Programování PLC podle normy IEC 61 131-3 v prostředí Mosaic

desáté vydání
listopad 2007
změny vyhrazeny

Historie změn

Datum	Vydání	Popis změn
Srpen 2004	1	První verze
Říjen 2004	2	Doplněn popis standardní knihovny
Leden 2005	3	Provedeny úpravy pro Mosaic Help
Únor 2005	4	Oprava příkladu 3.6.2.5 – doplněno slovo „DO“
Duben 2005	5	Doplněna tab.3.3 Speciální znaky v řetězcích Oprava rozsahu typu SINT v kap.3.2.1 v Tab.3.5 Tab.3.18 doplněna o vzory volání funkcí nad řetězcem znaků Doplněna kap.3.7.2 Knihovna funkcí nad řetězcem znaků
Listopad 2005	6	Doplněn popis knihovny konverzních funkcí Upravena Tab.3.20 Standardní funkce s typy datum a čas Doplněn popis datového typu PTR_TO
Únor 2006	7	Rozšířen popis datových typů a proměnných Přidán popis knihovny aritmetických funkcí
Březen 2006	8	Doplněny základní myšlenky programování podle normy Doplněn popis jazyka IL Popis knihoven přesunut do samostatného dokumentu TXV 003 22
Listopad 2006	9	Přidán popis direktiv
Listopad 2007	10	Přidán popis grafických jazyků LD a FBD

1 ÚVOD

1.1 Norma IEC 61 131

Norma IEC 61 131 pro programovatelné řídící systémy má pět základních částí a představuje souhrn požadavků na moderní řídící systémy. Je nezávislá na konkrétní organizaci či firmě a má širokou mezinárodní podporu. Jednotlivé části normy jsou věnovány jak technickému tak programovénu vybavení těchto systémů.

V ČR byly přijaty jednotlivé části této normy pod následujícími čísly a názvy:

ČSN EN 61 131-1	Programovatelné řídicí jednotky - Část 1: Všeobecné informace
ČSN EN 61 131-2	Programovatelné řídicí jednotky - Část 2: Požadavky na zařízení a zkoušky
ČSN EN 61 131-3	Programovatelné řídicí jednotky - Část 3: Programovací jazyky
ČSN EN 61 131-4	Programovatelné řídicí jednotky - Část 4: Podpora uživatelů
ČSN EN 61 131-5	Programovatelné řídicí jednotky - Část 5: Komunikace
ČSN EN 61 131-7	Programovatelné řídicí jednotky - Část 7: Programování fuzzy řízení

V Evropské unii jsou tyto normy přijaty pod číslem EN IEC 61 131.

Programovací jazyky definuje norma IEC 61 131-3, která je třetí částí z rodiny norem IEC 61 131 a představuje první vážný pokus o standardizaci programovacích jazyků pro průmyslovou automatizaci.

Na normu 61 131-3 je možné pohlížet z různých hledisek, např. tak, že je to výsledek náročné práce sedmi mezinárodních společností, které do vypracování normy vložily svoji desetiletou zkušenost na poli průmyslové automatizace, nebo tak, že ve svém souhrnu obsahuje asi 200 stran textu, a asi 60 tabulek. Na jejím vytváření pracoval tým patřící do pracovní skupiny SC65B WG7 mezinárodní standardizační organizace IEC (International Electrotechnical Comission). Výsledkem je *specifikace syntaxe* a *sémantiky unifikovaného souboru programovacích jazyků, včetně obecného softwarového modelu a strukturujícího jazyka*. Tato norma byla přijata jako směrnice u většiny významných výrobců PLC.

1.2 Názvosloví

Soubor norem pro programovatelné řídicí jednotky byl v ČR sice přijat, nikoliv však přeložen do češtiny. Z toho důvodu používá tato příručka názvosloví tak, jak je přednášeno na ČVUT FSI Praha při výuce automatizace. Zároveň je všude v textu uváděno i anglické názvosloví s cílem jednoznačně přiřadit české pojmy k anglickému originálu.

1.3 Základní myšlenky normy IEC 61 131-3

Norma IEC 61 131-3 je třetí částí celé rodiny norem řady IEC 61 131. Dělí se v podstatě na dvě základní části:

- Společné prvky
- Programovací jazyky

1.3.1 Společné prvky

Typy dat

V rámci společných prvků jsou definovány typy dat. Definování datových typů napomáhá prevenci chyb v samém počátku tvorby projektu. Je nutné definovat typy všech použitých parametrů. Běžné datové typy jsou **`BOOL`**, **`BYTE`**, **`WORD`**, **`INT`** (Integer), **`REAL`**, **`DATE`**, **`TIME`**, **`STRING`** atd. Z těchto základních datových typů je pak možné odvozovat vlastní uživatelské datové typy, tzv. odvozené datové typy. Tímto způsobem můžeme např. definovat jako samostatný datový typ analogový vstupní kanál a opakovaně ho používat pod definovaným jménem.

Proměnné

Proměnné mohou být přiřazeny explicitně k hardwarovým adresám (např. vstupům, výstupům) pouze v konfiguracích, zdrojích nebo programech. Tímto způsobem je dosaženo vysokého stupně hardwarové nezávislosti a možnosti opakovaného využití softwaru na různých hardwarových platformách.

Oblast působnosti proměnných je běžně omezena pouze na tu programovou organizační jednotku, ve které byly deklarovány (proměnné jsou v ní lokální). To znamená, že jejich jména mohou být používána v jiných částech bez omezení. Tímto opatřením dojde k eliminaci řady dalších chyb. Pokud mají mít proměnné globální působnost, např. v rámci celého projektu, pak musí být jako globální deklarovány (**`VAR_GLOBAL`**). Aby bylo možné správně nastavit počáteční stav procesu nebo stroje, může být parametrům přiřazena počáteční hodnota při startu nebo studeném restartu.

Konfigurace, zdroje a úlohy

Na nejvyšší úrovni je celé softwarové řešení určitého problému řízení formulováno jako tzv. *konfigurace* (Configuration). Konfigurace je závislá na konkrétním řídicím systému, včetně uspořádání hardwaru, jako jsou například typy procesorových jednotek, paměťové oblasti přiřazené vstupním a výstupním kanálům a charakteristiky systémového programového vybavení (operačního systému).

V rámci konfigurace můžeme pak definovat jeden nebo více tzv. *zdrojů* (Resource). Na zdroj se můžeme dívat jako na nějaké zařízení, které je schopno vykonávat IEC programy.

Uvnitř zdroje můžeme definovat jednu nebo více tzv. *úloh* (Task). Úlohy řídí provádění souboru programů a/nebo funkčních bloků. Tyto jednotky mohou být prováděny buď periodicky nebo po vzniku speciální spouštěcí události, což může být např. změna proměnné.

Programy jsou vystavěny z řady různých softwarových prvků, které jsou zapsány v některém z jazyků definovaném v normě. Často je program složen ze sítě funkcí a funkčních bloků, které jsou schopny si vyměňovat data. Funkce a funkční bloky jsou základní stavební kameny, které obsahují datové struktury a algoritmus.

Programové organizační jednotky

Funkce, funkční bloky a programy jsou v rámci normy IEC 61 131 nazývány společně *programové organizační jednotky* (Program Organization Units, někdy se pro tento důležitý a často používaný pojem používá zkratka POUs).

Funkce

IEC 61 131-3 definuje standardní funkce a uživatelem definované funkce. Standardní funkce jsou např. **`ADD`** pro sčítání, **`ABS`** pro absolutní hodnotu, **`SQRT`** pro odmocninu, **`SIN`** pro sinus a **`COS`** pro cosinus. Jakmile jsou jednou definovány nové uživatelské funkce, mohou být používány opakovaně.

Funkční bloky

Na funkční bloky se můžeme dívat jako na integrované obvody, které reprezentují hardwarové řešení specializované řídicí funkce. Obsahují algoritmy i data, takže mohu zachovávat informaci o minulosti, (tím se liší od funkcí). Mají jasně definované rozhraní a skryté vnitřní proměnné, podobně jako integrovaný obvod nebo černá skříňka. Umožňují tím jednoznačně oddělit různé úrovně programátorů nebo obslužného personálu. Klasickými příklady funkčního bloku jsou např. regulační smyčka pro teplotu nebo PID regulátor.

Jakmile je jednou funkční blok definován, může být používán opakovaně v daném programu, nebo v jiném programu, nebo dokonce i v jiném projektu. Je tedy univerzální a mnohonásobně použitelný. Funkční bloky mohou být zapsány v libovolném z jazyků definovaném v normě. Mohou být tedy plně definovány uživatelem. Odvozené funkční bloky jsou založeny na standardních funkčních blocích, ale v rámci pravidel normy je možno vytvářet i zcela nové zákaznické funkční bloky.

Interface funkcí a funkčních bloků je popsán stejným způsobem: Mezi deklarací označující název bloku a deklarací pro konec bloku je uveden soupis deklarací vstupních proměnných, výstupních proměnných a vlastní kód v tzv. těle bloku.

Programy

Na základě výše uvedených definic lze říci, že program je vlastně sítí funkcí a funkčních bloků. Program může být zapsán v libovolném z jazyků definovaných v normě.

1.3.2 Programovací jazyky

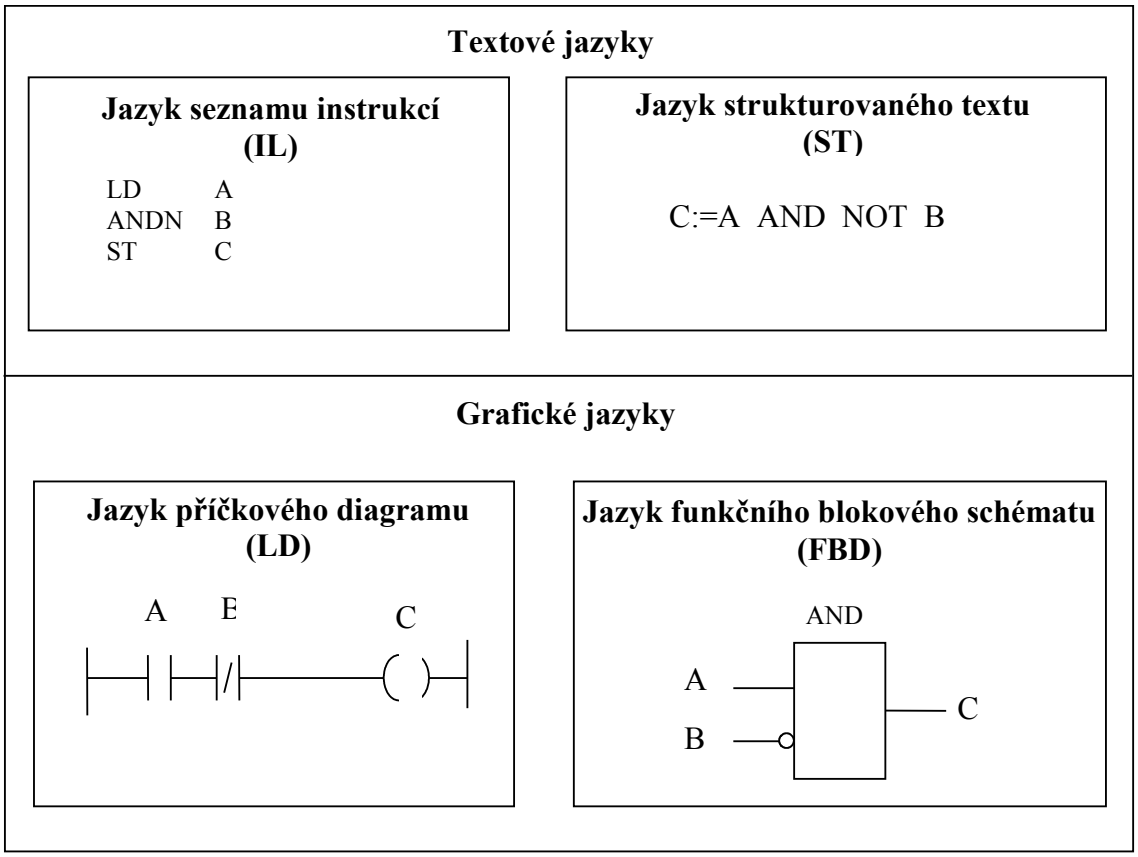
V rámci standardu jsou definovány čtyři programovací jazyky. Jejich sémantika i syntaxe je přesně definována a neponechává žádný prostor pro nepřesné vyjadřování. Zvládnutím těchto jazyků se tak otevírá cesta k používání široké škály řídicích systémů, které jsou na tomto standardu založeny.

Programovací jazyky se dělí do dvou základních kategorií:

Textové jazyky
IL - Instruction List - jazyk seznamu instrukcí
ST - Structured Text - jazyk strukturovaného textu

Grafické jazyky
LD - Ladder Diagram - jazyk příčkového diagramu (jazyk kontaktních schémat)
FBD - Function Block Diagram - jazyk funkčního blokového schématu

Pro první přehled je na Obr.1.1 stejná logická funkce, a to součin proměnné **A** a negované proměnné **B** s výsledkem ukládaným do proměnné **C**, vyjádřen ve všech čtyřech programovacích jazycích.



Obr. 1.1 Logická funkce ANDN ve čtyřech základních jazycích

Volba programovacího jazyka je závislá na zkušenostech programátora, na typu řešeného problému, na úrovni popisu problému, na struktuře řídicího systému a na řadě dalších okolností, jako jsou např. typ odvětví průmyslu, zvyklosti firmy implementující řídicí systém, zkušenosti spolupracovníků v týmu apod.

Všechny čtyři základní jazyky (IL, ST, LD a FBD) jsou vzájemně provázány. Aplikace v nich napsané tvoří určitý základní soubor informací, ke kterému přísluší velký objem technických zkušeností ("know-how"). Vytvářejí vlastně i základní komunikační nástroj pro domluvu odborníků z různých odvětví a oborů.

LD - Ladder Diagram - jazyk příčkového diagramu
- má původ v USA. Je založen na grafické reprezentaci reléové logiky.

IL - Instruction List - jazyk seznamu instrukcí
- je jeho evropský protějšek. Jako textový jazyk připomíná assembler.

FBD - Function Block Diagram - jazyk funkčního blokového schématu
- je velmi blízký procesnímu průmyslu. Vyjadřuje chování funkcí, funkčních bloků a programů jako soubor vzájemně provázaných grafických bloků, podobně jako v elektronických obvodových diagramech. Je to určitý systém prvků, které zpracovávají signály.

ST - Structured Text - jazyk strukturovaného textu
- je velmi výkonný vyšší programovací jazyk, který má kořeny ve známých jazycích Ada, Pascal a C. Obsahuje všechny podstatné prvky moderního programovacího jazyka, včetně větvení (**`IF-THEN-ELSE`** a **`CASE OF`**) a iterační smyčky (**`FOR`**, **`WHILE`** a **`REPEAT`**). Tyto prvky mohou být vnořovány. Tento jazyk je vynikajícím nástrojem pro definování komplexních funkčních bloků, které pak mohou být použity v jakémkoliv jiném programovacím jazyku.

Je známo, že pro systematické programování existují v podstatě dva přístupy, a to o*dshora - dolů* (Top-down) nebo *odspodu - nahoru* (Bottom-up).

Uvedený standard podporuje oba dva přístupy vyvíjení programů. Buď specifikujeme celou aplikaci a rozdělíme ji na části (subsystémy), deklarujeme proměnné atd. Anebo začneme programovat aplikaci odspodu, např. přes odvozené (uživatelské) funkce a funkční bloky. Ať si vybereme kteroukoliv cestu, vývojové prostředí Mosaic, které vyhovuje standardu IEC 11 131-3, ji bude podporovat a napomáhat při tvorbě celé aplikace.

2 ZÁKLADNÍ POJMY

Tato kapitola stručně vysvětluje význam a použití základních pojmů při programování PLC systému podle normy IEC 61 131-3. Tyto pojmy budou vysvětleny na jednoduchých příkladech. Detailní popis vysvětlovaných pojmů pak čtenář najde v dalších kapitolách.

2.1 Základní stavební bloky programu

Základním pojmem při programování podle normy IEC 61 131-3 je termín ***Programová Organizační Jednotka*** nebo zkráceně ***POU*** *(Program Organisation Unit)*. Jak vyplývá z názvu, POU je nejmenší nezávislá část uživatelského programu. POU mohou být dodávány od výrobce řídícího systému nebo je může napsat uživatel. Každá POU může volat další POU a při tomto volání může volitelně předávat volané POU jeden nebo více parametrů.

Existují tři základní typy POU :

- ***funkce*** *(function, FUN)*
- ***funkční blok*** *(function block, FB)*
- ***program*** *(program, PROG)*

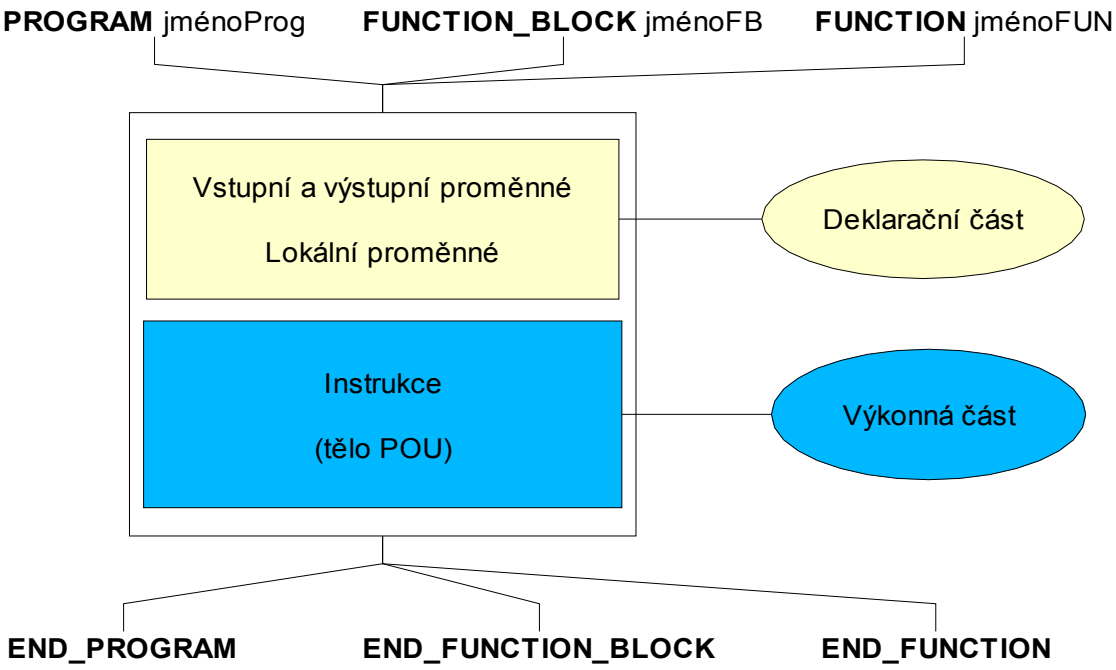
Nejjednodušší POU je ***funkce***, jejíž hlavní charakteristikou je to, že pokud je volána se stejnými vstupními parametry, musí produkovat stejný výsledek (funkční hodnotu). Funkce může vracet pouze jeden výsledek.

Dalším typem POU je ***funkční blok***, který si na rozdíl od funkce, může pamatovat některé hodnoty z předchozího volání (např. stavové informace). Ty pak mohou ovlivňovat výsledek. Hlavním rozdílem mezi funkcí a funkčním blokem je tedy schopnost funkčního bloku vlastnit paměť pro zapamatování hodnot některých proměnných. Tuto schopnost funkce nemají a jejich výsledek je tedy jednoznačně určen vstupními parametry při volání funkce. Funkční blok může také (na rozdíl od funkce) vracet více než jeden výsledek.

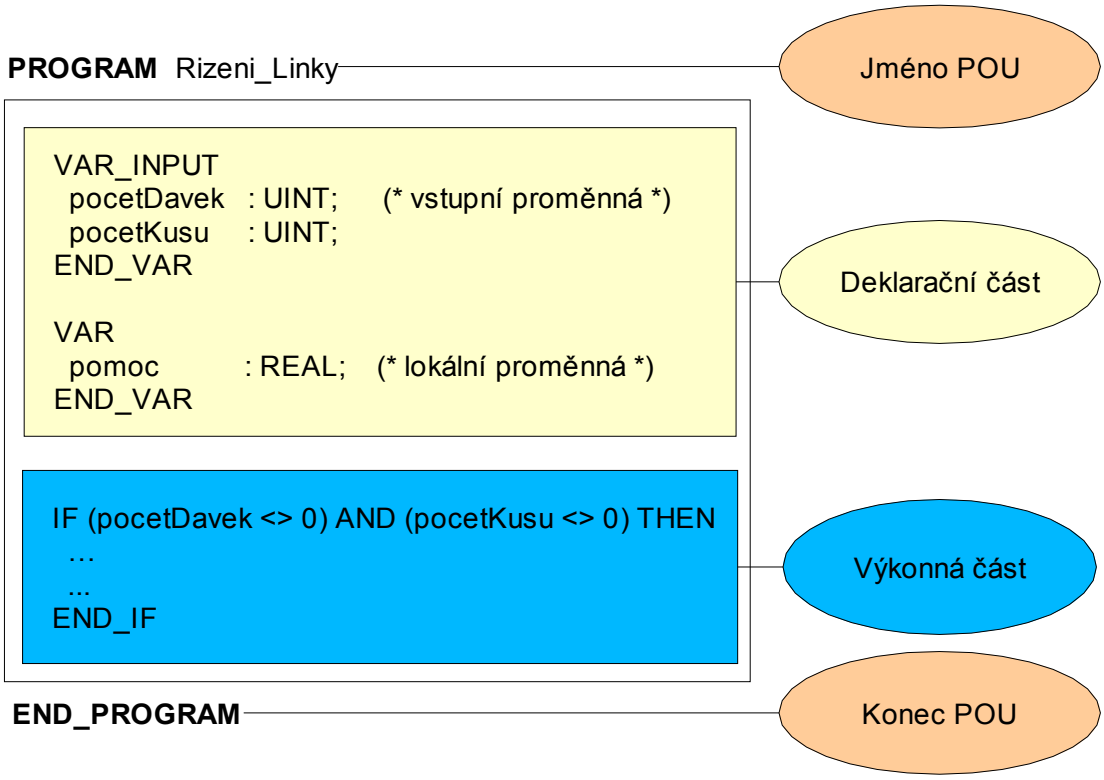
Posledním typem POU je ***program***, který představuje vrcholovou programovou jednotku v uživatelském programu. Centrální jednotka PLC může zpracovávat více programů a programovací jazyk ST obsahuje prostředky pro definice spouštění programů (v jaké periodě vykonávat program, s jakou prioritou, apod.).

Každá POU se skládá ze dvou základních částí : ***deklarační*** a ***výkonné***, jak je vidět na Obr.2.1. V deklarační části POU se definují proměnné potřebné pro činnost POU. Výkonná část pak obsahuje vlastní příkazy pro realizaci požadovaného algoritmu.

Definice POU na Obr.2.2 začíná klíčovým slovem **`PROGRAM`** a je ukončena klíčovým slovem **`END_PROGRAM`**. Tato klíčová slova vymezují rozsah POU. Za klíčovým slovem **`PROGRAM`** je uvedeno jméno POU. Poté následuje deklarační část POU. Ta obsahuje definice proměnných uvedené mezi klíčovými slovy **`VAR_INPUT`** a **`END_VAR`** resp. **`VAR`** a **`END_VAR`**. Na závěr je uvedena výkonná část POU obsahující příkazy jazyka ST pro zpracování proměnných. Texty uvedené mezi znaky **`(*`** a **`*)`** jsou poznámky (komentáře).



Obr. 2.1 Základní struktura POU



Obr.2.2 Základní struktura POU PROGRAM

2.2 Deklarační část POU

Deklarační část POU obsahuje definice proměnných potřebných pro činnost POU. Proměnné jsou používány pro ukládání a zpracování informací. Každá ***proměnná*** je definována ***jménem proměnné*** a ***datovým typem***. Datový typ určuje velikost proměnné v paměti a zároveň do značné míry určuje způsob zpracování proměnné. Pro definice proměnných jsou k dispozici standardní datové typy (**BOOL**, **BYTE**, **INT**, …). Použití těchto typů závisí na tom, jaká informace bude v proměnné uložena (např. typ **BOOL** pro informace typu ANO-NE, typ **INT** pro uložení celých čísel se znaménkem apod.). Uživatel má samozřejmě možnost definovat svoje vlastní datové typy. Umístění proměnných v paměti PLC systému zajišťuje automaticky programovací prostředí. Pokud je to potřeba, může umístění proměnné v paměti definovat i uživatel.

Proměnné můžeme rozdělit podle použití na ***globální*** a ***lokální***. Globální proměnné jsou definovány vně POU a mohou být použity v libovolné POU (jsou viditelné z libovolné POU). Lokální proměnné jsou definovány uvnitř POU a v rámci této POU mohou být používány (z ostatních POU nejsou viditelné).

A konečně proměnné jsou také používány pro předávání parametrů při volání POU. V těchto případech mluvíme o ***vstupních*** resp. ***výstupních*** proměnných.

Příklad 2.1 Deklarace proměnných POU

```
FUNCTION_BLOCK PrikladDeklaraceProm

  VAR_INPUT                          (* vstupní proměnné *)
    logPodminka         : BOOL;     (* binární hodnota *)
  END_VAR
  VAR_OUTPUT                         (* výstupní proměnné *)
    vysledek            : INT;      (* celočíselná hodnota se znaménkem *)
  END_VAR
  VAR                                (* lokální proměnné *)
    kontrolniSoucet     : UINT;     (* celočíselná hodnota *)
    mezivysledek        : REAL;     (* reálná hodnota *)
  END_VAR

END_FUNCTION_BLOCK
```

V příkladu 2.1 je uvedena definice vstupní proměnné POU, proměnná se jmenuje **logPodminka** a je typu **BOOL**, což znamená, že může obsahovat hodnoty **TRUE** (logická „1“) nebo **FALSE** (logická „0“). Tato proměnná slouží jako vstupní parametr předávaný při volání POU.

Další definovaná proměnná je výstupní, jmenuje se **vysledek** a je typu **INT** (integer), takže může obsahovat celočíselné hodnoty v rozsahu od –32 768 do +32 767. V této proměnné je předávána hodnota do nadřízené POU.

Proměnné definované mezi klíčovými slovy **VAR** a **END_VAR** jsou lokální a lze je tedy používat pouze v rámci POU. Proměnná **kontrolniSoucet** je typu **UINT** (unsigned integer) a může uchovávat celá čísla v rozsahu od 0 do 65535. Proměnná **mezivýsledek** je typu **REAL** a je určena pro práci s reálnými čísly.

2.3 Výkonná část POU

Výkonná část POU následuje za částí deklarační a obsahuje příkazy a instrukce, které jsou zpracovány centrální jednotkou PLC. Ve výjimečných případech nemusí definice POU obsahovat žádnou deklarační část a potom je výkonná část uvedena bezprostředně za definicí začátku POU. Příkladem může být POU, která pracuje pouze s globálními proměnnými, což sice není ideální řešení, ale může existovat.

Výkonná část POU může obsahovat volání dalších POU. Při volání mohou být předávány parametry pro volané funkce resp. funkční bloky.

2.4 Ukázka programu

Příklad 2.2 Ukázka programu

```
VAR_GLOBAL
  // inputs
  sb1 AT %X0.0,
  sb2 AT %X0.1,
  sb3 AT %X0.2,
  sb4 AT %X0.3    : BOOL;

  // outputs
  km1 AT %Y0.0,
  km2 AT %Y0.1,
  km3 AT %Y0.2,
  km4 AT %Y0.3    : BOOL;
END_VAR


FUNCTION_BLOCK fbStartStop
//-------------------------------------------------------------------------
  VAR_INPUT
    start         : BOOL R_EDGE;
    stop          : BOOL R_EDGE;
  END_VAR
  VAR_OUTPUT
    vystup        : BOOL;
  END_VAR

  vystup := ( vystup OR start) AND NOT stop;
END_FUNCTION_BLOCK



FUNCTION_BLOCK fbMotor
//-------------------------------------------------------------------------
  VAR_INPUT
    motorStart    : BOOL;
    motorStop     : BOOL;
  END_VAR
  VAR
    startStop     : fbStartStop;
    motorIsRun    : BOOL;
    startingTime  : TON;
  END_VAR
  VAR_OUTPUT
```

```
    star          : BOOL;
    triangle      : BOOL;
  END_VAR

  startStop( start := motorStart, stop := motorStop,
             vystup => motorIsRun);
  startingTime( IN := motorIsRun, PT := TIME#12s, Q => triangle);
END_FUNCTION_BLOCK


PROGRAM Test
//------------------------------------------------------------------------------
  VAR
    motor1        : fbMotor;
    motor2        : fbMotor;
  END_VAR

  motor1( motorStart := sb1, motorStop := sb2,
          star => km1, triangle => km2);
  motor2( motorStart := sb3, motorStop := sb4,
          star => km3, triangle => km4);
END_PROGRAM


CONFIGURATION exampleProgramST
  RESOURCE CPM
    TASK FreeWheeling(Number := 0);
    PROGRAM prg WITH FreeWheeling : Test ();
  END_RESOURCE
END_CONFIGURATION
```

3 SPOLEČNÉ PRVKY

Tato kapitola popisuje syntaxi a sémantiku základních společných prvků programovacích jazyků pro PLC systémy podle standardu IEC 61 131-3.

Syntaxe popisuje prvky, které jsou pro programování PLC k dispozici a způsoby, jakými mohou být vzájemně kombinovány.

Sémantika pak vyjadřuje jejich význam.

3.1 Základní prvky

Každý program pro PLC se skládá ze základních ***jednoduchých prvků***, určitých nejmenších jednotek, ze kterých se vytvářejí deklarace a příkazy. Tyto jednoduché prvky můžeme rozdělit na :

- *oddělovače* (Delimiters),
- *identifikátory* (Identifiers)
- *literály* (Literals)
- *klíčová slo*va (Keywords)
- *komentáře* (Comments)

Pro větší přehlednost textu jsou klíčová slova psána tučně, aby se dala lépe vyjádřit struktura deklarací a příkazů. V prostředí Mosaic jsou pak navíc barevně odlišena.

Oddělovače jsou speciální znaky (např. (,), =, :, mezera, apod.) s různým významem.

Identifikátory jsou alfanumerické řetězce znaků, které slouží pro vyjádření jmen uživatelských funkcí, návěští nebo programových organizačních jednotek (např. **`Tepl_N1`**, **`Spinac_On`**, **`Krok4`**, **`Pohyb_dopr`** apod.).

Literály slouží pro přímou reprezentaci hodnot proměnných (např. *0,1*; *84*; *3,79*; *TRUE* ; *zelena* apod.).

Klíčová slova jsou standardní identifikátory (např. **`FUNCTION`**, **`REAL`**, **`VAR_OUTPUT`**, apod.). Jejich přesný tvar a význam odpovídá normě IEC 61 131-3. Klíčová slova se nesmějí používat pro vytváření jakýchkoli uživatelských jmen. Pro zápis klíčových slov mohou být použita jak velká tak malá písmena resp. jejich libovolná kombinace.

K rezervovaným klíčovým slovům patří :

- *jména elementárních datových typů*
- *jména standardních funkcí*
- *jména standardních funkčních bloků*
- *jména vstupních parametrů standardních funkcí*
- *jména vstupních a výstupních parametrů standardních funkčních bloků*
- *prvky jazyka IL a ST*

Všechna rezervovaná klíčová slova jsou uvedena v příloze H normy IEC 61 131-3.

Komentáře nemají syntaktický ani sémantický význam, jsou však důležitou částí dokumentace programu. Komentář je možné v programu zapsat všude tam, kde je možné zapsat znak mezera. Při překladu jsou tyto řetězce ignorovány, takže mohou obsahovat i znaky národních abeced. Překladač rozeznává dva druhy komentářů :

- *obecné komentáře*
- *řádkové komentáře*

Obecné komentáře jsou řetězce znaků začínající dvojicí znaků **(*** a ukončené dvojicí znaků ***)**. To umožňuje zapisovat všechny potřebné typy komentářů, jak ukazuje dále uvedený příklad.

Řádkové komentáře jsou řetězce znaků začínající dvojicí znaků **//** a jsou ukončeny koncem řádku. Výhodou řádkových komentářů je možnost jejich vnořování do obecných komentářů (viz řádky s definicí proměnných **Pomoc1** a **Pomoc2** v následujícím příkladu, které budou považovány za komentář a nebudou překladačem překládány).

Příklad 3.1 Komentáře

```
(*************************************************
   toto je ukázka
   víceřádkového komentáře
*************************************************)

VAR_GLOBAL
  Start,              (* obecný komentář, např. : tlačítko START *)
  Stop      : BOOL;   (* tlačítko STOP *)
  Pomoc     : INT;    // řádkový komentář

(*
  Pomoc1    : INT;    // vnořený řádkový komentář
  Pomoc2    : INT;
*)

END_VAR
```

3.1.1 Identifikátory

Identifikátor je řetězec písmen (malých nebo velkých), číslic a podtrhovacích znaků, který se používá pro pojmenování následujících prvků :

- *jména konstant*
- *jména proměnných*
- *jména odvozených datových typů*
- *jména funkcí, funkčních bloků a programů*
- *jména úloh*

Identifikátor musí začínat písmenem nebo podtrhovacím znakem a nesmí obsahovat mezery. Znaky národních abeced (písmena s háčky a čárkami) nejsou v identifikátorech povoleny. Umístění podtrhovacího znaku je významné, tedy např. „BF_LM“ a „BFL_M“ jsou dva různé identifikátory. Více podtrhovacích znaků za sebou není povoleno. Velikost písmen nehraje v identifikátoru roli.

Například zápis „motor_off“ je rovnocený zápisu „MOTOR_OFF“ resp. „Motor_Off“. Pokud „motor_off“ bude jméno proměnné, pak všechny uvedené zápisy budou označovat stejnou proměnnou.

Maximální délka identifikátoru je 64 znaků.

Tab.3.1 Příklady platných a neplatných identifikátorů

Platné identifikátory	Neplatné identifikátory
XH2	2XH
MOTOR3F, Motor3F	3FMOTOR
Motor3F_Off, Motor3F_OFF	MOTOR3F__Off
SQ12	SQ$12
Prodleva_12_5	Prodleva_12.5
Rek	Řek
_3KL22	__3KL22
KM10a	KM 10a

Příklad 3.2 Identifikátory

```
TYPE
   _Phase           : ( star, triangle);
END_TYPE

VAR_GLOBAL CONSTANT
   _3KL22           : REAL := 3.22;
END_VAR

VAR_GLOBAL
  SQ12  AT %X0.0    : BOOL;
  KM10a AT %Y0.0    : BOOL;
  XH2               : INT;
END_VAR

FUNCTION_BLOCK MOTOR3F
  VAR_INPUT
    Start           : BOOL;
  END_VAR
  VAR
    Delay_12_5      : TIME;
    Status          : _Phase;
  END_VAR
  VAR_OUTPUT
    Motor3F_Off     : BOOL;
  END_VAR

END_FUNCTION_BLOCK
```

3.1.2 Literály

Literály slouží pro přímou reprezentaci hodnot proměnných.
Literály lze rozdělit do tří skupin:

- *numerické literály*
- *řetězce znaků*
- *časové literály*

Pokud chceme zdůraznit datový typ zapisovaného literálu, je možné zápis literálu zahájit jménem datového typu následovaný znakem # (např. **`REAL#12.5`**). V případě časových literálů je uvedení typu povinné (např. **`TIME#12h20m33s`**).

3.1.2.1 Numerické literály

Numerický literál je definován jako číslo (konstanta) v desítkové soustavě nebo v soustavě o jiném základu než deset (např. dvojková, osmičková a šestnáctková čísla). Numerické literály dělíme na integer literály a real literály. Jednoduché podtržítko umístěné mezi číslicemi numerického literálu nemá na jeho hodnotu vliv, je povoleno pro zlepšení čitelnosti. Příklady různých numerických literálů jsou uvedeny v Tab.3.2 .

Tab.3.2 Příklady numerických literálů

Popis	Numerický literál - příklad	Pozn.
Integer literál	14	
	INT#–9	-9
	12_548_756	12 548 756
Real literál	–18.0	
	REAL#8.0	
	0.123_4	0,1234
Real literál s exponentem	4.47E6	4 470 000
	652E–2	6,52
Literál o základu 2	2#10110111	183 desítkově
Literál o základu 8	USINT#8#127	87 desítkově
Literál o základu 16	16#FF	255 desítkově
Bool literál		
FALSE	FALSE	
	BOOL#0	0
TRUE	TRUE	
	BOOL#1	1

Příklad 3.3 Numerické literály

```
VAR_GLOBAL CONSTANT
  Const1          : REAL  := 4.47E6;
  Const2          : LREAL := 652E-2;
END_VAR

VAR_GLOBAL
  MagicNum        : DINT  := 12_548_756;
  Amplitude       : REAL  := 0.123_4;
  BinaryNum       : BYTE  := 2#10110111;
  OctalNum        : USINT := 8#127;
  HexaNum         : USINT := 16#FF;
  LogicNum        : BOOL  := TRUE;
END_VAR

FUNCTION Parabola : REAL
  VAR_INPUT
    x,a,b,c   : REAL;
  END_VAR

  IF a <> 0.0 THEN
    Parabola := a*x*x + b*x + c;
  ELSE
    Parabola := 0.0;
  END_IF;
END_FUNCTION


PROGRAM ExampleLiterals
  VAR
    x,y    : REAL;
  END_VAR

  y := Parabola(x := x, a := REAL#2.0, b := Const1, c := 0.0 );
END_PROGRAM
```

3.1.2.2 Literály řetězce znaků

Řetězec znaků je posloupnost žádného znaku (prázdný řetězec) nebo více znaků, která je uvozena a ukončena jednoduchou uvozovkou (‘). Příklady: ‘‘ (prázdný řetězec), ‘teplota‘ (neprázdný řetězec o délce sedm, obsahující slovo teplota).

Znak dolar, $, se používá jako prefix, který umožňuje uvedení speciálních znaků v řetězci. Speciální znaky, které se netisknou, se používají např. pro formátování textu pro tiskárnu nebo na displej. Pokud je znak dolaru před dvěma šestnáctkovými ciframi, je řetězec interpretován jako šestnáctková reprezentace osmibitového kódu znaku. Např. řetězec ‘$0D$0A‘ je chápán jako reprezentace dvou kódů, a to 00001101 a 00001010. První kód představuje v ASCII tabulce znak Enter, (CR, desítkově 13) a druhý kód odřádkování (LF, desítkově 10).

Literály řetězce znaků, tzv. stringy, se používají např. pro výměnu textů mezi různými PLC nebo mezi PLC a dalšími komponentami automatizačního systému, nebo při programování textů, které se zobrazují na řídicích jednotkách nebo operátorských panelech.

Tab.3.3 Speciální znaky v řetězcích

Zápis	Význam
$$	Znak dolar
$'	Znak jednoduchý apostrof
$L nebo $l	Znak Line feed (16#0A)
$N nebo $n	Znak New line
$P nebo $p	Znak New page
$R nebo $r	Znak Carriage return (16#0D)
$T nebo $t	Znak tabelátor (16#09)

Tab.3.4 Příklady literálů řetězce znaků

Příklad	Poznámka
''	Prázdný řetězec, délka 0
'temperature'	Neprázdný řetězec, délka 11 znaků
'Character $'A$''	Řetězec obsahující uvozovky (Character 'A')
' End of text $0D$0A'	Řetězěc ukončený znaky CR a LF
' Price is 12$$'	Řetězec obsahující znak $
'$01$02$10'	Řetězěc obsahující 3 znaky s kódy 1,2 a 16

Příklad 3.4 Řetězce znaků

```
PROGRAM ExampleStrings
  VAR
    message     : STRING := '';  // empty string
    value       : INT;
    valid       : BOOL;
  END_VAR

  IF valid THEN
    message := 'Temperature is ';
    message := CONCAT(IN1 := message, IN2 := INT_TO_STRING(value));
    message := message + ' [C]';
  ELSE
    message := 'Temperature is not available !';
  END_IF;
  message := message + '$0D$0A';
END_PROGRAM
```

3.1.2.3 Časové literály

Při řízení v podstatě potřebujeme dva různé typy údajů, které nějakým způsobem souvisí s časem. Za prvé je to ***údaj o trvání***, tj. o době, která uplynula nebo má uplynout v souvislosti s nějakou událostí. Za druhé je to údaj o „absolutním čase“, složeném z *data* podle kalendáře (Date) a z *časového údaje v rámci jednoho dne, tzv.* ***denního času*** (Time of Day). Tento časový údaj se může využívat pro synchronizaci začátku nebo konce řízené události vzhledem k absolutnímu časovému rámci. Příklady časových literálů jsou v Tab.3.6.

Doba trvání. Časový literál pro dobu trvání je uvozen některým z klíčových slov **`T#`**, **`t#`**, **`TIME#`**, **`time#`**. Vlastní časový údaj je vyjádřen v časových jednotkách: hodinách, minutách, sekundách a milisekundách. Zkratky pro jednotlivé části časového údaje jsou uvedeny v Tab.3.5. Mohou být vyjádřeny malým i velkým písmenem.

Tab.3.5 Zkratky pro časové údaje

Zkratka	Význam
ms, MS	Milisekundy (Miliseconds)
s, S	Sekundy (Seconds)
m, M	Minuty (Minutes)
h, H	Hodiny (Hours)
d, D	Dny (Days)

Denní čas a datum. Reprezentace údaje o datu a čase v rámci dne je stejná jako v ISO 8601. Prefix může být buď krátký nebo dlouhý. Klíčová slova pro datum jsou **`D#`** nebo **`DATE#`**. Pro časový údaj v rámci jednoho dne se používají klíčová slova **`TOD#`** nebo **`TIME_OF_DAY#`**. Pro souhrnný údaj o „absolutním čase” pak klíčová slova **`DT#`** nebo **`DATE_AND_TIME#`**. Velikost písmen opět nehraje roli.

Tab.3.6 Příklady různých časových literálů

Popis	Příklady
Doba trvání	T#24ms, t#6m1s, t#8.3s t#7h_24m_5s, TIME#416ms
Datum	D#2003-06-21 DATE#2003-06-21
Denní čas	TOD#06:32:15.08 TIME_OF_DAY#11:38:52.35
Datum a denní čas	DT#2003-06-21-11:38:52.35 DATE_AND_TIME#2003-06-21-11:38:52.35

Příklad 3.5 Časové literály

```
VAR_GLOBAL
  myBirthday         : DATE := D#1982-06-30;
  firsthManOnTheMoon : DT   := DT#1969-07-21-03:56:00;
END_VAR

PROGRAM ExampleDateTime
  VAR
    coffeeBreak  : TIME_OF_DAY := TOD#10:30:00.0;
    dailyTime    : TOD;
    timer        : TON;
    startOfBreak : BOOL;
    endOfBreak   : BOOL;
  END_VAR

  dailyTime    := TIME_TO_TOD( GetTime());
  startOfBreak := dailyTime > coffeeBreak AND dailyTime < TOD#12:00:00;
  timer(IN := startOfBreak, PT := TIME#15m, Q => endOfBreak);
END_PROGRAM
```

3.2 Datové typy

Pro programování v některém z jazyků podle normy IEC 61 131-3 jsou definovány tzv. ***elementární***, předdefinované datové typy, (Elementary data types), dále jsou definovány ***rodové*** datové typy (Generic data type) pro příbuzné skupiny datových typů. A konečně je k dispozici mechanizmus, kterým může uživatel vytvářet vlastní ***odvozené*** *(uživatelské)* datové typy (Derived data type, Type definition).

3.2.1 Elementární datové typy

Elementární datové typy jsou charakterizované šířkou dat (počtem bitů) a případně i rozsahem hodnot. Přehled podporovaných datových typů je uveden v Tab.3.7.

Tab.3.7 Elementární datové typy

Klíčové slovo	Anglicky	Datový typ	Bitů	Rozsah hodnot
BOOL	Boolean	Boolovské číslo	1	0,1
SINT	Short integer	Krátké celé číslo	8	–128 až 127
INT	Integer	Celé číslo	16	–32 768 až +32 767
DINT	Double integer	Celé číslo, dvojnásobná délka	32	–2 147 483 648 až +2 147 483 647
USINT	Unsigned short integer	Krátké celé číslo bez znaménka	8	0 až 255
UINT	Unsigned integer	Celé číslo bez znaménka	16	0 až 65 535
UDINT	Unsigned double integer	Celé číslo bez znaménka, dvojnásobná délka	32	0 až +4 294 967 295
REAL	Real (single precision)	Číslo v pohyblivé řádové čárce (jednoduchá přesnost)	32	±2.9E-39 až ±3.4E+38 Podle IEC 559
LREAL	Long real (double precision)	Číslo v pohyblivé řádové čárce (dvojnásobná přesnost)	64	Podle IEC 559
TIME	Duration	Trvání času	24d 20:31:23.647	
DATE	Date (only)	Datum	Od 1.1.1970 00:00:00	
TIME_OF_DAY nebo TOD	Time of day (only)	Denní čas	24d 20:31:23.647	
DATE_AND_TIME nebo DT	Date and time of day	„Absolutní čas“	Od 1.1.1970 00:00:00	
STRING	String	Řetězec	Max.255 znaků	
BYTE	Byte(bit string of 8 bits)	Sekvence 8 bitů	8	Není deklarován rozsah
WORD	Word (bit string of 16bits)	Sekvence 16 bitů	16	Není deklarován rozsah
DWORD	Double word (bit string of 32 bits)	Sekvence 32 bitů	32	Není deklarován rozsah

Inicializace elementárních datových typů

Důležitým principem při programování podle normy IEC 61 131-3 je, že všechny proměnné v programu mají inicializační (počáteční) hodnotu. Pokud uživatel neuvede jinak, bude proměnná inicializována implicitní (předdefinovanou, default) hodnotou podle použitého datového typu. Předdefinované počáteční hodnoty pro elementární datové typy jsou převážně nuly, u data je to D#1970-01-01. Souhrn předdefinovaných počátečních hodnot je uveden v Tab.3.8.

Tab.3.8 Předdefinované počáteční hodnoty pro elementární datové typy

Datový typ	Počáteční hodnota (Initial Value)
BOOL, SINT, INT, DINT	0
USINT, UINT, UDINT	0
BYTE, WORD, DWORD	0
REAL, LREAL	0.0
TIME	T#0s
DATE	D#1970-01-01
TIME_OF_DAY	TOD#00:00:00
DATE_AND_TIME	DT#1970-01-01-00:00:00
STRING	’ ’ (prázdný string)

3.2.2 Rodové datové typy

Rodové datové typy vyjadřují vždy celou skupinu (rod) datových typů. Jsou uvozeny prefixem **`ANY`**. Např. zápisem **`ANY_BIT`** se rozumí všechny datové typy uvedené v následujícím výčtu: **`DWORD, WORD, BYTE, BOOL`**. Přehled rodových datových typů je uveden v Tab.3.9. Názvy rodových datových typů začínající **`ANY`**_ nejsou podle IEC klíčovými slovy. Slouží pouze k označení skupiny typů se stejnými vlastnostmi.

Tab.3.9 Přehled rodových datových typů

<table>
<tr><th colspan="5">ANY</th></tr>
<tr><th>ANY_BIT</th><th colspan="3">ANY_NUM</th><th>ANY_DATE</th><th rowspan="3">TIME
STRING</th></tr>
<tr><td rowspan="2">BOOL
BYTE
WORD
DWORD</td><th colspan="2">ANY_INT</th><th>ANY_REAL</th><td rowspan="2">DATE
DATE_AND_TIME
TIME_OF_DAY</td></tr>
<tr><td>INT
SINT
DINT</td><td>UINT
USINT
UDINT</td><td>REAL
LREAL</td></tr>
</table>

3.2.3 Odvozené datové typy

Odvozené typy, tzn. typy specifikované buď výrobcem nebo uživatelem, mohou být deklarovány pomocí textové konstrukce **TYPE**…**END_TYPE**. Jména nových typů, jejich datové typy, případně i s jejich inicializačními hodnotami, jsou uvedena v rámci této konstrukce. Tyto odvozené datové typy se pak mohou dále používat spolu s elementárními datovými typy v deklaracích proměnných. Definice odvozeného datového typu je globální, tj. může být použita v kterékoliv části programu pro PLC. Odvozený datový typ dědí vlastnosti typu, ze kterého byl odvozen.

3.2.3.1 Jednoduché odvozené datové typy

Jednoduché odvozené datové typy vycházejí z přímo elementárních datových typů. Nejčastějším důvodem pro zavedení nového datového typu bývá odlišná inicializační hodnota nového typu, kterou lze přiřadit přímo v deklaraci typu pomocí přiřazovacího operátoru „:=“. Pokud není inicializační hodnota v deklaraci nového typu uvedena, dědí nový typ inicializační hodnotu typu, ze kterého byl odvozen.

Mezi jednoduché odvozené datové typy patří také výčet hodnot (Enumerated data type). Ten se zpravidla používá pro pojmenování vlastností resp. variant namísto přidělení číselného kódu ke každé variantě, což zlepšuje čitelnost programu. Inicializační hodnotou výčtového typu je vždy hodnota prvního prvku uvedeného ve výčtu.

Příklad 3.6 Příklad jednoduchých odvozených datových typů

```
TYPE
  TMyINT       : INT;                 // jednoduchy odvozeny datovy typ
  TRoomTemp    : REAL := 20.0;        // novy datovy typ s inicializaci
  THomeTemp    : TRoomTemp;
  TPumpMode    : ( off, run, fault); // novy typ deklarovany vyctem hodnot
END_TYPE

PROGRAM SingleDerivedType
  VAR
    pump1Mode    : TPumpMode;
    display      : STRING;
    temperature  : THomeTemp;
  END_VAR

  CASE pump1Mode OF
    off   : display := 'Pump no.1 is off';
    run   : display := 'Pump no.1 is running';
    fault : display := 'Pump no.1 has a problem';
  END_CASE;
END_PROGRAM
```

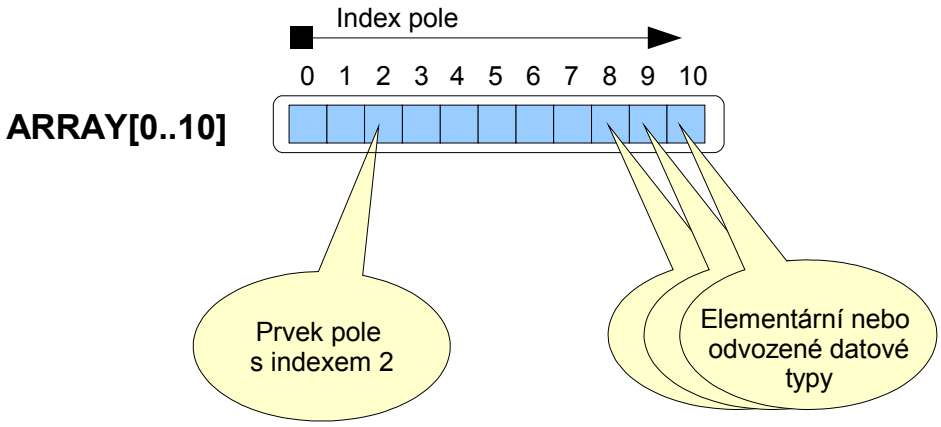
Jednoduché proměnné, které mají deklarován uživatelský typ, mohou být použity všude tam, kde může být použita proměnná s „rodičovským“ typem. Tedy např. proměnná „**temperature**“ z příkladu 3.6 může být použita všude, kde mohou být používány proměnné typu **REAL**. Toto pravidlo může být aplikováno rekurzivně.

Novým datovým typem může být také ***pole*** (Array) nebo ***struktura*** (Structure).

3.2.3.2 Odvozené datové typy pole

Jednorozměrná pole

Pole je uspořádaná řada prvků (elementů) stejného datového typu. Každý prvek pole má přidělen index, pomocí kterého lze k prvku přistupovat. Jinak řečeno hodnota indexu určuje, s kterým prvkem pole budeme pracovat. Index může nabývat pouze hodnot v rozsahu definice pole. Jestliže hodnota indexu překročí deklarovaný rozměr pole tak bude vyhlášena tzv. Run-time chyba (chyba vyhlášená za běhu systému). Jednorozměrné pole je pole, které má pouze jeden index, jak ukazuje Obr. 3.1.



Obr. 3.1 Jednorozměrné pole

Prvek pole může být elementárního nebo odvozeného datového typu. Pole instancí POU nejsou dosud podporována. Deklaraci odvozeného datového typu pole ukazuje příklad 3.7. Deklarace se provádí pomocí klíčového slova **`ARRAY`**, za kterým následuje rozměr pole v hranatých závorkách. Rozměr pole udává rozsah přípustných indexů. Za rozměrem pole je potom uvedeno klíčové slovo **`OF`** s uvedením datového typu pro prvky pole. Index prvního prvku pole musí být kladné číslo nebo nula. Záporné indexy nejsou přípustné. Maximální velikost pole je omezena rozsahem paměti proměnných v řídícím systému.

Deklarace typu pole může také obsahovat inicializaci jednotlivých prvků (viz typ **`TbyteArray`** a **`TRealArray`**). Inicializační hodnoty jsou uvedeny v deklaraci typu pole za přiřazovacím operátorem „:=“ v hranatých závorkách. Pokud je uvedeno méně inicializačních hodnot než odpovídá rozměru pole, pak jsou prvky s neuvedenými inicializačními hodnotami inicializovány počáteční hodnotou podle použitého datového typu. Pro inicializaci velkého počtu prvků pole stejnou hodnotou lze použít tzv. opakovač. V tomto případě se na místě pro inicializační hodnotu uvede počet opakování následovaný inicializační hodnotou v kulatých závorkách. Například zápisem **`25( -1)`** budeme inicializovat 25 prvků pole hodnotou -1.

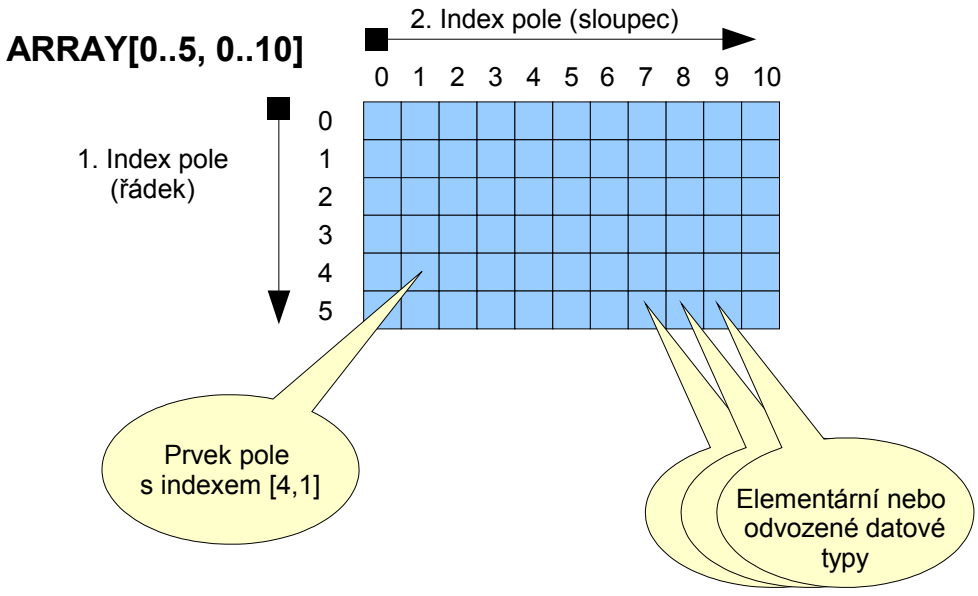
Příklad 3.7 Odvozený datový typ jednorozměrné pole

```
TYPE
  TVector     : ARRAY[0..10]  OF INT;
  TByteArray  : ARRAY[1..10]  OF BYTE := [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
  TRealArray  : ARRAY[5..9]   OF REAL := [ 11.2, 12.5, 13.1];
  TBigArray   : ARRAY[1..999] OF SINT := [ 499( -1), 0, 499( 1)];
END_TYPE

PROGRAM Example1DimArray
  VAR
    index      : INT;
    samples    : TVector;
    buffer     : TByteArray;
    intervals  : TRealArray;
    result     : BOOL;
  END_VAR

  FOR index := 0 TO 10 DO
    samples[index] := 0;              // clear all samples
  END_FOR;
  result := intervals[5] = 11.2;      // TRUE
  result := intervals[8] = 0.0;       // TRUE
END_PROGRAM
```

Vícerozměrná pole

Vícerozměrná pole jsou pole, kde pro přístup k jednomu prvku pole potřebujeme více než jeden index. Pole má potom dva a více rozměrů, které se mohou pro každý index lišit. Dvourozměrného pole si lze představit jako matici prvků jak ukazuje Obr.3.2. Prvky vícerozměrných polí mohou být elementárního nebo odvozeného datového typu, stejně jako u polí jednorozměrných.

Překladač v prostředí Mosaic podporuje práci s maximálně čtyřrozměrnými poli.



Obr. 3.2 Dvourozměrné pole

Inicializace u vícerozměrných polí se provádí stejně jako pro jednorozměrná pole s tím, že jsou nejprve inicializovány všechny prvky pro první rozměr (tj. například pole[0,0], pole[0,1], pole[0,2] až pole[0,n]) a poté se postup opakuje pro další hodnoty prvního indexu. Takže jako poslední jsou inicializovány prvky pole[m,0], pole[m,1], pole[m,2] až konečně pole[m,n]. Při inicializaci vícerozměrných polí lze také používat opakovače pro inicializaci více prvků najednou jak je ukázáno v příkladu 3.8 u typu **`TThreeDimArray1`**. V komentáři je pak uvedena stejná deklarace bez použití opakovačů.

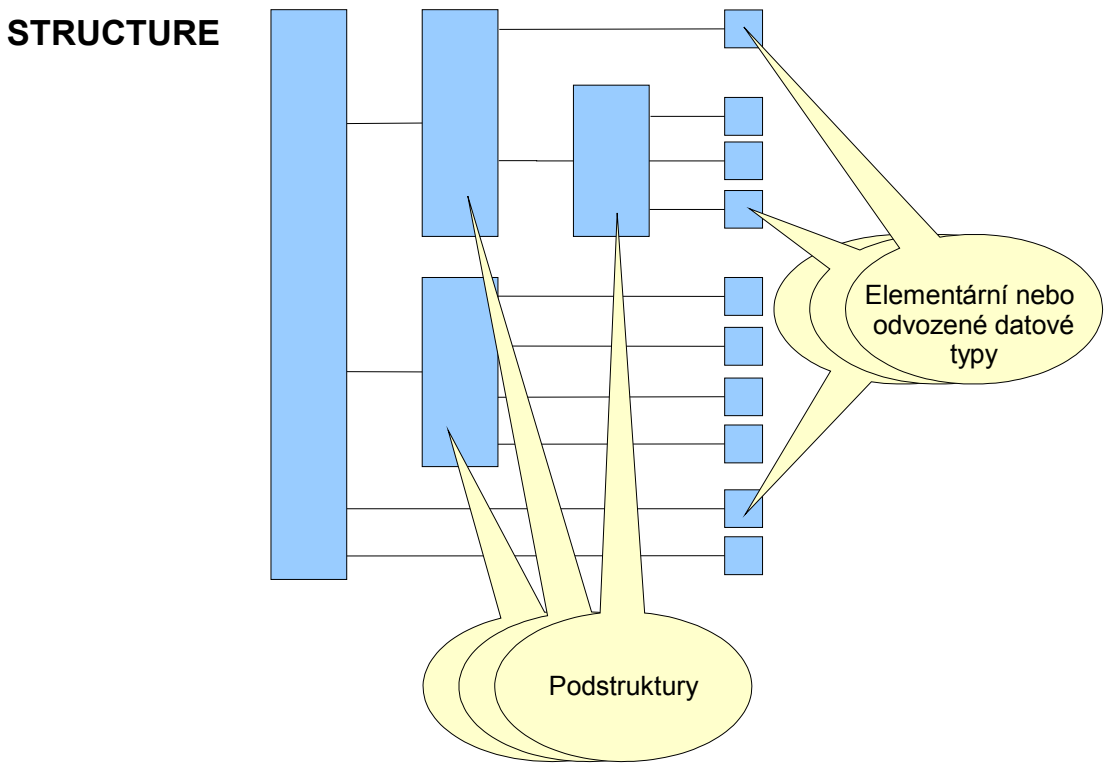
Příklad 3.8 Odvozený datový typ vícerozměrné pole

```
TYPE
  TTwoDimArray   : ARRAY [1..2,1..4] OF SINT := [ 11, 12, 13, 14,
                                                  21, 22, 23, 24 ];

  TThreeDimArray : ARRAY [1..2, 1..3, 1..4] OF BYTE :=
                                                 [ 111, 112, 113, 114,
                                                   121, 122, 123, 124,
                                                   131, 132, 133, 134,
                                                   211, 212, 213, 214,
                                                   221, 222, 223, 224,
                                                   231, 232, 233, 234 ];

  TThreeDimArray1 : ARRAY [1..2, 1..3, 1..4] OF BYTE :=
                                                 [ 4(11), 4(12), 4(13),
                                                   4(21), 4(22), 4(23) ];
(*
  TThreeDimArray1 : ARRAY [1..2, 1..3, 1..4] OF BYTE :=
                                                 [ 11, 11, 11, 11,
                                                   12, 12, 12, 12,
                                                   13, 13, 13, 13,
                                                   21, 21, 21, 21,
                                                   22, 22, 22, 22,
                                                   23, 23, 23, 23 ];
*)
END_TYPE


PROGRAM ExampleMultiDimArray
  VAR
    twoDimArray   : TTwoDimArray;
    threeDimArray : TThreeDimArray;
    element       : BYTE;
    result        : BOOL;
  END_VAR

   result  := twoDimArray[1, 4] = 14;     // TRUE
   element := threeDimArray[ 2, 1, 3];    // element = 213
END_PROGRAM
```

Podobně jako odvozený datový typ pole lze také deklarovat přímo proměnnou typu pole, jak je ukázáno v kap.3.

3.2.3.3 Odvozený datový typ struktura

Struktury jsou datové typy, které obsahují podobně jako pole více prvků (položek). Avšak na rozdíl od polí nemusí být všechny prvky ve struktuře stejného datového typu. Strukturu lze odvodit jak elementárních tak z již odvozených datových typů. Struktura může být vybudovaná hierarchicky, což znamená že prvkem struktury může být již definovaná struktura. Situaci popisuje Obr.3.3.



Obr. 3.3 Struktura

Definice nového datového typu struktura se provádí pomocí klíčových slov **`STRUCT`** a **`END_STRUCT`** v rámci konstrukce **`TYPE`** … **`END_TYPE`**. Uvnitř **`STRUCT`** … **`END_STRUCT`** jsou uvedena jména jednotlivých prvků struktury a jejich datové typy. Stejně jako tomu bylo u předchozích odvozených datových typů, lze i struktury inicializovat uvedením hodnoty prvku za znakem „**`:=`**“.

Pokud vytvoříme proměnnou typu struktura, pak přístup k jednotlivým prvkům struktury bude „**`jménoProměnné.jménoPrvku`**“ jak ukazuje příklad 3.9.

Příklad 3.9 Odvozený datový typ struktura

```
TYPE
  TProduct :
    STRUCT
      name         : STRING := 'Engine M11';
      code         : UINT;
      serie        : DINT;
      serialNum    : UDINT;
      expedition   : DATE;
    END_STRUCT;
END_TYPE

PROGRAM ExampleStruct
  VAR
    product      : TProduct;
    product1     : TProduct;
  END_VAR

  product.code       := 700;
  product.serie      := 0852;
  product.serialNum  := 12345;
  product.expedition := DATE#2002-02-13;
END_PROGRAM
```

Inicializace proměnné typu struktura se provádí s pomocí jmen prvků struktury při deklaraci proměnné. Rozdíl mezi inicializací datového typu struktura a inicializací proměnné typu struktura je ukázán v příkladech 3.9 a 3.10. Fukční rozdíl je zřejmý. Zatímco v příkladu 3.9 bude mít každá proměnná typu **`TProduct`** prvek **`name`** automaticky inicializovaný na hodnotu **`'Engine M11'`**, pak v příkladu 3.10 je implicitní inicializace prvku **`name`** prázdný řetězec nahrazena řetězcem **`'Engine M11'`** pouze v případě proměnné **`product`**.

Příklad 3.10 Inicializace proměnné typu struktura

```
TYPE
  TProduct :
    STRUCT
      name         : STRING;
      code         : UINT;
      serie        : DINT;
      serialNum    : UDINT;
      expedition   : DATE;
    END_STRUCT;
END_TYPE

PROGRAM ExampleStruct
  VAR
    product      : Tproduct := ( name := 'Engine M11');
    product1     : TProduct;
  END_VAR

  product.code       := 700;
  product.serie      := 0852;
  product.serialNum  := 12345;
  product.expedition := DATE#2002-02-13;
END_PROGRAM
```

3.2.3.4 Kombinace struktur a polí v odvozených datových typech

Pole a struktury lze v definici odvozených datových typů libovolně kombinovat. Prvkem struktury tedy může být pole nebo prvkem pole může být struktura jak ukazuje příklad 3.11.

Příklad 3.11 Struktura jako prvek pole

```
VAR_GLOBAL CONSTANT
  NUM_SENSORS    : INT := 12;
END_VAR

TYPE
  TLimit :
    STRUCT
      low          : REAL := 12.5;
      high         : REAL := 120.0;
    END_STRUCT;

  TSensor :
    STRUCT
      status       : BOOL;
      pressure     : REAL;
      calibration  : DATE;
      limits       : TLimit;
    END_STRUCT;

   TSenzorsArray  : ARRAY[1..NUM_SENSORS] OF TSensor;
END_TYPE

PROGRAM ExampleArrayOfStruct
  VAR
    sensors : TSenzorsArray;
    i       : INT;
  END_VAR

  FOR i := 1 TO NUM_SENSORS DO
    IF (sensors[i].pressure >= sensors[i].limits.low)  AND
       (sensors[i].pressure <= sensors[i].limits.high)
    THEN
       sensors[i].status := TRUE;
    ELSE
       sensors[i].status := FALSE;
    END_IF;
  END_FOR;
END_PROGRAM
```

3.2.4 Datový typ pointer

Datový typ pointer je rozšířením normy IEC 61 131. Jinými slovy pointer není zmíněnou normou definován a programy, ve kterých bude tento datový typ použit, nebude možné použít pro PLC programované v jiném prostředí než Mosaic.

Důvodem proč tento datový typ mezi normovanými datovými typy chybí je jednoznačně bezpečnost programování. Chybné použití pointeru může mít za následek úplné zhroucení programu, což je při řízení technologie samozřejmě naprosto nepřípustný stav. Přitom chybu nelze odhalit ani ve fázi překladu programu ani za běhu programu. Zkušenosti z jazyka C, kde se datových typ pointer hojně používá, ukazují, že velká část nekorektního chování programů je způsobena právě nesprávnou prací s pointery. Na druhou stranu existuje asi jen velmi málo programů napsaných v jazyce C, kde by datový typ pointer nebyl použit. Co z toho plyne? Pointer může být velmi dobrý sluha ale zlý pán. Odpovědnost za správnost programu s pointery leží jen a pouze na programátorovi, protože prostředky, které mu v jiných situacích pomáhají odhalovat chyby (překladač, typová kontrola, run-time kontroly, atd.) jsou v případě pointerů neúčinné. Výhodou pointerů je pak vyšší efektivnost programování. Pointery umožňují v řadě případů kratší a tím i rychlejší programy, zejména pokud jsou v programu použity struktury, pole a jejich kombinace. A konečně posledním důvodem pro pointery je prostě skutečnost, že existují problémy, které lze efektivně vyřešit pouze s použitím pointerů.

Pointer je vlastně ukazatel na proměnnou, která může být jak elementárního tak odvozeného typu. Deklarace pointeru se provádí pomocí klíčového slova **`PTR_TO`** za, kterým následuje jméno datového typu, na který pointer ukazuje. Datový typ pointer lze použít všude tam, kde lze použít elementární datový typ. Pointer na POU není podporován.

Proměnná typu pointer obsahuje vlastně adresu nějaké jiné proměnné. S pointerem lze pracovat dvojím způsobem. Buď lze měnit jeho hodnotu (zvyšovat, snižovat, atd.) a tím měnit, na kterou proměnnou pointer ukazuje. Pak lze samozřejmě pracovat s hodnotou proměnné, na kterou pointer ukazuje. První zmíněné operaci se říká pointerová aritmetika, druhá operace je pak většinou označována jako dereference pointeru.

Pointerová aritmetika

První operací, kterou musí každý program s pointerem provést, je naplnit adresu proměnné, na kterou bude pointer ukazovat. Implicitní inicializace datového typu pointer je -1, což vlastně znamená, že pointer neukazuje na žádnou proměnnou. To je také jediný případ, který může být zachycen run-time kontrolou řídícího systému a vyhlášen jako chyba.

Inicializace pointeru, tj, jeho naplnění adresou proměnné, na kterou bude ukazovat, se provádí pomocí systémové funkce **`ADR()`**. Parametrem této funkce je jméno proměnné, jejíž adresu chceme do pointeru naplnit. Například zápis **`myPtr := ADR( myVar)`** naplní adresu proměnné **`myVar`** do pointeru **`myPtr`**. Jinými slovy pointer **`myPtr`** bude ukazovat na proměnnou **`myVar`**.

S proměnnou typu pointer lze provádět aritmetické operace za účelem změny adresy proměnné. Ve výrazech lze typ **`PTR_TO`** kombinovat s datovými typy **`ANY_INT`**. Jestliže proměnná **`myVar`** bude umístěná v paměti na adrese %MB100 a proměnná **`yourVar`** bude ležet v paměti na adrese %MB101, potom výraz **`myPtr := myPtr + 1`** zvýší hodnotu pointeru o 1, takže pointer bude ukazovat na proměnnou **`yourVar`** (namísto původní **`myVar`**). Samozřejmě pouze za předpokladu, že obě proměnné jsou datového typu, který zabírá v paměti jeden byte. Aritmetika v případě typu **`PTR_TO`** funguje bytově, což znamená, že po přičtení hodnoty 15 bude pointer vždy ukazovat o 15 bytů dále v paměti.

Dereference pointeru

Dereference pointeru je operace, která umožňuje pracovat s proměnnou, na kterou pointer ukazuje. Pro derefenci je využíván znak ^. Takže zápis **`value := myPtr^`** naplní do proměnné **`value`** hodnotu proměnné **`myVar`** (samozřejmě za předpokladu, že **`myPtr`** ukazuje na **`myVar`** a proměnná **`value`** je stejného typu jako proměnná **`myVar`**).

Příklad 3.12 Pointery

```
VAR_GLOBAL
  arrayINT    : ARRAY[0..10] OF INT;
END_VAR

PROGRAM ExamplePtr
  VAR
    intPTR    : PTR_TO INT;
    varINT    : INT;
  END_VAR

  intPTR  := ADR( arrayINT[0]);          // init ptr
  intPTR^ := 11;                         // arrayINT[0] := 11;
  intPTR  := intPTR + sizeof( INT);      // ptr to next item
  intPTR^ := 22;                         // arrayINT[1] := 22;
  intPTR  := intPTR + sizeof( INT);      // ptr to next item
  varINT  := intPTR^;                    // varINT := arrayINT[2];
END_PROGRAM
```

Příklad 3.12 používá pro zvýšení adresy, na kterou pointer **intPTR** ukazuje, funkci **sizeof()**. Tato funkce vrací počet bytů zadaného datového typu nebo proměnné.

Další příklad ukazuje, jak snadno lze při práci s pointery udělat chybu. Program je totožný jako v příkladu 3.12 a liší se pouze inicializací pointeru **intPTR**. Zatímco v prvním případě je inicializace prováděna v každém cyklu příkazem **intPTR := ADR(arrayINT[0])**, v druhém příkladu je pointer inicializován už v deklaraci proměnné **intPTR : PTR_TO INT := ADR(arrayINT[0])**. To způsobí, že první cykl programu po restartu systému bude sice správně, ale již v druhém cyklu bude pointer začínat s adresou prvku **arrayINT[2]** namísto **arrayINT[0]**. Při cyklickém vykonávání programu to pak znamená, že program v příkladu 3.13 v krátké době popíše celou paměť proměnných hodnotami **INT#11** a **INT#22**, což jistě nechceme Takže mějme na paměti, že použití pointeru vyžaduje vždy zvýšenou opatrnost.

Příklad 3.13 Chybná inicializace pointeru

```
VAR_GLOBAL
  arrayINT    : ARRAY[0..10] OF INT;
END_VAR

PROGRAM ExamplePtrErr
  VAR
    intPTR    : PTR_TO INT := ADR( arrayINT[0]);
    varINT    : INT;
  END_VAR

                                         // for 1st cycle only !!!
  intPTR^ := 11;                         // arrayINT[0] := 11;
  intPTR  := intPTR + sizeof( INT);      // ptr to next item
  intPTR^ := 22;                         // arrayINT[1] := 22;
  intPTR  := intPTR + sizeof( INT);      // ptr to next item
  varINT  := intPTR^;                    // varINT := arrayINT[2];
END_PROGRAM
```

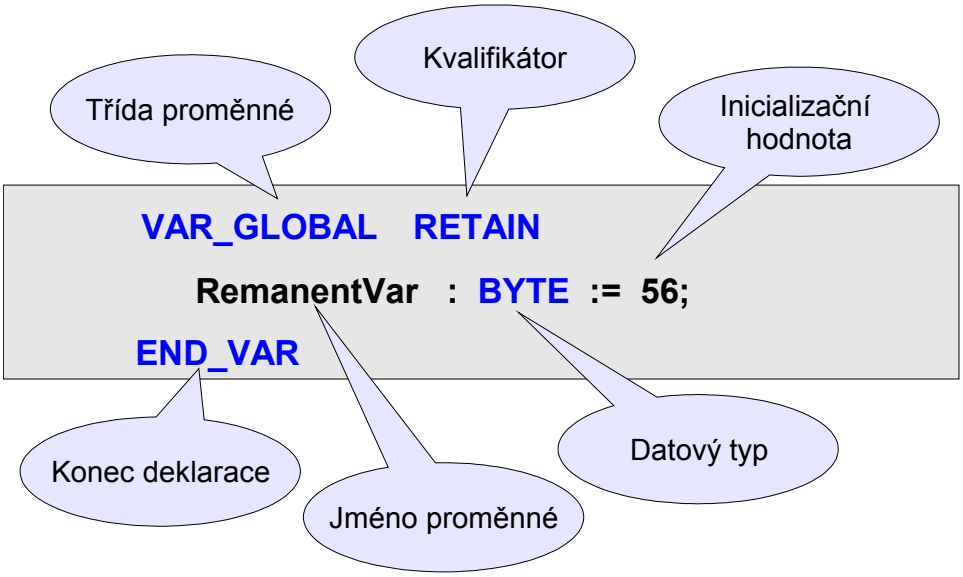
3.3 Proměnné

Podle normy IEC 61 131-3 jsou ***proměnné*** v podstatě prostředkem pro identifikaci datových objektů, jejichž obsah se může měnit, tzn. dat přiřazených ke vstupům, výstupům nebo paměti PLC. Proměnná může být deklarována některým z elementárních datových typů nebo některým z odvozených (uživatelských) datových typů.

Tím se programování podle IEC 61 131-3 přiblížilo k běžným zvyklostem. Místo dříve používaných hardwarových adres nebo symbolů jsou zde definovány proměnné tak, jak se používají ve vyšších programovacích jazycích. Proměnné jsou identifikátory (jména) přiřazené programátorem, které slouží v podstatě pro rezervaci místa v paměti a obsahují hodnoty dat programu.

3.3.1 Deklarace proměnných

Každá deklarace POU (tzn. každá deklarace programu, funkce nebo funkčního bloku) má mít na začátku alespoň jednu deklarační část, která specifikuje datové typy proměnných používaných v POU. Tato deklarační část má textovou podobu a používá jedno z klíčových slov **`VAR`**, **`VAR_TEMP`**, **`VAR_INPUT`**, **`VAR_OUTPUT`**. Za klíčovým slovem **`VAR`** může být volitelně uveden kvalifikátor **`CONSTANT`**. Za uvedenými klíčovými slovy následuje jedna nebo více deklarací proměnných oddělených středníkem a ukončených klíčovým slovem **`END_VAR`**. Součástí deklarace proměnných může být deklarace jejich počátečních (inicializačních) hodnot.



Obr. 3.4 Deklarace proměnné podle IEC

Rozsah platnosti deklarací umístěných v deklarační části POU je ***lokální*** pro tu programovou organizační jednotku, ve které je deklarace uvedena. To znamená, že deklarované proměnné nebudou přístupné ostatním POU kromě explicitního předávání parametrů přes proměnné, které byly deklarovány jako ***vstupní proměnné*** (**`VAR_INPUT`**) resp. ***výstupní proměnné*** (**`VAR_OUTPUT`**). Jedinou výjimkou z tohoto pravidla jsou proměnné, které byly deklarovány jako ***globální***. Tyto proměnné jsou definovány vně deklarací všech POU a začínají klíčovým slovem **`VAR_GLOBAL`**. Za klíčovým slovem **`VAR_GLOBAL`** může být volitelně uveden kvalifikátor **`RETAIN`** nebo **`CONSTANT`**.

3.3.1.1 Třídy proměnných

Třída proměnné určuje její použití a rozsah platnosti (scope). Z tohoto pohledu lze proměnné rozdělit do následujících skupin:

- **globální proměnné**
  - **VAR_GLOBAL** - nezálohované proměnné
  - **VAR_GLOBAL RETAIN** - zálohované proměnné
  - **VAR_GLOBAL CONSTANT** - konstanty
  - **VAR_EXTERNAL** - externí proměnné
- **lokální proměnné**
  - **VAR** - lokální proměnné
  - **VAR_TEMP** - přechodné proměnné
- **proměnné pro předávání parametrů**
  - **VAR_INPUT** - vstupní proměnné
  - **VAR_OUTPUT** - výstupní proměnné
  - **VAR_IN_OUT** - vstup-výstupní

Tab.3.10 Třídy proměnných

Třída proměnné	Význam	Určení
VAR_INPUT	vstupní	Pro předávání vstupních parametrů do POU Tyto proměnné jsou viditelné z ostatních POU a jsou z nich také nastavovány
VAR_OUTPUT	výstupní	Pro předávání výstupních parametrů z POU Tyto proměnné jsou viditelné z ostatních POU, kde je možné provádět pouze jejich čtení Změnu hodnoty těchto proměnných lze provádět pouze v rámci POU, ve které byly proměnné deklarovány
VAR_IN_OUT	vstup / výstupní	Pro nepřímý přístup k proměnným ležícím vně POU Proměnné lze číst i měnit jejich hodnotu uvnitř i vně POU
VAR_EXTERNAL	globální	Proměnné definované v mnemokódu PLC
VAR_GLOBAL	globální	Proměnné, které jsou dostupné ze všech POU
VAR	lokální	Pomocné proměnné používané v rámci POU Z ostatních POU nejsou viditelné, to znamená, že je lze číst resp. měnit jejich hodnotu pouze v rámci POU, ve které jsou deklarovány Tyto proměnné mohou uchovávat hodnotu i mezi jednotlivými voláními příslušné POU
VAR_TEMP	lokální	Pomocné proměnné používané v rámci POU Z ostatních POU nejsou viditelné Tyto proměnné vznikají při vstupu do POU a zanikají po ukončení POU – nemohou tedy uchovávat hodnotu mezi dvěma voláními POU

Tab.3.11 Použití tříd v jednotlivých POU

Třída proměnné	PROGRAM	FUNCTION_BLOCK	FUNCTION	Vně POU
VAR_INPUT	ano	ano	ano	ne
VAR_OUTPUT	ano	ano	ne	ne
VAR_IN_OUT	ano	ano	ano	ne
VAR_EXTERNAL	ano	ano	ano	ne
VAR_GLOBAL	ne	ne	ne	ano
VAR	ano	ano	ano	ne
VAR_TEMP	ano	ano	ano	ne

3.3.1.2 Kvalifikátory v deklaraci proměnných

Kvalifikátory umožňují definovat dodatečné vlastnosti deklarovaných proměnných. Klíčové slovo pro kvalifikátor se uvádí za klíčovým slovem třídy (`VAR,`...). V deklaraci proměnných lze použít následující kvalifikátory :

- `RETAIN` – zálohované proměnné (proměnné, které uchovávají hodnotu i během vypnutí napájení PLC)
- `CONSTANT` – konstantní hodnota (hodnota proměnné nemůže být změněna)
- `R_EDGE` – náběžná hrana proměnné
- `F_EDGE` – sestupná hrana proměnné

Tab.3.12 Použití kvalifikátorů v deklaraci proměnných

Třída proměnné	Význam	RETAIN	CONSTANT	R_EDGE F_EDGE
VAR	lokální	ne	ano	ne
VAR_INPUT	vstupní	ne	ne	ano
VAR_OUTPUT	výstupní	ne	ne	ne
VAR_IN_OUT	vstup / výstupní	ne	ne	ne
VAR_EXTERNAL	globální	ne	ne	ne
VAR_GLOBAL	globální	ano	ano	ne
VAR_TEMP	lokální	ne	ne	ne

3.3.2 Globální proměnné

Z hlediska dostupnosti lze proměnné rozdělit na ***globální*** a ***lokální.***

Globální proměnné jsou takové proměnné, které jsou dostupné ze všech POU. Jejich definice začíná klíčovým slovem **VAR_GLOBAL** a není uvedena uvnitř žádné POU jak ukazuje příklad 3.14. Globální proměnná může být umístěna na konkrétní adresu v paměti PLC pomocí klíčového slova **AT** v deklaraci proměnné. Pokud klíčové slovo **AT** chybí, přidělí potřebné místo v paměti překladač automaticky.

Pokud je v deklaraci uveden kvalifikátor **CONSTANT** jde o definici proměnných, jejichž hodnota je pevně dána deklarací a nelze jí v programem měnit. Takže to vlastně nejsou proměnné v pravém slova smyslu nýbrž konstanty. A pokud jsou navíc elementárního datového typu, překladač jim nepřiděluje žádné místo v paměti, pouze ve výrazech použije příslušnou konstantu.

Proměnné třídy **VAR_EXTERNAL** mohou být jak globální tak lokální. Jestliže je deklarace proměnných této třídy uvedena uvnitř POU, jedná se o proměnné lokální, v opačném případě jde o proměnné globální.

Příklad 3.14 Deklarace globálních proměnných

program v mnemokódu:

```
#reg word mask                      ; deklarace promenne v mnemokodu
P 0
        ld      $1111
        wr      mask
E 0
```

program v jazyce ST:

```
VAR_EXTERNAL
  mask            : WORD;           // odkaz na promennou v mnemokodu
END_VAR

VAR_GLOBAL RETAIN
  maxTemp         : REAL;           // zalohovana promenna
END_VAR

VAR_GLOBAL CONSTANT
  PI       : REAL := 3.14159;       // konstanta
END_VAR

VAR_GLOBAL
  globalFlag      : BOOL;
  suma            : DINT := 0;
  temp   AT %XF10 : REAL;          // temperature
  minute AT %S7   : USINT;
END_VAR

PROGRAM ExampleGlobal

  globalFlag := mask = 16#1111;                 // true
  maxTemp := MAX(IN1 := temp, IN2 := maxTemp);
END_PROGRAM
```

3.3.3 Lokální proměnné

Lokální proměnné jsou deklarovány uvnitř POU a jejich platnost a viditelnost je omezena na tu POU, ve které jsou deklarovány. Z ostaních POU je není možné používat. Deklarace lokálních proměnných začíná klíčovým slovem **VAR** nebo **VAR_TEMP**.

Proměnné deklarované v třídě **VAR** jsou tzv. statické proměnné. Těmto proměnným je při překladu přiděleno pevné místo v paměti proměnných, které se během provádění programu nemění. Což znamená, že čím více proměnných ve třídě **VAR** nadefinujeme, tím více paměti bude obsazeno. Další důležitou vlastností proměnných třídy **VAR** je, že se jejich hodnota zachovává mezi dvěmi voláními POU, ve které jsou deklarovány.

Proměnné deklarované ve třídě **VAR_TEMP** jsou proměnné, které jsou dynamicky vytvářeny v okamžiku, kdy se zahajuje výpočet POU s příslušnou deklarací. V okamžiku, kdy výpočet POU končí, je dynamicky přidělená paměť uvolněna a proměnné třídy **VAR_TEMP** zanikají. Z toho vyplývá, že deklarace proměnných ve třídě **VAR_TEMP** neovlivňuje spotřebu paměti. Tyto proměnné také nemohou uchovávat hodnotu mezi dvěma voláními POU, neboť po ukončení POU přestávají existovat.

Rozdíl mezi proměnnými tříd **VAR** a **VAR_TEMP** je také v jejich inicializaci. Proměnné třídy **VAR** jsou inicializovány pouze při restartu systému zatímco proměnné třídy **VAR_TEMP** jsou inicializovány pokaždé, když je jim přidělována paměť (tj. při každém zahájení výpočtu příslušné POU). Uvedené vlastnosti ukazuje následující příklad.

Příklad 3.15 Deklarace lokálních proměnných

```
PROGRAM ExampleLocal
  VAR
    staticCounter    : UINT;
    staticVector     : ARRAY[1..100] OF BYTE;
  END_VAR
  VAR_TEMP
    tempCounter      : UINT;
    tempVector       : ARRAY[1..100] OF BYTE;
  END_VAR

  staticCounter := staticCounter + 1;
  tempCounter   := tempCounter   + 1;
END_PROGRAM
```

Hodnota lokální proměnné **staticCounter** se bude při opakovaném volání programu **ExampleLocal** plynule zvyšovat, protože každé další volání zahájí výpočet s hodnotou **staticCounter** z minulého volání. Naproti tomu hodnota proměnné **tempCounter** bude na konci programu **ExampleLocal** vždy 1 nezávisle na počtu volání programu, protože tato proměnná vznikne a je inicializovaná hodnotou 0 vždy při vyvolání programu **ExampleLocal**.

Na příkladu 3.15 lze také ukázat rozdíly ve spotřebě paměti proměnných. Proměnná **staticVector** zabere 100 bytů v paměti proměnných zatímco proměnná **tempVector** velikost obsazené paměti proměnných vůbec neovlivní.

3.3.4 Vstupní a výstupní proměnné

Vstupní a výstupní proměnné slouží pro předávání parametrů mezi POU. Pomocí těchto proměnných tedy můžeme definovat vstupní a výstupní rozhraní (interface) POU.

Pro předávání parametrů směrem do POU slouží proměnné třídy **VAR_INPUT** a jedná se tedy o **vstupní** proměnné. Pro předávání parametrů směrem z POU slouží proměnné třídy **VAR_OUTPUT** a jedná se o **výstupní** proměnné. Představíme-li si např. **funkční blok** jako integrovaný obvod, pak proměnné **VAR_INPUT** představují vstupní signály obvodu a proměnné **VAR_OUTPUT** představují jeho výstupní signály.

Definice proměnných typu **BOOL** ve třídě **VAR_INPUT** může být doplněna kvalifikátory **R_EDGE** a **F_EDGE**, které umožňují detekovat náběžnou respektive sestupnou hranu proměnné. Proměnné definované s kvalifikátorem **R_EDGE** budou nabývat hodnoty **true** pouze v případě, kdy se stav proměnné mění z hodnoty **false** na hodnotu **true**. Takovou proměnnou je i proměnná **in** v příkladu 3.16. Funkční blok **FB_EdgeCounter** v tomto příkladu bude tedy čítat náběžné hrany (změny z hodnoty false na hodnotu true) vstupní proměnné **in**.

Příklad 3.16 Detekce náběžné hrany vstupní proměnné

```
FUNCTION_BLOCK FB_EdgeCounter
  VAR_INPUT
    in          : BOOL R_EDGE;
  END_VAR
  VAR_OUTPUT
    count       : UDINT;
  END_VAR

  IF in THEN count := count + 1; END_IF;
END_FUNCTION_BLOCK


PROGRAM ExampleInputEdge
  VAR_EXTERNAL
    AT  %X0.0   : BOOL;
  END_VAR
  VAR
    edgeCounter : FB_EdgeCounter;
    howMany     : UDINT;
  END_VAR

  edgeCounter(in := %X0.0, count => howMany);
END_PROGRAM
```

Parametry předávané prostřednictvím vstupních a výstupních proměnných jsou předávány hodnotou. Jinými slovy to znamená, že při volání POU je třeba předat hodnoty vstupních proměnných. Po návratu z POU je pak potřebné předat hodnoty výstupních proměnných.

Proměnné třídy **VAR_IN_OUT** mohou sloužit jako vstupní i jako výstupní zároveň. Parametry předávané POU prostřednictvím proměnných třídy **VAR_IN_OUT** nejsou předávány hodnotou, ale referencí. To znamená, že se při volání POU předává adresa proměnné místo její hodnoty, což umožňuje použít proměnnou podle potřeby jako vstupní nebo jako výstupní.

Rozdíl mezi předáváním parametrů hodnotou a referencí je vidět v příkladu 3.17.

Příklad 3.17 Rozdíl v použití proměnných VAR_INPUT a VAR_IN_OUT

```
TYPE
  TMyUsintArray : ARRAY[1..100] OF USINT;
END_TYPE

FUNCTION Suma1   : USINT
  VAR_INPUT
    vector       : TMyUsintArray;
    length       : INT;
  END_VAR
  VAR
    i            : INT;
    tmp          : USINT := 0;
  END_VAR

  FOR i := 1 TO length DO tmp := tmp + vector[i]; END_FOR;
  Suma1 := tmp;
END_FUNCTION

FUNCTION Suma2   : USINT
  VAR_IN_OUT
    vector       : TMyUsintArray;
  END_VAR
  VAR_INPUT
    length       : INT;
  END_VAR
  VAR
    i            : INT;
    tmp          : USINT := 0;
  END_VAR

  FOR i := 1 TO length DO tmp := tmp + vector[i]; END_FOR;
  Suma2 := tmp;
END_FUNCTION

PROGRAM ExampleVarInOut
  VAR
    buffer       : TMyUsintArray := [1,2,3,4,5,6,7,8,9,10];
    result1,
    result2      : USINT;
  END_VAR

  result1 := Suma1( buffer, 10);         // 55
  result2 := Suma2( buffer, 10);         // 55

END_PROGRAM
```

Zadáním tohoto příkladu bylo vytvořit funkci, která spočítá součet zadaného počtu prvků pole typu **USINT**.

Funkce **Suma1** používá pro vstupní proměnnou **vector** třídu **VAR_INPUT**, což znamená, že se při volání této funkce musí předat hodnoty všech prvků pole **buffer** do vstupní proměnné **vector**. V tomto případě to znamená 100 bytů dat. Výpočet pak probíhá nad proměnnou **vector**.

Funkce **Suma2** má proměnnou **vector** definovanou ve třídě **VAR_IN_OUT** a tak se při volání této funkce předává adresa proměnné **buffer** místo hodnot všech prvků. To jsou pouze 4 byty oproti 100 bytům v prvním případě. Vstupní proměnná **vector** tedy obsahuje adresu proměnné **buffer** a výpočet pak probíhá nad proměnnou **buffer**, která je přes proměnnou **vector** nepřímo adresovaná.

3.3.5 Jednoduché a složené proměnné

Z hlediska datového typu lze proměnné rozdělit na ***jednoduché*** a ***složené***. Jednoduché proměnné jsou proměnné základního typu. Složené proměnné jsou typu ***pole*** nebo ***struktury,*** případně jejich kombinace. Norma IEC 61 131-3 označuje tyto proměnné jako víceprvkové proměnné (multi-element variables).

3.3.5.1 Jednoduché proměnné

Jednoduchá proměnná je definována jako proměnná, která reprezentuje jednoduchý datový prvek jednoho z elementárních datových typů nebo uživatelského datového typu (výčet hodnot nebo typ odvozený rekurzivně tak, že se lze zpětně postupně dopracovat opět až k výčtu hodnot nebo nebo elementárním datovým typům). Ukázky jednoduchých proměnných jsou uvedeny v příkladu 3.18.

Příklad 3.18 Jednoduché proměnné

```
TYPE
  TColor        : (white, red, gree, black);
  TMyInt        : INT := 100;
END_TYPE

VAR_GLOBAL
  basicColor    : TColor := red;
  lunchTime     : TIME   := TIME#12:00:00;
END_VAR

PROGRAM ExapleSimpleVar
  VAR
    tmpBool     : BOOL;
    count1      : INT;
    count2      : TMyInt;
    currentTime : TIME;
  END_VAR
  VAR_TEMP
    count3      : REAL := 100.0;
  END_VAR

END_PROGRAM
```

3.3.5.2 Pole

Pole je soubor datových prvků stejného datového typu, na které je možné se odkázat pomocí jednoho nebo více indexů uzavřených v závorkách a oddělených čárkami. Index musí být některým z typů zahrnutých v rodovém typu **ANY_INT**. Maximální počet indexů (rozměr pole) je 4 a maximální rozsah indexů musí odpovídat typu **INT**.

Proměnnou typu pole lze definovat dvojím způsobem. Buď je nejprve definován odvozený datový typ pole a poté je založena proměnná tohoto typu. To je například proměnná **rxMessage** v příkladu 3.19. Nebo lze pole definovat přímo v deklaraci proměnné, viz proměnná **sintArray** ve stejném příkladu. Pro oba způsoby deklarace pole platí pravidla uvedená v kapitole 3.2.3.2. Rovněž způsob zápisu inicializačních hodnot je shodný.

Příklad 3.19 Pole proměnných

```
TYPE
  TMessage    : ARRAY[0..99] OF BYTE;
END_TYPE

VAR_GLOBAL
  delay       : ARRAY [1..5] OF TIME := [ TIME#1h,
                                          T#10ms,
                                          time#3h_20m_15s,
                                          t#15h5m10ms,
                                          T#3d];
END_VAR

PROGRAM ExampleArrayVar
  VAR
    rxMessage   : TMessage;
    txMessage   : TMessage;
    sintArray   : ARRAY [1..2,1..4] OF SINT := [ 11, 12, 13, 14,
                                                 21, 22, 23, 24 ];
  END_VAR
  VAR_TEMP
    pause       : TIME;
    element     : SINT;
  END_VAR

   pause   := delay[3];              // 3h 20m 15s
   element := sintArray[2, 3];       // 23
END_PROGRAM
```

3.3.5.3 Struktury

Strukturovaná proměnná je proměnná, která je deklarována s typem, který byl předtím specifikován jako struktura dat, tj. datovým typem složeným ze souboru pojmenovaných prvků. Deklarace odvozeného typu struktura je popsaná v kapitole 3.2.3.3.

Přímá deklarace struktury v deklaraci proměnné není podporována.

V příkladu 3.20 je definovaná proměnná **`presure`**, která typu struktura **`Tmeasure`**. V definici tohoto typu je pak použita další struktura **`Tlimit`**. Příklad ukazuje také inicializaci všech prvků proměnné **`presure`**, včetně vnořených struktur. Zároveň je v příkladu vidět jakým způsobem se v programu přistupuje na jednolivé prvky strukturované proměnné (např. **`presure.lim.low`**). Konstrukce **`AT %XF10`** je vysvětlena v následující kapitole.

Příklad 3.20 Strukturované proměnné

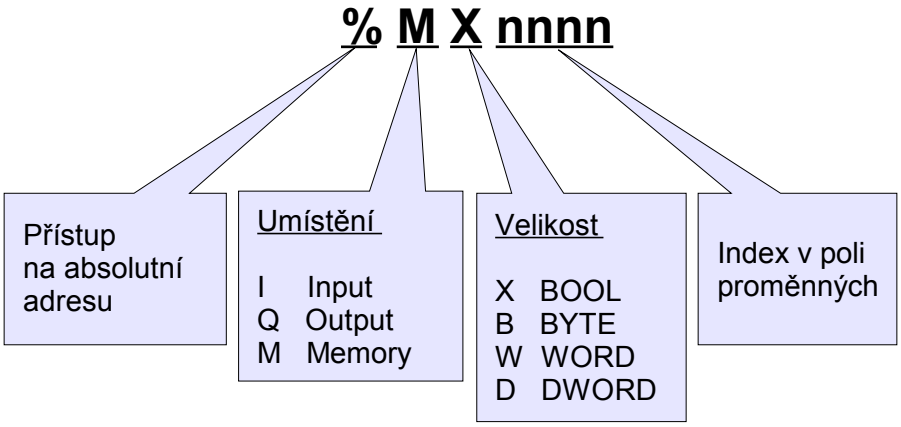
```
TYPE
  TLimit :
    STRUCT
      low, high : REAL;
    END_STRUCT;

  TMeasure :
    STRUCT
      lim       : TLimit;
      value     : REAL;
      failure   : BOOL;
    END_STRUCT;
END_TYPE

VAR_GLOBAL
  AT      %XF10 : REAL;
  presure       : TMeasure := ( lim     := ( low := 10, high := 100.0),
                                value   := 0,
                                failure := false);
END_VAR

PROGRAM ExampleStructVar

  presure.value := %XF10;                 // input sensor
  IF presure.value < presure.lim.low  OR
     presure.value > presure.lim.high
  THEN
    presure.failure := TRUE;
  ELSE
    presure.failure := FALSE;
  END_IF;
END_PROGRAM
```

3.3.6 Umístění proměnných v paměti PLC

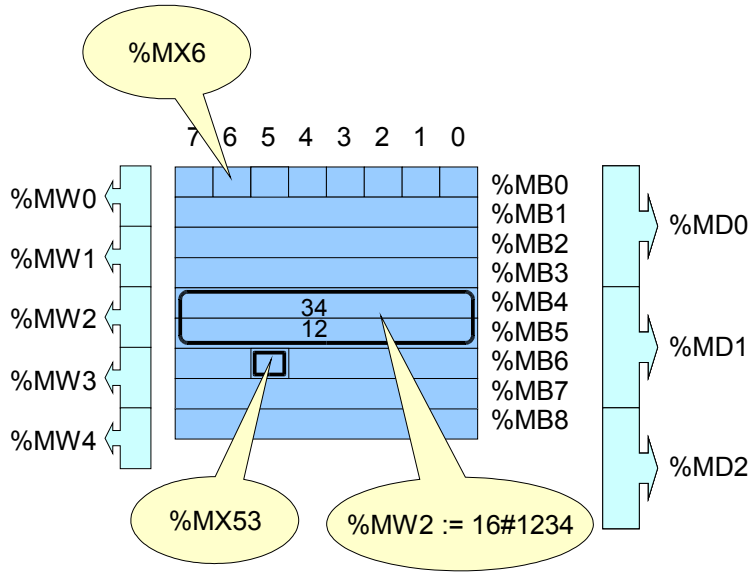
Umístění proměnných v paměti PLC provádí automaticky překladač. Pokud je z nějakého důvodu nutné umístit proměnnou na konkrétní adresu, lze to specifikovat v deklaraci proměnné pomocí klíčového slova **AT**, za kterým následuje zápis adresy proměnné.

Zápis adresy proměnné

Pro zápis adresy proměnných se používá speciální znak procento, „**%**“, ***prefix umístění*** (Location prefix) a ***prefix šíře dat*** (Size prefix). Za těmito znaky následuje jeden nebo více znaků typu UINT oddělených tečkami.

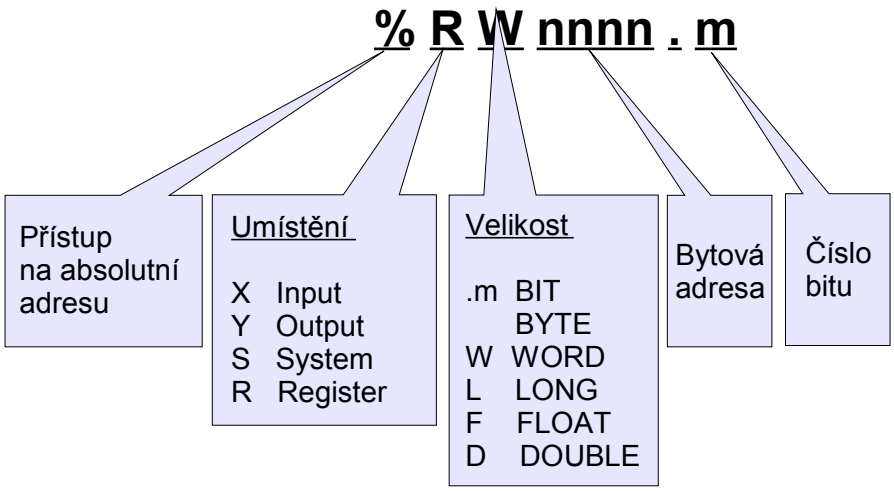


Obr. 3.5 Přímá adresa v paměti PLC podle IEC

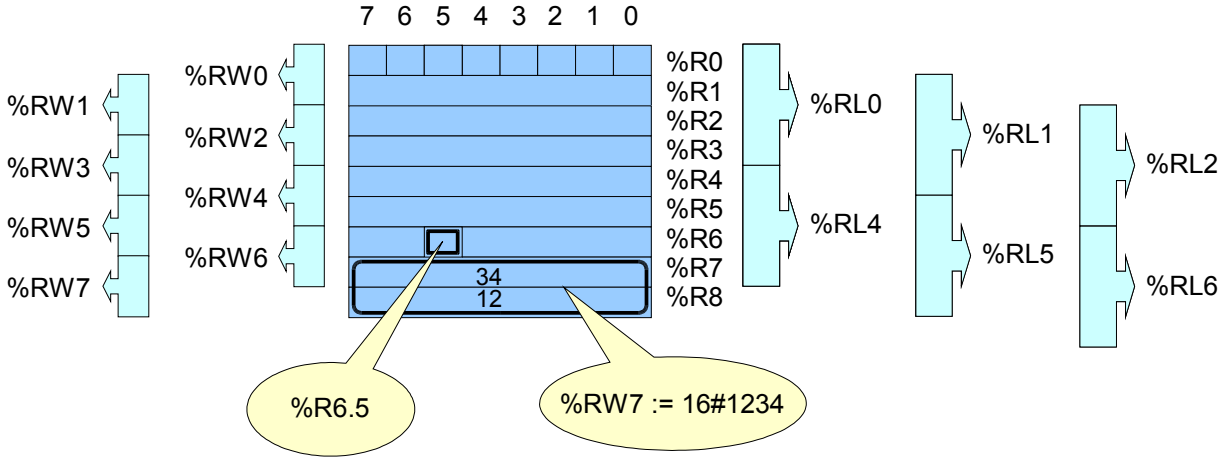


Obr. 3.6 Značení paměti PLC podle IEC

Přímé adresy v programech pro PLC lze také zapisovat způsobem tradičně používaným v prostředí Mosaic. Překladač automaticky rozpozná, který ze způsobů zápisu přímé adresy byl použit.



Obr. 3.7 Tradiční zápis přímé adresy proměnné v PLC



Obr. 3.8 Tradiční značení paměti PLC v prostředí Mosaic

Zápisy **`%MB10`** (podle IEC) a **`%R10`** (tradiční) tedy označují stejné místo v paměti. Zápis **`%RW152.9`** označuje devátý bit proměnné velikosti **`WORD`**, která je uložena v paměti od adresy **`152`**. U proměnných, které v paměti zabírají více než jeden byte, je na nejnižší adrese ukládán významově nejnižší byte, významově nejvyšší byte pak leží na nejvyšší adrese (Little Endian).

Specifikaci přímé adresy v deklaraci proměnné lze použít pouze ve třídách **VAR_GLOBAL** a **VAR_EXTERNAL**. Klíčové slovo **AT**, které uvozuje přímou adresu proměnné, se zapisuje mezi jméno proměnné a specifikaci datového typu.

Proměnné, které mají v deklaraci uvedenou pouze přímou adresu bez jména proměnné se nazývají přímo reprezentované proměnné. Při přístupu na tyto proměnné v programu se pak místo jména proměnné použije její adresa. To případ proměnných **%MB121** a **%R122** v příkladu 3.21.

Příklad 3.21 Specifikace adresy v deklaraci proměnné

```
VAR_GLOBAL
  SymbolicVar AT %MB120 : USINT;
              AT %MB121 : USINT;
              AT %R122  : USINT := 242;
  counterOut  AT %Y0.0  : BOOL;              // PLC output
END_VAR

PROGRAM ExampleDirectVar
  VAR_EXTERNAL
    AT %S6              : USINT;             // second counter
    AT %X0.0, AT %X0.1  : BOOL;              // PLC input
  END_VAR
  VAR
    counter : CTU;
  END_VAR

  SymbolicVar := %MB121 + %R122;
  counter(CU := %X0.0, R := %X0.1, PV := 100, Q => counterOut);
END_PROGRAM
```

Přímé adresy se používají pro deklaraci těch proměnných, jejichž umístění se nemá v průběhu úprav programu měnit. Příkladem mohou být proměnné určené pro vizualizační program nebo proměnné, které představují vstupy resp. výstupy PLC.

Pokud není adresa v deklaraci proměnné uvedena, tak umístění symbolické proměnné do paměti PLC (přiřazení adresy) provede překladač. Ten přitom také zajistí, aby se proměnné v paměti nepřekrývaly.

U přímo reprezentovaných proměnných rozhoduje o umístění proměnné v paměti programátor a ten také musí zajistit, aby nedošlo k nežádoucí kolizi proměnných (překrytí jejich adres v paměti).

3.3.7 Inicializace proměnných

Programovací model podle IEC 61 131 zajišťuje, že každá proměnná má po restartu řídícího systému přidělenou počáteční (inicializační) hodnotu. Tato hodnota může být:

- Hodnota, kterou měla proměnná v okamžiku zastavení výpočtu – typicky při výpadku napájení řídícího systému (retained value)
- Uživatelem specifikovaná počáteční hodnota (uvedená v deklaraci proměnné)
- Předdefinovaná (default) počáteční hodnota podle datového typu

Uživatel může deklarovat, že má být proměnná ***retentivní*** (zálohovaná, tzn. že se má uchovat její poslední hodnota) pomocí kvalifikátoru **RETAIN**. Tento kvalifikátor je možné použít pouze pro globální proměnné.

Inicializační hodnotu proměnné je možné specifikovat v rámci deklarace proměnné. Pokud není inicializační hodnota v deklaraci proměnné uvedena, bude proměnná inicializovaná počáteční hodnotou podle použitého datového typu.

Počáteční hodnota proměnné

Počáteční hodnota proměnné po restartu systému se určí podle těchto pravidel:

- Pokud je startovací operací tzv. teplý restart, pak počáteční hodnotou retentivních proměnných budou jejich retentivní (poslední zachované) hodnoty
- Pokud je startovací operací tzv. studený restart, pak počáteční hodnotou retentivních proměnných budou uživatelem specifikované počáteční hodnoty
- Neretentivní proměnné budou inicializovány na hodnoty specifikované uživatelem nebo na předdefinované hodnoty pro příslušné datové typy u všech proměnných, kde není počáteční hodnota uživatelem specifikována
- Proměnné, které reprezentují vstupy systému PLC, budou inicializovány podle stavu signálů, připojených na tyto vstupy
- Proměnné reprezentující výstupy systému budou inicializovány hodnotou 0, která odpovídá stavu „bez napětí“

U proměnných ve třídě **VAR_EXTERNAL** nemohou být počáteční hodnoty přiřazovány, protože se vlastně jedná o odkazy na proměnné, které jsou deklarovány jinde v programu. Inicializaci není možné deklarovat také u proměnných třídy **VAR_IN_OUT**, neboť tyto proměnné obsahují pouze pointery na proměnné nikoliv proměnné samotné.

Příklad 3.22 Inicializace proměnných

```
TYPE
  MY_REAL : REAL := 100.0;
END_TYPE

VAR_GLOBAL RETAIN
  remanentVar1 : BYTE;
  remanentVar2 : BYTE := 56;
END_VAR

PROGRAM ExampleInitVar
  VAR
    localVar1  : REAL;
    localVar2  : REAL := 12.5;
    localVar3  : MY_REAL;
  END_VAR
  VAR_TEMP
    tempVar1   : BYTE;
    tempVar2   : REAL;
  END_VAR

    tempVar1   := remanentVar1 AND remanentVar2;
    tempVar2   := localVar1 + localVar2;
END_PROGRAM
```

Při studeném restartu systému bude mít zálohovaná proměnná **remanentVar1** inicializační hodnotu **0** podle default inicializační hodnoty datového typu **BYTE**. Proměnná **remanentVar2** bude mít inicializační hodnotu 56, protože tato hodnota je předepsaná v deklaraci proměnné.

Při teplém restartu systému budou mít proměnné **remanentVar1** a **remanentVar2** takové hodnoty, jaké měly tyto proměnné při vypnutí napájení systému.

Proměnná **localVar1** bude nezávisle na typu restartu inicializovaná hodnotou **0.0**, neboť v deklaraci proměnné není inicializační hodnota uvedena a tak se použije předdefinovaná inicializační hodnota datového typu **REAL**. Proměnná **localVar2** bude po restartu vždy inicializovaná hodnotou **12.5**. Proměnná **localVar3** bude po restartu inicializovaná hodnotou **100.0**, protože to je inicializační hodnota odvozeného datového typu **MY_REAL**.

3.4 Programové organizační jednotky

Programové organizační jednotky (Program Organization Units, POUs) jsou ***funkce***, ***funkční bloky*** a ***programy***. Mohou být dodány od výrobce nebo je může napsat uživatel.

Programové organizační jednotky ***nejsou rekurzivní***, tzn. že vyvolání jedné programové organizační jednotky nesmí způsobit vyvolání jiné programové organizační jednotky stejného typu! Zjednodušeně lze říci, že POU nemůže volat sama sebe.

3.4.1 Funkce

Pro účely programovacích jazyků pro PLC je funkce definována jako programová organizační jednotka, která po provedení vygeneruje vždy jeden datový element (může být složen z více hodnot, jako je např. pole nebo struktura). Volání funkce se může použít v textových jazycích jako operand ve výrazu.

Funkce neobsahují žádnou vnitřní stavovou informaci, tzn. že volání funkce se stejnými argumenty (vstupními parametry) vytvoří vždycky stejné hodnoty (výstup).

Deklarace funkce

Textová deklarace funkce se skládá z těchto prvků:

- Klíčového slova **`FUNCTION`**, za kterým je uvedeno jméno deklarované funkce, dvojtečka a datový typ hodnoty, kterou bude funkce vracet
- Definice vstupních proměnných **`VAR_INPUT`**, kde jsou uvedeny specifikace jmen a typů vstupních parametrů funkce
- Definice lokálních proměnných **`VAR`** případně **`VAR_TEMP`**, kde jsou uvedeny specifikace jmen a typů vnitřních proměnných funkce
- Definice konstant **`VAR CONSTANT`**
- Tělo funkce (Function body) zapsané v některém z jazyků IL, ST, LD nebo FBD. Tělo funkce specifikuje operace, které se mají provádět nad vstupními parametry za účelem přiřazení jedné nebo více hodnot proměnné, která má stejné jméno, jako má funkce, a která reprezentuje návratovou hodnotu funkce
- Závěrečného klíčového slova **`END_FUNCTION`**

Volání funkce

V jazyce ST lze funkci vyvolat zápisem jména funkce následovaným předávanými parametry v kulatých závorkách. Počet a datový typ předávaných parametrů musí odpovídat vstupním proměnným v definici funkce. Pokud ve volání nejsou uvedeny názvy vstupních proměnných funkce, pak pořadí parametrů musí přesně odpovídat pořadí vstupních proměnných v definici funkce. Pokud jsou parametry přiřazeny ke jménům vstupních parametrů (formal call), pak pořadí parametrů při volání funkce nehraje roli.

Příklad 3.23 Definice funkce a její volání v jazyce ST

```
FUNCTION MyFunction : REAL
  VAR_INPUT
    r, h    : REAL;
  END_VAR
  VAR CONSTANT
    PI      : REAL := 3.14159;
  END_VAR

  IF r > 0.0 AND h > 0.0
  THEN MyFunction := PI * r**2 * h;
  ELSE MyFunction := 0.0;
  END_IF;
END_FUNCTION


PROGRAM ExampleFunction
  VAR
    v1, v2  : REAL;
  END_VAR

  v1 := MyFunction( h := 2.0, r := 1.0);
  v2 := MyFunction( 1.0, 2.0);
END_PROGRAM
```

Funkce **MyFunction** v příkladu 3.23 má definovány dvě vstupní proměnné **r** a **h** typu **REAL**. Návratová hodnota této funkce je typu **REAL** a je reprezentována jménem **MyFunction**. Ve volání této funkce **v1 := MyFunction(h := 2.0, r := 1.0)** jsou uvedena jména vstupních proměnných. V tomto případě nezáleží na pořadí vstupních parametrů v závorkách. Volání **v2 := MyFunction(1.0, 2.0)** jména vstupních proměnných neobsahuje a tak se vstupní parametry předpokládají v takovém pořadí, v jakém jsou definovány vstupní proměnné v deklaraci funkce. Obě volání v uvedeném příkladu jsou tedy rovnocenná a dávají shodný výsledek.

3.4.1.1 Standardní funkce

Standardní funkce použitelné ve všech programovacích jazycích pro PLC jsou podrobně definovány v normě IEC 61 131-3 v kapitole 2.5.1.5. Souhrn standardních funkcí, které jsou podporovány překladačem v prostředí Mosaic, je uveden v této kapitole.

Přetížení funkce (Overloading)

O funkci nebo operaci říkáme, že je ***přetížená*** (*overloaded)*, pokud může pracovat nad prvky vstupních dat různých typů v rámci rodového jména typu. Např. přetížená funkce sčítání rodového typu **`ANY_NUM`** může pracovat nad datovými typy **`LREAL`**, **`REAL`**, **`DINT`**, **`INT`** a **`SINT`**. Pokud PLC systém podporuje přetíženou funkci nebo operaci, pak se může daná funkce aplikovat na všechny datové typy daného rodového typu, které jsou systémem podporovány.

Informace o tom, které funkce jsou přetížené, jsou uvedeny dále.Uživatelsky definované funkce nemohou být přetížené.

Pokud jsou všechny formální vstupní parametry standardní funkce stejného rodového typu, potom i všechny aktuální parametry musí být stejného typu. Pokud je to nutné, mohou se použít za tímto účelem funkce pro *konverzi* typu. Výstupní hodnota funkce potom bude stejného typu jako aktuální vstupy.

Rozšiřitelné funkce (Extensible)

Některé standardní funkce jsou ***rozšiřitelné*** *(extensible)*, to znamená mohou mít proměnný počet vstupů. U těchto funkcí se předpokládá, že operace definované funkcí budou prováděny nad všemi aplikovanými vstupy. Pokud je funkce rozšiřitelná, maximální počet vstupů není omezen.

Rozdělení standardních funkcí

Standardní funkce jsou rozděleny do několika základních skupin :

- Funkce pro konverzi typu
- Numerické funkce
  - numerické funkce jedné proměnné
  - aritmetické funkce více proměnných
- Funkce nad řetězcem bitů
  - rotace bitů
  - boolovské funkce
- Funkce výběru
- Funkce porovnávání
- Funkce nad řetězcem znaků
- Funkce s typy datum a čas
- Funkce nad datovými typy „výčet“

Sloupec s názvem **Ovr** v následujících tabulkách udává, je-li funkce přetížená (overloaded). Sloupec s názvem **Ext** nese informaci o tom, je-li příslušná funkce rozšiřitelná (extensible). Přesná specifikace standardních funkcí viz příloha.

Tab.3.13 Standardní funkce, skupina konverze typu

Standardní funkce, skupina konverze typu					
Jméno funkce	**Datový typ vstupu**	**Datový typ výstupu**	**Popis funkce**	**Ovr**	**Ext**
…_TO_…	ANY	ANY	Konverze datového typu uvedeného na prvním místě na datový typ uvedený na druhém místě	ano	ne
TRUNC	ANY_REAL	ANY_INT	„Ořezávání“	ano	ne

Tab.3.14 Standardní funkce, skupina numerické funkce jedné proměnné

Standardní funkce, skupina numerické funkce jedné proměnné				
Jméno funkce	**Datový typ vstupu / výstupu**	**Popis funkce**	**Ovr**	**Ext**
ABS	ANY_NUM / ANY_NUM	Absolutní hodnota	ano	ne
SQRT	ANY_REAL / ANY_REAL	Odmocnina	ano	ne
LN	ANY_REAL / ANY_REAL	Přirozený logaritmus	ano	ne
LOG	ANY_REAL / ANY_REAL	Desítkový logaritmus	ano	ne
EXP	ANY_REAL / ANY_REAL	Přirozená exponenciální funkce	ano	ne
SIN	ANY_REAL / ANY_REAL	Sinus vstupního úhlu uvedeného v radiánech	ano	ne
COS	ANY_REAL / ANY_REAL	Kosinus vstupního úhlu uvedeného v radiánech	ano	ne
TAN	ANY_REAL / ANY_REAL	Tangens vstupního úhlu uvedeného v radiánech	ano	ne
ASIN	ANY_REAL / ANY_REAL	Arcus sinus	ano	ne
ACOS	ANY_REAL / ANY_REAL	Arcus kosinus	ano	ne
ATAN	ANY_REAL / ANY_REAL	Arcus tangens	ano	ne

Tab.3.15 Standardní funkce, skupina numerické funkce - aritmetické funkce více proměnných

Standardní funkce, skupina numerické funkce - aritmetické funkce více proměnných					
Jméno funkce	**Datový typ vstupu / výstupu**	**Symbol**	**Popis funkce**	**Ovr**	**Ext**
ADD	ANY_NUM, .. ANY_NUM / ANY_NUM	+	Součet OUT:=IN1+ IN2+…+INn	ano	ano
MUL	ANY_NUM, .. ANY_NUM / ANY_NUM	*	Součin OUT:=IN1* IN2*…*INn	ano	ano
SUB	ANY_NUM, ANY_NUM / ANY_NUM	-	Rozdíl OUT:=IN1-IN2	ano	ne
DIV	ANY_NUM, ANY_NUM / ANY_NUM	/	Podíl OUT:=IN1/IN2	ano	ne
MOD	ANY_NUM, ANY_NUM / ANY_NUM		Modulo OUT:=IN1 modulo IN2	ano	ne
EXPT	ANY_REAL, ANY_NUM / ANY_REAL	**	Umocnění OUT:=IN1**IN2	ano	ne
MOVE	ANY_NUM / ANY_NUM	:=	Přesunutí, přiřazení OUT:=IN	ano	ne

Tab.3.16 Standardní funkce, skupina funkce nad řetězcem bitů - rotace bitů

Standardní funkce, skupina funkce nad řetězcem bitů - rotace bitů				
Jméno funkce	**Datový typ vstupu / výstupu**	**Popis funkce**	**Ovr**	**Ext**
SHL	ANY_BIT, N / ANY_BIT	Posun vlevo OUT := IN posunutý vlevo o N bitů, zprava doplněno nulami	ano	ne
SHR	ANY_BIT, N / ANY_BIT	Posun vpravo OUT := IN posunutý vpravo o N bitů, zleva doplněno nulami	ano	ne
ROR	ANY_BIT, N / ANY_BIT	Rotace vpravo OUT := IN odrotovaný vpravo o N bitů, zprava doplněno o zleva odrotované bity	ano	ne
ROL	ANY_BIT, N / ANY_BIT	Rotace vlevo OUT := IN odrotovaný vlevo o N bitů, zleva doplněno o zprava odrotované bity	ano	ne

Tab.3.17 Standardní funkce, skupina funkce nad řetězcem bitů - boolovské funkce

Standardní funkce, skupina funkce nad řetězcem bitů - boolovské funkce					
Jméno funkce	**Datový typ vstupu / výstupu**	**Symbol**	**Popis funkce**	**Ovr**	**Ext**
AND	ANY_BIT, .. ANY_BIT / ANY_BIT	&	Log. součin, „a současně“, OUT:=IN1& IN2&…&INn	ano	ano
OR	ANY_BIT, .. ANY_BIT / ANY_BIT		Log. součet, „nebo“, inklu-zivní součet, OUT:=IN1 OR IN2 OR … OR INn	ano	ano
XOR	ANY_BIT, .. ANY_BIT / ANY_BIT		Výlučný součet, „buď a nebo“, exkluzivní součet OUT:=IN1 XOR IN2 XOR … XOR INn	ano	ano
NOT	ANY_BIT / ANY_BIT		Negace, „ne“, OUT:=NOT IN1	ano	ne

Tab.3.18 Standardní funkce, skupina funkce výběru

Standardní funkce, skupina funkce výběru				
Jméno funkce	**Datový typ vstupu / výstupu**	**Popis funkce**	**Ovr**	**Ext**
SEL	BOOL, ANY, ANY / ANY	Binární výběr OUT := IN0 if G = 0 OUT := IN1 if G = 1	ano	ne
MAX	ANY, .. ANY / ANY	Maximum OUT := MAX(IN1, IN2, .. INn)	ano	ano
MIN	ANY, .. ANY / ANY	Minimum OUT := MIN(IN1, IN2, .. INn)	ano	ano
LIMIT	MN, ANY, MX / ANY	Omezovač OUT := MIN(MAX(IN, MN), MX)	ano	ne

Tab.3.19 Standardní funkce - funkce porovnávání

Standardní funkce, skupina funkce porovnávání				
Jméno funkce	**Datový typ vstupu / výstupu**	**Popis funkce**	**Ovr**	**Ext**
GT	ANY, .. ANY / BOOL	Klesající sekvence OUT:=(IN1> IN2)& (IN2>IN3)&… &(INn-1>INn)	ano	ano
GE	ANY, .. ANY / BOOL	Monotónní sekvence směrem dolů OUT:=(IN1>= IN2)& (IN2>=IN3)&…&(INn-1>=INn)	ano	ano
EQ	ANY, .. ANY / BOOL	Rovnost OUT:=(IN1= IN2)& (IN2=IN3)&… &(INn-1=INn)	ano	ano
LE	ANY, .. ANY / BOOL	Monotónní sekvence směrem nahoru OUT:=(IN1<= IN2)& (IN2<=IN3)&…&(INn-1<=INn)	ano	ano
LT	ANY, .. ANY / BOOL	Vzrůstající sekvence OUT:=(IN1< IN2)& (IN2<IN3)&… &(INn-1<INn)	ano	ano
NE	ANY, ANY / BOOL	Nerovnost OUT := (IN1<>IN2)	ano	ne

Tab.3.20 Standardní funkce, skupina funkce nad řetězcem znaků

Standardní funkce, skupina funkce nad řetězcem znaků				
Jméno funkce	**Datový typ vstupu / výstupu**	**Popis funkce**	**Ovr**	**Ext**
LEN	STRING / INT	OUT := LEN(IN); Délka řetězce IN	ne	ne
LEFT	STRING, ANY_INT / STRING	OUT := LEFT(IN, L); Ze vstupního řetězce IN přesunout L znaků zleva do výstupního řetězce	ano	ne
RIGHT	STRING, ANY_INT / STRING	OUT := RIGHT(IN, L); Ze vstupního řetězce IN přesunout L znaků zprava do výstupního řetězce	ano	ne
MID	STRING, ANY_INT, ANY_INT / STRING	OUT := MID(IN, L, P); Ze vstupního řetězce IN přesunout od P-tého znaku L znaků do výstupního řetězce	ano	ne
CONCAT	STRING, …. STRING /STRING	OUT := CONCAT(IN1, IN2, ...); Připojení jednotlivých vstupních řetězců do výstupního řetězce	ne	ano
INSERT	STRING, STRING, ANY_INT / STRING	OUT := INSERT(IN1, IN2, P); Vložení řetězce IN2 do řetězce IN1 počínaje od P-té pozice	ano	ano
DELETE	STRING, ANY_INT, ANY_INT / STRING	OUT := DELETE(IN, L, P); Smazání L znaků z řetězce IN počínaje od P-té pozice	ano	ano
REPLACE	STRING, STRING, ANY_INT, ANY_INT / STRING	OUT := REPLACE(IN1, IN2, L, P); Náhrada L znaků řetězce IN1 znaky řetězce IN2, vkládání od P-tého místa	ano	ano
FIND	STRING, STRING / INT	OUT := FIND(IN1, IN2); Nalezení pozice prvního znaku prvního výskytu řetězce IN2 v řetězci IN1	ano	ano

Tab.3.21 Standardní funkce, skupina funkce s typy datum a čas

Standardní funkce, skupina funkce s typy datum a čas					
Jméno funkce	**IN1**	**IN2**	**OUT**	**Ovr**	**Ext**
ADD_TIME	TIME	TIME	TIME	ne	ne
ADD_TOD_TIME	TIME_OF_DAY	TIME	TIME_OF_DAY	ne	ne
ADD_DT_TIME	DATE_AND_TIME	TIME	DATE_AND_TIME	ne	ne
SUB_TIME	TIME	TIME	TIME	ne	ne
SUB_DATE_DATE	DATE	DATE	TIME	ne	ne
SUB_TOD_TIME	TIME_OF_DAY	TIME	TIME_OF_DAY	ne	ne
SUB_TOD_TOD	TIME_OF_DAY	TOD	TIME	ne	ne
SUB_DT_TIME	DATE_AND_TIME	TIME	DATE_AND_TIME	ne	ne
SUB_DT_DT	DATE_AND_TIME	DT	TIME		
MULTIME	TIME	ANY_NUM	TIME	ano	ne
DIVTIME	TIME	ANY_NUM	TIME	ano	ne
CONCAT_DATE_TOD	DATE	TIME_OF_ DAY	DATE_AND_TIME	ne	ne
Funkce konverze typu					
DATE_AND_TIME_TO_TIME_OF_DAY, DAT_TO_TIME DATE_AND_TIME_TO_DATE, DAT_TO_DATE					

TOD … TIME_OF_DATE
DT … DATE_AND_TIME

3.4.2 Funkční bloky

Při programování podle IEC 61 131-3 je ***funkční blok*** taková organizační jednotka programu, která po provedení vygeneruje jednu nebo více hodnot. Z funkčních bloků se dají vytvářet násobné pojmenované ***instance*** (kopie). Každá instance má přiřazený identifikátor (*jméno instance*) a datovou strukturu, která obsahuje její vstupní, vnitřní a výstupní proměnné. Všechny hodnoty proměnných v této datové struktuře se uchovávají od jednoho provedení funkčního bloku k dalšímu jeho provedení. Vyvolání jednoho funkčního bloku se stejnými argumenty (vstupními parametry), tedy nemusí vždy vést ke stejným výstupním hodnotám. Instance funkčního bloku se vytváří použitím deklarovaného typu funkčního bloku v rámci třídy **VAR** nebo **VAR_GLOBAL**.

Jakýkoli funkční blok, který již byl deklarován, může být znovu použit v deklaraci jiného funkčního bloku nebo programu.

Rozsah působnosti instance funkčního bloku je lokální pro tu programovou organizační jednotku, v níž je *instanciován*, (tj. kde je vytvořena jeho pojmenovaná kopie), pokud ovšem není deklarován jako globální.

Následující příklad ukazuje postup při deklaraci funkčního bloku, vytvoření jeho instance v programu a konečně jeho vyvolání (provedení).

Příklad 3.24 Funkční blok v jazyce ST

```
FUNCTION_BLOCK fbStartStop              // deklarace FB
  VAR_INPUT
    start       : BOOL R_EDGE;          // vstupni promenne
    stop        : BOOL R_EDGE;
  END_VAR
  VAR_OUTPUT
    output      : BOOL;                 // vystupni promenna
  END_VAR

  output := (output OR start) AND not stop;
END_FUNCTION_BLOCK


PROGRAM ExampleFB
  VAR
    StartStop   : fbStartStop;          // instance FB
    running     : BOOL;
  END_VAR

  // vyvolani instance funkcniho bloku
  StartStop( stop := FALSE, start := TRUE, output => running);

  // alternativni zpusob volani FB
  StartStop.start := TRUE;
  StartStop.stop  := FALSE;
  StartStop();
  running         := StartStop.output;

  // volani s nekompletnim seznamem parametru
  StartStop( start := TRUE);
  running := StartStop.output;

END_PROGRAM
```

Vstupní a výstupní proměnné instance funkčního bloku mohou být reprezentovány jako prvky datového typu struktura.

Pokud je instance funkčního bloku globální, tak může být také deklarována jako retentivní.V tomto případě platí pouze pro vnitřní a výstupní parametry funkčního bloku.

Zvenku instance jsou přístupné pouze vstupní a výstupní parametry funkčního bloku, tzn. že vnitřní proměnné funkčního bloku zůstávají uživateli funkčního bloku skryté. Přiřazení hodnoty zvenku do výstupní proměnné funkčního bloku není dovoleno, tuto hodnotu přiřazuje jenom zvnitřku sám funkční blok. Přiřazení hodnoty vstupu funkčního bloku je dovoleno kdekoliv v nadřízené POU (typicky je to součást volání funkčního bloku).

Deklarace funkčního bloku

- Oddělovací klíčová slova pro deklaraci funkčních bloků jsou **FUNCTION_BLOCK**… **END_FUNCTION_BLOCK**
- Funkční blok může mít více než jeden výstupní parametr, deklarovaný ve třídě **VAR_OUTPUT**
- Hodnoty proměnných, které jsou předávány funkčnímu bloku ve třídě **VAR_IN_OUT** nebo **VAR_EXTERNAL** mohou být modifikovány zvnitřku funkčního bloku.
- V deklaraci vstupních proměnných funkčního bloku mohou být použity kvalifikátory **R_EDGE** a **F_EDGE**. Tyto kvalifikátory označují funkci detekce hran na Boolovských vstupech. Tím je vyvolána implicitní deklarace funkčního bloku **R_TRIG** nebo **F_TRIG**.
- Konstrukce definovaná pro inicializaci funkcí se používá i pro deklaraci defaultních hodnot vstupů funkčního bloku a pro počáteční hodnoty jeho vnitřních a výstupních proměnných

Pomocí třídy **VAR_IN_OUT** mohou být do funkčního bloku předávány pouze proměnné (předávání instancí funkčních bloků není podporováno). Kaskádování proměnných **VAR_IN_OUT** je dovoleno.

3.4.2.1 Standardní funkční bloky

Standardní funkční bloky jsou podrobně definovány v normě IEC 61 131-3 v kapitole 2.5.2.3.

Standardní funkční bloky jsou rozděleny do následujících skupin (viz Tab.3.22)

- Bistabilní prvky
- Detekce hrany
- Čítače
- Časovače

Standardní funkční bloky jsou uloženy v knihovně StdLib_Vxx_*.mlb, kde Vxx je verze knihovny.

Tab.3.22 Přehled standardních funkčních bloků

Jméno standardního funkčního bloku	Jméno vstupního parametru	Jméno výstupního parametru	Popis
Bistabilní prvky (klopné obvody)			
SR	S1, R	Q1	dominantní nastavení (sepnutí)
RS	S, R1	Q1	dominantní mazání (vypnutí)
Detekce hrany			
R_TRIG	CLK	Q	detekce náběžné hrany
F_TRIG	CLK	Q	detekce sestupné hrany
Čítače			
CTU	CU, R, PV	Q, CV	dopředný čítač
CTD	CD, LD, PV	Q, CV	zpětný čítač
CTUD	CU, CD, R, LD, PV	QU, QD, CV	oboustranný (reverzibilní) čítač
Časovače			
TP	IN, PT	Q, ET	pulzní časovač
TON (T--0)	IN, PT	Q, ET	zpoždění náběžné hrany
TOF (0--T)	IN, PT	Q, ET	zpoždění sestupné hrany

Názvy, významy a datové typy proměnných používané u standardních funkčních bloků :

Název vstupu / výstupu	Význam	Datový typ
R	Mazací (resetovací) vstup	BOOL
S	Nastavovací (setovací) vstup	BOOL
R1	Dominantní mazací vstup	BOOL
S1	Dominantní nastavovací vstup	BOOL
Q	Výstup (standardní)	BOOL
Q1	Výstup (pouze u klopných obvodů)	BOOL
CLK	Hodinový (synchronizační) signál	BOOL
CU	Vstup pro dopředné čítání	BOOL
CD	Vstup pro zpětné čítání	BOOL
LD	Nastavení předvolby čítače	BOOL
PV	Předvolba čítače	INT
QD	Výstup (zpětného čítače)	BOOL
QU	Výstup (dopředného čítače)	BOOL
CV	Aktuální hodnota (čítače)	INT
IN	Vstup (časovače)	BOOL
PT	Předvolba časovače	TIME
ET	Aktuální hodnota časovače	TIME
PDT	Předvolba - datum a čas	DT
CDT	Aktuální hodnota - datum a čas	DT

3.4.3 Programy

Program je v normě IEC 61 131-1 definován jako „logický souhrn prvků programovacích jazyků a konstrukcí nutných pro zamýšlené zpracování signálů, které je vyžadováno pro řízení stroje nebo procesu systémem programovatelného automatu“.

Jinak řečeno funkce a funkční bloky lze přirovnat k podprogramům (subroutines) zatímco POU program je hlavní program (main program). Deklarace a používání *programů* je identické s deklarací a používáním funkčních bloků.

Odlišnosti v přístupu k programům oproti funkčním blokům:

- Klíčová slova vymezující deklaraci programu jsou **PROGRAM**….**END_PROGRAM**
- *Programy* mohou být instanciovány pouze v rámci *zdrojů (Resources)*, jak je uvedeno v kap. 3.5. Naproti tomu, *funkční bloky* mohou být instanciovány pouze v rámci *programů* nebo jiných *funkčních bloků*
- Programy mohou volat funkce a funkční bloky. Naopak volání programů z funkcí nebo fuknčních bloků není možné

Příklad 3.25 POU Program v jazyce ST

```
PROGRAM test
  VAR
    motor1 : fbMotor;
    motor2 : fbMotor;
  END_VAR

  motor1( startMotoru := sb1, stopMotoru := sb2,
          hvezda => km1, trojuhelnik => km2);
  motor2( startMotoru := sb3, stopMotoru := sb4,
          hvezda => km3, trojuhelnik => km4);

END_PROGRAM
```

Při psaní programu v jazyce ST je důležité si uvědomit, že POU **PROGRAM** je stejně jako funkční blok pouze „předpisem“, ve kterém je definována struktura dat a algoritmy, prováděné nad touto datovou strukturou. Pro vykonávání definovaného programu je potřebné založit jeho instanci a přiřadit (asociovat) program k některé ze stadradních úloh, ve které pak bude prováděn. Tyto úkony popisuje následující kapitola.

3.5 Konfigurační prvky

Konfigurační prvky popisují run-time vlastnosti programů a přiřazují provádění programů ke konkrétnímu hardwaru v PLC. Představují tak vrcholový předpis celého programu pro PLC.

Při programování PLC systémů se používají následující konfigurační prvky :

- ***Konfigurace*** *(Configuration)* – označuje konkrétní PLC systém, který bude provádět všechny naprogramované POU
- ***Zdroj*** *(Resource)* – označuje konkrétní procesorový modul v PLC, který zajistí provádění programu
- ***Úloha*** *(Task)* – přiřazuje úlohu (proces), v rámci kterého bude příslušná POU **PROGRAM** prováděná

Příklad 3.26

```
CONFIGURATION Plc1
  RESOURCE CPM
    TASK FreeWheeling(Number := 0);
    PROGRAM prg WITH FreeWheeling : test ();
  END_RESOURCE
END_CONFIGURATION
```

V programovacím prostředí Mosaic jsou všechny konfigurační prvky generovány automaticky po vyplnění konfiguračních dialogů.

3.5.1 Konfigurace

Konfigurace označuje PLC systém, který poskytne zdroje pro provádění uživatelského programu. Jinými slovy konfigurace označuje řídící systém, pro který je uživatelský program určen.

Deklarace konfigurace

- Klíčová slova vymezující *konfiguraci* jsou **CONFIGURATION….END_ CONFIGURATION**
- Za klíčovým slovem **CONFIGURATION** je uvedeno pojmenování konfigurace, v programovacím prostředí Mosaic jméno konfigurace odpovídá jménu projektu
- *Konfigurace* slouží jako rámec pro definici *Zdroje (Resource)*

Příklad 3.27

```
CONFIGURATION jméno_konfigurace

  // deklarace zdroje

END_CONFIGURATION
```

3.5.2 Zdroje

Zdroj definuje, který modul v rámci PLC poskytne výpočetní výkon pro vykonávání uživatelského programu. V PLC řady TC700 je to vždy procesorový modul systému.

Deklarace zdroje

- Klíčová slova vymezující *zdroj* jsou **RESOURCE**….**END_ RESOURCE**
- Za klíčovým slovem **RESOURCE** je uvedeno pojmenování zdroje, v programovacím prostředí Mosaic je toto jméno implicitně „CPM“
- *Zdroje (Resources)* mohou být deklarovány pouze v rámci *konfigurace*

Příklad 3.28

```
CONFIGURATION Plc1
  RESOURCE CPM

    // deklarace úloh

    // přiřazení programů do deklarovaných úloh

  END_RESOURCE
END_CONFIGURATION
```

3.5.3 Úlohy

Pro účely normy IEC 61 131-3 je *úloha* definována jako výkonný řídicí prvek, který je schopen vyvolávat buď periodicky nebo na základě výskytu vzestupné hrany specifikované boolovské proměnné provádění souboru programových organizačních jednotek (POUs). Těmito programovými organizačními jednotkami mohou být *programy* a v nich deklarované *funkční bloky.*

V prostředí Mosaic je pojem úloha totožný s tradičně používaným pojmem proces.

Deklarace úloh

- Klíčové slovo pro označení *úlohy* je **TASK**
- Za klíčovým slovem **TASK** je uvedeno jméno úlohy
- Za jménem úlohy jsou uvedeny vlastnosti úlohy, konkrétně číslo odpovídajícího procesu
- *Úlohy (Tasks)* mohou být deklarovány pouze v rámci deklarace *zdroje (Resource)*

Přiřazení programů k úlohám

- Přiřazení programu ke konkrétní úloze je uvozeno klíčovým slovem **PROGRAM**, kterým se zároveň automaticky zakládá instance uvedeného programu
- Za klíčovým slovem **PROGRAM** následuje jméno instance programu
- Klíčové slovo **WITH** uvozuje jméno úlohy, ke které bude program přiřazen
- Na závěr je za dvojtečkou uvedeno jméno asociovaného programu včetně specifikace vstupních a výstupních parametrů
- S jednou úlohou může být asociováno několik programů, pořadí jejich vykonávání v rámci úlohy pak odpovídá pořadí, v jakém byly asociovány
- Přiřazení programů může být deklarováno pouze v rámci deklarace *zdroje (Resource)*

Příklad 3.29

```
CONFIGURATION Plc1
  RESOURCE CPM
    TASK FreeWheeling(Number := 0);
    PROGRAM prg WITH FreeWheeling : test ();
  END_RESOURCE
END_CONFIGURATION
```

4 TEXTOVÉ JAZYKY

V normě IEC 61 131-3 jsou definovány dva textové jazyky: jazyk seznamu instrukcí (IL, Instruction List) a jazyk strukturovaného textu (ST, Structured Text). Oba tyto jazyky jsou překladačem v prostředí Mosaic podporovány.

4.1 Jazyk seznamu instrukcí IL

Jazyk seznamu instrukcí (Instruction List) je nízkoúrovňový jazyk typu assembler. Tento jazyk patří mezi řádkově orientované jazyky.

4.1.1 Instrukce v IL

Seznam instrukcí se skládá ze sekvence (posloupnosti) *instrukcí*. Každá instrukce (příkaz) začíná na novém řádku a obsahuje *operátor*, který může být doplněn *modifikátory*, a pokud je to pro konkrétní instrukce nutné, tak dále obsahuje jeden nebo více *operandů* oddělených čárkami. Na místě operandů mohou být libovolné reprezentace dat definované pro literály (viz kap. 3.1.2) a proměnné (viz kap. 3.3).

Pro účely identifikace může být před instrukcí uvedeno *návěští,* za kterým následuje dvojtečka (:). Návěští slouží k označení místa v programu pro instrukce volání resp. skoku. Na posledním místě na řádku instrukce může být uveden komentář. Mezi instrukcemi mohou být vloženy prázdné řádky. Ukázka programu v jazyku IL je uvedena v příkladu 4.1.

Příklad 4.1 Program v jazyce IL

```
VAR_GLOBAL
  AT %X1.2       : BOOL;
  AT %Y2.0       : BOOL;
END_VAR

PROGRAM Example_IL
  VAR
    tmp1, tmp2   : BOOL;
  END_VAR

Step1:  LD       %X1.2    // load bit from PLC input
        AND      tmp1     (* AND temporary variable *)
        ST       %Y2.0    (* store to PLC output *)
                          (* empty instruction *)
Step2:                    (* label *)
        LDN      tmp2
END_PROGRAM
```

4.1.2 Operátory, modifikátory a operandy

Standardní operátory spolu s přípustnými modifikátory jsou uvedeny v Tab. 4.1 až Tab. 4.4. Pokud není v tabulkách uvedeno jinak, sémantika operátorů je tato:

```
výsledek := výsledek OPERATOR operand
```

To znamená, že hodnota výrazu, který je vyhodnocován, je nahrazena jeho novou hodnotou, která je zpracována z jeho aktuální hodnoty pomocí operátoru a případného operandu. Např. instrukce **`AND %X1`** je interpretována takto:

```
výsledek := výsledek AND %X1
```

Operátory porovnávání jsou interpretovány s aktuálním výsledkem vlevo od znaku porovnávání a operandem vpravo od znaku porovnávání. Výsledkem porovnávání je boolovská proměnná. Např. instrukce **`LT %IW32`** bude mít ve výsledku boolovskou „1“, pokud aktuální výsledek je menší než hodnota vstupního slova číslo 32, ve všech ostatních případech bude mít ve výsledku boolovskou „0“.

Modifikátor **`N`** označuje boolovskou negaci operandu. Např. instrukce **`ORN %X1.5`** je interpretována takto:

```
výsledek := výsledek OR NOT %X1.5
```

Modifikátor levé závorky **`(`** označuje, že operátor má být „odložen“, tj. vykonání operátoru odloženo, (deferred), dokud není zaznamenán operátor pravé závorky **`)`**. Např. sekvence instrukcí

```
AND(        %X1.1
OR          %X1.3
)
```

je interpretována takto:

```
výsledek := výsledek AND (%X1.1 OR %X1.3)
```

Tab.4.1 Operátory a modifikátory pro datový typ ANY_BIT

ANY_BIT Operátory		
Operátor	**Modifikátor**	**Popis funkce**
LD	**N**	Nastaví aktuální výsledek na hodnotu rovnou operandu
AND	**N, (**	Boolovské AND
OR	**N, (**	Boolovské OR
XOR	**N, (**	Boolovské XOR
ST	**N**	Uloží aktuální výsledek na místo operandu
S		Nastaví boolovský operand na „1“ Operace se provede pouze pokud je aktuální výsledek boolovská „1“
R		Smaže boolovský operand na „0“ Operace se provede pouze pokud je aktuální výsledek boolovská „1“
)		Vyhodnocení odložené operace

Některé operátory mohou být doplněny více modifikátory současně. Například operátor **AND** má v kombinaci s modifikátory čtyři různé tvary jak ukazuje tabulka Tab. 4.2.

Tab.4.2 Modifikace operátoru AND

AND	Boolovské AND
AND(	Odložené (deferred) boolovské AND
ANDN	Boolovské AND s negovaným operandem
ANDN(	Odložené boolovské AND s negovaným výsledkem

Tab.4.3 Operátory a modifikátory pro datový typ ANY_NUM

ANY_NUM Operátory		
Operátor	**Modifikátor**	**Popis funkce**
LD	**N**	Nastaví aktuální výsledek na hodnotu rovnou operandu
ST	**N**	Uloží aktuální výsledek na místo operandu
ADD	**(**	Přičíst operand k výsledku
SUB	**(**	Odečíst operand od výsledku
MUL	**(**	Násobit výsledek operandem
DIV	**(**	Dělit výsledek operandem
GT	**(**	Porovnání výsledek > operand
GE	**(**	Porovnání výsledek >= operand
EQ	**(**	Porovnání výsledek = operand
NE	**(**	Porovnání výsledek <> operand
LE	**(**	Porovnání výsledek <= operand
LT	**(**	Porovnání výsledek < operand
)		Vyhodnocení poslední odložené operace

Tab.4.4 Operátory a modifikátory pro skoky a volání

ANY_BIT Operátory		
Operátor	**Modifikátor**	**Popis funkce**
JMP	**C, N**	Skok na návěští
CAL	**C, N**	Volání funkčního bloku
Func_name		Volání funkce
RET	**C, N**	Návrat z funkce nebo funkčního bloku

Modifikátor **C** označuje, že přiřazená instrukce se má provést pouze v případě, že aktuální vyhodnocený výsledek je boolovská „1“ (případně boolovská „0“, pokud je operátor doplněn modifikátorem **N**).

4.1.3 Definice uživatelské funkce v jazyce IL

Příklad 4.2 Uživatelská funkce v jazyce IL

```
FUNCTION UserFun : INT
  VAR_INPUT
    val    :    INT;            // input value
    minVal :    INT;            // minimun
    maxVal :    INT;            // maximum
  END_VAR

        LD      val             // load input value
        GE      minVal          // test if val >= minVal
        JMPC    NXT_TST         // jump if OK
        LD      minVal          // low limit value
        JMP     VAL_OK
NXT_TST:
        LD      val             // load input value
        GT      maxVal          // test if val > maxVal
        JMPCN   VAL_OK          // jump if not
        LD      maxVal          // high limit value
VAL_OK: ST      UserFun         // return value
END_FUNCTION
```

4.1.4 Volání funkcí v jazyce IL

Funkce se v jazyce IL vyvolávají umístěním jména funkce do pole operátoru. Aktuální výsledek se použije jako první parametr funkce. Pokud jsou vyžadovány další parametry, vkládají se do pole operandu a jsou vzájemně odděleny čárkami. Hodnota navrácená funkcí po úspěšném provedení instrukce **RET** nebo po dosažení fyzického konce funkce se pak stane právě aktuálním výsledkem.

Další dva možné způsoby volání odpovídají volání funkce v jazyce ST. Za jménem funkce je v kulatých závorkách uveden seznam parametrů předávaných do funkce. Seznam parametrů může být buď s uvedením jmen parametrů (formal call) nebo bez (informal call).

V příkladu 4.3 jsou ukázány všechny popsané způsoby volání. Volaná funkce je definovaná uživatelsky v příkladu 4.2.

Příklad 4.3 Volání funkce v jazyce IL

```
VAR_GLOBAL
  AT %YW10 : INT;
END_VAR

PROGRAM Example_IL1
  VAR
     count     : INT;
  END_VAR

        // calling function, first parameter is current result
        LD      Count
        UserFun 100, 1000
        ST      %YW10

        // calling function using an informal call
        UserFun( Count, 100, 1000)
        ST      %YW10

        // calling function using a formal call
        UserFun( val := Count, minVal := 100, maxVal := 1000)
        ST      %YW10

END_PROGRAM
```

4.1.5 Volání funkčních bloků v jazyce IL

Funkční bloky se v jazyce IL vyvolávají podmíněně nebo nepodmíněně pomocí operátoru **CAL** (Call). Jak ukazuje následující příklad, funkční blok je možné volat dvěma způsoby.

Funkční blok se může volat jeho jménem, za nímž je uveden v závorce seznam parametrů (formal call). Druhou možností je uložení parametrů do příslušných paměťových míst instance funkčního bloku a jeho následné zavolání (informal call). Oba způsoby je možné kombinovat.

Příklad 4.4 Volání funkčního bloku v jazyce IL

```
VAR_GLOBAL
  in1   AT %X1.0  : BOOL;
  out1  AT %Y1.0  : BOOL;
END_VAR

PROGRAM Example_IL2
  VAR
    timer       : TON;
    timerValue  : TIME;
  END_VAR

  // calling FB using an informal call
  LD    in1
  ST    timer.IN        // parameter IN
  LD    T#10m12s
  ST    timer.PT        // parameter PT
  CAL   timer           // calling FB TON
  LD    timer.ET
  ST    timerValue      // timer value
  LD    timer.Q
  ST    out1            // timer output

  // calling FB using an informal call
  CAL   timer( IN := in1, PT := T#10m12s, Q => out1, ET => timerValue)

  // another way
  LD    in1
  ST    timer.IN
  CAL   timer( PT := T#10m12s, ET => timerValue)
  LD    timer.Q
  ST    out1

END_PROGRAM
```

4.2 Jazyk strukturovaného textu ST

Jazyk strukturovaného textu je jedním z jazyků definovaných normou IEC 61 131-3. Je to velmi výkonný vyšší programovací jazyk, který má kořeny ve známých jazycích Ada, Pascal a C. Je objektově orientován a obsahuje všechny podstatné prvky moderního programovacího jazyka, včetně větvení (**`IF-THEN-ELSE`** a **`CASE OF`**) a iterační smyčky (**`FOR`**, **`WHILE`** a **`REPEAT`**). Tyto prvky mohou být vnořovány. Tento jazyk je vynikajícím nástrojem pro definování komplexních funkčních bloků.

Algoritmus zapsaný v jazyce ST lze rozdělit na jednotlivé ***příkazy*** *(statements)*. Příkazy se používají pro výpočet a přiřazení hodnot, řízení toku vykonávání programu a pro volání resp. ukončení POU. Část příkazu, která vypočítává hodnotu, je nazývána ***výraz***. Výrazy produkují hodnoty nezbytné pro provádění příkazů.

4.2.1 Výrazy

Výraz je konstrukce, ze které se po vyhodnocení vygeneruje hodnota odpovídající některému z datových typů, které byly definovány v kapitole 3.2.

Výraz se skládá z ***operátorů a operandů***. Operandem může být literál, proměnná, volání funkce nebo jiný výraz.

Operátory jazyka strukturovaného textu ST jsou přehledně uspořádány v Tab.4.5.

Tab.4.5 Operátory v jazyce strukturovaného textu ST

Operátor	Operace	Priorita
( )	Závorky	Nejvyšší
******	Umocňování	
- **NOT**	Znaménko Doplněk	
***** **/** **MOD**	Násobení Dělení Modulo	
+ **-**	Sčítání Odčítání	
<, >, <= ,>=	Porovnávání	
= **<>**	Rovnost Nerovnost	
&, AND	Boolovské AND	
XOR	Boolovské exkluzivní OR	
OR	Boolovské OR	Nejnižší

Pro operandy operátorů platí stejná omezení jako pro vstupy odpovídajících funkcí definovaných v kapitole 3.4.1.1. Např. výsledek vyhodnocení výrazu **A**B** je stejný jako výsledek vyhodnocení funkce **EXPT(A, B)**, jak je definována v Tab.3.14.

Vyhodnocení výrazu spočívá v aplikaci operátorů na operandy a to s ohledem na prioritu vyjádřenou v Tab.4.5. Operátory s nejvyšší prioritou ve výrazu jsou aplikovány nejdříve, pak následují další operátory směrem k nižší prioritě dokud není vyhodnocování dokončeno.Operátory se stejnou prioritou se vyhodnocují tak jak jsou zapsány ve výrazu směrem odleva doprava.

Příklad 4.5 Priorita operátorů při vyhodnocování výrazů

```
PROGRAM PRIKLAD
  VAR                                  // lokální proměnné
    A       : INT := 2;
    B       : INT := 4;
    C       : INT := 5;
    D       : INT := 8;
    X,Y     : INT;
    Z       : REAL;
  END_VAR

  X := A + B - C * ABS(D);             // X = -34
  Y := (A + B - C) * ABS(D);           // Y = 8
  Z := INT_TO_REAL( Y);
END_PROGRAM
```

Vyhodnocením výrazu **A + B- C * ABS(D)** v příkladu 4.5 dostaneme hodnotu –34. Pokud požadujeme jiné pořadí vyhodnocování než je uvedeno, musíme použít závorky. Pro stejné hodnoty proměnných pak vyhodnocením výrazu **(A + B - C) * ABS(D)** dostaneme hodnotu 8.

Funkce se volají jako prvky výrazů, které se skládají ze jména funkce, za nímž následuje seznam argumentů v závorce.

Pokud má operátor dva operandy, operátor, který je nejvíce vlevo se bude vyhodnocovat jako první. Například výraz součinu dvou goniometrických funkcí **COS(Y) * SIN(X)** bude proto vyhodnocen v tomto pořadí: výpočet výrazu **COS(Y)**, výpočet výrazu **SIN(X)** a poté teprve výpočet součinu **COS(Y) a SIN(X)**.

Boolovské výrazy mohou být vyhodnocovány pouze v rozsahu nutném pro získání jednoznačné výsledné hodnoty. Například pokud platí, že **C <= D**, pak může být z výrazu **(C > D) & (F < A)** vyhodnocena pouze první závorka **(C > D)**. Její hodnota je vzhledem k předpokladu nulová a to již postačuje k tomu, aby byl nulový celý logický součin. Druhý výraz **(F < A)** se tedy už vyhodnocovat nemusí.

Pokud operátor ve výrazu může být reprezentován jako jedna z přetížených funkcí definovaných v kapitole 3.4.1.1, probíhá konverze operandů a výsledků podle pravidel a příkladů uvedených v této kapitole.

4.2.2 Souhrn příkazů v jazyce ST

Seznam příkazů jazyka strukturovaného textu ST je souhrnně uveden v Tab.4.6. Příkazy jsou ukončeny středníkem.Se znakem konec řádku se v tomto jazyku zachází stejně jako se znakem mezery (space).

Tab.4.6 Seznam příkazů jazyka strukturovaného textu ST

Příkaz	Popis	Příklad	Poznámka
:=	Přiřazení	A **:=** 22;	Přiřazení hodnoty vypočtené na pravé straně do identifikátoru na levé straně
	Volání funkčního bloku	**InstanceFB(** par1 := 10, par2 := 20);	Volání funkčního bloku s předáváním parametrů
IF	Příkaz výběru	**IF** A > 0 **THEN** B := 100; **ELSE** B := 0; **END_IF;**	Výběr alternativy v podmíněný výrazem BOOL
CASE	Příkaz výběru	**CASE** kod **OF** 1 : A := 11; 2 : A := 22; **ELSE** A := 99; **END_CASE;**	Výběr bloku příkazů podmíněný hodnotou výrazu „kod“

FOR	Iterační příkaz smyčka FOR	**FOR** i := 0 **TO** 10 **BY** 2 **DO**
		j := j + i;
		END_FOR;
			Vícenásobná smyčka bloku příkazů s počáteční a koncovou podmínkou a hodnotou inkrementu

WHILE	Iterační příkaz smyčka WHILE	**WHILE** i > 0 **DO**
		n := n * 2;
		END_WHILE;
			Vícenásobná smyčka bloku příkazů s podmínkou ukončení smyčky na začátku

REPEAT	Iterační příkaz smyčka REPEAT	**REPEAT**
		k := k + i;
		UNTIL i < 20;
		END_REPEAT;
			Vícenásobná smyčka bloku příkazů s podmínkou ukončení smyčky na konci

EXIT	Ukončení smyčky	**EXIT;**	Předčasné ukončení iteračního příkazu
RETURN	Návrat	**RETURN**;	Opuštění právě vykonávané POU a návrat do volající POU
;	Prázdný příkaz	**;;**	

4.2.2.1 Příkaz přiřazení

Přiřazovací příkaz nahrazuje aktuální hodnotu jednoduché nebo složené proměnné výsledkem, který vznikne po vyhodnocení výrazu. Přiřazovací příkaz se skládá z odkazu na proměnnou na levé straně, za ním následuje operátor přiřazení „:=“, za kterým je uveden výraz, který se má vyhodnotit.

Příkaz přiřazení je velmi mocný. Může přiřadit jednoduchou proměnnou ale i celou datovou strukturu. Jak ukazuje příklad 4.6, kde přiřazovací příkaz **`A := B`** je použit pro nahrazení hodnoty jednoduché proměnné **`A`** aktuální hodnotou jednoduché proměnné **`B`** (obě proměnné jsou základního typu **`INT`**). Ovšem přiřazení lze s úspěchem použít i pro složené proměnné **`AA := BB`** a pak se tímto přiřazovacím příkazem přepíší všechny položky složené proměnné **`AA`** položkami složené proměnné **`BB`**. Proměnné musí být samozřejmě stejného datového typu.

Příklad 4.6 Přiřazení jednoduché a složené proměnné

```
TYPE
  tyZAZNAM : STRUCT
    vyrobniCislo     : UDINT;
    barva            : (cervena, zelena, bila, modra);
    jakost           : USINT;
  END_STRUCT;
END_TYPE

PROGRAM PRIKLAD
  VAR                        // lokální proměnné
    A, B   : INT;
    AA, BB : tyZAZNAM;
  END_VAR

  A  := B;                        // přiřazení jednoduché proměnné
  AA := BB;                       // přiřazení složené proměnné
END_PROGRAM
```

Přiřazovací příkaz se může použít také pro přiřazení návratové hodnoty funkce, a to umístěním jména funkce na levé straně přiřazovacího operátoru v těle deklarace funkce. Návratová hodnota funkce bude výsledkem posledního vyhodnocení tohoto přiřazovacího příkazu.

Příklad 4.7 Přiřazení návratové hodnoty funkce

```
FUNCTION PRIKLAD : REAL
  VAR_INPUT                  // vstupní proměnné
    F, G   : REAL;
    S      : REAL  := 3.0;
  END_VAR

  PRIKLAD := F * G / S;      // návratová hodnota funkce
END_FUNCTION
```

4.2.2.2 Příkaz volání funkčního bloku

Funkce může být volána jako součást vyhodnocení výrazu, jak bylo uvedeno v této kapitole, odstavec výrazy.

Funkční bloky se volají příkazem, který se skládá ze jména instance funkčního bloku, za kterým následuje seznam pojmenovaných vstupních parametrů s přiřazenými hodnotami.Na pořadí, v němž jsou parametry v seznamu při volání funkčního bloku uvedeny, nezáleží. Při každém volání funkčního bloku nemusí být přiřazeny všechny vstupní parametry. Pokud nějakému parametru není přiřazena hodnota před voláním funkčního bloku, pak se použije hodnota naposledy přiřazená (nebo hodnota počáteční, pokud nebylo ještě provedeno žádné přiřazení).

Příklad 4.8 Příkaz volání funkčního bloku

```
// deklarace funkčního bloku
FUNCTION_BLOCK fb_OBDELNIK
  VAR_INPUT
    A,B               : REAL;          // vstupní proměnné
  END_VAR
  VAR_OUTPUT
    obvod, plocha     : REAL;          // výstupní proměnné
  END_VAR

  obvod := 2.0 * (A + B); plocha := A * B;
END_FUNCTION_BLOCK

// globální proměnné
VAR_GLOBAL
  OBDELNIK : fb_OBDELNIK;             // globální instance FB
END_VAR

// deklarace programu
PROGRAM main
  VAR
    o,s     : REAL;                   // lokální proměnné
  END_VAR

  // volání FB s úplným seznamem parametrů
  OBDELNIK( A := 2.0, B := 3.0, obvod => o , plocha => s);
  IF  o > 20.0 THEN
    ....
  END_IF;

  // volání FB s neúplným seznamem parametrů
  OBDELNIK( B := 4.0, A := 2.5);
  IF  OBDELNIK.obvod > 20.0 THEN
    ....
  END_IF;
END_PROGRAM
```

4.2.2.3 Příkaz IF

Příkaz **IF** specifikuje, že se má provádět skupina příkazů jedině v případě, že se přiřazený boolovský výraz vyhodnotí jako pravdivý (**TRUE**). Pokud je podmínka nepravdivá, pak se neprovádí buď žádný příkaz nebo se provádí skupina příkazů, které jsou uvedeny za klíčovým slovem **ELSE** (nebo za klíčovým slovem **ELSIF**, pokud jemu přiřazená podmínka je pravdivá).

Příklad 4.9 Příkaz IF

```
FUNCTION PRIKLAD : INT
  VAR_INPUT
    kod     : INT;                    // vstupní proměnná
  END_VAR

  IF kod < 10 THEN PRIKLAD := 0;      // při kod < 10 fce vrátí 0
  ELSIF kod < 100 THEN PRIKLAD := 1;  // při 9 < kod < 100 fce vrátí 1
  ELSE PRIKLAD := 2;                  // při kod > 99 fce vrátí 2
  END_IF;
END_FUNCTION
```

4.2.2.4 Příkaz CASE

Příkaz **CASE** obsahuje výraz, který se vyhodnotí do proměnné typu **INT** (to je tzv. „selektor“), a dále seznam skupin příkazů, kde každá skupina je označena jedním nebo více přirozenými čísly nebo rozsahem přirozených čísel. Tím je vyjádřeno, že se bude provádět první skupina příkazů, do jejíchž mezí patří vypočítaná hodnota selektoru. Pokud se vypočítaná hodnota nehodí ani do jedné skupiny příkazů, provede se sekvence příkazů, které jsou uvedeny za klíčovým slovem **ELSE** (pokud se v příkazu **CASE** vyskytuje). Jinak se neprovede žádná sekvence příkazů.

Příklad 4.10 Příkaz CASE

```
FUNCTION PRIKLAD : INT
  VAR_INPUT
    kod     : INT;              // vstupní proměnná
  END_VAR

  CASE kod OF
    10      : PRIKLAD := 0;     // při kod = 10 fce vrátí 0
    20,77   : PRIKLAD := 1;     // při kod = 20 nebo kod = 77 fce vrátí 1
    21..55  : PRIKLAD := 2;     // při 20 < kod < 56 fce vrátí 2
    100     : PRIKLAD := 3;     // při kod = 100 fce vrátí 3
  ELSE
              PRIKLAD := 4;     // jinak fce vrátí 4
  END_CASE;
END_FUNCTION
```

4.2.2.5 Příkaz FOR

Příkaz **FOR** se používá, pokud počet iterací může být určen předem, jinak se používají konstrukce **WHILE** nebo **REPEAT**.

Příkaz **FOR** indikuje, že sekvence příkazů se má provádět opakovaně až do výskytu klíčového slova **END_FOR**, přičemž se zvyšují hodnoty řídicí proměnné smyčky **FOR**. Řídicí proměnná, počáteční hodnota a koncová hodnota jsou výrazy stejného typu.integer (**SINT**, **INT** nebo **DINT**) a nesmí se měnit vlivem jakéhokoli z opakovaných příkazů. Příkaz **FOR** zvyšuje nebo snižuje hodnotu řídicí proměnné cyklu od počáteční do koncové hodnoty, a to po přírůstcích určených hodnotou výrazu (defaultně je tento přírůstek roven jedné). Test ukončovací podmínky se provádí na začátku každé iterace, takže pokud počáteční hodnota řídicí proměnné cyklu překročí hodnotu koncovou, sekvence příkazů se neprovede.

Příklad 4.11 Příkaz FOR

```
FUNCTION FAKTORIAL : UDINT
  VAR_INPUT
    kod     : USINT;                    // vstupní proměnná
  END_VAR
  VAR_TEMP
    i       : USINT;                    // pomocná proměnná
    tmp     : UDINT := 1;               // pomocná proměnná
  END_VAR

  FOR i := 1 TO kod DO
    tmp := tmp * USINT_TO_UDINT( i);
  END_FOR;
  FAKTORIAL := tmp;
END_FUNCTION
```

4.2.2.6 Příkaz WHILE

Příkaz **WHILE** způsobí, že se sekvence příkazů až do klíčového slova **END_WHILE** bude provádět opakovaně, až do té doby, dokud není přiřazený boolovský výraz nepravdivý. Pokud je přiřazený boolovský výraz na začátku nepravdivý, pak se sekvence příkazů neprovede vůbec. Smyčka **FOR ... END_FOR** se dá přepsat použitím konstrukce **WHILE** … **END_WHILE**. Příklad 4.12 lze s použitím příkazu **WHILE** přepsat následovně:

Příklad 4.12 Příkaz WHILE

```
FUNCTION FAKTORIAL : UDINT
  VAR_INPUT
    kod     : USINT;                    // vstupní proměnná
  END_VAR
  VAR_TEMP
    i       : USINT;                    // pomocná proměnná
    tmp     : UDINT := 1;               // pomocná proměnná
  END_VAR

  i := kod;
  WHILE i <> 0 DO
    tmp := tmp * USINT_TO_UDINT( i);  i := i - 1;
  END_WHILE;
  FAKTORIAL := tmp;
END_FUNCTION
```

Pokud, je příkaz **WHILE** použit v algoritmu, pro který není zaručeno splnění podmínky ukončení nebo provedení příkazu **EXIT**, pak řídící systém vyhlásí chybu cyklu.

4.2.2.7 Příkaz REPEAT

Příkaz **REPEAT** způsobí, že se sekvence příkazů až do klíčového slova **UNTIL** bude provádět opakovaně (a alespoň jednou) až do té doby, dokud není přiřazený boolovský výraz pravdivý. Smyčka **WHILE**...**END_WHILE** se dá přepsat použitím konstrukce **REPEAT** ... **END_REPEAT**, což opět ukážeme na stejném příkladu :

Příklad 4.13 Příkaz REPEAT

```
FUNCTION FAKTORIAL : UDINT
  VAR_INPUT
    kod     : USINT;                    // vstupní proměnná
  END_VAR
  VAR_TEMP
    i       : USINT := 1;               // pomocná proměnná
    tmp     : UDINT := 1;               // pomocná proměnná
  END_VAR

  REPEAT
    tmp := tmp * USINT_TO_UDINT( i);  i := i + 1;
  UNTIL i > kod
  END_REPEAT;
  FAKTORIAL := tmp;
END_FUNCTION
```

Pokud, je příkaz **REPEAT** použit v algoritmu, pro který není zaručeno splnění podmínky ukončení nebo provedení příkazu **EXIT**, pak řídící systém vyhlásí chybu cyklu.

4.2.2.8 Příkaz EXIT

Příkaz **EXIT** se používá pro ukončení iterací před splněním ukončovací podmínky.

Pokud je příkaz **EXIT** umístěn uvnitř vnořené iterační konstrukce (příkazy **FOR**, **WHILE**, **REPEAT**), odchod nastane z nejhlubší smyčky, ve které je **EXIT** umístěn, tzn. že se řízení předá na další příkaz za prvním ukončením smyčky (**END_FOR**, **END_WHILE**, **END_REPEAT**), který následuje za příkazem **EXIT**.

Příklad 4.14 Příkaz EXIT

```
FUNCTION FAKTORIAL : UDINT
  VAR_INPUT
    kod     : USINT;                    // vstupní proměnná
  END_VAR
  VAR_TEMP
    i       : USINT;                    // pomocná proměnná
    tmp     : UDINT := 1;               // pomocná proměnná
  END_VAR

  FOR i := 1 TO kod
    IF i > 13 THEN
      tmp := 16#FFFF_FFFF; EXIT;
    END_IF;
    tmp := tmp * USINT_TO_UDINT( i);
  END_FOR;
  FAKTORIAL := tmp;
END_FUNCTION
```

Při výpočtu faktoriálu pro čísla větší než 13 bude výsledek větší než maximální číslo, které lze uložit do proměnné typu **UDINT**. Tato situace je v příkladu 4.14 ošetřena pomocí příkazu **EXIT**.

4.2.2.9 Příkaz RETURN

Příkaz **RETURN** se používá k opuštění funkce, funkčního bloku nebo programu před jeho dokončením.

V případě použití příkazu **RETURN** ve funkci je potřebné nastavit výstup funkce (proměnnou, která se jmenuje stejně jako funkce) před provedením příkazu **RETURN**. V opačném případě nebude výstupní hodnota funkce definována.

Pokud bude příkaz **RETURN** použit ve funkčním bloku, měl by programátor zajistit nastavení výstupních proměnných funkčního bloku před provedením příkazu. Nenastavené výstupní proměnné budou mít hodnotu odpovídající inicializační hodnotě pro příslušný datový typ nebo hodnotu nastavenou v předchozím voláním funkčního bloku.

Příklad 4.15 Příkaz RETURN

```
FUNCTION FAKTORIAL : UDINT
  VAR_INPUT
    kod     : USINT;                    // vstupní proměnná
  END_VAR
  VAR_TEMP
    i       : USINT;                    // pomocná proměnná
    tmp     : UDINT := 1;               // pomocná proměnná
  END_VAR

  IF kod > 13 THEN FAKTORIAL := 16#FFFF_FFFF; RETURN; END_IF;
  i := kod;
  WHILE i <> 0 DO
    tmp := tmp * USINT_TO_UDINT( i);  i := i - 1;
  END_WHILE;
  FAKTORIAL := tmp;
END_FUNCTION
```

Při výpočtu faktoriálu pro čísla větší než 13 bude výsledek větší než maximální číslo, které lze uložit do proměnné typu **UDINT**. Tato situace je v příkladu 4.15 ošetřena pomocí příkazu **RETURN**.

5 GRAFICKÉ JAZYKY

V normě IEC 61 131-3 jsou definovány dva grafické jazyky: jazyk kontaktních schémat (LD, Ladder Diagram, jazyk kontaktních schémat) a jazyk funkčního blokového schématu (FBD, Function Block Diagram). Oba tyto jazyky jsou v prostředí Mosaic podporovány.

5.1 Společné prvky grafických jazyků

Stejně jako je tomu u textových jazyků obsahuje každá deklarace POU v grafickém jazyce deklarační a výkonnou část. Deklarační část je naprosto shodná s textovými jazyky, výkonná část je rozdělena do tzv. obvodů (v anglické literatuře označovaných jako networks). Každý obvod se skládá z následujících prvků:

- návěští obvodu
- komentář obvodu
- grafika obvodu

Návěští obvodu

Každý obvod může být opatřen návěštím, což je uživatelem definovaný identifikátor zakončený znakem dvojtečka. Návěští pak může být cílem skoku při větvení výkonného programu POU. Rozsah působnosti obvodu a jeho návěští je *lokální* v rámci té programové organizační jednotky, ve které je obvod umístěn. Návěští obvodu není povinné.

V programovacím prostředí Mosaic je navíc každý obvod opatřen pořadovým číslem obvodu. To je generováno automaticky a slouží k lepší orientaci ve složitých POU. Při vložení nového obvodu se následující obvody automaticky přečíslují. Grafický editor pak umožňuje rychle vyhledávat obvody v POU podle jejich čísel.

Komentář obvodu

Mezi návěštím obvodu a grafikou obvodu může být umístěn komentář obvodu. Ten může být víceřádkový a může obsahovat znaky národních abeced. Komentář obvodu není povinný.

Grafika obvodu

Grafika obvodu obsahuje grafické prvky propojené spojnicemi. Grafickým prvkem může být například spínací kontakt, blok časovače nebo výstupní cívka. Spojnice (propojovací čáry) určují tok informace, například z výstupu časovače na výstupní cívku. Každý grafický prvek může být volitelně opatřen komentářem.

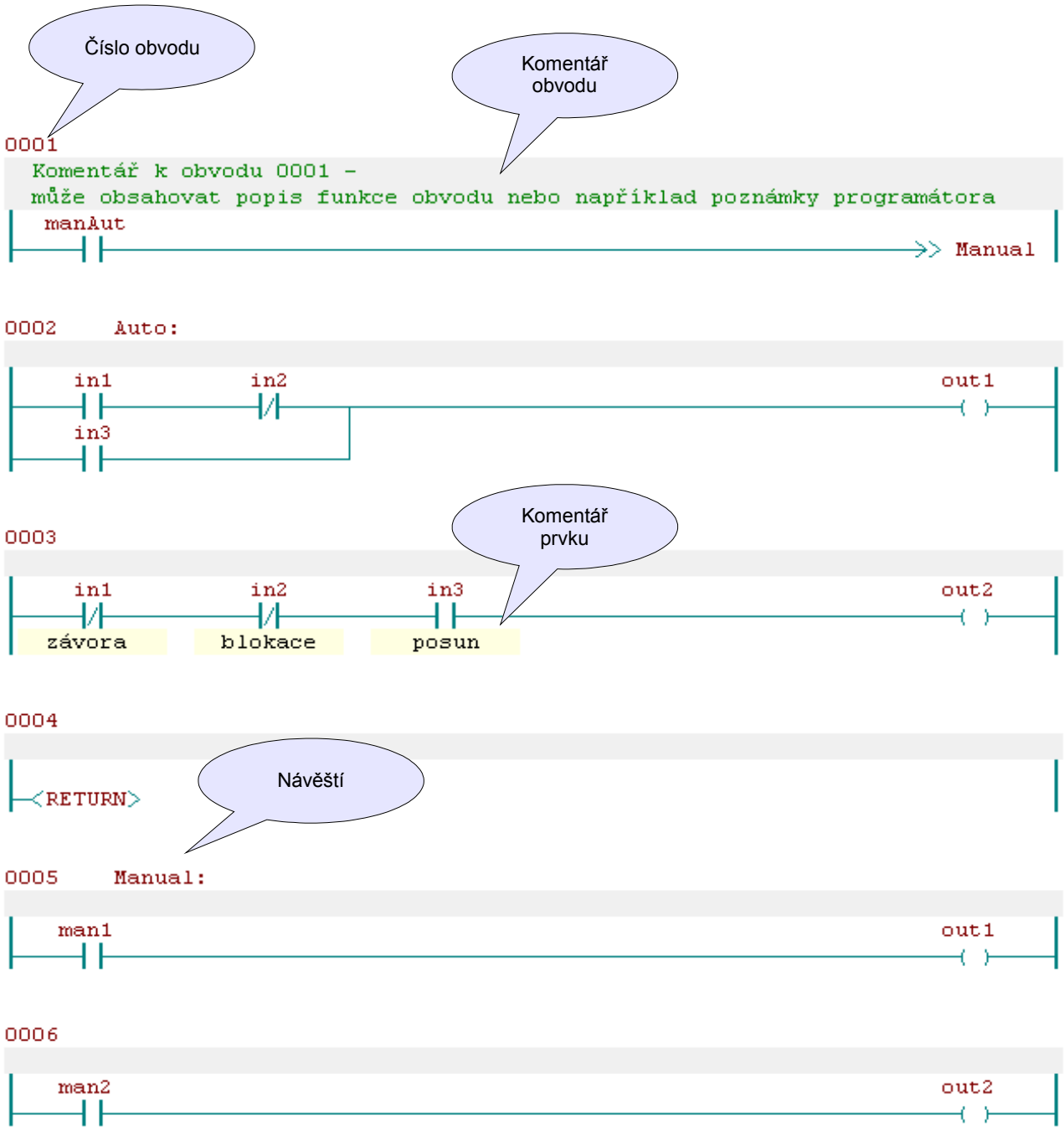
Směr toku v obvodech

Grafické jazyky se používají pro reprezentaci toku "myšleného množství" skrze jeden nebo více obvodů reprezentujících algoritmus řízení. Toto myšlené množství můžeme chápat jako:

- *"tok energie"*, analogický k toku elektrické energie v elektromechanických reléových systémech, který se běžně používá v reléových schématech
- *"tok signálu"*, analogický k toku signálů mezi prvky systému zpracovávajícího signály, který se běžně používá ve funkčních blokových diagramech (schématech)

Příslušné "myšlené množství" protéká podél čar mezi prvky sítě podle následujících pravidel:

- Tok energie v jazyku LD probíhá odleva doprava.
- Tok signálu v jazyku FBD probíhá od výstupu (na pravé straně) funkčního bloku ke vstupu (na levé straně) dalšího připojeného funkčního bloku

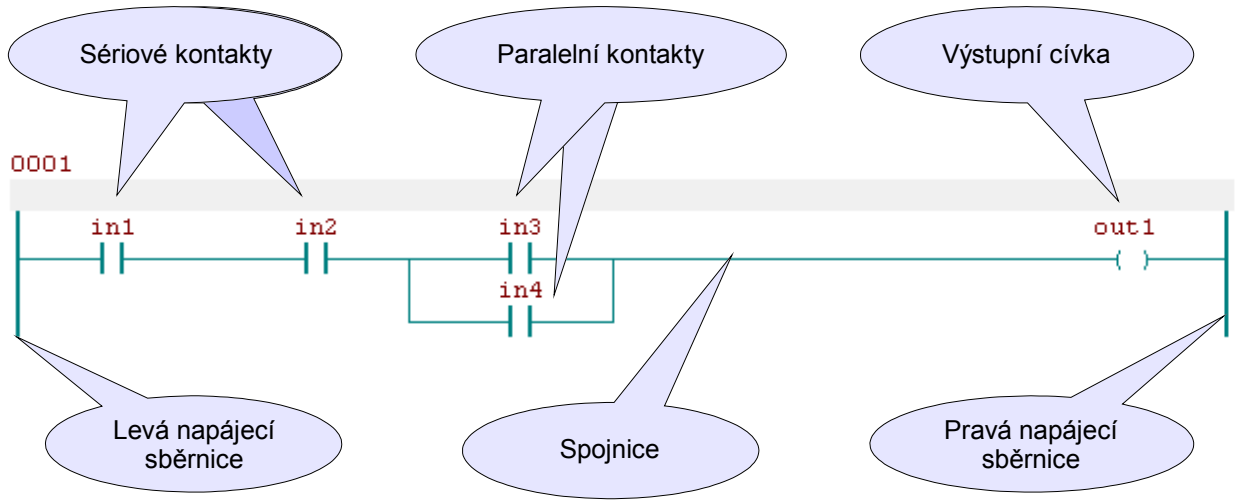


Obr. 5.1 Společné prvky grafických jazyků

5.2 Jazyk kontaktních schémat LD

Jazyk kontaktních schémat (Ladder Diagram) pochází z elektromechanických reléových obvodů je založen na grafické reprezentaci reléové logiky. Tento jazyk je primárně určen pro zpracování booleovských signálů.

Jak už bylo řečeno, výkonná část POU v jazyce LD je složena z obvodů (networks). Obvod je v jazyce LD ohraničen tzv. napájecími sběrnicemi (power rails) na levé a pravé straně. Z levé napájecí sběrnice „vede“ logická jednička (TRUE) do všech na ni připojených grafických prvků, typicky spínacích a rozpínacích kontaktů. V závislosti na jejich stavu se pak logická jednička propouští nebo nepropouští do následujících prvků zapojených v obvodu. Poslední prvek vpravo bývá výstupní a je připojen na pravou napájecí sběrnici. Typickým představitelem výstupního prvku je cívka.



Obr. 5.2 Sériové a paralelní propojení prvků v obvodu

5.2.1 Grafické prvky v jazyce LD

Obvod v jazyce LD může obsahovat následující grafické prvky :

- napájecí sběrnice
- spojnice
- kontakty a cívky
- grafické prvky pro řízení provádění programu (skoky)
- grafické prvky pro volání funkcí nebo funkčních bloků

Grafické prvky mohou být propojovány sériově nebo paralelně. Na obr.5.2 jsou proměnné **in1** a **in2** propojeny v sérii (AND) s paralelně propojenými proměnnými **in3** a **in4** (OR). Tyto proměnné jsou nazývány kontakty a program testuje jejich hodnotu (čte je). Proměnná **out1** je nazývána cívka a program do ní zapisuje.

Obvod na obr.5.2 realizuje výraz **`out1 := in1 AND in2 AND (in3 OR in4);`**

5.2.1.1 Napájecí sběrnice

Obvod v jazyce LD je ohraničen zleva svislou čarou, která se nazývá *levá napájecí sběrnice*, a napravo svislou čarou, která se nazývá *pravá napájecí sběrnice*. Stav levé napájecí sběrnice je vždy "ON". Pro pravou napájecí sběrnici není definován žádný stav

5.2.1.2 Spojnice v jazyce LD

Prvky spojnic mohou být vodorovné nebo svislé. Stav spojnice může být označen "ON" nebo "OFF", a to podle jeho boolovské hodnoty 1 nebo 0. Pojem *stav spojnice* je synonymem k pojmu *tok energie*.

Vodorovná spojnice je indikována vodorovnou čarou. Vodorovná spojnice předává stav prvku, který je bezprostředně vlevo, k prvku, který je bezprostředně vpravo od něho.

Svislá spojnice se skládá ze svislé čáry protínající jednu nebo více vodorovných spojnic na každé straně. Stav svislých spojnic reprezentuje inkluzivní OR stavů ON vodorovných spojnic po jeho levé straně, to znamená, že stav svislých spojnic bude:

- OFF, pokud stavy všech připojených vodorovných spojnic po jeho levé straně jsou OFF
- ON, pokud stav jedné nebo více připojených spojnic po jeho levé straně je ON

Stav svislé spojnice se kopíruje do všech připojených vodorovných spojnic napravo od ní. Stav svislých spojnic se nekopíruje do žádné připojené vodorovné spojnice vlevo od ní.

Tab.5.1 Spojnice v jazyce FBD

Grafický objekt	Jméno	Funkce
	Vodorovná spojnice	Vodorovná spojnice kopíruje stav prvku připojeného na levé straně do prvku připojeného na pravé straně
	Svislá spojnice s vodorovnými připojeními	Stav vodorovné spojnice vlevo je kopírován do všech vodorovných spojnic vpravo
	Svislá spojnice s vodorovnými připojeními OR	Stav vodorovné spojnice vpravo je výsledkem logické funkce OR stavu všech vodorovných spojnic vlevo

5.2.1.3 Kontakty a cívky

Kontakt umožňuje logickou operaci mezi stavem vodorovné spojnice zleva a proměnnou, která je ke kontaktu přiřazena. Typ logické operace závisí na typu kontaktu. Výsledná hodnota je předávána do spojnice vpravo. Kontakt neovlivňuje hodnotu přiřazené boolovské proměnné.

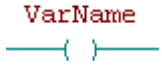
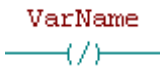
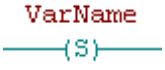
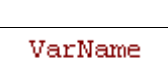
Kontakty mohou být spínací nebo rozpínací. Spínací kontakt je stejně jako elektro-mechanický kontakt v klidovém stavu rozpojen (hodnota proměnné je FALSE) a po přivedení napětí sepne (hodnota proměnné je TRUE). Funkce rozpínacího kontaktu je přesně opačná. V klidovém stavu (bez přivedeného napětí) je kontakt sepnutý (tj. testovaná hodnota je TRUE) a po přivedení napětí kontakt rozepne (testovaná hodnota je FALSE). Funkce kontaktů v jazyce LD je vysvětlena v tab.5.2.

Tab.5.2 Kontakty v jazyce LD

Grafický objekt	Jméno	Funkce
	Spínací kontakt (Open contact)	Pravá spojnice := levá spojnice AND VarName; (Kopíruje stav levé spojnice do pravé spojnice jestliže stav proměnné **VarName** je **TRUE**, jinak do pravé spojnice zapisuje **FALSE**)
	Rozpínací kontakt (Closed contact)	Pravá spojnice := levá spojnice AND NOT VarName; (Kopíruje stav levé spojnice do pravé spojnice jestliže stav proměnné **VarName** je **FALSE**, jinak do pravé spojnice zapisuje **FALSE**)

Cívka kopíruje stav spojnice vlevo od ní do spojnice vpravo a zároveň uloží tento stav do přiřazené boolovské proměnné. Typy cívek a jejich funkce jsou uvedeny v tab.5.3.

Tab.5.3 Cívky v jazyce LD

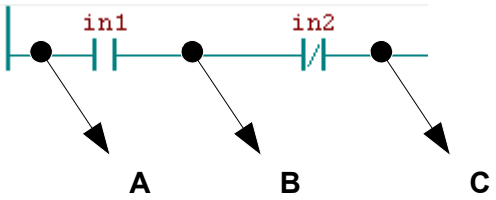
Grafický objekt	Jméno	Funkce
	Cívka (Coil)	Proměnná := levá spojnice; (Kopíruje stav levé spojnice do proměnné **VarName** a zároveň do pravé spojnice)
	Negovaná cívka (Negated coil)	Proměnná := NOT levá spojnice; (Kopíruje negaci stavu levé spojnice do proměnné **VarName** a zároveň do pravé spojnice)
	Cívka Set (Set coil)	Nastaví do proměnné **VarName** hodnotu **TRUE** v případě, že stav levé spojnice je **TRUE**, jinak ponechá proměnnou v původním stavu. Stav pravé spojnice kopíruje stav levé spojnice.
	Cívka Reset (Reset coil)	Nastaví do proměnné **VarName** hodnotu **FALSE** v případě, že stav levé spojnice je **TRUE**, jinak ponechá proměnnou v původním stavu. Stav pravé spojnice kopíruje stav levé spojnice.

Vyhodnocování toku energie v obvodech

Tok energie v obvodech je vyhodnocován zleva doprava. Při výpočtu programu jsou pak jednotlivé obvody v POU vyhodnocovány v pořadí shora dolů.

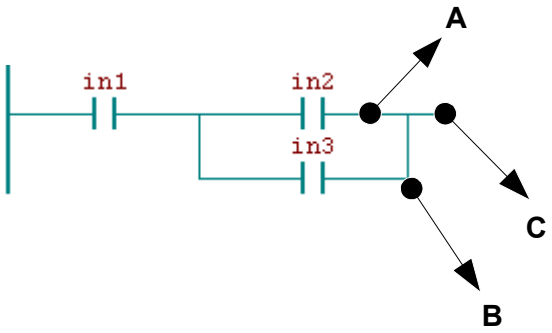
Příklad vyhodnocování sériových kontaktů je uveden v tab.5.4. Uvedený obvod realizuje výraz **`C := in1 AND NOT in2`**. Příklad vyhodnocování paralelních kontaktů uvádí tab.5.5. Uvedený obvod realizuje výraz **`C := in1 AND (in2 OR in3)`**.

Tab.5.4 Vyhodnocování toku energie – sériové kontakty



in1	in2	NOT in2	A	B	C
0	0	1	1	0	0
0	1	0	1	0	0
1	0	1	1	1	1
1	1	0	1	1	0

Tab.5.5 Vyhodnocování toku energie – paralelní kontakty



in1	in2	in3	A	B	C
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	0	0	0
1	0	1	0	1	1
1	1	0	1	0	1
1	1	1	1	1	1

5.2.1.4 Řízení provádění programu v jazyce LD

Pro řízení provádění programu máme v jazyce LD dvě možnosti : skok na určitý obvod v aktuální POU a ukončení POU. Grafické symboly jsou uvedeny v tab.5.6.

Skoky se znázorňují vodorovnou čarou ukončenou dvojitou šipkou. Předání řízení programu na určené návěští obvodu proběhne, pokud je boolovská hodnota spojovací čáry 1 (TRUE). Spojovací čára pro podmínku skoku může začínat u boolovské proměnné, u boolovského výstupu funkce nebo funkčního bloku nebo na levé napájecí sběrnici. Nepodmíněný skok je proto speciálním případem podmíněného skoku. Cílem skoku je návěští sítě v rámci té programové organizační jednotky, v níž se skok objeví. Nelze tedy skákat jinak než v rámci jedné POU.

Podmíněné návraty z funkcí a funkčních bloků se implementují použitím konstrukce RETURN. Provádění programu se předá zpět do vyvolávající POU, pokud je boolovský vstup 1 (TRUE). Provádění programu bude pokračovat v normálním běhu, pokud má boolovský vstup hodnotu 0. Nepodmíněné návraty vzniknou na fyzickém konci funkce nebo funkčního bloku nebo pomocí prvku RETURN, který je připojen k levé napájecí sběrnici.

Tab.5.6 Předání řízení programu v jazyce LD

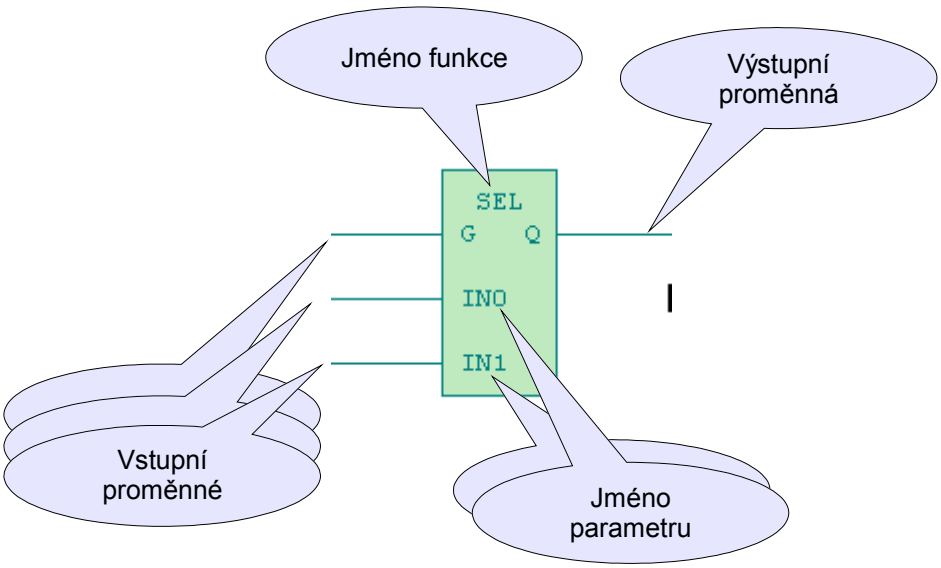
Grafický objekt	Jméno	Funkce
	Nepodmíněný skok (jump)	Skok na obvod s návěštím **`Label`**
	Podmíněný skok (Conditional jump)	Skok na obvod s návěštím **`Label`** jestliže proměnná **`VarName`** má hodnotu **`TRUE`**, jinak program pokračuje v řešení následujícího obvodu

Grafický objekt	Jméno	Funkce
	Nepodmíněný návrat z POU (Return)	Ukončí POU a vrátí řízení do volající POU. POU je rovněž ukončena, pokud se vyřeší všechny její obvody

Grafický objekt	Jméno	Funkce
	Podmíněný návrat z POU (Conditional return)	Ukončí POU a vrátí řízení do volající POU jestliže proměnná **`VarName`** má hodnotu **`TRUE`**, jinak program pokračuje v řešení následujícího obvodu

5.2.1.5 Volání funkcí a funkčních bloků v jazyce LD

Jazyk LD podporuje volání funkcí a funkčních bloků. Volané POU jsou ve schématu reprezentovány obdélníkem. Vstupní proměnné jsou reprezentovány spojnicemi zleva, výstupní proměnné spojnicemi zprava. Jména vstupních a výstupních formálních parametrů jsou uvedena uvnitř obdélníku naproti spojnicím, přes které se připojují aktuální hodnoty parametrů (proměnné nebo konstanty). U rozšiřitelných funkcí (např. ADD, XOR, atd.) se jména vstupních parametrů neuvádějí. Jméno funkce nebo typ funkčního bloku je pak uvedeno v horní části obdélníku. Jméno instance funkčního bloku je uvedeno nad obdélníkem. Obdélníky funkcí jsou kresleny zeleně, funkční bloky jsou modré.



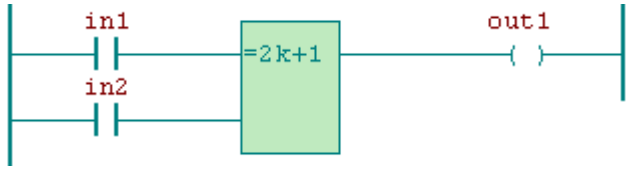
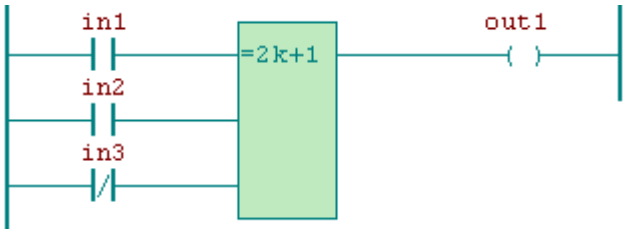
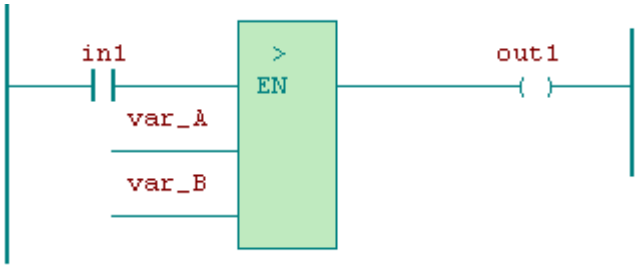
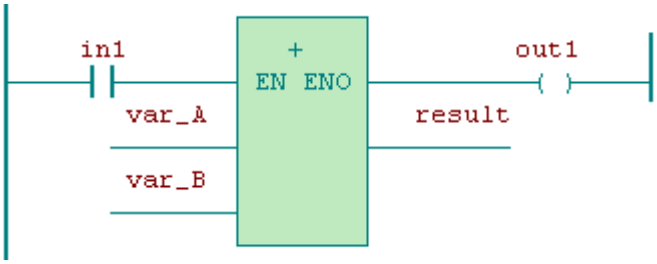
Obr. 5.3 Grafická reprezentace funkce v jazyce LD

Volání funkcí

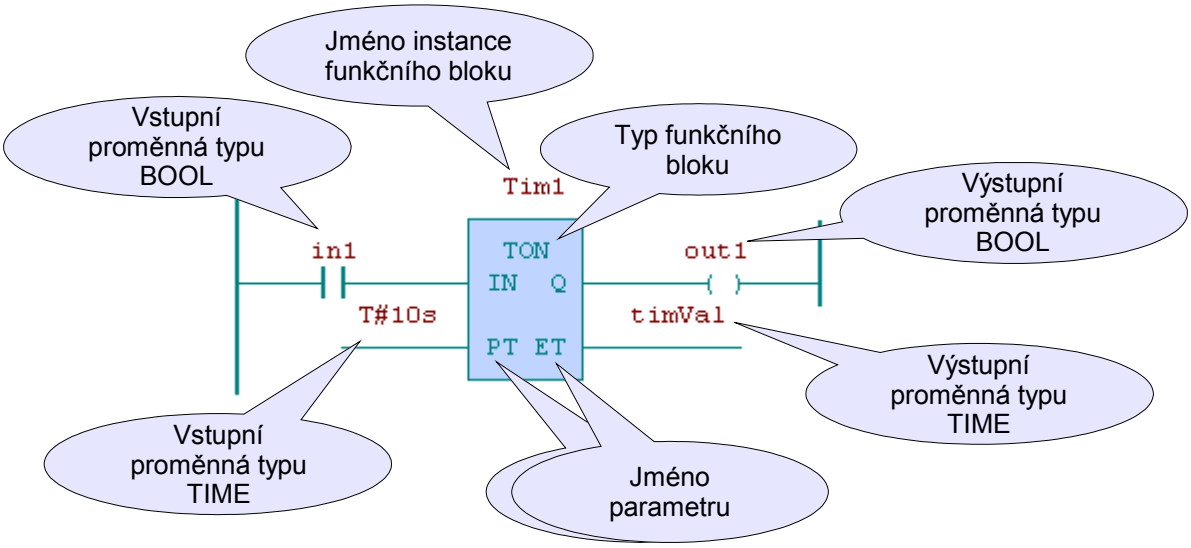
Pokud má funkce alespoň jeden vstup typu BOOL je tento vstup připojen k levé spojnici v obvodu. Pokud má funkce výstup typu BOOL je tento výstup připojen k pravé spojnici v obvodu.

Jinak jsou pro zapojení funkce do obvodu použity implicitní booleovské proměnné EN a ENO. EN je vstupní proměnná typu BOOL, která podmiňuje volání funkce. Pokud je na vstup EN přivedena hodnota TRUE, volání funkce se provede. V opačné případě funkce nebude volaná. V každém případě se hodnota vstupu EN kopíruje do výstupu funkce ENO. Zapojení výstupu ENO není povinné. Použití EN / ENO je typické například v případě aritmetických funkcí.

Tab.5.7 Volání funkcí v jazyce LD

Obvod	Popis
	Volání standardní funkce **XOR** Obvod realizuje výraz **out1 := IN1 XOR in2**
	Volání standardní funkce **XOR** s rozšířeným počtem vstupů Obvod realizuje výraz **out1 := in1 XOR in2 XOR NOT in3**
	Volání funkce **GT** s použitím implicitního vstupu **EN**. Implicitní výstup **ENO** není použit. Jestliže má vstup **EN** hodnotu **TRUE** obvod realizuje výraz **out1 := var_A > var_B** Jinak se hodnota proměnné **out1** nepočítá.
	Volání funkce **ADD** s použitím implicitního vstupu **EN** a implicitního výstupu **ENO**. Jestliže má vstup **EN** hodnotu **TRUE** obvod realizuje výraz **result := var_A + var_B** Jinak se hodnota proměnné **result** nepočítá. Výstup **ENO** kopíruje stav vstupu **EN**.

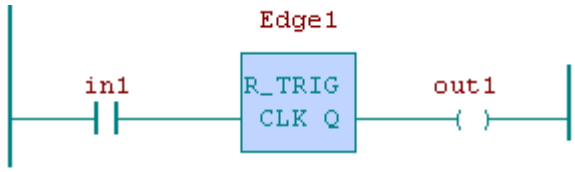
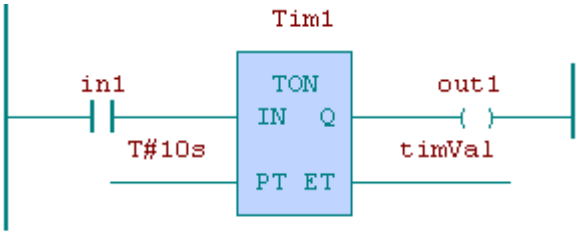
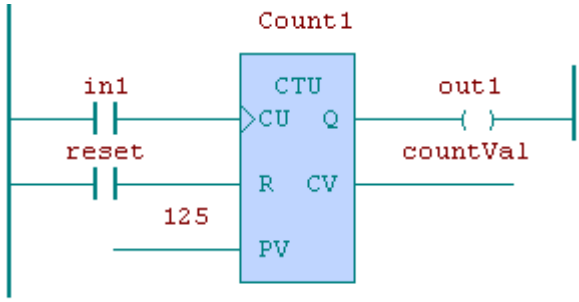
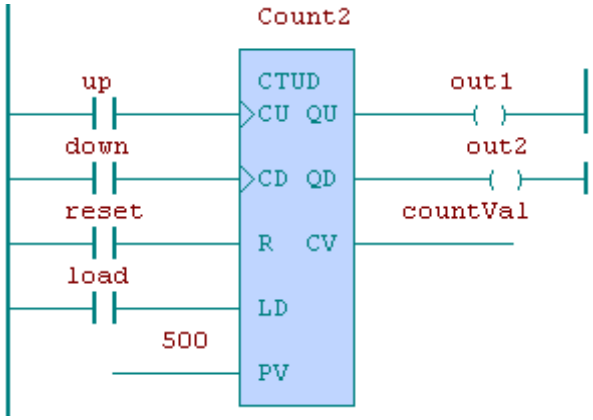
Volání funkčních bloků



Obr. 5.4 Volání funkčního bloku v jazyce LD

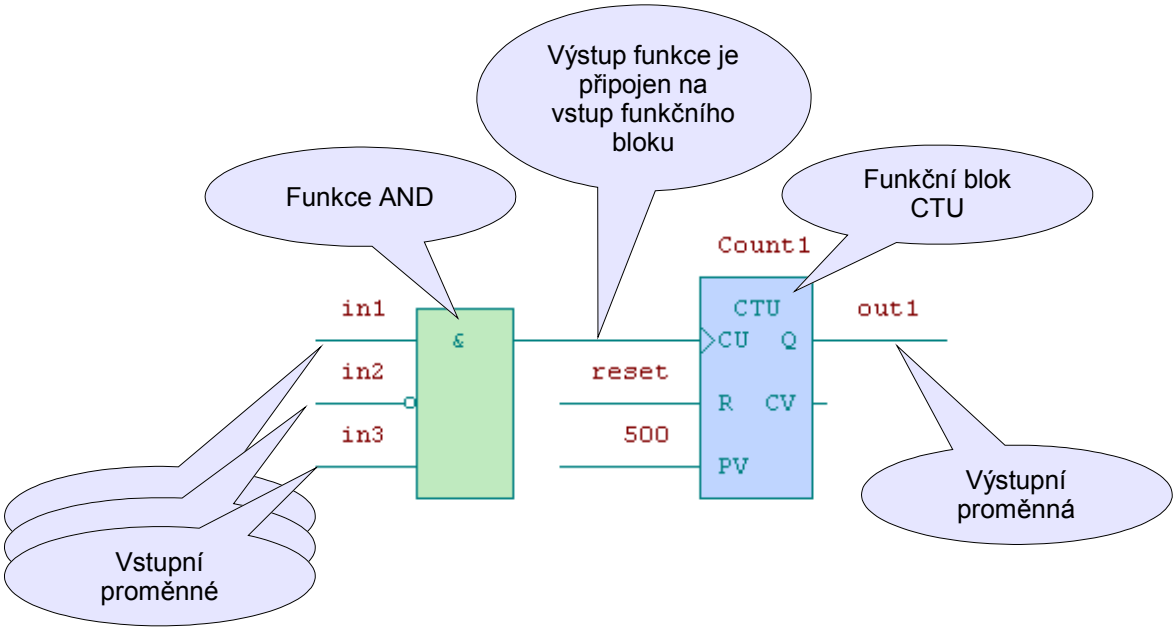
Pro volání funkčních bloků v jazyce LD platí podobná pravidla jako pro volání funkcí. Abychom mohli zapojit funkční blok do obvodu v jazyce LD, musí mít nějaký vstup typu BOOL (protože tok signálu v LD obvodu začíná na levé napájecí spojnici, na kterou lze připojit pouze prvky BOOL). Pokud funkční blok nemá žádný vstup typu BOOL, je možné použít implicitní vstup EN (enable), který podmiňuje vykonávání funkčního bloku. Tento vstup mají automaticky všechny funkce a funkční bloky, což zajišťuje programovací prostředí. Vstup EN bude tedy k dispozici i pro funkční bloky definované uživatelem a to i v případě, že definice bloku žádný takový vstup neuvádí. Totéž platí pro implicitní výstup ENO (Enable Output). Stejně jako u funkcí je hodnota vstupu EN kopírována do výstupu ENO.

Tab.5.8 Volání funkčních bloků v jazyce LD

Obvod	Popis
	Volání standardního funkčního bloku **R_TRIG** Výstup **out1** je nastaven pouze v případě přechodu proměnné **in1** z hodnoty **0** na hodnotu **1** (náběžná hrana)
	Volání standardního funkčního bloku **TON** Vstupní proměnná **PT** (předvolba časovače) je typu **TIME** a není proto připojena na levou napájecí sběrnici. V tomto případě do této proměnné zapisuje konstanta **T#10s** (10 sekund)
	Volání standardního funkčního bloku **CTU** Vstup **CU** je ve funkčním bloku **CTU** definován následovně: **VAR_INPUT** **CU : BOOL R_EDGE;** **END_VAR** Z toho důvodu je vstupní spojnice tohoto signálu ukončena značkou vyhodnocení náběžné hrany 
	Volání standardního funkčního bloku **CTUD** Vstupy **CU** a **CD** jsou typu **BOOL** s detekcí náběžných hran. Vstup **PV** (Preset Value) není typu **BOOL** a není tedy připojen na napájecí sběrnici. V daném případě se do tohoto vstupu zapisuje konstanta **500**. Obdobně výstup **CV** také není typu **BOOL** a tak není rovněž připojen na napájecí sběrnici. Jeho hodnota je zapisována do proměnné **countVal**.

5.3 Jazyk funkčního blokového schématu FBD

Jazyk funkčního blokového schématu (Function Block Diagram) je založen na propojování funkčních bloků a funkcí. Stejně jako v jazyce LD jsou i v jazyce FBD funkce a funkční bloky reprezentovány obdélníkem. Rozdíl je v tom, že v jazyce LD lze spojnicemi mezi prvky přenášet pouze hodnoty typu BOOL zatímco v jazyce FBD mohou spojnice mezi grafickými prvky přenášet hodnoty libovolného typu.



Obr. 5.5 Grafika obvodu v jazyce FBD

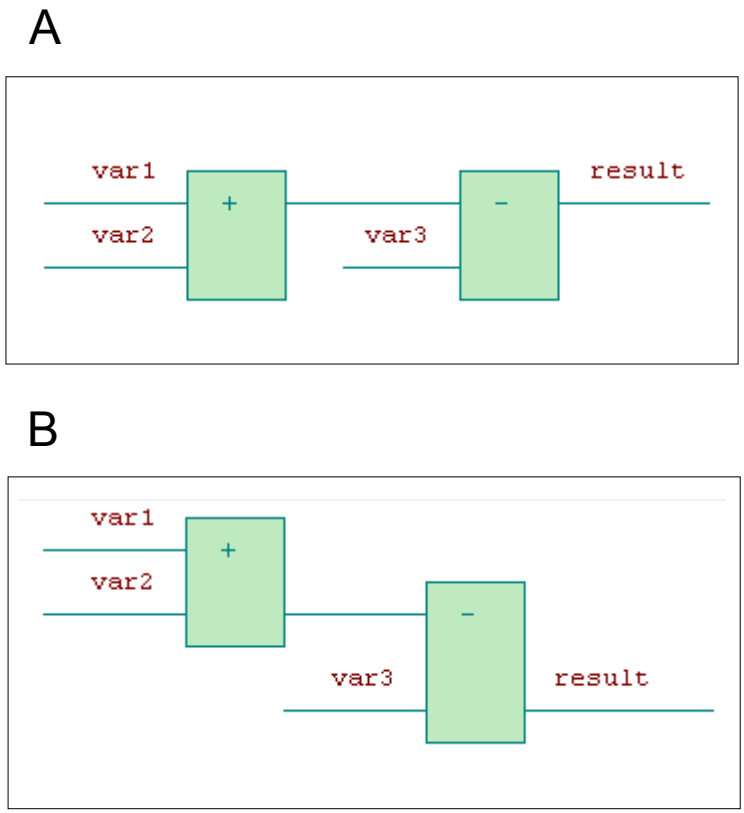
5.3.1 Grafické prvky v jazyce FBD

Obvod v jazyce FBD může obsahovat následující grafické prvky :

- spojnice
- grafické prvky pro řízení provádění programu (skoky)
- grafické prvky pro volání funkcí nebo funkčních bloků

Jazyk FBD neobsahuje žádné další grafické prvky jako jsou kontakty nebo cívky v jazyce LD. Prvky jazyka FBD se propojují spojnicemi toku signálu. Výstupy funkčních bloků se spolu nepropojují. Obzvlášť konstrukce “wired OR” jazyka LD není v jazyce FBD dovolena. Místo toho se používá blok boolovského OR.

Obvod v jazyce FBD může být vykreslen dvěma způsoby, jak je vidět na obr.5.6. Způsob zobrazení lze kdykoliv přepnout. Obvod realizuje výraz `result := (var1 + var2) – var3`.



Obr. 5.6 Způsoby zobrazení obvodu v jazyce FBD

Vyhodnocování toku signálu v obvodech

Tok signálu v obvodech je vyhodnocován zleva doprava. Při výpočtu programu jsou pak jednotlivé obvody v POU vyhodnocovány v pořadí shora dolů. V obvodu na obr.5.6 budou tedy nejprve sečteny proměnné **`var1`** a **`var2`** a poté bude odečtena proměnná **`var3`**. Výsledek bude uložen do proměnné **`result`**.

5.3.1.1 Spojnice v jazyce FBD

Prvky spojnic mohou být vodorovné nebo svislé. Stav spojnice představuje hodnotu připojené proměnné. Pojem *stav spojnice* je synonymem k pojmu *tok signálu*.

Vodorovná spojnice je indikována vodorovnou čarou. Vodorovná spojnice předává stav prvku, který je bezprostředně vlevo, k prvku, který je bezprostředně vpravo od něho.

Svislá spojnice se skládá ze svislé čáry připojující jednu nebo více vodorovných spojnic na pravé straně. Stav svislé spojnice se kopíruje do všech připojených vodorovných spojnic napravo od ní. Stav svislých spojnic se nekopíruje do žádné připojené vodorovné spojnice vlevo od ní.

Tab.5.9 Spojnice v jazyce FBD

Grafický objekt	Jméno	Funkce
	Vodorovná spojnice	Vodorovná spojnice kopíruje stav prvku připojeného na levé straně do prvku připojeného na pravé straně
	Svislá spojnice s vodorovnými připojeními	Stav vodorovné spojnice vlevo je kopírován do všech vodorovných spojnic vpravo
	Nedovolená konstrukce	Tato konstrukce (označovaná jako wired OR) není v jazyce FBD dovolena. Místo ní lze použít standardní funkci boolovského OR.

5.3.1.2 Řízení provádění programu v jazyce FBD

Pro řízení provádění programu máme stejně jako v jazyce LD dvě možnosti : skok na určitý obvod v aktuální POU a ukončení POU. Grafické symboly pro jazyk FBD jsou uvedeny v tab.5.10.

Skoky se znázorňují vodorovnou čarou ukončenou dvojitou šipkou. Předání řízení programu na určené návěští obvodu proběhne, pokud je boolovská hodnota spojovací čáry 1 (TRUE). Spojovací čára pro podmínku skoku může začínat u boolovské proměnné nebo u boolovského výstupu funkce nebo funkčního bloku. Pokud podmínka není uvedena jedná se o nepodmíněný skok. Cílem skoku je návěští sítě v rámci té programové organizační jednotky, v níž se skok objeví. Nelze tedy skákat jinak než v rámci jedné POU.

Podmíněné návraty z funkcí a funkčních bloků se implementují použitím konstrukce RETURN. Provádění programu se předá zpět do vyvolávající POU, pokud je boolovský vstup 1 (TRUE). Provádění programu bude pokračovat v normálním běhu, pokud má boolovský vstup hodnotu 0. Nepodmíněné návraty vzniknou na fyzickém konci funkce nebo funkčního bloku nebo pomocí nepodmíněného prvku RETURN.

Tab.5.10 Předání řízení programu v jazyce FBD

Grafický objekt	Jméno	Funkce
	Nepodmíněný skok (jump)	Skok na obvod s návěštím **Label**
	Podmíněný skok (Conditional jump)	Skok na obvod s návěštím **Label** jestliže proměnná **VarName** má hodnotu **TRUE**, jinak program pokračuje v řešení následujícího obvodu
	Nepodmíněný návrat z POU (Return)	Ukončí POU a vrátí řízení do volající POU. POU je rovněž ukončena, pokud se vyřeší všechny její obvody
	Podmíněný návrat z POU (Conditional return)	Ukončí POU a vrátí řízení do volající POU jestliže proměnná **VarName** má hodnotu **TRUE**, jinak program pokračuje v řešení následujícího obvodu

5.3.1.3 Volání funkcí a funkčních bloků v jazyce FBD

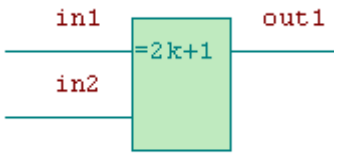
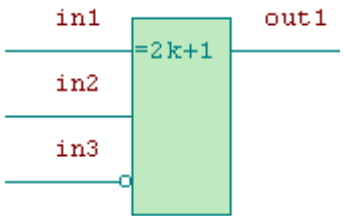
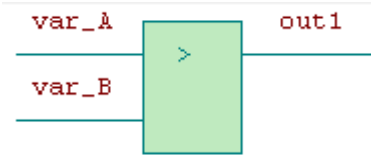
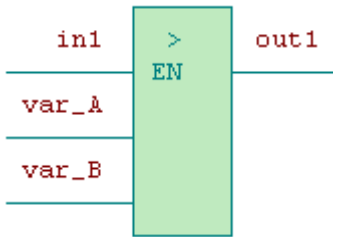
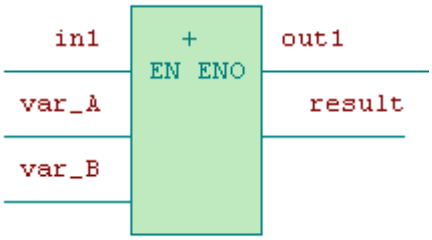
Grafická reprezentace funkcí a funkčních bloků je velmi podobná. Tyto POU jsou ve schématu reprezentovány obdélníkem stejně jako je tomu v jazyce LD. Vstupní proměnné jsou reprezentovány spojnicemi zleva, výstupní proměnné spojnicemi zprava. Jména vstupních a výstupních formálních parametrů jsou uvedena uvnitř obdélníku naproti spojnicím, přes které se připojují aktuální hodnoty parametrů (proměnné nebo konstanty). U rozšiřitelných funkcí (např. ADD, XOR, atd.) se jména vstupních parametrů neuvádějí. Jméno funkce nebo typ funkčního bloku je pak uvedeno v horní části obdélníku. Jméno instance funkčního bloku je uvedeno nad obdélníkem. Obdélníky funkcí jsou kresleny zeleně, funkční bloky jsou modré.

V jazyce FBD nemusí mít funkce nebo funkční blok žádný vstup typu BOOL proto, aby mohl být zapojen do obvodu. Použití implicitního vstupu EN není tedy v principu nutné, není však zakázáno. Obdobně to platí také pro implicitní výstup ENO. Pokud jsou EN a ENO použity, jejich význam a chování jsou stejné jako v jazyce LD.

Volání funkcí

Příklady volání funkcí v jazyce FBD jsou uvedeny v tab.5.11.

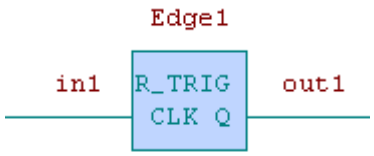
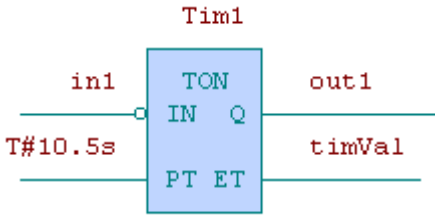
Tab.5.11 Volání funkcí v jazyce FBD

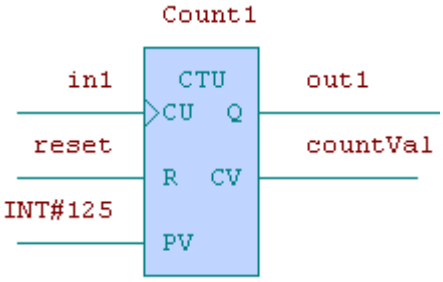
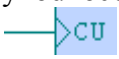
Obvod	Popis
	Volání standardní funkce **XOR** Obvod realizuje výraz `out1 := IN1 XOR in2`
	Volání standardní funkce **XOR** s rozšířeným počtem vstupů Obvod realizuje výraz `out1 := in1 XOR in2 XOR NOT in3`
	Volání funkce **GT** bez použití **EN** a **ENO** Obvod realizuje výraz `out1 := var_A > var_B`
	Volání funkce **GT** s použitím implicitního vstupu **EN**. Implicitní výstup **ENO** není použit. Jestliže má vstup **EN** hodnotu **TRUE** obvod realizuje výraz `out1 := var_A > var_B` Jinak se hodnota proměnné `out1` nepočítá.
	Volání funkce **ADD** s použitím implicitního vstupu **EN** a implicitního výstupu **ENO**. Jestliže má vstup **EN** hodnotu **TRUE** obvod realizuje výraz `result := var_A + var_B` Jinak se hodnota proměnné `result` nepočítá. Výstup **ENO** kopíruje stav vstupu **EN**.

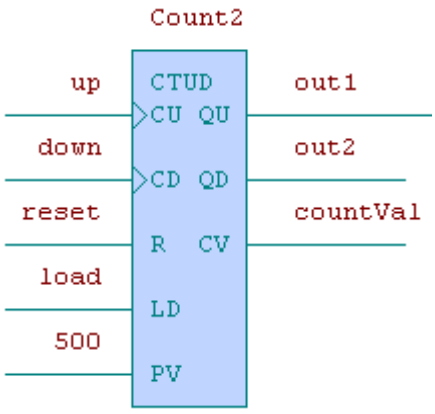
Volání funkčních bloků

Příklady volání funkcí v jazyce FBD jsou uvedeny v tab.5.12.

Tab.5.12 Volání funkčních bloků v jazyce FBD

Obvod	Popis
	Volání standardního funkčního bloku **R_TRIG** Výstup **out1** je nastaven pouze v případě přechodu proměnné **in1** z hodnoty **0** na hodnotu **1** (náběžná hrana)
	Volání standardního funkčního bloku **TON** Vstupní proměnná **in1** je negovaná. Vstupní proměnná **PT** (předvolba časovače) je typu **TIME** a do této proměnné zapisuje konstanta **T#10.5s** (10,5 sekundy)

	Volání standardního funkčního bloku **CTU**
	
	Vstup **CU** je ve funkčním bloku **CTU** definován následovně:
	`VAR_INPUT`
	`  CU : BOOL R_EDGE;`
	`END_VAR`
	Z toho důvodu je vstupní spojnice tohoto signálu ukončena značkou vyhodnocení náběžné hrany
	 

	Volání standardního funkčního bloku **CTUD** Vstupy **CU** a **CD** jsou typu **BOOL** s detekcí náběžných hran. Do vstupu **PV** (Preset Value) se zapisuje konstanta **500**. Výstup **CV** (Counter Value) je zapisován do proměnné **countVal**.

6 PŘÍLOHY

6.1 Direktivy

Programy zapsané v některém z textových jazyků mohou obsahovat direktivy pro překladač, které umožňují řídit jeho práci. Direktivy se zapisují do složených závorek.

Například direktiva **{$DEFINE new_name}** definuje jméno new_name.

6.1.1 Direktiva PUBLIC

Direktiva **{PUBLIC}** slouží k označení veřejné proměnné. Popis takto označené proměnné bude při překladu uložen do souboru s příponou „.pub“, který slouží pro přenos definic proměnných do vizualizačních programů apod.

Tyto direktivy lze používat v rámci deklarace datového typu nebo v rámci deklarace proměnné.

Syntaxe zápisu je následující:

```
TYPE MyINT {PUBLIC} : INT; END_TYPE
VAR
  Var1 {PUBLIC} : BOOL;
  Var2 {PUBLIC} AT %R2000 : BYTE;
END_VAR
```

6.1.2 Direktivy pro podmíněný překlad programu

Pro podmíněný překlad programu jsou zavedeny následující direktivy:

```
{$IF <vyraz>}
{$IFDEF <name>}
{$IFNDEF <name>}
{$DEFINE <name>}
{$UNDEF <name>}
{$END_IF}
{$ELSE}
{$DEFINED(<name>)}
{$ELSEIF <name>}
```

Tyto direktivy lze používat jak v deklarační tak i výkonné části programu.

6.1.2.1 Direktivy $IF … $ELSE … $END_IF

Direktiva **{$IF <výraz>}** je určena pro podmíněný překlad je-li splněn výraz. Může být volitelně podmíněna i větev **{$ELSE}**. Podmíněně překládaná část programu je ukončena direktivou **{$END_IF}**. Výraz musí obsahovat pouze proměnné definované jako **VAR_GLOBAL CONSTANT**, konstanty, nebo **{$DEFINED(<name>)}**. Operátory ve výrazu mohou být pouze:

'>'	- větší
'<'	- menší
'='	- rovno
NOT	- negace ve výrazu
AND	- booleovský součin
OR	- booleovský součet
')'	- závorka
'('	- závorka

Syntaxe zápisu je:

```
{$IF <výraz>} .... [{$ELSE}....] {$END_IF}
```

6.1.2.2 Direktivy $IFDEF a $IFNDEF

Tyto direktivy jsou určeny pro podmíněný překlad. Program, který následuje direktivou **{$IFDEF <name>}** je překládán v případě, že jméno uvedené v direktivě existuje (je definované). Naopak program uvedený za direktivou **{$IFNDEF <name>}** bude překládán pouze v případě, že jméno uvedené v direktivě není definované. Tyto direktivy lze kombinovat s direktivami **{$ELSE}** a **{$ELSEIF}** a vytvářet tak alternativně překládané části programů. Konec podmíněného překladu je označen direktivou **{$END_IF}.**

Syntaxe zápisu je:

```
{$IFDEF <name>} .... [{$ELSE}....] {$END_IF}
{$IFNDEF <name>} .... [{$ELSE}....] {$END_IF}
```

6.1.2.3 Direktivy $DEFINE a $UNDEF

Tyto direktivy jsou určeny pro přidání resp. odebrání definice jména. Direktiva **{$DEFINE <name>}** přidá definici jména **<name>**. Jméno pak lze použít v direktivách **{$IFDEF <name>}** a **{$IFNDEF <name>}**. Direktiva **{$UNDEF <name>}** zruší definice jména, uvedeného v direktivě.

Syntaxe zápisu je:

```
{$DEFINE <name>}
{$UNDEF <name>}
```

6.1.2.4 Direktiva DEFINED

Tato direktiva je určena pro test platnosti definice jména **<name>** a lze ji použít v kombinaci s direktivou **{$IF <výraz>}** jako součást výrazu.

Syntaxe zápisu je:
```
DEFINED (name)
```

Příklad:
```
{$IF DEFINED( ALFA) OR DEFINED( BETA)}
  VAR counter : INT; END_VAR
{$ELSE}
  VAR counter : DINT; END_VAR
{$END_IF}
```

6.1.3 Direktivy ASM a END_ASM

Direktiva **{ASM}** slouží k vložení programu v mnemokódu do programu v některém z IEC jazyků. Konec vkládaného mnemokódu je pak označen direktivou **{END_ASM}**.

Syntaxe zápisu je:
```
{ASM}
{END_ASM}
```

6.1.4 Direktiva ST_WARNING

Direktiva **{ST_WARNING}** slouží k potlačení varovných hlášení překladače ST. Direktiva **{ST_WARNING OFF}** označí místo v programu, od kterého budou varovná hlášení ST překladače potlačena. Direktiva **{ST_WARNING ON}** označí místo v programu, od kterého budou varovná hlášení ST překladače opět vydávána.

Syntaxe zápisu je:
```
{ST_WARNING ON}
{ST_WARNING OFF}
```

6.1.5 Direktiva OFFSET_REG

Direktiva **{OFFSET_REG=10000}** slouží k nastavení bázove adresy v paměti %R (%M), kam budou mapovány proměnné a instance. První proměnná bude umístěna na adrese o jednu vyšší, než je uvedeno v direktivě (%R10001). Direktiva **{END_OFFSET_REG}** ukončí posunutou alokaci proměnných. Umísťování proměnných v paměti PLC bude poté pokračovat na adrese o 2 vyšší, než byla před použitím direktivy **{OFFSET_REG=..}**.

Pozor!
Tyto direktivy vyřadí z činnosti automatickou kontrolu překrytí adres proměnných. Při překrytí proměnných nebude překladač hlásit žádnou chybu!

Syntaxe zápisu je:
{OFFSET_REG=xxx} kde xxx je adresa %R, kde začne nové mapování
{END_OFFSET_REG}

6.2 Rezervovaná klíčová slova

Následující tabulka uvádí klíčová slova, jejichž použití je rezervováno pro programovací jazyky IEC 61 131-3 a nelze je tedy použít pro uživatelsky definované symboly.

Tab.6.1 Rezervovaná klíčová slova

A	ABS ANY ANY_NUM AT	ACOS ANY_BIT ANY_REAL ATAN	ACTION ANY_DATE ARRAY	ADD ANY_INT ASIN
B	BOOL	BY	BYTE	
C	CAL CD CONFIGURATION CTU	CALC CDT CONSTANT CTUD	CALCN CLK COS CU	CASE CONCAT CTD CV
D	D DINT DT	DATE DIV DWORD	DATE_AND_TIME DO	DELETE DS
E	ELSE END_CONFIGURATION END_IF END_STEP END_VAR EQ EXPT	ELSIF END_FOR END_PROGRAM END_STRUCT END_WHILE ET	END_ACTION END_FUNCTION END_REPEAT END_TRANSITION EN EXIT	END_CASE END_FUNCTION_BLOCK END_RESOURCE END_TYPE ENO EXP
F	FALSE FOR	F_EDGE FROM	F_TRIG FUNCTION	FIND FUNCTION_BLOCK
G	GE	GT		
I	IF INT	IN INTERVAL	INITIAL_STEP	INSERT
J	JMP	JMPC	JMPCN	
L	L LEFT LN LWORD	LD LEN LOG	LDN LIMIT LREAL	LE LINT LT
M	MAX MOVE	MID MUL	MIN MUX	MOD
N	N	NE	NEG	NOT
O	OF	ON	OR	ORN
P	P PV	PRIORITY	PROGRAM	PT
Q	Q	QI	QU	QD

R	R READ_WRITE REPLACE RETC ROL R_EDGE	RI REAL RESOURCE RETCN ROR	R_TRIG RELEASE RET RETURN RS	READ_ONLY REPEAT RETAIN RIGHT RTC
S	S SEMA SINGLE SR STRING	ST SHL SINT ST STRUCT	SD SHR SL STEP SUB	SEL SIN SQRT STN
T	TAN TIME_OF_DAY TON TYPE	TASK TO TP	THEN TOD TRANSITION	TIME TOF TRUE
U	UDINT USINT	UINT	ULINT	UNTIL
V	VAR VAR_INPUT	VAR_ACCESS VAR_IN_OUT	VAR_EXTERNAL VAR_OUTPUT	VAR_GLOBAL
W	WHILE	WITH	WORD	
X	XOR	XORN		

OBSAH