

Structured Text (ST)

TM246



Requirements

Training modules:	TM210 – The Basics of Automation Studio TM213 – Automation Runtime TM223 – Automation Studio Diagnostics
Software	Automation Studio 3.0.90 or higher
Hardware	None

TABLE OF CONTENTS

1 INTRODUCTION.....	4
1.1 Training module objectives.....	4
2 GENERAL INFORMATION.....	5
2.1 Features of Structured Text.....	5
2.2 Editor functions.....	6
3 BASIC ELEMENTS.....	7
3.1 Expressions.....	7
3.2 Assignments.....	7
3.3 Source code documentation: Comments.....	7
3.4 Order of operations.....	8
3.5 Reserved keywords.....	9
4 COMMAND GROUPS.....	10
4.1 Boolean operations.....	10
4.2 Arithmetic operations.....	11
4.3 Comparison operators and decisions.....	14
4.4 CASE statement - State machines.....	17
4.5 Loops.....	18
5 FUNCTIONS, FUNCTION BLOCKS AND ACTIONS.....	24
5.1 Calling functions and function blocks.....	24
5.2 Calling actions.....	26
6 ADDITIONAL FUNCTIONS, AUXILIARY FUNCTIONS.....	27
6.1 Pointers and references.....	27
6.2 Preprocessor for IEC programs.....	27
7 DIAGNOSTIC FUNCTIONS.....	28
8 EXERCISES.....	29
8.1 Box lift exercise.....	29
9 SUMMARY.....	30
10 EXERCISE SOLUTIONS.....	31

1 INTRODUCTION

Structured Text is a high-level programming language whose basic concept includes elements taken from the languages BASIC, Pascal and ANSI C. With its easy-to-understand standard constructs, Structured Text (ST) is a fast and efficient way of programming in the field of automation.



The following chapters will introduce you to the commands, keywords and syntax used in Structured Text. Simple examples will give you a chance to use put these concepts into practice to make it easier for you to understand them.

1.1 Training module objectives

In this training module, you will learn ...

- ... The ST programming language and its features
- ... How to use the standard constructs in ST
- ... Keywords
- ... How to implement simple automation applications
- ... Function and function block calls

2 GENERAL INFORMATION

2.1 Features of Structured Text

General information

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

Structured Text is included in the IEC standard¹.

Features

Structured Text is characterized by the following features:

- High-level, text-based language
- Structured programming
- Easy-to-use standard constructs
- Fast and efficient programming
- Self-explanatory and flexible
- Similar to Pascal
- Conforms to the IEC 61131-3 standard

Possibilities

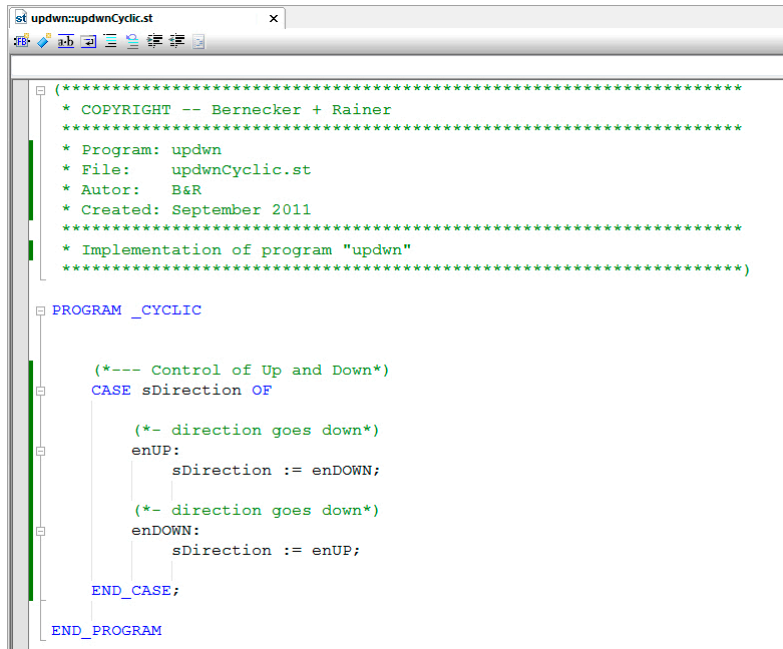
Automation Studio supports the following functions:

- Digital and analog inputs and outputs
- Logical operations
- Logical comparison expressions
- Arithmetic operations
- Decisions
- Step sequencers
- Loops
- Function blocks
- Dynamic variables
- Calling actions
- Integrated diagnostics

¹ The IEC 61131-3 standard is the only valid international standard for programming languages used on programmable logic controllers. This standard also includes Ladder Diagram, Instruction List and Function Block Diagram.

2.2 Editor functions

The editor is a text editor with many additional functions. Commands and keywords are shown in color. Areas can be expanded and collapsed, with autocomplete available for variables and constructs (SmartEdit).



```
st updwn:updwnCyclic.st
{
  *****
  * COPYRIGHT -- Bernecker + Rainer
  *****
  * Program: updwn
  * File:   updwnCyclic.st
  * Autor:  B&R
  * Created: September 2011
  *****
  * Implementation of program "updwn"
  *****
}

PROGRAM _CYCLIC

  (*--- Control of Up and Down*)
  CASE sDirection OF

    (*- direction goes down*)
    enUP:
      sDirection := enDOWN;

    (*- direction goes down*)
    enDOWN:
      sDirection := enUP;

  END_CASE;

END_PROGRAM
```

User program in the ST editor

The editor has the following functions and features:

- Distinction between uppercase and lowercase letters (case sensitive)
- Autocomplete (SmartEdit, <CTRL> + <SPACE>, <TAB>)
- Inserting and managing code snippets (<CTRL> + q, k)
- Identification of corresponding pairs of parentheses
- Expanding and collapsing constructs (outlining)
- Inserting block comments
- URL recognition
- Markers for modified lines



[Programming \ Editors \ Text editors](#)

[Programming \ Editors \ General operations \ SmartEdit](#)

[Programming \ Editors \ General operations \ SmartEdit \ Code snippets](#)

[Programming \ Programs \ Structured Text \(ST\)](#)

3 BASIC ELEMENTS

The following section describes the basic elements of ST in more detail.

3.1 Expressions

An expression is a construct that returns a value after it has been calculated. Expressions consist of operators and operands. An operand can be a variable, constant or function call. Operators are used to connect the operands (3.4 "Order of operations").



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

Table: Examples of expressions

3.2 Assignments

Assignments consists of a variable on the left side which is assigned the result of a calculation or expression on the right side using the assignment operator ":=". All assignments must be completed with a semicolon ";".



```
Result := ProcessValue * 2; (*Result ← (ProcessValue * 2) *)
```

Table: Assignment takes place from left to right

When this line of code is executed, the value of the "Result" variable is twice the value of the "ProcessValue" variable.

Accessing variable bits

It is also possible to only deal with particular bits when making assignments. To do so, simply place a period (".") after the name of the variable. Access then takes place using the bit number, beginning with 0. Constants can also be used in the place of the bit number.



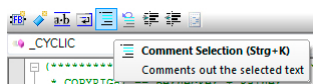
```
Result := ProcessValue.1;
```

Table: Accessing the second bit of "ProcessValue"

3.3 Source code documentation: Comments

Comments are an important part of source code. They describe the code and make it more understandable and readable. Comments make it possible for you or others to understand a program long after it has been completed. They are not compiled and have no influence on program execution. Comments must be placed between a pair of parenthesis and asterisks, e.g. (*comment*).

An additional comment form is introduced with "//". Several lines can be selected in the editor and commented out with one of the icons in the toolbar. This variant is an expansion to the existing IEC standard.



Commenting out a block of text


	Single-line comment	<code>(*This is a single comment line.*)</code>
	Multi-line comment	<code>(* These are several lines of commenting. *)</code>
	Comment with "//"	<code>// This is a general // text block. // It is commented out.</code>


Table: Comment variations

	Programming \ Editors \ Text editors \ Commenting out a selection
	Programming \ Structured software development \ Program layout

3.4 Order of operations

The use of different operators brings up the question of priority. The priority of the operators is important when solving an expression.

Expressions are solved by taking the operators with the highest priority into account first. Operators with the same priority are executed from left to right as they appear in the expression.

Operator	Syntax	
Parentheses	<code>()</code>	Highest priority
Function call	<code>Call (Argument)</code>	
Exponent	<code>**</code>	
Negation	<code>NOT</code>	
Multiplication, division, modulo operator	<code>*, /, MOD</code>	
Addition, subtraction	<code>+, -</code>	
Comparisons	<code><, >, <=, >=</code>	
Equality comparisons	<code>=, <></code>	
Boolean AND	<code>AND</code>	
Boolean XOR	<code>XOR</code>	
Boolean OR	<code>OR</code>	Lowest priority

The resolution of the expression is performed by the compiler. The following examples show that different results can be achieved through the use of parentheses.

**Solving an expression without parentheses:**

```
Result := 6 + 7 * 5 - 3;      (* Result is 38 *)
```

Multiplication is performed first, followed by addition. Subtraction is performed last.

Solving an expression with parentheses:

```
Result := (6 + 7) * (5 - 3);  (* Result is 26 *)
```

The expression is executed from left to right. The operations in parentheses are executed before the multiplication since the parentheses have higher priority. You can see that using parentheses can lead to different results.

3.5 Reserved keywords

In programming, all variables must adhere to certain naming conventions. In addition, there are reserved keywords that are already recognized as such by the editor and are shown in a different color. These keywords cannot be used as variables.

The libraries OPERATOR and AslecCon are a standard part of new projects. The functions they contain are IEC functions that are interpreted as keywords.

In addition, the standard also defines literals for numbers and character strings. This makes it possible to represent numbers in different formats.



[Programming \ Structured software development \ Naming conventions](#)

[Programming \ Standards \ Literals in IEC languages](#)

[Programming \ Programs \ Structured Text \(ST\) \ Keywords](#)

[Programming \ Libraries \ IEC 61131-3 functions](#)

4 COMMAND GROUPS

The following command groups are the basic constructs of high-level language programming.

They can be categorized according to the following command group categories:

- Boolean operations
- Arithmetic operations
- Comparison operators and decisions
- State machines - Case statement
- Loops

4.1 Boolean operations

The operands must not necessarily be of data type BOOL.

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 ;

Table: Overview of Boolean connectives

The truth table for these operations looks like this:

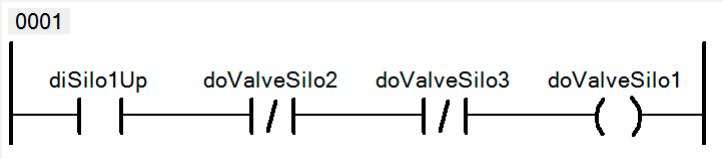
Input		AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Table: Truth table for Boolean connectives

Boolean operations can be combined in any way. Additional sets of parentheses increase program readability and ensure that the expression is solved correctly. The only possible results of the expression are TRUE (logical 1) or FALSE (logical 0).



Boolean connective - Comparison between Ladder Diagram and Structured Text



Linking normally open and normally closed contacts

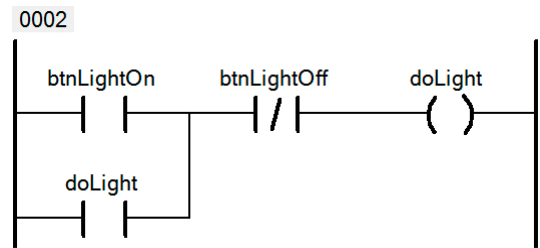
```
doValveSilo1 := diSilo1Up AND NOT doValveSilo2 AND NOT doValveSilo3;
```

Table: Implementation with Boolean connectives

Parentheses are not needed since NOT has a higher priority than AND. Nevertheless, it is always a good idea to use parentheses to make the program clearer.

Exercise: Light control

The output "DoLight" should be ON when the "BtnLightOn" button is pressed and should remain ON until the "BtnLightOff" button is pressed. Solve this exercise using Boolean operations.



On-off switch, relay with latch

4.2 Arithmetic operations

A key advantage of high-level programming languages is their easy handling of arithmetic operations.

Overview of arithmetic operations

Structured Text provides basic arithmetic operations for the application:

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, remainder of integer division	a := b MOD c;

Table: Overview of arithmetic operations

The data type of variables and values is always crucial to calculations. The result is calculated on the right side of the expression and only then assigned to the corresponding variable. The result depends on the data types and syntax (notation) being used. The following table illustrates this.

Expression / Syntax	Data types			Result
	Result	Operand1	Operand2	
<code>Result := 8 / 3;</code>	INT	INT	INT	2
<code>Result := 8 / 3;</code>	REAL	INT	INT	2.0
<code>Result := 8.0 / 3;</code>	REAL	REAL	INT	2.66667
<code>Result := 8.0 / 3;</code>	INT	REAL	INT	*Error

Table: Performed by the compiler implicitly convert

The resulting value "*Error" stands for the following compiler error message: "Error 1140: Incompatible data types: Cannot convert REAL to INT.". This is because it is not possible to assign the expression to this particular data type.

4.2.1 Data type conversion

When programming, one is inevitably confronted with different data types. In principle, one can mix data types within a program. Assignment of different data types is also possible. Nevertheless, you should be cautious with this.

Implicit data type conversion

Every time assignment is performed in the program code, the compiler checks the data types. In a statement, assignments are always made from right to left. The variable must therefore have enough space to hold the value. Converting from a smaller data type to a larger is carried out implicitly by the compiler without any action required by the user.

Attempting to assign a value with a larger data type to a variable of a smaller data type will result in a compiler error. In this case, explicit data type conversion is necessary.



Even if implicit data type conversion is carried out, it's still possible that an incorrect result will occur depending on the compiler and what's being converted.

One danger here involves assigning an unsigned value to a signed data type.

This can lead to an overflow error when adding or multiplying. This depends on the platform being used, however. The compiler does not issue a warning in this case.



	Expression	Data types	Note
	Result := Value; (*UINT USINT*)	UINT, USINT	No further action necessary for implicit conversion
	Result := Value; (*INT UINT*)	INT, UINT	Caution when using negative numbers!
	Result := value1 + value2; (*UINT USINT USINT*)	UINT, USINT, USINT	Danger of range overflow when adding


Table: Examples of implicit data type conversion

Explicit data type conversion

Although implicit data type conversion is often the more convenient method, it should not always be the first choice. Clean programming necessitates that types are handled correctly using explicit data type conversion. The examples below highlight some of the cases where explicit conversion is necessary.



All variable declarations are listed in IEC format. Variable declarations can be entered in this format in the text view of the .var file for the respective program.



Here there is the danger of an overflow error when the addition takes place:

Declaration:	<pre>VAR TotalWeight: INT; Weight1: INT; Weight2: INT; END_VAR</pre>
Program code:	<pre>TotalWeight := Weight1 + Weight2;</pre>

Table: An overflow error can occur directly to the right of the assignment operator.

In this case, the data type for the result must have enough space to hold the sum of the addition operation. As a result, a larger data type is necessary. When adding, at least one of the operands must be converted to the larger data type. The conversion of the second operand is then handled implicitly by the compiler.

Declaration:	<pre>VAR TotalWeight: DINT; Weight1: INT; Weight2: INT; END_VAR</pre>
Program code:	<pre>TotalWeight := INT_TO_DINT(Weight1) + Weight2;</pre>

Table: Overflow errors can be prevented by using explicit conversion.

On 32-bit platforms, the operands being calculated are converted to 32-bit values. In this case, adding does not cause an overflow error.



Programming \ Variable and data types \ Data types \ Basic data types

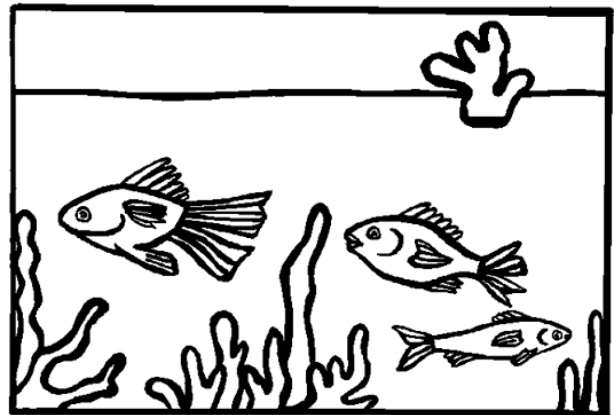
Programming \ Libraries \ IEC 61131-3 functions \ CONVERT

Programming \ Editors \ Text editors

Programming \ Editors \ Table editors \ Declaration editors \ Variable declaration

Exercise: Aquarium

The temperature of an aquarium is measured at two different places. Create a program that calculates the average temperature and passes it along to an analog output. Don't forget that analog inputs and outputs must be of data type INT.



Aquarium

Declaration:

```
VAR
    aiTemp1: INT;
    aiTemp2: INT;
    aoAvgTemp: INT;
END_VAR
```

Table: Suggestion for variable declaration

4.3 Comparison operators and decisions

In Structured Text, simple constructs are provided to compare variables. They return either the value TRUE or FALSE. Comparison operators and logical operations are mainly used in statements such as IF, ELSIF, WHILE and UNTIL as conditions.

Symbol	Comparison type	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

Table: Overview of logical comparison operators

Decisions

The IF statement is one way for the program to handle decisions. You already know the comparison operators. They can be used here.

Keywords	Syntax	Description
IF .. THEN	IF a > b THEN	1st comparison
	Result := 1;	Statement if 1st comparison is TRUE
ELSIF .. THEN	ELSIF a > c THEN	2nd comparison
	Result := 2;	Statement if 2nd comparison is TRUE
ELSE	ELSE	Alternative branch, no comparisons TRUE
	Result := 3;	Statement for the alternative branch
END_IF	END_IF	End of decision

Table: Syntax of an IF statement

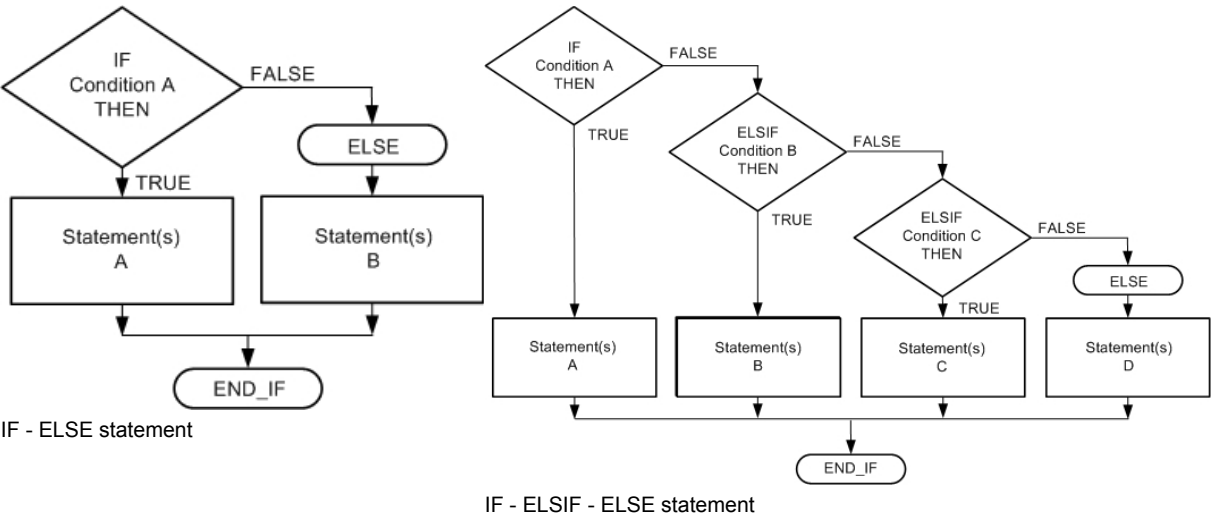


Table: IF statements at a glance

Comparison expressions can be combined with Boolean connectives so that multiple conditions can be tested simultaneously.


	Explanation:	If "a" is greater than "b" and if "a" is less than "c", then "Result" is equal to 100.
	Program code:	<pre>IF (a > b) AND (a < c) THEN Result := 100; END_IF;</pre>

Table: Using multiple comparison expressions

An IF statement can also include additional nested IF statements. It is important that there are not too many nesting levels as this has a tendency to make the program too confusing.



The SmartEdit feature of the editor can be used to enter information more easily. If you want to insert an IF statement, simply type in "IF" and then press the <TAB> key. This automatically adds the basic structure of the IF statement in the editor.



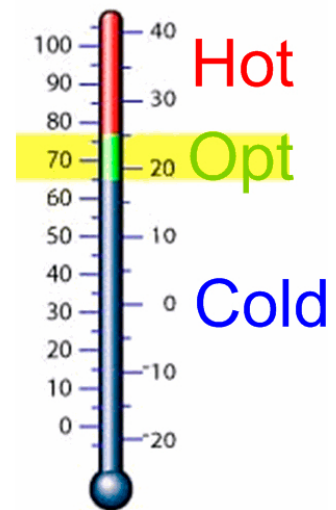
Programming \ Programs \ Structured Text (ST) \ IF statement
Programming \ Editors \ General operations \ SmartEdit

Exercise: Weather station - Part I

A temperature sensor measures the outside temperature. This temperature is read using an analog input and should be output as textual information inside the house.

- 1) If the temperature is below 18°C, the display should read "Cold".
- 2) If the temperature is between 18°C and 25°C, the display should read "Opt" (optimal).
- 3) If the temperature is over 25°C, the display should read "Hot".

Create a solution using IF, ELSIF and ELSE statements.



Thermometer



A variable of data type STRING is needed to output the text. The assignment would look like this: `sShowText := 'COLD';`

Exercise: Weather station - Part II

Evaluate the humidity in addition to the temperature.

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.



If several IF statements are checking the same variable, then it's useful to think about whether using a CASE statement might be a better, clearer solution.

The CASE statement has an additional advantages over the IF statement in that comparisons only have to be made once, making the program code more effective.

4.4 CASE statement - State machines

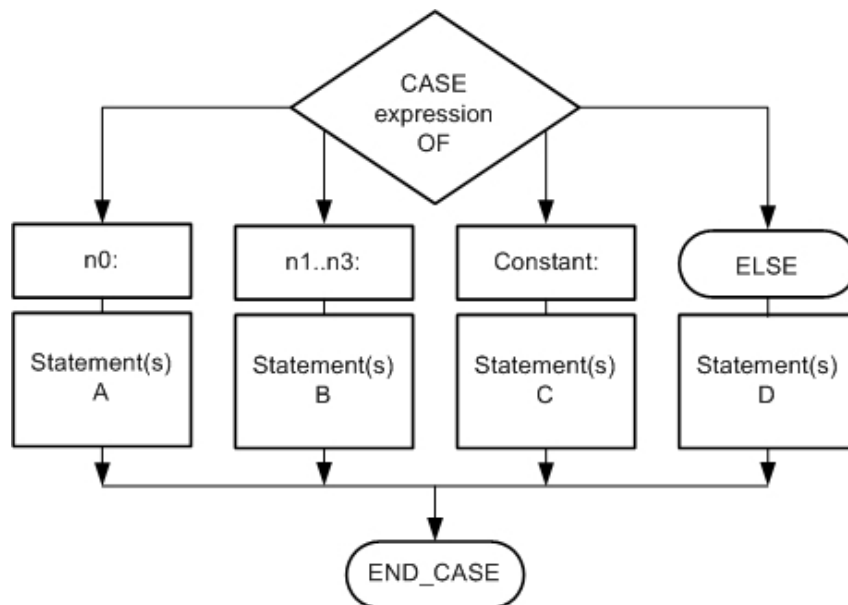
The CASE statement compares a step variable with multiple values. If one of the comparisons matches, the statements that are associated with that step are executed. If none of the comparisons match, then the program code under the ELSE statement is executed, as is the case with the IF statement.

Depending on the application, the CASE statement can be used to set up state machines or automata.

Keywords	Syntax	Description
<code>CASE .. OF</code>	<code>CASE sStep OF</code>	Beginning of CASE
	<pre> 1, 5: Show := MATERIAL; </pre>	For 1 and 5
	<pre> 2: Show := TEMP; </pre>	For 2
	<pre> 3, 4, 6..10: Show := OPERATION; </pre>	For 3, 4, 6, 7, 8, 9 and 10
<code>ELSE</code>	<code>ELSE</code>	Alternative branch
	<code>(* . . . *)</code>	
<code>END_CASE</code>	<code>END_CASE</code>	End of CASE

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

The step variable must be a positive integer data type (e.g. USINT, UINT, UINT).



Overview of the CASE statement



Instead of fixed numerical values, constants or elements of enumerated data types should be used in the program code. Replacing a value with text makes the program code must easier to read. In addition, if these values need to be changed in the program, then the only change that needs to take place is in the declaration itself, not in the program code.



Programming \ Programs \ Structured Text (ST) \ CASE statement

Programming \ Variables and data types \ Variables \ Constants

Programming \ Variables and data types \ Data types \ Derived data types \ Enumerations

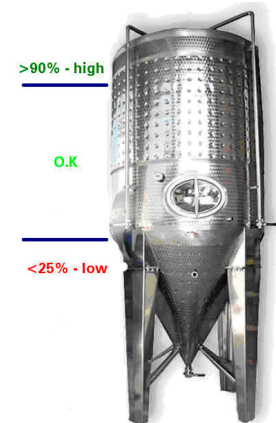
Exercise: Level control

The level in a container should be monitored for three areas: low, high and OK.

Use one output for each of the low, OK, and high levels.

The level of liquid in the tank is read as an analog value (0 - 32767) and is converted internally to a percentage value (0-100%).

A warning tone should be given if the contents fall below 1%. Create a solution using the CASE statement.



Evaluating the fill level of a container

Declaration:

```
VAR
    aiLevel : INT;
    PercentLevel : UINT;
    doLow : BOOL;
    doOk : BOOL;
    doHigh : BOOL;
    doAlarm : BOOL;
END_VAR
```

Table: Suggestion for variable declaration

4.5 Loops

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

Loops are used to make programs clearer and shorter. The ability to expand the program's functionality is also an issue here.

Depending on how a program is structured, it is possible that an error will prevent the program from leaving the loop until the CPU's time monitoring mechanism intercedes. To avoid such infinite loops, always provide a way to terminate the loop after a specified number of repetitions.

There are two main types of loops: those where loop control begins at the top and those where it begins at the bottom.

Loops where control begins at the top (FOR, WHILE) check the termination condition before entering the loop. Loops where control begins at the bottom (REPEAT) check the condition at the end of the loop. These will always be cycle through at least once.

4.5.1 FOR statement

The FOR statement is used to execute a program section for a limited number of repetitions. WHILE and REPEAT loops are used for applications where the number of cycles is not known in advance.

Keywords	Syntax
FOR .. TO .. BY ² .. DO	FOR i:= StartVal TO StopVal BY Step DO
	Res := Res + 1;
END_FOR	END_FOR

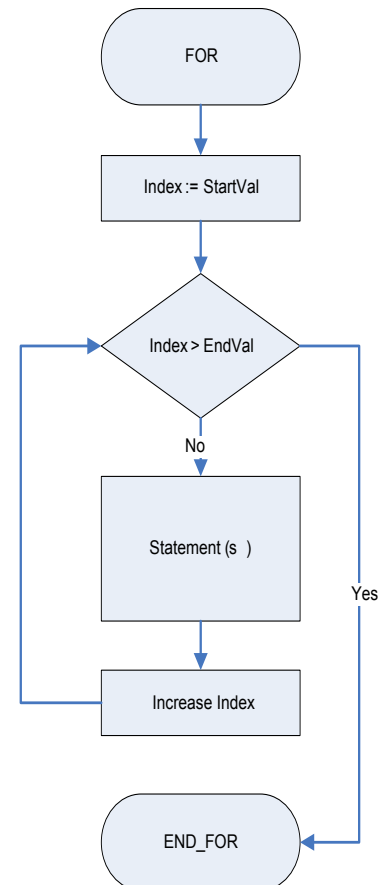
Table: Elements of the FOR statement

The loop counter "Index" is pre-initialized with the start value "StartVal". The loop is repeated until the value of the terminating variable "StopVal" is reached. Here, the loop counter is always increased by 1, known as the "BY step". If a negative value is used for the "step" increment, then the loop will count backwards.

The loop counter, the start value and the end value must all have the same data type. This can be achieved by explicit data conversion(4.2.1 "Data type conversion").



If the starting and ending values are the same at the beginning, this type of loop will cycle through at least once (for example, if the starting and ending values are both 0).



Overview of the FOR statement

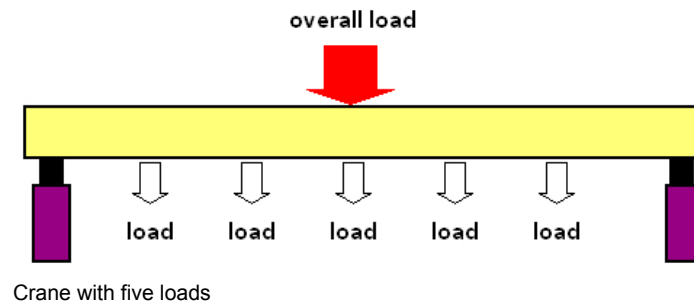


Programming \ Programs \ Structured Text (ST) \ FOR statement

² Specifying of the keyword "BY" is optional.

Exercise: Crane

A crane can lift five loads at once. In order to determine the total load, the individual loads have to be added together. Create a solution for this exercise using a FOR loop.



Declaration:

```
VAR
    aWeights : ARRAY [0..4] OF INT;
    sumWeight : DINT;
END_VAR
```



Where possible, use array declarations for the loads and constants to limit the loop's end values. This considerably improves the readability of declarations and programs. Making changes later also becomes much easier.

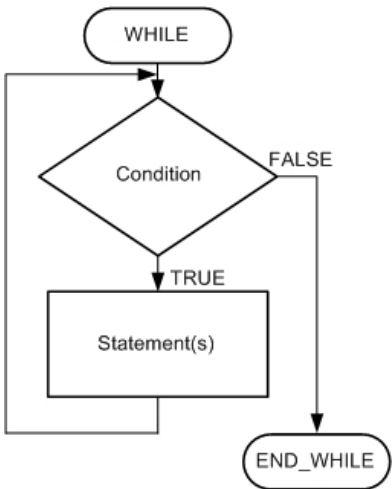
4.5.2 WHILE statement

Unlike the FOR statement, WHILE loops do not rely on a loop counter. This type of loop is executed as long as a condition or expression is TRUE. It is important to ensure that the loop has a defined end so that a cycle time violation does not occur at runtime.


Keywords	Syntax
WHILE .. DO	WHILE i < 4 DO
	Res := value + 1;
	i := i + 1;
END_WHILE	END_WHILE

Table: Executing a WHILE statement

The statements are executed repeatedly as long as the condition is TRUE. If the condition returns FALSE the first time it is evaluated, then the statements are never executed.



Overview of the WHILE statement

 Programming \ Programs \ Structured Text (ST) \ WHILE statement

4.5.3 REPEAT statement

A REPEAT loop is different from the WHILE loop as the termination condition is checked only after the loop has executed. This means that the loop runs at least once, regardless of the termination condition.

Keywords	Syntax
REPEAT	REPEAT
	(*program code*)
	i := i + 1;
UNTIL	UNTIL i > 4
END_REPEAT	END_REPEAT

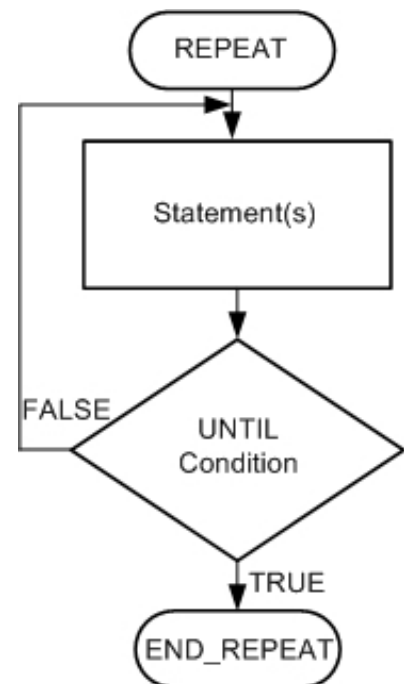
Table: Executing a REPEAT statement

Command groups

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



If the UNTIL condition never takes on the value TRUE, the statements are repeated infinitely, resulting in a runtime error.



Overview of the REPEAT statement



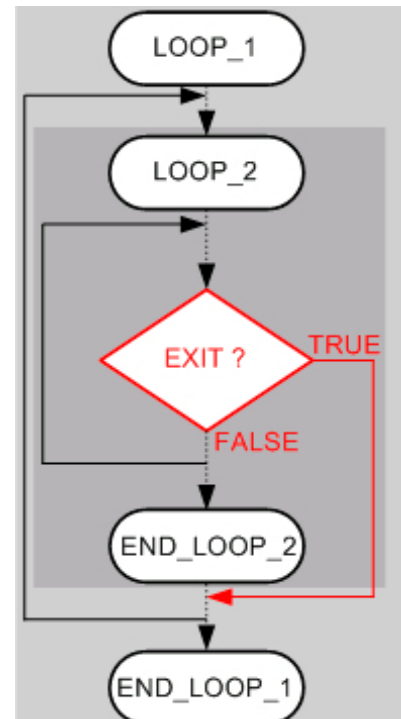
Programming \ Programs \ Structured Text (ST) \ REPEAT statement

4.5.4 EXIT statement

The EXIT statement can be used in all of the different types of loops before the termination condition is met. If EXIT is executed, the loop is terminated.

Keywords	Syntax
	REPEAT
	IF setExit = TRUE THEN
EXIT	EXIT ;
	END_IF
	UNTIL i > 5
	END_REPEAT

Once the EXIT statement in the loop has been executed, the loop is terminated, regardless of whether the termination condition or the loop's ending value of the loop has been reached. In nested loops, only the loop containing the EXIT statement is terminated.



The EXIT statement only terminates the inner loop.



Programming \ Programs \ Structured Text (ST) \ EXIT statement

Exercise: Searching with termination

A list of 100 random numbers should be searched to find one specific number. If the number 10 is found, the search process should be terminated. However, it's possible that the number is not in the list.

Use the REPEAT and EXIT statements in your solution. Keep the two terminating conditions in mind.

Declaration:

```

VAR
    aValues : ARRAY [0..99] OF INT;
END_VAR
  
```



The individual elements of arrays can either be pre-initialized with values in the program code or in the variable declaration window.



Programming \ Editors \ Table editors \ Declaration editors \ Variable declaration

5 FUNCTIONS, FUNCTION BLOCKS AND ACTIONS

Various functions and function blocks extend the functionality of a programming language. Actions are used to structure the program more effectively. Functions and function blocks can be inserted from the toolbar.



Inserting functions and function blocks from the toolbar

5.1 Calling functions and function blocks

Functions

Functions are subroutines that return a particular value when called. One way a function can be called is in an expression. The transfer parameters, also referred to as arguments, are the values that are passed to a function. They are separated by commas.


Declaration:

```
VAR
    sinResult : REAL;
    xValue : REAL;
END_VAR
```

Program code:

```
xValue := 3.14159265;
sinResult := SIN(xValue);
```

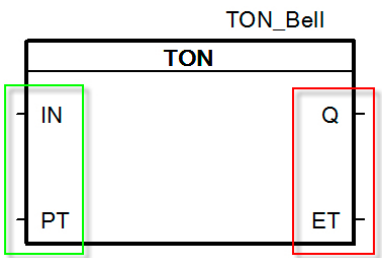
Table: Calling a function with one transfer parameter

 [Programming \ Programs \ Structured Text \(ST\) \ Calling functions](#)

Function blocks

A function block is characterized by the fact that it can take on several transfer parameters and return multiple results.

Unlike a function, it is necessary to declare an instance variable that has the same data type as the function block. The advantage here is that a function block can handle more complex tasks by calculating a result over a period of several cycles. By using different instances, several multiple function blocks of the same type can be called with different transfer parameters.



Transfer parameters (green) and results (red) for the TON() function block

When calling a function block, it is possible to pass all arguments or just some of them. Parameters and results can be accessed in the program code using the elements of the instance variable.

Declaration:	<pre> VAR diButton : BOOL; doBell : BOOL; TON_Bell : TON; END_VAR </pre>
Call variant 1:	<pre> TON_bell(IN := diButton, PT := T#1s); doBell := TON_bell.Q; </pre>
Call variant 2:	<pre> (*-- Parameters*) TON_bell.IN := diButton; TON_bell.PT := T#1s (*-- Call function block*) TON_bell(); (*-- Read results*) doBell := TON_bell.Q; </pre>

Table: Debouncing a button with the TON() function block

In the call for variant 1, all parameters are passed directly when the function block is called. In the call for variant 2, the parameters are assigned to the elements of the instance variable. In both cases, the desired result must be read from the instance variable after the call has taken place.



[Programming \ Programs \ Structured Text \(ST\) \ Calling function blocks](#)

Exercise: Function blocks

Call some of the function blocks in the STANDARD library. Prior to doing so, have a look at the function and parameter descriptions found in the Automation Studio online help.

- 1) Call the TON switch-on delay function block.
- 2) Call the CTU upward counter function block.



A detailed description of library functions can be found in the Automation Studio online help documentation. Pressing **<F1>** opens up the help documentation for the selected function block.

Many libraries also include different application examples that can be imported directly into the Automation Studio project.



[Programming \ Libraries \ IEC 61131-3 functions \ STANDARD](#)

[Programming \ Examples \ Libraries](#)

5.2 Calling actions

An action is program code that can be added to programs and libraries. Actions are an additional possibility for structuring programs and can also be formulated in a programming language other than the program which calls it. Actions are identified by their names.

Calling an action is very similar to calling a function. The difference is that there are no transfer parameters and a value is not returned.



If you use a CASE statement to control a complex sequence, for example, the contents of the individual CASE steps can be transferred on to actions. This keeps the main program compact. If the same functionality is needed elsewhere, then it's as easy as calling the action again.



Program:

```
CASE Sequence OF
  WAIT:
    IF cmdStartProc = 1 THEN
      Sequence := STARTPROCESS;
    END_IF

    STARTPROCESS:
      acStartProc; (*Machine startup*)

      IF Process_finished = 1 THEN
        Sequence := ENDPROCESS;
      END_IF

      ENDPROCESS:
        acEndProc; (*Machine shutdown*)
        (*...*)
    END_CASE
```

Action:

```
ACTION acStartProc:
  (*Add your sequence code here*)
  Process_finished := 1;
END_ACTION
```

Table: Calling actions in the main program



Actions

6 ADDITIONAL FUNCTIONS, AUXILIARY FUNCTIONS

6.1 Pointers and references

To extend the functionality of the existing IEC standard, B&R also offers pointers in ST. This allows a dynamic variable to be assigned a particular memory address during runtime. This procedure is referred to as referencing or the initialization of a dynamic variable.

Once the dynamic variable is initialized, it can be used to access to the contents of the memory address it is "pointing" to. For this operation, the keyword ACCESS is used.

Declaration:

```
VAR
    iSource : INT;
    pDynamic : REFERENCE TO INT;
END_VAR
```

Program code:

```
pDynamic ACCESS ADR(iSource);
(*pDynamic reference to iSource*)
```

Table: Referencing a pointer



Extended IEC standard functionality can be enabled in the project's settings in Automation Studio.

6.2 Preprocessor for IEC programs

In text-based programming languages, it is possible to use preprocessor directives. With regard to syntax, the statements implemented broadly correspond to those of the ANSI C preprocessor.

These preprocessor commands are an IEC expansion. They must be enabled in the project settings.

Preprocessor directives are used for the conditional compilation of programs or entire configurations. When turned on, compiler options enable functionality that affects how the compilation process takes place.

A description and full list of available commands can be found in the Automation Studio online help.



Project management \ The workspace \ General project settings \ Settings for IEC compliance
Programming \ Programs \ Preprocessor for IEC programs

7 DIAGNOSTIC FUNCTIONS

Comprehensive diagnostic tools make the entire programming process more efficient. Automation Studio includes several tools for troubleshooting high-level programming languages:

- Monitor mode
- Variable watch
- Line coverage
- Tooltips
- Debugger
- Cross reference list



Diagnostics and service \ Diagnostics tool \ Debugger

Diagnostics and service \ Diagnostics tool \ Variable watch

Diagnostics and service \ Diagnostics tool \ Monitors \ Programming languages in monitor mode
\ Line coverage

Diagnostics and service \ Diagnostics tool \ Monitors \ Text-based programming languages in
monitor mode

Project Management \ The workspace \ Output window \ Cross reference

8 EXERCISES

8.1 Box lift exercise

Exercise: Box lift

Two conveyor belts (**doConvTop**, **doConvBottom**) are being used to transport boxes to a lift.

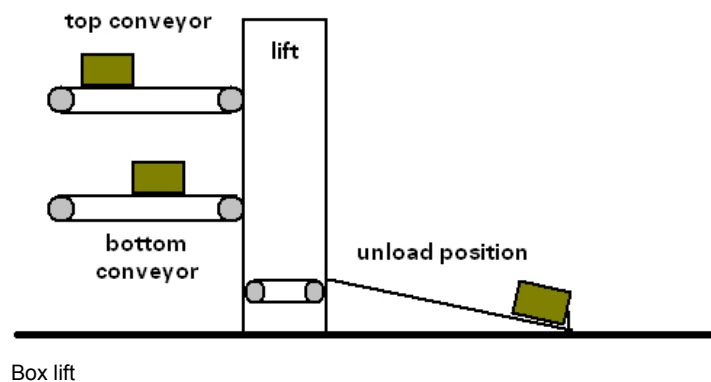
If the photocell (**diConvTop**, **diConvBottom**) is activated, the corresponding conveyor belt is stopped and the lift is requested.

If the lift is not being requested, it returns to the appropriate position (**doLiftTop**, **doLiftBottom**).

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

The lift then moves to the unloading position (**doLiftUnload**). When it reaches the position (**diLiftUnload**), the box is moved onto the unloading belt.

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



9 SUMMARY

Structured Text is a high-level programming language that offers a wide range of functionality. It contains everything necessary to create an application for handling a particular task. You now have an overview of the constructs and capabilities possible in ST.



The Automation Studio help documentation contains a description of all of these constructs. Remember, this programming language is especially powerful when using arithmetic functions and formulating mathematical calculations.

10 EXERCISE SOLUTIONS

Exercise: Light control

Declaration:	<pre> VAR btnLightOn: BOOL; btnLightOff: BOOL; doLight: BOOL; END_VAR </pre>
Program code:	<pre> doLight := (btnLightOn OR doLight) AND NOT bntLightOff; </pre>

Table: On-off switch, relay with latch

Exercise: Aquarium

Declaration:	<pre> VAR aiTemp1 : INT; aiTemp2 : INT; aoAvgTemp : INT; END_VAR </pre>
Program code:	<pre> aoAvgTemp := DINT_TO_INT((INT_TO_DINT(aiTemp1) + aiTemp2) / 2); </pre>

Table: Explicit data type conversion before addition and after division

Exercise: Weather station - Part 1

Declaration:	<pre> VAR aiOutside : INT; sShowText : STRING[80]; END_VAR </pre>
Program code:	<pre> IF aiOutside < 18 THEN sShowText := 'Cold'; ELSIF (aiOutside >= 18) AND (aiOutside <= 25) THEN sShowText := 'Opt'; ELSE sShowText := 'Hot'; END_IF; </pre>

Table: IF statement

Exercise: Weather station - Part 2

Declaration:

```
VAR
    aiOutside : INT;
    aiHumidity: INT;
    sShowText : STRING[80];
END_VAR
```

Program code:

```
IF aiOutside < 18 THEN
    sShowText := 'Cold';
ELSIF (aiOutside >= 18) AND (aiOutside <= 25) THEN
    IF (aiHumid >= 40) AND (aiHumid <= 75) THEN
        SShowText := 'Opt';
    ELSE
        SShowText := 'Temp. Ok';
    END_IF
ELSE
    sShowText := 'Hot';
END_IF;
```

Table: Nested IF statement

Exercise: Fill level exercise

Declaration:

```
VAR
    aiLevel : INT;
    PercentLevel : UINT;
    doLow : BOOL;
    doOk : BOOL;
    doHigh : BOOL;
    doAlarm : BOOL;
END_VAR
```

Table: CASE statement for querying values and value ranges

Program code:

```

(*Scaling the analog input to percent*)
PercentLevel := INT_TO_UINT(aiLevel / 327);
(*Reset all outputs*)
doAlarm := FALSE;
doLow := FALSE;
doOk := FALSE;
doHigh := FALSE;

CASE PercentLevel OF
  0:      (*-- Level alarm*)
    doAlarm := TRUE;
  1..24:  (*-- level is low*)
    doLow := TRUE;
  25..90: (*-- Level is ok*)
    doOk := TRUE;
  ELSE    (*-- Level is high*)
    doHigh := TRUE;
END_CASE

```

Table: CASE statement for querying values and value ranges

Exercise: Searching with termination**Declaration:**

```

VAR CONSTANT
  MAXNUMBERS : UINT := 99;
END_VAR
VAR
  aNumbers : ARRAY[0..MAXNUMBERS] OF INT;
  nCnt : INT;
END_VAR

```

Program code:

```

nCnt := 0;
REPEAT
  IF aNumbers[nCnt] = 10 THEN
    (*found the number 10*)
    EXIT ;
  END_IF
  nCnt := nCnt + 1;
UNTIL nCnt > MAXNUMBERS
END_REPEAT

```

Table: REPEAT statement, terminating search results, limiting cycles

Exercise: Crane

Declaration:

```
VAR CONSTANT
    MAXWEIGHT : UINT := 4;
END_VAR
VAR
    aWeights : ARRAY[0..MAXWEIGHT] OF INT;
    wCnt : INT;
    sumWeight : DINT;
END_VAR
```

Program code:

```
sumWeight := 0;
FOR wCnt := 0 TO MAXWEIGHT DO
    sumWeight := sumWeight + aWeights[wCnt];
END_FOR
```

Table: FOR statement, adding up weights

Exercise: Function blocks

Declaration:

```
VAR
    TON_Test : TON;
    CTU_Test : CTU;
    diSwitch : BOOL;
    diCountImpuls : BOOL;
    diReset : BOOL;
END_VAR
```

Program code:

```
TON_Test(IN := diSwitch, PT := T#5s);
CTU_Test(CU := diCountImpuls, RESET := diReset);
```

Table: Calling TON and CTU

Exercise: Box lift**Declaration:**

```

VAR CONSTANT
    WAIT :  UINT:= 0;
    TOP_POSITION :  UINT:= 1;
    BOTTOM_POSITION :  UINT:= 2;
    GETBOX :  UINT:= 3;
    UNLOAD_POSITION :  UINT:= 4;
    UNLOAD_BOX :  UINT:= 5;
END_VAR

VAR
    (*-- Digital outputs*)
    doConvTop:  BOOL;
    doConvBottom:  BOOL;
    doConvLift:  BOOL;
    doLiftTop:  BOOL;
    doLiftBottom:  BOOL;
    doLiftUnload:  BOOL;
    (*-- Digital inputs*)
    diConvTop:  BOOL;
    diConvBottom:  BOOL;
    diLiftTop:  BOOL;
    diLiftBottom:  BOOL;
    diLiftUnload:  BOOL;
    diBoxLift:  BOOL;
    (*-- Status variables*)
    selectLift:  UINT;
    ConvTopOn:  BOOL;
    ConvBottomOn:  BOOL;
END_VAR

```

Table: Possible solution to the box lift exercise

Program code:

```
doConvTop := NOT diConvTop OR ConvTopOn;
doConvBottom := NOT diConvBottom OR ConvBottomOn;
CASE selectLift OF
  (*-- Wait for request*)
  WAIT:
    IF (diConvTop = TRUE) THEN
      selectLift := TOP_POSITION;
    ELSIF (doConvBottom = TRUE) THEN
      selectLift := BOTTOM_POSITION;
    END_IF

  (*-- Move lift to top position*)
  TOP_POSITION:
    diLiftTop := TRUE;
    IF (diLiftTop = TRUE) THEN
      doLiftTop := FALSE;
      ConvBottomOn := TRUE;
      selectLift := GETBOX;
    END_IF
```

Table: Possible solution to the box lift exercise

```

(*-- 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

```

Table: Possible solution to the box lift exercise

TRAINING MODULES

TM210 – Working with Automation Studio
TM213 – Automation Runtime
TM220 – The Service Technician on the Job
TM223 – Automation Studio Diagnostics
TM230 – Structured Software Development
TM240 – Ladder Diagram (LD)
TM241 – Function Block Diagram (FBD)
TM242 – Sequential Function Chart (SFC)
TM246 – Structured Text (ST)
TM250 – Memory Management and Data Storage
TM261 – Closed Loop Control with LOOPCONR
TM400 – Introduction to Motion Control
TM410 – Working with Integrated Motion Control
TM440 – Motion Control: Basic Functions
TM441 – Motion Control: Multi-axis Functions
TM450 – ACOPOS Control Concept and Adjustment
TM460 – Initial Commissioning of Motors
TM480 – The Basics of Hydraulics
TM481 – Valve-based Hydraulic Drives
TM482 – Hydraulic Servo Pump Drives
TM500 – Introduction to Integrated Safety
TM510 – Working with SafeDESIGNER
TM530 – Developing Safety Applications
TM540 – Integrated Safe Motion Control
TM600 – Introduction to Visualization
TM610 – Working with Integrated Visualization
TM630 – Visualization Programming Guide
TM640 – Alarms, Trends and Diagnostics
TM670 – Advanced Visual Components
TM800 – APROL System Concept
TM811 – APROL Runtime System
TM812 – APROL Operator Management
TM813 – APROL XML Queries and Audit Trail
TM830 – APROL Project Engineering
TM890 – The Basics of LINUX
TM920 – Diagnostics and Service for End Users

