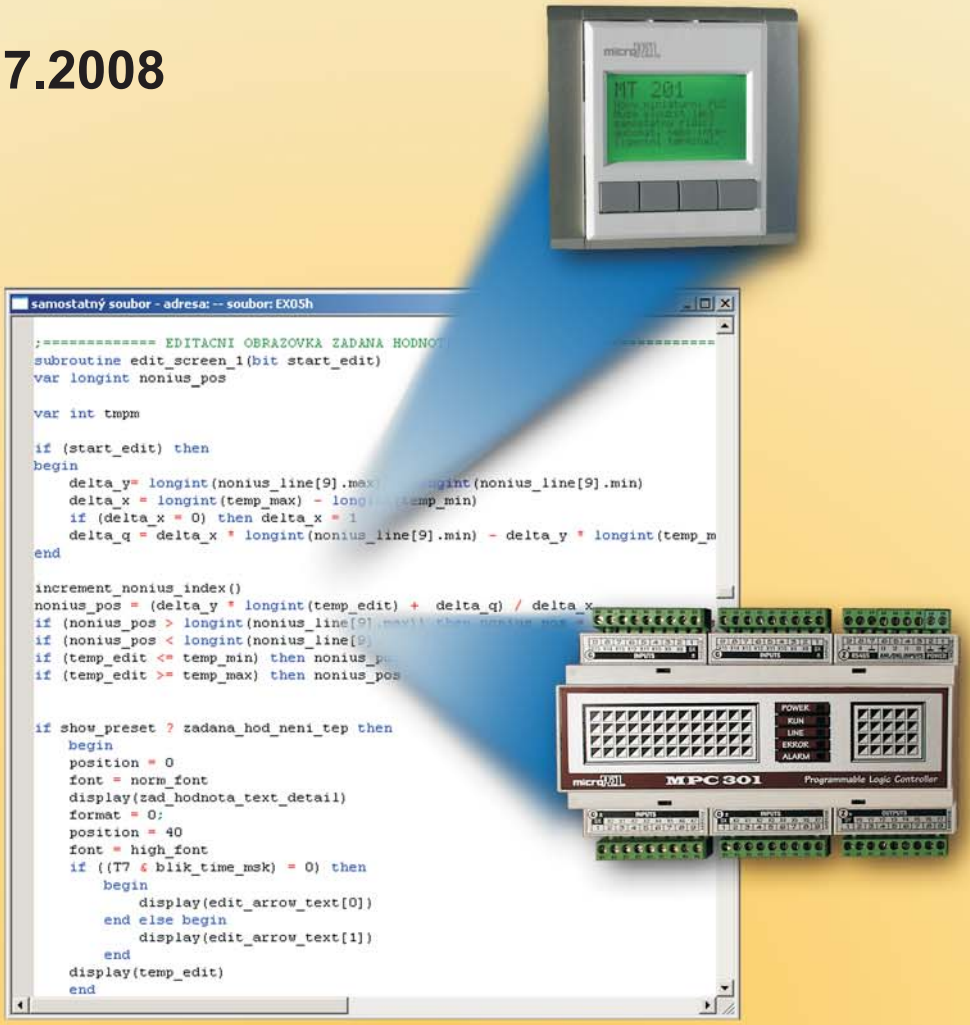
microPEL

SIMPLE 4

Praktická příručka programových konstrukcí jazyka SIMPLE 4 pro začátečníky i pokročilé

7.2008



STRUČNÝ PRŮVODCE JAZYKEM SIMPLE 4

Publikace shrnuje do tématických okruhů vlastnosti a použití jazyka Simple 4 pro programování řídicích automatů MICROPEL. Jazyk je popsán na řešených příkladech doplněných praktickým komentářem.

edice 07-2008
verze 1.3

Stručný průvodce jazykem Simple 4

http:/www.micropel.cz

Obsah

1 Úvod do jazyka ... 4
1.1 Proměnné a konstanty ... 4
1.2 Systémové proměnné ... 5
Vstupy a výstupy ... 5
Datové proměnné ... 5
Speciální funkční registry ... 6
Časovače ... 6
Reálný čas ... 7
Klávesnice ... 7
Displej ... 9
Systémové proměnné ... 11
2 Syntaxe jazyka Simple 4 ... 12
Ovládání vstupů a výstupů ... 12
Uživatelská jména vstupů a výstupů ... 13
Sdílené proměnné ... 14
Pravidla pro uživatelská jména ... 15
Deklarace a použití konstant a proměnných ... 16
Deklarace a použití polí ... 17
Řetězce znaků jako zvláštní typ polí ... 18
Uživatelské znaky ... 19
Formátování řetězců ... 21
Uživatelské datové typy ... 22
Struktury dat ... 23
Tabulky dat ... 24
Deklarace procedury a funkce ... 25
Předávání parametrů, zápis těla funkcí a podprogramů ... 26
Bitové operace ... 27
Aritmetické operace a operace bitových manipulací ... 28
Přiřazení, výraz, složený výraz ... 29
Přístup k bitu ... 30
Podmíněný příkaz ... 31
Použití bitu RESET ... 32
Logický výraz ... 33
Programový přepínač ... 34
Přetypování ... 35
Fixace proměnných ... 36
Konstrukce absolute ... 37
3 Základní úlohy z programování v jazyce Simple 4 ... 38

Jak funguje zobrazování 38
Omezení počtu tisků na displej 39
Světelná signalizace poruchy 40
Časovač zpoždění 42
4 Závěr 44
Rejstřík 45

1 Úvod do jazyka

Programovací jazyk Simple 4 je určen pro programování řídicích automatů MICROPEL vyšších typových řad např. MPC, K apod. Jazyk v maximální míře usnadňuje vývoj řídicích algoritmů a jejich implementaci v programovatelných automatech uvedených typových řad.

Vzhledem k tomu, že jazyk Simple 4 je navržen pro řešení průmyslových řídicích algoritmů, zachovává si všechny charakteristiky jazyka bezpečného programování a běhu programu v reálném čase. Program je vždy vykonáván od začátku až do posledního příkazu a od tohoto bodu se vrací vykonávání programu zpět na začátek. Programovací jazyk není vnitřně vybaven na programování interních programových smyček a neumožňuje programovat žádné skoky. Tím se do značné míry potlačuje vznik takových fatálních chyb, jako je například zacyklení nebo bloudění kódu. Konstrukce jazyka tak nutí programátora používat programovací obraty, které odpovídají struktuře tzv. stavového stroje. Výsledný program se pak blíží konstrukci logického automatu z klasických logických obvodů a to vše při zachování variability řešení dané právě programovými nástroji jazyka Simple 4.

1.1 Proměnné a konstanty

Jako každý programovací jazyk, pracuje i jazyk Simple 4 s **proměnnými, konstantami, výrazy, podmínkami, podprogramy, funkcemi** atp. Pro uchování a předdefinování hodnot slouží v jazyce Simple 4 proměnné a konstanty.

Proměnné můžeme třídit podle několika hledisek, přičemž jedním z nich je třídění podle velikosti či rozsahu zobrazovaných čísel. Toto dělení ukazuje Tab. 1.

TYP (Klíčové slovo)	ČÍSELNÝ ROZSAH
bit	0 / 1
byte	0 - 255
(safe) word	0 - 65535
(safe) int	-32767 - 32768
(safe) longword	0 - 4294967296
(safe) longint	-2147483648 - 2147483647
float	~ -10E38 - 10E38
string	řetězec znaků

Tab. 1 Základní datové typy jazyka Simple 4

Proměnné můžeme z hlediska chování a vlastností dělit na **systémové** a **uživatelské**. **Systémové** jsou ty, které má předdefinovány systém automatu pro svoji ale i pro uživatelskou potřebu. Jedná se o proměnné, které **slouží k ovládání vstupů, výstupů, časovačů, reálného času a některých dalších funkcí automatu**. Všechny tyto **proměnné doplňuje** na systémové úrovni **zásobník,** tj. pole položek typu „word“ (typicky se jedná o 11776 položek).

K základním typům proměnných řadíme v jazyce Simple 4 i tzv. bezpečné varianty označené klíčovým slovem „safe“. Tyto bezpečné varianty se vyznačují tím, že pokud s nimi provádíme aritmetické operace, jsou výpočty omezeny maximální a minimální hodnotou daného datového typu a aritmetika tedy nemůže přetéci. Jedná se o tzv. aritmetiku s „limiterem“.

Uživatelské proměnné jsou pak ty, které **definuje programátor aplikace**. Tyto proměnné jsou umísťovány do datové paměti RAM automatu, která svou velikostí současně množství uživatelských proměnných omezuje.

Programové konstanty mohou být buď číselné nebo textové. Konstanty se umísťují do programové paměti automatu konstrukce FLASH EEPROM, která zajišťuje neměnnost hodnoty i bez přítomnosti napájecího napětí a bez nutnosti doplňkových opatření.

Číselné konstanty, které nemají specifikován datový typ uživatelem (příkladem může být použití třeba čísla 3), jsou co do typu interpretovány podle kontextu použití.

Textové konstanty jsou vždy chápány jako typ „string“ a interpretovány ve formě ASCII kódů. Jsou uloženy v poli typu byte, které je ukončeno hodnotou 0 (jedná se skutečně o hodnotu, neplést s ASCII kódem číslovky „0“). Takto uložené texty zabírají v paměti vždy o jeden byte více, než je jejich počet znaků.

1.2 Systémové proměnné

Systémové proměnné jsou předdefinované proměnné, které slouží k ovládání, nastavování a práci se vstupně výstupními bloky, časovači, reálným časem, sítí automatů atd., tj. s technickými prostředky implementovanými v automatech. Proměnné mají vyhrazené názvy a k těmto názvům náleží typy těchto proměnných.

Vstupy a výstupy

Vstupy a výstupy zahrnují celou škálu typů vstupů a výstupů jimiž jsou automaty MICROPEL vybaveny. Jedná se o digitální vstupy napěťové s galvanickým oddělením nebo bez něho (unipolární nebo bipolární), analogové vstupy proudové, napěťové a pro měření teplotních čidel (Pt100, Pt1000, Ni1000), digitální výstupy napěťové s galvanickým oddělením nebo bez něj, analogové výstupy napěťové. Seznam proměnných pro manipulaci s fyzickými vstupy a výstupy automatu ukazuje Tab. 2.

Jméno	Popis
X[0] - X[31]	Pole bitů pro mapování fyzických digitálních vstupů
Y[0] - Y[31]	Pole bitů pro mapování fyzických digitálních vstupů
I[0] - I[31]	Pole wordů pro mapování fyzických digitálních vstupů
CALIB[0]-CALIB[23]	Pole wordů pro kalibraci analogových vstupů automatu
ADCMODE, ADCMODE2	Registry pro nastavení typu připojení pro měření Pt100
O[0] - O[31]	Pole wordů pro mapování fyzických analogových výstupů

Tab. 2 Seznam vstupů a výstupů a souvisejících proměnných

Datové proměnné

V definici systémového prostoru dat automatů MICROPEL nalezneme tři datová pole uzpůsobená pro ukládání různých typů proměnných. Tato datová pole jsou shrnuta v Tab. 3. Pole D a M mají polovinu sdílenou na síti automatů, pole NETLI, NETLW a NETF jsou sdílena celá. Pozor pole NETLI, NETLW a NETF v skutečnosti představují pouze různé typy pohledů na sdílenou oblast systémové paměti a představují tak vlastně jediné pole čtyřbajtových hodnot.

Komunikační proměnné jsou z hlediska použití speciálním typem proměnných. **Slouží k automatizovanému přenášení dat mezi různými automaty na síti** automatů. V Tab. 3 jsou komunikační proměnné specifikovány sloupcem sdílená část. Sdílená část obsahuje rozsah indexů datového pole daného jména, čímž specifikuje položky pole, které systém automatů sdílí v rámci sítě.

Jméno	Lokální část	Sdílená část		Typ	Položek
D	0 - 31	32 - 63		word	64
M	0 - 63	64 -127		bit	128
NETLW	není	0-255	**mapovány přes sebe**	longword	255
NETLI	není	0-255		longint	255
NETF	není	0-255		float	255

Tab. 3 Tabulka datových proměnných

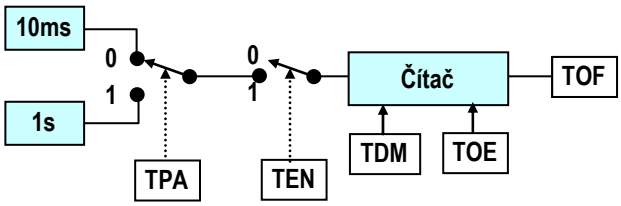
Zmíněné sdílení můžeme zjednodušeně chápat tak, že **v každém okamžiku obsahují** uvedená **datová pole** ve specifikovaném rozsahu **indexů stejné hodnoty ve všech automatech připojených do sítě**. Dále je nutné poznamenat, že pole **NETLW, NETLI a NETF** jsou ve skutečnosti polem jediným (říkáme, že **jsou mapovány přes sebe**). To je možné díky tomu, že datové typy položek těchto polí zabírají v paměti automatu shodně po 4byte na položku. Popisované proměnné pro nás tedy znamenají proměnné, které jsou společné pro celou síť automatů a označujeme je též termínem „**Síťové proměnné**“

Speciální funkční registry

Pojmem speciální funkční registry označujeme pole W a pole B. V prvním případě se jedná o pole se 128mi položkami typu word, v druhém případě pak o pole 128mi bitů. Tato pole jsou definována pro všechny typy automatů MICROPEL. Tak, jak se automaty MICROPEL rozvíjely a rozvíjejí, dochází k postupnému mapování ovládacích proměnných vnitřních hardwarových bloků právě do zmíněných polí. Příkladem může být registr T[0] systémového časovače číslo 0, který je mapován do pole W na index číslo 0.

Časovače

Časovače představují základní nástroj pro odměřování časových událostí, pro generování časových zpoždění, časování dějů na vstupech i výstupech automatu. Časovače jsou co do čítané hodnoty v systému reprezentovány datovým typem word, tj. umožňují rozlišit až 65535 časových jednotek. S využitím informace o přetečení čítané hodnoty je možné pomocí programových prostředků čítaný rozsah rozšířit. Každý automat má k dispozici osm těchto časovačů, přičemž každý z nich může pracovat buď **v rozlišení 10ms nebo 1s**. Časový interval 10ms nebo 1s tak tvoří časovou základnu časovače. Na Obr. 1 je uvedeno zjednodušené zapojení jednoho z osmi systémových časovačů. Každý časovač je vybaven bitovou proměnnou TPA pro volbu časové základny, bitem pro povolení čítání TEN, bitem pro nastavení módu čítání TDM, bitem pro povolení přetečení TOE a bitovým příznakem přetečení TOF.



Obr. 1 Schématické znázornění funkce časovače

Tabulka Tab. 4 shrnuje řídicí bity časovače a jejich vlastnosti.

Jméno	Popis
TPA	Volba časové základny 1 = 1s, 0 = 10ms
TEN	Povolení čítání impulsů časové základny
TDM	Volba směru čítání 0 = nahoru, 1 = dolů
TOE	Povolení přetečení časovače 0 = zakázáno, 1 = povoleno
TOF	Příznak přetečení časovače

Tab. 4 Popis řídicích bitů časovače

Vzhledem k tomu, že jsou řídicí bity časovačů seskupovány po osmicích, přistupujeme k jednotlivým bitům v zápisu programu pomocí indexu pole. Index, který současně znamená i číslo časovače, můžeme volit z rozsahu od 0 do 7.

Reálný čas

Reálný čas a kalendář je v řídicích automatech MICROPEL reprezentován pomocí sedmi proměnných typu word. Zápisem hodnoty do těchto proměnných nastavujeme aktuální reálný čas automatu. V Tab. 5 jsou shrnuty všechny registry pro práci s reálným časem automatu.

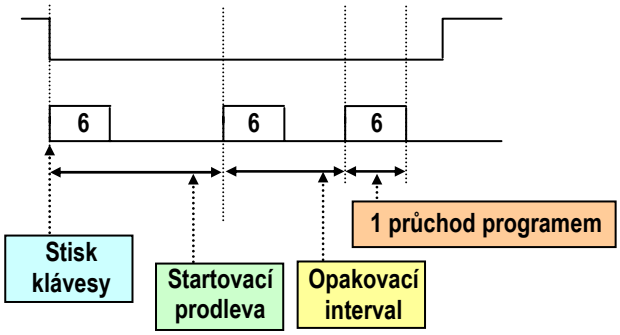
Jméno	Popis
SECOND	Proměnná obsahuje aktuální počet sekund v rozsahu 0-59
MINUTE	Proměnná obsahuje aktuální počet minut v rozsahu 0-59
HOUR	Proměnná obsahuje aktuální hodinu v rozsahu 0-23
DAY	Obsahuje den v měsíci, rozsah podle aktuálního měsíce od 1 - 31
MONTH	Obsahuje číslo měsíce v rozsahu 1 - 12
WEEK	Den v týdnu v rozsahu od 1 do 7, 1 = neděle
YEAR	Dvojčíslí roku ve století, rozsah 0 - 99

Tab. 5 Seznam proměnných reálného času

Klávesnice

Pro obsluhu klávesnice automatů MICROPEL je vyhrazeno několik speciálních registrů z nichž nejdůležitější **je registr KBCODE**, který obsahuje kód stisknuté klávesy. Při práci s tímto registrem je vhodné si uvědomit, že stisk klávesy je předán řídicímu programu systémem pouze jednou na začátku (při volání uživatelského programu) a v okamžiku, kdy uživatelský program

skončí jeden běh, tj. vykoná poslední příkaz hlavní smyčky, systém tento registr automaticky vynuluje a uvede ho tím do stavu, který signalizuje, že není stisknuta žádná klávesa. Krom tohoto základního registru, disponuje automat ještě registry pro funkci „autorepeat“ (generování opakovaného stisku při dlouhém držení tlačítka), kterou známe z prostředí osobních počítačů a též bitem pro ovládání akustické signalizace stisku. Časový diagram na Obr. 2 zobrazuje časování obsluhy klávesnice včetně funkce „autorepeat“.



Obr. 2 Zjednodušené časování obsluhy klávesnice

Předpokládejme stisk klávesy s kódem 6 (šipka dolů). Stisk se vyhodnotí, kód 6 se zapíše do proměnné KBCODE a spustí se časování startovací prodlevy. Mezi tím proběhne minimálně jeden průchod hlavní smyčkou programu. Pokud stisk klávesy trvá i potom, co je ukončeno časování startovací prodlevy (hodnota registru KBDELAY) vloží systém znovu kód 6 do proměnné KBCODE a spustí časování opakovacího intervalu. **Po dokončení průchodu programovou smyčkou dojde k vynulování KBCODE**. V okamžiku, kdy je ukončeno časování opakovacího intervalu, nastaví systém proměnnou KBCODE opět na kód 6. Pokud tedy uživatel klávesu drží, předává systém řídicímu programu aplikace kód stisknuté klávesy v rytmu opakovacího intervalu.

Jméno	Funkce	Typ	Poč. hodnota
KBCODE	kód stisknuté klávesy	word	0
KBDELAY	startovací prodleva pro "autorepeat"	word	100 [x10ms]
KBREPEAT	opakovací interval pro autorepeat	word	10 [x10ms]
KBREPEN	zapnutí funkce "autorepeat"	bit	0
KBSOUND	zapnutí akustické indikace	bit	1

Tab. 6 Proměnné pro obsluhu klávesnice

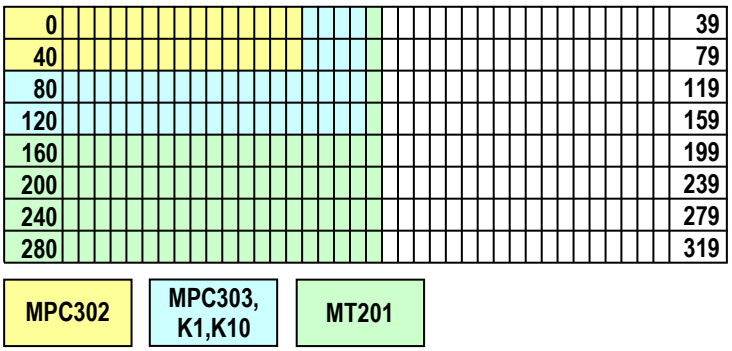
Tab. 6 shrnuje řídicí proměnné pro obsluhu klávesnice včetně hodnot, které proměnné obsahují po resetu automatu. Pro vyhodnocení stisku klávesy aplikací je nutné znát kódy jednotlivých kláves tak, jak se objevují v proměnné KBCODE. Tyto kódy jsou uvedeny v Tab. 7. Tabulka ve třech sloupcích uvádí vždy společné klávesové kódy pro jednotlivé typy a řady automatů. Pro jednoduchou přenositelnost zdrojového textu jsou základní kódy kláves (viz. první sloupec) pro všechny automaty stejné. U automatu MT201 jsou kódy kláves závislé na aktuální volbě aktivní klávesnice. Volbu klávesnice blíže vysvětluje uživatelská příručka pro MT201.

MPC300,K,MT201		K		MT201	
šipka vlevo	1	+/-	7	F1	8
šipka vpravo	2	F1	8	F2	9
šipka nahoru	5	F2	9	F3	10
šipka dolů	6	F3	10	menu vlevo	11
ESC	3	0	48	menu vpravo	12
ENT	4	1	49	neaktivní	13
		2	50		
		3	51		
		4	52		
		5	53		
		6	54		
		7	55		
		8	56		
		9	57		
		.	46		

Tab. 7 Tabulka kódů kláves pro automaty MICROPEL

Displej

Tisk hodnot na displej automatů MICROPEL **je realizován pomocí** sady tiskových **funkcí s názvem „Display“ a** dvojicí globálních **proměnných FORMAT a POSITION**.

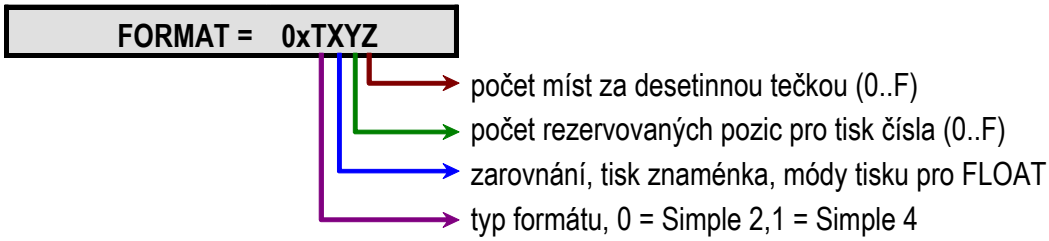


Obr. 3 Zobrazovací paměť při tisku na displej

Tisk na displej probíhá jako společný proces systému automatu a uživatelské aplikace. Oba tyto procesy využívají zobrazovací paměť dle Obr. 3. Paměť je tvořena osmicí řádků dlouhých 40 znaků. Jednotlivé pozice paměti jsou očíslovány způsobem jaký naznačuje obrázek, tj. zleva doprava a shora dolů. Systémový proces **zobrazování** paměti **pracuje na principu cyklického přenosu obsahu zobrazovací paměti na fyzický displej** automatu. Přenos obsahu paměti neprobíhá najednou ale znak po znaku v pevném časovém rastru. Překreslení celého displeje automatu trvá cca 200ms a displej se tak překresluje max. 5x za sekundu.

Z hlediska aplikace, je zobrazovací paměť přístupná pomocí sady zobrazovacích procedur „Display“. Tyto procedury zajišťují formátování tiskového výstupu podle hodnoty proměnné

FORMAT a výsledný formátovaný tisk zapisují do zobrazovací paměti na místo určené hodnotou proměnné POSITION. Hodnota této proměnné a pozice na displeji odpovídá přesně značení pozic na Obr. 3. Na tomto místě je vhodné upozornit na záludnost použití univerzálního formátu 0. S tímto formátem tisknou zobrazovací funkce „Display“ hodnotu proměnné na právě potřebný počet



Pole X	Zarovnání	Znaménko "+"	Tisk proměnných FLOAT	Tisk celočíselných proměnných
0	vlevo	ne	AUTO - tisk buď v klasickém tvaru (±XXX.XXXX), nebo v exponenciálním (±X.XXXE±XX). Pole Z udává počet platných míst	DECI standardní dekadický tvar
1	vpravo	ne		
2	vlevo	ano		
3	vpravo	ano		
4	vlevo	ne	FIX - tisk s pevným počtem desetin. Tisk čísla je ve tvaru ±XXX.XXXX. Pole Z udává počet desetinných míst	HEX tisk čísla v hexadecimálním tvaru (desetiny se ignorují)
5	vpravo	ne		
6	vlevo	ano		
7	vpravo	ano		
8	vlevo	ne	EXP - exponenciální vyjádření. Tisk čísla ve tvaru ±X.XXXE±XX, pole Z = počet desetinných míst, exponent je dvoumístný	DECI
9	vpravo	ne		0-DECI, **předsaz. nuly**
A	vlevo	ano		DECI
B	vpravo	ano		DECI
C	vlevo	ne		HEX
D	vpravo	ne		0-HEX, **předsaz. nuly**
E	vlevo	ano		HEX
F	vpravo	ano		HEX

Tab. 8 Formátování tisku číselných hodnot

znaků. To znamená, že pokud bez jakéhokoli opatření tiskneme proměnnou s hodnotou 100 a tato hodnota se v čase změní na 99, bude na displeji zobrazeno 990 a ne 99, jak by se na první pohled zdálo. Přebytečná 0 zůstala zobrazena z předešlého tisku hodnoty 100. Pro tisk proměnných tedy doporučujeme důsledně používat formátování tisku pomocí zarovnání a vyhrazení tiskových pozic nastavením vhodné hodnoty proměnné FORMAT. **Formátování výpisu** proměnných na displej **se řídí hodnotou proměnné FORMAT**. Tato proměnná je typu word a jejích 16 bitů můžeme rozdělit po čtveřicích bitů na čtyři pole. To je důležité proto, že pomocí hodnot těchto polí řídíme vybrané parametry tisku. Formátování podle Simple 4 shrnuje Tab. 8.

Jednotlivá pole tak můžeme pomocí hexadecimálního zápisu zapisovat s použitím hodnot 0-15 a A-F. Pole označované písmenem T (typ formátu) rozlišuje nové formáty jazyka Simple 4 oproti staršímu typu z jazyka Simple 2. Ze starší verze jazyka má smysl používat pouze formáty s hodnotami:

- 120 (0x0078) - tisk celočíselné hodnoty jako ASCII kódu, formát pro tisk jednotlivých písmen řetězce
- 121 (0x0079) - **formát sloužící pro definici vlastních znaků** (automaty řady MPC a K umožňují definovat 8 vlastních znaků, MT201 pak všech 256 znaků).

Pro automaty MT201 vybavené novým grafickým displejem byly dále stanoveny nové speciální formáty tisku některých pomocných grafických prvků. Tyto speciální formáty pro MT201 jsou uvedeny v Tab. 9.

Grafická podoba	Formát	Pro délku 1-31 znaků
pravítko s jezdcem, dělení 1/2 znaku	0x100 + délka pásku	0x101-0x11F
obdélník s výplní, dělení 1/2 znaku	0x120 + délka pásku	0x121-0x13F
pravítko s jezdcem, dělení 1 znak	0x140 + délka pásku	0x141-0x15F
obdélník s výplní, dělení 1 znak	0x160 + délka pásku	0x161-0x17F

Tab. 9 Speciální grafické formáty

S vývojem automatu MT201 byl do registrů pro ovládání tisku na displej **přidán registr FONTCTRL, kterým se nastavují atributy písma**. Přehledně je dokumentuje Tab. 10, kde

- U podtržené znaky (0 = zapnuto, 1 = vypnuto)
- I inverzní znaky (0 = zapnuto, 1 = vypnuto)
- MAG velikost znaku (1,2,4).

FONTCTRL po jednotlivých bitech																
bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
funkce:											U	I		MAG		

Tab. 10 Funkční registr FONTCTRL

Systémové proměnné

Posledním neméně důležitým typem proměnných je skupina označovaná jako systémové. Do této skupiny řadíme proměnné:

- POINTER - proměnná typu word reprezentující ukazatel pro zápis nebo čtení z paměťové oblasti zásobníku automatu
- SPEED - proměnná ukazuje počet průběhů programu aplikace hlavní programovou smyčkou za jednu sekundu
- RESET - příznakový bit systémového resetu, **v použití tohoto bitu se často chybuje a proto doporučujeme pečlivě prostudovat kapitolu „Použití bitu RESET“**
- Y29 - zapnutí uživatelské akustické indikace
- Y30 - výstup pro zapnutí podsvícení displeje
- Y31 - výstup pro spínání vyššího kontrastu displeje (MPC300, K)
- O30 - intenzita podsvícení v rozsahu 0-31 (MT201)
- O31 - nastavení kontrastu displeje hodnoty 0-15 (MT201)

2 Syntaxe jazyka Simple 4

V následujících odstavcích je formou příkladů ukázána syntaxe a použití jazyka Simple 4 pro programování řídicích automatů MICROPEL.

Ovládání vstupů a výstupů

Ovládání vstupů a výstupů automatu se z hlediska programátorského jeví jako **čtení nebo zápis hodnoty**. Zde musíme rozlišovat datový typ výstupu nebo vstupu a používat odpovídající proměnné v přiřazovacím příkazu.

- Digitální vstupy a výstupy

  Y[0] = 1 ; nastavení výstupu Y[0] na logickou hodnotu 1
  Y[0] = 0 ; nastavení výstupu Y[0] na logickou hodnotu 0
  Y[0] = Y[0]' ; negace hodnoty na výstupu Y[0]

- Digitální vstupy

  Y[0] = X[0] ; nastavení hodnoty výstupu podle hodnoty vstupu X[0]
  Y[0] = X[0]‘ ; nastavení hodnoty výstupu negovanou hodnotou vstupu X[0]

- Analogové výstupy

  O[0] = 123 ; nastavení výstupu O[0] na hodnotu 123
  O[0] = O[1] ; nastavení hodnoty výstupu O[0] na hodnotu výstupu O[1]

- Analogové vstupy

  O[0] = I[0] ; nastavení hodnoty výstupu O[0] na hodnotu vstupu I[0]

Shrnutí:

V uvedených příkladech je ukázáno principielní použití přiřazovacího příkazu pro nastavení hodnoty na výstup automatu a pro předání hodnoty ze vstupu automatu na jeho výstup. Přiřazovací příkaz má formálně tvar

Cíl = Zdroj,

kde **„cíl“** je proměnná (výstup) kam se kopíruje hodnota proměnné (vstupu), kterou představuje **„zdroj“**. V zápisu zdrojového textu je ještě použit **znak „;“**, který uvozuje **řádkový komentář**. Řádkový komentář je libovolný text umístěný za znakem **„;“** do konce řádku. Tento text se nevyhodnocuje a slouží pro lepší orientaci v zápisu programu. V případě bitového přiřazovacího výrazu, můžeme v textu nalézt i postup jak přiřadit hodnotu negace. Použijeme k tomu znak **„'“**, který píšeme za proměnnou (vstup nebo výstup).

Uživatelská jména vstupů a výstupů

Chceme-li zachovat srozumitelnost zdrojového textu déle něž několik dní po dokončení aplikace a nebo jsme postaveni před problém napsat program u něhož není specifikováno přiřazení vstupů a výstupů, použijeme **uživatelské pojmenování vstupů a výstupů pomocí maker**. Zápis makra odpovídá formálně zápisu:

známý_symbol **#** nový_symbol

Pro potřeby programu podle příkladu „Bitové operace“ (Obr. 12) označíme vstupy pro přepínače a výstup pro světlo uživatelským symbolem a program pak zapíšeme následujícím způsobem:

```
X[0] # spinac_1        ; symbol pro označení vstupu spínače
X[1] # spinac_2        ; symbol pro označení vstupu spínače
Y[0] # svetlo          ; symbol pro označení výstupu
svetlo = (spinac_1 ^ spinac_2)‘  ; zápis programu pomocí symbolů
```

Nyní si můžeme položit otázku na výhodu popisovaného zápisu. Odpověď je jednoduchá a hlavně praktická. Pokud si představíme situaci, že máme program zapsán bez symbolického označení a jsme postaveni před úlohu např. přesunout spinac_1 ze vstupu X[0] na vstup X[8], musíme ve všech místech, kde máme použitý vstup X[0] změnit zápis na X[8]. To může být dosti pracná záležitost zvláště v případě rozsáhlých zdrojových textů programu.

Pokud budeme mít program zapsán pomocí uživatelských symbolů, postačí, když změníme přiřazení vstupů pouze v místě zápisu makra. Celý zbytek programu zůstane nezměněn.

Výborným trikem v používání maker, je využití znaménka pro bitovou operaci negace. Elegantně se tak řeší případ záměny čidel NO („normally open“) za čidla NC („normally closed“), tj. záměna spínacího kontaktu za vypínací a naopak. Použití maker je v tomto případě extrémně výhodné. Máme-li například havarijní termostat s výstupem typu NC můžeme zapsat připojení vstupu pomocí makra takto:

```
X[0] # hav_termostat   ; havarijní termostat s výstupem NC
```

Po záměně termostatu za model NO upravíme zápis na tvar:

```
X[0]‘ # hav_termostat  ; havarijní termostat s výstupem NO
```

POZOR!

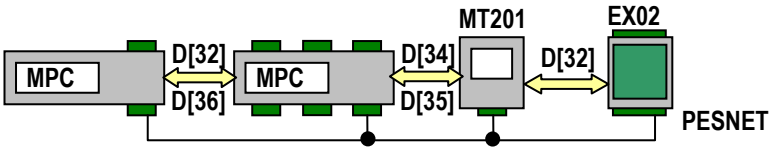
Popsaný způsob zavedení negace je možné použít pouze u vstupních signálů, protože ty se vyskytují na pravé straně přiřazovacího příkazu. U výstupů, které se vyskytují obvykle vlevo, tento postup uplatnit nelze, protože na levé straně přiřazovacího výrazu je vždy uvedeno cílové paměťové místo operace a ne aritmetická nebo bitová operace.

Shrnutí:

Použití zápisu pomocí maker je výhodné pro případy, kdy chceme uživatelským způsobem pojmenovat předdefinované názvy vstupů a výstupů popř. sdílených proměnných nebo konstant. S pomocí těchto jmen můžeme udělat program přehlednější a můžeme ho též snadno uzpůsobit v případě změny zapojení vstupů, výstupů popř. změny typu vstupních čidel.

Sdílené proměnné

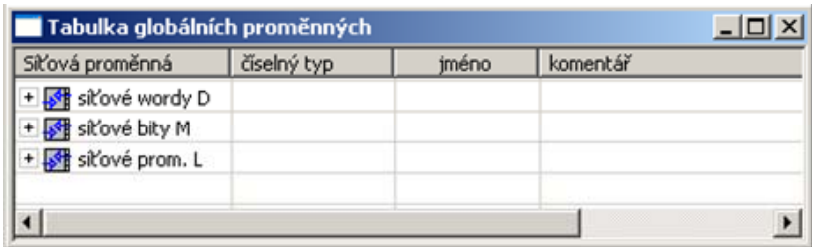
Sdílené proměnné zmíněné v odstavci „Datové proměnné“ kapitoly „1.2 - Systémové proměnné“, představují společnou paměťovou oblast pro všechny automaty a zařízení připojené do společné sítě automatů.



Obr. 4 Znázornění sdílení proměnných v rámci sítě automatů

Jak znázorňuje Obr. 4, jsou na síti zapojeny dva automaty řady MPC300, jeden automat MT201 a jedna periferie EX02. Dále obrázek naznačuje sdílení dat pomocí proměnných D. Tak např. proměnnou D[32] sdílí oba automaty MPC a periferie EX02. Proměnnou D[34] a D[35] automaty MPC300 a MT201 a konečně proměnnou D[36] automaty MPC300. Nyní je důležité si uvědomit, že pro každý automat je k dispozici zdrojový text programu a pro periferii pak konfigurační soubor. Tyto texty kromě interních regulačních funkcí daných programem ještě zapisují a nebo čtou data ze sdílených proměnných. Je zřejmé, že v popisovaném případě bude vhodné **pojmenovat** si **sdílené proměnné jednotně pro všechny prvky sítě**.

Pokud bude pojmenování proměnných společné pro všechny zdrojové texty, bude orientace v těchto programech mnohem snazší než dohledávání hodnoty podle zápisu D[32]. Pojmenování sdílených proměnných umístíme do souboru se zdrojovým textem, který bude společný pro všechny použité automaty a periferie. Abychom nemuseli zmiňovaný společný soubor složitě vytvářet, ponecháme jeho vytvoření nástroji StudioWin a dodáme pouze obsah, který vepíšeme do **tabulky globálních proměnných** (Obr. 5).



Obr. 5 Tabulka globálních proměnných

Shrnutí:

Pro pojmenování globálních proměnných preferujeme použití tabulky globálních proměnných v prostředí StudioWin. Pokud budeme chtít používat společné názvy např. vstupů a výstupů pro větší počet souborů nebo knihoven v rámci jednoho automatu, musíme si pomoci zápisem maker.

Pravidla pro uživatelská jména

Jedním z nejčastějších úkonů, který provádíme při zápisu zdrojového textu je deklarace proměnných, konstant, deklarace funkcí, podprogramů atd. S tím souvisí volba uživatelských jmen pro tyto prvky programu. **Uživatelské jméno je** pro překladač jazyka Simple 4 **symbolem**, jehož zápis **podléhá** následujícím **omezující pravidlům**

- **symbol** musí být umístěn **celý na jednom řádku**, rozdělení symbolu na více řádku není dovoleno
- symbol smí být **maximálně 512 znaků dlouhý**
- symbol může obsahovat malá a velká písmena **bez diakritiky**, číslice a znak podtržítko
- symbol **musí začínat písmenem nebo podtržítkem**
- symbol **nesmí být klíčovým slovem**
- malá a velká písmena se nerozlišují
- symbol musí být až na výjimky jedinečný
- duplikace symbolu je povolena ve jménech podprogramů a funkcí a v případě jmen lokálních proměnných podprogramů a funkcí vůči globálním symbolům.
- podobná pravidla platí i pro jména použitá v makrech. Rozdíl je v tom, že symboly použité v makrech jsou platné v rámci celého zdrojového kódu bez rozlišení na lokální a globální. Proto musí být tyto symboly jedinečné pro celý zdrojový text.

Doporučení:

Volbou jména se snažíme o zpřehlednění zápisu zdrojového textu. Příkladem nevhodného jména může být pojmenování vstupu X[0] například podle čísla kontaktu

X[0] # k123

V tomto případě vidíme, že jeho srozumitelnost není příliš vysoká. Lepší je zvolit jméno např.

X[0] # koncovy_spinac

V této volbě je **znak podtržítka použít ve funkci mezery**, což umožňuje relativně srozumitelně zapsat víceslovná jména. Mezeru jako takovou použít nemůžeme proto, že mezera je znakem, který je chápán jako oddělovač symbolů. Dá se říci, že z hlediska programátora není symbol zvolen optimálně, protože obsahuje celkem hodně znaků, které musí programátor při každém použití symbolu opisovat.

Shrnutí:

Uživatelská jména volíme tak, aby nám orientaci ve zdrojovém textu ulehčovala a ne naopak. Snažíme se volit jména podle technologických nebo projektových názvů, abychom mohli snadno identifikovat a přiřadit technologický prvek k programové konstrukci, která ho řídí.

Deklarace a použití konstant a proměnných

Kromě předdefinovaných proměnných pro vstupy, výstupy, reálný čas, časovače atd. zmíněných v odstavci 1.2 můžeme využívat pro tvorbu programu aplikace uživatelských proměnných a konstant. Abychom mohli proměnnou nebo konstantu použít, musí být tzv. deklarována. **Překladač** jazyka Simple 4 **nevyžaduje dopřednou deklaraci** proměnných nebo konstant, což znamená, že proměnná nebo konstanta může být použita ve zdrojovém text dříve než je deklarována. Samotná deklarace je však vyžadována.

- **Deklarace konstant**

Deklaraci konstanty uvozuje klíčové slovo „const“ za nímž následuje inicializace konstanty číselnou hodnotou pomocí přiřazovacího příkazu. Deklaraci konstanty zapisujeme:

```
const pomocna_hodnota = 12
```

Pro deklaraci konstant můžeme použít též systém výčtu představující zápis, při němž doplní hodnoty konstant překladač. Zápis má tvar:

```
const zakladni,zvyseny,havarie        ; výčet konstant
```

V tomto případě překladač nastaví hodnotu prvního symbolu na 0 a pro každý další symbol použije hodnotu o jedničku vyšší než má symbol předchozí. V uvedeném případě tedy platí

```
základni = 0, zvyseny = 1, havarie = 2
```

Výčet konstant má nejlepší využití v kódování stavů pro programování stavového diagramu, dále pak pro kódování chybových hlášení.

- **Deklarace proměnných**

Deklarace jednoduchých uživatelských proměnných a konstant odpovídá formálně zápisu:

```
var typ_promenne jmeno_promenne
code typ_promenne jmeno_promenne = hodnota_promenne
```

První zápis deklaruje **proměnnou s umístěním v uživatelské datové paměti**, tj. paměti kterou je možné měnit. **Proměnné** deklarované tímto způsobem **mohou být použity i na levé straně přiřazovacího příkazu**.

Druhý zápis deklaruje **proměnnou**, která bude uložena **v kódové paměti automatu** (FLASH EEPROM). Data v kódové paměti není možné za běhu programu měnit a tudíž proměnná funguje jako konstanta a **nemůže být použita na levé straně přiřazovacího příkazu**. Příkladem deklarací proměnných může být zápis:

```
var word teplota_zadana
code int parametr_regulatoru
```

Shrnutí:

Pro deklaraci proměnných v datové i kódové paměti používáme kromě typu proměnné i uživatelské symboly jejichž tvar musíme přizpůsobit pravidlům v odstavci „Pravidla pro uživatelská jména“. Dále musíme uvést typ proměnné u něhož je vyžadována dopředná deklarace, tj. typ proměnné musí být znám před jeho použitím. Tento požadavek je u jednoduchých předdefinovaných datových typů zaručen vždy, pro složené nebo uživatelské typy ho musí zajistit programátor.

Deklarace a použití polí

Pod pojmem **pole** máme na mysli **proměnnou, která obsahuje známý počet položek stejného datového typu**. K jednotlivým položkám pole pak přistupujeme na základě indexu položky. Ten je počítán od 0 a je platný do hodnoty n-1, kde n je počet položek pole. Abychom mohli pole použít musíme ho deklarovat**. V okamžiku deklarace musí být znám počet položek** pole a ten musí být konstantní. Deklarace pole odpovídá formálně zápisu:

```
var typ_položky[počet_položek] jméno_proměnné
```

nebo

```
code typ_položky[počet_položek] jméno_proměnné = (hodnota,hodnota....)
```

nebo

```
table typ_položky[počet_položek] jméno_proměnné = (hodnota,hodnota....)
```

Klíčové slovo „table“ je synonymem ke slovu „code“. Typické použití polí je v případech, kdy potřebujeme uchovat seznam nebo řadu hodnot a tyto hodnoty nemají buď žádnou nebo naopak složitou funkční závislost, kterou bychom aritmeticky obtížně vyjadřovali. **Výhodnost polí** hodnot **se** většinou **plně projeví v přístupu** k jednotlivým položkám pole **pomocí proměnného indexu**. Příkladem může být zápis:

```
code word[8] tabulka = (0,100,141,173,200,224,245,265)
var word n
var word sqrtn
sqrtn = tabulka[n]        ;zjištění druhé odmocniny v setinách
```

Uvedený příklad ukazuje použití tabulky pro hrubý výpočet odmocniny. Funkce odmocniny není automaty MICROPEL podporována a tak, pokud ji potřebujeme, můžeme ji řešit pomocí tabulky. Uvedený příklad obsahuje tabulku odmocniny pro čísla od 0 do 7. Hodnotu odmocniny zjistíme tak, že číslo, pro nějž chceme odmocninu znát, použijeme jako index do tabulky. Hodnoty odmocnin jsou uvedeny v setinách proto, že proměnná word použitá jako položka tabulky neumožňuje zobrazovat necelá čísla.

Programovací **jazyk** Simple 4 **podporuje** krom jednorozměrného pole **i pole vícerozměrná**. Počet rozměrů není jazykově omezen, nicméně z praktického hlediska mají význam pole do rozměru 3. Deklarace vícerozměrných polí odpovídá formálnímu tvaru:

```
var typ_položky[]....[počet_sloupců][počet_řádků] jméno_proměnné
code typ_položky[]....[počet_položek] jméno_proměnné = ((hodnota,hodnota),....)
table typ_položky[]..[počet_položek] jméno_proměnné = ((hodnota,hodnota),....)
```

Uvedeme příklad tabulky se dvacítkou sloupců každý po patnácti řádcích:

```
var byte[20][15] tabulka
```

Shrnutí:

Použití polí proměnných je vhodné pro práci s tabelovanými nebo výčtovými hodnotami. Programovací jazyk podporuje jedno a vícerozměrná pole, kdy počet rozměrů není omezen. Přístup k jednotlivým položkám pole je možný pomocí proměnné v indexu s jedinou výjimkou a tou je pole bitů. ***Pole bitů není možné indexovat proměnným indexem ale pouze konstantním indexem nebo výrazem***.

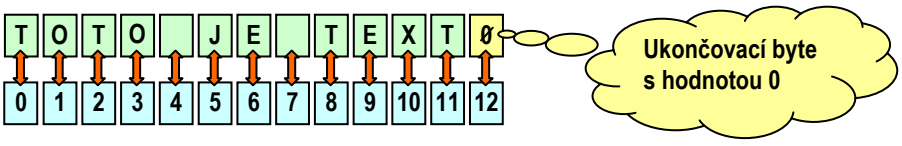
Řetězce znaků jako zvláštní typ polí

Automaty MICROPEL jsou ve své většině typů vybaveny LCD displejem, který je programově dostupný z aplikace a je určen pro zobrazování uživatelských dat. V řadě případů je vhodné na displeji zobrazovat stav nějakého zařízení. Tento stav obvykle zjišťujeme z několika informací. Plynové kotle například poskytují výstup porucha a mají signalizaci požadavku „chod kotle“. Abychom byli schopni zobrazit stav kotle musíme mnohdy napsat poměrně složitý kód, neboť někteří výrobci poskytují informaci o poruše kotle pouze tehdy, pokud je požadavek na chod kotle. V těchto případech se pomocí podmíněných příkazů snažíme stav zařízení zakódovat. Výhodné je použít číselné kódy stavů počínaje hodnotou 0 a konče kladným celým číslem označující koncový stav. Pro příklad zmíněného kotle bychom použili kódování stop = 0, běh = 1, porucha = 2. Pokud použijeme tyto kódy můžeme je zobrazit na displeji automatu buď ve formě čísla a nebo daleko lépe pomocí textu. A právě v případě textu je vhodné řešit zobrazení pomocí polí. Řešení je jednoduché a může vypadat takto:

```
var byte stav_kotle
table string[3] text = („stop“, „beh“, „porucha“)  ;vyjádření stavu
Display(text[stav_kotle])
```

V ukázce zdrojového textu je vidět zadání textů do trojice položek pole řetězců. V dalším řádku pak výpis textu podle hodnoty proměnné stav_kotle. Tímto postupem jsme převedli číselnou hodnotu proměnné „stav_kotle“ na text.

Řetězce jako takové **musí být použity společně s klíčovým slovem „table“ nebo „code“** a to proto, že jazyk Simple 4 podporuje řetězce znaků pouze v kódové paměti, tj. na pravé straně výrazu. Každý jednotlivý řetězec znaků představuje ve své podstatě pole bajtů ovšem s tím, že počet těchto položek může být v tomto případě různý tzv. „kus od kusu“. Tato speciální vlastnost pole je implementována pouze u textových řetězců. V případě textového řetězce se délka pole určuje testem jednotlivých bajtů řetězce. **Konec řetězce je označen bytem s hodnotou 0.** Tato znalost je důležitá, pokud chceme na displeji automatu realizovat některé efektové tisky textů, jako je například rotující text nebo text zobrazovaný náhodně po jednotlivých písmenech. Formát uložení jednotlivého textu v kódové paměti automatu názorně ukazuje Obr. 6. V modře podbarvených polích je hodnota indexu, zelená pole obsahují ASCII kódy znaků a ve žlutém poli je poslední „ukončovací“ byte řetězce s hodnotou 0.



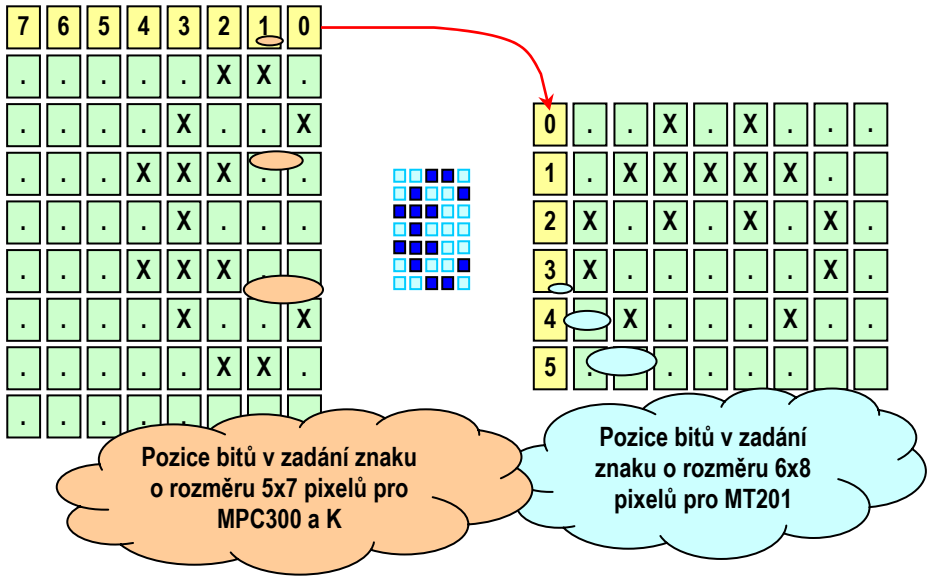
Obr. 6 Formát uložení řetězce znaků

Shrnutí:

Řetězec textu je zvláštním případem pole byte. Tento typ datové proměnné označujeme klíčovým slovem „string“ a pro tento typ představující pole byte není stanovena konkrétní délka pole. Ta se stanovuje pro každý řetězec v průběhu překladu. Aby bylo možné zjistit tuto délku programově a též aby byl umožněn přístup k jednotlivým znakům řetězce, je za text řetězce připojen ukončovací byte s hodnotou 0.

Uživatelské znaky

Programovatelné automaty řady MPC300, K a terminál MT201 umožňují definovat grafickou podobu uživatelských znaků. Zatímco u typové řady MPC300 a K je k dispozici těchto znaků pouze 8, terminál MT201 umožňuje předefinovat všech 256 znaků a není tudíž žádný problém přizpůsobit znakovou sadu téměř libovolné národní abecedě. Zadání znaku provádíme pomocí speciálního formátu 121 pro definici znaků. V tomto formátu označuje hodnota proměnné POSITION pro MPC300 nebo K resp. MT201 grafický řádek resp. sloupec znaku. Na Obr. 7 jsou



Obr. 7 Definice nového znaku "euro"

uvedeny dvě bitové mapy, které představují uživatelský znak „euro“. Bitové mapy jsou kresleny ve formátu který odpovídá zadání uživatelského znaku tzv. bajt po bajtu. Před zápisem bitové mapy pomocí formátu 121 je nutné správně nastavit hodnotu proměnné POSITION. Pro automaty řady MPC300 a K platí nastavení proměnné podle rovnice:

POSITION = 8 * kod_znaku

Pro terminál MT201 se nastavení liší a použijeme pro něj rovnici

POSITION = 6 * kod_znaku

V obou rovnicích představuje „kod_znaku“ ASCII kód definovaného znaku. Pro řadu MPC300 a K můžeme volit hodnotu „kod_znaku“ z rozsahu 0-7. Pro terminál MT201 pak z rozsahu 0-255. Vhodné kódy pak představují rozsahy od 0 do 31 a 128 do 255. V těchto rozsazích se vyskytují buď netisknutelné znaky, volné kódy popř. speciální znaky národní abecedy. Pro výpočet kódu platí následující rovnice, kde koeficienty b0 - b7 představují hodnotu 0 nebo 1 podle toho zda má být bod znaku rozsvícen nebo zhasnut. Pro zápis hodnoty je výhodné použití hexadecimální soustavy.

kód = $b_7 \cdot 2^7 + b_6 \cdot 2^6 + b_5 \cdot 2^5 + b_4 \cdot 2^4 + b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0$

Pro řadu MPC300 bude zdrojový kód definice znaku „euro“ odpovídat zápisu:

```
FORMAT = 121        ; formát pro definici uživatelského znaku
POSITION = 16       ; umístění znaku na ASCII kód 2
DISPLAY(0x06)       ; zápis řádku bitové mapy znaku
DISPLAY(0x06)
```

..... atd až .....
DISPLAY(0x00)

K uvedenému zdrojovému textu je vhodné poznamenat, že při každém volání podprogramu „Display“ se automaticky posune hodnota proměnné o 1, tj. stejně jako jsme zvyklí i při ostatních způsobech tisku. Je patrné, že uvedený zápis v zásadě skrývá grafický tvar uživatelského znaku, nicméně podoba znaku není příliš názorná. Nevýhodu můžeme obejít alternativním typem definice znaku, který představuje bitový zápis.

Bitový zápis řetězce využívá operátoru „+“ ke spojení jednotlivých znaků řetězce a zápis pomocí znaku písmena „X“ a „.“ pro simulaci skutečné podoby znaků pseudo bitovou mapou. Všimněme si, že takto zapsaný řetězec má v neproporcionálním písmu podobu zjednodušené bitmapy a znak lze tudíž mnohem snáze editovat než např. jeho číselný zápis.

```
code string euro = ( \  .  .  .  .  .  X  X  .  +
                     \  .  .  .  .  X  .  .  X  +
                     \  .  .  .  X  X  X  .  .  +
                     \  .  .  .  .  X  .  .  .  +
                     \  .  .  .  X  X  X  .  .  +
                     \  .  .  .  .  X  .  .  X  +
                     \  .  .  .  .  .  X  X  .      ) ; definice znaku euro
```

Obdobně je snadná editace i v případě znaků pro MT201, které jsou oproti svému zobrazení na displeji otočeny o 90 stupňů.

```
code string euro = ( \  .  .  X  .  X  .  .  .  +
                     \  .  X  X  X  X  X  .  .  +
                     \  X  .  X  .  X  .  X  .  +
                     \  X  .  .  .  .  .  X  .  +
                     \  .  X  .  .  .  X  .  .  +
                     \  .  .  .  .  .  .  .  .      ) ; definice znaku euro
```

Protože operační systém automatů nepodporuje zápis řetězce znaků při formátu 121, musí být řetězec při definici znaků předán „bajt po bajtu“. Zdrojový text pro MT201 má tvar:

```
POSITION = 12
FORMAT = 121
DISPLAY(euro[0])
DISPLAY(euro[1])
DISPLAY(euro[2])
DISPLAY(euro[3])
DISPLAY(euro[4])
DISPLAY(euro[5])
```

Shrnutí:

Pro definici uživatelských znaků automatů řady MPC300, K a terminálu MT201 používáme speciální formát 121 a zápis jednotlivých byte znaku, které představují jednotlivé řádky resp. sloupce definovaného znaku. Pro definici znaku je možné s výhodou použít též pseudografický bitový zápis.

Formátování řetězců

Se základním typem formátování do uvozovek vystačíme v naprosté řadě případů. Okolnosti se ale mohou změnit, pokud chceme například využít uživatelské znaky, které je možné definovat pro automaty řady MPC300, K a MT201 a nebo chceme ze znakové sady displeje vytisknout nějaký znak, který není k dispozici na klávesnici. Problém tedy je, jak znak zapsat, aby ho bylo možno vytisknout společně s ostatním textem. K formátování použijeme tyto postupy:

- Tisk uvozovek v textovém řetězci

  code string text = "\"teplota\" je v uvoz."

Při tisku tohoto řetězce na displej se objeví text: **"teplota" je v uvoz**. Všimněme si, že výstupní text na displeji automatu obsahuje uvozovky. Ty bychom při standardní syntaxi pro zápis řetězce neuměli jako součást řetězce vytisknout.

- Pro definice uživatelských znaků můžeme využít **bitový zápis řetězce** v němž používáme zástupné znaky „X" a „." pro bit v hodnotě 1 a 0 společně se zpětným lomítkem pro označení specifického formátování znaku řetězce a s využitím operátoru „+" pro spojení jednotlivých bajtů (znaků) do řetězce. Postup zevrubně naznačuje odstavec „**Uživatelské znaky**" .
- **Znak** zadán **pomocí číselného kódu**, kdy nahradíme znak v řetězci kódem znaku. Aby toto formátování překladač rozlišil je k dispozici řídící znak zpětného lomítka „\". Pokud uvedeme znak zpětného lomítka očekává za ním překladač číselný kód v dekadickém nebo hexadecimálním tvaru. Vyhodnocení kódu je ukončeno buď s prvním znakem, který do dané soustavy nepatří nebo hodnotou, která nesmí překročit 255 nebo maximálním počtem vyhrazených pozic pro dekadickou (3 číselné pozice) nebo hexadecimální (2 číselné pozice) soustavu. Uveďme několik příkladů tisku např. číslice 0. Tato číslice má kód 0x30 nebo 48.

  "\0x300" → „00"
  "\0480" → „00"
  "\480" → „00"

V uvedených příkladech zápisu vidíme tisk číslice 0, kdy kód tisknutého znaku zadáváme pomocí čísla. Díky kódu 48 bude i poslední zápis vytisknut jako dvojice nul. Pokud bychom chtěli vytisknout před číslicí 0 znak „euro" pod kódem 2 (viz. odstavec „**Uživatelské znaky**") pak by zápis

"\20" → „ "

vytisknul na displeji většiny automatů prázdný znak, neboť by tiskl znak s kódem 20 a ne znak s kódem 2 a číslovku 0. Je to proto, že hodnota 20 nepřesáhne 255 a překladač ji tedy považuje za kód znaku. Pro tisk znaku s kódem 2 následovaný číslicí 0 musíme použít zápis:

"\0020" → „€0"

nebo

"\0x020" → „€0"

Shrnutí:

Pro formátování jednotlivých znaků řetězce můžeme využít **text v uvozovkách, číselné kódy znaku a to uvedené hexadecimálně i dekadicky**, nebo zápis **pomocí speciálních bitových znaků** „." a „X". Jednotlivé části řetězce můžeme spojovat operátorem „+". Číselným kódem v řetězci nelze tisknout znak s kódem 0, protože by byl interpretován jako ukončovací byte řetězce.

Uživatelské datové typy

Uživatelské datové typy představují programovou konstrukci, která umožňuje **definování nových datových typů na základě již známých a definovaných typů**. Z toho plyne, že v prvním kroku můžeme definovat nové typy na základě základních datových typů a ve všech dalších krocích můžeme použít k definici nových typů všechny typy předešlé, tj. i typy, které jsme definovali jako uživatelské. Definice nového uživatelského typu tedy odpovídá formálnímu zápisu:

type známý_typ jméno_nového_typu

Jako příklad uveďme definici nového uživatelského typu „array“, který představuje pole s obsahem 32 položek typu word. Zápis uživatelského typu vypadá takto:

```
type word[32] array ;definice uživatelského typu array, který představuje datové pole
var array pole  ;deklarace uživatelské proměnné typu array v programu
pole[0] = pole[1]          ; použití uživatelského typu
```

Z uvedeného zápisu je patrné, že **po té, co definujeme nový datový typ, můžeme deklarovat proměnnou**, která bude typu „array“. S touto proměnnou pak můžeme zacházet obdobným způsobem jako s proměnnou základního typu avšak s tím rozdílem, že musíme respektovat její strukturu např. v přiřazovacím příkazu nebo v aritmetických či logických výrazech.

Popsaný nový uživatelský typ proměnné můžeme, obdobně jako typ word v odstavci „Deklarace a použití polí“, organizovat do pole. Pravý přínos uživatelský typů ale najdeme při jejich použití ve strukturách (viz. odstavec „Struktury dat“) nebo při volání procedur či funkcí s parametry předávanými odkazem (viz. odstavec „Předávání parametrů“).

Svázání uživatelského datového typu do datového pole zapíšeme:

```
type word[32] array ;definice uživatelského typu array, který představuje datové pole
var array[16] pole ; deklarace proměnné array do pole se 16ti položkami
```

V uvedeném příkladu vidíme, že jsme deklarovali pole 16ti položek. Každá položka pole je typu „array“ a tento typ je sám polem 32 položek typu word. Ve svém důsledku jsme tedy definovali dvourozměrné pole s položkami typu word o rozměrech 32 řádků a 16 sloupců. Tento poznatek musíme následně využít při přístupu k jednotlivým položkám. Za příklad může sloužit zápis:

```
var array[16] pole ; deklarace proměnné array do pole se 16ti položkami
var word pomocna ; pomocná proměnná pro výměnu položek pole
pomocna = pole[3][5] ; zdrojový text, který vymění položku 5 sloupce 3
pole[3][5] = pole[8][1] ; za položku 1 sloupce 8
pole[8][1] = pomocna
```

Shrnutí:

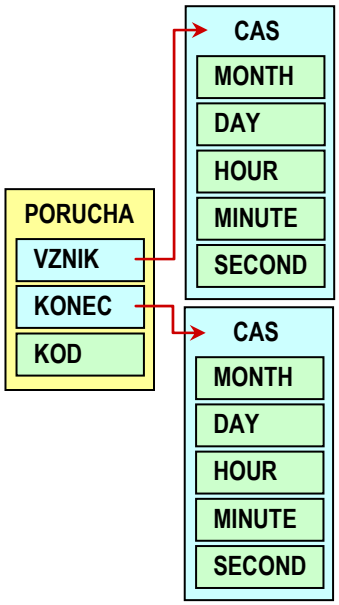
Uživatelské datové typy představují možnost pro vyšší úroveň zapouzdření dat při strukturovaném programování. Výhoda uživatelských typů se projeví při použití datových struktur a naplno pak při zpracování dat pomocí podprogramů a funkcí.

Struktury dat

Definování datových struktur je velmi silným nástrojem, který umožňuje strukturovat zdrojový text podle logiky ovládání řízené technologie. **Datovou strukturu můžeme chápat jako datový záznam**. Definice datové struktury odpovídá formálně zápisu

```
type struct
 typ_položky jméno_položky, ....., typ_položky jméno_položky
end jméno_typu
```

Pro příklad definice a použití datové struktury vyjdeme ze záznamu poruchové události. Tu bude představovat záznam času, kdy poruch vznikla, záznam času, kdy byla odstraněna a číselný kód poruchy, který budeme volit tak, aby šel překódovat do textu pomocí tabulky textů jak to popisuje odstavec „Řetězce znaků jako zvláštní typ polí“. Na Obr. 8 je schématicky znázorněna datová struktura záznamu typu „porucha“. Záznam je tvořen dvěma položkami typu „cas“ a jednou položkou typu „byte“, která obsahuje kód poruchy. Typ „cas“ představuje datovou strukturu s kopií systémových registrů reálného času v době vzniku a odstranění poruchy. Zápis definice datového typu bude vypadat takto:



Obr. 8 Datová struktura porucha

```
type struct
word month, word day, word hour,
word minute, word second
end  cas ;definice struktury cas.

type struct
cas vznik, cas konec,
word kod
end  porucha ;definice struktury porucha
```

Pokud máme takto definovanou strukturu „porucha“, můžeme tento uživatelský typ použít pro deklaraci proměnné. Na jednotlivé prvky datové struktury přistupujeme pomocí jména položky odděleného tečkou. Ukažme příklad pro naplnění položky „hour“ v položce „vznik“

```
var porucha zaznam ; deklarace datového typu porucha
zaznam.vznik.hour = HOUR
```

/* naplnění položky „hour“ v položce „vznik“ obsahem systémového registru reálného času */

Shrnutí:

S pomocí klíčového slova „type“, „struct“ a „end“ umožňuje jazyk Simple 4 definovat uživatelské datové typy ve formě datové struktury. Položky struktury mohou mít libovolný známý (definovaný) datový typ. **Je** tedy **možné**, pokud spojíme zde popsané poznatky s odstavcem „Deklarace a použití polí“, **definovat strukturu struktur, strukturu polí, pole struktur**. K jednotlivým položkám struktur přistupujeme s pomocí jejich názvu a oddělujeme je od názvu proměnné pomocí „.“ (tečky).

Tabulky dat

Tabulky dat jsou vhodným nástrojem **pro uložení seznamů textů**, iniciačních nebo parametrických konstant, tabulek **konverzních hodnot** apod. do kódové paměti automatu (FLASH EEPROM). Zadány mohou být datové tabulky libovolných, tj. i uživatelských typů včetně struktur a polí. Zápis tabulek dat odpovídá formálně zápisu:

table datový_typ[pocet_polozek] jméno_tabulky = (seznam_hodnot, seznam_hodnot....)

Pod seznamem hodnot je nutné si představit

- **zápis samostatné hodnoty** - jednoduchý datový typ
- **seznam hodnot uzavřený do okrouhlých závorek** - položka složeného datového typu.

Typickým příkladem pro použití tabulek je seznam textů. Setkáváme se s ním např. v režimech chodu technologie „VYP“, „ZAP“ nebo „MANUAL“, „AUTOMAT“, „ODSTAVKA“ atp. Seznamy textů hojně využívá knihovna „menu“ a knihovna „mar“ pro zobrazování a editaci výčtových datových typů (datová proměnná, která nabývá vybrané hodnoty ze seznamu povolených hodnot). Příklady použití tabulek jsou:

- **Tabulky textů**

  table string[2] text = („text 1“, „text 2“)
  Použití:
  Display(text[0])

- **Tabulky datových typů**

  table bit[8] bits = (1,1,0,1,0,1,0,1)
  Použití:
  Y[0] = bits[7]

- **Tabulky složených typů**

  type struct
  int typ_behu,
  int typ_vyhodnoceni,
  bit[3] priznaky
  end popis
  table popis[2] data = ((6,3,(0,1,0)),(9,11,(1,1,0)))
  Použití:
  Y[0] = data[1].priznaky[2]

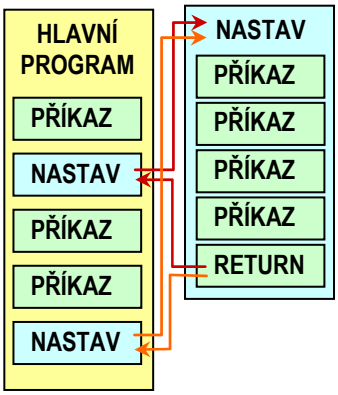
Shrnutí:

Z uvedených příkladů je zřejmé, že **tabulky dat** vždy představují **pole s položkou daného datového typu.** Jako každé pole musí mít definován počet položek viz. formální zápis. V případě, že chceme uložit do kódové paměti automatu pouze jednu datovou položku můžeme buď zadat počet položek tabulky 1 nebo můžeme místo tabulky použít klíčové slovo „code“, které je synonymem k tabulce o jedné položce. **Pokud je položka tabulky datová struktura, je nutné při zadávání hodnot oddělit pomocí okrouhlých závorek jednotlivé seznamy hodnot.**

Deklarace procedury a funkce

Procedury a funkce představují vícenásobně spustitelné části programu z hlavní programové smyčky. Začlenění procedury nebo funkce do řídicího programu automatu a jejich použití ukazuje Obr. 9. Procedura se jménem „nastav“ je volána dvakrát během vykonávána hlavní programové smyčky. Dvojí volání ve většině případů smysl pouze tehdy, pokud se do procedury předávají při každém volání jiné parametry a nebo tehdy, když obsahuje velké množství příkazů jejichž dvojí kopie by obsadila příliš mnoho programové paměti automatu.

Funkce se od procedury liší tím, že funkce vrací hodnotu zatímco procedura ne. Funkce tedy může být součástí výrazu, zatímco procedura může stát v textu pouze mimo výraz. Procedura a funkce mohou a nemusí mít parametry, jejichž konkrétní hodnota (pokud jsou) je předávána v okamžiku volání procedury nebo funkce. Parametry mohou být předávány „odkazem“ nebo „hodnotou“. Při předávání odkazem se předává adresa proměnné, při předávání hodnotou se předává hodnota proměnné. Složené datové typy je možné předávat pouze odkazem. Typ „string“ se vždy předává s atributem „const“, který označuje, že proměnná nemůže stát na levé straně přiřazovacího příkazu.



Obr. 9 Princip použití procedury

Deklarace funkce odpovídá formálně zápisu:

function zákl_typ jméno_funkce(seznam_parametrů)

Seznam parametrů označuje jednotlivé parametry funkce oddělené čárkou. Zápis jednotlivých parametrů odpovídá některému s následujících tvarů:

- **datový_typ jméno_parametru** → parametr předávaný hodnotou
- **const datový_typ jméno_parametru** → parametr předávaný odkazem, nesmí být použit na levé straně přiřazovacího příkazu
- **var datový_typ jméno_parametru** → parametr předávaný odkazem, může být použit na levé straně výrazu.

Deklarace procedury odpovídá zápisu:

subroutine jméno_procedury(seznam_parametrů)

Seznam parametrů i jejich zápis je v případě procedury totožný se zápisem parametrů funkce.

Shrnutí:

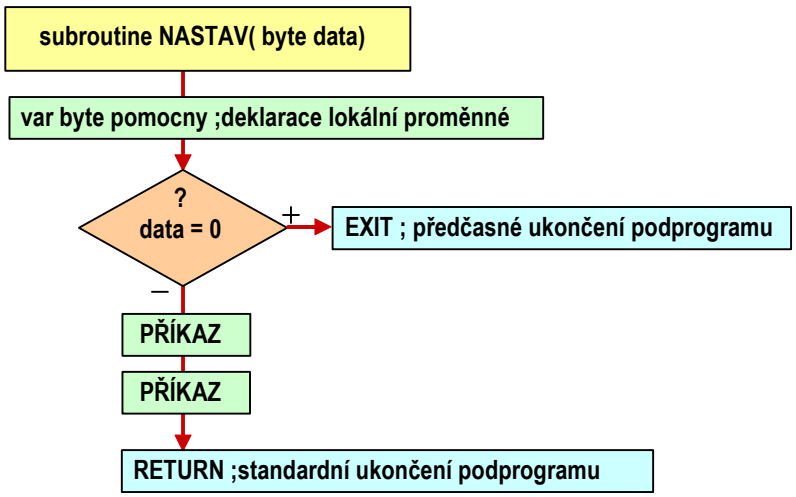
Procedury nebo funkce vytváříme tehdy, pokud potřebujeme vykonat sadu příkazů programu pro různé datové položky. Příkladem může být regulátor UT, který bude mít pro všechny větve UT stejnou sadu příkazů, nicméně každá z regulovaných větví bude poskytovat jiná aktuální data. Dalším vhodným případem pro použití funkce nebo procedury jsou často se opakující úseky programu jenž je potřeba vykonávat na různých místech. Typickým příkladem je např. mazání displeje u automatů řad MPC300, K a MT201.

Použití procedury nebo funkce je naopak nevýhodné, pokud je opakovaný úsek programu krátký tj, pokud je tvořen několika instrukcemi. V takovém případě je lepší příkazy opsat nebo zkopírovat.

Předávání parametrů, zápis těla funkcí a podprogramů

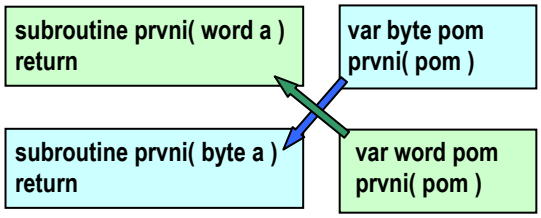
Tělo podprogramů a funkcí obvykle obsahuje:

- **deklaraci lokálních** pomocných **proměnných** jejichž jména jsou platná a známá pouze uvnitř podprogramu nebo funkce
- **výkonné příkazy** pro zpracování dat z parametrů
- **příkaz exit** ukončující předčasně podprogram nebo funkci. V případě funkce je doplněný o výraz předávající návratovou hodnotu např. exit 0 nebo exit data atd.. Příkaz se dá výhodně použít ve spolupráci s podmíněným příkazem, kdy z vyhodnocení podmínky plyne, že není nutné vykonávat zbytek podprogramu nebo funkce.
- **příkaz return** ukončující podprogram nebo funkci. V případě funkce je doplněný o výraz předávající návratovou hodnotu např. return 0 nebo return data atd....



Obr. 10 Struktura podprogramu a funkce

Strukturu podprogramu a funkce demonstruje Obr. 10.



Obr. 11 Princip přetěžování parametrů

Samostatnou kapitolu v popisu podprogramů a funkcí tvoří problematika předávání parametrů. Programovací jazyk Simple 4 podporuje tzv. **přetěžování parametrů funkcí a procedur**. Tato vlastnost ve svém důsledku umožňuje zapsat několik funkcí se stejným jménem ale s různými typy parametrů. Kterou z takto deklarovaných funkcí překladač použije, vyplyne z kontextu volání, kdy překladač hledá takovou variantu procedury nebo funkce u níž se typy parametrů co nejvíce blíží předaným typům proměnných. Situaci dokumentuje Obr. 11, kde šipky ukazují, kterou z variant vybere podle kontextu volání překladač.

Shrnutí:

Pro používání funkcí a podprogramů je k dispozici předávání parametrů odkazem nebo hodnotou, přetěžování parametrů funkcí a procedur a možnost předčasného ukončení procedury nebo funkce pomocí klíčového slova „exit“. **Odkazem nelze předávat proměnné typu bit a síťové proměnné libovolného typu**. Hodnotu lokálních proměnných nelze zobrazit ve vývojovém prostředí StudioWin.

Bitové operace

Pro ovládání digitálních vstupů a výstupů a obecně bitových proměnných jsou kromě přiřazovacího příkazu určeny i bitové operace. Pomocí bitových operací můžeme provádět **operace bitového součtu, součinu, výhradního součtu a negace**. Při použití bitových operací využíváme znalostí pravdivostních tabulek pro jednotlivé typy operací a též jejich priority.

V Tab. 11 jsou uvedeny pravdivostní tabulky všech bitových operací nabízených syntaxí

A	B	Negace A‘	Součet „|“	Výhradní součet „^“	Součin „&“
0	0	1	0	0	0
0	1	1	1	1	0
1	0	0	1	1	0
1	1	0	1	0	1

Tab. 11 Podporované bitové operace

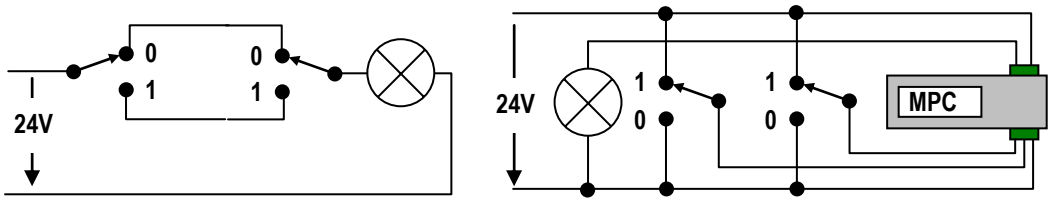
jazyka Simple 4 a to včetně vztahu k prioritě operace. Zmíněné operace můžeme používat ve výrazu na pravé straně přiřazovacího příkazu nebo v podmínce podmíněného příkazu nebo při předávání parametrů podprogramů a funkcí.

- Použití bitových operací ve výrazech

  Y[0] = X[0] | X[1] ; sepne výstup je-li alespoň na jednom vstupu log. 1
  Y[0] = X[0] & X[1] ; sepne výstup jsou-li na obou vstupech log. 1

- Realizace schodišťového spínače

Na Obr. 12 je typické zapojení schodišťového spínače, tj. zapojení dvojice přepínačů s nimiž se dá ovládat osvětlení ze dvou míst.



Obr. 12 Schematická realizace schodišťového spínače

Bitový výraz, který realizuje stejnou funkci pomocí dvojice vstupů automatu a jednoho výstupu má tvar:

```
Y[0] = (X[0] ^ X[1])‘
/*potřebná funkce odpovídá negaci výhradního součtu
  dvojice vstupních signálů */
```

Shrnutí:

V zápisu příkladu se objevují krom bitového výrazu ještě **kulaté závorky s jejichž pomocí řešíme prioritu ve zpracování výrazu** obdobně jako při zápisu aritmetických výrazů. Dále pak je zde ukázán **blokový komentář**, který je uvozen dvojznakem „/*“ a uzavřen dvojznakem „*/“. Text komentáře se nevyhodnocuje a slouží ke zpřehlednění zdrojového textu.

Aritmetické operace a operace bitových manipulací

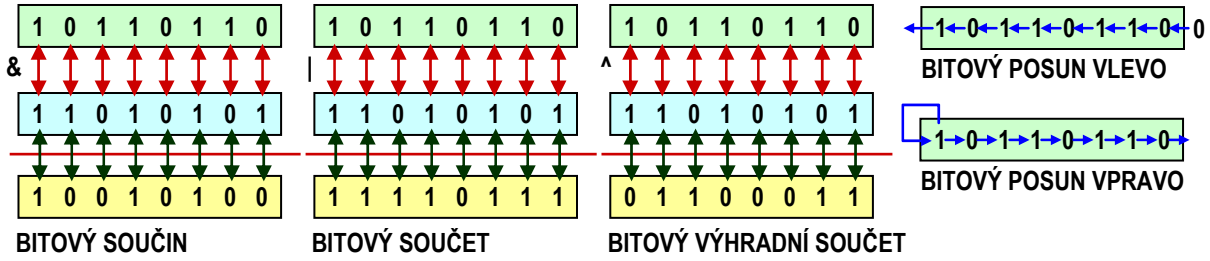
Programovací jazyk Simple 4 podporuje u základních datových typů všechny základní aritmetické operace společně s operacemi pro bitové manipulace. Podporované operace pro aritmetické datové typy shrnuje Tab. 12. Datový typ bit podporuje jazyk bitovými operacemi, které shrnuje odstavec „Bitové operace str. 27“.

	+	-	*	/	%	|	&	<<	>>	^
byte	√	√	√	√	√	√	√	√	√	√
word	√	√	√	√	√	√	√	√	√	√
int	√	√	√	√	√	√	√	√	√	√
longword	√	√	√	√	√	√	√	√	√	√
longint	√	√	√	√	√	√	√	√	√	√
float	√	√	√	√[1]	-	-	-			-

[1] - u typu float je operace dělení v plovoucí čárce, jinak je celočíselná.

Tab. 12 Datové typy a podporované operace

Vlastnosti jednotlivých aritmetických operací jsou standardní včetně priority násobení a dělení před součtem a rozdílem. **Pro celočíselné typy** jsou k dispozici **operace typu zbytek po dělení „%“ a dále pak operace bitových manipulací**, kdy se operandy zpracovávají bit po bitu. Programovací jazyk podporuje bitový součin (and „&“), bitový součet (or „|“) a výhradní bitový součet (xor „^“). Jednotlivé operace bitových manipulací si můžeme představit tak, že bity obou operandů se na odpovídajících pozicích provede zadaná bitová operace podle pravdivostní tabulky (viz. Tab. 11). Pokud se týká operací bitových posunů dochází k posunu jednotlivých bitů o zadaný počet vlevo nebo vpravo s tím, že při posunu vlevo se prázdné místo doplňuje 0, při posunu vpravo se kopíruje nejvyšší bit. Postup je přehledně dokumentován na Obr. 13.



Obr. 13 Znázornění operací bitových manipulací

Shrnutí:

Pro zpracování dat řídicími automaty MICROPEL poskytuje programovací jazyk Simple 4 základní aritmetické operace pro součet („+“), rozdíl („-“), součin („*“) a podíl („/“). Operace podílu je u celočíselných typů chápána jako celočíselná, u typu float pak v plovoucí řádové čárce. Pro operandy celočíselných datových typů je na rozdíl od typu float k dispozici ještě operace zbytek po celočíselném dělení („%“) a dále operace pro bitové manipulace typu bitový součet („|“), součin („&“), výhradní součet („^“), bitový posun vlevo („<<“) a bitový posun vpravo („>>“).

Přiřazení, výraz, složený výraz

Přiřazení je základním kamenem zdrojového textu v jazyce Simple 4. Najdeme ho obvykle jako cílový úkon při čtení a vyhodnocování vstupů, při ovládání výstupů a při aritmetických a bitových operacích jako operaci pro uložení výsledku. Přiřazení můžeme formálně zapsat jako:

Cíl = Zdroj

Příkladem pro přiřazení může být zdrojový text:

```
;deklarace proměnných pomocí seznamu
var byte a_byte, b_byte, c_byte
a_byte = c_byte

var bit a_bit, b_bit
a_bit = b_bit
```

Pojmem **výraz** označujeme **zápis operandů vzájemně svázaných aritmetickými nebo bitovými operátory**. Ve výrazu je automaticky uplatňována standardní hierarchie operací a to včetně závorkových konvencí. Jako příklad uveďme zdrojový text:

```
var byte a_byte, b_byte, c_byte
a_byte = (c_byte + b_byte) * a_byte
a_byte = c_byte | b_byte & a_byte

var bit a_bit, b_bit, c_bit
a_bit = b_bit | a_bit & c_bit
```

Složený výraz je žádný, jeden nebo **skupina výrazů uzavřených mezi klíčová slova „begin“ a „end“**. Při psaní zdrojového textu, zvláště pak u podmíněných výrazů, je někdy nutné provést více než jeden příkaz. Protože podmíněný příkaz předpokládá vykonání jediného příkazu v závislosti na vyhodnocované podmínce, pomůže nám v tomto případě použití složeného příkazu vykonat více než jeden běžný příkaz. Složený příkaz ve zdrojovém textu zapisujeme následujícím způsobem:

```
var byte a_byte, b_byte, c_byte
var bit a_bit, b_bit, c_bit
begin
a_byte = (c_byte + b_byte) * a_byte
a_byte = c_byte | b_byte & a_byte
a_bit = b_bit | a_bit & c_bit
end
```

Shrnutí:

Přiřazení, výraz a složený výraz jsou základními prvky programové konstrukce. Syntaxe programovacího jazyka Simple 4 podporuje volné formátování textu pro zvýšení jeho přehlednosti a úpravy. Aby bylo možné zdrojový text volně formátovat i v případech, kdy syntaxe jazyka podporuje zpracování jednoho prostého příkazu, je k dispozici příkaz složený. Výhoda složeného příkazu je nejlépe vidět v případě podmíněného příkazu (viz. **Podmíněný příkaz**) nebo v případě konstrukce programového přepínače (viz. **Programový přepínač**)

Přístup k bitu

Pod pojmem **přístup k bitu** můžeme rozumět hned několik variant programových konstrukcí pro manipulaci s hodnotou jednotlivého bitu v celočíselných proměnných. Jedním z možných postupů je použít operací pro **bitové manipulace** (v příkladech jsou použité proměnné typu word a hexadecimální zápis konstant) s nimiž můžeme například:

- nulovat vybraný bit

```
a_word = a_word & 0xFFF7 ;nulování bitu B3 v proměnné a_word
a_word = a_word & 0xFFF3 ;nulování bitu B3 a B2 v proměnné a_word
```

- nastavit vybraný bit

```
a_word = a_word | 0x0008 ;nastavení bitu B3 v proměnné a_word
a_word = a_word | 0x000C ;nastavení bitu B3 a B2 v proměnné a_word
```

- negovat vybraný bit

```
a_word = a_word ^ 0x0008 ;negace bitu B3 v proměnné a_word
a_word = a_word ^ 0x000C ;negace bitu B3 a B2 v proměnné a_word
```

- kopírovat vybraný bit

```
a_word = (a_word & 0x0010') | ((a_word & 0x0008) << 1) ; bit B3 na B4
a_word = (a_word & 0x0030') | ((a_word & 0x000C) << 2)
;kopie bitu B3 a B2 na pozici B4, B5
```

Pokud budeme zkoumat předchozí zdrojový text dojdeme k závěru, že pomocí **operací pro bitové manipulace můžeme v jedné operaci pracovat s jedním i více bity**. V řadě případů v praxi ovšem potřebujeme manipulovat pouze s jedním bitem. Pro manipulaci s jedním bitem však není použití bitových manipulací optimální. Pro tento případ je k dispozici speciální operace, která se označuje názvem přístup k bitu. Formální zápis operace odpovídá tvaru:

```
celočíselná_proměnná ? číslo_bitu
```

V uvedeném zápisu musí být číslo bitu konstanta. Zajímavou vlastností operace přístupu k bitu je to, že může být použita i na levé straně přiřazovacího příkazu. S tímto poznatkem tedy můžeme pracovat s jednotlivým bitem proměnné word takto:

- **nulovat vybraný bit**

```
a_word ? 3 = 0  ;nulování bitu B3 v proměnné a_word
```

- **nastavit vybraný bit**

```
a_word ? 3 = 1 ;nastavení bitu B3 v proměnné a_word
```

- **negovat vybraný bit**

```
a_word ? 3 = a_word ? 3‘  ;negace bitu B3 v proměnné a_word
```

- **kopírovat vybraný bit**

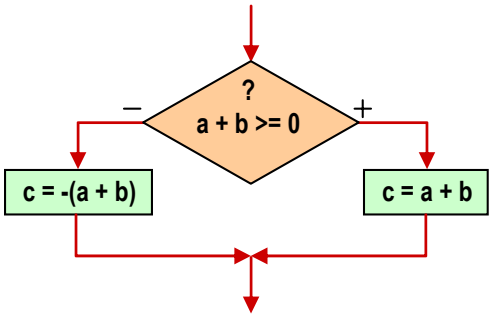
```
a_word ? 4 = a_word ? 3 ;kopie bitu B3 na pozici B4
```

Shrnutí:

S pomocí bitových manipulací můžeme efektivně pracovat s vybranými bity celočíselných proměnných. **V případě, že potřebujeme pracovat s jedním bitem použijeme s výhodou operace „přístup k bitu“**.

Podmíněný příkaz

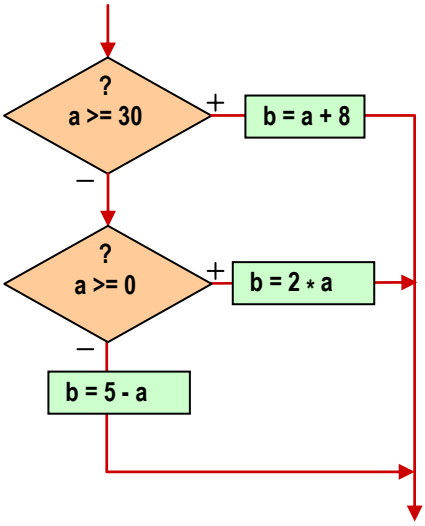
Podmíněný příkaz slouží **k větvení programu podle platnosti podmínky příkazu**. Výkonná část příkazu má dvě větve kódu. Jedna větev se vykoná tehdy, pokud je podmínka platná, druhá pak v případě, že podmínka neplatí. Situaci znázorňuje Obr. 14. Z obrázku je patrné, že se testuje součet proměnných „a“ a „b“ vůči nule. Pokud podmínka platí (součet je větší nebo roven nule), uloží se do proměnné c právě výsledek tohoto součtu. Pokud podmínka neplatí, tj. výsledek součtu je záporné číslo, uloží se do proměnné hodnota výsledku s opačným znaménkem, tj. číslo kladné. Dá se tedy ukázat, že uvedený podmíněný příkaz realizuje funkci absolutní hodnoty součtu dvou proměnných „a“ a „b“. Zápis podmíněného příkazu v jazyce Simple 4 odpovídá formálně tvaru:



Obr. 14 Zobrazení podmíněného příkazu

if podmínka then příkaz else příkaz, kde na místě „příkaz“ může být jednoduchý příkaz, složený příkaz nebo podmíněný příkaz. Zápis podmíněného příkazu z Obr. 14 tedy je:

```
if (a+b) >= 0 then c = a+b else c = -(a+b)
```



Obr. 15 Řetězený podmíněný příkaz

Další typickou možností **využití podmíněného příkazu je třídění hodnot** podle nějakého klíče. Na Obr. 15 je uvedena úloha roztřídit hodnotu a na větší nebo rovnu 30, na menší než 30 a současně větší než 0 a na menší než 0. Pokud bychom využili jednoduchého podmíněného příkazu, obdrželi bychom pro prostřední skupinu hodnot podmínku složenou ze dvou operací a to a < 30 a a >= 0. Pokud seřadíme podmínky třídění sestupně nebo vzestupně, můžeme využít pro třídění v této prostřední skupině neplatné větve z prvního podmíněného příkazu. Zápis zdrojového textu má v tomto případě tvar:

```
if a >= 30 then b = a + 8
else if a >= 0 then b = 2 * a
else b = 5 - a
```

Shrnutí:

Podmíněný příkaz slouží k větvení programu. Jazyk Simple 4 podporuje prostý i řetězený tvar podmíněného příkazu. Podmíněný příkaz je k dispozici včetně části, která se vykoná tehdy, pokud není splněna podmínka příkazu. Pokud je nutné vykonat více příkazů a to ve větvi, kdy podmínka platí nebo ve větvi, kde podmínka neplatí, použijeme pro zápis požadovaných příkazů příkaz složený.

Použití bitu RESET

Systémový **bit RESET se nastavuje automaticky po zapnutí napájení a spuštění aplikace** automatu. Vzhledem k tomu, že detekce zapnutí automatu je mnohdy naprosto stěžejní záležitost pro správnou inicializaci řídicích a regulačních funkcí celé aplikace, je nutné chovat se k bitu RESET dostatečně sofistikovaně, aby nemohlo dojít k nesprávné interpretaci jeho hodnoty. Princip použití bitu je v zásadě jednoduchý. Aplikace zjistí, že byl nastaven bit RESET. Provede volání inicializačních procedur a bit RESET nastaví do 0. Tím se zabezpečí to, že aplikace bude provádět inicializační procedury právě jednou. Typickou programovou konstrukci v tomto případě představuje zápis:

```
if RESET then
        begin
                Inicializace()
                RESET = 0
        end
MeInit(0,4)
```

Uvedená konstrukce zpracování bitu RESET, byť se zdá logická, není v řadě případů vhodná. Pokud totiž budeme používat např. knihovní funkce menu nebo knihovnu MaR, jak je uvedeno v textu, můžeme se dočkat nepříjemností. Ty spočívají v tom, že obě knihovny, byť bit RESET využívají, tak ho nenulují. To je z důvodu, aby poskytly informaci o RESETu i navazujícím částem aplikace. Podobně by se tedy měla chovat i aplikace jako celek. Je zřejmé, že výše uvedený text takové chování nerespektuje. Za podmíněným příkazem pro zpracování bitu RESET následuje volání první procedury MeInit právě zmíněné knihovny Menu, ale to už je bit vynulován. Procedura MeInit musí být volána v každém průběhu programové smyčky. Současně využívá bit RESET pro získání informace o zapnutí automatu a z toho plyne nevhodné umístění nulování bitu RESET v uvedeném zdrojovém textu. Řešení problému spočívá v tom, že nulování **bitu RESET přesuneme až na závěr zdrojového textu jako poslední nepodmíněný příkaz**. Bit RESET se tak bude sice nulovat při každém průchodu programovou smyčkou ale to principiálně nevadí. Ztráta času je dokonce nižší, než kdybychom nulování bitu podmínili. Zdrojový text odpovídající tomuto rozboru má tvar:

```
if RESET then
        begin
                Inicializace()
        end
MeInit(0,4)
..................................... ; programové řádky aplikace
.....................................
RESET = 0
end
```

Shrnutí:

Nulování bitu RESET provádíme vždy na konci zdrojového textu na místě posledního vykonávaného příkazu hlavní programové smyčky. Umožníme tak reakci na stav RESET všem částem programu včetně funkcí použitých knihoven. Nulování bitu provádíme nepodmíněně v každé programové smyčce, protože podmíněné volání požaduje delší čas a větší délku programového kódu.

Logický výraz

Pod pojmem logický výraz máme na mysli výraz nebo výrazy, s jejichž pomocí **konstruujeme podmínku podmíněného příkazu**. Podmínka se skládá z jednotlivých logických výrazů pro porovnání hodnot, kdy hodnotou může být například výsledek aritmetického výrazu, návratová hodnota funkce apod. Jednotlivé logické výrazy spojujeme do výsledného tvaru podmínky pomocí logických operátorů. Všechny logické výrazy včetně operátorů pak tvoří logický výraz podmínky.

Operátory porovnání					
>	**>=**	**=**	**<**	**<=**	**<>**
větší	**větší nebo rovno**	**rovno**	**menší**	**menší nebo rovno**	**různý**

Tab. 13 Operátory porovnání pro logické operace

Tab. 13 shrnuje operátory porovnání pro logické výrazy, které podporuje jazyk Simple 4. **Operátory porovnání představují základní prvky logického výrazu**, neboť výsledek porovnání je buď „platí“ nebo „neplatí“. Společně s těmito operátory vystupují v logickém výrazu logické operátory které slouží k logickému spojování operací porovnání. K dispozici je operátor:

- **and** - operátor logického součinu
- **or** - operátor logického součtu
- **not** - operátor logické negace

Použití operátorů v logickém výrazu podmínky podmíněného příkazu můžeme dokumentovat příkladem:

if a > 5 and a < 10 or a <> 0 then a = 1

Pro vyhodnocování logického výrazu platí samozřejmě priority logických operací, kdy nejvyšší prioritu má vždy porovnání. Tím se získají údaje o platnosti nebo neplatnosti porovnávacích operací a výsledek se vyhodnotí pomocí operátorů and, or a not.

Shrnutí:

Logický výraz najdeme v podmínce podmíněného příkazu. Sestává ze zápisu jednotlivých porovnání hodnot proměnných a výsledky porovnání spojuje pomocí logických operátorů do výsledné podmínky. V případě, že testujeme hodnotu na nulovost nebo nenulovost operací, nemusíme porovnání zapisovat podmínku a zápis pak vypadá např. takto:

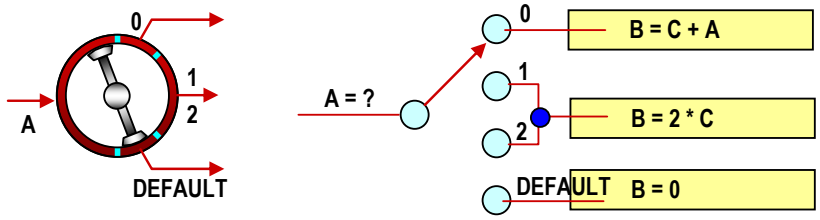
if a then .....

Osamocená proměnná a ve skutečnosti znamená zkrácený zápis

if a <> 0 then .....

Programový přepínač

Programový přepínač je obdobou běžného elektrického přepínače ovšem v podobě zápisu zdrojového textu. Tato **konstrukce** nám **umožňuje realizovat několik větví programu** a volit mezi nimi na základě hodnoty vybrané proměnné. Všechny podporované varianty větví



Obr. 16 Analogie mezi elektrickým a programovým přepínačem

programového přepínače vidíme na Obr. 16. Pro případ, že je hodnota proměnné A = 0, znázorňuje obrázek **jednoduchou variantu** programové větve. **Násobnou programovou větev** představují hodnoty 1 a 2 proměnné A. V násobné větvi se vykoná pro obě hodnoty stejný příkaz. Posledním typem je větev „default“, jejíž příkaz se vykoná ve všech nejmenovaných (nepoužitých) případech hodnoty proměnné A.

Zápis programového přepínače odpovídá formálnímu tvaru:

switch (aritmetický_výraz)
case hodnota: výraz
.....
case hodnota: výraz
default:
end

Na jeho základě vytvoříme zdrojový text pro případ zobrazený na Obr. 16. Zdrojový text bude mít podobu:

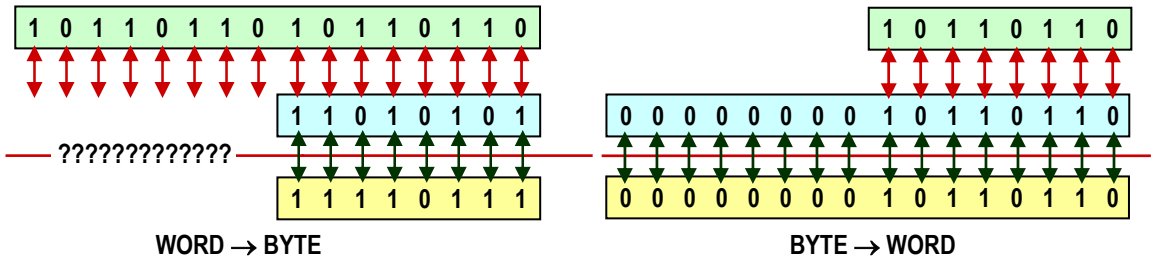
```
switch (A)
case 0: B = C + A ; jednoduchá programová větev
case 1:
case 2: B = 2 * C ; násobná programová větev
default: B = 0 ; větev pro nespecifikované hodnoty proměnné A
end
```

Shrnutí:

Programová konstrukce „switch“ umožňuje realizovat programový přepínač, představující konstrukci, která na základě hodnoty aritmetického výrazu volí jednu z variant programového kódu (příkazu). Na pozici „aritmetický_výraz“ formálního zápisu programového přepínače může stát, jak plyne z příkladu, i samostatná proměnná. V takovém případě není nutné uvádět okrouhlé závorky. Na pozici „výraz“ zmiňovaného zápisu můžeme použít i složený nebo podmíněný příkaz. Je zde možné samozřejmě použít i další programový přepínač, neboť ten je pouze speciálním případem podmíněného příkazu.

Přetypování

Přetypování představuje operaci, která umožní **přizpůsobení jednotlivých typů proměnných**. Do situace, kdy přetypování potřebujeme, se dostáváme tehdy, pokud potřebujeme předat hodnotu mezi nestejnými typy proměnných. **V některých případech** o tomto problému ani nevíme, protože **ho** za nás **řeší automaticky překladač jazyka** Simple 4. Jsou však případy, kdy uvedený problém vyřešit automaticky nelze.



Obr. 17 Princip operace přetypování

Na Obr. 17 jsou uvedeny dva principielní případy přetypování. Případ vlevo ukazuje situaci, kterou překladač automaticky vyřešit neumí a tou je situace, kdy předáváme např. datový typ „word“ do proměnné typu „byte“. Jak je znázorněno na obrázku při tomto předávání ztratíme vyšších 8 bitů proměnné word. Tato ztráta nemusí a nebo může být na závadu. To, zda je na závadu a nebo není, nemůže posoudit překladač, nýbrž to musí udělat programátor z kontextu úlohy, kterou řeší. Pokud se programátor rozhodne, že uvedené oříznutí bitů nevadí sdělí to překladači operací přetypování. Tím odstraní chybové hlášení a umožní vygenerovat výsledný přeložený kód. Zápis operace přetypování má formální tvar:

nový_typ (aritmetický_výraz)

Přetypování pro případ zobrazený na obrázku zapíšeme takto:

b = byte (w)

Případ vpravo na Obr. 17 naopak překladač vyřeší automaticky tak, že chybějící bity hodnoty doplní 0. Tak obdrží číslo, které má v proměnné typu „word“ stejnou hodnotu jako v proměnné typu „byte“.

Případů kdy proměnnou nelze přetypovat je několik. Uveďme ale nejdůležitější dva. Prvním je případ přetypování **aritmetická_proměnná → bit**. Pokud potřebujeme tento případ řešit musíme si pomoci podmíněným příkazem např.

if a = 0 then b = 0 else b = 1

Druhý případ je ten, kdy se proměnná do podprogramu nebo funkce předává odkazem. Na pozici této proměnné musí být proměnná daného typu. Pokud potřebujeme tento případ řešit, je nutné použít pomocnou proměnnou a hodnotu pro předání parametru předávat pomocí této pomocné proměnné.

Shrnutí:

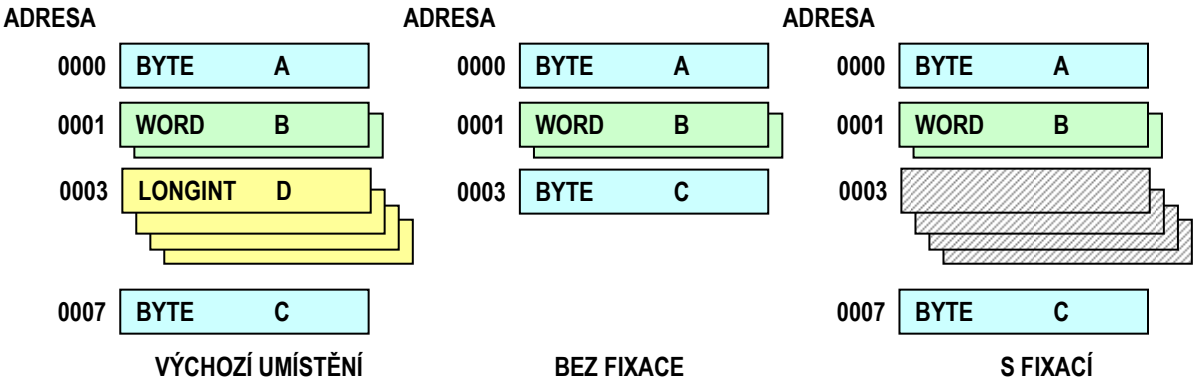
Pomocí operace přetypování můžeme v řadě případů přizpůsobit „bitovou šířku“ proměnných použitých ve výrazu. Pokud to není možné a to je v případech, kdy potřebujeme přetypovat proměnnou na hodnotu typu bit a nebo v případě předávání proměnné do procedury nebo funkce odkazem, musíme si vypomoci pomocnými programovými konstrukcemi podle povahy řešeného problému.

Fixace proměnných

V některých případech aplikací jsme postaveni před **problém stanovení pevných adres uživatelských proměnných**. To je v případech pokud chceme navázat na program automatu některý typ vizualizace. Vizualizace jako taková, je založena na zobrazování případně editaci hodnot datových proměnných aplikace a mnohdy představuje sama o sobě poměrně rozsáhlý a složitý projekt. Propojení vizualizačních kanálů je obvykle nutno řešit ručně a proto je vhodné, aby zůstalo správně definováno i v případě úprav řídicího programu v automatu. To je možné pouze v případě, že použijeme tzv. fixaci adres proměnných.

Přidělování adres při překladu zdrojového textu provádí automaticky překladač a není možné zaručit, že nedojde ke změnám adres proměnných v průběhu vývoje případně úprav zdrojového textu. Pokud je na adresy proměnných napojena vizualizace, dojde k tomu, že jednotlivé kanály vizualizace jsou odtrženy od svých zdrojů a je nutné projekt vizualizace upravit.

Aby k tomuto nežádoucímu jevu nedocházelo, jsou k dispozici funkce, které umožňují adresy proměnných fixovat. Princip fixace adres ukazuje Obr. 18.



Obr. 18 Principielní zobrazení fixace proměnných

K tomu abychom mohli proměnnou úspěšně fixovat využíváme dvě předdefinované funkce a jeden příkaz pevného umístění (fix):

- **sizeof(jméno_proměnné)** - funkce vrací velikost proměnné v bajtech
- **endof(jméno_proměnné)** - funkce vrací adresu, která následuje za posledním byte proměnné
- **fix jméno_proměnné = (adresa_proměnné,velikost)** - definuje pro symbol proměnné parametry umístění v datové paměti. Jedná se o počáteční adresu a velikost vyhrazeného prostoru v bajtech. Požadavek na fixaci proměnných dle Obr. 18 můžeme zapsat:

  fix A = (0,sizeof(A))
  fix B = (endof(A),sizeof(B))
  fix C = (7,sizeof(C))

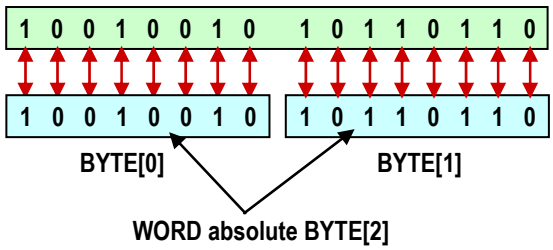
Shrnutí:

Pro fixaci proměnných je v prostředí StudioWin k dispozici „pomocník“. Jeho služeb využijeme nejlépe po prvním úspěšném odladění programu. „Pomocník“ na základě výsledků překladu nabídne seznam proměnných a s pomocí zaškrtnutí požadavku na fixaci, provede automatické vygenerování sady výše uvedených příkazů pro vybrané proměnné.

Pokud proměnná se jménem použitým pro fixaci neexistuje, považuje překladač fixovanou oblast za vyhrazenou část paměti, která je označena jménem.

Konstrukce absolute

Konstrukce absolute je určena pro zkušené a znalé programátory. Primárně je tato konstrukce určena pro definování umístění speciálních proměnných, které souvisí s „hardwarem“ automatů. S její pomocí jsou například definovány proměnné reálného času, časovače atd. Sekundárně mohou tuto konstrukci využívat i programátoři aplikací. Její použití nicméně vyžaduje jisté hlubší znalosti. Dále pak je nutné poznamenat, že **v budoucnu nemusí tato konstrukce fungovat na všech typech automatů stejně**. Tato posledně zmíněná vlastnost je tedy asi tou nejzávažnější, která hovoří spíše proto, abychom se konstrukci „absolute“ vyhnuli. Princip konstrukce „absolute“ ukazuje Obr. 19.



Obr. 19 Grafické znázornění konstrukce absolute

Konstrukce ve své podstatě **překrývá různé typy proměnných stejné délky,** tzv. přes sebe, tj. umísťuje je v datové paměti na stejnou adresu. Zajímavý je v tomto kontextu pojem stejná délka, protože v uvedeném příkladu nelze dosáhnout stejné délky proměnné typu word a byte. Je možné však stejné délky dosáhnout, a to je uvedený případ, pro proměnnou word a proměnnou představující pole dvou byte. Indexem v poli pak určujeme zda používáme vyšší nebo nižší byte proměnné „word“. To na který z obou byte míří index 0 je dáno typem uložení vícebajtových proměnných v paměti automatu. Existuje typ „big“ a „little endian“. Který z obou typů se v tom či onom případě volí, závisí na typu mikroprocesoru použitého v tom či onom typu automatu. Z hlediska použití konstrukce se oba typy uložení liší právě v umístění vyššího a nižšího byte. Ty jsou umístěny buď tak, jak ukazuje obrázek a nebo jsou prohozeny. Pro automaty řady MPC300, K a terminál MT201 platí rozmístění podle Obr. 19. Stejné rozmístění i v budoucích řadách automatů nelze zaručit.

Formální tvar zápisu konstrukce absolute je:

var typ_proměnné jméno_proměnné absolute jméno_proměnné

Pro případ uvedený na obrázku tedy můžeme konstrukci zapsat takto:

```
var word word_as_word
var byte[2] word_as_byte absolute word_as_word
```

Shrnutí:

Konstrukce absolute umožňuje generovat jednoduchý a rychlý přístup, k jednotlivým byte vícebajtových proměnných. Její nevýhodou je, že výsledek přístupu je závislý na typu umístění vícebajtových proměnných v datové paměti automatu a neměnnost tohoto umístění není možné do budoucna zaručit.

3 Základní úlohy z programování v jazyce Simple 4

Přestože se to nezdá, tak pokud řešíme některé i velmi jednoduché úlohy nevhodně, nebudeme s funkčností napsaného kódu spokojeni. Nespokojenost s kódem se nejrychleji projeví při zobrazování na displeji automatu. V této základní úloze nalézáme v řadě případů zásadní chyby. Abychom se jich vyvarovali, musíme mít vždy na paměti jak funguje zobrazování na displej.

Jak funguje zobrazování

Tisk na virtuální obrazovku a vlastní fyzický zápis dat do obvodů displeje jsou dva zcela nezávislé děje, které běží v automatu paralelně. Odezva elektroniky displeje je totiž pomalá a kdyby měl program čekat, až se skutečně vytiskne nějaký delší text, došlo by v daném bodě programu k nepřijatelnému zdržení. Celý tisk pomocí sady procedur „Display“ se proto uloží do paměti RAM, kde je umístěna virtuální obrazovka displeje automatu. Zároveň s programem aplikace běží na pozadí automatický proces, který z virtuální obrazovky stále čte znaky a posílá je jednotlivě na displej. Tato metoda v zásadě vůbec nezdržuje hlavní program (aplikaci), avšak skrývá jednu záludnost. Chceme-li například smazat řádek displeje a vytisknout na něj novou informaci můžeme to provést např. takto:

```
subroutine novy_radek()
        POSITION = 40          ;budeme pracovat s 2. řádkem
        Display(“                “);tisk mezer přes celý řádek
        POSITION = 40          ;obnovení počátku tisku
        Display(“NOVY TEXT“); na začátek řádku výpis nového textu
return
```

Přepsání celého displeje trvá asi 200ms. Pokud budeme tuto proceduru volat často (alespoň 3x až 4x za sekundu), může se stát, že zrovna po vytisknutí prázdného řádku se několik těchto znaků přenese na displej a bude trvat asi 200ms, než dojde k jejich přepisu na novou hodnotu. **Shodou náhod** může být v tento okamžik obslužný program zase v bodě, kdy do inkriminované oblasti opět napíše mezery..... **Obraz** v takových případech různě **chaoticky poblikává**.

Řešením je buď omezit frekvenci volání takové procedury (max 2x za sekundu), nebo tisknout na displej tak, aby nedocházelo bezprostředně po sobě k přepisu týchž pozic displeje různou informací. Např. takto:

```
subroutine novy_radek()
POSITION = 40                  ;obnovení počátku tisku
Display(“NOVY TEXT     “)     ;na začátek řádku výpis nového textu
return
```

Další metodou, která tento **problém** zcela **řeší**, je **důsledné používání formátů rezervujících určitý prostor pro tisk čísla** (viz. Tab. 8 Formátování tisku číselných hodnot). Tím je zajištěno, že vytištěné číslo zabere na displeji stále stejný počet znaků (tisková funkce jej doplní mezerami) bez ohledu na svou momentální číselnou velikost (pokud ovšem zvolíme dostatečnou rezervu). Pak můžeme jednotlivé textové a číselné elementy naskládat na displej těsně vedle sebe a k žádnému přepisování hodnot a mrkání pak nebude docházet, ani když budeme zobrazování hodnot tisknout v každé programové smyčce.

Příkladem může být program pro zobrazení kusů obrobků:

```
subroutine zobraz_kusy()
        POSITION = 43
        FORMAT = 0x1140     ;rezervují se 4 místa, zarovnání vpravo
        Display("Počet:")           ;text
        Display(kusy)       ;číslo - tisk proměnné kusy
        Display("ks")       ;text
return
```

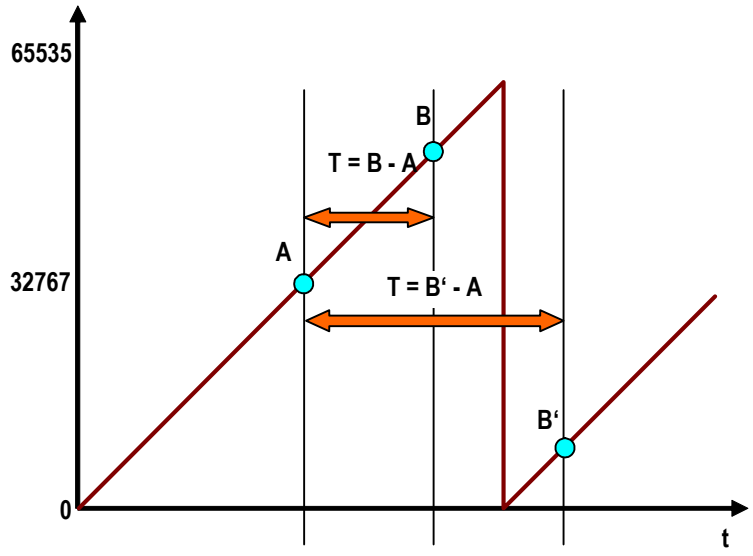
Tento podprogram je možné volat jakkoli rychle, protože všechny položky mají své pevné pořadí a místo, nic se nepřekrývá (tedy až do okamžiku kdy proměnná „kusy" bude větší než 9999).

Na druhou stranu je potřeba počítat s tím, že **tisk na displej** (volání funkce Displej) **zabere poměrně dost strojového času** a příliš mnoho tisků v každém průchodu programové smyčky výrazně zvýší dobu průchodu celým programem a sníží tak celkovou rychlost aplikace.

Omezení počtu tisků na displej

Zásadní důvod pro omezení počtu tisků na displej **představuje** mnohdy **zbytečná časová zátěž procesoru automatu tím, že tiskneme** na displej mnohokrát za sekundu **stejnou informaci a údaj na displeji se nemění**. Časovou zátěž v tomto případě představuje volání formátovacích a tiskových systémových funkcí, které ač jsou optimalizovány, přece jen nějaký čas pro svůj běh potřebují. Omezení počtu tisků na displej můžeme dosáhnout tím, že **výstup na displej budeme spouštět ve zvoleném časovém rastru** např. 2x sekundu. Tato rychlost změn informací na displeji obvykle naprosto vyhovuje. Lidské oko je schopno postřehnout cca. 11 změn za sekundu nemluvě o tom, jak rychle jsme schopni získaný údaj vyhodnotit a na hodnotu zareagovat.

Mějme tedy úlohu blikat na pozici 0 displeje automatu znakem „:" (dvojtečka) v časovém rastru 0.5s. S touto úlohou se můžeme setkat, pokud chceme například tisknout na displej automatu údaj o čase.



Obr. 20 Princip výpočtu pro odměřování času

Řešení úlohy je v zásadě snadné. Jedná se pouze o to, vyřešit **odměřování časového intervalu** a vždy po jeho uplynutí provést výtisk znaku na displej. Pro odměřování časového údaje použijeme jeden z osmice dostupných časovačů. **Přestože je v průběhu časování možné hodnotu časovače měnit** (oblíbené je nulování), **důrazně apelujeme na programátory, aby si tyto metody nechali na nejhorší a hlavně opodstatněné případy a ne na tyto prosté časovací**

úlohy. V našem řešení budeme používat volně běžící časovač a jednoduchou aritmetiku. Výhodou tohoto řešení je to, že volně běžícím časovačem je možné jednoduše časovat téměř „libovolný“ počet dějů a ne celkově 8 (podle počtu časovačů), jak obvykle namítají začínající programátoři. Odměření časového intervalu ukazuje Obr. 20. Zde je v grafu znázorněn průběh hodnoty časovače automatu. Bod A představuje referenční bod a je to vlastně okamžik začátku měření intervalu. Začátek měření je představován buď aktuálním časem (první bod měření) a nebo časem předchozí změny. Abychom mohli bod A použít k vyhodnocení, musíme pro něj vyhradit pomocnou proměnnou. Body B a B‘ představují okamžiky, kdy provádíme test, zda-li uplynula zadaná doba. Aby výpočet doby byl za všech okolností správný a to i tehdy, kdy má v bodě B‘ časovač menší hodnotu než v bodě A, musíme pro vyhodnocení použít takovou **aritmetiku**, která bude **generovat přetečení stejně jako samotný časovač**. Pro výpočet a test použijeme tedy proměnnou typu „word“. Další podmínkou úspěšného řešení úlohy je správné nastavení časovače, který je nutné spustit s povoleným přetečením a v časovém rastru 10ms. Interval 0.5 sekundy pak bude odpovídat rozdílu 50 v hodnotě časovače. Nyní můžeme uvést řešení úlohy.

Řešení:

```
var word bod_A
var byte znak
if reset then
        begin
        TOE[0] = 1 ; povolit přetečení
        TEN[0] = 1 ; spustit časovač
        bod_A = T[0] ; uložit první měření
        znak = ':'      ; nastavení znaku pro tisk
        end
if (T[0] - bod_A) >= 50 then
        begin
        bod_A = T[0]   ; uložit referenční bod
        POSITION = 0 ; nastavit pozici tisku
        FORMAT = 120          ; nastavit formát pro tisk znaku
        Display(znak)  ; vytisknout znak
        if znak = ':' then znak = ' ' else znak = ':' ; nast. nový znak
        end
reset = 0
```

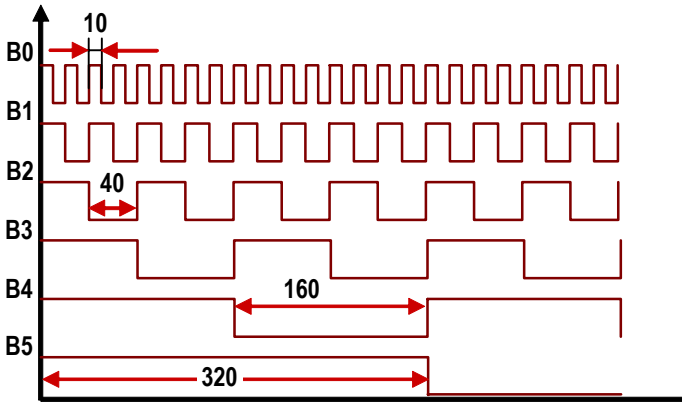
Jak plyne z uvedeného zdrojového textu dochází k přepisu displeje pouze tehdy, když má, tj. ve správném časovém rastru 0.5s.

Světelná signalizace poruchy

Pokud budeme potřebovat řešit světelnou signalizaci poruchy s kvitací poruchy, můžeme postupovat opět velmi elegantně. Předpokládejme, že když porucha nastane světelná signalizace se rozbliká. Pokud v tomto okamžiku stiskneme klávesu „Enter“ blikání se zastaví a světlo zůstane svítit až do doby kdy bude porucha odstraněna. Porucha je hlášena vstupem X[0], světelná signalizace je připojena k výstupu Y[0].

V této úloze jsme opět postaveni před problém **odměření časového intervalu**. Narozdíl od předchozí úlohy není interval zadán a tudíž ho zvolíme **s ohledem na co nejjednodušší kód**. Princip odměřování ukazuje Obr. 21. Hlavní úlohu v odměřování zde hraje opět časovač spuštěný

ve volnoběžném režimu. Hodnota časovače se opět mění každých 10ms. Povšimněme si změn jednotlivých bitů časovače tak, jak nám je ukazuje Obr. 21. Je evidentní, že s každým významnějším bitem časovače počínaje bitem B0, se časový interval mezi změnami bitu zdvojnásobuje. Z toho plyne, že pokud se bit B0 mění každých 10ms, bit B1 už každých 20ms atd. až k bitu B5, který má změny každých 320ms. Tento interval je právě ten, který použijeme k řešení úlohy. Je dostatečně rychlý, aby provokoval činnost a na druhou stranu dostatečně pomalý, abychom blikání jednoznačně odlišili. Za zmíněného předpokladu budeme **blikání** na výstupu Y0 **řešit pouhým kopírováním bitu** 5 časovače.



Obr. 21 Odměřování času pomocí změn bitu

Řešení celé indikace poruchy za tohoto předpokladu bude již pouhým vyjádřením podmínek pro ovládání výstupu Y[0]. Zdrojový text řešení je:

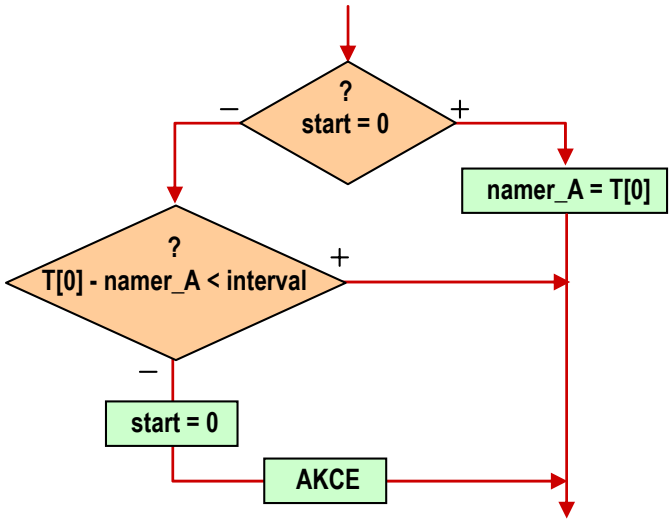
```
var bit kvitace
if reset then
        begin
        TOE[0] = 1 ; povolit přetečení
        TEN[0] = 1 ; spustit časovač
        kvitace = 0  ; uložit počáteční nastavení kvitace poruchy
        end
if X[0] = 0 then
        begin
        ; zde porucha není nastavíme výchozí podmínky
        kvitace = 0  ; uložit počáteční nastavení kvitace poruchy
        Y[0] = 0 ; vypnout signalizaci
        end
else
        begin
        ; zde porucha je, tudíž testujeme zda byla potvrzena
        if (kvitace = 0) then Y[0] = T[0] ? 5 ; blikání
        else  Y[0] = 1 ; porucha kvitována, svítíme trvale
        if (KBCODE = 4) then kvitace = 1 ; test na stisk tlačítka
        end
reset = 0
```

Ze zdrojového textu je vidět značné zjednodušení měření časového intervalu oproti předchozímu příkladu. Dále stojí za povšimnutí, že oba způsoby odměřování je možné použít

současně a to dokonce v řadě instancí. To je proto, že ani jeden způsob měření času nemění obsah časovače a tudíž nemůže ovlivňovat žádné další děje navázané na jeho chod.

Časovač zpoždění

Úloha **generování zpoždění** je jednou z velmi častých úloh odměřování času. Úloha spočívá v tom, že na zadaný povel **pro spuštění zařízení proběhne odčasování zadaného časového úseku** a teprve potom se přenese povel na výstup automatu. Úloha může najít uplatnění při řešení potlačení špičky proudového odběru při spouštění několika motorů současně. Odběrovou špičku pomocí časovačů spuštění tak snadno rozložíme podél časové osy. K realizaci časování využijeme opět volně běžící časovač a logiku spouštění postavíme podle Obr. 22.



Obr. 22 Vývojový diagram časovače zpoždění

Z Obr. 22 je patrné, že máme k dispozici **strukturu časovače**, která obsahuje tři proměnné. Bitovou proměnnou „start“, proměnnou typu word „namer_A“ a proměnnou „interval“ taktéž typu word. První krok řešení zadané úlohy bude vytvořit funkci pro realizaci řídicího algoritmu podle Obr. 22, Parametrem funkce bude proměnná se strukturou časovače, výstupem funkce bude bit, který oznámí, zda se má sepnout výstup automatu nebo ne. Jako spouštěcí vstup celého časování pro trojici motorů bude vstup X[0]. Motory budeme spouštět v časových intervalech 0, 15 a 30 sekund.

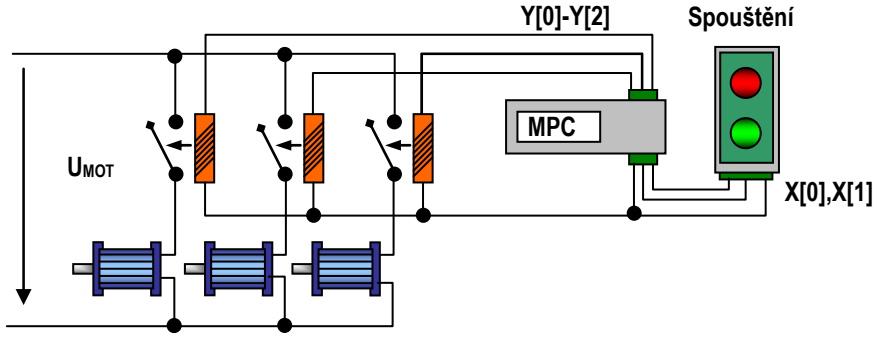
Řešení:

```
function bit spoustec ( var casovac data )
if reset = 1 or data.start = 0 then begin
        ; inicializaci společně po resetu a v klidovém stavu
        data.start = 0 ; inicializace spouštěcího bitu
        data.namer_A = T[0] ; inicializace výchozího času
        exit 0 ;ukončení funkce s neplatnou hodnotou spouštění
        end
if (T[0] - data.namer_A) < data.interval then exit 0 ; čas neuběhl
data.start = 0 ; čas uběhl
return 1 ; spustit motor
```

V další části zdrojového textu uvedeme podobu hlavní programové smyčky.

```
type struct
        bit start,
        word namer_a,
        word interval
end casovac
var casovac motor_1
var casovac motor_2
if (reset) then
        begin
        ; inicializace intervalů spouštění
        motor_1.interval = 1500
        motor_2.interval  = 3000
        ; spuštění časovače T[0]
        TOE[0] = 1
        TEN[0] = 1
        ; inicializace výstupů
        Y[0] = 0
        Y[1] = 0
        Y[2] = 0
        end
Y[0] = X[0] | Y[0] ; výstup přejde do 1 po stisku tlačítka X[0]
motor_1.start = X[0] ; stiskem tlačítka X[0] se zahájí časování
motor_2.start = X[0]
Y[1] = Y[1] | spoustec(motor_1) ; spuštění časování motoru 1
Y[2] = Y[2] | spoustec(motor_2) ; spuštění časování motoru 2
reset = X[1] ; tlačítko vypnutí motorů nastaví bit reset
```

Z uvedeného zdrojového textu je v zásadě patrné využití funkce spouštěč pro realizaci postupného zapínání motorů 0,1 a 2 a stejně tak jejich vypínání a znovu inicializace datových struktur zprostředkovaně přes bit reset nastavovaný pomocí vstupu X[1]. Principielní zapojení popsaného příkladu je na Obr. 23.



Obr. 23 Schématické znázornění zapojení spouštěče motorů

Je nutné poznamenat, že se jedná pouze o demonstrační příklad pro generování zpoždění sepnutí výstupů automatu a ne o skutečnou realizaci řízení motorů. **Podstatou zdrojového textu je ukázka časování několika dějů** (v daném případě dvou) pomocí jednoho volně běžícího časovače.

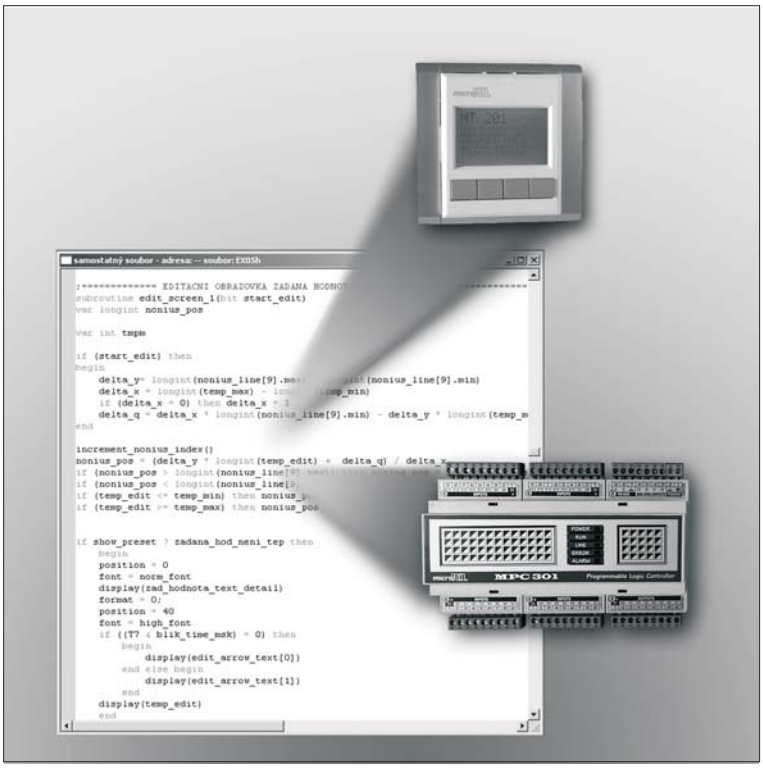
4 Závěr

Příručka „Stručný průvodce jazykem Simple 4“ shrnuje popis základních programových konstrukcí při použití jazyka k programování aplikací pro programovatelné logické automaty MICROPEL řady MPC300, K, terminálu MT201 popř. řad vyšších. Text je základním studijním materiálem pro seznámení se s použitím jazyka v ovládání, vyhodnocování, řízení a zpracování vstupních a výstupních signálů automatů, ovládání klávesnice a displeje a spolupráce automatů přes sdílené proměnné sítě. Text, krom stručného popisu „hardware“ automatů uvedeném v první kapitole a popisu syntaxe základních programových konstrukcí podaném v kapitole 2, obsahuje i několik typických příkladů a základních úloh prezentovaných kapitolou 3.

Rejstřík

Časovač, 6
Časová základna časovače, 6
TPA,TEN,TDM,TOE,TOF, 6
Zjednodušené zapojení, 6
Displej
FONTCTRL, 11
FORMAT, POSITION, 9
Formátování výpisu, 10
Formáty pro MT201, 11
Formáty Simple 2, 10
Přenos obsahu zobrazovací paměti, 9
Tisk na displej, 9
Zobrazovací paměť, 9
Klávesnice
Časování, 7
KBCODE,KBDELAY,KBREPEAT....., 8
Klávesové kódy, 8
Klíčová slova
? - přístup k bitu, 30
+,-,*,/,%,|,&,<<,>>,^, 28
>,<,>=,<=,=,<>, 33
and, 33
begin, 29
bit, 4, 24
byte, 4
case, 34
code, 16, 18
const, 16, 25
default, 34
else, 31
end, 23, 24, 29, 34
endof, 36
exit, 26
fix, 36
float, 4
function, 25
if, 31
int, 4, 24
longint, 4
longword, 4
not, 33
or, 33
return, 26
safe, 4
sizeof, 36
string, 18, 24, 25
struct, 23
subroutine, 25
switch, 34
table, 17, 18, 24
then, 31
type, 22, 23, 24
var, 16, 25
word, 4
Komentáře
Blokový, 27
Řádkový, 12
Konstanty, 4
Číselné konstanty, 5
Programové konstanty, 5
Textové konstanty, 5
Operace
Aritmetické operace, 28
Aritmetický součet, 28
Aritmetický součin, 28
Bitové operace, 27, 30
Bitový posun vlevo, 28
Bitový posun vpravo, 28
Bitový součet, 27, 28, 30
Bitový součin, 27, 28, 30
Negace bitu, 12, 30
Podíl, 28
Přetypování, 35
Přístup k bitu, 30
Rozdíl, 28
Výhradní bitový součet, 28

Zbytek po celočíselném dělení, 28
Podprogramy
Deklarace, 25
Funkce, 4
Podprogramy, 4
Předávání parametrů, 26, 35
Přetěžování parametrů, 26
Tělo, 26
Proměnné, 4
Bezpečné varianty, 4
Deklarace, 16
Deklarace seznamem, 29
Fixace adres, 36
Komunikační proměnné, 6
Pole B, 6
Pole jedno a vícerozměrná, 17, 22
Pole NETLW, NETLI a NETF, 6
Pole W, 6
Řetězce, 18
formátování \, 21
formátování bitově, 21
formátování kódem, 21
Sdílená část, 6
Síťové proměnné, 6, 26
Speciální funkční registry, 6
Struktury, 23
Systémové a uživatelské, 4
Uživatelská jména - symboly, 15
Uživatelské datové typy, 22
Základní typy, 4
Reálný čas
SECOND,MINUTE,HOUR....., 7
Vstupy a výstupy
NC-normally closed, 13
NO-normally open, 13
Ovládání vstupů a výstupů, 12
Vstupy a výstupy, 5
Výrazy, 4
Absolute, 37
Aritmetický výraz, 28, 34
Logický výraz, 33
Makra, 13
Makra a sdílené proměnné, 14
Podmíněný příkaz, 31, 34
Podmínky, 4
Použití maker s negací bitu, 13
Programový přepínač, 34
Přiřazovací příkaz, 12, 29
Složený výraz, 29
Výraz, 29



SIMPLE 4

PRAKTICKÁ PŘÍRUČKA PROGRAMOVÝCH KONSTRUKCÍ JAZYKA SIMPLE 4 PRO ZAČÁTEČNÍKY I POKROČILÉ

edice 7.2008 1.3 verze dokumentu