Structured Text (ST)

TM246

Perfection in Automation

www.br-automation.com

Requirements

Training modules:	TM210 – The Basics of Automation Studio
	TM211 – Automation Studio Online Communication
	TM213 – Automation Runtime
	TM223 – Automation Studio Diagnostics
Software:	None
Hardware:	None

Table of contents

1. INTRODUCTION

Structured Text is a high level language. For those who are comfortable programming in Basic, PASCAL or Ansi C, learning Structured Text is simple. Structured Text (ST) has standard constructs that are easy to understand, and is a fast and efficient way to program in the automation industry.

Fig. 1 Book printing: then and now

The following chapters will introduce you to the use of commands, key words, and syntax in Structured Text. Simple examples will give you a chance to use these functions and more easily understand them.

1.1 Objectives

Participants will get to know the programming language Structured Text (ST) for programming techincal applications.

You will learn the individual command groups and how they work together.

You will get an overview of the reserved keywords in ST.

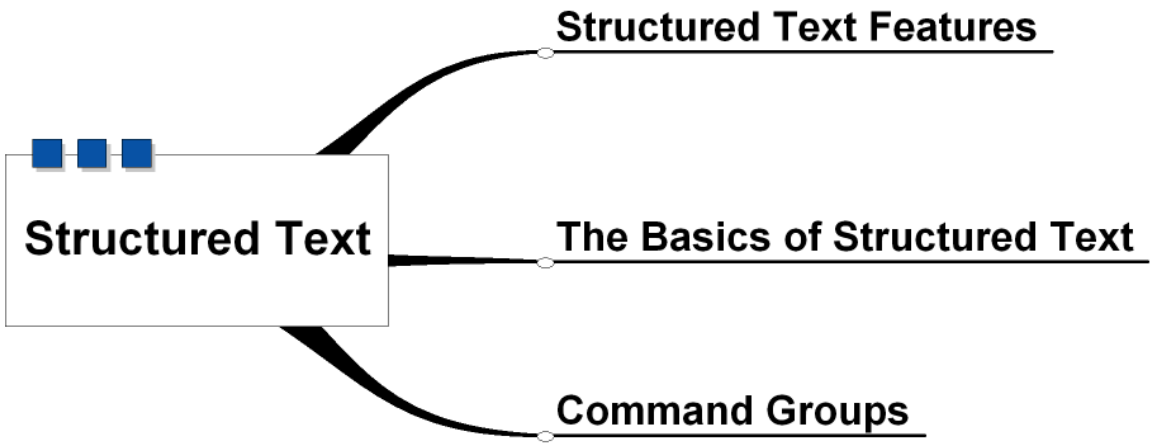


Fig. 2 Overview

2. STRUCTURED TEXT FEATURES

2.1 General information

ST is a text-based high-level language for programming automation systems. Simple standard constructs enable fast and efficient programming. ST uses many traditional qualities of high level languages, including variables, operators, functions, and elements for controlling the program flow.

So, what is Structured Text? The "Structured" refers to the qualities of a high level language that make structured programming possible. The "Text" refers to the ability to use text in place of symbols, as in the ladder diagram.

No other programming language can replace ST. Every programming language has its advantages and disadvantages. The main advantage of ST is that complex mathematical calculations can be programmed easily.

2.2 Properties

Structured Text is characterized by the following features:

- High-level text language
- Structured programming
- Easy to use standard constructs
- Fast and efficient programming
- Self explanatory and flexible use
- Similar to PASCAL
- Easy to use for people with experience in PC programming languages
- Conforms to the IEC 61131-3 standard

2.3 Possibilities

Automation Studio supports the following functions:

- Digital and analog inputs and outputs
- Logical operation
- Logical comparison expressions
- Arithmetic operations
- Decisions
- Step sequencers
- Loops
- Function blocks
- Optional use of dynamic variables
- Diagnostic tools

3. THE BASICS OF STRUCTURED TEXT

3.1 Expressions

An expression is a construct that returns a value after it has been evaluated. Expressions are composed of operators and operands. An operand can be a constant, a variable, a function call or another expression.

Example: Expressions

```
b + c

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

SIN(a) * COS(b)
```

Fig. 3 Expressions

3.2 Assignment

The assignment of a value to a variable through a result of an expression or a value. The assignment consists of a variable on the left side, which is designated to the result of a calculation on the right side by the assignment operator ":=". All assignments must be closed with a semicolon ";".

Example: Assignment

```
Var1 := Var2 * 2; (*  Var1 <-- (Var2 * 2) *)
```

Fig. 4 Assignment

When the code line has been processed, the value of variable "Var1" is twice as big as the value of variable "Var2".

3.3 Comments

Comments are sometimes left out, but are nevertheless an important component of the source code. They describe the code and make it more easy to read. Comments make it possible for you or others to read a program easily, even long after it was written. They are not compiled and have no influence over the execution of the program. Comments must be placed between a pair of parenthesis and asterix "(*comment*)".

Example: Comment

```
(* This is one line comment *)
```

Fig. 5 One-line comment

```
(* This
is more
lines
comment *)
```

Fig. 6 Multi-line comment

3.4 Operator priorities

The use of several operators in one line brings up the question of priority (order of execution). The execution is determined by priority.

Expressions are executed starting with the operator of highest priority, followed by the next highest, and so on until the expression has been completely executed. Operators with the same priority are executed from left to right as they appear in the expression.

Operator	Symbol / Syntax:	
Parentheses	()	Highest priority
Function call Examples	Call argument(s) LN(A), MAX(X), etc.	
Exponent	**	
Negation	NOT	
Multiplication Division Modulo division (whole number remainder of division)	* / MOD	
Addition Subtraction	+ -	
Comparisons	<, >, <=, >=	
Equal to Not equal to	= <>	
Boolean AND	AND	
Boolean exclusive OR	XOR	
Boolean OR	OR	Lowest priority

The order of execution at runtime:

Example: Operator priorities without parentheses

```
Result := 6 + 7 * 5 - 3;     (*The multiplication first; higher precendence *)
Result := 6 +   35  - 3;     (*The addition; rule from left to right *)
Result := 41 - 3;            (*Substraction at the end *)
Result := 38;
```

Fig. 7 Order of execution

Multiplication is executed first, then addition, and finally subtraction.

The order of operations can be changed by putting higher priority operations in parentheses. This is shown in the next example.

Example: Operator priorities with parentheses

As shown in the following figure, the use of parentheses influences the execution of the expression.

```
Result := (6 + 7) * (5 - 3);(*operations inside the parentheses first *)
Result :=    13   *    2;   (*then the multiplication *)
Result :=    26;
```

Fig. 8 Order of execution

The expression is executed from left to right. The operations in parentheses are executed first, then the multiplication, since the parentheses have higher priority. You can see that the parentheses lead to a different result.

4. COMMAND GROUPS

ST has the following command groups:

- Boolean operations
- Arithmetic operations
- Comparison operations
- Decisions
- Case statements

4.1 Boolean operations

The operands must not necessarily be the data type BOOL.

Boolean operations:

Symbol	Logical operation	Examples
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;`

Truth table:

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

These operators can be used to formulate logical expressions, or they can be used to represent conditions. The result is always TRUE (logical 1) or FALSE (logical 0).

Example: Boolean operation

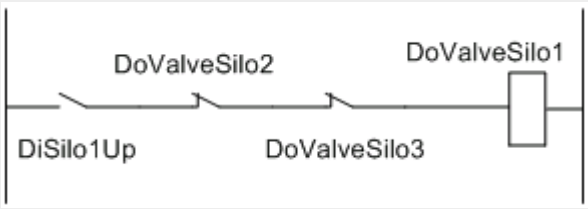


Fig. 9 AND operation

```
DoValveSilo1 := (DiSilo1Up AND (NOT DoValveSilo2) AND (NOT DoValveSilo3));
```

Fig. 10 Source code for AND operation

ST allows any number of parenthesis.

Task: Light control

The output "DoLight" should be ON when the button "BtnLigntOn" is pressed. It should remain ON until the button "BtnLightOff" is pressed.

Create a solution for this task using boolean operations.

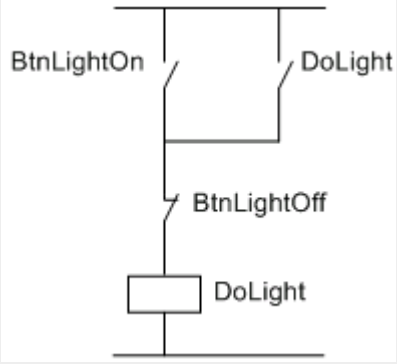


Fig. 12 Light control

4.2 Arithmetic operations

A key factor in favor of using a high level language is the accessibility of arithmetic operations.

4.2.1 Basic arithmetic operations

ST provides basic arithmetic operations for your operation:

Symbol	Arithmetic operation	Example
:=	Assignment	a := b;
+	Addition	a := b + c;
-	Subtraction	a := b - c;
*	Multiplication	a := b * c;
/	Division	a := b / c;
MOD	Modulo (display division remainder)	a := b mod c;

The data type is a very important factor. Note the following table:

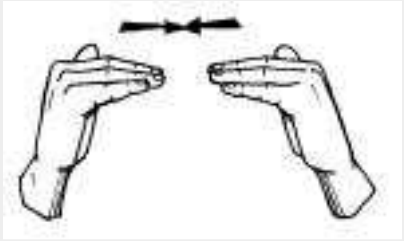
Syntax	Data types			Result
	Res	Op 1	Op 2	
Res := 8 / 3;	INT	INT	INT	2
Res := 8 / 3;	REAL	INT	INT	2.0
Res := 8.0 / 3;	REAL	REAL	INT	2.6667
Res := 8.0 / 3;	INT	REAL	INT	Error

* Compiler error message: **Type mismatch: Cannot convert REAL to INT.**

You can see the that the result is dependent of the syntax as well as the data types used.

Note:

Left data type := right data type;



4.2.2 Implicit data type conversion

This type of conversion is done by the compiler. It automatically converts the smaller data types to the larger one used in the expression. If an expression contains one or more operators with different data types, they are all converted to the same data type before the expression is resolved.

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	

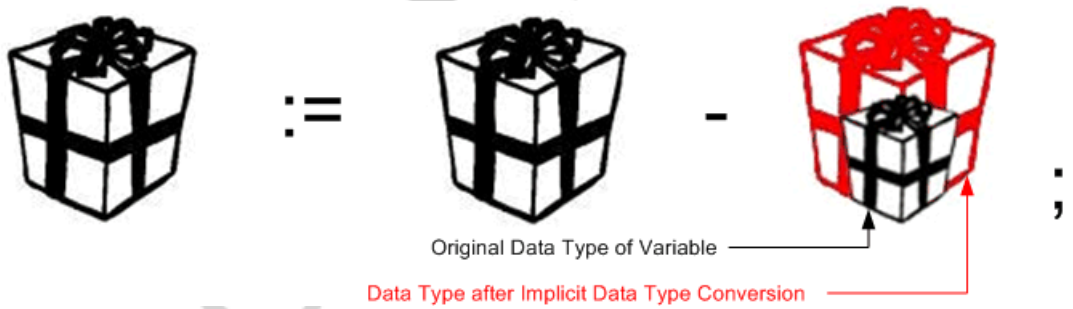


Fig. 13 Implicit data type conversion by the compiler

Example: Data conversion

```
INT_Result := INT_Var1 + SINT_Var2
(* [INT]       [INT]     [SINT] *)
```

Fig. 14 Implicit data type conversion

SINT_Var2 is converted to INT, then added, then assigned to the result variable (INT_Result).

4.2.3 Explicit data type conversion

Explicit data type conversion is also known simply as type conversion or as Typecast. As you already know, the expression should have the same data type on both sides, but there is something else to remember.

Example: Overflow?!

```
INT_TotalWeight := INT_Weight1 + INT_Weight2
(*   [INT]              [INT]          [INT]  *)
```

At first sight, everything looks OK. However, the sum (INT_Weight1 + INT_Weight2) can be larger than can be stored as data type INT. In this case, an explicit data type conversion must be carried out.

Example: Overflow taken into consideration.

```
DINT_TotalWeight := INT_TO_DINT(INT_Weight1) + INT_Weight2
(*   [DINT]                          [INT]          [INT]  *)
```

The variable DINT_TotalWeight must be the data type DINT. At least one variable on the right side of the expression must be converted to the data type DINT. The conversions functions are found in the OPERATOR library.

Task: Aquarium

The temperature of an aquarium is measured at two different places. Create a program that calculates the average temperature and displays it at an analog output.

Don't forget that analog inputs and outputs must be data type INT.

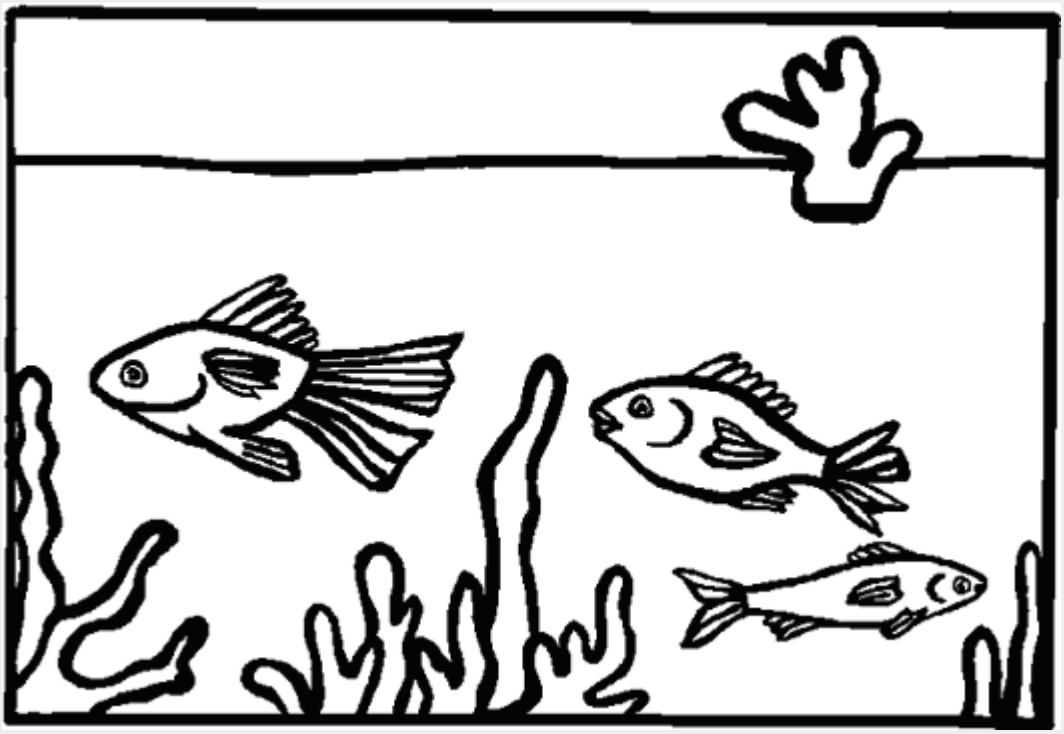


Fig. 15 Aquarium

4.3 Comparison operators

In high level languages like ST, simple constructs can be used to compare variables. These return either the value TRUE or FALSE.

Symbol	Logical comparison expression	Example
=	Equal to	IF a = b THEN
<>	Not equal to	IF a <> b THEN
>	Greater than	IF a > b THEN
>=	Greater than or equal to	IF a >= b THEN
<	Less than	IF a < b THEN
<=	Less than or equal to	IF a <= b THEN

Note:

The comparison operations and logical operations are mainly used as logical conditions for IF, ELSEIF, WHILE and UNTIL statements.
(IF (a > b) AND (d >= e) THEN)

4.4 Decisions

The IF statement is used to create decisions in the program. You are already familiar with the comparison operators, and they can be used here. There are several types of IF statements:

- Simple IF statement
- IF – ELSE statement
- IF – ELSIF statement
- Nested IF

Decision	Syntax	Description
IF THEN	IF a > b THEN	1. Comparison
	Result := 1;	1. Statement(s)
ELSIF THEN	ELSIF a > c THEN	2. Comparison (optional)
	Result := 2;	2. Statement(s)
ELSE	ELSE	Above IF statments are not TRUE (optional)
	Result := 3;	3. Statement(s)
END_IF	END_IF	End of decision

4.4.1 IF

This is the most simple IF statment.

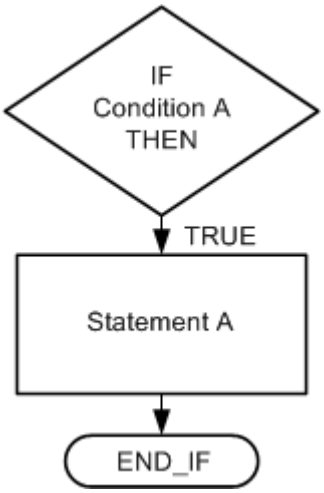


Fig. 16 Simple IF statement

```
(*******************************)
(******   Single   IF    ********)
(*******************************)
IF V1 > V2 THEN
  V3 := 99;          (* comment *)
END_IF
(*******************************)
```

Fig. 17 Simple IF statement in a program

The IF statement is tested for the result TRUE. If the result is FALSE, the program advances to the line after the END_IF statement. The function of the IF statement can be a single comparison, but it can also be multiple comparisons connected by AND, OR, etc..

Example: IF statement with multiple comparisons

```
IF ( ((Userlevel > 10) OR (diKeySwitch = TRUE)) AND (operationMode = 0) ) THEN
    LedEdit:= TRUE;
END_IF
```

Fig. 18 Statement with multiple comparisons

4.4.2 ELSE

The ELSE statement is an extension of the simple IF statement. Only one ELSE statment can be used per IF statement.

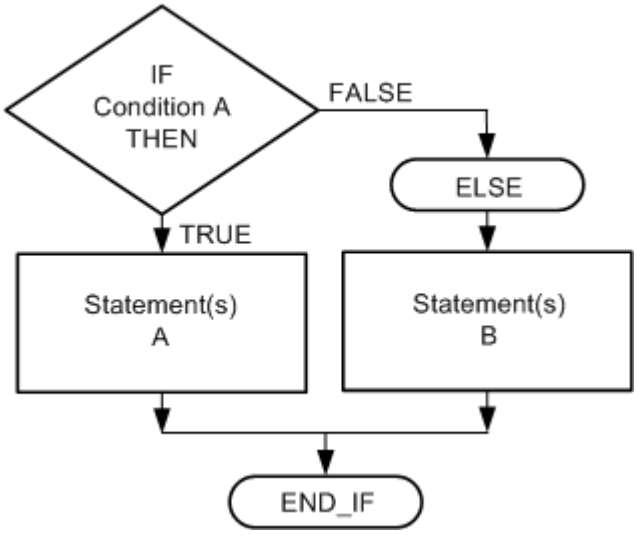


Fig. 19 IF – ELSE statment

```
(*******************************)
(******     IF ELSE      ********)
(*******************************)
IF V1 > V2 THEN
  V3 := 99;          (* comment *)
ELSE
  V4 := 66;          (* comment *)
END_IF
(*******************************)
```

Fig. 20 IF – ELSE in a program

If the IF meets condition A, then the A instruction(s) are executed. If the IF does not meet condition A, then the B instruction(s) are executed.

4.4.3 ELSIF

One or more ELSE_IF statements allow you to test a number of conditions without creating a confusing software structure with many simple IF statements.

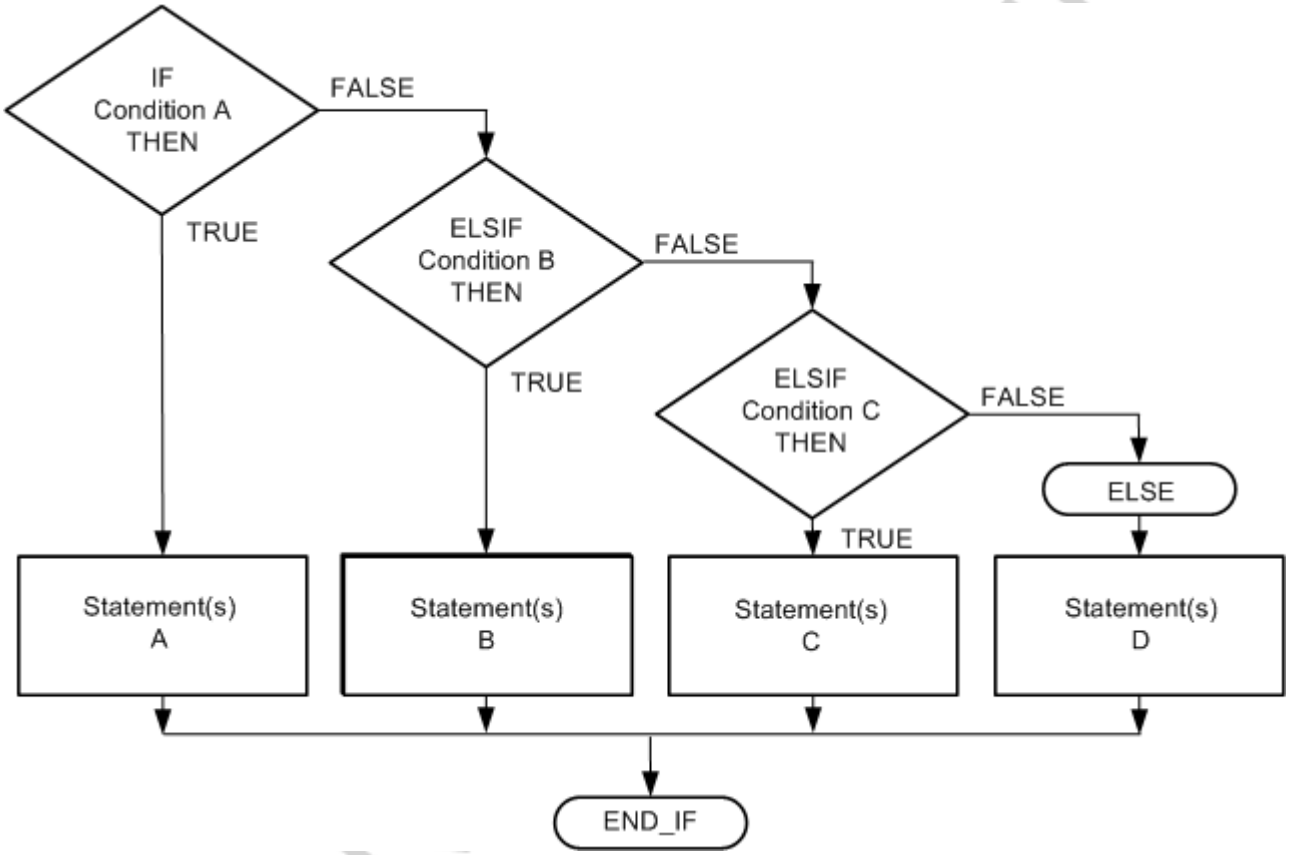


Fig. 21 IF-ELSIF-ELSE statement

```
(*******************************)
(****** IF, ELSIF, ELSE *******)
(*******************************)
IF V1 > V2 THEN
  V3 := 99;          (* comment *)
ELSIF V1 > V4 THEN
  V5 := 88;          (* comment *)
ELSIF V1 > V6 THEN
  V7 := 77;          (* comment *)
ELSE
  V8 := 66;          (* comment *)
END_IF
(*******************************)
```

Fig. 22 IF-ELSIF-ELSE in a program

At runtime the decisions are processed from top to bottom. If the result of a decision is TRUE, the corresponding statements are executed. Then the program continues from after the END_IF. Only those decisions that

correspond to the first TRUE decision are executed, even if subsequent decisions are also TRUE. If none of the IF or ELSE_IF decisions is TRUE, the statement in the ELSE branch is executed.

Task: Weather station - Part I

A temperature sensor measures the outside temperature. The temperature is read via an analog input (1°=10), and should be displayed inside the house in text form.

- If the temperature is under 18°C, the display should read "Cold".
- If the temperature is between 18°C and 25°C, the display should read "Opt".
- If the temperature is over 25°C, the display should read "Hot".

Create a solution using IF, ELSEIF, and ELSE statements.

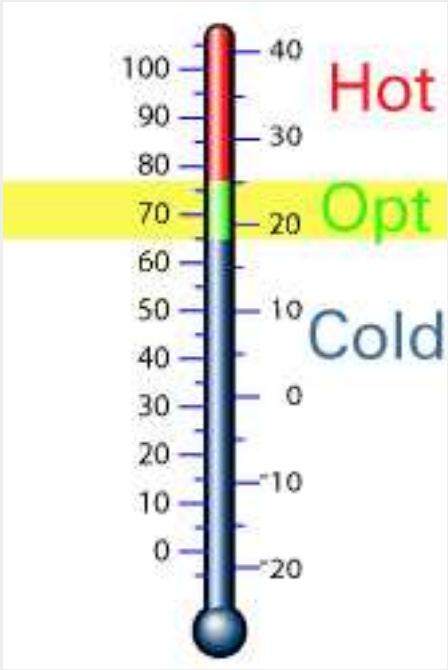


Fig. 23 Thermometer

Note:

In ST, text is assigned to a string variable as follows:

```
StringVar := 'COLD'
```

4.4.4 Nested IF statement

A nested IF statement is tested only if previous condidions have been met. Every IF requires its own END_IF so that the order of conditions is correct.

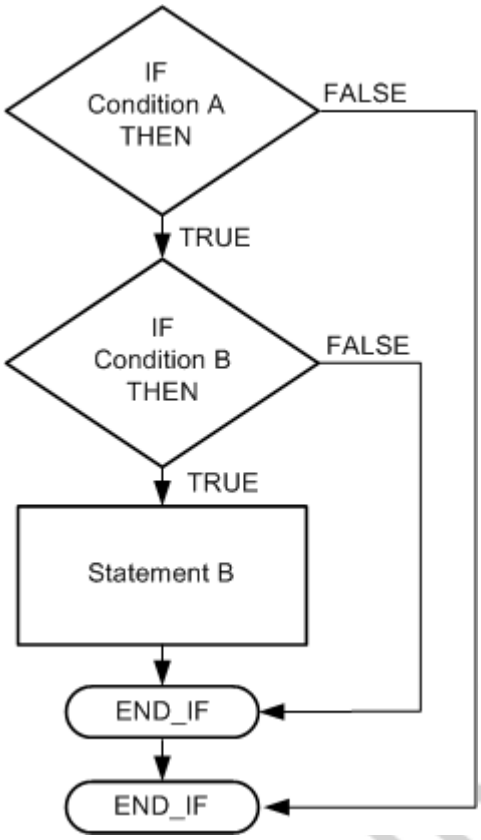


Fig. 24 Nested IF statement

```
(*********************************)
(******    Nested  IF     ********)
(*********************************)
IF V1 > V2 THEN
  IF V2 > V4 THEN
    V3 := 99;          (* comment *)
  END_IF
END_IF
(*********************************)
```

Fig. 25 Nested IF statement in a program

It is helpful to indent every nested IF statement and the corresponding expressions. As many IF statments can be nested as needed. It is possible, however, that the compiler will run out of memory after the 40th level. Also, that kind of extensive nesting is evidence of poor programming style. It becomes nearly impossible to get a clear overview of the code.

After three nesting levels, it is better to find another way to structure the program.

Task: Weather station - Part II

Evaluate the temperature and the humidity.
The text "OPT" should only appear when the humidity is between 40 and 75% and the temperature is between 18 and 25°C. Otherwise "Temp. OK" should be displayed.

Solve this task using a nested IF statement.

Two simple IF statements produce nearly the same effect as one nested IF statement. A marker variable, or flag, can be requested in multiple statements. The first IF statment describes the flag, which is then utilized by other IF statements.

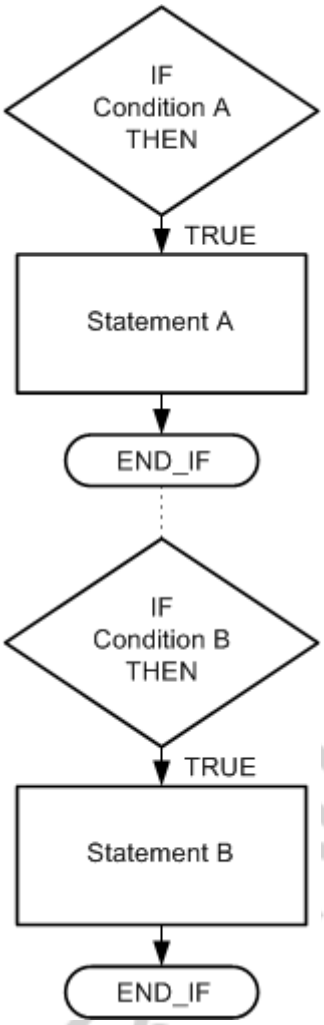


Fig. 26 Two IF statements

```
(********************)
(* 1. IF Statement *)
(********************)
IF V1 > V2 THEN
  V3 := 99;
END_IF
(********************)
(********************)
(* 2. IF Statement *)
(********************)
IF V3 = 99 THEN
  Burner := ON;
END_IF
(********************)
```

Fig. 27 Two IF statements

In this case, both IF statements have the same priority. If both IF statements evaluate the same variable for different values, you should use a CASE statement.

The CASE statement should be used when:

- IF constructs require too many levels
- Too many ELSE_IF statments are used

The CASE statement is much easier to read in these cases.

In comparison to the IF statment, the CASE statment also has the advantage that comparisons are only made once, which makes the program code more effective.

4.5 Case statements

The CASE statement compares a step variable with multiple values. If one of these comparisons is a match, the steps that compare to that step are executed. If none of the comparisons is a match, there is an ELSE branch similar to an IF statement that is then executed.

After the statements have been executed, the program continues from after the END_CASE statement.

Keywords	Syntax	Description
CASE OF	CASE step variable OF	Beginning of CASE
	1,5: Display := MATERIAL	For 1 and 5
	2: Display := TEMP	For 2
	3,4,6..10: Display := OPERATION	For 3,4,6,7,8,9,10
END_CASE	END_CASE	End of CASE

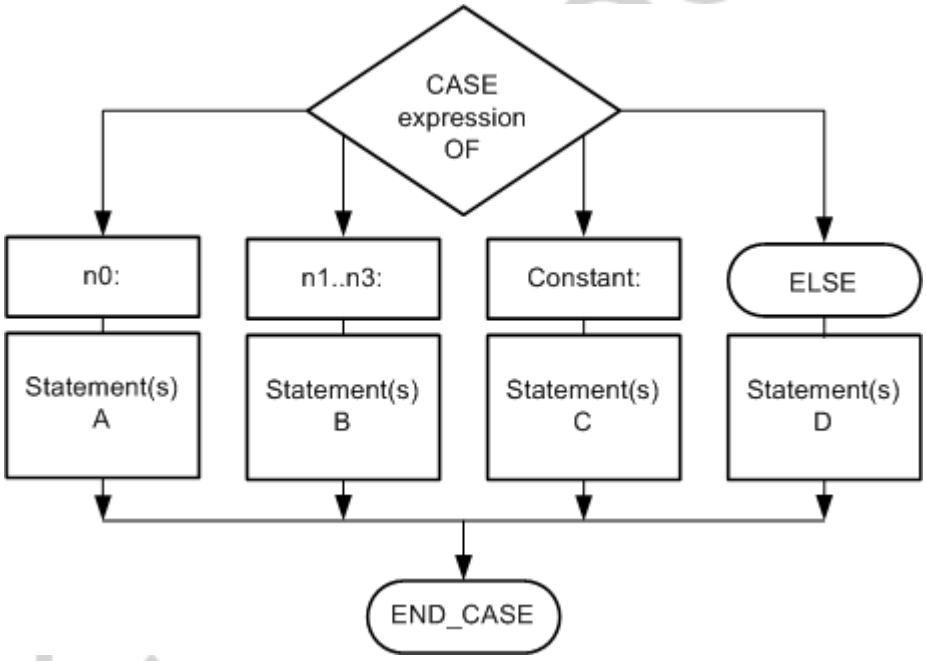


Fig. 28 CASE statement

Only one step of the CASE statement is processed per program cycle.

```
(*****************************************)
(*************** CASE ********************)
(*****************************************)
CASE NumbSelectItem OF
    0:      heat:= LOW;     (*Commands A*)
            fan:= LOW;

    1..3:   heat:= MEDIUM;  (*Commands B*)
            fan:= MEDIUM;

    SELECTIONHIGH:
            heat:= HIGH;    (*Commands C*)
            fan:= HIGH;


ELSE
    heat:= OFF;             (*Commands D*)
    fan:= OFF;
END_CASE
```

Fig. 29 CASE statement in a program

Note:

Constants can be used instead of numbers for the steps in a CASE statement. This makes the program much easier to read.

Syntax of the CASE statement:

- A CASE statement begins with CASE and is terminated with END_CASE. Each of these key words must occupy its own line.
- The variable between CASE and OF must be a UINT variable.
- Only whole number numerical expressions can be used for the definition of the individual steps in the CASE statement.
- The ranges and values of the step variable(s) may not overlap each other.

Task: Brewing tank

The fill level of a brewing tank is monitored for low, ok, and high levels. Use an output for each of the low, ok, and high levels.

The level of liquid in the tank is read as an analog value and is internally converted to 0-100%. If the contents sink below 1%, a warning tone should be triggered.

Create a solution using the CASE statement.

Fig. 30 Brewing tank

4.6 Loops

In many applications, it is necessary for sections of code to be executed multiple times during a cycle. This type of processing is also referred to as a loop. The code in the loop is executed until a defined termination condition is met.

Loops help make programs shorter and easier to follow. Program expandability is also an issue here. Loops can be nested.

Depending on the structure of a program, it is possible for an error in the program to cause the processing gets stuck repeating the loop until the time monitor in the central unit responds with an error.
To prevent such endless loops from occuring, one should almost always include in the programming a way for the loop to be aborted after a defined number of repetitions or to run to a certain limit.

ST offers several types of loops to choose from:

- **Limited**
  - FOR
- **Unlimited**
  - WHILE
  - REPEAT

4.6.1 FOR

The FOR statement is used to run a program section for a limited number of repetitions. For all other applications, WHILE or REPEAT loops are used.

Key words	Syntax	Description
FOR TO BY DO	FOR i:=StartVal TO StopVal {BY Step} DO	The section in {} is optional.
	Res := value + 1;	Loop body statement(s)
END_FOR	END_FOR	End of FOR

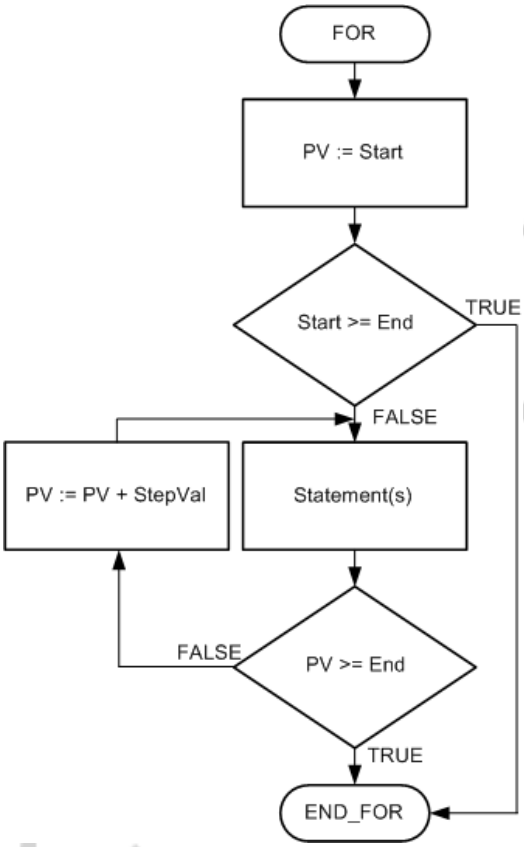


Fig. 31 FOR statement

```
(*********************************************)
(*************       FOR       ***************)
(*********************************************)
FOR i := StartValue TO EndValue BY StepValue DO
  VarFor := VarFor + 1;
END_FOR
(*********************************************)
```

Fig. 1 FOR statement in a program

The statements in the FOR loop are repeated. At every repetition, the loop counter is raised by "StepVal". The two control variables "StartVal" and "EndVal" determine the start value and end value of the loop counter. After the end value is reached, the program continues from after the END_FOR statement. The control variables must both be the same data type and cannot be described by any of the statements in the loop body.

The FOR statement raises or lowers the loop counter until it reaches the end value. The step size is always 1, unless otherwise specified with "BY".

The termination condition, the loop counter, is evaluated with every repetition of the loop.

Task: Crane

5 separate loads are suspended from a crane. In order to determine the total load, you have to add the individual loads together.

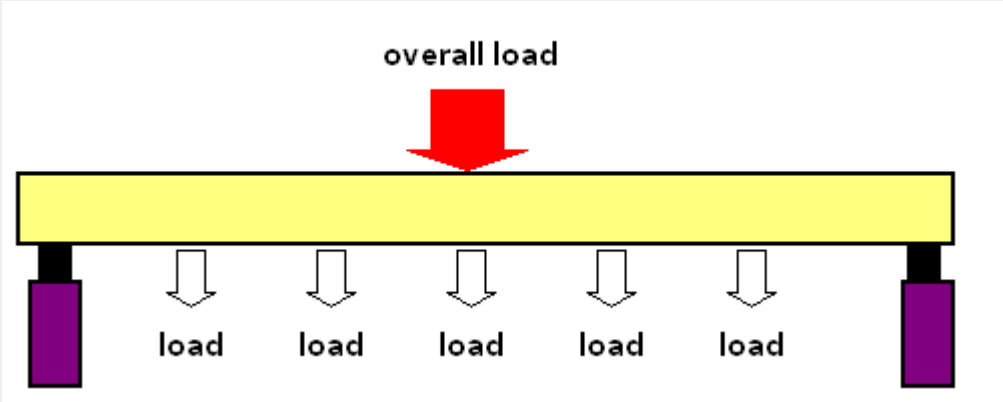


Fig. 32 Crane

Create a solution for this task using a FOR loop.

4.6.2 WHILE

The WHILE loop can be used in the same way as the FOR loop, except that the condition can be any boolean expression. If the condition is met, then the loop is executed. The WHILE loop is used to repeat statements as long as a particular condition remains TRUE.

Keywords	Syntax	Description
WHILE DO	WHILE i<4 DO	Boolean condition
	Res := value + 1;	Statement
	i := i + 1;	Statement
END_ WHILE	END_ WHILE	End of WHILE

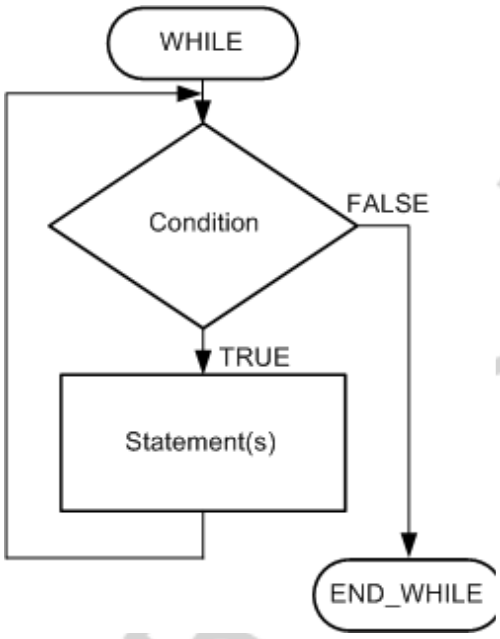


Fig. 33 WHILE statement

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

Fig. 2 WHILE statement in a program

The instructions are executed repeatedly for as long as the condition returns TRUE. If the condition returns FALSE during the first evaluation, the instructions are never executed.

Note:

If the condition never assumes the value FALSE, the statements are repeated endlessly, resulting in a runtime error.

4.6.3 REPEAT

The REPEAT loop differs from the WHILE loop in that the termination condition is only checked once the loop has been executed. This means that the loop runs at least once, regardless of the termination condition.

Keywords	Syntax	Description
REPEAT	REPEAT	Start loop
	Res := value + 1;	Statement
	i := i + 1;	Statement
UNTIL	UNTIL i > 4	Termination condition
END_ REPEAT	END_ REPEAT	End loop

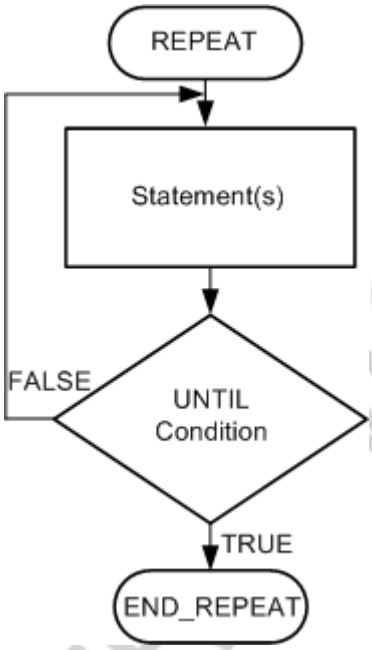


Fig. 34 REPEAT statement

```
(************************)
(******* REPEAT ********)
(************************)
  REPEAT
    Var1:=Var1*2;
    Counter:=counter-1;
  UNTIL
    Counter=0
  END_REPEAT;
```

Fig. 35 REPEAT statement in a program

The statements are executed repeatedly until the UNTIL condition is TRUE. If the UNTIL condition is true from the beginning, the statements are only executed once.

Note:

If the UNTIL condition never assumes the value TRUE, the statements are repeated endlessly, resulting in a runtime error.

4.6.4 EXIT

The EXIT statement can be used with all types of loops before their termination condition occurs.

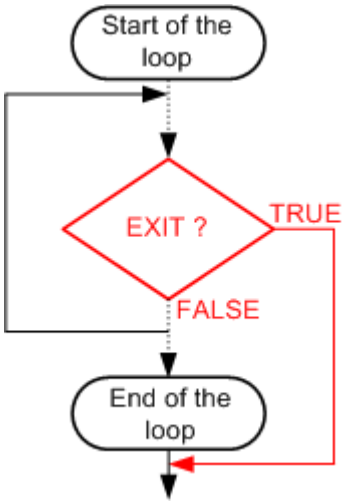


Fig. 36 EXIT statement

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

Fig. 37 EXIT statement in a program

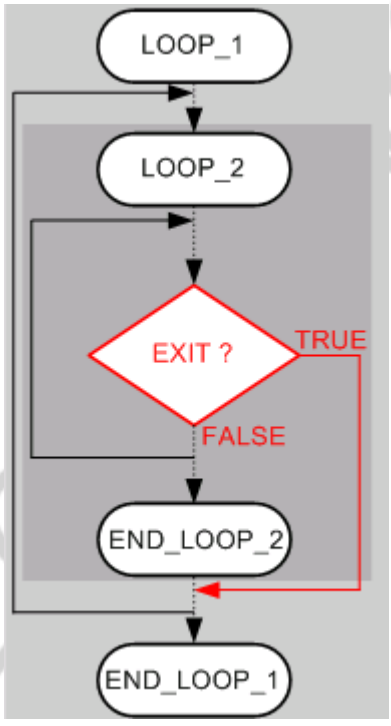


Fig. 38 EXIT statement in a nested FOR statment

```
(***********************************)
(******* EXIT NESTED LOOP **********)
(***********************************)
WHILE (indexWhile < EndIndexWhile)DO
  VarWhile := VarWhile + 1;
  FOR i := StartValue TO EndValue DO
    VarFor_1 := VarFor_1 + 1;
    IF VarFor_1 = VarExit THEN
      EXIT;(*** <<<-- EXIT loop  ***)
    END_IF
  END_FOR
  indexWhile := VarWhile;
END_WHILE
(***********************************)
```

Fig. 39 EXIT statement in a nested FOR statement in a program

If the EXIT statement is used in a nested loop, only the loop in which the EXIT statement is located is ended. After the loop is ended, the program continues from after the END_ statement.

4.7 Calling function blocks

In ST, a function block is called a function block instance, and the necessary transfer parameters are placed in parentheses.

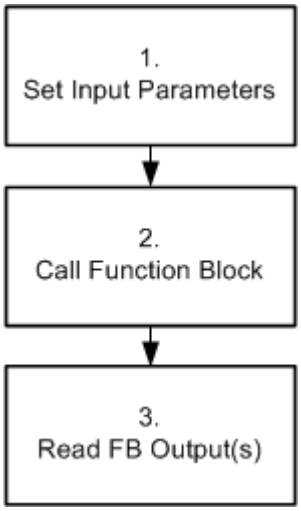


Fig. 40 Calling a function block

```
(*************************************)
(****  Calling of  Function Block   ****)
(*************************************)
PresetTime := T#3s;
TON_0(IN:= DiSensConv1, PT:= PresetTime);
DoConv1 := TON_0.Q;
(*************************************)
```

Fig. 41 Calling a function block in a program

Before a function block is called, one must describe the variables that are to be used as input parameters. The code for calling a function block occupies one line. Then the outputs of the function block can be read.

Function block call in detail:

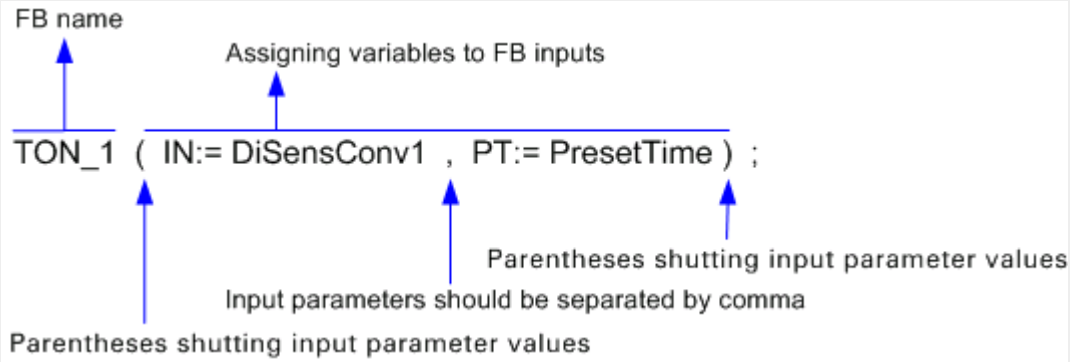


Fig. 42 Detail view of a function block call

First the function block name is entered, then the transfer parameters are assigned in parentheses, separated by commas. The code for calling a function block is closed with a semicolon.

Task: Bottle counter

Create a program that counts the bottles on a conveyor belt. Use the **CTU** (up counter) function block found in the **STANDARD** library.

Fig. 43 Bottle counter

Note:

The Automation Studio online help comes in handy when working with function blocks.

4.8 Pointers and dynamic variables

B&R offers the option of using pointers in ST.

A dynamic variable can be assigned a memory address during runtime. This procedure is referred to as the referencing or initialization of a dynamic variable.

As soon as the dynamic variable is initialized, it can be used to access the memory content to which it now "points".

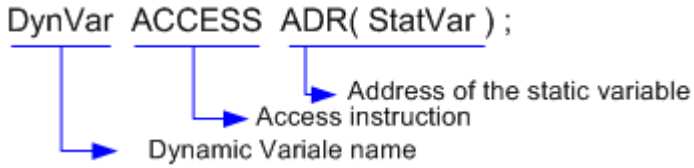


Fig. 44 Referencing a dynamic variable

As you can see, the operator ADR() is used. It returns the memory address of the variable in parentheses. The data type of this address is UDINT. The statement is then closed with a semicolon.

5. SUMMARY

Structured Text is a high level language that offers a wide range of functionality. It contains all the tools necessary for an application.

Fig. 45 Book printing: then and now

After completing this training module, you are ready to program your own ST tasks. You can always use the module as a reference.
This programming language is especially powerful when using arithmetic functions and formulating mathematical calculations.

6. EXERCISES

Task: Box lift

Two conveyor belts (doConvTop, doConvBottom) transport boxes to a lift.

If the photocell is (diConvTop, diConvBottom) is activated, the corresponding conveyor belt is stopped and the lift is called.

If the lift has not been called, it returns to the appropriate position (doLiftTop, doLiftBottom).

When the lift is in the correct position (diLiftTop, diLiftBottom), the lift conveyer belt (doConvLift) is turned on until the box is completely on the lift (diBoxLift).

Then the lift moves to the unloading position (doLiftUnload). When it reaches this position (diLiftUnload), the box is moved to the unloading belt.

As soon as the box has left the lift, the lift is free for the next request.

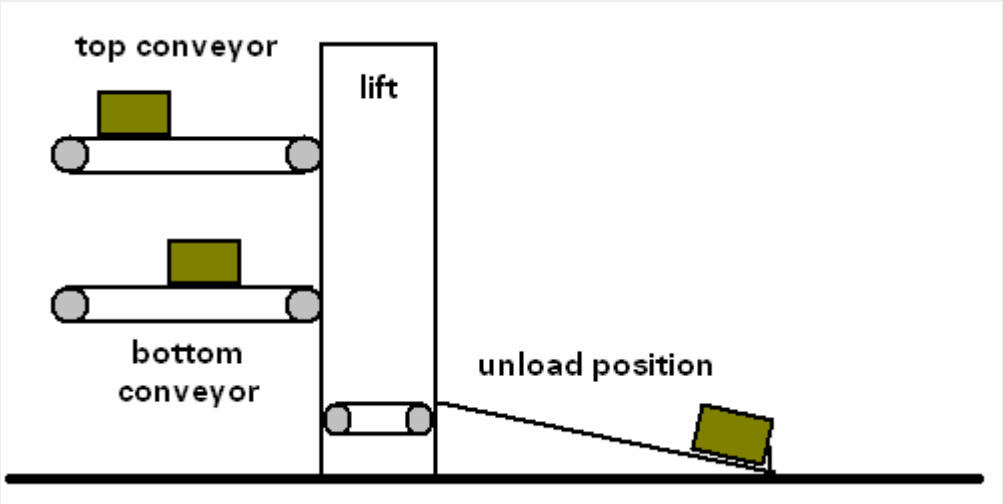


Fig. 46 Box lift

7. APPENDIX

7.1 Keywords

Key words are commands that can be used in ST. In the Automation Studio Editor, these are displayed in blue. You are already familiar with many of them; here is a list of more. Key words may not be used as variable names.

Keyword	Description
ACCESS	Assignment of an address to a dynamic variable.
BIT_CLR	A := BIT_CLR(IN, POS) A is the value of the variable IN that results when the bit at position POS is deleted. IN remains unchanged.
BIT_SET	A := BIT_CLR(IN, POS) A is the value of the variable IN that results when the bit at position POS is set. The IN operand remains unchanged.
BIT_TST	Determination of a bit within a value. A is the state of the bit of the IN value that is at position POS.
BY	See FOR statement.
CASE	See CASE statement.
DO	See WHILE statement.
EDGE	Determines the negative and positive edges of a bit.
EDGENEG	Determines the negative edge of a bit.
EDGEPOS	Determines the positive edge of a bit.
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	Can be used to end a function.
THEN	See IF statement.
TO	See FOR statement.
UNTIL	See REPEAT statement.
WHILE	See WHILE statement.

7.2 Functions

There are some functions that can be used in ST that do not require you to insert a library into the project. In the Automation Studio Editor, these function calls are displayed in blue. You are already familiar with some of them. More are listed here.

Function	Example
ABS	Returns the absolute value of a number. ABS(-2) returns 2.
ACOS	Returns the cosine of a number. (inverted cosine function).
ADR	Returns a variable's address.
AND	Logical AND for bit operations.
ASIN	Returns the arc sine of a number (inverse function of sine).
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	Returns the arc tangent of a number (inverse function of tangent).
COS	Returns the cosine of a number.
EXP	Exponential function: 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, SINT, INT, UINT, UDINT, DINT type variable by another variable of one of these types.
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 of the operators IN1, IN2, ... INX is returned as a 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); The bits in IN 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 bytes required by the specified variable.
SQRT	Returns the square root of a number.
TAN	Returns the tangent of a number.
TRUNC	Returns the integer part of a number.
XOR	Logical EXCLUSIVE OR operation by bit.

7.3 Solutions

Task: Light control

```
DoLight:= (BtnLightOn OR DoLight) AND NOT(BtnLightOff);
```

Task: Aquarium

```
aoAvgTemp:= DINT_TO_UINT((UINT_TO_DINT(aiTemp1) + aiTemp2) / 2);
```

Task: Weather station - Part I

```
IF (AItmp < 180) THEN
    gtxt:= 'COLD';
ELSIF (AItmp > 250) THEN
    gtxt:= 'HOT';
ELSE
    gtxt:= 'OPT';
END_IF
```

Task: Weather station - Part II

```
IF (AItmp < 180) THEN
    gtxt:= 'COLD';
ELSIF (AItmp > 250) THEN
    gtxt:= 'HOT';
ELSE
    IF (AIhum > 400) AND (AIhum < 750) THEN
        gtxt:= 'OPT';
    ELSE
        gtxt:= 'Temp.OK';
    END_IF
END_IF
```

Task: Brewing tank

```
(*convert to percent value*)
level:= DINT_TO_USINT((INT_TO_DINT(aiLevel)*100)/32767);

doHorn:= 0;
doLow:= 0;
doOk:= 0;
doHigh:= 0;

CASE level OF
    (* <1% turn the horn on*)
    0,1:
            doHorn:= 1;
            doLow:= 1;
    (* <25% *)
    2..24:
            doLow:= 1;
    (* >90% *)
    91..100:
            doHigh:= 1;
ELSE
    doOk:= 1;
END_CASE
```

Task: Crane

```
(* cyclic program *)
overall_load:= 0;
FOR i:=0 TO 4 DO
    overall_load:= overall_load + load[i];
END_FOR
```

Task: Bottle counter

```
CTU_0(CU:=EDGEPOS(diBottle), RESET:=diReset, PV:=cntCompare);
cntBottle:= CTU_0.CV;
```

Task: Box lift

```
(* conveyor *)
doConvTop:= NOT (diConvTop) OR ConvTopOn;
doConvBottom:= NOT (diConvBottom) OR ConvBottomOn;
(* lift *)
CASE selectLift OF
    (* wait for request *)
    WAIT:
        IF (diConvTop = TRUE) THEN
            selectLift:= TOP_POSITION;
        ELSIF (diConvBottom = TRUE) THEN
            selectLift:= BOTTOM_POSITION;
        END_IF
    (* move lift to top position *)
    TOP_POSITION:
        doLiftTop:= TRUE;
        IF (diLiftTop = TRUE) THEN
            doLiftTop:= FALSE;
            ConvTopOn:= TRUE;
            selectLift:= GETBOX;
        END_IF
    (* move lift to bottom position *)
    BOTTOM_POSITION:
        doLiftBottom:= TRUE;
        IF (diLiftBottom = TRUE) THEN
            doLiftBottom:= FALSE;
            ConvBottomOn:= TRUE;
            selectLift:= GETBOX;
        END_IF
    (* move box to lift *)
    GETBOX:
        doConvLift:= TRUE;
        IF (diBoxLift = TRUE) THEN
            doConvLift:= FALSE;
            ConvTopOn:= FALSE;
            ConvBottomOn:= FALSE;
            selectLift:= UNLOAD_POSITION;
        END_IF
    (* move lift to unload position *)
    UNLOAD_POSITION:
        doLiftUnload:= TRUE;
        IF (diLiftUnload = TRUE) THEN
            doLiftUnload:= FALSE;
            selectLift:= UNLOAD_BOX;
        END_IF
    (* unload the box *)
    UNLOAD_BOX:
        doConvLift:= TRUE;
        IF (diBoxLift = FALSE) THEN
            doConvLift:= FALSE;
            selectLift:= WAIT;
        END_IF
END_CASE
```

Notes

Overview of training modules

TM200 – B&R Company Presentation **
TM201 – B&R Product Spectrum **
TM210 – The Basics of Automation Studio
TM211 – Automation Studio Online Communication
TM212 – Automation Target **
TM213 – Automation Runtime
TM220 – The Service Technician on the Job
TM223 – Automation Studio Diagnostics
TM230 – Structured Software Generation
TM240 – Ladder Diagram (LAD)
TM241 – Function Block Diagram (FBD)
TM246 – Structured Text (ST)
TM247 – Automation Basic (AB)
TM248 – ANSI C
TM250 – Memory Management and Data Storage
TM260 – Automation Studio Libraries I
TM261 – Closed Loop Control with LOOPCONR

TM400 – The Basics of Motion Control
TM410 – The Basics of ASiM
TM440 – ASiM Basic Functions
TM441 – ASiM Multi-Axis Functions
TM445 – ACOPOS ACP10 Software
TM450 – ACOPOS Control Concept and Adjustment
TM460 – Starting up Motors

TM500 – The Basics of Integrated Safety Technology
TM510 – ASiST SafeDESIGNER

TM600 – The Basics of Visualization
TM610 – The Basics of ASiV
TM630 – Visualization Programming Guide
TM640 – ASiV Alarm System
TM650 – ASiV Internationalization
TM660 – ASiV Remote
TM670 – ASiV Advanced

TM700 – Automation Net PVI
TM710 – PVI Communication
TM711 – PVI DLL Programming
TM712 – PVIServices
TM730 – PVI OPC

TM800 – APROL System Concept
TM810 – APROL Setup, Configuration and Recovery
TM811 – APROL Runtime System
TM812 – APROL Operator Management
TM813 – APROL XML Queries and Audit Trail
TM830 – APROL Project Engineering
TM840 – APROL Parameter Management and Recipes
TM850 – APROL Controller Configuration and INA
TM860 – APROL Library Engineering
TM865 – APROL Library Guide Book
TM870 – APROL Python Programming
TM890 – The Basics of LINUX

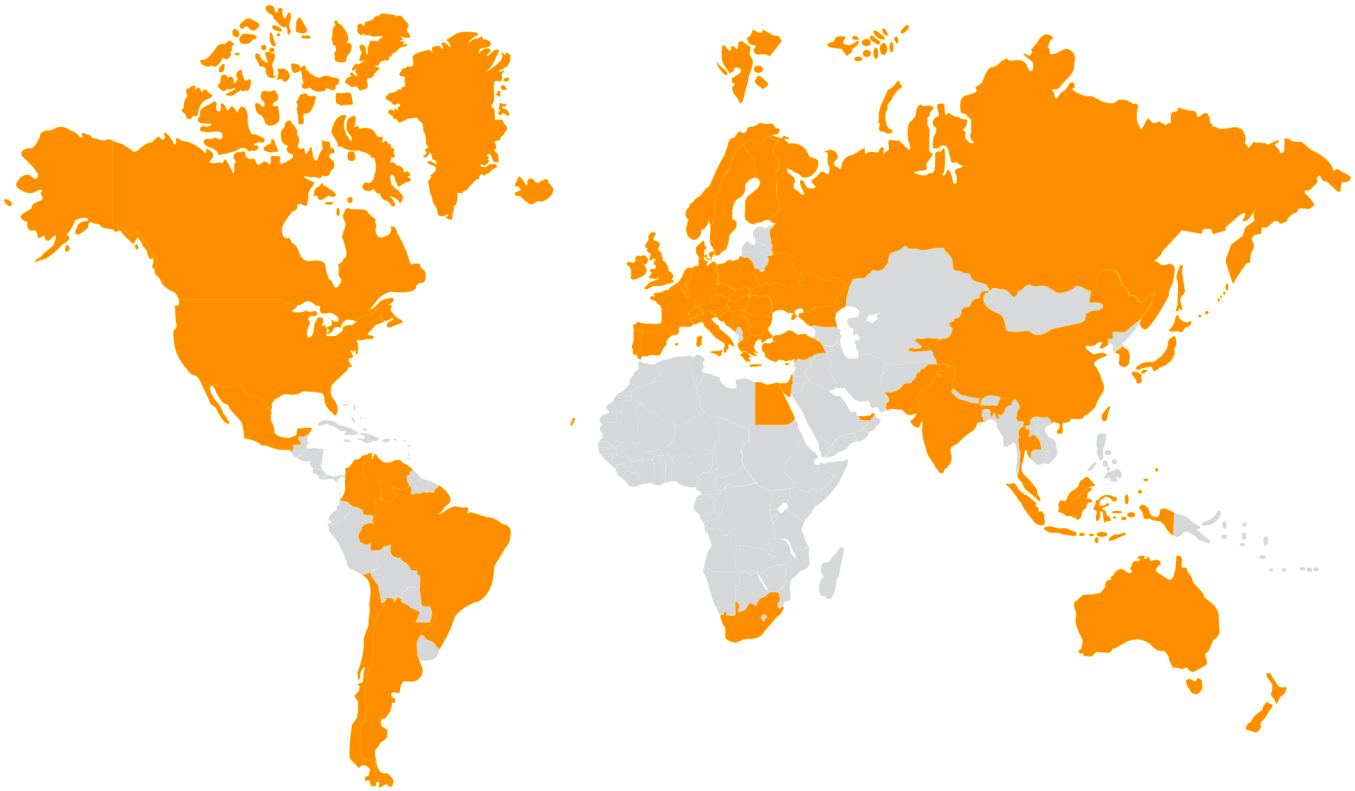
**) see Product Catalog

CORPORATE HEADQUARTERS

Bernecker + Rainer Industrie-Elektronik Ges.m.b.H.
B&R Strasse 1
5142 Eggelsberg
Austria
Tel.: +43 (0) 77 48 / 65 86 - 0
Fax: +43 (0) 77 48 / 65 86 - 26
info@br-automation.com
www.br-automation.com

TM246TRE.00-ENG 0907

140 offices in more than 55 countries - **www.br-automation.com/contact**



Australia • Argentina • Austria • Belarus • Belgium • Brazil • Bulgaria • Canada • Chile • China • Colombia • Croatia • Cyprus
Czech Republic • Denmark • Egypt • Emirates • Finland • France • Germany • Greece • Hungary • India • Indonesia
Ireland • Israel • Italy • Japan • Korea • Luxemburg • Kyrgyzstan • Malaysia • Mexico • The Netherlands • New Zealand
Norway • Pakistan • Poland • Portugal • Romania • Russia • Serbia • Singapore • Slovakia • Slovenia • South Africa
Spain • Sweden • Switzerland • Taiwan • Thailand • Turkey • Ukraine • United Kingdom • USA • Venezuela • Vietnam