

# TGMotion

## Virtuální PLC v6.2

### Obsah

<b>OBSAH</b> .....	<b>1</b>
<b>TEORETICKÝ ÚVOD</b> .....	<b>3</b>
<b>SDÍLENÁ PAMĚŤ</b> .....	<b>3</b>
PLC CONTROL MEMORY .....	3
<i>Status</i> .....	3
<i>Čítač cyklů</i> .....	4
PLC SHARED MEMORY .....	4
<i>Skupina typu Servo</i> .....	4
<i>Skupina typu Dio</i> .....	6
<i>Skupina typu Cnc</i> .....	7
<i>Skupina typu Spf</i> .....	8
<i>Skupina typu Write</i> .....	9
<i>Vnitřní časovače</i> .....	9
<i>Klávesnice a displej</i> .....	9
<i>EEPROM</i> .....	9
<i>Volná paměť</i> .....	9
<b>PROGRAMOVÁNÍ PLC</b> .....	<b>10</b>
DEKLARACE KONSTANT .....	10
DEKLARACE PROMĚNNÝCH.....	10
<i>Jednoduchý datový typ Integer</i> .....	10
<i>Strukturovaný datový typ Array</i> .....	10
<i>Klíčové slovo Absolute</i> .....	11
PŘEDDEFINOVANÉ KONSTANTY A PROMĚNNÉ.....	11
PRIORITA PROCESU .....	12
KLÍČOVÁ SLOVA .....	12
PROCESY .....	12
PŘÍRAZENÍ .....	13
MATEMATICKÉ OPERACE .....	14
LOGICKÉ A BITOVÉ OPERACE .....	14
RELACE.....	14
ŘÍDÍCÍ KONSTRUKCE .....	14
<i>Konstrukce If – Then – Else</i> .....	14
<i>Konstrukce While – Do</i> .....	15
<i>Konstrukce Repeat – Until</i> .....	15
<i>Konstrukce For – To – Do</i> .....	15
<i>Konstrukce Case – Of</i> .....	16
<i>Konstrukce Begin – End</i> .....	16
<i>Příkaz Break</i> .....	16
<i>Nepodmíněný skok GoTo</i> .....	17
PROCEDURY A FUNKCE .....	17
<i>Procedury</i> .....	17
<i>Funkce</i> .....	18
<i>Předdefinované procedury a funkce</i> .....	18
KOMENTÁŘE .....	18
DIREKTIVY KOMPILERU .....	19
<i>Použití direktivy DEFINE</i> .....	19

Řízení překladu pomocí direktiv <i>IFDEF, IFNDEF, ELSE, ENDIF</i> .....	19
Vkládání zdrojového textu pomocí direktivy <i>INCLUDE</i> .....	20
Nastavení přidělování paměti pomocí direktivy <i>M</i> .....	20
PARAMETRY PŘEKLADAČE.....	21
<i>Parametr W</i> .....	21
PŘÍKLADY PROGRAMŮ .....	21
<i>Příklad 1: Hello world</i> .....	21
<i>Příklad 2: Využití paměti EEPROM a procesu typu Interrupt</i> .....	21
<b>BNF DEFINICE JAZYKA.....</b>	<b>22</b>
ZÁKLADNÍ PRVKY .....	22
VÝRAZY .....	22
ŘÍDÍCÍ KONSTRUKCE .....	23
DEKLARACE KONSTANT A PROMĚNNÝCH.....	23
DEKLARACE PROCEDUR A FUNKCÍ .....	23
PROCESY .....	24
<b>WIN32 APLIKACE .....</b>	<b>25</b>
PŘÍSTUP KE SDÍLENÉ PAMĚTI.....	25
<i>API funkce RTX prostředí</i> .....	25
<i>Vytvoření sdílené paměti</i> .....	25
<i>Otevření sdílené paměti</i> .....	26
SPOLUPRÁCE S VIRTUÁLNÍM PLC .....	26
<i>Definice pro spolupráci s virtuálním PLC</i> .....	26
<i>Inicializace paměti</i> .....	26
<i>Nahrání nového programu do PLC</i> .....	27
<i>Displej virtuálního PLC</i> .....	27
<i>Klávesnice virtuálního PLC</i> .....	28
<b>HLÁŠENÍ CHYB.....</b>	<b>29</b>

## Teoretický úvod

Virtuální PLC umožňuje řídit stroje prostřednictvím vstupů, výstupů a digitálních servopohonů. Výkonná část virtuálního PLC běží v prostředí reálného času pod Windows. Pro interakci s obsluhou slouží druhá aplikace, která běží ve Windows v normálním režimu a komunikuje s Virtuálním PLC pomocí sdílené paměti.

Programování PLC se provádí ve vyšším programovacím jazyce, který je velmi podobný jazyku Pascal. Zdrojový kód je přeložen do assembleru a uložen v binární formě. Výsledný soubor má příponu BIN a do virtuálního PLC se nahrává pomocí sdílené paměti, jak bude popsáno níže.

Virtuální PLC pracuje s blokem paměti, ve kterém jsou stínovány registry vstupů, výstupů a rozhraní pro ovládání digitálních servopohonů. Tato paměť je také sdílená (aby byla dostupná pro ostatní aplikace) což umožňuje interakci s uživatelem a s ostatními programy.

Pro implementaci virtuálního PLC do konkrétní aplikace je tedy nutné napsat Win32 aplikaci, která bude zobrazovat důležité proměnné ze sdílené paměti (např. polohu serv, obsah displeje, apod.) a bude v paměti nastavovat potřebné registry (např. rychlost, číslo stisklé klávesy, apod.). Návod na vytvoření takové aplikace i s příkladem pro vývojové prostředí Borland Delphi bude uveden dále.

## Sdílená paměť

Pro vytvoření nebo otevření bloku sdílené paměti mezi Win32 a TGMotion aplikacemi je nutné použít API funkce z prostředí reálného času (*RtCreateSharedMemory*, *RtOpenSharedMemory*).

Po spuštění virtuálního PLC se v paměti vytvoří dva bloky sdílené paměti. Blok pojmenovaný *PLCControlMemory* slouží pro řízení funkce virtuálního PLC, blok *PLCSharedMemory* je zveřejněná hlavní paměť.

## PLC Control Memory

Tato paměť je velká 256B a je rozdělená takto:

Offset [B]	Velikost [B]	Význam
0	1	Status
1	1	Rezerva
2	1	Čítač cyklů
3	252	Jméno souboru

### Status

Bit	0	1	2	3	4	5	6	7
Význam	Load	Reset	Pause	Error	Manual In	Rezerva	Rezerva	Rezerva

*Load* – pokud je tento bit nastaven na 1, pak virtuální PLC nahraje do své paměti nový program, který je zadán v proměnné *Jméno souboru*. Kontrola tohoto bitu se provádí dvakrát za vteřinu, takže změna programu může mít drobné zpoždění. Jméno souboru musí být zadáno i s plnou cestou ve formátu programovacího jazyka C (nulou zakončený řetězec). Po úspěšném nahrání binárního souboru do paměti dojde k vynulování bitu *Load*. Pokud dojde při otevírání souboru k chybě, pak se nastaví bit *Error* na 1. Bit *Load* zůstane nastaven na 1 a v paměti zůstane nahraný původní program.

*Reset* – po nastavení tohoto bitu dojde k vynulování datové paměti a k novému spuštění programu. Po dokončení resetu virtuálního PLC dojde k vynulování tohoto bitu.

*Pause* – po nastavení tohoto bitu dojde k pozastavení vykonávání programu. K novému spuštění dojde až po vynulování tohoto bitu.

*Manual In* – nastavení vstupů. Pokud je *Manual In* nastaven na 0, aktualizuje virtuální PLC stínové vstupy a výstupy při každém cyklu (tzn. každou 1ms). Pokud je nastaven na 1, pak se přestanou vstupy aktualizovat a je možné je simulovat z jiného programu.

## Čítač cyklů

Tato proměnná se inkrementuje při každém cyklu virtuálního PLC. Slouží pro kontrolu správné funkce.

## **PLC Shared Memory**

Tato sdílená paměť je velká 64kB a je organizována jako 16384 proměnných typu Int32. Při startu a resetu PLC se tato paměť nuluje.

Offset [Int32]	Velikost [Int32]	Význam
0	1024	16 x skupina typu Servo
1024	384	6 x skupina typu Dio
1408	384	3 x skupina typu Cnc
1792	128	1 x skupina typu Spf
1920	1	Čas smyčky PLC
1921	3	1 x skupina typu Write
1924	168	Rezerva
2092	16	Vnitřní časovače
2108	16	Priority procesů
2124	1	Klávesnice
2125	256	Displej
2509	128	EEPROM
2637	383	Rezerva
3016	16384	Volná paměť

## Skupina typu Servo

Skupina umožňující monitorovat a ovládat jednotlivé servopohony připojené na sběrnici CAN. Číslo nódu na sběrnici CAN pak určuje číslo skupiny Servo. Systém je dimenzován na max. 16 servopohonů, v současné době je možné ovládat max. 6 servopohonů. Přehled jednotlivých členů skupiny typu Servo je uveden v tabulce.

Offset [int32]	Přístup	Jméno
0	R	Stav
1	R	Position
2	R	RefPosition
3	W	Reset
4	R/W	Offset
5	R/W	Mode
6	R	Error
7	R	WritePosition
8	R	SerialNumber
9	W	PG.Acc
10	W	PG.Dec
11	R/W	PG.APos
12	W	PG.DPos
13	R/W	PG.ASpeed
14	W	PG.PosSpeed
15	R/W	PG.Speed
16	R/W	PG.Mode
17	R/W	PG.Rdy
18	-	-

19	-	-
20	R/W	SDO.Control
21	R	SDO.Status
22	R/W	SDO.NumberByte
23	R/W	SDO.Index
24	R/W	SDO.SubIndex
25	R/W	SDO.Data
26	R/W	GEAR.Mode
27	R/W	GEAR.Position
28	R/W	GAER.Offset
29	W	GEAR.SourceNumber
30	W	GEAR.SourcePosition
31	W	GEAR.In
32	W	GEAR.Out
33	-	-
34	-	-
35	-	-
36	-	-
37	-	-
38	-	-
39	-	-
40	-	-
41	W	CNC.Number
42	W	CNC.NumberAxes
43	R	CNC.Offset
44	-	-
45	-	-
46	-	-
47	-	-
48	-	-
49	-	-
50	-	-
51	-	-
52	-	-
53	-	-
54	-	-
55	-	-
56	-	-
57	-	-
58	-	-
59	-	-
60	-	-
61	-	-
62	-	-
63	-	-

Přístup *R* znamená, že při každém cyklu virtuálního PLC se nahraje aktuální stav do této stínové paměti. Pokud program v PLC tuto hodnotu změní, pak ji změní jen ve stínové paměti a tato změna se nepromítne do řídicích registrů serv.

Při přístupu *R/W* se také při každém cyklu virtuálního PLC nahraje aktuální stav do této stínové paměti ale každá změna v této části paměti se okamžitě promítne do řídicích registrů serv.

Při přístupu *W* se při každém cyklu virtuálního PLC, změna dané proměnné typu Servo s přístupem *W* okamžitě promítne do řídicích registrů serv.

**Pozor!! Přepis je akceptován pouze z programu PLC, přímé zapsání do PLC shared memory neprovede zápis do řídicích registrů serv.** Přímý zápis je možný pomocí skupiny typu Write.

## Skupina typu Dio

Skupina umožňující monitorovat a ovládat jednotlivé kontroléry vstupů a výstupů připojených na sběrnici CAN. Číslo nódu na sběrnici CAN pak určuje číslo skupiny Dio. Systém je dimenzován na max. 6 kontrolérů, v současné době je možné ovládat max. 3 kontroléry. Jeden kontrolér pak může ovládat 64 digitálních vstupů, 64 digitálních výstupů, 4 analogové vstupy, 4 analogové výstupy, 1 seriovou linku RS422.

Stínování vstupů a výstupů se provádí tak, že před začátkem cyklu PLC se do paměti načtou aktuální stavy vstupů a na konci cyklu se na výstupy přepíší aktuální stavy ze stínové paměti. Digitální vstupy a výstupy jsou dostupné i pod symbolickými názvy I0, I1 .. I47 a O0, O1 .. O47.

Přehled jednotlivých členů skupiny typu Dio je uveden v tabulce.

Offset [int32]	Přístup	Jméno
0	W	Reset
1	R	Stav
2	R	NumberIn
3	R	In0
4	R	In1
5	R	In2
6	R	In3
7	R	In4
8	R	In5
9	R	In6
10	R	In7
11	R	NumberOut
12	W	Out0
13	W	Out1
14	W	Out2
15	W	Out3
16	W	Out4
17	W	Out5
18	W	Out6
19	W	Out7
20	R	NumberAI
21	R	AI0
22	R	AI1
23	R	AI2
24	R	AI3
25	R	NumberAO
26	W	AO1
27	W	AO2
28	W	AO3
29	W	AO4
30	-	-
31	-	-
32	-	-
33	-	-
34	-	-
35	-	-
36	-	-
37	-	-
38	-	-
39	-	-
40	-	-
41	-	-
42	-	-
43	-	-
44	W	RxPointerRead
45	R	RxPointerWrite

46	W	TxPointerRead
47	R	TxPointerWrite
48..55	R	RxBufer
56..63	W	TxBufer

Přístup *R* znamená, že při každém cyklu virtuálního PLC se nahraje aktuální stav do této stínové paměti. Pokud program v PLC tuto hodnotu změní, pak ji změní jen ve stínové paměti a tato změna se nepromítne do řídicích registrů kontrolérů.

Při přístupu *R/W* se také při každém cyklu virtuálního PLC nahraje aktuální stav do této stínové paměti ale každá změna v této části paměti se okamžitě promítne do řídicích registrů kontrolérů.

Při přístupu *W* se při každém cyklu virtuálního PLC, změna dané proměnné typu Dio s přístupem *W* okamžitě promítne do řídicích registrů kontrolérů.

**Pozor!! Přepis je akceptován pouze z programu PLC, přímé zapsání do PLC shared memory neprovede zápis do řídicích registrů kontrolérů.** Přímý zápis je možný pomocí skupiny typu Write.

### Skupina typu Cnc

Skupina umožňující monitorovat a ovládat jednotlivá virtuální CNC. Maximální počet současně pracujících virtuálních CNC jsou 3.

Přehled jednotlivých členů skupiny typu Cnc je uveden v tabulce.

Offset [int32]	Přístup	Jméno
0	R/W	Control
1	R	Stav
2	W	Trajectory
3..65	W	GKodName
66	R/W	PLCFunc
67	R	Line
68	R	NumberAxes
69	R	Position 1
70	R	Position 2
71	R	Position 3
72	R	Position 4
73	R	Positon 5
74	R	Position 6
75	R	TransPosition 1
76	R	TransPosition 2
77	R	TransPosition 3
78	R	TransPosition 4
79	R	TransPosition 5
80	R	TransPosition 6
81	W	GEAR.X1mod
82	W	GEAR.X1div
83	W	GEAR.Y1mod
84	W	GEAR.Y1div
85	W	GEAR.Z1mod
86	W	GEAR.Z1div
87	W	GEAR.X2mod
88	W	GEAR.X2div
89	W	GEAR.Y2mod
90	W	GEAR.Y2div
91	W	GEAR.Z2mod
92	W	GEAR.Z2div
93	W	DYNAMIC.Acc
94	W	DYNAMIC.Dec
95	W	DYNAMIC. EmergencyAcc

96	W	DYNAMIC.EmergencyDec
97	W	DYNAMIC.Speed
98	W	DYNAMIC.RelSpeed
99	R	GEN.Mode
100	R	GEN.Rdy
101	R	GEN.NumberTrajectory
102	R	GEN.Trajectory
103	R	GEN.NumberPart
104	R	GEN.Part
105	R	GEN.Inc
106	W	RefPosition 1
107	W	RefPosition 2
108	W	RefPosition 3
109	W	RefPosition 4
110	W	RefPositon 5
111	W	RefPosition 6
112	-	-
113	-	-
114	-	-
115	-	-
116	-	-
117	-	-
118	-	-
119	-	-
120	-	-
121	-	-
122	-	-
123	-	-
124	-	-
125	-	-
126	-	-
127	-	-

Přístup *R* znamená, že při každém cyklu virtuálního PLC se nahraje aktuální stav do této stínové paměti. Pokud program v PLC tuto hodnotu změní, pak ji změní jen ve stínové paměti a tato změna se nepromítne do řídicích registrů virtuálních Cnc.

Při přístupu R/W se také při každém cyklu virtuálního PLC nahraje aktuální stav do této stínové paměti ale každá změna v této části paměti se okamžitě promítne do řídicích registrů virtuálních Cnc.

Při přístupu W se při každém cyklu virtuálního PLC, změna dané proměnné typu Cnc s přístupem W okamžitě promítne do řídicích registrů virtuálních Cnc.

**Pozor!! Přepis je akceptován pouze z programu PLC, přímé zapsání do PLC shared memory neprovede zápis do řídicích registrů virtuálních Cnc.** Přímý zápis je možný pomocí skupiny typu Write.

## Skupina typu Spf

Skupina zastřešující skupiny CNC. Přehled jednotlivých členů skupiny typu Spf je uveden v tabulce.

Offset [int32]	Přístup	Jméno
0	R	NumberCNC
1..63	W	SpfName
64	R	CycleTime

Přístup *R* znamená, že při každém cyklu virtuálního PLC se nahraje aktuální stav do této stínové paměti. Pokud program v PLC tuto hodnotu změní, pak ji změní jen ve stínové paměti a tato změna se nepromítne do řídicích registrů Spf.

Při přístupu R/W se také při každém cyklu virtuálního PLC nahraje aktuální stav do této stínové paměti ale každá změna v této části paměti se okamžitě promítne do řídicích registrů Spf.

Při přístupu W se při každém cyklu virtuálního PLC, změna dané proměnné typu Cnc s přístupem W okamžitě promítne do řídicích registrů Spf.

**Pozor!! Přepis je akceptován pouze z programu PLC, přímé zapsání do PLC shared memory neprovede zápis do řídicích registrů Spf.** Přímý zápis je možný pomocí skupiny typu Write.

## Skupina typu Write

Skupina umožňující přímý zápis do proměnných typu W všech skupin typu Servo,Dio,Cnc a Spf mimo běh programu PLC. Určeno pro ladění PLC programů popř. pro přímé norealtime řízení z jakékoliv WIN32 aplikace. Přehled jednotlivých členů skupiny typu Write je uveden v tabulce.

Offset [int32]	Přístup	Jméno
0	R/W	ControlWrite
1	R/W	AdresaWrite
2	R/W	DataWrite

AdresaWrite určuje adresu proměnné sdílené paměti PLC (0..16384). DataWrite určuje data zapsaná do sdílené paměti PLC. Control Write provádí zápis do sdílené paměti PLC(1 - zápis vyžadován, 0 - zápis proveden).

## Vnitřní časovače

Pro vnitřní použití (procesy typu interrupt a procedura Delay) má každý proces přiřazen jeden časovač. Proto počet časovačů odpovídá maximálnímu počtu procesů (16). Časovače jsou také dostupné pod symbolickými jmény T0, T1 .. T15.

Časovače jsou dekrementovány při každém cyklu PLC (každou 1ms) až do nuly. Tuto vlastnost je možné využít při programování i přímo, ale je třeba dávat pozor aby nedocházelo ke konfliktům při používání jednoho časovače z několika různých částí programu. Obecně se přímé využívání časovačů nedoporučuje.

## Klávesnice a displej

Obsluha klávesnice se provádí takto:

- do přiděleného místa ve stínové paměti vloží řídicí Win32 aplikace číslo stisknuté klávesy
- program v PLC kontroluje obsah této proměnné pomocí funkce *Keypress*
- pokud *Keypress* vrátí *TRUE*, vyzvedne si program v PLC číslo klávesy pomocí funkce *ReadKey*
- funkce *Readkey* vrátí číslo stisknuté klávesy a automaticky vynuluje příslušnou proměnnou

Displej je maximálně čtyřřádkový. Každý řádek začíná počtem platných znaků, následuje 63 ASCII hodnot jednotlivých znaků. Aplikace v PLC by tuto část paměti měla obsluhovat pouze prostřednictvím procedury *Write*, řídicí Win32 aplikace ji interpretuje jako jednotlivé řádky displeje. Pokud není potřeba používat všechny řádky, může Win32 aplikace zobrazovat jen některé řádky. Příklad bude uveden dále.

## EEPROM

Tato část paměti je určena pro ukládání parametrů do souboru. Při každé změně v této oblasti se celá tato oblast uloží. K ukládání dochází dvakrát za sekundu. Soubor *EEPROM.BIN* se ukládá do hlavního adresáře.

## Volná paměť

Tato část je určena pro globální proměnné programu. K přidělování této paměti jednotlivým proměnným dochází při překladu.

## Programování PLC

Program v PLC se píše v programovacím jazyce, který je velmi podobný programovacímu jazyku Pascal. Jsou implementovány některé speciality, které umožňují psaní víceúlohových programů a přístup k jednotlivým bitům. Na druhou stranu je programátorovi k dispozici pouze jeden jednoduchý a jeden strukturovaný datový typ a jsou provedena některá další zjednodušení. Identifikátory mohou být složeny z písmen, čísel nebo znaku ‘\_’ a musí začínat písmenem. Velikost písmen nehraje roli.

### Deklarace konstant

Každý blok deklarací konstant začíná klíčovým slovem *CONST*. Deklarace se skládá z identifikátoru, znaku rovná se, přiřazené hodnoty a středníku.

Konstanty mohou být číselné, znakové nebo řetězcové. Číselné konstanty je možné používat místo proměnných, řetězcové a znakové jsou určeny pro využití s procedurou *Write*.

Příklad:

*CONST*

```
CISLO = 10 ;           {číselná konstanta}  
ZNAK1 = #65 ;         {znaková konstanta}  
ZNAK2 = "A" ;         {znaková konstanta}  
RETEZEC = 'Ahoj' ;   {řetězcová konstanta}
```

### Deklarace proměnných

Každý blok deklarací proměnných začíná klíčovým slovem *VAR*.

Jednotlivé deklarace se skládají ze seznamu identifikátorů proměnných oddělených čárkou, následuje dvojtečka a za ní datový typ. Poté může následovat klíčové slovo *Absolute* a umístění proměnné v paměti. Řádek je zakončen středníkem. Takovýchto řádků s deklaracemi může být v jednom bloku deklarací neomezené množství. Blok deklarací je možné zařadit na různá místa – mezi deklarace procesů, do deklarace procesu, do deklarace funkcí nebo procedur (i vkládaných).

#### Jednoduchý datový typ Integer

Jako jediný datový typ je implementován typ *Integer*. Je to 32-bitové znaménkové celé číslo.

Příklad:

*VAR*

```
A, B, C : Integer ;  
D : Integer ;
```

#### Strukturovaný datový typ Array

Pro práci s větším množstvím dat je určen typ *Array* (pole). Při jeho používání je nutné dávat pozor na přetečení indexů. Překročení deklarovaných mezí se za běhu nekontroluje.

Příklad:

*VAR*

```
P, Q : Array [1..10] of Integer;  
R : Array [0..100] of Integer;
```

## Klíčové slovo Absolute

Pomocí klíčového slova Absolute je možné proměnnou umístit na libovolné místo do paměti. Při použití tohoto klíčového slova se však v paměti nealokuje příslušné místo, jen se vytvoří nový název pro danou oblast paměti. Jako adresu v paměti je možné použít číslo, číselnou konstantu, nebo jméno jiné proměnné.

Příklad:

*CONST*

*POS = 10;*

*VAR*

*A : Integer ;*

*B : Integer Absolute 10;*

*C : Integer Absolute A;*

*D : Integer Absolute POS;*

*E : Array [1..10] of Integer;*

*F : Array [1..10] of Integer absolute B;*

*G : Array [1..10] of Integer absolute 10;*

*H : Array [1..10] of Integer absolute POS*

*I : Integer Absolute F;*

*J : Integer Absolute F[10];*

V tomto příkladu jsou ukázány možnosti přímého i nepřímého zadávání adres. Například proměnné A a C sdílí stejnou část paměti, proměnná J je posunuta o 10 oproti proměnné F, apod..

## **Předdefinované konstanty a proměnné**

Pro snadný přístup k některým částem paměti jsou předem předdefinované některé konstanty a systémové proměnné. Vnitřní definice odpovídají těmto definicím:

*CONST*

*CR = #13; {Carriage return – jdi na začátek řádku}*

*LF = #10; {Line feed – jdi na nový řádek}*

*VAR*

*{Hlavní paměť}*

*MEMORY : Array [0..16383] Integer Absolute 0;*

*{Paměť EEPROM}*

*EEPROM : Array [0..63] Integer Absolute 2509;*

*{Časovače}*

*T0 : Integer Absolute 2092;*

*...*

*T15 : Integer Absolute 2107;*

*{Vstupy}*

*I0 : Integer Absolute 1027;*

*...*

*I47 : Integer Absolute 1354 ;*

{Výstupy}

O0 : Integer Absolute 1036;

...

O47 : Integer Absolute 1363;

## Priorita procesu

Každému procesu je přiřazena proměnná, která určuje jeho prioritu při vykonávání programu. Jméno této proměnné je odvozeno od názvu procesu přidáním koncovky **\_PRIORITY**.

Počáteční hodnota této proměnné po spuštění programu je 50. To znamená, že vykonávání procesů se střídá po padesáti instrukcích od každého procesu. Modifikací této proměnné lze proces zvýhodnit nebo znevýhodnit na úkor ostatních procesů.

Příklad:

*Program Test;*

*Begin*

*Test\_Priority:=100; { Proces se zvýšenou prioritou }*

*End.*

## Klíčová slova

Identifikátory konstant, proměnných, procedur a funkcí musí být jednoznačné a nesmějí to být klíčová slova. Seznam klíčových slov:

ABSOLUTE	FALSE	PROCEDURE
AND	FOR	PROGRAM
ARRAY	FUNCTION	REPEAT
BEGIN	GOTO	THEN
BREAK	IF	TO
CASE	INTEGER	TRUE
CONST	INTERRUPT	UNTIL
DIV	LABEL	VAR
DO	MOD	WHILE
DOWNT0	NOT	WRITE
ELSE	OF	XOR
END	OR	

## Procesy

Virtuální PLC se chová jako šestnáct nezávislých procesorů, které sdílejí stejnou paměť. Každý naprogramovaný proces běží na jednom virtuálním procesoru. Strojový čas se dělí rovnoměrně mezi aktivní procesory, kterým je přiřazen proces. Pokud procesoru není přiřazen proces (nebo proces proběhne a skončí), pak je neaktivní a spotřebovává pouze minimum strojového času.

Jen pro hrubou představu o výkonu: během každého cyklu PLC (1ms) je vykonáno tisíc elementárních instrukcí. Zdrojový text v Pascalu se překládá na elementární instrukce zhruba v poměru 1:10. To by měl být dostatečný výkon pro většinu aplikací, přesto je dobré při rozdělování funkcí mezi jednotlivé procesy postupovat uvážlivě a časově kritickým funkcím přiřadit vlastní proces.

Nejjednodušší proces vypadá takto:

*Program Prvni;*

*Begin*

*End.*

Tento proces nic nevykonává a hned skončí. Aby měl nějaký smysl, musíme mezi klíčová slova *Begin* a *End* napsat vlastní výkonnou část.

Pokud chceme, aby se proces spouštěl v pravidelných intervalech, můžeme použít klíčové slovo *Interrupt*. Číslo za tímto klíčovým slovem udává interval v milisekundách, ve kterém se proces bude automaticky spouštět.

*Program Prvni; Interrupt 1000;*

*Begin*  
*End.*

Výkonná část programu se skládá z:

- přiřazení
- volání procedur
- řídicích konstrukcí

Cílem následujících kapitol je ukázat použití jazyka na příkladech. Přesnou syntaxi můžete najít v kapitole, která je věnovaná BNF definici jazyka.

## **Přiřazení**

Pomocí přiřazení můžeme měnit hodnotu proměnných. Na levé straně je vždy identifikátor proměnné typu integer nebo jeden její bit. Na pravé straně je výraz, který se může skládat z matematických a logických operací mezi proměnnými, konstantami a funkcemi.

Příklad:

*CONST*

*C = 10;*

*VAR*

*A,B : Integer;*

*D : Array[1..10] of Integer;*

*Program Pokus;*

*Begin*

*A := B;*

*A := B+C-2;*

*D[1] := A;*

*B := D[2];*

*End.*

Přístup k jednotlivým bitům je umožněn prostřednictvím tečkové konvence. Ve vztazích je možné kombinovat proměnné typu Integer s jednotlivými bity. Pokud použijeme bit na pravé straně, pak *true* má hodnotu  $-1$  a *false* má hodnotu  $0$ . Pokud přiřazujeme nějaký výraz na pravé straně nějakému bitu, pak jakákoliv nenulová hodnota pravé strany nastaví bit na *true*. Za tečkou lze používat i proměnné a konstanty.

Příklad:

*Program Pokus;*

*Begin*

*A := false;            { A=0 }*

*A.0 := 1;             { A=1 }*

*A.1 := true;          { A=3 }*

*A=A.0;                { A=-1 }*

*B.0 := D[2].1;*

*B.C := B.D*

*End.*

## Matematické operace

Protože jediný jednoduchý datový typ je celočíselný, jsou všechny matematické operace celočíselné.

Sčítání:	A:=B+C;
Odečítání:	A:=B-C;
Násobení:	A:=B*C;
Celočíselné dělení:	A:=B div C;
	A:=B / C;
Zbytek po celočíselném dělení:	A:=B mod C;

## Logické a bitové operace

Logický součet:	A:=B or C;
Logický součin:	A:=B and C;
Nonekvivalence:	A:=B xor C;
Rotace vlevo:	A:=B rol C;
Rotace vpravo:	A:=B ror C;
Posun vlevo:	A:=B shl C;
Posun vpravo:	A:=B shr C;

## Relace

Výsledkem relace je *true* nebo *false*. V celočíselném vyjádření je to -1 nebo 0.

Rovná se	A.0 := B=C;
Větší	A.0 := B>C;
Větší nebo rovná se	A.0 := B>=C;
Menší	A.0 := B<C;
Menší nebo rovná se	A.0 := B<=C;
Nerovná se	A.0 := B<>C;

## Řídící konstrukce

### Konstrukce If – Then – Else

Nejjednodušší z řídicích konstrukcí je podmínka. Pokud je výsledek výrazu za klíčovým slovem *If* *true*, pak je vykonán příkaz, který následuje za klíčovým slovem *Then*. V opačném případě je vykonán příkaz, který je za klíčovým slovem *Else*. Definice části s *Else* není povinná. Před *Else* se nepíše středník. Příklady:

```
If A=1 Then B:=0;  
If A>B Then C:=1 Else C:=0;
```

Pokud chceme vykonat více příkazů najednou, můžeme použít konstrukci *Begin – End*.

```
If A=1 Then  
  Begin  
    B:=0;  
    C:=1;  
  End;
```

**If  $A > B$  Then**

```
Begin
  B:=0;
  C:=1;
End
```

**Else**

```
Begin
  B:=1;
  C:=0;
End;
```

Řídící výrazy mohou být libovolně složité. Mohou obsahovat matematické a logické operace nebo volání funkcí. Operátory se vyhodnocují v tomto pořadí:

- Závorky
- Multiplikační operátory: \*, DIV, MOD, AND
- Aditivní operátory: +, -, OR, XOR
- Relační operátory: =, >, >=, <, <=, <>

Pokud si nejste jistí, jak se nějaký složitější výraz vyhodnotí, pak raději použijte ve sporných částech závorky.

### Konstrukce While – Do

Podmíněný cyklus s vyhodnocováním na začátku. Příkaz v tomto cyklu se opakuje tak dlouho, dokud výraz za klíčovým slovem *While* má hodnotu *True*. Pokud již na začátku má tento výraz hodnotu *False*, pak cyklus neproběhne ani jednou.

Příklad:

```
While A>1 Do A:=A-1;
While A>1 And B=0 Do A:=A-1;
```

Místo příkazu je samozřejmě možné vložit konstrukci Begin – End.

**While  $A > 1$  Do**

```
Begin
  A:=A-1;
End;
```

### Konstrukce Repeat – Until

Tato konstrukce je obdobná jako předchozí, jen k vyhodnocování dochází až na konci cyklu, takže cyklus proběhne minimálně jednou. Cyklus skončí, pokud podmínka platí.

Příklad:

```
Repeat
  A:=A-1;
Until A>1;
```

Navíc je možné psát přímo více příkazů. Není tedy nutné používat konstrukci Begin – End.

### Konstrukce For – To – Do

Nepodmíněný cyklus začíná klíčovým slovem *For* za kterým následuje přiřazení počáteční hodnoty do proměnné, která bude uvnitř cyklu použita pro čítání. Následuje klíčové slovo *To* pokud chceme čítat vzestupně nebo *DownTo* pro čítání sestupně. Poté následuje cílová hodnota proměnné, klíčové slovo *Do* a příkaz.

Příklad:

```
For A:=1 To 10 Do B:=B+1;  
For A:=10 DownTo 1 Do B:=B+1;
```

Opět je možné použít místo jednoho příkazu konstrukci Begin – End a v cyklu používat více příkazů.  
Příklad:

```
For A:=1 To 10 Do  
  Begin  
    B:=B+1;  
    C:=C-1;  
  End;
```

## Konstrukce Case – Of

Tato konstrukce je určena k náhradě podmíněného příkazu tam, kde je nutné porovnávat hodnotu jedné proměnné s mnoha různými. Porovnávat můžeme hodnotu proměnné nebo výsledek výrazu. Hodnoty, se kterými porovnáваме, je možné zadat jako seznam výrazů, které jsou oddělené čárkou. Místo příkazu je možné použít konstrukci Begin – End.

```
Case A Of  
  1      : B:=1;  
  2,3,10 : B:=2;  
  11     : Begin  
           B:=3;  
         End;  
End;
```

## Konstrukce Begin – End

Konstrukci Begin – End je možné psát kamkoli, kde je očekáván příkaz. Umožňuje nahradit jeden příkaz více příkazy tam, kde je to třeba.

## Příkaz Break

Pomocí příkazu *Break* lze ukončit řídicí blok. Jeho použití způsobí ukončení právě vykonávané úlohy, funkce, procedury nebo řídicí konstrukce.

Příklad:

```
Program test;  
Begin  
  ...  
  While a>0 Do  
    Begin  
      ...  
      If chyba then break; { Pokud nastane chyba, ukonci se vykonavani cyklu While-Do ... }  
    End;  
  ...  
End. { ... a program bude pokračovat zde }
```

## Nepodmíněný skok GoTo

Obecně pro přehlednost programu by se nepodmíněný skok měl používat co nejméně.

Použití nepodmíněného skoku je omezeno několika podmínkami:

- všechna návěští musí být předem definována a to lokálně (v bloku, kde je chceme použít)
- skákat lze pouze v rámci jednoho bloku (úloha, procedura, funkce)
- nepodmíněný skok by se neměl použít pro opuštění konstrukce *Case-Of*

Deklarace návěští se provádí v bloku deklarací pomocí klíčového slova *Label*. Takto předem definované návěští lze umístit libovolně do aktuálního bloku (stejně jako proceduru) ale pro odlišení se za identifikátor návěští píše místo středníku dvojtečka.

Skok na požadované návěští se provádí pomocí klíčového slova *GoTo* za kterým následuje identifikátor návěští.

Příklad:

```
Program Test;  
  Label Skok1, Skok2;
```

```
  Begin  
    Skok1:  
    ...  
    GoTo Skok1;  
  End.
```

## **Procedury a funkce**

Možnosti použití procedur a funkcí jsou poněkud omezené. Hlavní omezení vycházejí z toho, že funkce a procedury mají staticky definované vnitřní proměnné. Proto není možné je volat v jednom okamžiku z více různých procesů.

Aby se u globálně definovaných procedur a funkcí zaručilo, že k tomu nedojde, pozastaví se po dobu vykonávání procedury nebo funkce vykonávání všech ostatních procesů. Z tohoto důvodu je třeba dávat pozor, aby vykonávání globální funkce nebo procedury netrvalo příliš dlouho.

U procedur a funkcí definovaných lokálně pro jeden proces toto omezení neplatí.

Dalším omezením statického přístupu je, že funkce a procedury by neměli volat sami sebe (obzvláště pokud mají parametry nebo lokální proměnné). Proto se nedoporučuje rekurze.

## Procedury

Definice procedury vždy začíná klíčovým slovem *Procedure*. Poté následuje v závorce seznam parametrů a jejich typů (tato část je nepovinná). Následuje výkonná část mezi klíčovými slovy *Begin* a *End*.

Příklady:

```
Procedure Vynuluj;
```

```
  Begin  
    A:=0;  
    B:=0;  
  End;
```

```
Procedure Nastav(C : Integer);
```

```
  Begin  
    A:=C;  
    B:=C;  
  End;
```

## Funkce

U funkcí je to podobné, jen je třeba definovat v hlavičce návratový typ a na konci funkce je třeba přiřadit návratovou hodnotu do proměnné, která má stejné jméno a typ jako funkce.

**Function** *Vynuluj* : Integer;

**Begin**

*A:=0;*

*B:=0;*

*Vynuluj:=C;*

**End;**

**Function** *Soucet*(*A, B* : Integer) : Integer;

**Begin**

*Soucet:=A+B;*

**End;**

## Předdefinované procedury a funkce

**Procedure CLI** – tato procedura deaktivuje všechny ostatní procesy. Je určena k vyřešení situací, kdy by mohlo dojít ke konfliktu při obsluze proměnných z více procesů.

**Procedure STI** – opak procedury CLI, povoluje vykonávání ostatních procesů

**Function Keypressed** : Integer – tato funkce vrací *true* pokud byla stisknuta klávesa

**Function ReadKey** : Integer – tato funkce vrací číslo stisknuté klávesy

**Procedure Delay** (**Cas** : Integer) – tato funkce udělá pauzu na čas, který je zadán jako parametr v milisekundách. K vygenerování zpoždění využívá vnitřní časovač procesu, proto se nedoporučuje její použití v procesech typu Interrupt. Také by ji nelze použít v procedurách a funkcích, které jsou definovány globálně.

**Procedure Write** ( seznam parametrů ) – tato procedura slouží pro psaní na displej. Počet parametrů ani jejich typ nejsou omezené, viz příklady.

**Procedure Break** – způsobí ukončení aktuálního bloku. Týká se to procesů, procedur, funkcí, cyklů *While-Do*, *For-To-Do*, *Repeat-Until* a konstrukce *Case-Of*.

## **Komentáře**

Kamkoliv do zdrojového textu je možné vkládat komentáře. Komentář může být uzavřen složenými závorkami, kombinací lomítka a hvězdičky nebo je možné vytvořit komentář do konce řádku pomocí dvou lomítek. Různé druhy komentářů lze vzájemně kombinovat.

*Begin*

*{ Komentar 1 }*

*/\* Komentar 2 \*/*

*// Komentar do konce radku*

*End.*

## Direktivy kompilátoru

Pokud komentář ve složené závorce začíná znakem „\$“, očekává se direktiva kompilátoru. Pomocí těchto direktiv lze definovat makra nebo ovlivňovat zdrojový text před jeho překladem.

### Použití direktivy DEFINE

Pomocí direktivy kompilátoru *Define* je možné definovat symbolická jména nebo makra. Před překladem se takto nadefinované identifikátory nahradí předdefinovanou sekvencí.

Příklady:

```
{Toto je normalni komentar, nasleduje definice makra}  
{$Define Ahoj Write('Ahoj'); Delay(1000);}  
{$Define Vystup 00.1 }
```

Původní zdrojový text:

```
Program Test;  
Begin  
  Ahoj  
  Vystup:=1;  
End;
```

Před vlastním překladem se identifikátor Ahoj nahradí zbytkem závorky, takže to vypadá takto:

```
Program Test;  
Begin  
  Write('Ahoj');  
  Delay(1000);  
  00.1:=1;  
End;
```

Takto lze definovat symbolická jména i pro jednotlivé bity, což jiným způsobem nelze.

### Řízení překladu pomocí direktiv IFDEF, IFNDEF, ELSE, ENDIF

Při překladu je možné vynechávat části zdrojového textu. Můžeme si tak například při ladění vytvořit části kódu pro detekování chyb, které se při překladu do finální verze vyřadí. Při dalším ladění se opět snadno aktivují.

Pro tyto účely si nejprve musíme definovat nějaký symbol pomocí direktivy *Define*. Tento symbol může být prázdný, nebo může obsahovat nějaké makro. Poté si definujeme jednotlivé bloky programu, které budeme chtít ovlivňovat. Každý takový blok začíná direktivou *IfDef* nebo *IfNDef* a končí direktivou *EndIf*. Uprostřed může být volitelně direktiva *Else*.

Příklad:

```
{$DEFINE Ladeni}
```

```
Program Test;  
Begin
```

```
{$IFDEF Ladeni}  
    // tato cast programu bude prelozena  
{$ELSE}  
    // tato cast programu bude pri prekladu vynechana  
{$ENDIF}
```

```
{$IFDEF LADENI}  
// tato cast programu bude pri prekladu vynechana  
{$EndIf}
```

```
End.
```

Po odladění programu pak stačí odstranit definici symbolu *Ladeni*, (například jejím zakomentováním) a znovu přeložit finální verzi.

## Vkládání zdrojového textu pomocí direktivy INCLUDE

Pro zpřehlednění delších programů je možné rozdělit zdrojový text do více souborů. Všechny soubory se potom spojí do jednoho programu pomocí direktiv INCLUDE.

Zdrojový text ze zvoleného souboru se vloží do místa, kde je direktiva INCLUDE. Vkládat je možné libovolné množství souborů, a to i ve více úrovních. Jméno souboru může obsahovat i relativní nebo absolutní cestu.

Příklady:

```
{$Include pokus.pas}  
{$Include c:\plc\ouskovacka\casovace.pas} // Absolutni cesta na disku  
{$Include lib\casovace.pas} // Cesta do podadresare LIB v aktualnim adresari  
{$Include ..\lib\casovace.pas} // Cesta do adresare LIB v nadrazenem adresari
```

## Nastavení přidělování paměti pomocí direktivy M

Tato direktiva určuje adresu v paměti, od které se začne přidělovat paměť pro jednotlivé proměnné. Blok paměti před touto adresou je možné použít např. pro ruční přidělování pomocí direktivy Absolute.

Tato direktiva platí pro celý program bez ohledu na její umístění ve zdrojovém textu.

Upozornění: Na začátku paměti od adresy 0 je oblast registrů. Na tuto oblast nemá direktiva M vliv. Při ručním přidělování je nutné brát na to ohled a adresy 0..8 je lepší vůbec nepoužívat.

Příklad:

```
{$M 10}
```

```
var
```

```
A : Integer; // Tato proměnná bude umístěna na adrese 10
```

## Parametry překladače

### Parametr W

Pomocí parametru W (wait) lze nastavit čekání na stisk klávesy po dokončení překladu.

wd = čeká na stisk klávesy pokud proběhne překlad v pořádku

we = čeká na stisk klávesy pokud nastane chyba při překladu

w0 = ukončení programu hned po dokončení překladu

w1 = čeká na stisk klávesy pokud nastane chyba při překladu

w2 = čeká na stisk klávesy v každém případě

Příklad použití:

```
pascalcompiler hello.pas -wd //počká, pokud nedojde k chybě
pascalcompiler hello.pas -we //počká, pokud dojde k chybě
pascalcompiler hello.pas -we -wd //počká v obou případech
pascalcompiler hello.pas -w2 //počká v obou případech
```

## Příklady programů

### Příklad 1: Hello world

Tento program pouze vypíše na obrazovku pozdrav a skončí.

```
program HelloWorld;
begin
  write('Hello World!',LF);
end.
```

### Příklad 2: Využití paměti EEPROM a procesu typu Interrupt

Tento program se skládá ze dvou procesů. První proces po spuštění inkrementuje počet spuštění a vypíše ho v rámci uvítání na displej. Poté v nekonečné smyčce testuje klávesnici a vypisuje na displej stisknuté klávesy. Druhý proces zajišťuje blikání výstupu 00.0 v sekundovém intervalu.

```
program hlavni; {Hlavni program}
begin
  EEPROM[0]:=EEPROM[0]+1;
  write(LF,'Vítejte');
  delay(1000);
  write(LF,'Spusteni cislo ',EEPROM[0]);
  delay(1000);
  write(LF,'Stisknete cokoli ... ');
  delay(1000);
  write(LF);
  while true do
    begin
      while not keypressed do;
      write(CR,'Stisknuta klavesa ',readkey)
    end;
end.
```

```
program blikej; interrupt 1000; {Blikani}
begin
  00.0:=not 00.0;
end.
```

## BNF definice jazyka

### Základní prvky

<b>&lt;digit&gt;</b>	<code>:= '0' '1' '2' '3' '4' '5' '6' '7' '8' '9'</code>
<b>&lt;hexdigit&gt;</b>	<code>:= '0' '1' '2' '3' '4' '5' '6' '7' '8' '9' 'a' 'b' 'c' 'd' 'e' 'f'</code>
<b>&lt;letter&gt;</b>	<code>:= 'a' 'b' 'c' 'd' 'e' 'f' 'g' 'h' 'i' 'j' 'k' 'l' 'm' 'n' 'o' 'p' 'q' 'r' 's' 't' 'u' 'v' 'w' 'x' 'y' 'z'</code>
<b>&lt;sign&gt;</b>	<code>:= '+' '-'</code>
<b>&lt;addition-operator&gt;</b>	<code>:= '+' '-' 'or' 'xor'</code>
<b>&lt;multipl-operator&gt;</b>	<code>:= '*' 'div' 'mod' 'and'</code>
<b>&lt;relational-operator&gt;</b>	<code>:= '=' '&lt;' '&gt;' '&lt;=' '&gt;=''</code>
<b>&lt;special&gt;</b>	<code>:= '(' ')' ':' ';' '\' '=' +' '-' '*' '/' '&gt;' '&lt;' '&gt;=' '&lt;=' '&lt;' '&gt;' '..'</code>
<b>&lt;bool-literal&gt;</b>	<code>:= 'true' 'false'</code>
<b>&lt;decnumber&gt;</b>	<code>:= &lt;digit&gt;+</code>
<b>&lt;hexnumber&gt;</b>	<code>:= '\$[&lt;hexdigit&gt;]+'</code>
<b>&lt;number&gt;</b>	<code>:= &lt;decnumber&gt;   &lt;hexnumber&gt;</code>
<b>&lt;identifier&gt;</b>	<code>:= &lt;letter&gt; [&lt;letter&gt;   &lt;digit&gt;]+'</code>
<b>&lt;variable&gt;</b>	<code>:= &lt;identifier&gt; [ '.' &lt;number&gt; ]   &lt;identifier&gt; '[' &lt;expression&gt; ']' [ '.' &lt;number&gt; ]</code>
<b>&lt;constant&gt;</b>	<code>:= &lt;identifier&gt;</code>
<b>&lt;keyword&gt;</b>	<code>:= 'program' 'begin' 'end' 'if' 'then' 'else' 'for' 'to' 'downto' 'do' 'while' 'repeat' 'until' 'div' 'mod' 'true' 'false' 'and' 'or' 'xor' 'not' 'const' 'var' 'integer' 'case' 'of' 'procedure' 'function' 'absolute' 'interrupt' 'array'</code>
<b>&lt;comment&gt;</b>	<code>:= '{' [ &lt;character&gt; ]* '}'</code>

### Výrazy

<b>&lt;factor&gt;</b>	<code>:= 'not' &lt;factor&gt;   '(' &lt;expression&gt; ')'   &lt;variable&gt;   &lt;function&gt;   &lt;constant&gt;   'true'   'false'</code>
<b>&lt;term&gt;</b>	<code>:= &lt;factor&gt; [&lt;multipl-operator&gt; &lt;factor&gt;]</code>
<b>&lt;simple-expression&gt;</b>	<code>:= [ &lt;sign&gt; ] &lt;term&gt; [&lt;addition-operator&gt; &lt;term&gt; ]*</code>
<b>&lt;expression&gt;</b>	<code>:= &lt;simple-expression&gt; [&lt;relational-operator&gt; &lt;simple-expression&gt; ]*</code>
<b>&lt;procedure&gt;</b>	<code>:= &lt;identifier&gt; '[' &lt;expression&gt; [ '.' &lt;expression&gt; ]* ']'</code>

## Řídící konstrukce

<b>&lt;statement&gt;</b>	:= nothing   <compound-statement>   <assignment>   <procedure>   <if-statement>   <while-statement>   <for-statement>   <case-statement>
<b>&lt;assignment&gt;</b>	:= <variable> '=' <expression>
<b>&lt;if-statement&gt;</b>	:= 'if' <expression> 'then' <statement> [ 'else' <statement> ]
<b>&lt;compound-statement&gt;</b>	:= 'begin' <statement-sequence> 'end'
<b>&lt;while-statement&gt;</b>	:= 'while' <expression> 'do' <statement>
<b>&lt;repeat-statement&gt;</b>	:= 'repeat' <statement> 'until' <expression>
<b>&lt;for-statement&gt;</b>	:= 'for' <assignment> 'to' 'downto' <expression> 'do' <statement>
<b>&lt;case-label-list&gt;</b>	:= <expression> [ ',' <expression> ]*
<b>&lt;case-limb&gt;</b>	:= <case-label-list> ':' <statement> ';'
<b>&lt;case-statement&gt;</b>	:= 'case ' <expression> 'of' [ <case-limb> ]+ 'end'

## Deklarace konstant a proměnných

<b>&lt;declaration&gt;</b>	:= [ <const-declaration-part>   <var-declaration-part>   <procedure-declaration>   <function-declaration> ]*
<b>&lt;const-declaration-part&gt;</b>	:= 'const' [ <identifier> '=' <number> ';' ]*
<b>&lt;type&gt;</b>	:= 'integer' 'array'
<b>&lt;identifier-list&gt;</b>	:= <identifier> [ ',' <identifier> ]*
<b>&lt;var-declaration&gt;</b>	:= <identifier-list> ':' <type> [ 'absolute' <number>   <variable> ]
<b>&lt;var-declaration-part&gt;</b>	:= 'var' [ <var-declaration> ";" ]*

## Deklarace procedur a funkcí

<b>&lt;statement-sequence&gt;</b>	:= <statement> [ ';' <statement> ]*
<b>&lt;formal-parameter-list&gt;</b>	:= '(' <var-declaration> [ ';' <var-declaration> ] ')'
<b>&lt;procedure-heading &gt;</b>	:= 'procedure' <identifier> [ <formal-parameter-list> ]
<b>&lt;procedure-declaration&gt;</b>	:= <procedure-heading> ';' [ <declaration> ] <compound-statement> ';'
<b>&lt;function-heading &gt;</b>	:= 'function' <identifier> [ <formal-parameter-list> ] ':' <type>
<b>&lt;function-declaration&gt;</b>	:= <function-heading> ';' [ <declaration> ] <compound-statement> ';'

## **Procesy**

**<program-block>** := 'begin' <statement-sequence> 'end.'

**<program>** := [<declaration>] [ 'program' <identifier> ';' [ 'interrupt' <number> ';' ]  
 [<declaration>] <program-block> ]\*

## Win32 aplikace

Pro interakci s uživatelem není možné použít přímo aplikaci napsanou v real-time režimu. Je nutné napsat standardní Win32 aplikaci, která bude komunikovat s virtuálním PLC pomocí sdílené paměti. Po startu tato aplikace nahraje přeložený program do virtuálního PLC a poté vytváří „tvář“ celého stroje.

Řídící Win32 aplikaci lze napsat v libovolném vývojovém prostředí ze kterého lze otevřít sdílenou paměť. Jako nejvhodnější pro tyto účely se jeví RAD nástroje jako jsou Borland Delphi, Borland C++ Builder nebo Microsoft Visual Studio .NET. tato kapitola se bude věnovat vývoji řídicí aplikace v prostředí Borland Delphi.

### **Přístup ke sdílené paměti**

#### API funkce prostředí TGMotion

Pro použití sdílené paměti je třeba použít funkce, které nabízí prostředí TGMotion. Pro prostředí s programovacím jazykem C++ lze využít pro přístup k TGMotion API funkcím přímo dodávané hlavičkové soubory. V prostředí Borland Delphi je situace poněkud odlišná. Všechny používané funkce, proměnné a konstanty je třeba znovu definovat s ohledem na rozdíly typů mezi jazyky C++ a Pascal.

Definice potřebné pro práci se sdílenou pamětí v Delphi vypadají takto:

*CONST*

```
SHM_MAP_WRITE = 2;  
SHM_MAP_READ = 1;  
SHM_MAP_ALL_ACCESS = (SHM_MAP_WRITE + SHM_MAP_READ);  
PAGE_READONLY = 2;  
PAGE_READWRITE = 4;
```

```
function RtCreateSharedMemoryA(flProtect: dword;  
                             dwMaximumSizeHigh: dword;  
                             dwMaximumSizeLow: dword;  
                             lpName: pchar;  
                             var location: Pointer): dword stdcall; external 'rtapi_w32.dll';
```

```
function RtOpenSharedMemoryA(dwDesiredAccess: dword;  
                             bInheritHandle: dword;  
                             lpName: pchar;  
                             var location: pointer): dword stdcall; external 'rtapi_w32.dll';
```

#### Vytvoření sdílené paměti

*TYPE*

```
tcontrol=array[0..255] of char;  
pcontrol=^tcontrol;  
control:pcontrol;
```

*VAR*

```
PLCMemory:dword;
```

*Procedure* *Init*;

*begin*

```
PLCMemory := RtCreateSharedMemoryA(PAGE_READWRITE, 0,256, 'PLCControlMemory'  
                                   ,pointer(control));
```

```
if PLCMemory=0 then
```

*begin*

```
{Chyba při otevírání sdílené paměti}
```

*end*

```
else  
begin  
  {V pořádku}  
end;  
end;
```

## Otevření sdílené paměti

*Procedure Init;*

```
begin  
  PLCMemory := RtOpenSharedMemoryA(SHM_MAP_WRITE, 0, 'PLCControlMemory',pointer(control));  
  if PLCMemory=0 then  
    begin  
      {Chyba při vytváření sdílené paměti}  
    end  
  else  
    begin  
      {V pořádku}  
    end;  
end;
```

## **Spolupráce s virtuálním PLC**

### Definice pro spolupráci s virtuálním PLC

*CONST*

```
Display_Length      =          256;  
Display_Line_Length =          64;
```

```
MEM_Servo           =          00000;  
MEM_Timers          =          02092;  
MEM_Key             =          02124;  
MEM_Display         =          02125;  
MEM_EEPROM          =          02509;  
MEM_Variable_begin =          03016;  
MEM_Variable_end   =          16384;  
MEM_END             =          16384;
```

*TYPE*

```
memory=array[0..MEM_END-1] of longint;  
pmemory=^memory;  
tcontrol=array[0..255] of char;  
pcontrol=^tcontrol;
```

*VAR*

```
memory:pmemory;  
control:pcontrol;  
PLCControlMemoryAllocated:boolean;  
PLCSharedMemoryAllocated:boolean;
```

### Inicializace paměti

Před použitím sdílené paměti je nutné ji nejprve otevřít. Aby se zajistila korektní funkce i v okamžiku, kdy ještě není spuštěno virtuální PLC, je optimální vložit následující část kódu do časovače s periodou řádově ve stovkách milisekund.

```
if not PLCControlMemoryAllocated then
begin
  PLCMemory := RtOpenSharedMemoryA(SHM_MAP_WRITE, 0, 'PLCControlMemory',pointer(control));
  if PLCMemory=0 then
  begin
    Line4.Caption:='PLC control shared memory not found.';
    Line3.Caption:='You must run Virtual PLC first.';
    Line2.Caption:='';
    Line1.Caption:='';
    PLCControlMemoryAllocated:=false;
  end
  else PLCControlMemoryAllocated:=true;
end;
```

```
if not PLCSharedMemoryAllocated then
begin
  PLCMemory := RtOpenSharedMemoryA(SHM_MAP_WRITE, 0, 'PLCSharedMemory',pointer(memory));
  if PLCMemory=0 then
  begin
    Line4.Caption:='PLC shared memory not found.';
    Line3.Caption:='You must run Virtual PLC first.';
    Line2.Caption:='';
    Line1.Caption:='';
    PLCSharedMemoryAllocated:=false;
  end
  else PLCSharedMemoryAllocated:=true;
end;
```

## Nahrání nového programu do PLC

Po startu aplikace je nutné nahrát do PLC program pomocí sdílené paměti PLCControlMemory. Nejprve musíme připravit do sdílené paměti jméno binárního souboru i s celou cestou. Jméno se do PLC předává jako nulou zakončený string. Poté vynulujeme bit *Error* a nastavíme bit *Load* ve stavovém registru a počkáme na výsledek. Pokud proběhne nahrání nového programu v pořádku, vynuluje PLC bit *Load*. Při chybě se nastaví bit *Error*. Celá operace by měla skončit do jedné vteřiny.

```
for i:=1 to length(FileName) do control[i+2]:=FileName[i];
control[length(FileName)+3]:=#0;
control[0]:=char(byte(control[0]) and not 2);
control[0]:=char(byte(control[0]) or 1);
```

Pokud změníme program a chceme ho znovu nahrát do PLC, stačí jen znovu nastavit bit *Load*.

## Displej virtuálního PLC

Displej PLC je ve sdílené paměti organizován do čtyř řádků. Každý řádek má maximálně 63 znaků. Před začátkem řádku je uložen počet platných znaků v řádku. Zobrazování obsahu displeje je nutné provádět dostatečně často. K tomu je vhodné použít časovač.

Pro překreslení jednoho řádku displeje slouží následující část kódu. Proměnná *displej* je pomocná proměnná typu string, proměnná *radek* je číslo požadovaného řádku (0..3).

```
if PLCControlMemoryAllocated and PLCSharedMemoryAllocated then
begin
  displej:='';
  for i:=1 to memory[MEM_Display+radek*Display_Line_Length] do
  begin
```

```
displej:=displej+chr(memory[MEM_Display+radek*Display_Line_Length+i])  
end;  
Line1.Caption:=displej;  
end;
```

### Klávesnice virtuálního PLC

Pokud má běh programu v PLC ovlivňovat uživatel pomocí tlačítek na ovládacím panelu, pak nejjednodušším způsobem, jak toho dosáhnout je přiřadit každému tlačítku unikátní číslo do property *Tag* a do obsluhy události *ButtonClick* dát následující kód.

```
procedure THlavniPanel.Button1Click(Sender: TObject);  
var  
  temp:TButton;  
  
begin  
  if PLCControlMemoryAllocated and PLCSharedMemoryAllocated then  
    begin  
      temp:=Sender as TButton;  
      memory[MEM_Key]:=temp.tag;  
    end;  
end;
```

## Hlášení chyb