



TM246

Structured Text

Requirements

Training modules:	TM210 – Working with Automation Studio TM213 - Automation Runtime TM223 – Automation Studio diagnostics
Software	Automation Studio 3.0.90 or later
Hardware	None

Table of contents

1 Introduction.....	4
1.1 Learning objectives.....	4
2 General information.....	5
2.1 Structured Text features.....	5
2.2 Editor functions.....	5
3 Basic elements.....	7
3.1 Expressions.....	7
3.2 Assignments.....	7
3.3 Source code documentation - Comments.....	7
3.4 Operator priorities.....	8
3.5 Reserved keywords.....	9
4 Command groups.....	10
4.1 Boolean operators.....	10
4.2 Arithmetic operations.....	11
4.3 Data type conversion.....	12
4.4 Comparison operators and decisions.....	14
4.5 State machines - CASE statement.....	16
4.6 Loops.....	17
5 Functions, function blocks and actions.....	23
5.1 Calling functions and function blocks.....	23
5.2 Calling actions.....	24
6 Further functions, additional functions.....	26
6.1 Pointers and references.....	26
6.2 Preprocessor for IEC programs.....	26
7 Diagnostic functions.....	27
8 Exercises.....	28
8.1 Practice exercise - Box lift.....	28
8.2 Exercise - Dispersion mixer.....	29
9 Summary.....	30
10 Appendix.....	31
10.1 Arrays.....	31
10.2 Exercise solutions.....	33

1 Introduction

Structured Text is a high-level programming language. Elements from the languages BASIC, PASCAL and ANSI C were used for the concept. Thanks to its easily comprehensible standard constructs, Structured Text (ST) is a fast and efficient type of programming for the automation sector.



The following chapters provide information about commands, keywords and the syntax that can be used in Standard Text. These functionalities can be applied in simple examples to make them easier to understand.

1.1 Learning objectives

This training module uses selected exercises to help participants learn the basics of high-level language programming with Structured Text (ST).

- Participants will learn about the high-level language editor and Automation Studio's SmartEdit features.
- Participants will learn the basics of high-level language programming, as well as how to use Structured Text commands.
- Participants will learn how to use command groups and arithmetic functions.
- Participants will learn how to use comparison and Boolean operators.
- Participants will learn how to call the elements used to control program flow.
- Participants will learn how to work with reserved keywords.
- Participants will learn the difference between actions, functions and function blocks, as well as how to use them.
- Participants will learn how to use the diagnostic functions available for high-level language programming.

2 General information

2.1 Structured Text features

General information

ST is a textual high-level programming language for programming automation systems. Simple standard constructs allow a fast and efficient way of programming. ST makes use of many traditional characteristics of high-level programming languages, including the use of variables, operators, functions and elements for program flow control. The programming language ST is standardized according to IEC¹.

Properties

Structured Text is characterized by the following properties:

- Textual high-level programming language
- Structured programming
- Easy-to-use standard constructs
- Fast and efficient programming
- Self-explanatory and flexible use
- Similar to the language PASCAL
- Conforms to the IEC 61131-3 standard

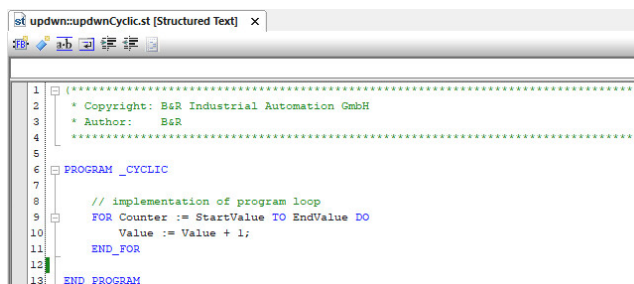
Possibilities

The following functions are supported in Automation Studio:

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

2.2 Editor functions

The editor is a text editor with many additional functions. Commands and keywords are displayed in color. Areas can be opened and closed. Autocomplete is integrated for variables and constructs (SmartEdit).



```

1 | *****
2 | * Copyright: B&R Industrial Automation GmbH
3 | * Author:   B&R
4 | *****
5 |
6 | PROGRAM _CYCLIC
7 |
8 |   // implementation of program loop
9 |   FOR Counter := StartValue TO EndValue DO
10 |     Value := Value + 1;
11 |   END_FOR
12 |
13 | END_PROGRAM
  
```

Figure 1: Application program in the ST editor

¹ The IEC 61131-3 standard is a worldwide valid standard for programming languages of programmable logic controllers. In addition to Structured Text, the programming languages Sequential Function Chart, Ladder Diagram, Instruction List and Function Block Diagram are defined.

The editor has the following functions and features:

- Differentiates between upper and lower case letters (case sensitive).
- Autocomplete (SmartEdit, <CTRL> <SPACE>, <TAB>)
- Manages and inserts code snippets (<CTRL> +q, k).
- Identifies corresponding pairs of brackets.
- Collapses or expands the construct (outlining).
- Inserts block comments.
- URL detection
- Markers for modified rows



Programming \ Editors \ Text editors
Programming \ Editors \ General operations \ SmartEdit
Programming \ Editors \ General operations \ Smart Edit \ Code snippets
Programming \ Programs \ Structured Text (ST)

The variable declaration editor supports the initialization of variables, constants, user data types and structures. In addition, the variables used can be commented on and thus documented. The declaration editors also support the SmartEdit function.



Programming \ Editors \ Table editors \ Declaration editors

3 Basic elements

The following chapter describes the basic elements of ST in more detail. Among other things, expressions, assignments, the use of comments in the program code and reserved keywords are explained.

3.1 Expressions

An expression is a construct that returns a value after it has been calculated. Expressions are made up of operators and operands. An operand can be a variable, constant or function call. The operators connect the operands ([3.4 "Operator priorities"](#)). Every expression, regardless of whether it is a function call or an assignment, must end with a semicolon (";").



Various different expressions

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

3.2 Assignments

The assignment consists of a variable on the left side, which is assigned with the assignment operator ":", the result of a calculation or an expression on the right side. All statements must end with a semicolon (";").



The assignment is done from right to left.

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

When the line of code has been processed, the value of the variable "Result" is twice the value of the variable "ProcessValue".

Bitwise access

With assignments, individual bits of variables can also be addressed. A dot (".") is placed behind the variable. The access is then carried out via the bit number, beginning with 0. Constants can also be used in place of the bit number.



Access to the second bit of "ProcessValue"

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

3.3 Source code documentation - Comments

Comments are an important part of the source code. They describe the code and make it easier to understand and read. Comments allow you or others to understand a program even after a long time. They are not compiled and have no influence on the program execution. Comments must be placed between a pair of parentheses and asterisks "(*comment*)".

An additional comment variant is introduced with "//". Multiple lines can be marked in the editor and commented out using an icon from the editor bar. This variant represents an extension to the existing IEC standard.

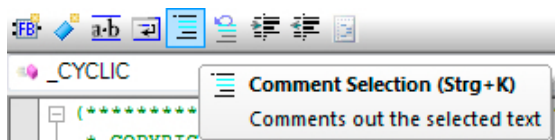


Figure 2: Setting text block as a comment

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

Table 1: Comment variants



Programming \ Editors \ Text editors \ Selecting or commenting out a line
 Programming \ Structured software development \ Program layout

3.4 Operator priorities

The use of different operators raises the question of precedence. The resolution of an expression is determined by the precedence (binding) of the operators.

Expressions are resolved according to the operators, starting with the highest precedence. Operators with the same precedence are resolved from left to right as written in the expression.

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

The expression is already being resolved by the compiler. The following examples show that different results can be achieved using parameters.

**Resolution of an expression without parentheses:**

```
// Result equals 38  
Result := 6 + 7 * 5 - 3;
```

The multiplication is performed first, followed by the addition. The subtraction is performed last.

Resolution of an expression with parentheses:

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

The expression is resolved from left to right. The operations in parentheses are calculated first, followed by the multiplication; this is because the content in the parentheses has a higher precedence than the multiplication. You can see that parentheses lead to different results.

3.5 Reserved keywords



All variables must follow the naming conventions in the programming. Furthermore, there are reserved keywords that are already recognized as such by the editor and are colored. These cannot be used as variables.

The OPERATOR and AslecCon libraries are part of the standard scope of a new project. The functions contained therein are IEC functions; these 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 represent the basic constructs of high-level language programming. These constructs can be flexibly combined and nested within one another.

A differentiation is made between the following command groups:

- [4.1 "Boolean operators"](#)
- [4.2 "Arithmetic operations"](#)
- [4.4 "Comparison operators and decisions"](#)
- [4.5 "State machines - CASE statement"](#)
- [4.6 "Loops"](#)

4.1 Boolean operators

Boolean operators can be used for the binary linking of variable values. A distinction is made between NOT, AND, OR and XOR. The operands do not necessarily have to be of data type BOOL. Operator priorities should be taken into consideration. Parentheses can be used.

Symbol	Logical operation	Examples are:
NOT	Binary negation	<code>a := NOT b;</code>
AND	Logical AND	<code>a := b AND c;</code>
OR	Logical OR	<code>a := b OR c;</code>
XOR	Exclusive OR	<code>a := b XOR c;</code>

Table 2: Overview of Boolean operators

The truth table for the 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 3: Truth table for Boolean operators

Boolean operators 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 operator - Comparison between Ladder Diagram and Structured Text

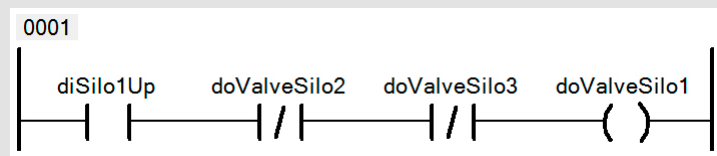


Figure 3: Linking normally open and closed contacts

Implementation with Boolean operators

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

No parentheses are needed since the NOT has a higher binding than the AND. For clearer programming, however, it is important to use parentheses.

Exercise: Lighting control

The output "DoLight" should be ON when the "ButtonLightOn" button is pressed and should remain ON until the "ButtonLightOff" button is pressed. Complete this exercise using Boolean operators.



Figure 4: On-Off switch, relay with latch

4.2 Arithmetic operations

A decisive advantage of high-level programming languages is the simple handling of arithmetic operations.

Overview of arithmetic operations

Structured Text provides basic arithmetic operations for the application. It is important to pay attention to the operator precedences in the application. Multiplication is performed before addition, for example. The desired behavior can be achieved by using parentheses.

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, integer division remainder	a := b MOD c;

Table 4: Overview of arithmetic operations

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

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

Table 5: Conversion implicitly performed by the compiler


The result value "*Error" stands for the compiler error message "Error 1140: Incompatible data types: Cannot convert REAL to INT." since it is not possible to assign the expression to the data type for the result.

4.3 Data type conversion

When programming, one is inevitably confronted with different types of data. In principle, you can mix the data types in the program. Different types of assignments are also possible. Caution is required, however.

Implicit data type conversion

Whenever an assignment should be made in the program code, the compiler checks the data types. Assignments are always made from right to left in the statement. The result must therefore find room in the result variable. A conversion from a small data type to a larger one is thus performed implicitly by the compiler without a request from the user. If an attempt is made to assign a large data type to a small data type, a compiler error occurs. An explicit data type conversion is required.



Depending on the compiler and constellation, an incorrect result can be obtained despite implicit data type conversion. A hazard may occur when assigning unsigned data types to signed data types. A value overflow may occur during addition or multiplication. This is platform dependent. No warning is issued by the compiler.



	Expression	Data types	Note
	<pre>// UINT USINT Result := Value;</pre>	UINT, USINT	Can easily be converted implicitly
	<pre>// INT UINT Result := Value;</pre>	INT, UINT	Be careful with negative numbers!
	<pre>// UINT USINT USINT Result := Value1 + Value2;</pre>	UINT, USINT, USINT	Risk of range overflow during addition

Table 6: Examples of implicit data type conversions

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. The declaration must be entered in this format in the text view for the program's VAR file.



There is already a risk of an overflow when the addition is carried out:

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

Table 7: An overflow can occur to the right of the assignment operator

In the case described, the result data type must be able to record the range of values that was increased by the addition. A larger result data type is therefore necessary. During addition, the data type must be a larger data type for at least one operand. The second operand is then implicitly converted 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 8: With explicit conversion an overflow cannot happen

On 32-bit platforms, the compiler converts the operands to 32-bit for calculation. In this case, no value overflow occurs during the addition.



Programming \ Variables 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 points. Create a program that calculates the average temperature and outputs it to an analog output. Don't forget that analog inputs and outputs must be data type INT!

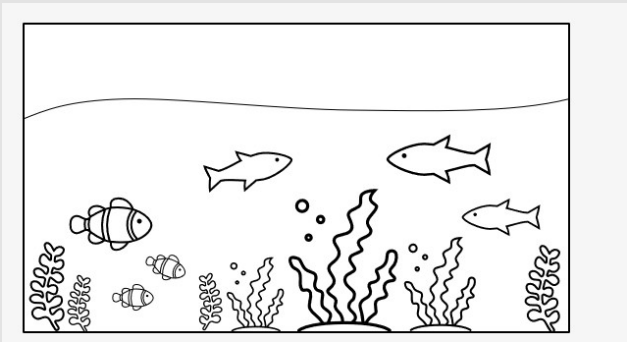


Figure 5: Aquarium

Declaration:	<pre>VAR aiTemperatureTop: INT; aiTemperatureBottom: INT; aoAverageTemperature: INT; END_VAR</pre>
---------------------	--

4.4 Comparison operators and decisions

Structured Text provides simple constructs for comparing variables. These return the value TRUE or FALSE. The comparison operators and logical operations are mainly used as conditions for statements such as IF, ELSIF, WHILE and UNTIL.

Symbol	Comparative expression	Example
=	Equal to	<code>IF a = b THEN</code>
<>	Not equal to	<code>IF a <> b THEN</code>
>	Greater than	<code>IF a > b THEN</code>
≥	Greater than or equal to	<code>IF a ≥ b THEN</code>
<	Less than	<code>IF a < b THEN</code>
≤	Less than or equal to	<code>IF a ≤ b THEN</code>

Table 9: Overview of the logical comparison operators

Decisions

The IF statement is used for decisions in the program. You are already familiar with the comparison operators. These can be used here.

Keywords	Syntax	Description
<code>IF .. THEN</code>	<code>IF a > b THEN</code>	1. Compare
	<code>Result := 1;</code>	Statement when 1st comparison is TRUE
<code>ELSIF .. THEN</code>	<code>ELSIF a > c THEN</code>	2. Compare
	<code>Result := 2;</code>	Statement when 2nd comparison is TRUE
<code>ELSE</code>	<code>ELSE</code>	Alternate branch when no comparison is TRUE
	<code>Result := 3;</code>	Statement of the alternate branch
<code>END_IF</code>	<code>END_IF</code>	End of the decision

Table 10: Syntax of the IF statement

Decisions are mapped in the program using IF statements. Comparison operators are used for this purpose. If a condition is true, the associated source code is executed. All other conditions are then no longer queried.

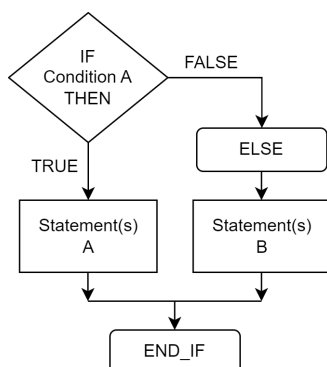


Figure 6: IF - ELSE statement

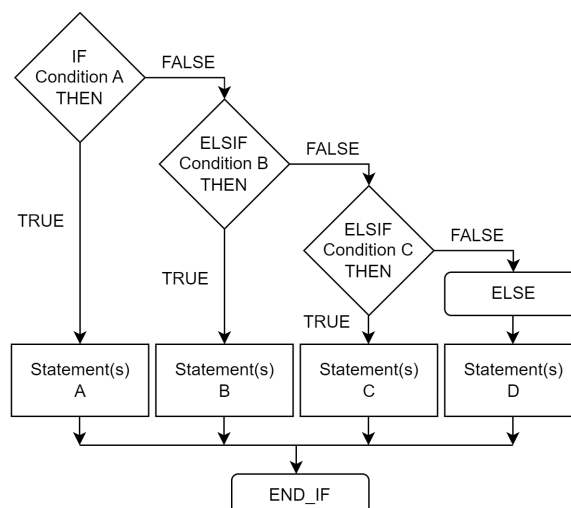


Figure 7: IF - ELSIF - Else statement

The comparison expressions can be linked with boolean operators so that several conditions can be queried simultaneously.





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

Table 11: Using multiple comparison expressions

A new IF statement can be embedded in any IF statement. It is important to ensure that there are not too many nesting levels; otherwise, the program becomes confusing.

 Further notes and guidelines can be found in TM231 - C310 rule.

 The editor's SmartEdit function can be used to simplify entering the code. If an IF statement is required, just type "IF" and press the <TAB> key. The basic framework of the IF statement is automatically added to the editor.

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

Exercise: Weather station - Part 1

A temperature sensor measures the outside temperature. The temperature is read via an analog input and should be displayed in text form in the house.

- 1) If the temperature is below 18°C, "Cold" should be displayed.
 - 2) If the temperature is between 18°C and 25°, "Optimal" should be displayed.
 - 3) If the temperature is above 25°C, then "Hot" should be displayed.
- Create a solution for this application by using IF, ELSIF and ELSE statements.

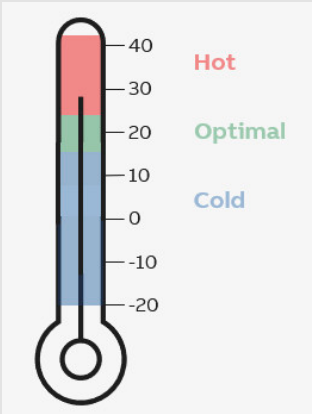



Figure 8: Thermometers

 To output a text, you need a variable with the data type STRING. The assignment can look like this: Show-Text := 'COLD';

Exercise: Weather station - Part 2

Evaluate the humidity in addition to the temperature.
The text "Optimal" should only appear if the humidity is between 40% and 75% and the temperature is between 18°C and 25°C; otherwise, "Temp. OK" should be displayed.
Extend your statement from Part I with a nested IF statement.



If several IF statements check the same variable value, it must be checked whether the request cannot be solved more elegantly and clearly with the CASE statement. Compared to the IF statement, the CASE statement has the further advantage that comparisons are only made once, thus creating more effective program code.

4.5 State machines - CASE statement

The CASE statement compares a variable with several values. If one of these comparisons is true, the statements associated with the step in question are executed. If none of these comparisons apply, there is an ELSE branch, similar to the IF statement, whose program code is processed in this case.

Depending on the application, the CASE statement is also used as a construct for the implementation of state machines.

Keywords	Syntax	Description
<code>CASE .. OF</code>	<code>CASE Step OF</code>	Start of CASE statement
	<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 (6..10 = range from 6 to 10)
<code>ELSE</code>	<code>ELSE</code>	Alternate branch
	<code>(*...*)</code>	
<code>END_CASE</code>	<code>END_CASE</code>	End of CASE

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

The step variable must be an integer data type.

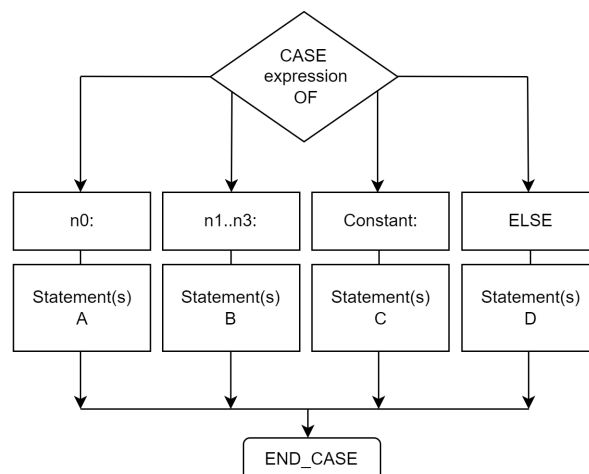


Figure 9: Overview - CASE statement



Constants or elements of enumeration data types should be used in the program code rather than fixed numerical values. Because a text is used in the program code instead of a value, it is easier to read. If values have to be changed in the program, a change in the declaration is necessary, but 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 \ Enumeration

Exercise: Fill-level control

The level in a container should be monitored in three ranges: low, ok and high.

The fill level of the tank is indicated on the outside by three lamps. The corresponding lamp lights up depending on the fill level. If the content drops below 1%, an acoustic alarm should also sound.

The level is read via an analog value (0 - 32767) and should be converted internally to 0-100%.

Create a solution for this application by using the CASE statement.

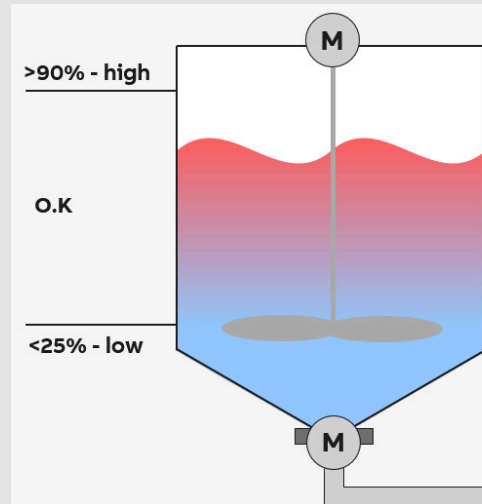


Figure 10: The fill level of the tank should be monitored.

Declaration:

```
VAR
  aiLevel : INT;
  PercentLevel : UINT;
  doLevelLow : BOOL;
  doLevelOk : BOOL;
  doLevelHigh : BOOL;
  doAlarm : BOOL;
END_VAR
```

4.6 Loops

In many applications, code parts must be processed repeatedly in the same cycle. This type of processing is also called "loop". The code in the loop is executed until a defined terminating condition is met.

Loops serve to make programs clearer and shorter. The expandability of programs is also an issue here.

Depending on how a program is structured, it could happen that an error in the program does not leave the loop until CPU time monitoring responds. In order to avoid such endless loops, a path must always be provided that terminates the loop after a defined number of repetitions.

Among other things, head and foot-controlled loops are often referred to.

Loops where control begins at the top (FOR, WHILE) check the terminating 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 cycled through at least once.

4.6.1 FOR statement

The FOR statement is used to execute a limited number of repetitions of a program part. The WHILE and REPEAT loops are used for applications where the number of cycles cannot be permanently defined.

Keywords	Syntax
FOR .. TO .. BY ² .. DO	FOR i:= StartValue TO StopValue BY Step DO
	Result := Result + 1;
END_FOR	END_FOR

Table 12: Elements of the FOR statement

The loop counter "Index (abbr.: i)" is preinitialized with the start value "Start-Value". The loop is repeated until the end value "StopValue" is reached. The loop counter is always increased by 1, or by "BY step". If a negative numerical value is used as step size "Step", the loop counter counts backwards. The loop counter, the start value and the end value must have the same whole number data type. This can also be achieved by explicit data type conversion (4.3 "Data type conversion").



If the start and end values are already the same at the beginning, this loop type is always run through at least once! (for example, if the start and end value are equal to 0)



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

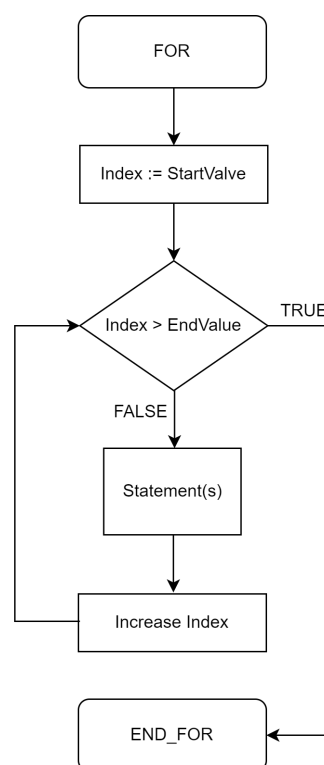


Figure 11: Overview - FOR statement

² Specifying the keyword "BY" is optional.

Exercise: Total crane load

There are five load receptors on one crane. The load receptors are each connected to an analog input and provide values in the range from 0 to 32767. To determine the total load and the average value, the individual loads must first be totaled and divided by the number of load receptors. Complete the exercise using a FOR statement.

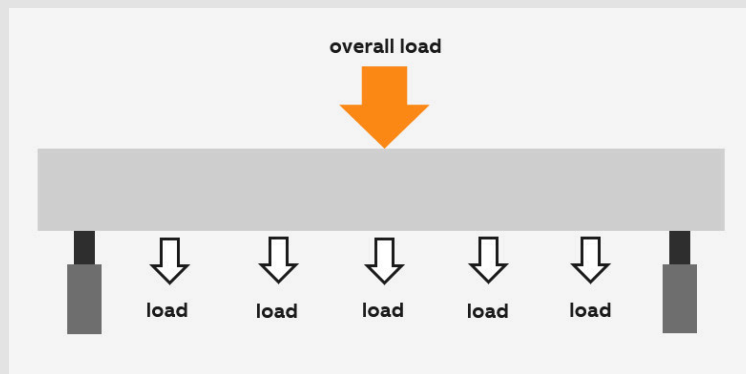


Figure 12: Crane with five load receptors

Declaration:

```
VAR
  Weights : ARRAY [0..4] OF INT;
  Counter : USINT;
  TotalWeight : DINT;
  AverageWeight : INT;
END_VAR
```



Arrays are required for completing this exercise. For additional information about arrays, see the appendix or Automation Help.

If possible, use constants for field declarations and for limiting loop end values. This improves the readability of the declarations and programs while making it easier to implement changes.



Programming \ Programs \ Variables and data types \ Data types \ Derived data types \ Arrays
See [10.1 "Arrays" on page 31](#).

Exercise: Total crane load - Improve the program code

As a result of the previous task, the sum of the individual loads could be calculated using a loop. Up to now, fixed numerical values have been included in the variable declaration and in the program code. The purpose of this exercise is to replace as many fixed numerical values as possible (from declaration and program code) with constants.

Declaration:

```
VAR CONSTANT
  MAX_INDEX : USINT := 4;
END_VAR
```

4.6.2 WHILE statement

Unlike the FOR statement, the WHILE loop does not have a loop counter. This loop type is called as long as a condition or expression is TRUE. It is important to ensure that the loop has an end so that no cycle time violation occurs at runtime.

Keywords	Syntax
WHILE .. DO	WHILE i < 4 DO
	Result := Value + 1;
	i := i + 1;
END_WHILE	END_WHILE

Table 13: Calling the WHILE statement

The statements are executed repeatedly as long as the condition is TRUE. If the condition is already FALSE during the first evaluation, the statements are never executed.

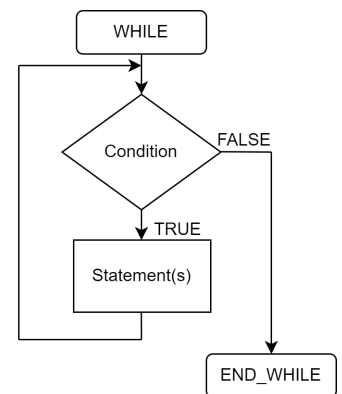


Figure 13: Overview - WHILE statement



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


4.6.3 REPEAT statement

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

Keywords	Syntax
REPEAT	REPEAT
	// program code
	i := i + 1;
UNTIL	UNTIL i > 4
END_REPEAT	END_REPEAT

Table 14: Calling the REPEAT statement

The statements are executed until the UNTIL condition is TRUE. If the UNTIL condition already returns TRUE during the first evaluation, the statements are executed one time only.



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

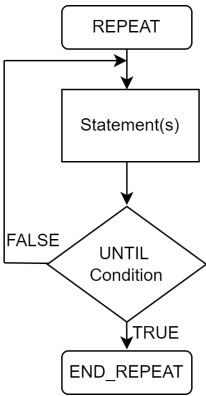



Figure 14: Overview of the REPEAT statement



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

4.6.4 EXIT statement

The EXIT statement can be used for all loop types before their terminating condition applies. If EXIT is called, the loop is aborted.

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

As soon as the EXIT statement is called in the loop, the loop is terminated, regardless of whether the terminating condition or the end value of the loop was reached. In nested loops, the system terminates the loop in which the EXIT statement occurs.

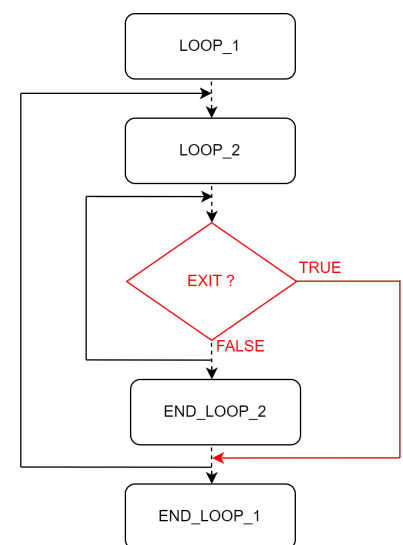


Figure 15: EXIT statement terminates the inner loop.



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

Exercise: Search with abort

A certain number should be selected from a list of 100 numbers. The list contains random numbers. If the number 10 is found, the search is aborted. It is possible, however, that the number is not available in the list. Use the REPEAT and EXIT statements for the solution. Pay attention to the two terminating conditions.

Declaration:

```

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



The individual elements of fields can be preinitialized with values in the program code or in the variable declaration.

5 Functions, function blocks and actions

Various functions and function blocks add system-specific functionality to a programming language. Actions are used to give the program a better structure. Functions and function blocks can be added using the toolbar.



Figure 16: Adding functions and function blocks via the menu bar

5.1 Calling functions and function blocks

Functions

Functions are like subroutines that return a value when called. A function can be called in an expression, for example. The command line parameters, also called arguments, are the values that are passed to a function. They are transferred separated by commas.

Declaration:	<pre>VAR SineResult : REAL; Value : REAL; END_VAR</pre>
Program code:	<pre>Value := 3.14159265; SineResult := SIN(Value);</pre>

Table 15: Calling the SIN() function with transfer parameter "Value"

Programming \ Programs \ Structured Text (ST) \ Calling functions

Function blocks

A function block is distinguished by the fact that they have multiple command line parameters and can return multiple results.

Unlike a function, a function block requires the declaration of an instance variable that corresponds to the data type of the function block. This has the advantage that a function block can also calculate a result over several cycles for more complex tasks. By using different instances, multiple function blocks of the same type can be called and supplied with different transfer parameters.

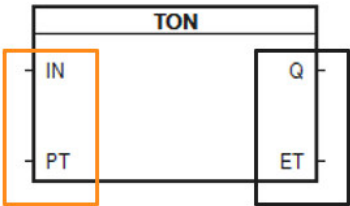


Figure 17: Passing parameters (orange) and results (black) of the TON() function block

When calling, it is possible to choose to transfer only some of the transfer parameters or all of them. The parameters and results can be accessed in the program code using the elements of the instance variable.


Declaration:	<pre>VAR diButton : BOOL; doBell : BOOL; Timer : TON; END_VAR</pre>
Call variant 1:	<pre>Timer(IN := diButton, PT := T#1s); doBell := Timer.Q;</pre>

Table 16: Debouncing a button with the TON() function block

Call variant 2:	<pre>// parameters Timer.IN := diButton; Timer.PT := T#1s; // call function block Timer(); // read results doBell := Timer.Q;</pre>
------------------------	---

Table 16: Debouncing a button with the TON() function block

In call variant 1, all parameters are transferred directly when the function block is called. In call 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 been made.


 Programming \ Programs \ Structured Text (ST) \ Calling function blocks


Exercise: Function blocks

Call some of the function blocks in the STANDARD library. Before doing so, have a look at the function and parameter descriptions found in Automation Help.

- 1) Call TON switch-on delay.
Setting the variable "diSwitch" should start the timer.
- 2) Call the CTU upward counter.
Each rising edge that results from setting the variable "diCountImpuls" should increase the upward counter by 1.
Setting the variable "diReset" should reset the CV output to 0.


Declaration:	<pre>VAR TestTimer : TON; TestCounter : CTU; diSwitch : BOOL; diCountImpulse : BOOL; diReset : BOOL; END_VAR</pre>
---------------------	--

 For a more detailed description of the library function, see Automation Help. Pressing <F1> opens the help documentation for the selected function block.
There are application examples for many libraries. These can be imported directly into the Automation Studio project.

 Programming \ Libraries \ IEC 61131-3 functions \ STANDARD
Programming \ Examples

5.2 **Calling actions**

An action is a program section that can be added to programs and libraries. They represent another way to structure programs and can be created in a programming language different from the program being called. Actions are identified by their name.
Calling an action is very similar to calling a function. There are no transfer parameters and no return value, however.

 If a CASE statement is used to control a more complex process, the content of the individual CASE steps can be outsourced to actions. This keeps the main program compact. If the same functionality is required again in another place, it simply needs to be called again.

	<pre>CASE Sequence OF WAIT: IF CmdStartProcess = 1 THEN Sequence := START_PROCESS; END_IF START_PROCESS: // machine startup StartProcess; IF ProcessDone = 1 THEN Sequence := END_PROCESS; END_IF END_PROCESS: // machine shutdown EndProcess; // ... END_CASE</pre>
Program:	
Action:	<pre>ACTION StartProcess: // add your sequence code here ProcessDone := 1; END_ACTION</pre>
Table 17: Calling actions in the main program	


6 Further functions, additional functions


6.1 Pointers and references

B&R offers pointers in ST as an extension to the existing IEC standard. A dynamic variable can be assigned a memory address at runtime. This process is called referencing or initializing a dynamic variable. As soon as the dynamic variable is initialized, it can be used to access the memory content that it now "points" to. The keyword ACCESS is used for this process.

Declaration:	<pre>VAR Source : INT; PointerDynamic : REFERENCE TO INT; END_VAR</pre>
Program code:	<pre>// PointerDynamic references to iSource PointerDynamic ACCESS ADR(Source);</pre>


Table 18: Referencing a pointer

 IEC standard extensions can be enabled in the project settings of Automation Studio.

 Programming\Variables and data types\Variables\Dynamic variables

6.2 Preprocessor for IEC programs

Using preprocessor directives is possible in text-oriented programming languages. The syntax of the implemented directives corresponds to a large extent to that of the ANSI C preprocessor. The preprocessor directives are an IEC extension. This must be enabled in the project settings. Preprocessor directives are used for the conditional compilation of programs or entire configurations. Compiler options can be used to enable options that have an influence on the compilation. A description and complete list of all available commands can be found in Automation Help.

 Project management \ Workspace \ General project settings \ IEC compliance settings
Programming \ Programs \ Preprocessor for IEC programs

7 Diagnostic functions

Only comprehensive diagnostic tools make programming efficient. Automation Studio provides several tools for program diagnostics in high-level programming languages:

- Monitor mode
- Watch window
- Line coverage
- Tooltips
- Debugger
- Cross-reference list



Diagnostics and service \ Diagnostics tool \ Debugger

Diagnostics and service \ Diagnostics tool \ Watch window

Diagnostics and Service \ Diagnostics tools \ Monitors mode \ Programming languages in monitor mode
\ Line coverage

Diagnostics and Service \ Diagnostics tools \ Monitors mode \ Programming languages in monitor mode
\ Powerflow

Project management \ Workspace \ Output window \ Cross reference

8 Exercises

8.1 Practice exercise - Box lift

Exercise: Box lift

The objective of the exercise is to analyze the process of the box lift and then to program it step by step in Structured Text.

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

If the photoelectric sensor (**diConvTop** or **diConvBottom**) detects a box, the corresponding conveyor belt is stopped and the lift is requested.

If the lift has not yet received a request, it is moved to the corresponding position (**doLiftTop**, **doLiftBottom**).

When the lift is in the requested position (**diLiftTop**, **diLiftBottom**), the lift conveyor (**doConvLift**) is switched on until the box is positioned correctly on the lift (**diBoxLift**).

Then the lift moves to the unloading position (**doLiftUnload**). Once it has reached the position (**diLiftUnload**), the box is lifted to the unloading conveyor.

As soon as the box is removed from the lift, it is ready for the next request.

The following steps must be carried out:

- 1) Outline the procedure, e.g. with steps, states and actions in the program structure
- 2) Implement the requirement in a Structured Text program.

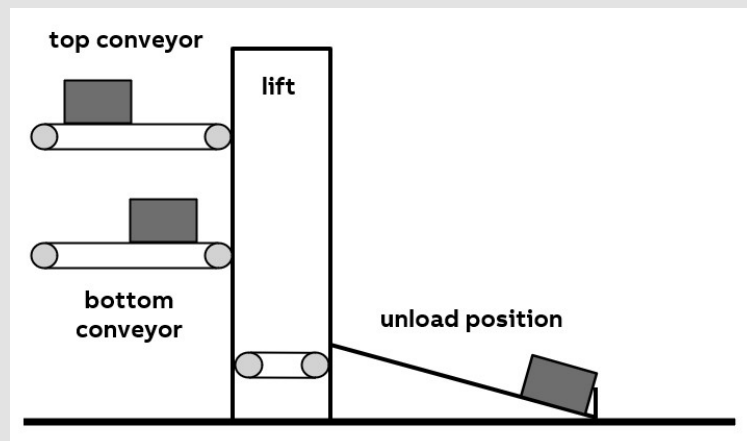


Figure 18: Box lift

8.2 Exercise - Dispersion mixer

A mixing plant must be configured. The elements water and paint will be mixed into a dispersion.

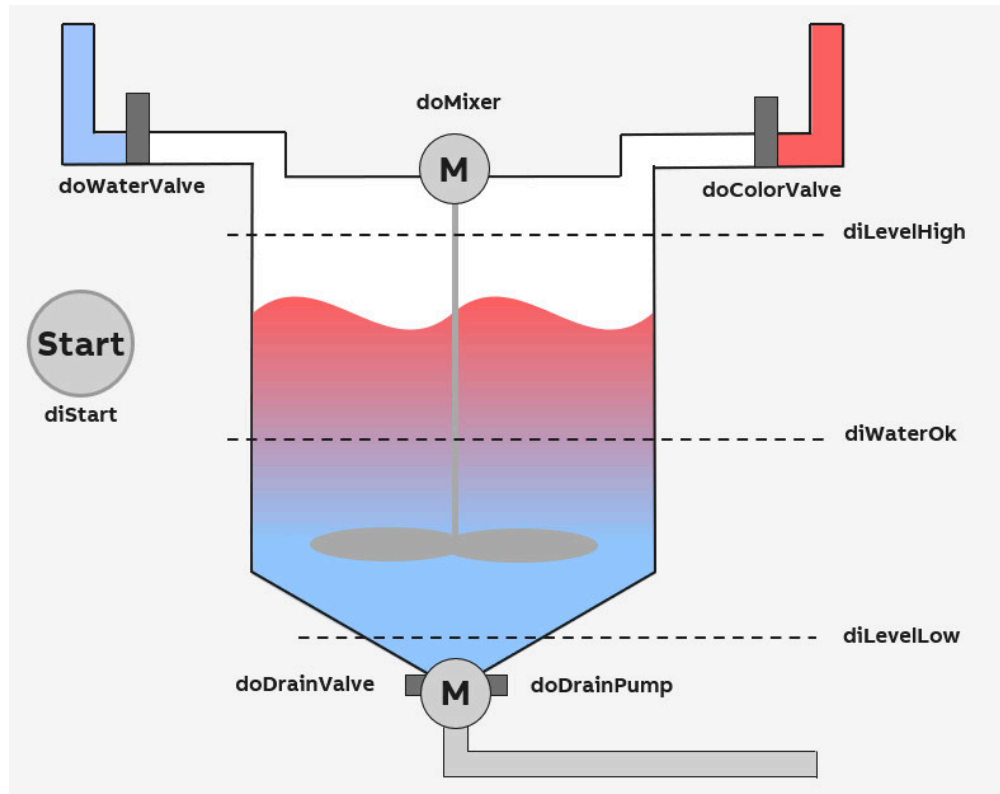


Figure 19: Schematics of a paint-mixing plant

The mixing program will run according to the following procedure:

- The mixing program waits until the start button is pressed (**diStart**).
- Water (**doWaterValve**) is filled into the container until the sensor "**diWaterOk**" is triggered.
- The mixing unit (**doMixer**) is started and paint (**doColorValve**) is filled into the container until the sensor "**diLevelHigh**" is triggered.
- It takes 30 seconds for the mixing time to elapse.
- The drain valve (**doDrainValve**) and drain pump (**doDrainPump**) are switched on for the filling process.
- The filling process ends when signal "**diLevelLow**" is triggered.
- The starting situation is restored.

Exercise: Implement the dispersion mixer

The exercise is to analyze the process of the dispersion mixer and then to program it step by step in Structured Text. The following steps must be carried out for this:

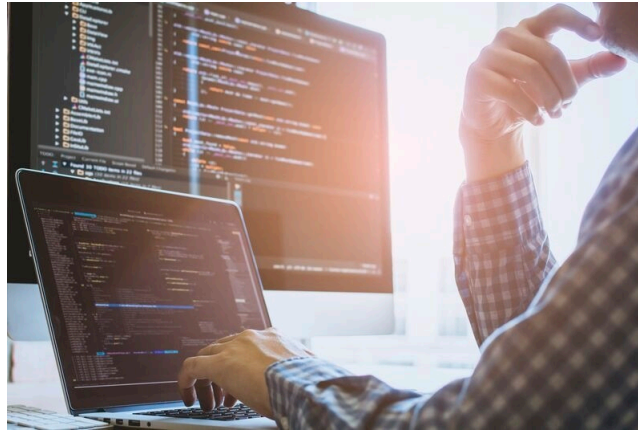
- 1) Outline the procedure, e.g. with steps, states and actions in the program structure
- 2) Implement the requirement in a Structured Text program.



A timer block from the STANDARD library must be used to implement the mixing time. It is important to ensure that the timer is reset after the time has expired.

9 Summary

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



Automation Help contains a description of all of these constructs. This programming language is especially powerful when using arithmetic functions and formulating mathematical calculations.

10 Appendix

10.1 Arrays

In contrast to basic data type variables, values with the same data type are combined in arrays. The individual elements can be addressed with the array name and an index.

The value of the array index is not permitted to exceed the array size. The size of an array is defined by the variable declaration.

In the program, the index can be a fixed value, a variable, a constant or an enumerated element.

Name	Type	Value
Pressure	INT[0..9]	
Pressure[0]	INT	123
Pressure[1]	INT	555
Pressure[2]	INT	0
Pressure[3]	INT	552
Pressure[4]	INT	32767
Pressure[5]	INT	9700
Pressure[6]	INT	0
Pressure[7]	INT	9
Pressure[8]	INT	0
Pressure[9]	INT	13

Figure 20: An array of data type INT with a range of 0 to 9 corresponds to 10 different array elements.

Declaring and using arrays

When an array is declared, it must be given a data type and a dimension. It is common that the smallest index of a field is 0. It is important to note that the maximum index for an array of 10 elements is 9.



Declaration

```
VAR
    Pressure : ARRAY[0..9] OF INT := [10(0)];
END_VAR
```

Program code

```
// Assigning value 123 to index 0
Pressure[0] := 123;
```

Table 19: Declaring an array of 10 elements, starting index = 0

If attempting to access an array element with index 10, the compiler outputs the following error message:

Program code

```
Pressure[10] := 75;
```

Error message

Error 1230: The constant value '10' is not in range '0..9'.

Table 20: Accessing an array index outside of the valid range



If an array of 10 elements should be declared, it can be done in the declaration editor with either "USINT[0..9]" or "USINT[10]"³. In both cases, a field is created with a start index of 0 and a maximum index of 9.

Exercise: Create the "Pressure" array

- 1) Add new "TestArray" program in the Logical View.
- 2) Open the variable declaration window.
- 3) Declare the "Pressure" array.

The array should contain 10 elements. The smallest array index is 0. The data type must be INT.


- 4) Use the array in program code.

Use the index to access the array in the program code. Use fixed numbers, constants and a variable for this.

³ This input method is only supported by the table editor in Automation Studio.

5) Force an invalid array access in the program code.

Access index value 10 of the array and then analyze the output in the message window.



When assigning `Pressure[10] := 123;`, the compiler reports the following error message.

Error 1230: The constant value '10' is not in range '0..9'.

The compiler is not able to check array access if the assignment is made using a variable.

```
Index := 10;
Pressure[Index] := 123;
```

In the worst case, this erroneous access can cause the system to crash because unauthorized access to the memory takes place.

Declaring an array using constants

Since using fixed numeric values in declarations and the program code itself usually leads to programming that is unmanageable and difficult to maintain, it is a much better idea to use numeric constants. The upper and lower indexes of an array can be defined using these constants. These constants can then be used in the program code to limit the changing array index.




	Declaration	<pre>VAR CONSTANT MAX_INDEX : USINT := 9; END_VAR VAR Pressure : ARRAY[0..MAX_INDEX] OF INT ; Index : USINT := 0; END_VAR</pre>
	Program code	<pre>IF Index > MAX_INDEX THEN Index := MAX_INDEX; END_IF Pressure[Index] := 75;</pre>

Table 21: Declaring a field using a constant and assigning values to a field element



The program code has now been updated so that the index used to access the array is limited to the maximum index of the array. An advantage of this is that arrays can be resized (larger or smaller) without having to make a change in the program code.



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

Exercise: Calculate sum and average value


The average value should be calculated from the contents of the "Pressure" array. The program has to be structured in such a way that the least amount of changes to the program are necessary when modifying the size of the array.

- 1) Calculate the sum using a loop.

Fixed numeric values are not permitted to be used in the program code.

- 2) Calculate the average value.

The data type of the average value must be the same as the data type of the array (INT).



A constant that is already being used to determine the array size for the declaration of an array variable can also be used as end value for the loop. When generating the average, the same constant can also be used for division. A data type that is larger than the output data type (e.g. DINT) must be used when adding up the individual array elements. The resulting value can then be converted back to the INT data type using an explicit data type conversion.

Optional exercise: Search for largest and smallest element

- 1) Create two new variables "Maximum" and "Minimum"
- 2) Find the smallest element in field "Pressure" and store its value in variable "Minimum".
- 3) Find the largest element in field "Pressure" and store its value in variable "Maximum".

Multidimensional arrays

Arrays can also be composed of several dimensions. The declaration and usage in this case can look something like this:


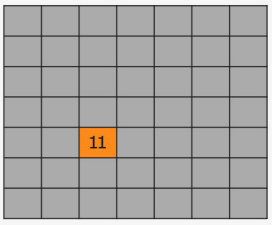


	Declaration	<pre>VAR 2DArray : ARRAY [0..6,0..6] OF INT; END_VAR</pre>	
	Program code	<pre>// Zählweise mit 0 beginnend 2DArray[4,2] := 11;</pre>	

Figure 21: Accessing the value in Column 2, Row 4

Table 22: Declaring and accessing a 7x7 two-dimensional array


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



An invalid attempt to access an array in the program code using a fixed number, a constant or an enumerated element will be detected and prevented by the compiler.

An invalid attempt to access an array in the program code using a variable cannot be detected by the compiler and may lead to a memory error at runtime. Runtime errors can be avoided by limiting the array index to the valid range.

The IEC Check library can be imported into an Automation Studio project to help locate runtime errors.

Programming \ Libraries \ IEC Check library

10.2 Exercise solutions

Exercise: Lighting control

Declaration:	<pre>VAR ButtonLightOn: BOOL; ButtonLightOff: BOOL; doLight: BOOL; END_VAR</pre>
Program code:	<pre>doLight := (ButtonLightOn OR doLight) AND NOT ButtonLightOff;</pre>

Table 23: On/Off button, relay with latch

Exercise: Aquarium

Declaration:	<pre> VAR aiTemperatureTop : INT; aiTemperatureBottom : INT; aoAverageTemperature : INT; END_VAR </pre>
Program code:	<pre> aoAverageTemperature := DINT_TO_INT((INT_TO_DINT (aiTemperatureTop) + aiTemperatureBottom) / 2); </pre>

Table 24: Explicit data type coding before addition, after division



On 32-bit platforms (e.g. X20CP1586), the compiler converts the operands to 32-bit for calculation. In this case, no value overflow occurs during an addition.

This is the case here, for example. On an X20CP1586, it will make no difference whether the explicit data type conversion is performed or not.

For the calculation "aiTemperatureTop := (aiTemperatureTop + aiTemperatureBottom) / 2;", the intermediate result of the content in parentheses is stored in the variable "aiTemperatureTop" before being divided by 2.

This could not work on a 16-bit system, however. An explicit data type conversion would therefore have to be performed in order to obtain the correct result.

Exercise: Weather station - Part 1

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

Table 25: IF statement

Exercise: Weather station - Part 2

Declaration:	<pre> VAR aiOutsideTemperature : INT; aiHumidity: INT; ShowText : STRING[80]; END_VAR </pre>
Program code:	<pre> IF aiOutsideTemperature < 18 THEN ShowText := 'Cold'; ELSIF (aiOutsideTemperature >= 18) AND (aiOutsideTemperature <= 25) THEN IF (aiHumidity >= 40) AND (aiHumidity <= 75) THEN ShowText := 'Optimal'; ELSE ShowText := 'Temperature Ok'; END_IF ELSE ShowText := 'Hot'; END_IF; </pre>

Table 26: Nested IF statement

Exercise: Fill-level control exercise

Declaration:	<pre> VAR aiLevel : INT; PercentLevel : UINT; doLevelLow : BOOL; doLevelOk : BOOL; doLevelHigh : BOOL; doAlarm : BOOL; END_VAR </pre>
Program code:	<pre> // scaling the analog input to percent PercentLevel := INT_TO_UINT(aiLevel / 327); // reset all outputs doAlarm := FALSE; doLevelLow := FALSE; doLevelOk := FALSE; doLevelHigh := FALSE; CASE PercentLevel OF 0: // -- level alarm doAlarm := TRUE; doLevelLow := TRUE; 1..24: // -- level is low doLevelLow := TRUE; 25..90: // -- level is ok doLevelOk := TRUE; ELSE // -- level is high doLevelHigh := TRUE; END_CASE </pre>

Table 27: CASE statement for querying values and value ranges

Exercise: Total crane load

Declaration:	<pre> VAR CONSTANT MAX_INDEX: USINT := 4; END_VAR VAR Weights : ARRAY[0..MAX_INDEX] OF INT; Counter : USINT; TotalWeight : DINT; AverageWeight : INT; END_VAR </pre>
Program code:	<pre> TotalWeight := 0; FOR Counter := 0 TO MAX_INDEX DO TotalWeight := TotalWeight + Weights[Counter]; END_FOR AverageWeight := DINT_TO_INT (TotalWeight / (MAX_INDEX + 1)); </pre>

Table 28: FOR - Statement, totaling the weights

Exercise: Search with abort

Declaration:	<pre> VAR CONSTANT MAX_NUMBERS : UINT := 99; END_VAR VAR Numbers : ARRAY[0..MAX_NUMBERS] OF INT; Counter : INT; END_VAR </pre>
Program code:	<pre> Counter := 0; REPEAT IF Numbers[Counter] = 10 THEN // found the number 10 EXIT; END_IF Counter := Counter + 1; UNTIL Counter > MAX_NUMBERS END_REPEAT </pre>

Table 29: REPEAT statement, aborting via search result, limitation

Exercise: Function blocks

Declaration:	<pre> VAR TestTimer : TON; TestCounter : CTU; diSwitch : BOOL; diCountImpuls : BOOL; diReset : BOOL; END_VAR </pre>
Program code:	<pre> TestTimer(IN := diSwitch, PT := T#5s); TestCounter(CU := diCountImpuls, RESET := diReset); </pre>

Table 30: Calling TON and CTU

Exercise: Box lift

Declaration:	<pre>VAR CONSTANT WAIT : UINT:= 0; TOP_POSITION : UINT:= 1; BOTTOM_POSITION : UINT:= 2; GET_BOX : UINT:= 3; UNLOAD_POSITION : UINT:= 4; UNLOAD_BOX : UINT:= 5; END_VAR (*Digital inputs*) VAR doConveyorTop: BOOL; doConveyorBottom: BOOL; doConveyorLift: BOOL; doLiftTop: BOOL; doLiftBottom: BOOL; doLiftUnload: BOOL; END_VAR (*Digital inputs*) VAR diConveyorTop: BOOL; diConveyorBottom: BOOL; diLiftTop: BOOL; diLiftBottom: BOOL; diLiftUnload: BOOL; diBoxLift: BOOL; END_VAR (*Status variables*) VAR SelectLift: UINT; ConveyorTopOn: BOOL; ConveyorBottomOn: BOOL; END_VAR</pre>
--------------	--

Table 31: Suggested solution for the box lift

Program code:

```

doConveyorTop := NOT diConveyorTop OR ConveyorTopOn;
doConveyorBottom := NOT diConveyorBottom OR ConveyorBottomOn;

CASE SelectLift OF
  // Wait for request
  WAIT:
    IF diConveyorTop THEN
      SelectLift := TOP_POSITION;
    ELSIF (diConveyorBottom = TRUE) THEN
      SelectLift := BOTTOM_POSITION;
    END_IF

  // Move lift to top position
  TOP_POSITION:
    doLiftTop := TRUE;
    IF diLiftTop THEN
      doLiftTop := FALSE;
      ConveyorTopOn := TRUE;
      SelectLift := GET_BOX;
    END_IF

  // Move lift to bottom position
  BOTTOM_POSITION:
    doLiftBottom := TRUE;
    IF diLiftBottom THEN
      doLiftBottom := FALSE;
      ConveyorBottomOn := TRUE;
      SelectLift := GET_BOX;
    END_IF

  // Move box to lift
  GET_BOX:
    doConveyorLift := TRUE;
    IF diBoxLift THEN
      doConveyorLift := FALSE;
      ConveyorTopOn := FALSE;
      ConveyorBottomOn := FALSE;
      SelectLift := UNLOAD_POSITION;
    END_IF

  // Move lift to unload position
  UNLOAD_POSITION:
    doLiftUnload := TRUE;
    IF diLiftUnload THEN
      doLiftUnload := FALSE;
      SelectLift := UNLOAD_BOX;
    END_IF

  // Unload the box
  UNLOAD_BOX:
    doConveyorLift := TRUE;
    IF NOT diBoxLift THEN
      doConveyorLift := FALSE;
      SelectLift := WAIT;
    END_IF
END_CASE

```

Table 31: Suggested solution for the box lift

Exercise: Implement the dispersion mixer**Declaration**

```

(*Digital inputs*)
VAR
    diStart : BOOL := FALSE;
    diWaterOk : BOOL;
    diLevelHigh : BOOL := FALSE;
    diLevelLow : BOOL := FALSE;
END_VAR

(*Digital outputs*)
VAR
    doWaterValve : BOOL := FALSE;
    doColorValve : BOOL := FALSE;
    doMixer : BOOL := FALSE;
    doDrainPump : BOOL := FALSE;
    doDrainValve : BOOL := FALSE;
END_VAR

(*Function block instances*)
VAR
    MixerTimer : TON;
END_VAR

(*State machine variables*)
VAR
    MixerState : MixerStateEnum;
END_VAR

```

Program code

```

// Reset all outputs - will be set in the individual states
doWaterValve := FALSE;
doColorValve := FALSE;
doMixer := FALSE;
doDrainValve := FALSE;
doDrainPump := FALSE;
MixerTimer.IN := FALSE;

// Implementation of dispersion mixer state machine
CASE MixerState OF
    // Wait until operator starts process by start button
    WAIT_FOR_START:
        IF diStart THEN
            MixerState := FILL_WATER;
        END_IF

    // Fill water into the reservoir until limit is reached
    FILL_WATER:
        IF diWaterOk THEN
            MixerState := FILL_COLOR;
        END_IF
        doWaterValve := TRUE

```

```
// Add color and mix dispersion until reservoir is full
FILL_COLOR:
    IF diLevelHigh THEN
        MixerState := MIX_TIME;
    END_IF
    doColorValve := TRUE;
    doMixer := TRUE;

// Mix color and water until time is elapsed
MIX_TIME:
    MixerTimer.IN := TRUE;
    MixerTimer.PT := T#30s;
    IF MixerTimer.Q THEN
        MixerState := BOTTLE_DISPERSION;
    END_IF
    doMixer := TRUE;

// Continue mixing and bottle dispersion until reservoir is empty
BOTTLE_DISPERSION:
    IF diLevelLow = TRUE THEN
        MixerState := WAIT_FOR_START;
    END_IF
    doDrainValve := TRUE;
    doDrainPump := TRUE;
    doMixer := TRUE;
END_CASE

// Call timer function block
MixerTimer();
```


Automation Academy

Your knowledge advantage

The Automation Academy provides **targeted training** courses for our customers as well as for our own employees. Expand your skills in the field of automation technology and learn to independently implement **efficient automation solutions** using B&R systems.

Decide for yourself which **learning concept** you prefer!



Classroom Learning



Virtual Classroom
Learning



Online courses

Classroom learning

B&R offers **standard seminars** at all B&R locations. Services include seminar documents, effective communication of learning content by experienced trainers and an Automation Diploma. A combination of group work and self-study provides the high level of flexibility needed to maximize the learning experience.

Virtual classroom

Remote Lectures supplement B&R's continuing education portfolio with a virtual classroom, offering an alternative to our on-site seminars. Selected content from our standard seminars is offered online. In addition to remote learning methods, powerful simulation tools and secure remote maintenance are used.

Online courses

Take control of the content and learn at your own pace. With B&R **online courses**, you can take your first steps in the world of B&R automation at any time. Based on a comprehensive narrative, you will independently work out how to use our products. The mix of different media allows a logical sequence to be followed when learning as well as a targeted choice of information to be used as a reference.

Contact

Would you like additional training? Are you interested in finding out what the B&R Automation Academy has to offer? If so, this is the right place.

Access additional information here:

<https://www.br-automation.com/de/academy/>

Enjoy your next training course!



AutomationAcademy





B&R
Industrial Automation GmbH
A member of the ABB Group
B&R Straße 1
5142 Eggelsberg, Austria
office@br-automation.com

t +43 7748 6586-0
f +43 7748 6586-26

br-automation.com



TM246TRE.001-ENG

V2.0.0.0 ©2023/08/03 by B&R, All rights reserved.
All registered trademarks are the property of their respective owners.
Subject to technical changes without notice.

