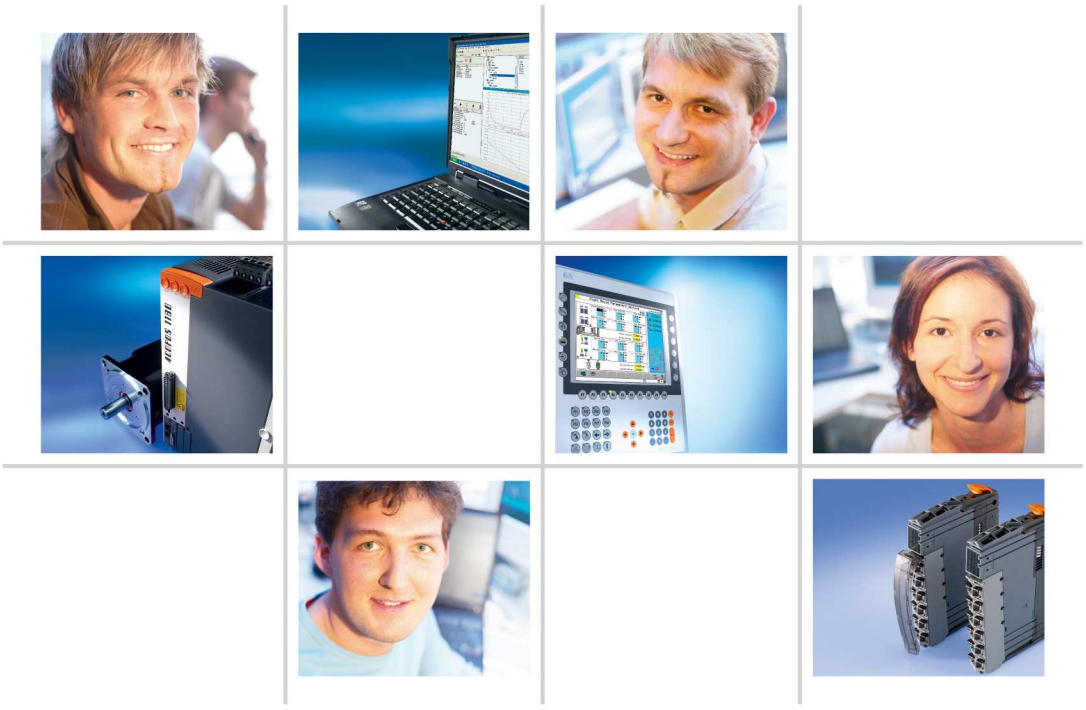
Automation Basic (AB)

TM247



Perfection in Automation

www.br-automation.com

Requirements

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

Table of contents

1. INTRODUCTION

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

Fig. 1 Book printing: then and now

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

1.1 Objectives

Participants will get to know the programming language Automation Basic (AB) for programming technical applications.

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

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

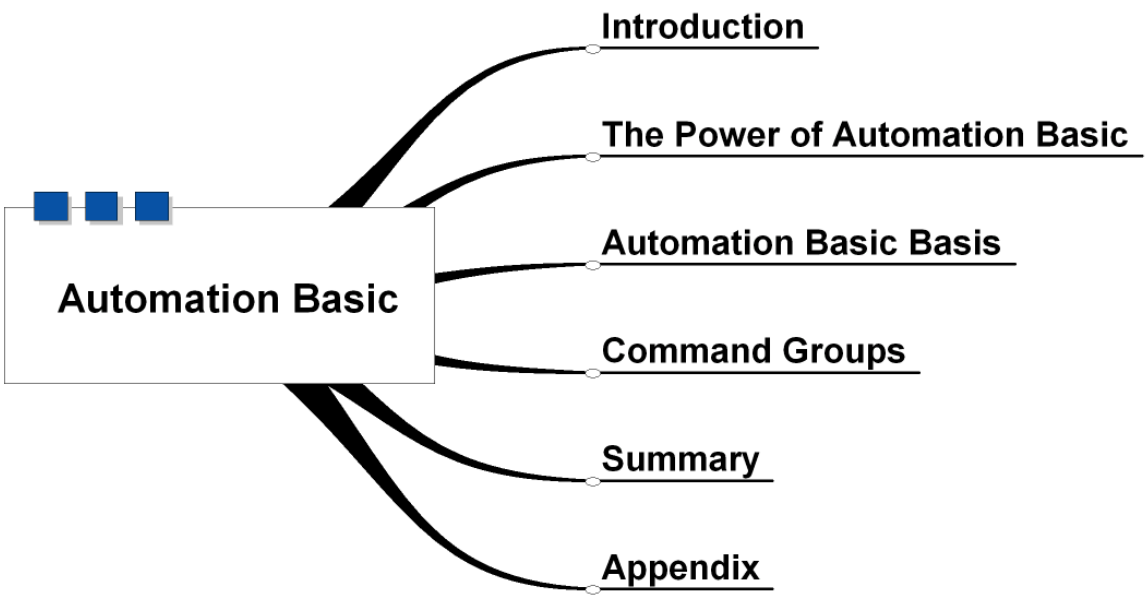


Fig. 2 Overview

2. AUTOMATION BASIC FEATURES

2.1 General information

Automation Basic is a text-based high-level language specially developed for programming modern automation tasks.

The language constructs mostly correspond to the ones used for Structured Text according to IEC61131-3 since Automation Basic is based on the standards from the late 1980's.

Automation Basic originated from the programming language PL2000 and is therefore used as its successor.

In addition, AB has been substantially expanded to include elements of a modern programming language.

2.2 Properties

Automation Basic is characterized by the following features:

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

2.3 Possibilities

Automation Studio supports the following functions:

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

3. THE FUNDAMENTALS OF AUTOMATION BASIC

3.1 Expressions

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

Example: Expressions

```
b + c

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

sin(a) * cos(b)
```

Fig. 3 Expressions

3.2 Assignment

The assignment of a value to a variable through a result of an expression or a value. The assignment consists of a variable on the left side, which is designated to the result of a calculation on the right side by the assignment operator "=". Alternatively, the assignment operator ":=" can also be used as in Structured Text. Assignments do not have to be closed with a semicolon ";".

Example: Assignment

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

Fig. 4 Assignment

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

3.3 Line break

It is possible to divide a line of code over several editor lines. For example, this might be useful to make the code more organized and easier to read.

The line break is made using the "\" character because delimiters do not have to be used in AB.

Example: Multi-line assignment

```
bOutput = bSwitch1 AND \
          ( bEmergencyStop1 OR bEmergencyStop2 ) AND \
          bLevelHigh
```

Fig. 5 Multi-line assignment

3.4 Comments

Comments describe the code and make it easier to understand. Comments make it possible for you or others to read a program easily, even long after it was written. They are not compiled and have no influence over the execution of the program. Comments can be single or multi-line.

Example: Comment

```
; This is one line comment
```

Fig. 6 Single-line comment

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

Fig. 7 Multi-line comment

3.5 Operator Priorities

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

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

Operator	Symbol / Syntax:	
Parentheses	()	Highest priority
Function call Examples	Call argument(s) LN(A), MAX(X), etc.	
Exponent	EXP(IN1,IN2)	
Negation	NOT	
Multiplication Division Modulo division (whole number remainder of division)	* / MOD	
Addition Subtraction	+ -	
Boolean AND	AND	
Boolean exclusive OR	XOR	
Boolean OR	OR	
Equal to Not equal to	= <>	
Comparisons	<= >= < >	Lowest priority

The order of execution at runtime:

Example: Operator priorities without parentheses

```
Result = 6 + 7 * 5 - 3  ; The multiplication first, higher priority
Result = 6 +   35  - 3  ; then addition; rule from left to right
Result = 41        - 3  ; Subtraction at the end
Result = 38
```

Fig. 8 Order of execution

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

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

Example: Operator priorities with parentheses

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

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

Fig. 9 Order of execution

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

4. COMMAND GROUPS

Automation Basic uses the following command groups

- Boolean operations (logical operations)
- Arithmetic operations
- Comparison operations
- Decisions
- Step sequencers

4.1 Boolean operations

The operands must not necessarily be the data type BOOL.

Boolean operations:

Symbol	Logical operation	Examples
NOT	Binary negation	`a = NOT b`
AND	Logical AND	`a = b AND c`
OR	Logical OR	`a = b OR c`
XOR	Exclusive OR	`a = b XOR c`

Truth table:

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

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

Example: Boolean operation

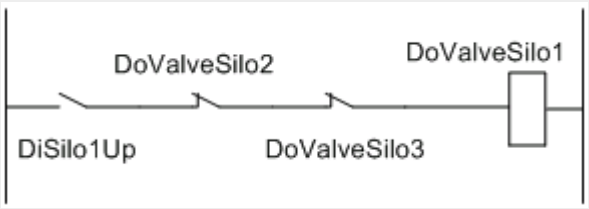


Fig. 11 AND operation

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

Fig. 11 AND operation source code

AB allows any number of parenthesis.

Exercise: Light control

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

Create a solution for this task using boolean operations in Automation Basic.

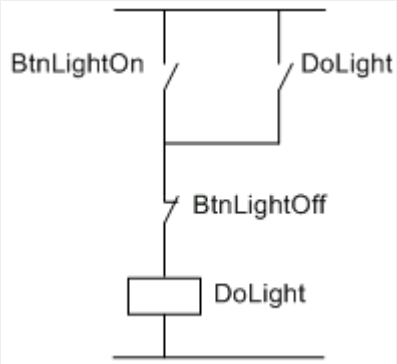


Fig. 12 Light control

4.2 Arithmetic operations

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

4.2.1 Basic arithmetic operations

Automation Basic provides basic arithmetic operations for your 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 (display division remainder)	a = b mod c

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

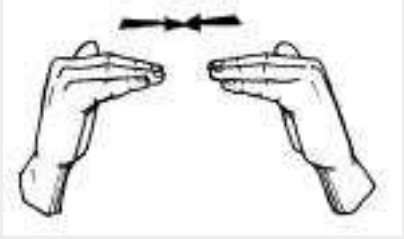
Syntax	Data types			Result
	Res	Op 1	Op 2	
Res = 8 / 3	INT	INT	INT	2
Res = 8 / 3	REAL	INT	INT	2.0
Res = 8.0 / 3	REAL	REAL	INT	2.6667
Res = 8.0 / 3	INT	REAL	INT	*Error

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

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

Note:

Left data type = right data type;



4.2.2 Implicit data type conversion

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

Data type	BOOL	SINT	INT	DINT	USINT	UINT	UDINT	REAL
BOOL	BOOL	x	x	x	x	x	x	x
SINT	x	SINT	INT	DINT	USINT	UINT	UDINT	REAL
INT	x	INT	INT	DINT	INT	UINT	UDINT	REAL
DINT	x	DINT	DINT	DINT	DINT	UDINT	UDINT	REAL
USINT	x	USINT	INT	DINT	USINT	UINT	UDINT	REAL
UINT	x	UINT	UINT	DINT	UINT	UINT	UDINT	REAL
UDINT	x	UDINT	UDINT	UDINT	UDINT	UDINT	UDINT	REAL
REAL	x	REAL	REAL	REAL	REAL	REAL	REAL	REAL

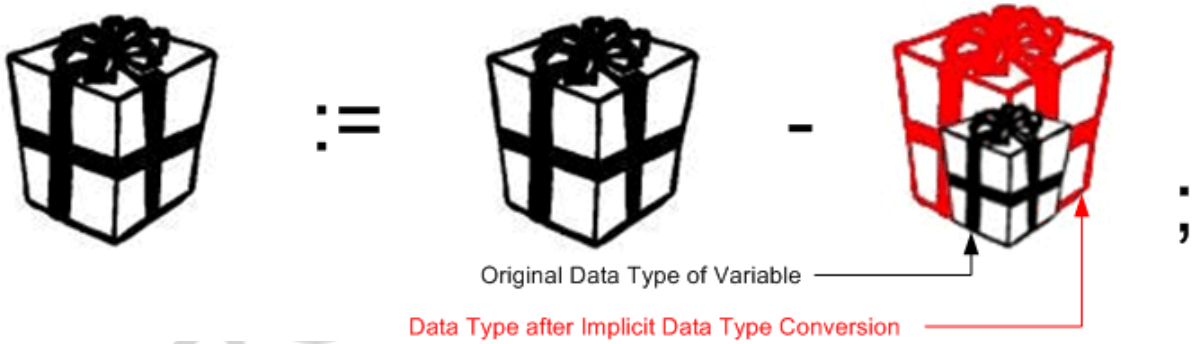


Fig. 13 Implicit data type conversion by the compiler

Example: Data conversion

```
INT_Result = INT_Var1 + SINT_Var2
;   [INT]        [INT]      [SINT]
```

Fig. 14 Implicit data type conversion

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

4.2.3 Explicit data type conversion

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

Example: Overflow?!

```
INT_TotalWeight = INT_Weight1 + INT_Weight2
;      [INT]           [INT]          [INT]
```

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

Example: Overflow taken into consideration.

```
DINT_TotalWeight = DINT(INT_Weight1) + INT_Weight2
;      [DINT]           [DINT]              [INT]
```

The variable DINT_TotalWeight must be the data type DINT. At least one variable on the right side of the expression must be converted to the data type DINT.

Exercise: Aquarium

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

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

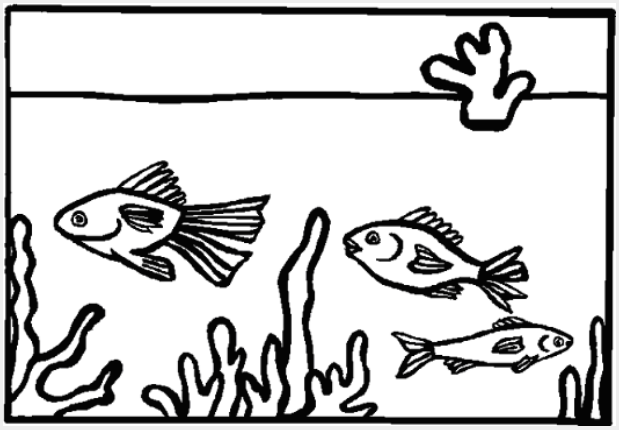


Fig. 15 Aquarium

4.3 Comparison operators

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

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

Note:

The comparison operations and boolean operations are used as logical conditions for IF, ELSE IF, EXITIF and WHEN statements.
(IF (a > b) AND (c >=d) THEN)

4.4 Decisions

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

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

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

4.4.1 IF

This is the most simple IF statement.

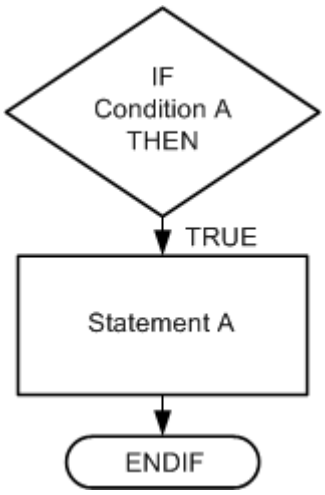


Fig. 16 Simple IF statement

```
;*******************************
;******** SINGLE IF ************
;*******************************
IF V1 > V2 THEN
    V3 = 99          ; comment
ENDIF
```

Fig. 17 Simple IF statement in a program

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

Example: IF statement with multiple comparisons

```
if ( ((Userlevel > 10) or (diKeySwitch = True)) and (operationMode = 0) ) then
    LedEdit= True
endif
```

Fig. 18 Statement with multiple comparisons

4.4.2 ELSE

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

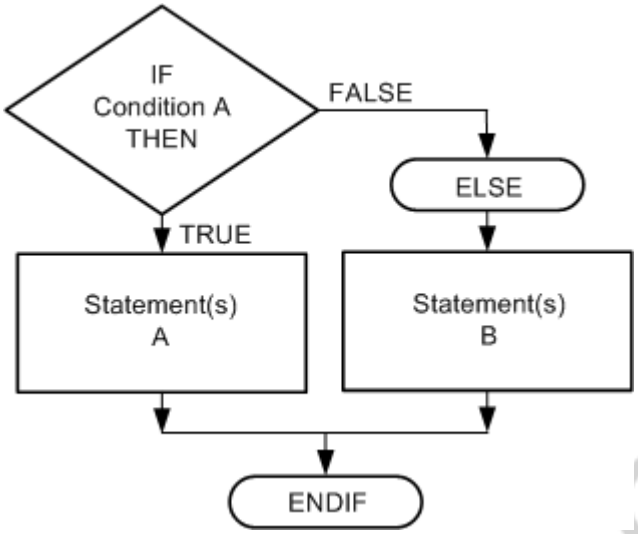


Fig. 19 IF – ELSE statement

```
;*******************************
;********** IF ELSE ************
;*******************************
IF V1 > V2 THEN
    V3 = 99          ; comment
ELSE
    V4 = 66          ; comment
ENDIF
```

Fig. 20 IF – ELSE in a program

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

4.4.3 ELSE IF

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

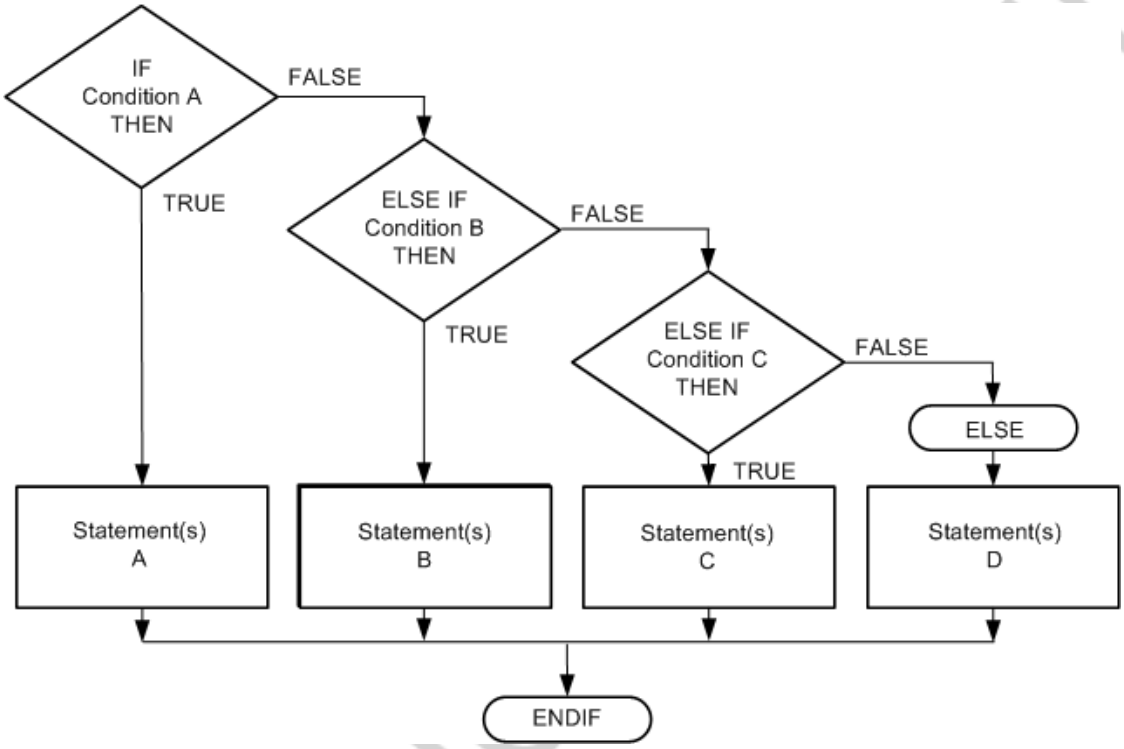


Fig. 21 IF – ELSE IF - ELSE statement

```
;*******************************
;*** IF, ELSE, ELSE IF ********
;*******************************
IF V1 > V2 THEN
    V3 = 99          ; comment
ELSE IF V1 > V4 THEN
    V5 = 88          ; comment
ELSE IF V1 > V6 THEN
    V7 = 77          ; comment
ELSE
    V8 = 66          ; comment
ENDIF
```

Fig. 22 IF – ELSE IF - ELSE in the program

The decisions are processed from top to bottom during runtime. The corresponding statements are executed if the result of a decision is TRUE. The program then continues from after the ENDIF. Only those decisions that correspond to the first TRUE decision are executed, even if subsequent decisions are also TRUE. The statement in the ELSE branch is executed if none of the IF or ELSE IF decisions are TRUE.

Exercise: Weather station - Part I

A temperature sensor measures the outside temperature.

- The **doCold** output should be set when the temperature is below 18°C.
- The **doOpt** output (optimal) should be set when the temperature is between 18°C and 25°C.
- The **doHot** output should be set when the temperature is above 25°C.

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

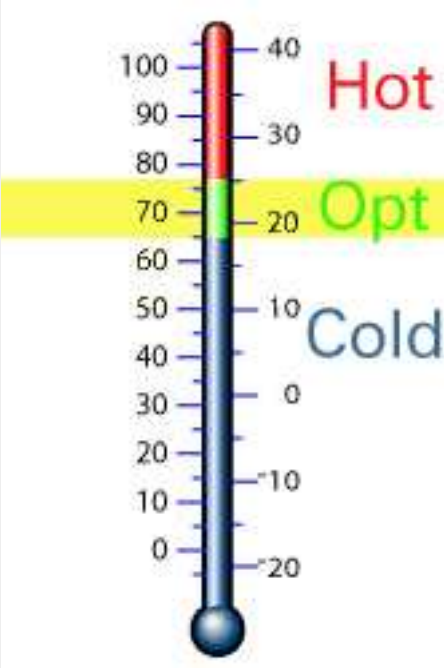


Fig. 23 Thermometer

4.4.4 Nested IF statement

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

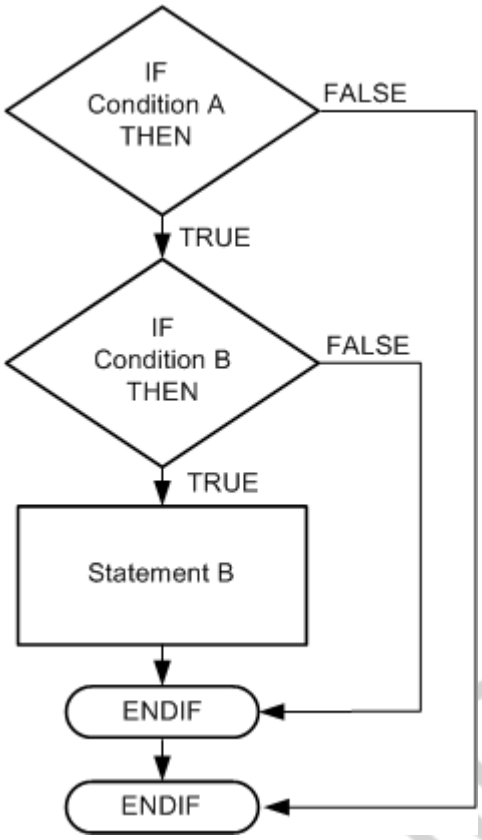


Fig. 24 Nested IF statement

```
;*******************************
;********** Nested IF *********
;*******************************
IF V1 > V2 THEN
    IF V2 > V4 THEN
        V3 = 99      ; comment
    ENDIF
ENDIF
```

Fig. 25 Nested IF statement in a program

It is helpful to indent every nested IF statement and the corresponding expressions. As many IF statements can be nested as needed. However, a high degree of nesting is generally evidence of poor programming style. It becomes nearly impossible to get a clear overview of the code.

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

Exercise: Weather station - Part II

Evaluate the temperature and the humidity.
The **doOPT** output should only be set when the humidity is between 40% and 75% and the temperature is between 18 and 25°C. Otherwise no output should be set.

Solve this task using a nested IF statement.

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

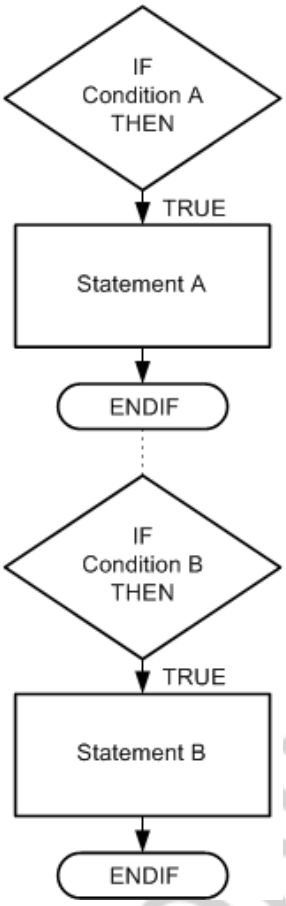


Fig. 26 Two IF statements

```
;*****************************
;*** 1. IF-Statement **********
;*****************************
IF V1 > V2 THEN
    V3 = 99          ; comment
ENDIF
;*****************************
;*** 2. IF-Statement **********
;*****************************
IF V3 = 99 THEN
    Burner = ON      ; comment
ENDIF
```

Fig. 27 Two IF statements

In this case, both IF statements have the same priority. A CASE statement should be used if both IF statements evaluate the same variable for different values.

The CASE statement should be used when:

- IF constructs require too many levels
- too many ELSE IF are used

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

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

4.5 Case statements

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

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

Keywords	Syntax	Description
CASE OF	CASE step variable OF	Beginning of CASE
ACTION ENDACTION	ACTION 1: ACTION 5: Display = "MATERIAL" ENDACTION	For 1 and 5
ACTION ENDACTION	ACTION 2: Display = "TEMP" ENDACTION	For 2
ENDCASE	ENDCASE	End of CASE

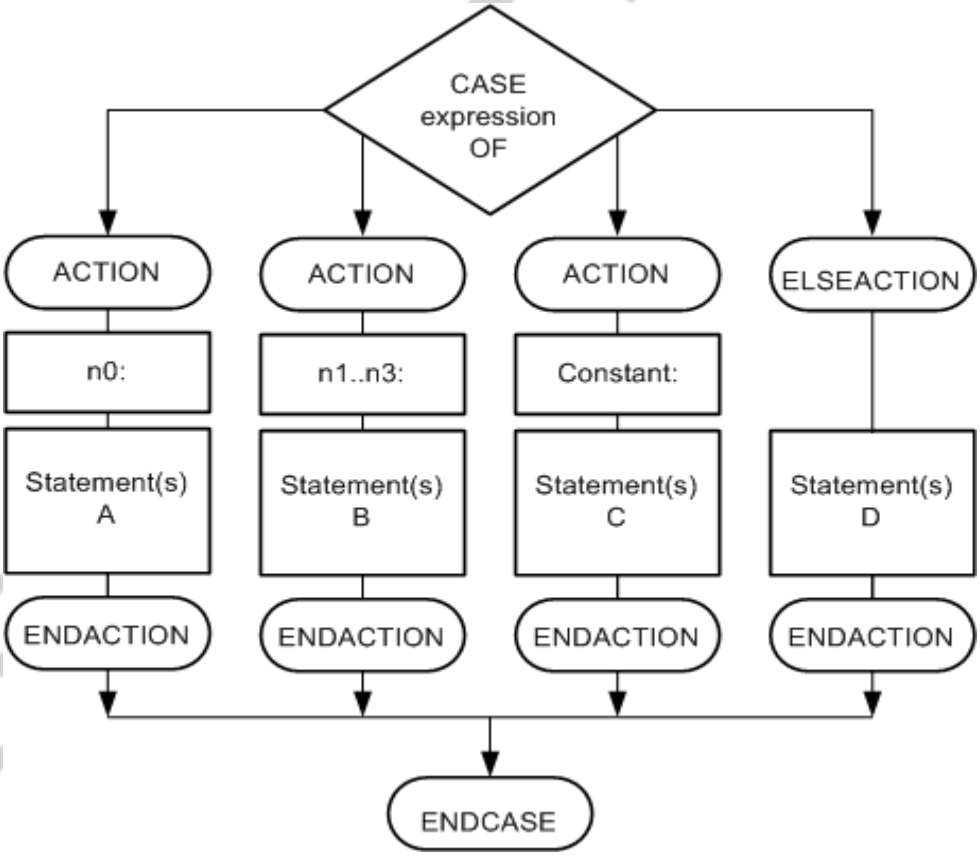


Fig. 28 CASE statement

```
;*****************************
;*********** CASE  ************
;*****************************
CASE stepPV OF
    ; Process when stepPV = 1
    ACTION 1:
        pv = pv + 10
    ENDACTION

    ; Process when stepPV = 2 to 10
    ACTION 2..10:
        pv     = pv - 1
        Output = TRUE
    ENDACTION

    ; Process when stepPV = 11 OR 15
    ACTION 11:
    ACTION 15:
        MachineState = RUN
    ENDACTION

    ; Process when stepPV = ERROR (constant value)
    ACTION ERROR:
        MachineState = ERROR
    ENDACTION

    ; Process when stepPV = all other
    ELSEACTION:
        gError = TRUE
    ENDACTION
ENDCASE
```

Fig. 29 CASE statement in a program

Note:

Constants (TRUE, RUN, ERROR) can be used instead of numbers for the steps in a CASE statement. This makes the program much easier to read.

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

Syntax of the CASE statement:

- A CASE statement begins with CASE and ends with ENDCASE. The ENDCASE keyword must be on a separate line.
- An ENDACTION is necessary for every ACTION and ELSEACTION.
- The value of the statement or variable can be positive and negative. However, it must be a WHOLE NUMBER!
- Whole-number constants must be used for defining the options (action). Variable names or expressions are not allowed.
- Multiple ACTION statements can be written underneath one another and closed off with one common ENDACTION statement in order to execute the same commands for several non-consecutive values:
```
ACTION 1:
ACTION 5:
ACTION 10..12:
 ...
ENDACTION
```
- Numbers cannot overlap or be used in several ranges:
```
ACTION 1..6:
...
ENDACTION
ACTION 5:
...
ENDACTION
```
would not be allowed because the number 5 appears in two actions.

Exercise: Brewing tank

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

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

Create a solution using the CASE statement.

Fig. 30: Brewing tank

4.6 Select statement

A step sequencer is a construction with a number of subprograms (steps). Only one of these subprograms is executed in each program cycle. Exiting from the step or moving to another one occurs depending on certain conditions within the step. Using step sequencers is one of the common programming techniques in PLC programming.

Syntax	Description
SELECT StatePV	Beginning of SELECT statement {optional step number variable}
Flag = 1	Global statement (always processed)
WHEN StopKey = 1 cmdMotor = 0 next DELAY	Global transition condition e.g. detecting a stop switch
STATE DELAY cmdMotor = 0 WHEN UpKey = 1 NEXT UP	DELAY state Motor OFF Check if the “UP” key is being pressed, if yes, change to “UP“ state.
STATE UP cmdMotor = 1 WHEN StopKey = 1 NEXT DELAY	UP state Switch on motor Check if the stop switch was pressed, if true, change to “DELAY“ state.
ENDSELECT	End of SELECT statement

```
;*****************************
;**********  SELECT  **********
;*****************************
SELECT step
    pv = 1
    WHEN tastStop = 1
         cmdMotor = 0
    NEXT WARTE

    STATE WARTE
        cmdMotor = 0
        WHEN tastAuf = 1
        NEXT UP

    STATE UP
        cmdMotor = 1
        WHEN tastStop = 1
        NEXT WARTE
ENDSELECT
```

Fig. 31 SELECT statement in a program

Syntax of the SELECT statement:

- The SELECT construction begins with the keyword SELECT and ends with the keyword ENDSELECT. ENDSELECT must be on a separate line.
- Each step starts with the keyword STATE followed by the step name. Both of these words must be on a separate line:
  ```
  STATE start
  ```
  The transition condition is a block statement. The first line starts with WHEN, followed by the expression formulating the condition. The last line starts with NEXT, followed by a step name located somewhere in the step sequencer:
  ```
  WHEN Sensor = 1
      commands(s)
  NEXT operation
  ```
  The command(s) between WHEN and NEXT are only carried out if the transition condition (in our example UpKey = 1) has been met (is true).
- The keyword WHEN cannot be placed inside of another block statement (e.g. an IF...THEN construction).
- The first run through the step sequencer always starts with the first step.
- If no transition condition is fulfilled (true), the same step is executed again during the next cycle through the SELECT construction (i.e. in the next program cycle).
- A variable containing the step number can also be entered in the SELECT line. This must be a UINT variable.
- The commands between SELECT and the first step (STATE) are always executed regardless of the step number. It can also contain a transitional condition (WHEN...NEXT).

Exercise: Chemical system

Create a program for the following chemical system.

- When the **diStart** key is pressed, the **doWater** water valve should be switched on until the **diWaterOK** water level is reached.
- The **doMixer** mixer and the **doColor** valve for the color are then switched on.
- When the **diFull** sensor is triggered, the color valve is closed again and the **doPumpOutflow** pump as well as the **doValveOutflow** valve for draining are switched on.
- The pump, the mixer and the valve are switched off once the level sinks below the **diLow** sensor.

Create a solution using the SELECT statement.

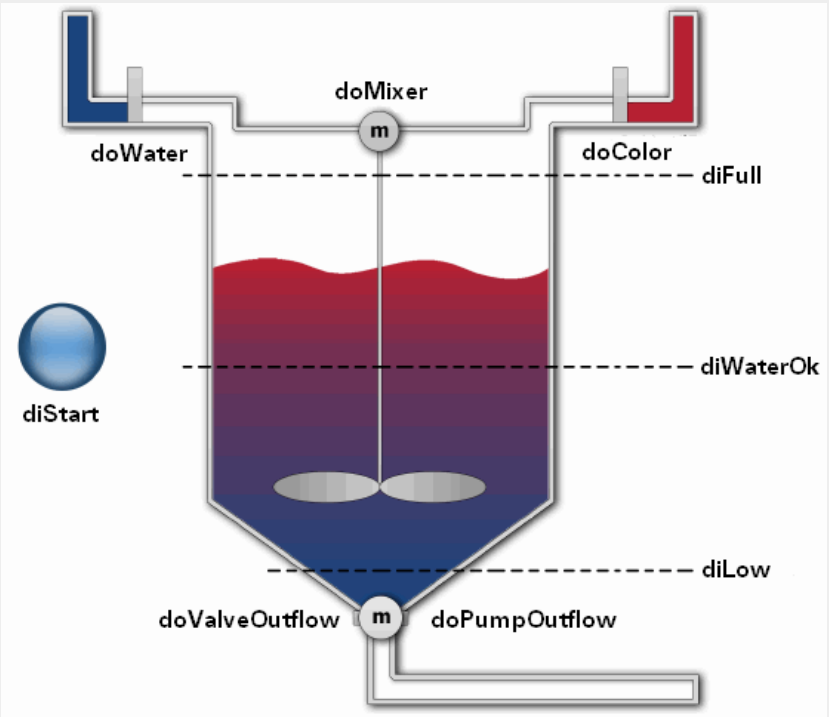


Fig. 32 Chemical system

4.7 Loops

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

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

Depending on the structure of a program, an error in the program could cause the loop to repeat until the time monitor in the CPU triggers an error (system halt).
To prevent such endless loops from occurring, nearly all programs include a way for the loop to be aborted after a defined number of repetitions or to run to a certain limit.

AB offers several types of loops to choose from:

- LOOP .. ENDLOOP
- Counting loop - LOOP .. TO .. DO
- Counting loop - LOOP .. DOWNTO .. DO
- Counting loops with exit condition EXITIF

4.7.1 LOOP

This type of loop is executed until the termination condition has been met.

Keywords	Syntax	Description
LOOP	LOOP	Beginning of the loop
	Counter = Counter + 1	Statement(s) A
EXITIF	EXITIF Counter > 100	Termination condition
	Result = Result + 10	Statement(s) B
ENDLOOP	ENDLOOP	End loop

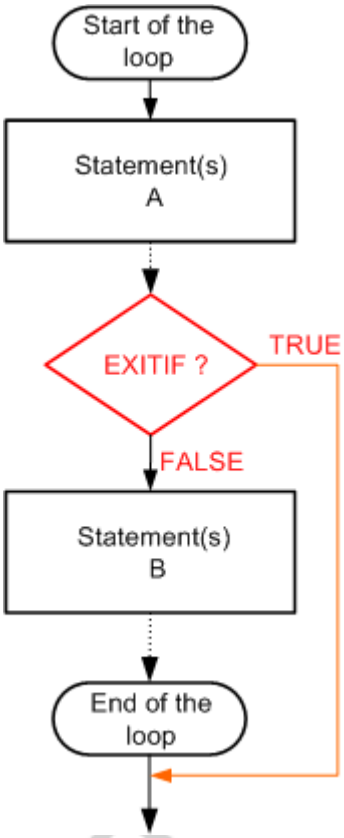


Fig. 33 LOOP statement

```
;**********************
;******** LOOP ********
;**********************
LOOP
    Var1= Var1*2
EXITIF counter <= 0
    counter= counter-1
ENDLOOP
```

Fig. 34 LOOP statement in a program

The A commands are executed first. A check is then made to see if the exit condition is met (true). If so, then the program sequence is continued in the line after ENDLOOP. If not, then the B commands are executed and the program sequence is continued in the line after LOOP (i.e. with command A).

You can also choose to leave out the A or B statements.

Note:

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

4.7.2 LOOP TO / LOOP DOWNTO

This loop LOOP TO and LOOP DOWNTO is used to run a program section for a limited number of repetitions.

Keywords	Syntax	Description
LOOP TO DO LOOP DOWNTO DO	LOOP i = 0 TO 4 DO LOOP i = 5 DOWNTO 1 D0	Beginning of the loop
	Res = value + 1	Loop body statement(s)
ENDLOOP	ENDLOOP	End loop

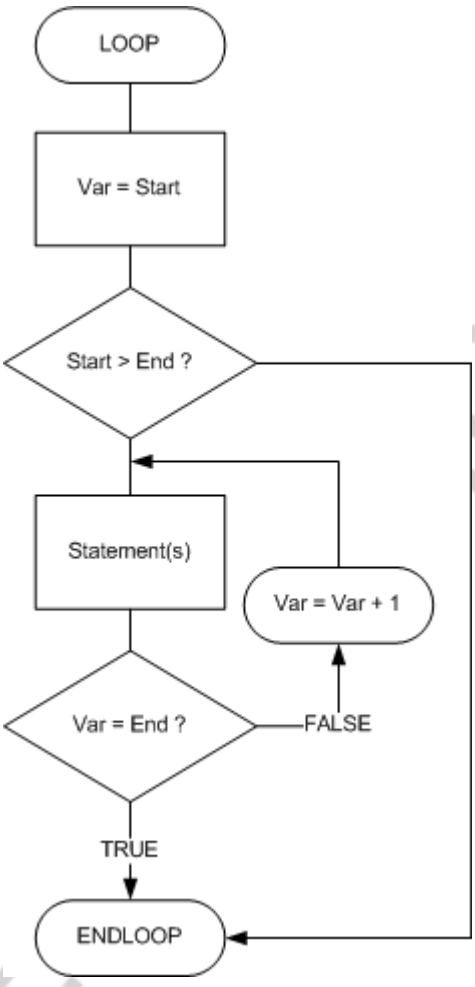


Fig. 35 LOOP statement

```
;******************************
;*** LOOP TO / LOOP DOWNTO ****
;******************************
LOOP Var=Start TO End DO
    Res = value + 1     ; Commands
ENDLOOP

LOOP Var=END DOWNTO Start DO
    Res = value + 1     ; Commands
ENDLOOP
```

Fig. 36 LOOP statement in a program

The statements in the LOOP loop are repeated. At every repetition, the loop counter "Var" is incremented or decremented. The two control variables "Start" and "End" determine the start value and end value of the loop counter. After the end value is reached, the program continues from after the ENDLOOP statement.

The LOOP TO statement increments the loop counter in each passage up to the end value. Conversely, the LOOP DOWNTO statement decrements the loop counter by one each time.

The entry condition is evaluated with every repetition of the loop.

Exercise: Crane

5 separate loads are suspended from a crane. The individual loads must be added together to determine the total load.

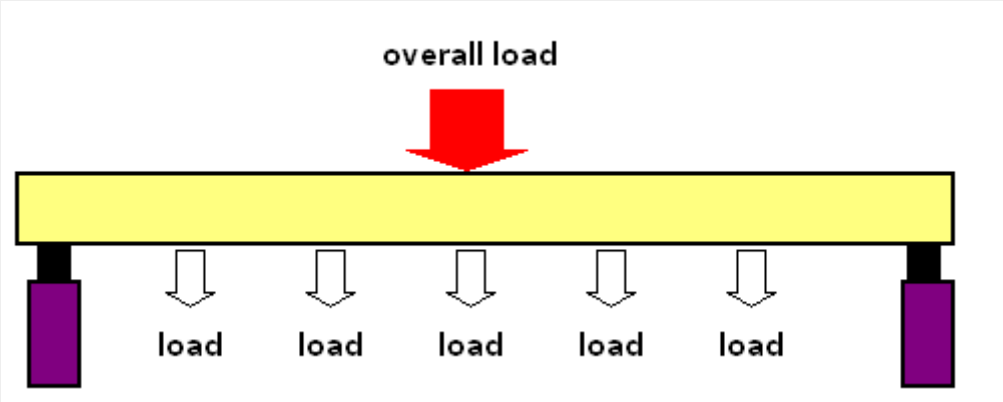


Fig. 37: Crane

Create a solution for this task using a FOR loop.

Warning:

Take not that, for example, an array with 5 elements is defined from index 0-4.

For example: The “load“ variable is declared as an array with 5 elements.

```
LOOP i=1 TO 5 DO     ;During the fifth passage, you then write...
load[i] = value      ;outside of the array, because the array
ENDLOOP              ;ranges only from index 0 – 4
```

4.7.3 EXITIF

The EXITIF statement can be set within loops to terminate the loop independently of the loop statement.

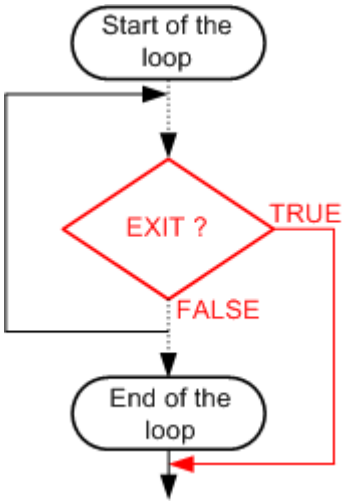


Fig. 38 EXITIF statement

```
;*******************************
;****** LOOP TO & EXITIF ******
;*******************************
LOOP i=0 TO 9 DO
    var = var + 1
    EXITIF gStatus = ERROR
    varOK = varOK + 1
ENDLOOP
```

Fig. 39 EXITIF statement in a program

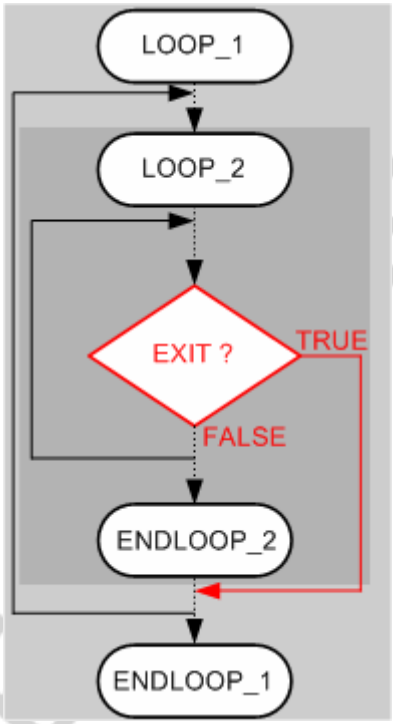


Fig. 40 EXIT statement in a nested loop

```
;*******************************
;***** EXITIF NESTED LOOP *****
;*******************************
LOOP i=0 TO 9 DO
    var = var + 1
    LOOP j=0 to 9 DO
        var1 = var1 + 1
        EXITIF gStatus = ERROR
    ENDLOOP
ENDLOOP
```

Fig. 41 EXIT statement in a nested LOOP statement in a program

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

4.8 Calling function blocks

A function block can be called two ways in Automation Basic.

- FBK call:
  The function block is called directly via the respective names. The input and output parameters are placed in brackets.
- Alias FBK call:
  The alias call mostly differs from previous procedures in how the values are assigned. This is done on a freely definable alias name and structure elements.

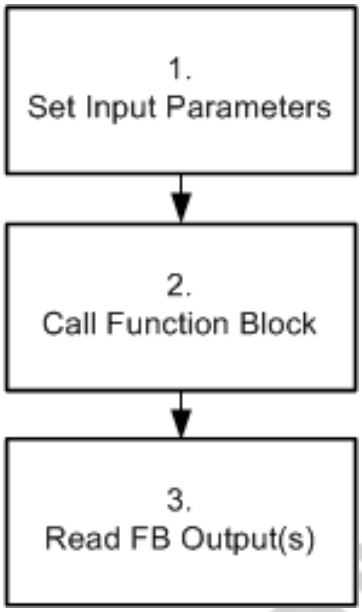


Fig. 42 Calling a function block

```
;*******************************
;***** ALIAS FUB CALL *********
;*******************************
TON_xx.IN= Input
TON_xx.PT= T#5s
TON_xx FUB TON()
Output = TON_xx.Q

;*******************************
;********** FUB CALL **********
;*******************************
TON(Input, T#5s, Output, ElapseTime)
```

Fig. 43 Calling a function block in a program

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

Function block call in detail:

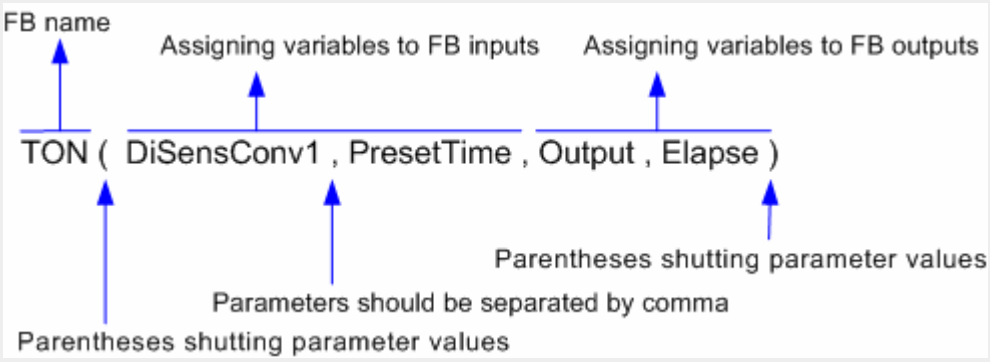


Fig. 44 Detail view of a function block call

First the function block name is entered, then the input and output parameters are assigned in parentheses, separated by commas.

Alias function block call in detail:

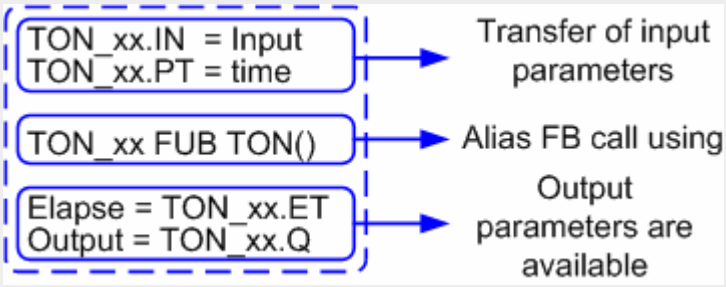


Fig. 45 Detail view of an alias function block call

The input parameters of the FBK structure are assigned first. The function block is then executed. The output parameters are now in the FBK structure and can be read by the application.

Exercise: Bottle counter

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

Fig. 46 Bottle counter

Note:

The Automation Studio online help is a useful resource when working with function blocks.

4.9 Pointers and dynamic variables

B&R also offers pointers in Automation Basic.

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

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

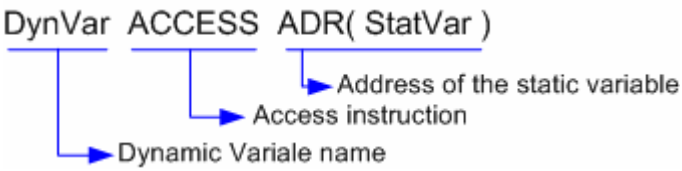


Fig. 47 Referencing a dynamic variable

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

5. EXERCISES

Exercise: Box lift

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

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

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

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

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

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

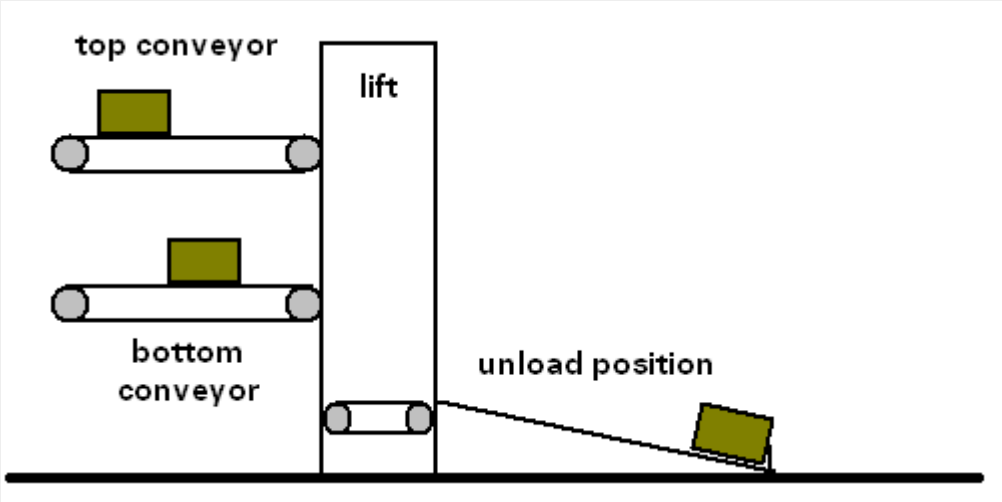


Fig. 48 Box lift

6. SUMMARY

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

Fig. 49 Book printing: then and now

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

7. APPENDIX

7.1 Keywords

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

Keyword	Description
ACCESS	Defines dynamic access.
ACTION	See Case Statement.
BIT	Simple data type for digital states.
BIT_CLR	A = BIT_CLR(IN, POS) A contains the value IN after the bit at position POS is deleted. However, the IN operand remains unchanged.
BIT_SET	A = BIT_SET(IN, POS) A contains the value IN after the bit at position POS is set. However, the IN operand remains unchanged.
BIT_TST	Determines the value of a bit: A = BIT_TST(IN, POS) A contains the value of the bit at position POS of operand IN.
CASE	See Case Statement.
DO	See Loop Statement.
DOWNTO	See Loop Statement.
EDGE	Detects positive and negative edges.
EDGENEG	Detects negative edges.
EDGEPOS	Detects positive edges.
ELSE	See If Statement.
ELSEACTION	See Case Statement.
ENDACTION	See Case Statement.
ENDCASE	See Case Statement.
ENDIF	See If Statement.
ENDLOOP	See Loop Statement.
ENDSELECT	See Select Statement.
EXITIF	See Loop Statement.
FLOAT	Simple data type, 32-bit size, for floating point values.
FBK	Alias for function block call.
GOTO	See Goto Statement.
IF	See If Statement.
INC	Increases the value of an operand by 1.
INT16	Simple data type, 16-bit size, for positive and negative values.

INT32	Simple data type, 32-bit size, for positive and negative values.
INT8	Simple data type, 8-bit size, for positive and negative values.
LONG	Simple data type, 32-bit size, for positive values.
LOOP	See Loop Statement.
NEXT	See Select Statement.
OF	See Case Statement.
SELECT	See Case Statement.
STATE	See Select Statement.
THEN	See If Statement.
TO	See Loop Statement.
WHEN	See Select Statement.

7.2 Functions

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

Function	Example
ABS	Returns the absolute value of a number. ABS(-2) returns 2.
ACOS	Returns the arc cosine (inverse function of cosine) of a number.
ADR	Returns a variable's address.
AND	Logical AND operation by bit.
ASIN	Returns the arc sine of a number (inverse function of sine).
ASR	Arithmetic shifting of an operand to the right: A = ASR (IN, N) IN is shifted N bits to the right, the left is filled with the sign bit.
ATAN	Returns the arc tangent of a number (inverse function of tangent).
COS	Returns the cosine of a number.
EXP	Exponential function: A = EXP (IN).
EXPT	One operand raised to the power of another operand: A = EXPT (IN1, IN2).
LIMIT	Limitation: A = LIMIT (MIN, IN, MAX) MIN is the lower limit, MAX is the upper limit for the result. If IN is less than MIN, then the MIN result is returned. If IN is greater than MAX, then the MAX result is returned. Otherwise, the IN result is returned.
LN	Returns the natural logarithm of a number.
LOG	Returns the base-10 logarithm of a number.
LSL	See SHL.
LSR	See SHR.
MAX	Maximum function. Returns the larger of two values.

MIN	Minimum function. Returns the lesser of two values.
MOD	Modulo division of a USINT, SINT, INT, UINT, UDINT, DINT, REAL type variable by another variable of one of these types.
MUX	Selection: A = MUX (CHOICE, IN1, IN2, ... INX) CHOICE specifies which of the operators IN1, IN2, etc. INX is returned as a result.
NOT	Negation of a bit operand by bit.
OR	Logical OR operation by bit.
ROL	Rotates an operand's bits to the left: A = ROL(IN, N) The bits in IN are shifted N times to the left, the far left bit being pushed in again from the right.
ROR	Rotates an operand's bits to the right: A = ROR (IN, N) IN is shifted N times to the right one bit position at a time, the far right bit being pushed in again from the left.
SEL	Binary selection: A = SEL (CHOICE, IN1, IN2) CHOICE must be type BOOL. If CHOICE is FALSE, then IN1 is returned. Otherwise, IN2 is returned.
SHL	Shifts an operand's bits to the left: A = SHL (IN, N) IN is shifted N bits to the left, the right side is filled with zeroes.
SHR	Shifts an operand's bits to the right: A = SHR (IN, N) IN is shifted N bits to the right, the left side is filled with zeroes.
SIN	Returns the sine of a number.
SIZEOF	This function returns the number of bytes required by the specified variable.
SQRT	Returns the square root of a number.
TAN	Returns the tangent of a number.
TRUNC	Returns the integer part of a number.
XOR	Logical EXCLUSIVE OR operation by bit.

7.3 Solutions

Exercise: Light control

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

Exercise: Aquarium

```
aoAvgTemp= UINT((DINT(aiTemp1) + aiTemp2) / 2)
```

Exercise: Weather station - Part I

```
doCold= false
doHOT= false
doOpt= false

if (AItmp < 180) then
    doCold= true
else if (AItmp > 250) then
    doHOT= true
else
    doOpt= true
endif
```

Exercise: Weather station - Part II

```
doCold= false
doHOT= false
doOpt= false

if (AItmp < 180) then
    doCold= true
else if (AItmp > 250) then
    doHOT= true
else
    if (AIhum > 400) and (AIhum < 750) then
        doOpt= true
    endif
endif
```

Exercise: Brewing tank

```
;convert to percent value
level= USINT((DINT(aiLevel)*100)/32767)

doHorn= 0
doLow= 0
doOk= 0
doHigh= 0

CASE level OF
    ; <1% turn the horn on
    ACTION 0..1:
        doHorn= 1
        doLow= 1
    ENDACTION
    ; <25%
    ACTION 2..24:
        doLow= 1
    ENDACTION
    ; >90%
    ACTION 91..100:
        doHigh= 1
    ENDACTION

    ELSEACTION:
    doOk= 1
    ENDACTION
ENDCASE
```

Exercise: Crane

```
overall_load:= 0;
LOOP i=0 TO 4 DO
    overall_load= overall_load + load[i]
ENDLOOP
```

Exercise: Chemical system

```
select
    doWater = 0
    doMixer = 0
    doColor = 0
    doPumpOutflow = 0
    doValveOutflow = 0

    ; Wait for Start button
    state WAIT
        when diStart = 1
        next WATER

    ; let in Water , wait for Water Ok
    state WATER
        doWater = 1
        when diWaterOk = 1
        next COLOR
    ; let color in, wait for sensor full
    state COLOR
        doMixer = 1
        doColor = 1

        when diFull = 1
        next OUTFLOW
    ; outflow, wait for sensor low
    state OUTFLOW
        doPumpOutflow = 1
        doValveOutflow = 1
        when diLow = 1
        next WAIT
endselect
```

Exercise: Box lift

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

Overview of training modules

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

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

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

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

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

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

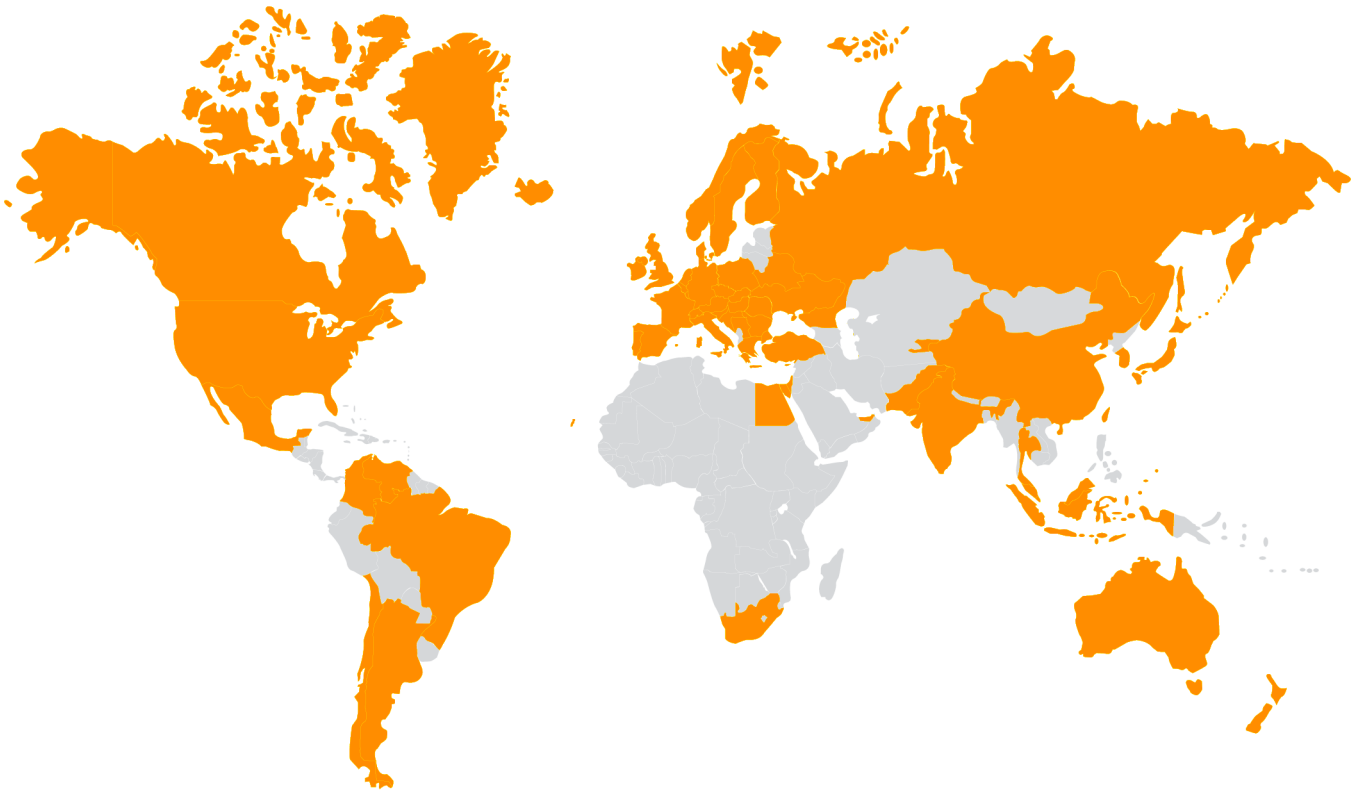
**) see Product Catalog

CORPORATE HEADQUARTERS

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

TM247TRE.00-ENG 0907

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



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