hodnota reálná

Instrukce **DIVB** vydělí registr DR (nebo DR_REAL) s obsahem paměti, na kterou ukazuje operand. Výsledek operace zůstane v DR registru (nebo DR_REAL). Dělení je celočíselné s ohledem na znaménko.
Pokud je požadováno dělení hodnotou BYTE nebo WORD, instrukce dělení vždy před samotnou operací znaménkově rozšíří DR registr na 64 bitů a operand na 32 bitů. Pokud je požadováno dělení hodnotou DWORD, předpokládá se, že rozšířený DR registr 64 bitů je před operací platný (to znamená, že předcházelo násobení, dělení nebo instrukce LOD). Instrukce vždy znaménkově rozšíří výsledek v DR registru až na 64 bitů.
Pokud přímo zadaná hodnota obsahuje desetinnou tečku, považuje se to za zadání čísla v reálném tvaru.
U reálných operací zůstává výsledek jak v reálném registru DR_REAL, tak v celočíselném registru DR64.

instrukce DIVB	typ operace (výsledek DR64)	deklarace proměnné	popis (přístup)
DIVB Value	DR16 / BYTE	Value: DS 1	podle deklarace proměnné
DIVB Value	DR32 / WORD	Value: DS 2	podle deklarace proměnné
DIVB Value	DR64 / DWORD	Value: DS 4	podle deklarace proměnné
DIVB Value	DR64 / Immed	EQUI Value,Immed	podle deklarace konstanty
DIVB BYTE.Value	DR16 / BYTE		přetypování
DIVB WORD.Value	DR32 / WORD		přetypování
DIVB DWRD.Value	DR64 / DWORD		přetypování
DIVB Immed	DR64 / Immed		přímá hodnota
DIVB CNST.Immed	DR64 / Immed		přímá hodnota
DIVB BYTE.Immed	DR16 / Immed		přímá hodnota + přetypování
DIVB WORD.Immed	DR32 / Immed		přímá hodnota + přetypování
DIVB DWRD.Immed	DR64 / Immed		přímá hodnota + přetypování
DIVB REAL.Value	DRr / REAL	Value: DS 8	reálná hodnota proměnné
DIVB CNST.Ireal	DRr / Ireal		přímá hodnota reálná

Příklad:
Hodnotu z buňky ALFA (BYTE) zmenšenou o 23h vynásobte hodnotou z buňky BETA (BYTE) a výsledek zapište do buňky GAMA (WORD).

```
LOD         ALFA            ;načte ALFA do DR
SU          CNST.23H        ;odečte 23h
MULB        BETA            ;vynásobení s BETA (výsledek 16 bitů)
STO         GAMA            ;zapíše do GAMA
```

Příklad:
Podíl 32 bitové buňky DELENEC (DWORD) a 16 bitové buňky DELITEL (WORD) zapište do buňky PODIL (WORD).

```
LOD         DELENEC         ;načte dělenec 32 bitů do DR
DIVB        DELITEL         ;dělení DR32 s dělitelem 16 bitů
STO         PODIL           ;výsledek v PODIL 16 bitů
```

Příklad:
Vynásobte DR registr (32 bitů) zlomkem, pro CITATEL (DWORD) a DELITEL (DWORD) zapište do buňky VYSLEDEK (DWORD).

```
MULB        CITATEL     ;násobení DR32*CITATEL, výsledek 64 bitů
DIVB        DELITEL     ;dělení DR64 s dělitelem 32 bitů
STO         VYSLEDEK    ;zápis výsledku 32 bitů
```

Příklad:
Reálné operace:

```
LOD         REAL.rBun   ;načte reálnou proměnnou rBun
MULB        REAL.rBun2  ;DR_REAL <- rBun * rBun2
DIVB        CNST.4.8    ;DR_REAL <- rBun * rBun2 / 4.8
STO         REAL.rBun3
```

instrukce	**ORB** **ANDB** **XORB**

funkce	**ORB**	**logický OR po bitech mezi DR registrem a pamětí,konst.**
	ANDB	**logický AND po bitech mezi DR registrem a pamětí, konst.**
	XORB	**logický XOR po bitech mezi DR registrem a pamětí, konst.**

syntax	**ORB (ANDB,XORB)**	**val**
	ORB (ANDB,XORB)	**[TYPE.] val**
	ORB (ANDB,XORB)	**TYPE. (val+n)**
	ORB (ANDB,XORB)	**CNTS. immed**
		TYPE = BYTE. WORD. HIGH. DWRD.

parametr	**„val"**	**název datové proměnné**

Instrukce **ORB,ANDB** a **XORB** jsou určeny k provádění logických operací, přičemž platí pravidlo, že operace se provádí mezi DR a obsahem paměti nebo konstantou. Adresa paměti se uvádí jako parametr instrukce.

Instrukce provedou logický OR, AND nebo XOR po jednotlivých bitech mezi DR registrem a pamětí nebo konstantou.

Přístupy a způsoby přetypování jsou stejné jako u instrukcí AD a SU.

3.13 Bezoperandové instrukce pro práci s DR registrem

Některé bezoperandové instrukce pro operace s datovým DR registrem mohou pracovat s rozšířeným 32 nebo 64 bitovým DR registrem (DWORD, QWORD) nebo reálným registrem DR_REAL. Jedná se o instrukce **INR, DCR, INV, ABS, RR, RL, CONDR, CONRD**. V tomto případě je nutné modifikovat bezoperandové instrukce pomocí parametru **DWRD** nebo **QWRD**.

instrukce	**INR** **DCR** **INRBCD**

funkce	**INR**	**inkrement DR registru**
	DCR	**dekrement DR registru**
	INRBCD	**inkrement DR registru v BCD tvaru**
syntax	**INR (DCR)**	
	INR (DCR)	**[DWRD]**
	INR (DCR)	**[QWRD]**
	INR (DCR)	**[REAL]**
	INRBCD	

Instrukce **INR** bez operandu zvětší registr DR o jedničku. Při překročení rozsahu začíná od nuly.
Instrukce **DCR** bez operandu zmenší registr DR o jedničku.
Instrukce **INRBCD** zvětší registr DR o jedničku v BCD kódu. Pracuje v rozsahu 0.. 9999.

Instrukce INR a DCR s parametrem „DWRD" zvětšují nebo zmenšují rozšířený 32 bitový DR registr.
Instrukce INR a DCR s parametrem „QWRD" zvětšují nebo zmenšují rozšířený 64 bitový DR registr.

Instrukce INR a DCR s parametrem „REAL" zvětšují nebo zmenšují reálný registr DR_REAL. U reálných operací zůstává výsledek jak v reálném registru DR_REAL, tak v celočíselném registru DR64.

instrukce	**BIN** **BCD**

funkce	**BIN**	**převod BCD → BIN kódu**
	BCD	**převod BIN → BCD kódu**
syntax	**BIN**	
	BCD	
	BIN	**[DWRD]**
	BCD	**[DWRD]**

Instrukce **BIN** bez operandu převede číslo ve tvaru BCD, uložené v registru DR (maximálně 16 bitů) na binární číslo. Výsledek zůstane v DR registru.
Instrukce **BIN** s parametrem **DWRD** převede číslo v rozšířeném 32 bitovém DR registru. Záporné BCD číslo před převodem má nastavenou na jedničku bit s váhou 31.

Instrukce **BCD** bez operandu převede číslo v binárním tvaru, uložené v registru DR na BCD číslo. Výsledek zůstane v DR registru. Hodnota v DR registru před převodem nesmí být větší než 9999d = 270Fh.
Instrukce **BCD** s parametrem **DWRD** převede číslo v rozšířeném 32 bitovém DR registru. Hodnota v DR registru před převodem nesmí být větší než 99999999d = 5F5E0FFh.

instrukce	RL RR

funkce	RL	logický posuv DR registru vlevo
	RR	logický posuv registru vpravo
syntax	RL (RR)	n
	RL (RR)	[TYPE.] n
	RL (RR)	[TYPE.] n [,DWRD]
	RL (RR)	[TYPE.] n [,QWRD]
		TYPE = BYTE. HIGH. CNST.
parametr	„n“	počet rotací

Instrukce **RL** provede logický posuv DR vlevo o "n" bitů. Operand je konstanta nebo hodnota v buňce (maximálně o velikosti 8 bitů) udávající počet rotací.
Instrukce **RR** provede logický posuv DR vpravo o "n" bitů. Operand je konstanta nebo hodnota v buňce (maximálně o velikosti 8 bitů) udávající počet rotací.
Instrukce mohou mít druhý parametr „DWRD“ nebo „QWRD“, který udává, že se provede posuv rozšířeného 32 nebo 64 bitového DR registru.

instrukce	INV ABS

funkce	INV	negace DR registru (dvojkový doplněk)
	ABS	absolutní hodnota DR registru
syntax	INV (ABS)	
	INV (ABS)	[DWRD]
	INV (ABS)	[QWRD]
	INV (ABS)	[REAL]

Instrukce **INV** provede negaci DR registru, to je dvojkový doplněk.
Instrukce **ABS** provede absolutní hodnotu DR registru.
Instrukce mohou mít nepovinný parametr „DWRD“ nebo „QWRD“, který modifikuje instrukce tak, že negace nebo absolutní hodnota se provede nad rozšířeným 32 nebo 64 bitovým DR registrem.

Instrukce mohou mít nepovinný parametr „REAL“, který modifikuje instrukce tak, že negace nebo absolutní hodnota se provede v reálném registru DR_REAL. U reálných operací zůstává výsledek jak v reálném registru DR_REAL, tak v celočíselném registru DR64.

Příklad:
Různé operace:

```
LOD         ALFA             ;načtení ALFA (typ podle deklarace)
INR                          ;inkrementace DR (16 bitů)
ABS                          ;absolutní hodnota DR (16 bitů)
RL          POSUN            ;posun DR doleva o hodnotu v POSUN
STO         VYSLEDEK1        ;zápis do VYSLEDEK1
INV                          ;dvojkový doplněk DR (16 bitů)
RR          CNST.3           ;posun DR doprava o 3 bity
STO         VYSLEDEK2        ;zápis do VYSLEDEK2

LOD         DWRD.BETA        ;načtení 32 bitů z BETA do DR
ABS         DWRD             ;absolutní hodnota DR 32 bitů
INR         DWRD             ;inkrementace DR 32 bitů
RR          CNST.18,DWRD     ;posun o 18 bitů vpravo registru DR32
STO         VYSLEDEK3        ;zápis do VYSLEDEK3 32 bitů
```

3.14 Logické instrukce s operandy a DR registrem

Skupina logických instrukcí provede porovnání DR registru (nebo DR_REAL) s obsahem paměti nebo konstantou a na základě výsledku porovnání se nastaví bitový RLO registr.

instrukce	**EQ** **EQ1** **LT** **GT** **LE** **GE**

```
funkce      EQ    porovnávání DR registru s operandem
            EQ1   podmíněné porovnávání DR s operandem, když RLO = 1
            LT    DR je menší než operand
            GT    DR je větší než operand
            LE    DR je menší nebo rovno než operand
            GE    DR je větší nebo rovno než operand

syntax      EQ (EQ1,LT,GT,LE,GE)    val
            EQ (EQ1,LT,GT,LE,GE)    [TYPE.] val
            EQ (EQ1,LT,GT,LE,GE)    TYPE. (val+n)
            EQ (EQ1,LT,GT,LE,GE)    CNTS. immed
            EQ (EQ1,LT,GT,LE,GE)    CNTS. ireal
                                TYPE = BYTE. HIGH. WORD. DWRD. QWRD. REAL.

parametr    „val" název datové proměnné
```

Instrukce **EQ** je logická a provádí porovnání DR registru (nebo DR_REAL) s obsahem paměti, na kterou ukazuje operand nebo konstantou. Je-li DR shodný s obsahem paměti nebo s konstantou, je nastaven RLO registr do jedničky, v opačném případě je RLO vynulován. Operandy mohou být typu BYTE, WORD, DWORD, QWORD, REAL nebo konstanta .

Instrukce **LT** resp. **LE** jsou logické a provádí porovnání DR registru (nebo DR_REAL) s obsahem paměti, na kterou ukazuje operand nebo konstantou. Je-li DR menší resp. menší nebo roven než obsah paměti nebo konstanty, je nastaven RLO registr do jedničky, v opačném případě je RLO vynulován.

Instrukce **GT** resp. **GE** jsou logické a provádí porovnání DR registru (nebo DR_REAL) s obsahem paměti, na kterou ukazuje operand nebo konstantou. Je-li DR větší resp. větší nebo roven než obsah paměti nebo konstanty, je nastaven RLO registr do jedničky, v opačném případě je RLO vynulován.
Instrukce **EQ1** je zavedena pro přiblížení se ke kompatibilitě s automatem NS915. Porovnání se provede jenom tehdy, když je v registru RLO hodnota **1**, jinak zůstane v RLO hodnota 0. Instrukci `EQ1` možno použít na porovnání větších paměťových oblastí, protože instrukce EQ1 je možné zřetězit. Instrukce `EQ1` se dá nahradit pomocí dvou instrukcí a jednoho návěští:

```
            JL0   OBSKOK
            EQ    BUNKA       ekvivalent:       EQ1   BUNKA
OBSKOK:
```

Předefinování typu je popsáno v kapitole "Způsoby předefinování typu u datových proměnných".

instrukce EQ (LT,GT,LE,GE)	typ operace	deklarace proměnné	popis
EQ Value	DR8 ~ BYTE	Value: DS 1	přístup podle deklarace proměnné
EQ Value	DR16 ~ WORD	Value: DS 2	přístup podle deklarace proměnné
EQ Value	DR32 ~ DWORD	Value: DS 4	přístup podle deklarace proměnné
EQ Value	DR32 ~ Immed	EQUI Value,Immed	přístup podle deklarace konstanty
EQ BYTE.Value	DR8 ~ BYTE		přetypování na BYTE
EQ WORD.Value	DR16 ~ WORD		přetypování na WORD
EQ DWRD.Value	DR32 ~ DWORD		přetypování na DWORD
EQ QWRD.Value	DR64 ~ QWORD		přetypování na QWORD
EQ Immed	DR32 ~ Immed		přímá hodnota
EQ CNST.Immed	DR32 ~ Immed		přímá hodnota
EQ BYTE.Immed	DR8 ~ Immed		přímá hodnota + přetypování
EQ WORD.Immed	DR16 ~ Immed		přímá hodnota + přetypování
EQ DWRD.Immed	DR32 ~ Immed		přímá hodnota + přetypování
EQ REAL.Value	DRr ~ REAL	Value: DS 8	reálná hodnota proměnné
EQ CNST.Ireal	DRr ~ Ireal		přímá hodnota reálná

Příklad:
Skok na návěští MENSI, když buňka BUNKA1 je menší něž buňka BUNKA2

```
LOD         BUNKA1          ;načte BUNKA1 do DR
LT          BUNKA2          ;když je DR < BUNKA2 tak RLO=1, jinak 0
JL1         MENSI           ;skok na menší, když je RLO=1
```

Příklad:
Porovnejte hodnotu z buňky NASTAVENI s buňkou BUNKA, když výraz ALFA*NOT(BETA) je roven 1. Výsledek zapište do GAMA:

```
LDR         ALFA            ;načte bit ALFA do RLO
LA          -BETA           ;logický součin RLO s negací BETA
LOD         NASTAVENI       ;načte buňku NASTAVENI do DR
EQ1         BUNKA           ;podmíněné porovnání když RLO=1
WR          GAMA            ;zápis RLO do GAMA
```

Příklad:
Porovnejte mezi sebou oblasti paměti PAM1 a PAM2 o velikosti 8 BYTE.

```
LOD         DWRD.PAM1       ;načte 4 BYTE z PAM1 do DR 32 bitů
EQ          DWRD.PAM2       ;porovnání DR 32 bitů z 4 BYTE PAM2
LOD         DWRD.(PAM1+4)   ;načte další 4 BYTE s PAM1 do DR 32 bitů
EQ1         DWRD.(PAM2+4)   ;další porovnání se provede jen když
                            ;při prvním byla rovnost RLO=1
JL1         ROVNO           ;odskok při rovnosti
```

Příklad:
Nastavte bit AKCE do hodnoty 1, když buňka MATTL je rovna kódu ´W´

```
LOD         MATTL           ;načte buňku MATTL do DR
EQ          CNST.'W'        ;porovnání DR s konst. a nastavení RLO
FL1         1,AKCE          ;když RLO=1 nastaví bit AKCE na 1
```

3.15 Konverze a přesuny registrů a paměti

instrukce	CONDR CONRD

funkce	**CONDR**	**konverze DR → RLO**
	CONRD	**konverze RLO → DR**
syntax	**CONDR (CONRD)**	
	CONDR	**0,1,..,31**
	CONDR (CONRD)	**[DWRD]**
parametr	**„0,1,..,31"**	**číslo bitu pro přesun DR → RLO**

Instrukce **CONDR** provede konverzi registru DR do bitového registru RLO. Pokud je registr DR nulový, naplní se RLO registr na hodnotu 0. Pokud je registr DR nenulový, naplní se RLO registr na hodnotu 1. Instrukce CONDR neovlivní obsah DR registru.
Pokud instrukce CONDR obsahuje operand, kterým je číslo 0 - 31, instrukce přesune z datového registru DR do bitového registru RLO jen odpovídající bit.

Instrukce **CONRD** provede konverzi bitového registru RLO do datového DR registru. Pokud je registr RLO nulový, naplní se DR registr na hodnotu 0h. Pokud je registr RLO roven hodnotě 1, naplní se DR registr na hodnotu FFFFh. Instrukce CONRD neovlivní obsah RLO registru. Instrukce CONRD je ***koncová instrukce*** pro logické výrazy.
Při použití parametru **DWRD** se provede konverze s rozšířeným 32 bitovým DR registrem.

instrukce	MV

funkce	**MV**	**přesun paměti**
syntax	**MV**	**src, dest, num**
1.parametr	**„src"**	**adresa zdroje pro přesun dat**
2.parametr	**„dest"**	**adresa cíle, kam přesunout**
3.parametr	**„num"**	**počet přesouvaných bajtů**

Instrukce **MV** slouží pro přesun oblasti paměti. Parametr "src" je adresa zdroje - odkud se bude přesouvat, parametr "dest" je adresa cíle - kam se bude paměťová oblast přesouvat a parametr "num" je počet přesouvaných bajtů paměti.
Instrukce může mít parametry „src" a „dest" zadány i s offsetem a mohou být použity i pro plnění zálohované paměti PLC.

Příklad:

```
MV    PLC_MEM_BACKUP+100,BUN5,8      ;naplní 8 bajtů z BUN5 do zálohované paměti
```

instrukce	REAL	
funkce	**REAL**	**konverze DR -> DR_REAL**
syntax	**REAL**	
	REAL	**[DWRD]**
	REAL	**[QWRD]**

Instrukce **REAL** provede konverzi registru DR do reálného registru DR_REAL. Po operaci zůstává výsledek jak v reálném registru DR_REAL, tak v celočíselném registru DR64.
Pokud je požadován převod hodnoty BYTE, WORD nebo DWORD instrukce vždy před převodem znaménkově rozšíří DR registr na 64 bitů (DR64).
Pokud instrukce má parametr „QWRD" a je tedy požadován převod z rozšířeného 64 bitového DR registru, předpokládá se že rozšířený registr DR64 je před operací platný (to znamená, že předcházelo násobení, dělení nebo instrukce LOD).

instrukce	CLEAR	
funkce	**CLEAR**	**nulování paměti**
syntax	**CLEAR**	**begin, end**
	CLEAR	**GLOBAL, ALL**
	CLEAR	**INTERNAL, ALL**
1.parametr	**„begin"**	**adresa začátku nulované oblasti**
2.parametr	**„end"**	**adresa konce nulované oblasti**

Instrukce **CLEAR** slouží pro nulování paměti. Parametr "begin" je adresa začátku nulované oblasti a parametr "end" je adresa konce nulované oblasti. Instrukce vynuluje jak paměť pro globální proměnné, tak paměť pro lokální proměnné. Příslušná oblast je určena podle výskytu adres proměnných v parametru instrukce.

Instrukce "**CLEAR GLOBAL, ALL**" vynuluje všechna globální data včetně inicializačních proměnných mechanizmů a časovačů. Instrukce se používá například v modulu MODULE_CLEAR, aby nulování (start a stop) PLC programu bylo ekvivalentní se stavem, který proběhne jen při zapnutí stroje.

Instrukce "**CLEAR INTERNAL, ALL**" vynuluje všechna lokální data definovaná návrhářem PLC programu včetně "automatických" proměnných definovaných v rozvoji instrukcí jazyka TECHNOL. Instrukce se používá například v modulu MODULE_CLEAR.

3.16 Reálné operace

Dále uvedeme některé předpoklady pro práci s reálnými čísly v PLC programu. Reálná čísla se do PLC programu mohou dostat například z povelového bloku, z PLC tabulek nebo ze sdílené oblasti PLC paměti.

Instrukce, které mohou pracovat s reálnými čísly jsou:
`LOD, STO, STO1, AD, SU, MULB, DIVB, INR, DCR, INV, ABS, EQ, EQ1, LT, GT, LE, GE, REAL`

Instrukce pro práci s reálnými čísly musí mít povinně prefix „`REAL`" nebo „`CNST`" s číslem s desetinnou tečkou. Pro všechny instrukce platí, že po operaci zůstává výsledek jak v reálném registru DR_REAL, tak v celočíselném registru DR64. Proto je možno kdykoli pokračovat standardními celočíselnými operacemi. Nelze ale kombinovat celočíselné a reálné operace.

Pro zadání přímých reálných hodnot lze použít prefix „`CNST`". Pokud přímo zadaná hodnota obsahuje desetinnou tečku, považuje se to za zadání čísla v reálném tvaru. Hodnota může obsahovat znaménko.

Příklady:

```
rBUN1:      DS    8          ;pro reálnou hodnotu
rBUNDef:    DS    8
dwBUN5:     DS    4

      LOD   cnst.2.8         ;načte reálnou hodnotu 2.8
      STO   real.rBUN1       ;zapíše do reálné buňky rBUN1
      MULB  cnst.-4.8        ;DR_REAL <- rBUN1 * (-4.8)
      INR   real             ;inkrementuje DR_REAL
      DIVB  real.rBUN2       ;DR_REAL <- DR_REAL / rBUN2
      STO   real.rBUN3       ;zapíše do rBUN3

      LOD   cnst.2           ;načte celočíselnou hodnotu 2
      REAL  qwrd             ;převede na reálné číslo a nastaví DR_REAL
      AD    real.rBUN4       ;připočte rBUN4
      STO   real.rBUN5       ;zapíše reálnou hodnotu do rBUN5
      STO   dwBUN5           ;zapíše DWORD celočíselně do dwBUN5

      LOD   real.rBUN1       ;načte reálnou buňku rBUN1
      EQ    cnst.1.5         ;porovná s reálnou hodnotou 1.5
      WR    Rovnost          ;zapíše bit

      LOD_REAL_BLOCK H       ;načte hodnotu H z povelového bloku
      MULB  cnst.1000.0      ;vynásobí H * 1000 reálně
      STO   real.rBUN5       ;zapíše H*1000 reálně
      STO   dwBUN5           ;zapíše H*1000 celočíselně

      LOD   cnst.0.2         ;reálná defaultní hodnota pro konfiguraci
      STO   real.rBunDef
      CNF_GET_REAL rBUN1,'RealParam1',rBUNDef  ;načte PLC konfiguraci
```

3.17 Procedury

Ve všech souborech PLC programu mohou být definovány procedury (podprogramy) a také ve všech souborech mohou být libovolná volání procedur definovaných i v jiných souborech.

instrukce	**PROC_BEGIN** **PROC_END** **PROC_CALL**

funkce	**PROC_BEGIN**	**začátek definice procedury**
	PROC_END	**konec definice procedury**
	PROC_CALL	**volání procedury**
syntax	**PROC_BEGIN**	**name**
	PROC_END	**name**
	PROC_CALL	**name**
parametr	**„name"**	**jméno procedury**

Definice procedury se provede pomocí příkazů **PROC_BEGIN** a **PROC_END**, které mají jako parametr název procedury. Umístění procedury při její definici může být na libovolném místě v modulu MODULE_MAIN. Sled vykonávání instrukcí přeskočí definiční oblast procedury. Procedura se vykoná jen pomocí instrukce **PROC_CALL** s příslušným parametrem, který je názvem procedury. Po vykonání procedury se sled vykonávání instrukcí vrátí za místo volání procedury. Procedury mohou být volány různě ze všech souborů PLC.

Událostní procedury:
V jazyku TECHNOL jsou rezervovány některé názvy procedur, které slouží pro konkrétní účel a jsou automaticky spuštěny při dané události. Jedná se o tyto názvy procedur:

_ON_ESET	Procedura se zavolá automaticky při vzniku PLC chyby. Umístění procedury může být v libovolném souboru s PLC programem. Překladač TECHNOL zkoumá existenci takovéto procedury a v případě, že existuje, spustí ji automaticky z rozvoje instrukce ESET. (viz kapitolu „Chybová hlášení, varování a informační hlášení z PLC programu".
_ON_CMD	Procedura se zavolá automaticky při stisku tlačítka nebo softwarového menu, které mají v konfiguraci nastaven "PLCCommand". Procedura v PLC programu je nepovinná. Umístění procedury může být v libovolném souboru. Překladač TECHNOL zkoumá existenci takovéto procedury a v případě, že existuje, spustí ji automaticky při stisku tlačítka. Vstupní parametry procedury jsou: RLO = 1 ..... tlačítko stisknuto RLO = 0 ..... tlačítko puštěno DR (DWORD) ... kód z konfigurace "PLCCommand" kódem je PLC konstanta načtená pomocí instrukce CONST_GET (viz. „Konfigurace pro PLC")
_ON_EVENT	Procedura se zavolá automaticky při některých vybraných událostech systému. Procedura v PLC programu je nepovinná. Umístění procedury může být v libovolném souboru. Překladač TECHNOL zkoumá existenci takovéto procedury a v případě, že existuje, spustí ji automaticky při vybraných událostech. Typ události se předává v DR registru (DWORD):

	Hodnoty předávané v DR registru: PLCEVENT_STOP_REQ ..... požadavek na STOP PLCEVENT_STOP_EMERGENCY_REQ .... požadavek na Nouzový STOP PLCEVENT_STOP ..... STOP vykonán PLCEVENT_START ..... příkaz START PLCEVENT_START_REQ ..... požadavek na START

3.18 Práce s textovými řetězci

Pro práci s textovými řetězci je potřeba mít možnost definovat řetězec, skládat a přesouvat řetěze a konvertovat čísla na řetězce.

instrukce	**STR**

```
funkce      STR   definice textového řetězce

syntax      TX1:  STR   n
            TX1:  STR   n [,'ABCDabcd..']
            TX1:  STR   n [,PLC_MEM_BACKUP + xy]
            TX1:  STR   n [,val + xy]

1.parametr  „n"         maximální počet znaků v řetězci
2.parametr  „text2"     přímé nebo nepřímé zadání řetězce
```

Instrukce **STR** definuje textový řetězec. Instrukce musí mít povinně návěští, které budě dále sloužit pro identifikaci definovaného řetězce. Návěští bude známo a přístupno ve všech souborech PLC programu. Instrukce STR se může umístit v libovolném modulu PLC programu. (Překladač ve skutečnosti definuje také obecný pointer, který je nasměrovaný na příslušný řetězec.)

1. parametr „n"

První parametr instrukce je povinný a vyjadřuje maximální počet znaků v řetězci. (Překladač ve skutečnosti vymezí o jeden bajt víc pro koncový znak řetězce.)

Příklad:

```
TX_POKUS:   STR   50
```

2. parametr „text2"

Druhý parametr instrukce je nepovinný a může obsahovat:

Druhý parametr je přímé zadání textového řetězce. Text musí být ohraničen apostrofy a měl by mít maximálně tolik znaků, kolik vymezuje 1. parametr instrukce. (Překladač zabezpečí předplnění zadaného řetězce při každém přenosu nového PLC programu)

Příklad:

```
TX_ZPR:     STR   10,'Zapnuto'
```

Druhý parametr je odkaz do pamětí PLC_MEM_BACKUP. Paměťová oblast se používá jako zálohovaná paměť.

Příklady:

```
TX_SCR1:    STR   15, PLC_MEM_BACKUP+124

EQUI        TEXT_ZAP,124                ;symbolický offset
TX_SCR2:    STR   15, PLC_MEM_BACKUP+TEXT_ZAP
```

Druhý parametr je odkaz na libovolnou paměťovou proměnou, kromě definice textu, například definovaných pomocí instrukcí DFM a DS. Potom instrukce STR musí být umístěna v tom samém souboru.

Příklad:

```
PAM25:      DS    20          ;pole v datové části PLC programu

TX_BUN25:   STR   15,PAM25
```

instrukce	**STRCPY** **STRADD**

```
funkce   STRCPY      kopírování textových řetězců
         STRADD      spojení textových řetězců

syntax   STRCPY (STRADD)   text1, text2
         STRCPY (STRADD)   text1, 'ABCDabcd..'
         STRCPY            text1, text2 [,num]
         STRCPY            text1+xx, text2+yy
         STRCPY            text1+xx, text2+yy, num
         STRCPY            text1+BX, text2+yy, num

1.parametr „text1"    cílový textový řetězec
2.parametr „text2"    zdrojový textový řetězec
```

Instrukce **STRCPY** překopíruje textový řetězec na který ukazuje druhý operand, do textového řetězce na který ukazuje první operand. Když je druhý operand přímé zadání textového řetězce, překopíruje jej do řetězce podle prvního operandu. Oba řetězce jsou definovány pomocí instrukce STR. Řetězec podle prvního operandu musí mít rezervován dostatek prostoru.

Počet kopírovaných znaků může být určen:

Pokud v instrukci není zadán 3.parametr, který určuje počet znaků pro překopírování:		
	1.	Kopírování se ukončí výskytem koncového znaku ve zdrojovém řetězci (text2). Koncový znak je binární 0 a překopíruje se jako poslední. (cílový řetězec může být zkrácen)
	2.	Kopírování se ukončí, když je dosažena velkost cílového řetězce (text1). Zdrojový řetězec by se do cílového řetězce nevešel.

Pokud je v instrukci zadán 3. parametr, který určuje počet znaků:		
	1.	Kopírování se ukončí výskytem koncového znaku ve zdrojovém řetězci (text2). Koncový znak je binární 0 a překopíruje se jako poslední. (cílový řetězec může být zkrácen)
	2	Kopírování se ukončí, když je dosažena velkost cílového řetězce (text1). Zdrojový řetězec by se do cílového řetězce nevešel. (cílový řetězec zůstane v původní délce)
	3.	Kopírování se ukončí dosažením zadaného počtu znaků. (cílový řetězec nebude zkrácen)

Pokud potřebujeme kopírovat řetězec do jiného řetězce na konkrétní pozici, můžeme použít instrukci s offsetem (text1+10)

Když řetězce obsahují libovolná binární čísla (včetně binární 0), tak musíme místo instrukce STRCPY použít instrukci MEMCPY (viz dále).

Instrukce **STRADD** připojí textový řetězec na který ukazuje druhý operand, do textového řetězce na který ukazuje první operand. Když je druhý operand přímé zadání textového řetězce, připojí jej do řetězce podle prvního operandu.

Příklady:

```
TEXT1:      STR   50, 'prvni text '
TEXT2:      STR   20, 'druhy text '
TEXT3:      STR   50, STCH_OUT_FIELD + 100
TEXT4:      STR   3 , 'ABC'

      STRADD      TEXT1, TEST2      ;spojení textů TEXT1 <- TEXT1+TEXT2

      STRCPY      TEXT3,'OBR. '     ;naplní TEXT3 v STCH_OUT_FIELD+100
      STRADD      TEXT3,TEXT2       ;připojí k TEXT3 ještě TEXT2

      STRCPY      TEXT2+6, TEXT4    ;v řetězci TEXT2 přepíše ,text‘ na ,ABC‘
                                    ;řetězec bude zkrácen

      STRCPY      TEXT2+6, TEXT4,2  ;v řetězci TEXT2 přepíše ,te‘ na ,AB‘
                                    ;řetězec nebude zkrácen

      STRCPY      TEXT3+BX, TEXT4   ;v řetězci TEXT3 na offsetu BX se
                                    ;se přepíše 'ABC'
```

instrukce	**BINSTR** **BCDSTR**

funkce **BINSTR** **převod binárního čísla na řetězec**
BCDSTR **převod BCD hodnoty na řetězec**

syntax **BINSTR (BCDSTR) text1**
BINSTR (BCDSTR) text1 [,DWRD]

parametr **„text1"** **cílový textový řetězec**

Instrukce **BINSTR** převede binárně hodnotu z datového registra DR na řetězec na který ukazuje první parametr instrukce. První parametr instrukce ukazuje na řetězec, který musí mít velkost definovanou instrukcí STR minimálně 1 a maximálně 10 znaků (bajtů). Instrukce podle velkosti řetězce přizpůsobí převod. Pokud je číslo v DR registru menší, doplní se řetězec na prvních místech (první zleva) znaky nul.

Instrukce **BCDSTR** je podobná jako instrukce BINSTR, ale hodnotu v DR registru převádí na řetězec jako hodnotu v BCD kódu..

Instrukce mohou mít druhý parametr „DWRD", který udává, že se provede převod z rozšířeného 32 bitového DR registru.

Příklad:

```
TEXT1: STR        4                ;pro převod 4 cifry
TEXT2: STR        30

      LOD         BUN1             ;wordová buňka
      BINSTR      TEXT1            ;převede word na řetězec TEXT1
      STRCPY      TEXT2,'POCET-'
      STRADD      TEXT2,TEXT1      ;připojí převedený řetězec
```

instrukce	**STRCMP**

funkce **STRCMP** **porovnání textových řetězců**

syntax **STRCMP** **text1, text2**
STRCMP **text1, 'ABCDabcd..', num**
STRCMP **text1, text2 [,num]**
STRCMP **text1+xx, text2+yy**
STRCMP **text1+xx, text2+yy, num**

1.parametr **„text1"** **cílový textový řetězec**
2.parametr **„text2"** **zdrojový textový řetězec**

Instrukce **STRCMP** porovnává textové řetězce na které ukazuje první a druhý operand. Druhý operand může být přímé zadání textového řetězce. Řetězce jsou definovány pomocí instrukce STR. Pokud jsou textové řetězce stejné, instrukce nastaví registr RLO na hodnotu 1, jinak bude RLO mít hodnotu 0. Pokud je zadán 3. parametr „num", tak se z obou řetězců porovná jen zadaný počet znaků (pokud porovnávání nebude dřív ukončeno nerovností nebo výskytem koncového znaku 0).

Pokud instrukce neobsahuje 3. parametr o počtu znaků, bude se porovnávat počet znaků podle velkosti řetězce, na který ukazuje druhý parametr instrukce včetně koncového znaku.

Příklad:

```
TEXT1: STR        8,'abcdefgh'
TEXT2: STR        8,'abcde'
TEXT4: STR        3,'cde'


      STRCMP      TEXT1+2, TEXT4,3  ;porovnání řetězců - shoda
      JL1         OK                ;je nastaveno RLO=1

      STRCMP      TEXT1+2, TEXT4    ;porovnání řetězců - neshoda
      JL1         OK                ;je nastaveno RLO=0

      STRCMP      TEXT2+2, TEXT4    ;porovnání řetězců - shoda
      JL1         OK                ;je nastaveno RLO=1

      STRCMP      TEXT1+2, 'cde',3  ;porovnání řetězců - shoda
      JL1         OK                ;je nastaveno RLO=1
```

instrukce MEMCPY

funkce **MEMCPY** **kopírování binárních řetězců**

syntax **MEMCPY** **text1, text2 [,num]**
MEMCPY **text1, text2**
MEMCPY **text1+xx, text2+yy**
MEMCPY **text1+xx, text2+yy, num**

1.parametr „**text1**" **cílový binární řetězec**
2.parametr „**text2**" **zdrojový binární řetězec**

Instrukce **MEMCPY** překopíruje binární řetězec na který ukazuje druhý operand, do řetězce na který ukazuje první operand. Oba řetězce jsou definovány pomocí instrukce STR. Řetězec podle prvního operandu musí mít rezervován dostatek prostoru.

Instrukce je podobná instrukci STRCPY s rozdílem, že řetězce nemají definovaný koncový znak 0. Kopírování se proto neukončí výskytem koncového znaku ve zdrojovém řetězci (text2). V instrukci MEMCPY se proto vždy doporučuje definovat počet kopírovaných znaků pomocí 3. parametru. Pokud v instrukci není 3.parametr zadán, kopírování se ukončí, když je dosažena velkost cílového řetězce (text1).

instrukce MEMCMP

funkce **MEMCMP** **porovnání binárních řetězců**

```
syntax   MEMCMP          text1, text2 [,num]
         MEMCMP          text1, text2
         MEMCMP          text1+xx, text2+yy
         MEMCMP          text1+xx, text2+yy, num
```

Instrukce **MEMCMP** porovnává binární řetězce na které ukazuje první a druhý operand. Řetězce jsou definovány pomocí instrukce STR. Pokud jsou řetězce stejné, instrukce nastaví registr RLO na hodnotu 1, jinak bude RLO mít hodnotu 0.

Instrukce je podobná instrukci STRCMP s rozdílem, že řetězce nemají definovaný koncový znak 0. Porovnávání se proto neukončí výskytem koncového znaku v řetězci. V instrukci MEMCMP se vždy doporučuje definovat počet porovnávaných znaků pomocí 3. parametru.

Pokud je zadán 3. parametr „num", tak se z obou řetězců porovná jen zadaný počet znaků. Pokud instrukce neobsahuje 3. parametr o počtu znaků, bude se porovnávat počet znaků podle velkosti řetězce, na který ukazuje druhý parametr instrukce.