СТРУКТУРИРАН ТЕКСТ

1. ВЪВЕДЕНИЕ

Езиците за програмиране на ПЛК(PLC) отговарят на стандарта IEC **61131-3**. Но възможностите на ПЛК растат, съответно нарастват и изискванията към приложното програмно осигуряване (ППО). Все по-често се срещат PLC системи, изпълняващи сложни математически операции. В ПЛК се използват web-технологии, размито управление, многомерни интерполаторни системи за управление на движението (SoftMotion) и др. Тези технологии изискват специални познания.

Стандарта МЗК включва езика от високо ниво ST, който не отстъпва по възможности на езиците от високо ниво като С и Паскал. За съжаление в голяма част от системите езикът ST не е реализиран в пълните си възможности. По правило, тези ограничения са броят поддържани типове данни, опростен интерпретиращ транслатор или ограничен код, ограничения на достъпа до апаратните ресурси, невъзможност за обработка на прекъсвания, използване на динамично разпределена памет, управление на процесите и др. Но това не са ограничения на езика ST. Това са ограничения на средата за програмиране.

2. ВЪЗМОЖНОСТИ НА СТРУКТУРИРАНИЯ ЕЗИК

2.1 ОБЩА ИНФОРМАЦИЯ

Структурираният език ST е език от високо ниво предназначен за програмиране на PLC. Той наподобява езиците от високо ниво например PACSAL или C. Елементарната стандартна конструкция на програмата гарантира бързина и ефективност на програмиране. ST използва много елементи които са включени в езиците от високо ниво като променливи, оператори и др. Всеки език има своите преимущества и неудобства. Едно от основните преимущества за ST е неговата способност да опрости математическите уравнения за автоматизираната система.

1.2 Характеристики

Характеристики на ST:

- Език от високо ниво за програмиране;
- Структурирано планиране;
- Елементарен стандарт за конструкция на програмата;
- Бърз и експедитивен начин за програмиране;
- Гъвкавост при употреба;
- Подобен е на PASCAL;
- Хората които са работели на други езици, лесно могат да

се справят и с този език;
- Отговаря на IEC 61131-3 стандарта.

2.3 Възможности

Използвайки ST за програмиране в сред на TwinCat, ние можем да го използваме за:

- Връзка с цифрови и аналогови входни или изходни устройства;
- Реализиране на логически операции;
- Използване на Логически изрази за сравнения;
- Реализиране на аритметични операции;
- Контрол на статуса на различни устройства;
- Реализиране на циклични управляващи алгоритми;
- Използване на функционални блокове в реализираните управляващи алгоритми;
- Разнообразие от динамични променливи;
- Наличие на инструменти за диагностика на софтуера и др;

2. ОСНОВА НА СТРУКТУРИРАНИЯ ТЕКСТ

2 Типове данни

В TwinCat са реализирани всички стандартни типове данни. Всички типове са достъпни в 6-те поддържани езици. Липсват специални разширения за ST.

Целочислени типове: **SINT** (**char**), **USINT** (**unsigned char**), **INT** (**short int**), **UINT** (**unsigned int**), **DINT** (**long**), **UDINT** (**unsigned long**), **LINT** (64 бита), **ULINT** (64 бита без знак).

Реален тип: REAL (float), **LREAL** (double).

Специални типове BYTE, **WORD**, **DWORD**, **LWORD** представлява размер от 8, 16, 32 и 64 бита. В ST няма битови полета. В специалните битови типове може да се обръщаме директно към отделните битове.

Например: **a.3 := 1;** (* Установява бит от променливата a 3 в 1 *).

В C за тази цел се използват целочислени типове данни и побитови логически операции. В TwinCat към битовите редове могат да се приложат операции, достъпни за цели числа.

Логически тип BOOL. Може да приема стойност **TRUE** или **FALSE**. Физическа променлива от типа **BOOL** е с размер от еди бит. Това важи само по отношение на цифровите входове и изходи. По отношение на локалните променливи, размерът им е един байт.

Тип STRING. Представлява редица а не масив. Позволява сравняването и копиране посредством стандартните оператори. Например: strA := strB. В МЗК има стандартен набор функции за работи със редове.

Специални типове са Time (TIME), Time **OF** days(TOD), Calendar Date (DATE) и Time stamp (DT). Времеви променливи се използват много често при програмиране в PLC

Използването на структури (**STRUCT**) не се отличава от C. Описанието на структурата трябва да предхожда използването на променливи от този тип. Допуска се влагане на структури масиви.

Масива (**ARRAY**) се състои от елементи от произволен тип, включително структури и масиви. Позволено е задаването на стойности на масива при деклариранего му. Например: bX **ARRAY**[0..20] **OF BOOL** := **TRUE**, 10(**FALSE**), 9(**TRUE**); Стойността **FALSE** е повторено 10 пъти а **TRUE**, съответно 9 пъти. Масивите (и структурите) може да се копират с помощта на оператора за присвояване := Например: bY := bX;

Предефиниране на типове. Пример: **TYPE** TEMPO: (Adagio := 1, Andante := 2); **END_TYPE**.

На базата на цели числа може да се дефинира тип имащ ограничен диапазон от валидни стойности. Например: **TYPE** DAC10: **INT** (0..16#3FF); **END_TYPE**.

За всеки тип може да се създаде псевдоним (**TYPE**def в C).

Например:

TYPE
DEGR : **UINT**;
END_TYPE.

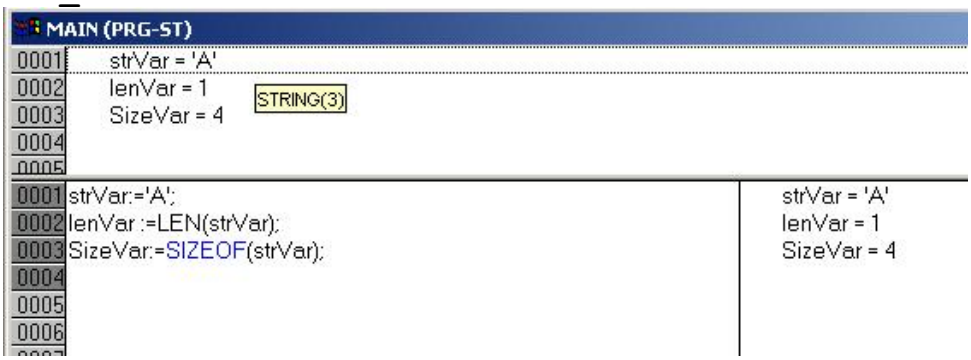
Елементарни типове данни

TYPE	ANY-TYPE	Key WORD	Data width (Bit)	Initial	Value range
BOOLean	ANY_Bit	**BOOL**	1	**FALSE**	**TRUE**/**FALSE**
Bit string(8)		**BYTE**	8	0	0..16#FF
Bit string(16)		**WORD**	16	0	0..16#FFFF
Bit string(32)		**DWORD**	32	0	0..16#FFFF_FFFF
Short integer	ANY_Num	**SINT**	8	0	$-2^7..2^7$
Integer		**INT**	16	0	$-2^{15}..2^{15}$
Double **int**eger		**DINT**	32	0	$-2^{31}..2^{31}$
Unsigned short integer		**USINT**	8	0	$0..2^8$
Unsigned integer		**UINT**	16	0	$0..2^{16}$
Unsigned double **int**eger		**UDINT**	32	0	$0..2^{32}$
Slide po**int**	ANY_REAL	**REAL**	32	0.0	$-1.18*10^{-38}.. 3.4*10^{38}$
Long slide po**int**		**LREAL**	64	0.0	$-2.22*10^{-308}.. 1.798*10^{308}$
Date	ANY_Date	DATE (D)	32	D#1970-01-01	
Time **OF** day		TIME_OF_DAY (TOD)	32	TOD#00:00	TOD#00:00.. TOD#23:59
Date time **OF** day		DATE_AND_TIME (DT)	32	DT#1970-01-01-00:00	
time	ANY_Time	TIME	32	T#0ms	
Sequential **char**acters	ANY_String	STRING	(80+1)*8	''	

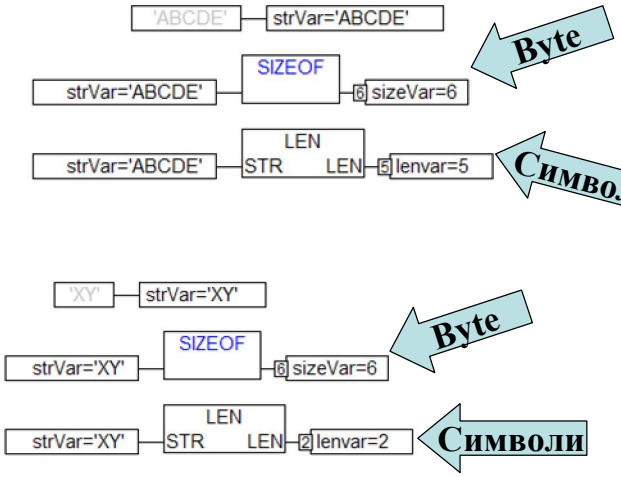
Константи:

Тип	Запис		
BOOL	0,1	16#0, 16#1	**FALSE,TRUE**
BYTE, WORD, DWORD	10	16#0A	2#1010
DWORD, UINT	32768	16#8000	2#1000_0000
INT	-10	---	---
TIME	T#1h2m4s11ms,	T#62m4s11ms,	T#3724011ms
REAL, LREAL	0.22, 2.2e-1 1000, 1000.0, 1e3 (		
STRING	'' empty string, 'constant' 'Text$0D$0A', 'Text$R$N' Special **char**acters		

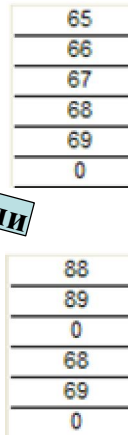
```
VAR
        strVar : STRING(3);
        lenVar : INT;
        sizeVar: INT;
END_VAR
```



Null терминатор, LEN и SIZEOF



SPS memory

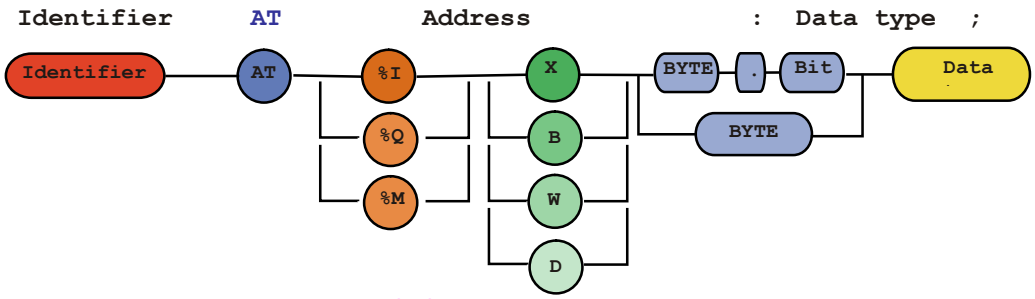



```
VAR
    strVar :STRING(5);
    lenVar: INT;
    sizeVar: INT;
END_VAR
```

Специални символи

character	description
$$	dollar signs
$'	Single quotation mark
$L or $l	Line feed
$N or $n	New line
$P or $p	Page feed
$R or $r	Line break
$T or $t	Tab

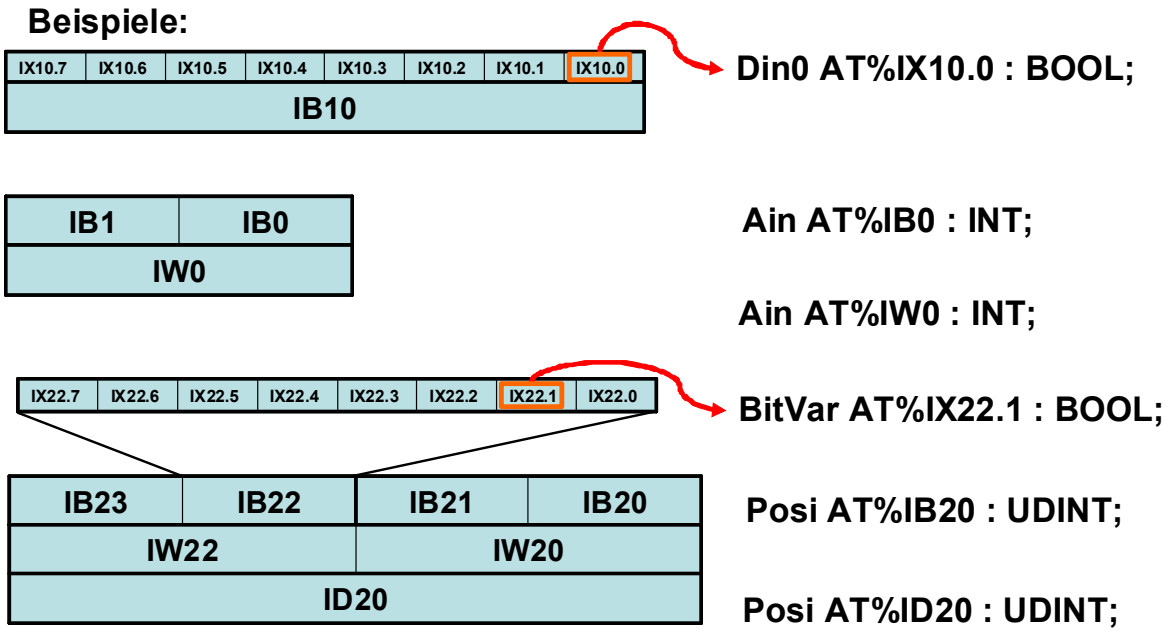
Деклариране на променливи асоциирани към I/O портове



В TwinCAT адресирането може да стане автоматично. След това програмата работи с вътрешната променлива за осъществяване на достъп до входно-изходната променлива.

```
bStellerUntenLinks AT%I*:BOOL:=TRUE;
```

Адреси



Валидност на променливите

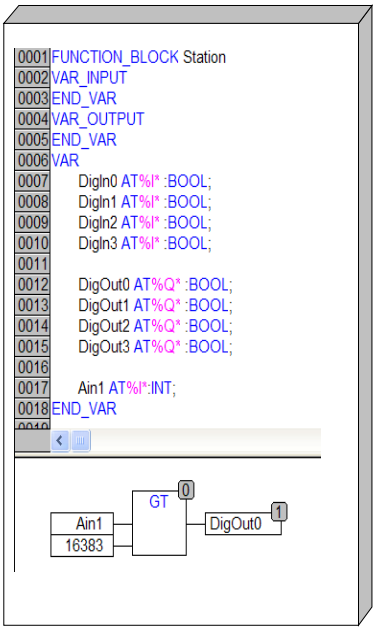
Локални променливи.
Валидни са само в блока в който са декларирани

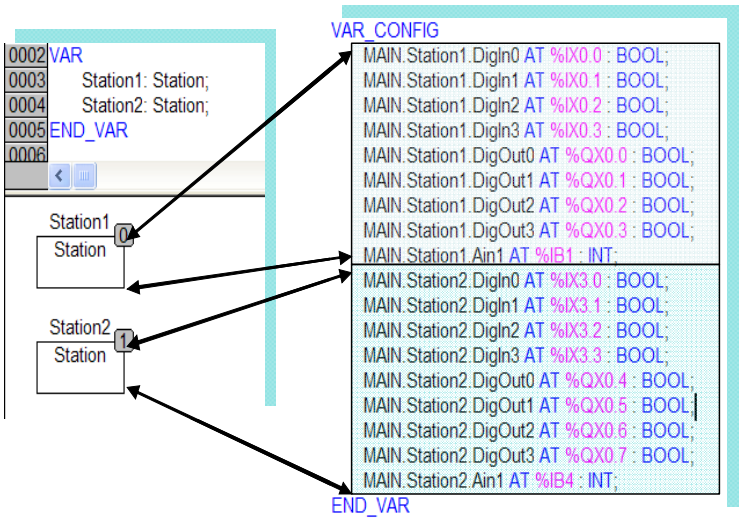
Глобални променливи.
Валидни са в цялата програма

Key WORDs
VAR ..
END_VAR
VAR_INPUT ..
END_VAR
VAR_IN_OUT ..
END_VAR
VAR_OUTPUT ..
END_VAR

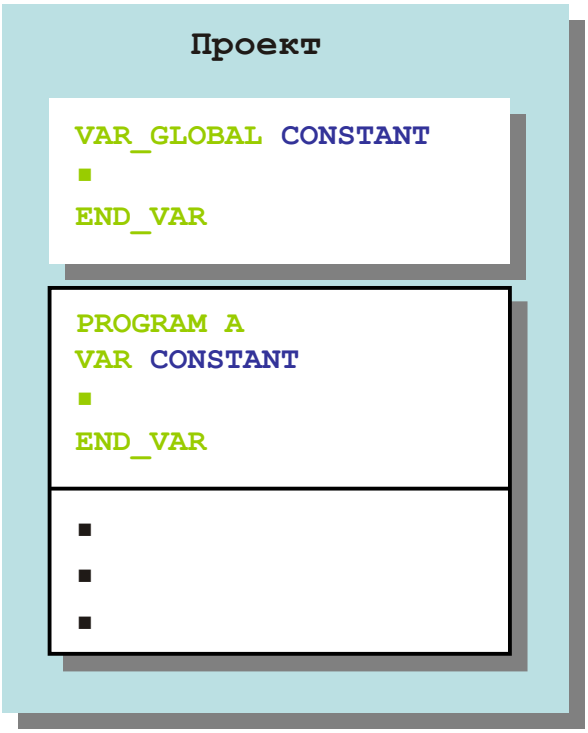
Key WORDs
VAR_GLOBAL ..
END_VAR
VAR_CONFIG ..
END_VAR

Директна връзка на локалните променливи с изходите





Деклариране на константи




```
VAR_GLOBAL CONSTANT
      pi:REAL:=3.141592654;
END_VAR
```

Деклариране на структури

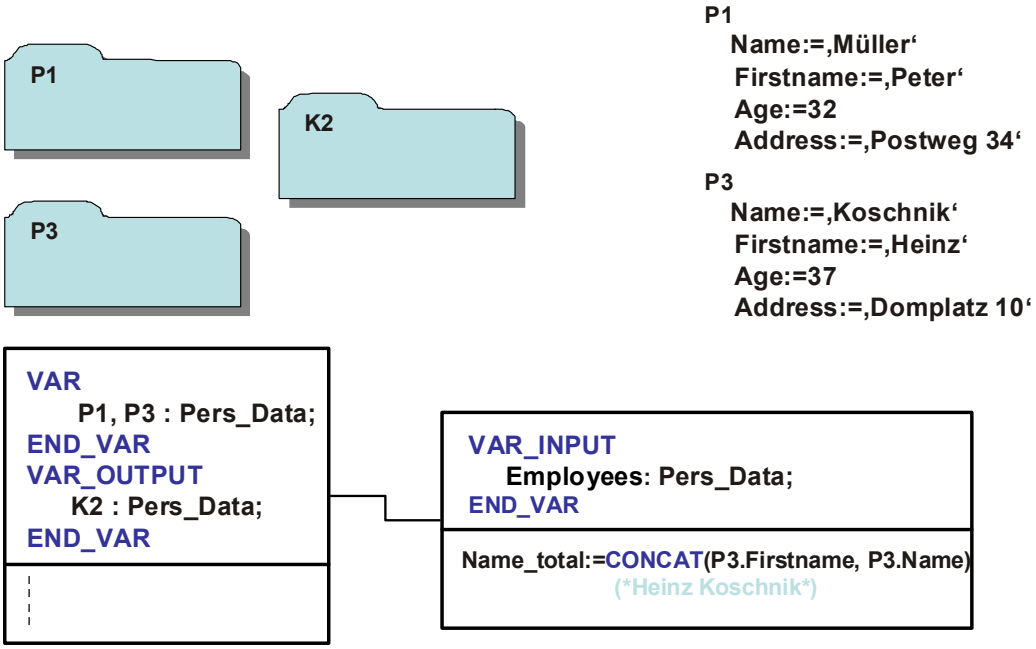
```
TYPE Pers_Data :
STRUCT
    Name: STRING(25);
    Firstname: STRING(25);
    Age:USINT;
    Address: STRING(55);
END_STRUCT
END_TYPE
```

Идентификатор на нов тип данни

Идентификатор : стар тип данни

- ■
- ■
- ■

Използване на структури



Масиви

Деклариране на едномерен и двумерен масив.

```
VAR
     Feld_1 : ARRAY[1..10] OF BYTE;        (* едномерен *)
     Feld_2 : ARRAY[1..10, 2..5] OF UINT; (* двумерен *)
     Feld_3 : ARRAY[1..10, 1..10, 1..10 ] OF DINT;(*тримерен*)
END_VAR
```

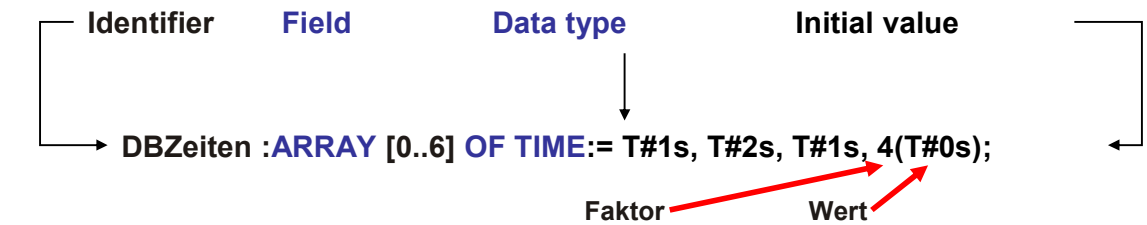
Деклариране на масив сочещ към адреси от реалната памет

```
VAR
   Feld_1 AT %MB100 : ARRAY[1..10] OF BYTE;
END_VAR
```

- **Използване на елементите на масива**

```
     Feld_1[2]   := 120;       (* задаване на стойност *)
     Feld_2[i,j] := EXPT(i,j);(* задаване на стойност *)
```

Първоначална инициализация на масив



0	1	2	3	4	5	6
T#1s	T#2s	T#1s	T#0s	T#0s	T#0s	T#0s

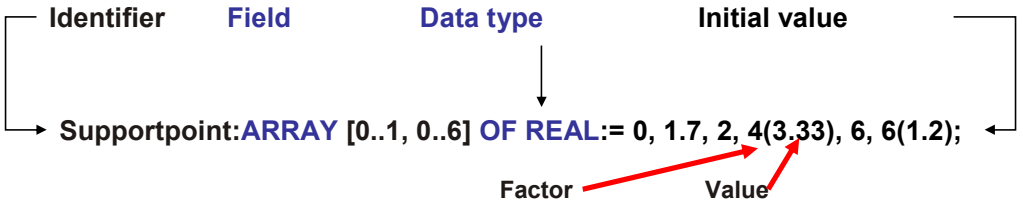
Полето за дължина може да бъде зададено изрично, или с помощта на константи.

Динамичната промяна на размер на полето е невъзможно.

Access:

```
VAR
    WertAusArray : TIME;
END_VAR
WertAusArray := DBZeiten[1];
```

Първоначална инициализация на двумерен масив

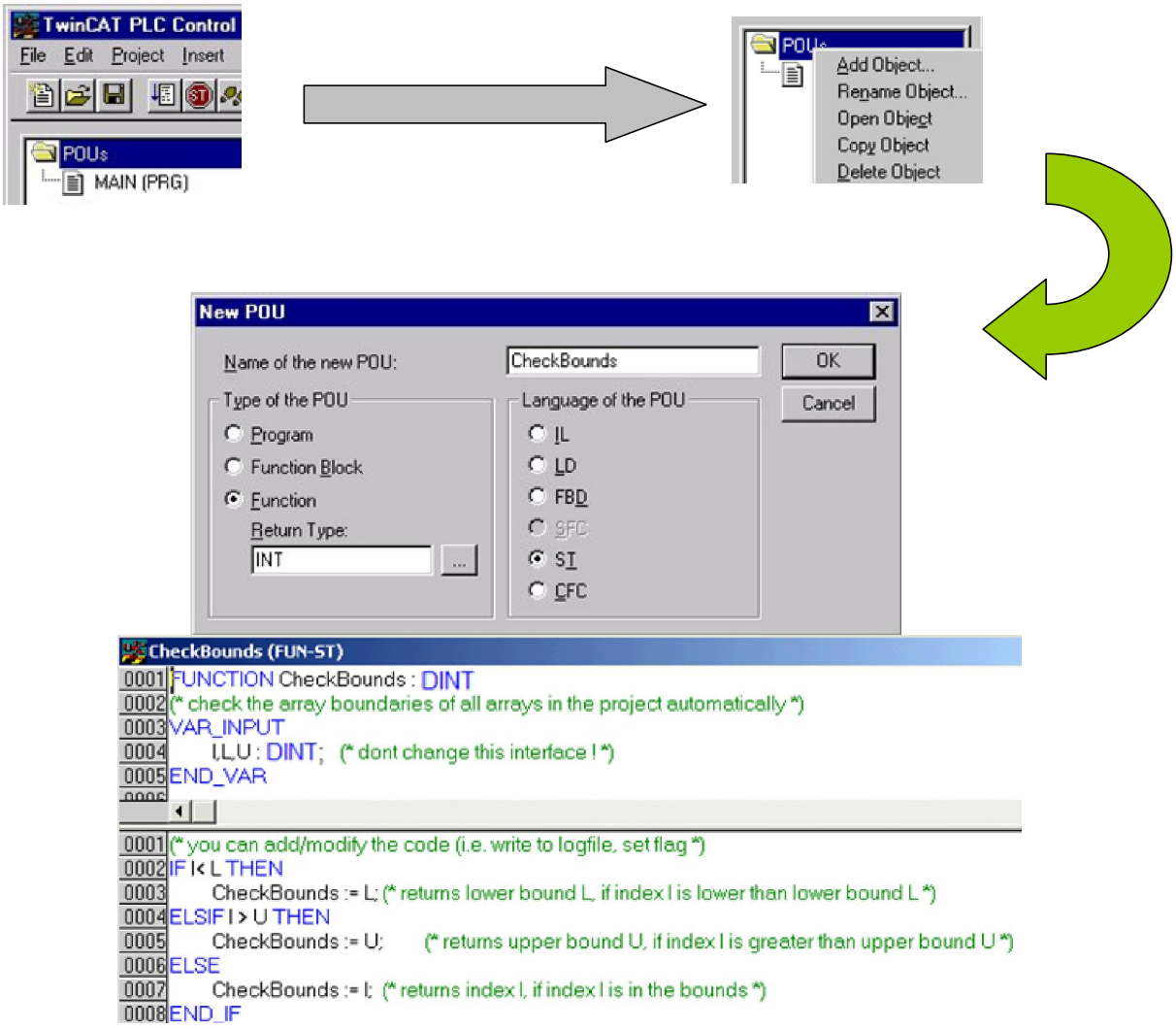


	0	1	2	3	4	5	6
0	0	1.7	2	3.33	3.33	3.33	3.33
1	6	1.2	1.2	1.2	1.2	1.2	1.2

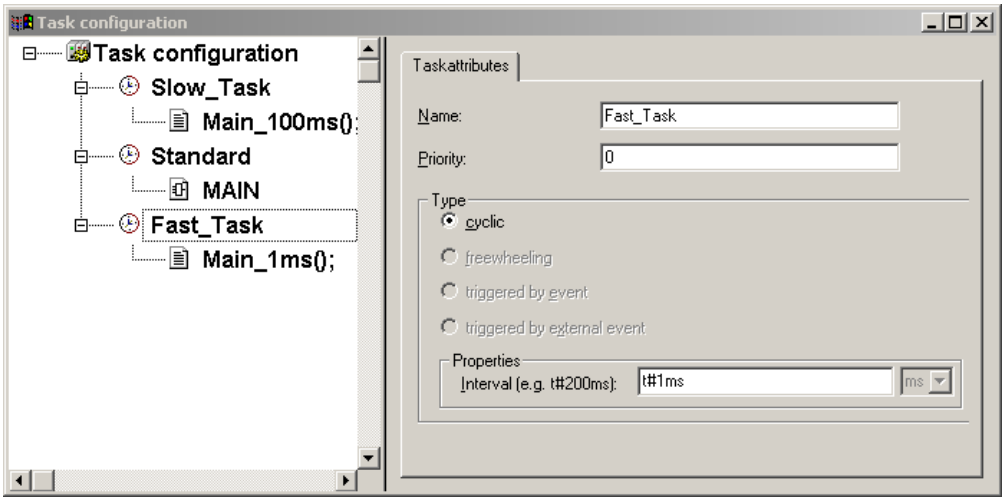
Access:

```
VAR
    WertAusArray : REAL;
END_VAR
WertAusArray := Supportpoint[1 ,0];
```

Създаване на програма



Конфигуриране на работните задачи



3.1 Изрази

Израза е конструкция, която връща стойност, след като той е бил оценен. Изразите са съставени от оператори и операнди. Операнда може да бъде константа, или друг израз.
Пример:

```
b +c
(a-b+c)*COS(b)
SIN(a) *COS(a)
```

3.2 Оператор за присвояване

Операторът за присвояване задава стойността на израза от дясно на променливата от ляво. Той се записва с две точки и равно **:=**. Задължително след израза се поставя **;**. Като операнд от ляво може да бъде зададена променлива, елемент на масив или входно/изходен порт. Не може от ляво на оператора за присвояване да се поставя константа или израз.
Пример:

Пример:

```
Var1 := Var2 + 2; (* Var1 ← (Var2 + 2)*)
```

След като се изпълни операцията Var1 ще съдържа стойността на Var2 увеличена с 2
3.3 Забележка
Въпреки че коментариите не са задължителни, те са изключително важни за подобряване четимостта на програмата. Те играят роля за четливост, яснота и разбираемост на написания код.
Коментарите се поставят между скоби и звезди.
Пример:

```
(* This is one line commentar *)
(* This is
    moree line
    commentar *)
```

3.4 Приоритет на операторите

Използването на няколко оператора в един израз води до въпроса за последователността на изпълнение операциите. За осигуряване на правилната последователност от действия при оценяването на израза се използва приоритетът на операциите. В таблица по-долу е показ този приоритет. При оценяване на изразите първи се изпълняват операциите с по-висок приоритет. За който и да е израз, в който операторът с най-висок приоритет се изпълнява първи, следван от следващия след него оператор по приоритет и т.н. докато не завърши програмата.

Оператори	Символи/Синтаксис	
Обозначения	()	Висок

		приоритет
Функционален блок call Examples	identifier(argument list) LN(A), MAX(X),и др.	
Експонента	**	
Отрицание	**NOT**	
Умножение Деление Модул	* / **MOD**	
Събиране Изваждане	+ -	
Сравнение		
Равенство Неравенство	= <>	
Логическо И	**AND**	
BOOLean exclusive **OR**	**XOR**	
Логическо ИЛИ	**OR**	Нисък приоритет

Пример 1:

```
Result := 5 + 6*3 -12; (* Израз за оценява според приоритете*)
Result := 5 + 18 - 12; (* Операциите ще се изпълнят от ляво на
дясно *)
Result := 23 - 12;     (* Изваждане и край*)
Result := 11;
```

Първо се изпълнява умножението поради най-високият си приоритет. След това операциите се изпълняват от ляво на дясно, като за операнд в следващата операция се използва резултатът от предходната.

Пример 2:

```
Result := (5 + 6)*(3 -12); (* Първо се извъшват операциите в
скобите*)
Result := 30 * -9; (* Умножени е и край на пресмятанията *)
Result := 270;
```

Изразът се чете от ляво на дясната. Операциите в скобите се извършват преди умножението. Както се вижда резултатът от изпълнението на двата примера независимо от еднаквите данни е различен.

4. КОМАНДНИ ГРУПИ

ST съдържа следните команди:

- Булеви логически операции (Boolean logic operations)
- Аритметични операции (Arithmetic operations)
- Операции за сравнение (Comparison operations)
- Управляващи оператори (Decision)
- Оператори за избор (Case statement)

4.1 Булеви логически операции

Не е задължително операндите да бъдат от булев тип.

Таблица: Логически операции

Символ	Логическа операция	Пример
NOT	Binary Negation	a := **NOT** b;
AND	Logical AND	a := b **AND** c;
OR	Logical OR	a := b **OR** c;
XOR	Exclusive OR	a := b **XOR** c;

Таблица на истинност на логическите операции:

Input		AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Операторите могат да бъдат използвани за формиране на изрази, които изискват условие, например **TRUE** или **FALSE.**

Пример 1

```
DoValveSilol:=(DiSilolUp AND (NOT DoValveSilo2) AND
(NOT DoValveSilo3))
```

Пример: 2

```
IF (Level >= MaxLevel) OR (E_Stop = 1) THEN
  Pump := 0;
END_IF
```

4.2 Аритметични операции

Решаващ фактор при вземането на решение за използване на езици от високо ниво е простотата при работа с аритметични операции.

4.2.1 Основни аритметични операции

Symbol	Arithmetic Operations	Example
:=	Присвояване	a := b;
+	Събиране	a := b + c;
-	Изваждане	a := b - c;
*	Умножение	a := b * c;
/	Деление	a := b / c;
MOD	Модулация/Модулиране	a := b mod c;
**		

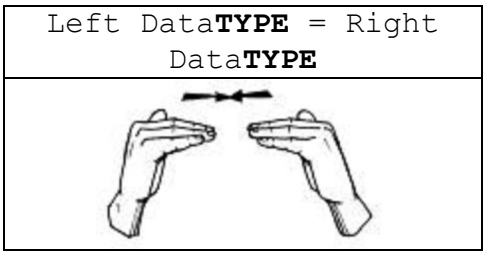
Спазването на конвенцията за типове на операторите е много важно при съставянето на изразите. В следващата таблица са показани възможните преобразования на типове данни.

Синтаксис	Тип на операнда			Резултат
	Res	Op 1	Op 2	
Res := 8 / 3	INT	INT	INT	2
Res := 8 / 3	REAL	INT	INT	2.0
Res := 8.0 / 3	REAL	REAL	INT	2.6667
Res := 8.0 / 3	INT	REAL	INT	*Error

* Състояние на грешка: Не може да се конвертира **REAL** в **INT**

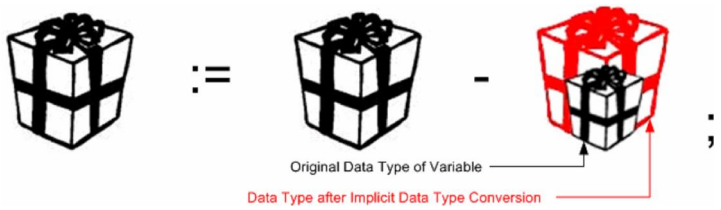
Както се вижда резултатът зависи от видът на данните.

Важно е от ляво на оператора за присвояване да стои операнд от тип с по-голям размер спрямо операнда намиращ се от дясно.



4.2.2 Конвертиране на типовете данни

Data TYPE	BOOL	SINT	INT	DINT	USINT	UINT	UDINT	REAL
BOOL	BOOL	x	x	x	x	x	x	x
SINT	X		INT	DINT	USINT	UINT	UDINT	REAL
INT	X	INT		DINT	INT	UINT	UDINT	REAL
DINT	X	DINT	DINT		DINT	UDINT	UDINT	REAL
USINT	X	USINT	INT	DINT		UINT	UDINT	REAL
UINT	X	UINT	UINT	DINT	UINT		UDINT	REAL
UDINT	X	UDINT	UDINT	UDINT	UDINT	UDINT		REAL
REAL	X	REAL	REAL	REAL	REAL	REAL	REAL	



Ако променливите имат по същия тип размер, те се конвертират в **"unsigned"** тип.

Пример:

```
INT_Result := INT_Varl + SINT_Var2;
  (*[INT]        [INT]       [SINT] *)
```

Променливата SINT_Var2 първо се преобразува в **INT** след което участва в операцията.

4.2.3 Явно преобразуване на типове

Правилата за преобразуване са еднакви като при typecast. Например:

```
INT_Totaltteight:= INTJtfeight1 + INTJtfeightS;
  (*     [INT]          [INT]           [INT] *)
```

Това изглежда добро на първи поглед, но стойността на сумата (INT_Weight1 +INT_Weight2) можеше да бъде по-висока отколкото е размерът на **INT**. За това е необходимо резултатът да се помести в променлива с по-голям размер а преобразуването да стане явно.

```
DINTJTotalWeight:= INT_TO_DINT(INT_Weight1) + INT_tteight2;
[*      [DINT]                  [INT]              [INT] *)
```

```
I     :=BOOL_TO_INT(TRUE);    (* Result is 1 *)
str   :=BOOL_TO_STRING(TRUE);(* Result is 'TRUE' *)
t     :=BOOL_TO_TIME(TRUE);   (* Result is T#1ms *)
tof   :=BOOL_TO_TOD(TRUE);    (* Result is TOD#00:00:00.001 *)
dat   :=BOOL_TO_DATE(FALSE); (* Result is D#1970-01-01 *)
dandt:=BOOL_TO_DT(TRUE); (*Result is DT#1970-01-01-00:00:01 *)
si    :=INT_TO_SINT(4223); (* Result is 127 *)
byt   :=DT_TO_BYTE(DT#1970-01-15-05:05:05); (* Result is 129 *)
str   :=DT_TO_STRING(DT#1998-02-13-14:20);(*'DT#1998-02-13-14:20' *)
vtod  :=DT_TO_TOD    (DT#1998-02-13-14:20);(* TOD#14:20     *)
vdate:=DT_TO_DATE   (DT#1998-02-13-14:20);(* D#1998-02-13 *)
vdw   :=DT_TO_DWORD (DT#1998-02-13-14:20);(* 16#34E45690   *)
```

4.3 Операции за сравнение

Езиците от високо ниво какъвто е и ST използват следните оператори за сравнение

Символ	Предназначение	Пример
=	Равно ли е	IF a = b THEN
<>	Не е равно	IF a <> b THEN
>	По-голямо	IF a > b THEN
>=	По-голямо или равно	IF a >= b THEN
<	По-малко	IF a < b THEN
<=	По-малко или равно	IF a <= b THEN

Резултата от сравнението е **TRUE** или **FALSE**. Те се използват като оператори за отношение в операторите **IF**, **ELSIF**, **WHILE**, **UNTIL** за логически условия.

4.4 Управляващи оператори

Instruction	Example
Присвояване	A:=B; CV := CV + 1; C:=SIN(X);
Извикване на функционален блок	CMD_TMR(IN := %IX5, PT := 300); A:=CMD_TMR.Q;
RETURN	**RETURN**;
IF	**IF** D < 0.0 **THEN** C:=A; **ELSIF** D = 0.0 **THEN** C:=B; **ELSE** C := D; **END_IF**;
CASE	**CASE** INT1 **OF** 1: BOOL1 := **TRUE**; 2: BOOL2 := **TRUE**; **ELSE** BOOL1 := **FALSE**; BOOL2 := **FALSE**; **END_CASE**;

FOR	**FOR** I:=1 **TO (DOWNTO)** 100 **BY** 2 **DO**
	IF ARR[I] = 70 **THEN**
	J:=I;
	EXIT;
	END_IF;
	END_FOR;

WHILE	**WHILE** J<= 100 **AND** ARR[J] <> 70 **DO** J:=J+2;

	END_WHILE;
REPEAT	**REPEAT** J:=J+2; **UNTIL** J= 101 **OR** ARR[J] = 70 **END_REPEAT;**
EXIT	**EXIT;**
Празна инструкция	;

Разклонението на изчислителните процеси в алгоритмите се извършва с помощта на условен оператор. Условен оператор се използва, когато се налага да се направи избор. Изборът може да бъде между две алтернативни възможности или да се избира да се изпълни ли даден оператор или да не се изпълни. В зависимост от вида на избора в език ST се използват три варианта на условен оператор:

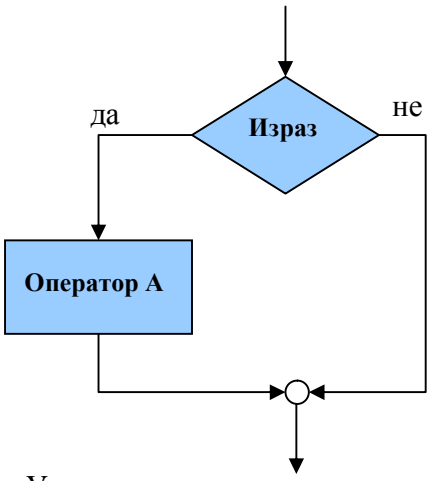
- Условен оператор в непълен формат: **IF**
- Условен оператор в пълен формат: **IF - ELSE**
- Разширен условен оператор: **IF - ELSIF – ELSE**
- Вложени условни оператори

Оператор	Синтаксис	Предназначение
IF THEN	**IF** a **>** b **THEN** Result **:=** 1;	1. Сравнение 1. изпълним блок
ELSIF THEN	**ELSIF** a **>** c **THEN** Result **:=** 2;	2. Сравнение (опционално) 2. изпълним блок
ELSE	**ELSE** Result **:=** 3;	Ако няма верен избор (опционално) 3. Изпълним блок
END_IF	**END_IF**	Край на оператора

4.4.1 Условен оператор в непълен формат IF

Това е най-простата конструкция за **IF**

Може да се представи в логическа форма посредством изречението: **Ако** *Израз* **тогава** *Оператор А;*



Условен оператор в непълен формат

```
(******    Single IF    *********)
IF VI > V2 THEN
    V3 := 99; (* Оператор А *)
END_IF
```

Стойността на логическия израз ***Израз*** определя дали да се изпълни оператора ***Оператор А*** или да не се изпълни. Ако стойността на ***Израз*** е истина (**TRUE**), операторът ***Оператор А*** ще бъде изпълнен. Ако състоянието се оценява за невярно, програмата продължава изпълнението с оператора намиращ се след **END_IF**. Схемата на тази синтактична конструкция е показана на фигурата. Като програмен оператор на ST той има видът:

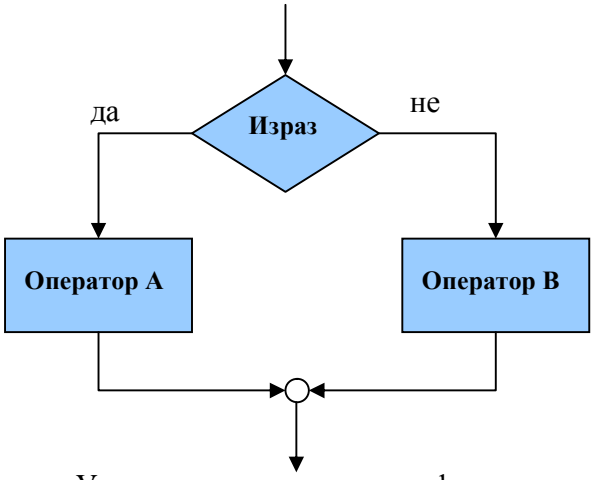
```
IF ( Израз )  THEN
   Оператор А;
END_IF
```

Ако условният израз се оцени като истина, тогава се изпълнява тялото на оператора. Условният израз може да бъде отделен релационен израз или многочислени изрази, които са свързани логично с оператори AND или OR.

4.4.2 Разширен условен оператор IF ELSE.

Този оператор може да се представи в логическа форма посредством изречението:

Ако Израз **тогава** Оператор А **иначе** Оператор В;



Условен оператор в пълен формат

```
(******** IF - ELSE **********)
IF VI > V2    THEN
    V3   := 99; (* Оператор А *)
ELSE
     V8   := 66;  (* Оператор В *)
END_IF
```

Тук ***Израз*** е логически израз (има логическа стойност), ***Оператор А*** и ***Оператор В*** са оператори, които могат да бъдат прости или съставни (блокове). Схемата на тази конструкция е показана на фигурата. Записан като програмен оператор на ST, той има вида:

```
IF Израз THEN
     Оператор А
ELSE
     Оператор В
END_IF
```

Действието на условния оператор в пълен формат е следното: проверява се стойността на логическия израз **Израз;**

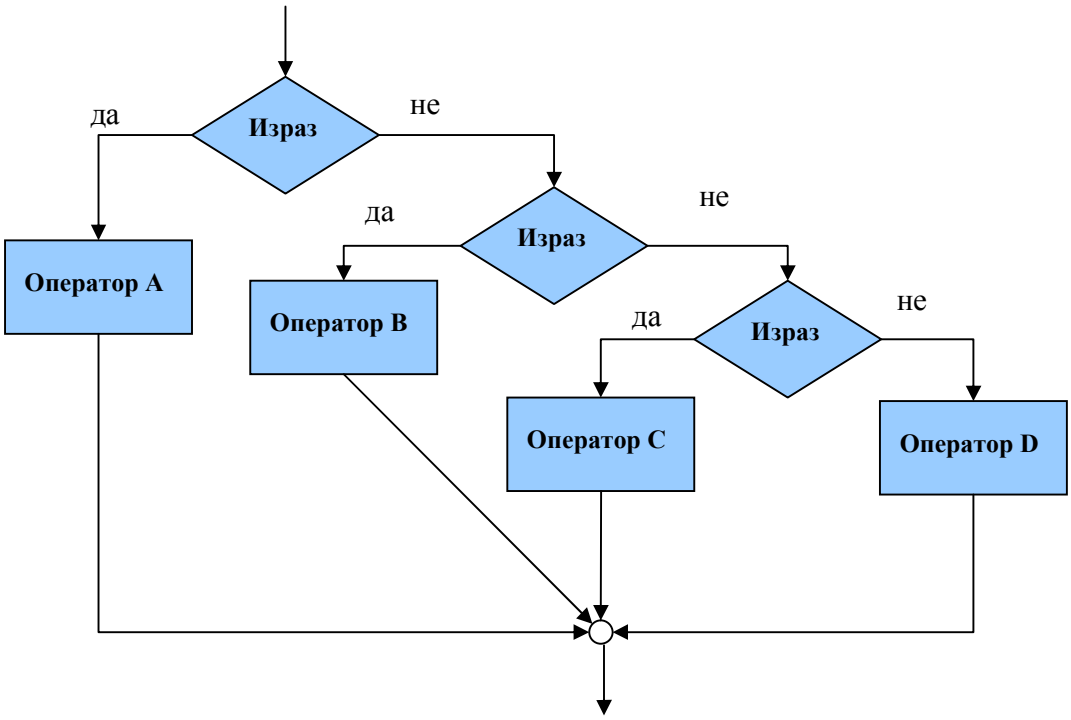
ако стойността е истина (**TRUE**), се изпълнява **Оператор A**, а ако стойността е неистина (**FALSE**) - се изпълнява оператор **Оператор B**.

4.4.3 Разширен условен оператор: IF - ELSIF - ELSE

Тази езикова конструкция се използва когато е необходимо да се реализират алгоритми с повече от две разклонения. Изразите в оператора се оценяват последователно. Изпълнява се този оператор чийто израз първи е оценен като истина. ***Ако има и други изрази оценени като истина, техните оператори не се изпълняват***. Разширен условен оператор може да бъде реализиран и с вложени условни оператори. Синтаксисът на оператора е следният:

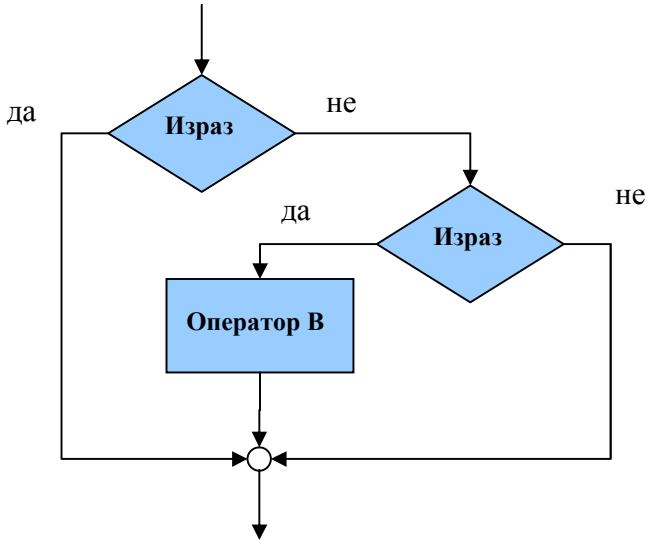
```
IF   Израз 1 THEN
     Оператор A
ELSIF Израз 2  THEN
     Оператор B
ELSIF Израз 3 THEN
     Оператор C
ELSIF Израз 4 THEN
     Оператор D
ELSE
     Оператор E
END_IF
```

```
(**** I F   ELSIF, ELSE *****)

IF VI > V2     THEN
   V3    := 99; (*Оператор A*)
ELSIF VI > V4 THEN
   V5   := 88;  (*Оператор B*)
ELSIF VI > V6 THEN
   V7   := 77;  (*Оператор C*)
ELSE
   V8   := 66;  (*Оператор E*)
END_IF
```



Разширен условен оператор

4.4.4 Вложени условни оператори



Вложен условен оператор

Всички операторите в частта след **then** и след **else** могат да бъдат произволни оператори, включително и други условни оператори. Тогава се получава вложение на условни оператори. Това дава възможност да се направи избор между повече от две възможности. Тъй като всеки от условните оператори може да бъде в пълен или непълен формат, понякога се създават условия за неоднозначно тълкуване. Проблемите, възникващи при вложение на условни оператори, могат да се пояснят с примери:

4.5 Оператор за многопосочно разклонение

В програмирането често се срещат ситуации изискващи многократно повторение на условен оператор. Много от тези ситуации могат да се решат по-бързо и елегантно с оператора **case.**

Ключови думи	Синтаксис	Предназначение
CASE OF	**CASE** ControlVar **OF**	Начало на оператора
	1,5: Display := MAT_TYPE;	При стойност 1 или 5
	2: Display := TEMPERATURE;	При стойност 2
	3,4,6..10: Display := OPERATIONS;	При стойности 3,4,6,7,8,9,10
ELSE	;	
END_CASE	**END_CASE**	Край на оператора

В примера се задава стойност на променливата Display в зависимост от стойността на променливата ControlVar.

Ако стойността на променливата ControlVar е 1 или 5 на Display **се присвоява стойност** MAT_TYPE.

Ако стойността на променливата ControlVar е 2 на Display **се присвоява стойност** TEMPERATURE.

При стойности 3,4,6,7,8,9,10 за ControlVar на Display **се присвоява стойност** OPERATIONS. Записът 6..10 указва, че ControlVar се сравнява с числата от затворения диапазон [6,10].

Ако стойността на ControlVar е друга на Dpisplay не се присвоява стойност.

След изпълнение на операторите за съответната клауза се отива на първия оператор след **END_CASE**

```
(************* CASE *****************)
CASE NumbSelectItem OF
   0: heat := HIGH;       (* Commands A *)
      Fan  := HIGH;
1..3: heat := LOW;        (* Commands B *)
      Fan  := LOW;
 5,6: heat := MEDIUM;     (* Commands C *)
      Fan  := MEDIUM;
ELSE
      heat := OFF;        (* Commands D *)
      Fan  := OFF;
END_CASE
```

4.6 Оператори за цикли

Многократното изпълнение на последователност от инструкции представлява важна алгоритмична концепция. Един от методите за организация на повторение на изпълнението на поредица от команди (който се явява итерационна структура) се нарича **цикъл.** Характерно за циклите е, че в конструкцията им се предвиждат специални инструкции, осигуряващи управлението на

цикъла набор от операции, които се изпълняват многократно. Управляващите инструкциите задават началото и края на цикъла и задават структурата и типа на цикъла, а многократно изпълняваните инструкции образуват тялото на цикъла. Инструкцията за начало на цикъла обикновено се нарича заглавен оператор.

Управлението на цикличните структури обикновено съдържа три операции: **инициализация**, **проверка на условие за край на цикъла и модификация**. При инициализацията **се установява началното състояние**, от което започва изпълнението на цикъла. Самият процес на инициализация представлява задаване на начални стойности на някои величини, които в процеса на изпълнение на цикъла се променят. Операциите за инициализация могат да бъдат вградени в управляващите оператори на цикъла или да бъдат извършени преди началото на цикъла.

Операцията за проверка на условието за край на цикъла е много важна, тъй като **тя определя броя на повторенията на инструкциите от тялото на цикъла**. Когато такова условие не се проверява или е зададено некоректно, цикълът може да се превърне в безкраен. Модификацията е операция, чрез която се **променя състоянието на определени обекти** (стойности на величини), така че ситуацията да се приближава към изпълнението на условието за край на цикъла.

Предназначението на оператора за цикъл е да **организира многократното изпълнение на група команди, които образуват тялото на оператора**. В ST се използват три различни оператора за цикъл. Всеки от тези оператори е подходящ за определени ситуации в алгоритмите, но почти винаги е възможно да се прилага кой да е от тях за конкретен случай. Поради тази причина, програмистите обикновено предпочитат да използват един от операторите за цикъл в повечето случаи и рядко другите оператори.

Операторите за цикъл имат три основни елемента в организацията на програмния процес:

- Задаване на начални стойности на величини, участващи в управлението на цикъла. Те се наричат инициализиращи елементи и в някои случаи могат да се разполагат преди оператора за цикъл;
- Обновяване на стойностите на управляващите променливи. При всяко изпълнение на операторите от тялото на цикъла трябва да се обновява стойността на една или няколко променливи, от които зависи докога се изпълнява цикълът.
- Проверка на условие за край на цикъла. Повторението на изпълнение на операциите от тялото на цикъла се контролира от логическо условие. При всяко изпълнение на операторите от тялото трябва да се проверява това условие.

Има три основни конструкции на цикли

- Цикъл тип **while**(докато е изпълнено условието - повтаряй тялото на цикъла);
- Цикъл тип **repeat until** (повтаряй тялото на цикъла докато е изпълнено условието);
- Цикъл тип **for** (вариант на цикъла **while**).

4.6.1 Цикъл FOR

В много случаи броят на изпълненията на операторите от тялото на цикъл е известен предварително и тогава може да се използва цикъл, управляван с променлива. Той може да се представи с логическата фраза:

за Idnt = S1 **до** S2 **прави** S3

където **Idnt** е идентификатор управляващ цикъла (индекс на цикъла), който представлява проста променлива; **S1** е израз чиято стойност е начално значение на **Idnt**; **S2** - крайна стойност на управляващата променлива и **S3** - оператор (оператори), който представлява тялото на цикъла.

Синтактическата форма на оператора в език ST има вида:

Ключови думи	Синтаксис	Предназначение
FOR TO BY DO	**FOR** i:=StartVal **TO** StopVal {**BY** Step} **DO**	Заглавен блок на цикъла. Секцията заградена в {} е опционална.
	Res := value + 1;	Тяло на цикъла
END_FOR	**END_FOR**	Край на оператора

Пример:

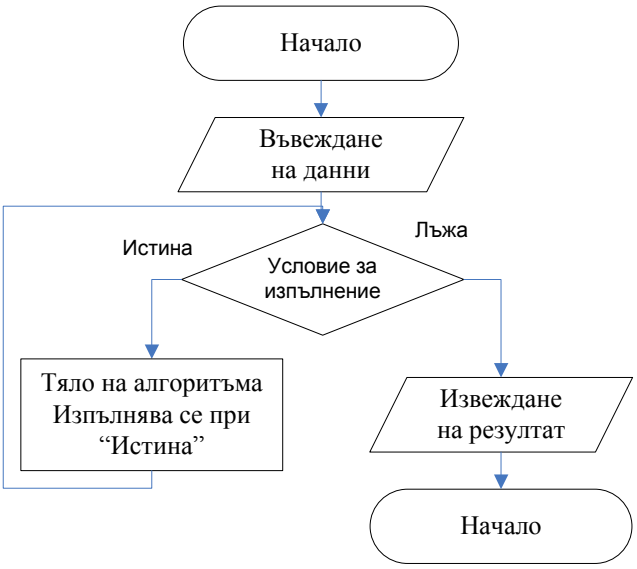
```
(****************** FOR ****************)
FOR counter:=1 TO 5 BY 1 DO
    Var1:=Var1*2; (* Тяло на цикъла *)
END_FOR;
Erg:=Var1;
```

4.6.2 Цикъл с предусловие (while).

Този цикъл може да се представи логически с помощта на изречението:

Докато Израз **повтаряй** Оператор.

Тук **Израз** е логически израз, а **Оператор** е изпълним оператор (група оператори) представляващ тялото на оператора. Схемата на оператора с предусловие има вида:



Цикличен алгоритъм с предусловие

Синтактическата форма на оператора в ST има вида:

Ключови думи	Синтаксис	Предназначение
WHILE Израз DO	**WHILE** i<4 **DO**	Заглавен блок на цикъла.
Оператор	Res := value + 1;	Тяло на цикъла
	I := i + 1;	
END_ WHILE	**END_ WHILE**	Край на оператора

където:

Израз - е израз, който дава като резултат стойност 0 или различна от 0;

Оператор - един оператор или съставен оператор.Това е тялото на цикъла.

В тази конструкция на оператор за цикъл инициализиращата част се намира извън оператора за цикъл (преди оператора). Условието за проверка за продължаване на цикъла се задава с логическия израз **Израз**. Ако стойността на **Израз** е истина, се изпълнява тялото на цикъла **Оператор**, а ако е лъжа цикълът се напуска и управлението се предава на първия оператор, непосредствено следващ оператора за цикъл. След изпълнение на операторите от тялото, управлението се предава в началото на цикъла за нова проверка на условието за продължаване. Цикълът се нарича с предусловие, защото най-напред се проверява условието и ако то е истина, се изпълняват операторите от тялото на оператора. Това означава, че е възможно тялото на цикъла да не се изпълни нито веднъж, ако още при влизането в цикъла условието е неистина. Обновяване на стойността на управляващите променливи трябва да се предвиди в тялото на цикъла.

Пример:

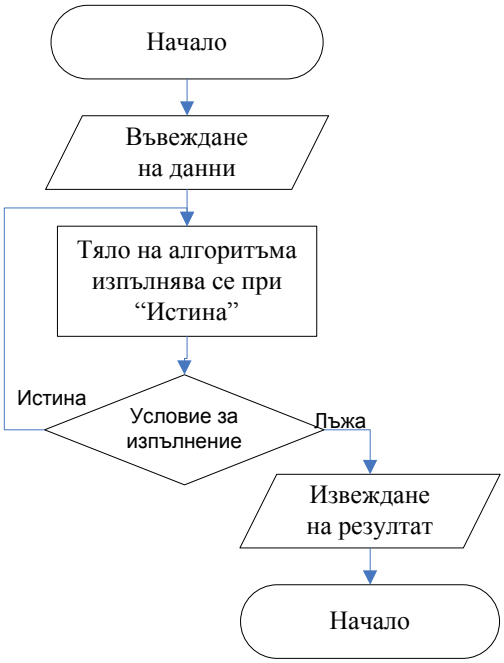
```
(****************** while ****************)
WHILE (indexWhile < EndIndexWhile) DO
    VarWhile   := VarWhile +1;
    indexWhile := VarWhile;
END_WHILE
```

4.6.3 Цикъл с постусловие (repeat - until)

Може да се представи логически с помощта на изречението:

Повтаряй Оператор **докато** Израз.

Израз е логически израз, а **Оператор** е изпълним оператор представляващ тялото на оператора. Алгоритмичната схема на оператора е показана на фигурата.



Цикличен алгоритъм със следусловие

Синтактическата форма на оператора в ST има вида:

Ключови думи	Синтаксис	Предназначение
REPEAT	**REPEAT**	Начало на цикъла
оператор	Res := value + 1;	Тяло на цикъла
	i := i + 1;	
UNTIL Израз	**UNTIL** i > 4	Условие за край
END_ REPEAT	**END_ REPEAT**	Край на оператора

където:

Израз - е израз, който дава като резултат стойност 0 или различна от 0;

оператор - един оператор или съставен оператор. Това е тялото на цикъла.

Тялото на цикъла се изпълнява. Изчислява се **Израз**. Ако резултатът е стойност, различна равна на нула ("неистина"), цикълът се изпълнява отново. Изпълнението на цикъла се прекратява, когато **Израз** получи стойност "истина".

Както се вижда от схемата, проверката на условието за продължаване на цикъла се прави след изпълнение на операторите от тялото на цикъла. Разликата между цикъл тип **while do** и цикъл тип **repeat .. until** е, че при **repeat .. until** тялото на цикъла се изпълнява винаги поне един път, независимо от стойността на **Израз**.

И тук инициализиращата част се намира преди оператора за цикъл, а обновяване на стойността на управляващите променливи се извършва в тялото на цикъла.

Пример:

```
(******************** REPEAT ****************************)
REPEAT
  VarRepeat := VarRepeat + 1; UNTIL VarRepeat > 3
END_REPEAT
```

4.6.4 EXIT

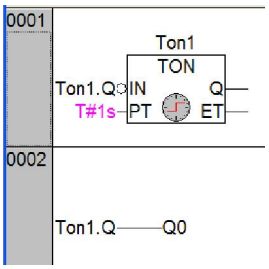
Оператора **EXIT** се използва за преждевременно напускане на цикъл. Действието му е еквивалентно на оператора **break** в C/C++.
Пример:

```
(******************** REPEAT ****************************)
REPEAT
  VarRepeat := VarRepeat + 1;
UNTIL VarRepeat > 3
  IF VarRepeat = VarExit THEN
     EXIT; (*** EXIT LOOP ***)
  END_IF
END_REPEAT
```

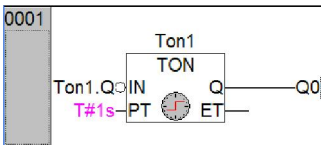
4.6.5. Извикване на функционален блок

Преди извикване на FB, трябва да се зададат необходимите стойности на променливите, използвани като вложени параметри. Функционалния блок се поставя на един ред завършващ с ;.
След извикването на FB се прочетат изходните му параметри.

```
VAR
        TON1:TON;
END_VAR
```




```
TON1 (IN:= NOT TON1.Q , PT:=T#1s );
Q0:= TON1.Q;
```



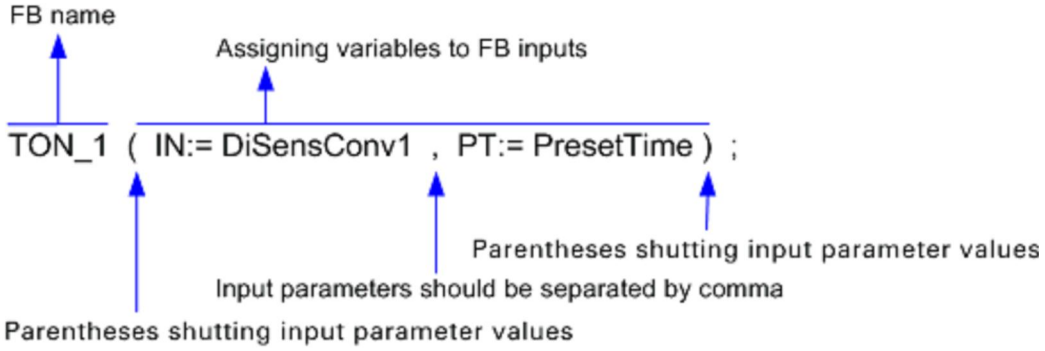

```
TON1(IN:= NOT TON1.Q, PT:=T#1s , Q=>Q0 );
```

1 Задаване на входните параметри

2. Извикване на функционалния блок

3. Прочитане на резултата

```
(***    CALLING FUNCTIONAL BLOCK    ***)
PresetTime := T#3s;
TON_1(IN := DiSensConv1, PT := PresetTime);
DoConv1 := TON_1.Q;
```



4.8 Указатели. Динамични Променливи

ST предлага възможност за работа с указатели. Тяхното използване не е задължително.

Динамични указатели може да се зададе адрес от паметта по време на работа. Тази процедура се нарича препратка или инициализиране. Веднага след като един динамичен указател е инициализиран, чрез него може да се получи достъп до съдържанието на паметта сочена от него.

В следващия пример може да се види, как се използва операторът ADR ()

Той връща адреса на променлива поставени между скоби като **UDINT** стойност. Оператора трябва да завършва с точка и запетая „;".

5. ОБОБЩЕНИЕ

ST е език за програмиране от високо ниво който ви предлага разнообразие в широк кръг. ST съдържа всички компоненти както на другите езици за програмиране, но предимство пред другите е неговата простота. По-лесно е да се учи, от ANSI C, и е по-ефективно, отколкото Ladder Diagram **OR I**nstruction List.

Ако преминете през това ръководство, да направите упражненията, вие сте в състояние да реализирате вашата задача в ST. Не се страхувайте, имате ли знанието, трябва само да го да го реализирате.

Успешно приложение на ST !

6. ДОПЪЛНЕНИЕ

6.1 КЛЮЧОВИ ДУМИ

Всички ключови думи в ST се показват в син цвят от редактора.

Кл. Думи	Описание
ACCESS	Referencing OR dynamic variable.
BITCLR	A := BIT_CLR(IN, POS) A contains the value IN after the bit at position POS has been deleted. However, the IN operAND remains unchanged.
BITSET	A := BIT_SET(IN, POS) A contains the value IN after the bit at position POS has been set. However, the IN

	operand remains unchanged.
BITTST	Determines the value **OF** a bit: A := BIT_TST(IN, POS) A contains the value **OF** the bit at position POS **OF** operAND IN.
BY	See FOR Statement.
CASE	See CASE Statement.
DO	See WHILE Statement.
EDGE	Detects positive AND negative edges.
EDGENEG	Detects negative edges.
EDGEPOS	Detects positive edges.
ELSE	See IF Statement.
ELSIF	See IF Statement.
END_CASE	See CASE Statement.
END_FOR	See FOR Statement.
END_IF	See IF Statement.
END_REPEAT	See REPEAT Statement.
END_WHILE	See WHILE Statement.
EXIT	See EXIT Statement.
FOR	See FOR Statement.
IF	See IF Statement.
REPEAT	See REPEAT Statement.
RETURN	This instruction can be used to end a function, depending on a condition fOR example.
THEN	See IF Statement.
TO	See FOR Statement.
UNTIL	See REPEAT Statement.
WHILE	See WHILE Statement.

6.2 Оператори

Операторите са функции на ST, които не изискват включване на всички библиотеки в проекта ви.

Оператор	Описание
ABS	Връща абсолютната стойност на числото ABS(-2) резултат 2.
ACOS	Returns the arc cosine on a number (inverse function **OF** cosine).
adr	Връща адреса на променливата
AND	Побитово ит.
ASIN	Връща аркус синус
ASR	Arithmetic shifting **OF** an operand to the right: A := ASR (IN, N) IN is shifted N bits to the right, the left is filled with the sign bit.
ATAN	Връща аркус тангенс

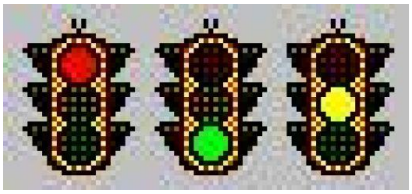
COS	Връща cos
EXP	Експонента: A := EXP (IN).
EXPT	One operand raised to the power **OF** another operand: A := EXPT (IN1, IN2).
LIMIT	Limitation: A = LIMIT (MIN, IN, MAX) MIN is the lower limit, MAX is the upper limit for the result. If IN is less than MIN, then the MIN result is returned. If IN is greater than MAX, then the MAX result is returned. Otherwise, the IN result is returned.
LN	Returns the natural logarithm **OF** a number.
LOG	Returns the base-10 logarithm **OF** a number.
MAX	Maximum function. Returns the larger **OF** two values.
MIN	Minimum function. Returns the lesser **OF** two values.
MOD	Modulo division **OF** a **USINT**, S**INT**, **INT**, **UINT**, UD**INT**, **DINT TYPE** variable by another variable **OF** one **OF** these **TYPE**s.
MOVE	The contents **OF** the input variable are copied to the output variable. The := symbol is used as the assignment operator. "A := B;" is the same as "A := MOVE (B);"
MUX	Selection: A := MUX (CHOICE, IN1, IN2, ... INX); CHOICE specifies which operator (IN1, IN2, ... INX) should be returned as the result.
NOT	Negation **OF** a bit operand by bit.
OR	Logical OR operation by bit.
ROL	Rotates an operand's bits to the left: A := ROL(IN, N) The bits in IN are shifted N times to the left, the far left bit being pushed in again from the right.
ROR	Rotates an operand's bits to the right: A := ROR(IN, N) IN's bit are shifted N times to the right, the far right bit being pushed in again from the left.
SEL	Binary selection: A := SEL (CHOICE, IN1, IN2) CHOICE must be **TYPE BOOL**. If CHOICE is **FALSE**, then IN1 is returned. Otherwise, IN2 is returned.
SHL	Shifts an operand's bits to the left: A := SHL (IN, N) IN is shifted N bits to the left, the right side is filled with zeroes.
SHR	Shifts an operand's bits to the right: A := SHR (IN, N) IN is shifted N bits to the right, the left side is filled with zeroes.
SIN	Returns the sine **OF** a number.
Sizeof	This function returns the number **OF BYTE**s required by the specified variable.
SQRT	Returns the square root **OF** a number.
TAN	Returns the tangent **OF** a number.
TRUNC	Returns the **int**eger part **OF** a number.

XOR	Logical EXCLUSIVE OR operation by bit.

Пример:

Този пример демонстрира използването на ST езика за реализиране управлението на светофарна уредба

State-Model (ST) Case Statement



В инициализационния сектор се описват всички декларации на използваните променливи в програмата. Това е декларационната секция:

```
    TYPE
        Lights : STRUCT
(* Structure OF lights in each direction *)
            Red_Stop    : BOOL;
            Yellow_Slow : BOOL;
            Green_Go    : BOOL;
            Green_Arrow : BOOL;
            Walk        : BOOL;
            Dont_Walk   : BOOL;
        END_STRUCT
    END_TYPE

    VAR_GLOBAL
(* Global variables hosting Stop Light tags/symbols *)
        North : Lights;
        South : Lights;
        East  : Lights;
        West  : Lights;
    END_VAR
```

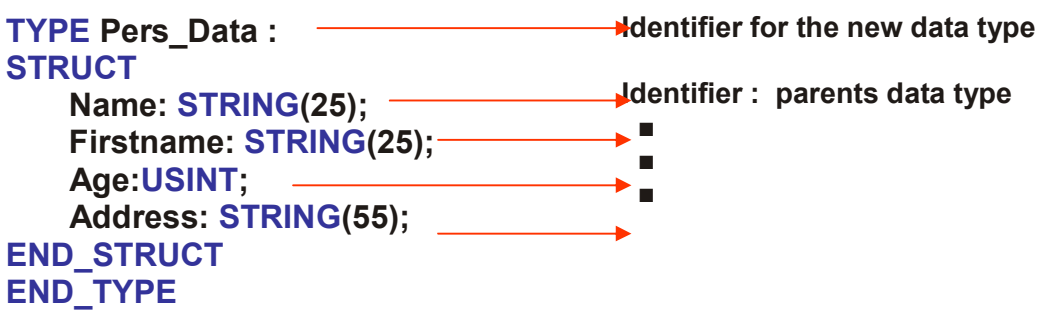
Във функциите и програмата се използват се използват UDT (User Defined Tag) структури и променливи.

```
    FUNCTION ClearLights : VOID
        VAR_IN_OUT
            LightSet : Lights;
        END_VAR

            LightSet.Red_Stop      := FALSE;
            LightSet.Yellow_Slow   := FALSE;
            LightSet.Green_Go      := FALSE;
            LightSet.Green_Arrow   := FALSE;
```

```
            LightSet.Walk          := FALSE;
            LightSet.Dont_Walk     := FALSE;
    END_FUNCTION

    FUNCTION ClearAllLights : VOID
            ClearLights(lightset := North);
            ClearLights(lightset := South);
            ClearLights(lightset := East);
            ClearLights(lightset := West);
    END_FUNCTION

PROGRAM StopLights
    VAR     (* Локални променливи за логиката на програмата*)
        State       : INT;
        GreenTimer  : TON;
        GreenTime   : TIME  := T#6s;
        YellowTimer : TON;
        YellowTime  : TIME  := T#2s;
        StateUpdate : BOOL;
        StateMem    : INT;
    END_VAR

    (* Таймер определящ времето за светене на зеленото *)
    GreenTimer( IN:= State= 0 OR State = 2,PT := GreenTime);

    (* Таймер определящ времето за светене на жълто *)
   YellowTimer(IN:= State=1 OR State = 3, PT := YellowTime);

    IF State <> StateMem THEN
        StateMem := State;
        StateUpdate := TRUE;
    ELSE;
        StateUpdate := FALSE;
    END_IF;

    CASE State OF
        0: (* Север & Юг са зелени а  East & West Червени *)
            IF StateUpdate THEN
                ClearAllLights();
                  (* Спиране на всички светлини *)
                East.Red_Stop   := TRUE;
                West.Red_Stop   := TRUE;
                North.Green_Go  := TRUE;
                South.Green_Go  := TRUE;
            END_IF;
            IF GreenTimer.Q THEN
                North.Green_Go      := FALSE;
                South.Green_Go      := FALSE;
                State := 1;
            END_IF;
        1:
      (* Север & Юг Жълто докато Изток & Запад са червени *)
                North.yellow_slow   := TRUE;
                South.Yellow_Slow   := TRUE;
                IF YellowTimer.Q THEN
                   ClearAllLights();
                      (* Спиране на всички светлини *)
                   State := 2;
```

```
            END_IF;
        2: (* East & West - зелено / Север & Юг – Червено *)
            IF StateUpdate THEN
                North.Red_Stop  := TRUE;
                South.Red_Stop  := TRUE;
                East.Green_Go   := TRUE;
                West.Green_Go   := TRUE;
            END_IF;
            IF GreenTimer.Q THEN
                East.Green_Go      := FALSE;
                West.Green_Go      := FALSE;
                State := 3;
            END_IF;
       3:  (* East & West Yellow a Север & Юг са Червени *)
                East_yellow_slow   := TRUE;
                West_Yellow_Slow   := TRUE;
                IF YellowTimer.Q THEN
                   State := 0;
                END_IF;
    ELSE;
    END_CASE;
END_PROGRAM
```

```
Syntax:
TYPE <Bezeichner>:(<Enum_0> ,<Enum_1>, ...,<Enum_n>);
END_TYPE
Beispiel:
TYPE Woche:(Mo, Di, Mi, Dn, Fr, Sa, So:=10);(*Mo = 0 Di = 1..
                                    .. Sa = 6 So = 10*)
END_TYPE
TYPE Richtung:(Up, Dn);(*Up = 0 Dn = 1*)
END_TYPE
```



Syntax:

TYPE <Bezeichner>:(<Enum_0> ,<Enum_1>, ...,<Enum_n>);
END_TYPE

Beispiel:

TYPE Woche:(Mo, Di, Mi, Dn, Fr, Sa, So:=10);(*Mo = 0 Di = 1..
.. Sa = 6 So = 10*)
END_TYPE

TYPE Richtung:(Up, Dn);(*Up = 0 Dn = 1*)
END_TYPE