ST Language Statements

Expressions and Statements

In a program you differentiate between expressions and statements.
Expressions are composed of operands and operators. They always have an expression value; the expression 2 + 2, for example, yields the value 4.
Statements are composed of one or more expressions. In ST, a statement is terminated by a semicolon.

Assignment

An assignment replaces the current value of a variable with another value. The assignment *A := 5;* for example replaces the current value of *A* with the value *5*. The assigned value itself can again be the result of an evaluation, such as *A := SIN(X);*.

Selection Statements

To control the sequence of the program run in dependency on values that can be changed at the runtime of the program, the IEC 1131-3 standard defines control structures which allow the selective execution of statements and groups of statements in conjunction with conditional expressions. The following control structures are supported:

- Selection with *IF*
- Multiple selection with *CASE*

Selection with IF

The keyword IF controls the sequence of the subsequent program run in dependency on an expression that can be evaluated. The expression must return either a logical TRUE or FALSE. If the condition is TRUE, the subsequent statement or statement group is executed, if the condition is FALSE, the subsequent statement or statement group is skipped.
IF statements can be nested as desired.

Syntax:
The conditional expression is entered in round brackets after the keyword IF. The closing bracket is followed by the keyword THEN to initiate the statement or statement group that is to be executed if the evaluation yields TRUE. The section of the statement that follows the keyword THEN can consist of one or more statements. To identify the end of the statement section, the statements to be executed if the condition is met are terminated by END_IF, followed by a semicolon.

***Example*:**

```
FUNCTION_BLOCK FunctionBlock1

VAR
    FB2 : FunctionBlock2;
END_VAR

VAR_INPUT
    Input1 : DINT;
    Input2 : LREAL;
END_VAR

VAR_OUTPUT
    Output1 : BOOL := TRUE;
    Output2 : LREAL;
END_VAR

IF (Input1 > 20) THEN
    Output1 := FALSE;
    FB2(Display := Input2);
END_IF;

Output2 := SIN(Input2);
END_FUNCTION_BLOCK
```

If the input variable *Input1* has a value > 20, the evaluation result of the conditional expression of the IF clause is *TRUE*. In this case, the two statements in the statement section of the IF clause are executed: The output variable *Output1* is assigned the value *FALSE*, and the function block instance *FB2* is called with the value of *Input2*. The sine is then calculated in the subsequent statement, and the result is assigned to the output variable *Output2*.
If the input variable *Input1* has a value < 20, the evaluation result of the conditional expression of the IF clause is *FALSE*. In this case, the statements in the statement section of the IF clause are not executed. The program evaluation continues directly with the sine calculation instead.

IF with Alternative: IF ELSE

The IF-ELSE selection extends the simple IF selection by offering a statement section that is executed if the evaluation of a condition yields the value FALSE, in addition to the statement section that is executed if the evaluation of a condition yields TRUE.

Syntax:
After the statement section that is executed if the evaluation of the conditional expression yields TRUE, the keyword ELSE signalizes that this is the point at which the statement section begins that is executed if the evaluation of the conditional expression is FALSE. The keyword ELSE at the same time delimits the first statement section. When you use ELSE, the keyword END_IF including the semicolon is no longer placed after the statement section for a true condition but after the statement section of the ELSE section.

Example:

```
FUNCTION_BLOCK FunctionBlock1

VAR
    FB2 : FunctionBlock2;
    FB3 : FunctionBlock3;
END_VAR

VAR_INPUT
    Input1 : DINT;
    Input2 : LREAL;
END_VAR

VAR_OUTPUT
    Output1 : BOOL := TRUE;
    Output2 : LREAL;
END_VAR

IF (Input1 > 20) THEN
    Output1 := FALSE;
    FB2(Display := Input2);
ELSE
    Output1 := TRUE;
    FB3(Display := Input2);
END_IF;

Output2 := SIN(Input2);

END_FUNCTION_BLOCK
```

Here, the example above has simply been expanded by an ELSE clause. If the evaluation of the conditional expression yields FALSE, the statement section of the ELSE section is executed. *Output1* is in this case assigned the value *TRUE* and, instead of the function block instance *FB2*, the function block instance *FB3* is called with the value of *Input2*.

Multiple Selection with ELSIF

With the help of the keyword *ELSIF*, decision chains can be achieved within an IF section. Checking the conditional expressions specified in the associated ELSEIF sections is performed in the order in which they appear in the source code. ConditionalExpression1 is checked first, and if it is TRUE the statement section following ConditionalExpression1 is executed. The execution of the other statement sections is then terminated. If the evaluation of ConditionalExpression1 yields *FALSE*, however, the assigned statement section is skipped and ConditionalExpression2 is evaluated. If TRUE, the statement section following ConditionalExpression2 is executed, and the checking of the other conditional expressions is terminated; otherwise, ConditionalExpression3 is evaluated, etc. If none of the evaluated conditional expressions yields *TRUE*, you can define a default statement section in a terminating ELSE clause, which is executed if none of the ELSIF conditions yields the value *TRUE*.

Syntax
The keyword *ELSIF* follows the statement section of a selection initiated by *IF* or *ELSIF*. The keyword *ELSIF* is followed by the conditional expression to be evaluated. The statement section for the case of TRUE is initiated by the keyword *THEN*.
A selection chain with *ELSIF* can be terminated by an ELSE clause which is executed by default if none of the evaluated conditions yields *TRUE*. The keyword *ELSE* is not followed by a conditional expression. The statement section following *ELSE* is not initiated by the keyword *THEN*.

Example:

```
FUNCTION_BLOCK FunctionBlock1

VAR
    FB2 : FunctionBlock2;
    FB3 : FunctionBlock3;
    FB4 : FunctionBlock4;
```

```
END_VAR

VAR_INPUT
    Input1 : DINT;
    Input2 : LREAL;
END_VAR

VAR_OUTPUT
    Output1 : BOOL := TRUE;
    Output2 : LREAL;

END_VAR

IF (Input1 = 20) THEN
    Output1 := FALSE;
    FB2(Display := Input2);
ELSIF (Input1 = 25) THEN
    Output1 := TRUE;
    FB3(Display := Input2);
ELSE
    Output1 := TRUE;
    FB4(Display := Input2);
END_IF;

Output2 := SIN(Input2);

END_FUNCTION_BLOCK
```

In the IF section, the variable Input1 is first checked as to whether it is equal to the value 20. If TRUE, the two subsequent statements are executed, and the evaluation of the subsequent conditional expressions is canceled.
If FALSE, the conditional expression following the ELSIF keyword is checked. If the evaluation yields TRUE, the subsequent statement section is executed, and the evaluation of the subsequent conditional expressions is canceled.
If FALSE, the statement section following the keyword ELSE is executed. Here, no further conditions will be evaluated since the statement section behind ELSE is always executed if none of the preceding conditions has yielded the value TRUE in the IF and ELSIF clauses.

Multiple Selection with CASE

Instead of using IF ELSIF, you can also achieve multiple selections with a CASE section. An integer variable is here used as the conditional expression. The value of this variable is compared with several other values or value ranges in the CASE section. If the value of the variable is within the value range of a selection condition in the CASE section, the statements following this selection are executed, and the evaluation of the subsequent selection conditions is canceled. Instead of the above IF ELSIF selection, you could also have written the following code:

```
FUNCTION_BLOCK FunctionBlock1

VAR
    FB2 : FunctionBlock2;
    FB3 : FunctionBlock3;
    FB4 : FunctionBlock4;
END_VAR

VAR_INPUT
    Input1 : DINT;
    Input2 : LREAL;
END_VAR

VAR_OUTPUT
    Output1 : BOOL := TRUE;
    Output2 : LREAL;
END_VAR

CASE Input1 OF
    20: Output1 := FALSE;
        FB2(Display := Input2);
    25: Output1 := TRUE;
        FB3(Display := Input2);
ELSE
    Output1 := TRUE;
    FB4(Display := Input2);
END_CASE;

Output2 := SIN(Input2);

END_FUNCTION_BLOCK
```

Syntax:
All conditional expressions and statement groups must be located between the keywords *CASE* and *END_CASE*. The CASE variable whose value is to be checked follows the keyword *CASE*.
Values or value ranges with which the variable value is to be compared are initiated by the keyword *OF*. Each value or value range is terminated by a colon. The colon is followed by one or more statements.
With the keyword *ELSE*, you can optionally provide a default statement group that is executed if none of the selection conditions is met.

The CASE variable may only be of the type ANY_INT.

Values or value ranges can be defined in the following ways:

- A single value is specified: e.g. *5:*
  In this case, the variable value is checked as to whether is equals the value *5*.
- Several values are specified, separated by commas: e.g.: *5,7,9,11:*
  In this case, the variable value is checked as to whether it equals one of the specified values.
- A value range is defined: e.g. *5..7:*
  In this case, the variable value is checked as to whether it is within the specified value range.

Example:

```
FUNCTION_BLOCK FunctionBlock1

VAR
    FB2 : FunctionBlock2;
    FB3 : FunctionBlock3;
    FB4 : FunctionBlock4;
END_VAR

VAR_INPUT
    Input1 : DINT;
    Input2 : LREAL;
END_VAR

VAR_OUTPUT
    Output1 : BOOL := TRUE;
    Output2 : LREAL;
END_VAR

CASE Input1 OF
    20:     Output1 := FALSE;  (* Selection with single value *)
            FB2(Display := Input2);
    21,22:  Output1 := TRUE;  (* Multiple selection with 2 values *)
            FB3(Display := Input2);
    23..30: Output1 := FALSE;  (* Selection with value range *)
            FB4(Display := Input2);
ELSE (* Default selection; only executed if CASE variable matches no CASE value. *)
    OutputDebugString("Value of Input1 is not between 20 and 30");

END_CASE;

END_FUNCTION_BLOCK
```

The CASE variable *Input1* is first compared with the value *20*. If the value of *Input1* matches this value, the corresponding statement group is executed, and the comparison with the remaining values is canceled.
If the value of *Input1* does not correspond to the value *20*, it is compared with the next CASE value, in this case with the values *20* and *21*. If the value of *Input1* matches one of these two values, the corresponding statement group is executed, and the comparison with the remaining CASE values is canceled.
If the value of *Input1* does not correspond to the values 20 or *21*, it is compared with the next CASE value, in this case as to whether *Input1* is within the value range from *23* to *30*. If the value of *Input1* matches this value, the corresponding statement group is executed, and the comparison with the remaining values is canceled.
If the value of *Input1* is not within the value range from *23* to *30*, the default selection initiated by the keyword ELSE is executed, and a message is issued in the Message view.

Loops

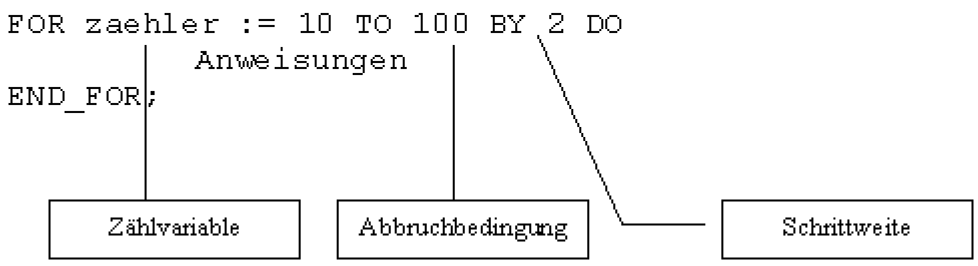
Loops allow executing one or more statements several times. The execution of a loop is controlled in the so-called loop heading whereas the statement section is executed in the loop body. The statement section can contain one or more statements that are executed repeatedly until the abort condition yields TRUE.

Three loop constructs are available for executing one or more statements repeatedly:

FOR Loop

With many of the jobs to be executed in a program, you already know in advance how often they are to be executed. To execute a specific number of repeats in the loop, you need a counting process for checking the number of times the loop has been executed and an abort condition for terminating the execution of the loop. These two data are combined in the heading of the FOR loop. In addition, the FOR loop heading receives a variable which defines the incrementation of the loop, i.e. the value by which the loop counter is incremented or decremented. Entering the incrementation is optional. If omitted, the loop counter will be incremented by the value 1 by default.

Syntax:



In the counter variable, the initial value of the loop counter is specified. The incrementation indicates by how much the value of the counter variable is to be incremented or decremented with each execution of the loop. The abort condition defines which value the counter variable must reach for the execution of the loop to be canceled. The statement section is initiated by the keyword DO and terminated by END_FOR.
In the example above, the counter variable has the initial value 10. It is incremented by the value 2 with each execution of the loop. When the counter variable reaches the value 100, the execution of the loop is canceled.

Example:

```
VAR
    Error : BOOL;
    ArrayLen : DINT := 100;
    ArrayStart : DINT := 1;
    FirstArray : ARRAY[1..100] OF DINT ;
    SecondArray : ARRAY[1..100] OF LREAL;
    counter : DINT;
END_VAR

IF (Error) THEN (* If error occurred, reinitialize array *)
    FOR counter := ArrayStart TO ArrayLen DO
        FirstArray[counter] := counter;
        SecondArray[counter] := DINT_TO_LREAL(counter);
    END_FOR;
END_IF;
```

In the case of an error, the two arrays *FirstArray* and *SecondArray* are to be reinitialized. The error case is queried in the IF clause with the variable *Error*. If *Error := TRUE*, the loop for initializing the array is executed. Both arrays have the size 1 to 100. The loop counter *counter* is thus initialized with the value of *ArrayStart* and incremented until the abort condition defined in *ArrayLen* (length of the array) is reached. Every time the loop is executed, a further element is initialized in the two arrays.

WHILE Loop

With the WHILE loop, the conditional expression in the loop heading is evaluated first. If the evaluation yields the value *TRUE*, the statements in the loop body are executed. The execution of the statement is continued until the conditional expression yields the value *FALSE*. If the value of the conditional expression is already *FALSE* in the first evaluation, the statements in the statement group are not executed.
In contrast to the FOR loop, the loop heading of the WHILE loop does not contain a loop counter. You must therefore ensure in the loop body of the WHILE loop that the conditional expression takes on the value *FALSE* sometime to avoid an endless loop.

Syntax:
The keyword *WHILE* is followed by the conditional expression. The conditional expression can be a calculated expression, e.g. *var1 - var 2 < 100*. The statement group is initiated by the keyword *DO* and terminated by *END_WHILE*.

Example:

```
sum   := 1;

WHILE sum < 100 DO
    sum := sum + 1;
END_WHILE;
```

The variable *sum* in the loop body in this example is not only used for storing a value but acts as a loop counter at the same time. Adding the value *1* each time the loop is executed not only serves purely for calculation but also ensures that the conditional expression in the loop heading reaches the value *FALSE*, and the execution of the loop is canceled.

REPEAT Loop

The REPEAT loop is executed until the evaluation of the conditional expression yields the value *TRUE*. In the REPEAT loop, in contrast to the WHILE loop, the statement group is executed first, even before the conditional expression is checked. This is because the conditional expression is located in the loop foot. The statement group is thus executed at least once, even if the first evaluation of the conditional expression already yields the value *FALSE*.
In contrast to the FOR loop, the loop foot of the REPEAT loop does not contain a loop counter. You must therefore ensure in the loop body of the REPEAT loop that the conditional expression takes on the value *TRUE* sometime to avoid an endless loop.

Syntax:
The keyword REPEAT is followed by the statement group. The statement group in turn is followed by the loop foot in which the conditional expression is evaluated.
The loop foot is initiated by the keyword *UNTIL*. It is followed by the conditional expression, which is terminated by the keyword *END_REPEAT*.

Example:

```
VAR
    sum : DINT;
END_VAR

sum   := 101;

REPEAT
    sum := sum + 1;
UNTIL sum > 100
END_REPEAT;
```

Since the sum variable is initialized with the value *101*, the execution of the loop is canceled immediately after the evaluation of the conditional expression. The statement group is nevertheless executed once, and the value of the sum variable is incremented by 1.

Canceling the Execution of a Loop with EXIT

The keyword EXIT is used especially in highly nested loop constructions to cancel the execution of the loop before the condition for terminating the loop is met in the conditional expression. After an EXIT statement, the loop execution of the loop in which the EXIT statement is located is interrupted immediately. The program run is continued after the first keyword that indicates the end of a loop (END_FOR, END_WHILE, END_REPEAT). Within a nested loop construction, a jump is performed from the innermost loop in which the EXIT statement is located to the next outer loop which continues its execution at the position after the entry into the next inner loop.

Example:

```
FOR I := 1 TO 5 DO
    FOR J := 2 TO 10 DO
        sum := sum + J;
        IF (sum = 5) THEN
           EXIT;
        END_IF;
    END_FOR;
    sum := sum + j;
END_FOR;
```

Exiting a Function, Function Block or Program with RETURN

A function, function block or program can be exited with the statement RETURN before the end of the corresponding POU is reached. In this case, a return jump is performed within the calling POU to the position after the call.

With functions, the return values must have been assigned before the return jump to the calling POU. If the return jump into the calling POU is performed within function blocks before the output variables have been assigned values, the default values of these variables are returned.

Example:

```
IF(EmergencyStop := TRUE) THEN
    RETURN;
END_IF;
```

Operators

If several operators are used in an expression, the order in which they are processed is governed by the priorities of the operators. Operators with a higher priority are given precedence in processing over operators with a lower priority. If operators have the same priority, they are processed from left to right.
The priority rules can be changed by parenthesizing. The parentheses are operators themselves and have highest priority.

Example:

```
VAR_OUTPUT
    vOut : DINT;
END_VAR


VAR
    a : DINT := 1;
    b : DINT := 2;
    c : DINT := 3;
    d : DINT := 4;
END_VAR

vOut := a + b * c + d;
```

The multiplication operator has precedence over the addition operators. This results in the following processing order:

1. *b* is multiplied by *c (b * c)*
2. The result of multiplication is added to *a* due to processing from left to right *a + (b * c)*
3. *d* is added to the result of the previous multiplication and addition *(a +(b * c)) + d*

If you want the additions to be performed first and the multiplication last, you can use parentheses.

```
vOut := (a + b) * (c + d);
```

In this version, the changed evaluation order results in the assignment of the value *21* to *vOut*, instead of the value *11* of the previous version.

The following table provides a list of all ST operators. The operators are listed in the order of their priorities. The further up an operator is in the table, the higher its priority. Operators of the same priorities are in the same line of the table.

Operation	Symbol
Parenthesizing	(Expression)
Evaluation of a function	Identifier (list of arguments)
Negation Complement	- NOT
Exponent	**
Multiplication Division Modulo	* / MOD
Addition Subtraction	+ -
Comparison Less than ... Greater than ... Less or equal ...	 < > <=

Greater or equal ...	>=
Equality check Inequality check	= <>
Boolean AND Boolean AND	& AND
Boolean Exclusive OR	XOR
Boolean OR	OR

Short Circuit Evaluation for Expressions with Boolean Operators

When logical expressions are evaluated, the shortest evaluation path is determined. This means that the evaluation is aborted as soon as it is certain that, after the evaluation of a partial expression, the result of the total expression can no longer change. This is the case, for example, if two expressions have been combined by a boolean AND and the first expression yields the value *FALSE* during evaluation. In this case, the total expression can no longer receive the value *TRUE*. The evaluation will therefore be aborted after the first expression. Example:

```
VAR_OUTPUT
    vOut : BOOL;
END_VAR

VAR
    a : DINT := 1;
    b : DINT := 2;
    c : DINT := 3;
    d : DINT := 4;
END_VAR

vOut := (a > b) & (c > d);
```

In this case, the expression *(c > d)* is no longer evaluated since the expression *(a > b)* is already *FALSE*, which means that the total expression can no longer change to *TRUE*.

Pay attention to the special evaluation of logical expressions when you assign values in this kind of expressions. These assignments will no longer be executed if the section in which the assignment is performed is no longer evaluated.

User-Defined Functions

Use and Syntax

Functions are used for executing frequently recurring jobs within a project. Instead of having these jobs executed repeatedly within function blocks or programs in the form of a series of statements, you can also code such a sequence of statements centrally within the body of a function. Every time a job is to be executed, the corresponding function is called instead of the sequence of statements.

You can pass one or more parameters to functions. The parameters, for example, specify with which values a function is to perform a calculation or can be used as values for conditional branches within the function body. The result of a function is returned to the calling POU as a return parameter and can be evaluated by the POU for further processing.

Since functions should always perform the same job in the same way, variables that are declared within functions are not static. This means they do not retain their last value from function call to function call, but are newly initialized every time the function is called. The result of a function is therefore the same for every function call with the same parameters. For this reason, in contrast to function blocks, no instance of the function, which can save its own state from instance call to instance call, is declared before a function is called.

A function is characterized by the following syntactic elements:

Parameter

Values can be passed to a function by the caller as parameters. Such values can be further processed within the function body. Parameters must be declared as *VAR_INPUT*, *VAR_OUTPUT* or *VAR_IN_OUT* within the declaration section of a function. Variables that are declared as parameters within *VAR_INPUT* or *VAR_IN_OUT* sections of a function must not be assigned an initial value. With variables within a *VAR_OUTPUT* section, however, the assignment of an initial value is possible.
During a function call, the parameters are assigned actual parameters by the calling POU. The assignment is performed by enclosing variables or literals (literals are only possible with input variables) in round brackets after the function name and separating them by commas. The order in which the values are specified within the brackets must correspond to the order in which the parameters are declared within the *VAR_INPUT*, *VAR_OUTPUT* or *VAR_IN_OUT* sections of the called function. The parameters and the passed actual parameters must be of the same data type.

Return value

As a result of the statements executed in the function body, a value can be returned to the calling POU. For this purpose, a value is assigned to the function name. When declaring the function, the data type of the function's return value must be specified after the function name, separated by a colon. The data type of the function that is specified during declaration and the data type of the value that is assigned to the function name must be the same. If a function returns a value, this value must be evaluated by the calling function in the form of an assignment or a comparison operation.

Example:

Declaration of *MyFunction* with a return value of type *DINT*:

```
FUNCTION MyFunction : DINT

VAR CONSTANT
    standard : DINT := 5;
END_VAR

VAR_INPUT (* Declaration of the parameters to be passed *)
    factor1 : DINT;
    factor2 : LREAL;
    condition : BOOL;
END_VAR

VAR
    ReturnValue : DINT;
END_VAR

(* Statements executed with the actual parameters that have been passed *)
If(condition) THEN
    ReturnValue := factor1 * LREAL_TO_DINT(factor2);
else
    ReturnValue := standard;
END_IF;

(* Assignment of the return value to the function name *)
MyFunction := ReturnValue;

END_FUNCTION
```

Call of the function *MyFunction* from the function block *CallFB*:

```
FUNCTION_BLOCK CallFB

VAR
    firstVar : DINT;
    secondVar : LREAL;
    Result : DINT;
END_VAR

firstVar := 24;
secondVar := 5.8;

(* Function call with assignment of the return value of the function to Result *)
Result := MyFunction(firstVar, secondVar, TRUE);

END_FUNCTION_BLOCK
```

The value returned by the function *MyFunction* is evaluated by the calling function block *CallFB* by assignment to the variable *Result*. It would also be possible to perform the evaluation in a comparison operation:

```
IF (Result > MyFunction(firstVar, secondVar, TRUE)) THEN
    (* Statements  *);
END_IF;
```

For a description of how to add a new function with the help of the wizard, refer to the section Creating a New Function.

Passing Variables by Copy or Reference

In most programming languages, it is usually not the parameters themselves that are passed to the function as actual parameters, but only copies of them. This method of passing values ensures that no side effects can occur, which means that there will be no effects on the variables in the calling POU even if the values of the passed variables are changed by the function. If, for example, a variable of type integer, which had the value *5* immediately before the function call, is passed to a function, and the called function increments this variable by the value *10*, this variable has the value *5* again immediately after the function call in the calling POU. This method of passing values is also referred to as *call-by-value*.

The case is quite different if the parameters of a function are declared in a *VAR_IN_OUT*, *VAR_OUTPUT* or *VAR_INPUT* section in 4CONTROL. In this case, the actual parameters of the called function are passed as pointers to their memory location. This means, the variable itself is passed to the called function so that the changes that are made to the variable within the function body also have effects on the variable declared in the calling POU. To take up our example above again: If the variable of type *DINT* with the value *5* is incremented by the value *10* by the called function, this variable has the value 15 in the calling POU immediately after the function call. This method of passing values is referred to as *call-by-reference*. With functions, however, this method is only significant for *VAR_IN_OUT* and *VAR_OUTPUT* variables since *VAR_INPUT* variables must not be written in within the function body and therefore cannot be changed. Put in programming terms: a *VAR_INPUT* variable cannot be used as an *L-value* within a function. For this reason, passing parameters to a *VAR_INPUT* variable will have no side effects in the calling POU on the value of the variable passed to the function. With *VAR_IN_OUT* and *VAR_OUTPUT* variables, side effects are possible since variables of this type can not only be read but also written within the function body. If the value of a *VAR_IN_OUT* variable is changed within the function, the value of the variable passed as a parameter therefore also changes in the calling POU.

You can use this method of passing values to *VAR_IN_OUT* variables if you need more than one value from the called function as a result of the function call. Since the variables passed to the function are changed directly, the changed variable values are retained after exiting the function and can be evaluated in the calling POU. If you would like to receive more than one result value from the function, you can therefore pass several parameters to a function with the call-by-reference method. Keep in mind, however, that when passing the values by reference the passed variable values are overwritten with new values if they are changed by the called function.

Example:

Function declaration with parameters called by reference:

```
FUNCTION swap:BOOL

(* Declare parameters to be called by reference *)
VAR_IN_OUT
    swapA : DINT;
    swapB : DINT;
END_VAR

(* Declare temporary variable for temporarily
storing swap values *)
VAR
    swapTemp : DINT;
END_VAR

(* Swap values *)
swapTemp := swapA;
swapA := swapB;
swapB := swapTemp;

(* Return value *)
swap := TRUE;

END_FUNCTION
```

Function call from function block *Caller*:

```
FUNCTION_BLOCK Caller

VAR
    a : DINT;
    b : DINT;
    aResult : DINT;
    bResult : DINT;
    RValue : BOOL;
END_VAR

a := 20;
b := 50;

RValue := swap(a,b);

aResult := a; (* a = 50 after function call *)
bResult := b; (* b = 20 after function call *)

END_FUNCTION_BLOCK
```

The function *swap* defined above has the task of swapping the values of the two variables that are passed. Before the function call, the variable *a* has the value *20* and the variable *b* has the value *50* in the instance of the function block *Caller*. The two values are swapped within the function body. After the execution of the statements in the function body, *a* has the value *50* and *b* the value *20*. Since both variables were called by reference, the two variables also have these values in the calling instance of the function block *Caller* after the function call.

In the declaration section of a function, you can insert a *VAR_INPUT*, a *VAR_OUTPUT* as well as a *VAR_IN_OUT* section. The order of the sections is not relevant. When calling the function with the actual parameters, however, the order of the actual parameters must correspond to the order in which the formal parameters are declared in the declaration sections of the function.

Functions without Parameters or Return Values

The definition of a parameter or return value is optional when you create a function. Functions can also be called without parameters and do not necessarily need to return a value.

If a function is to be called without parameters, the function must not contain a *VAR_INPUT* or *VAR_IN_OUT* section, or the two sections must be empty.
When you call the function, leave the parentheses for the actual parameters empty behind the function name.

In the case of functions that are not intended to return a value, the assignment of a return value to the function name is omitted. In addition, no data type may be indicated after the function name. If only one of the two is missing, a compiler error will occur.
Since a function without a return value does not return a value to the caller, such a function is called directly by its function name without the assignment of values or comparison operators.

Example:

```
(* Function declaration without data type for return value *)
FUNCTION DoSomething

(* Since the function does not accept any parameters,
   a VAR_INPUT or VAR_IN_OUT section is omitted here *)

OutputDebugString("Function DoSomething was called");

(* Since the function does not return a value,
   a value assignment to the function name is omitted here *)

END_FUNCTION
```

The function *DoSomething* neither provides a return value, nor can any values be passed to it. For this reason, it has not been declared with a data type behind the function name and does not assign a value to the function name within the function body.

This function is called with empty parameter parentheses. Since the return value is missing, no evaluation will be performed:

```
DoSomething();
```

Calling Functions from Functions

Functions can be nested, which means that another function can be called from a function body. The called function can be a function from a standard library as well as a user-defined function. Calling a function from a different function does not differ from calling a function from a program or function block. You can use the possibility of calling functions from functions to break jobs up into modular units and thus contribute to a clearer structure of a project.

Example:

```
PROGRAM FunctionTest

VAR
    ProgramResult : DINT;
END_VAR

ProgramResult := FunctionA(2,4);

END_PROGRAM

---------------------------------------
FUNCTION FunctionA : DINT

VAR_INPUT
    VarA : DINT;
    VarB : DINT;
END_VAR

VAR
    AddValue : DINT;
    Result : DINT;
END_VAR

AddValue := VarA + VarB;
Result := AddValue + Square(AddValue);

FunctionA := Result;
END_FUNCTION

---------------------------------------
FUNCTION Square: DINT
VAR_INPUT
    Number : DINT;
END_VAR

VAR
    result : DINT;
END_VAR

result := Number * Number;
Square := result;

END_FUNCTION
```

The calling hierarchy in the above example follows the following order: The program **FunctionTest** calls the function **FunctionA** with two parameters. **FunctionA** adds up the two passed parameters and calls the function **Square** to square the result of the addition. Since the value returned by **Square** represents a valid expression value, the function call can be inserted instead of a variable directly after the plus operator when adding the addition result in **AddValue** to the squared addition result. The result of the entire calculation is then returned to the calling program **FunctionTest**.

Restrictions in 4CONTROL

The IEC1131-3 standard permits calling functions alternatively by exclusively indicating the actual parameters or by indicating the formal parameters to which the actual parameters are passed by means of the assign operator. In the current version, 4CONTROL only supports the first of the two possibilities, as described above. Since the second possibility of passing parameters is not supported, it is not possible to call functions with parameters whose order is independent of the declaration order in the declaration section of the function. It is also not possible to omit irrelevant parameters in the function call.

Initial values are not permitted for parameters in *VAR_INPUT* and *VAR_IN_OUT* sections.

Only values of data type *ANY_SIMPLE* may be used as return values.

In *VAR* sections for the declaration of local variables, only variables of type *ANY_SIMPLE* are allowed.

In functions, variables must not be declared with the attributes *RETAIN* and *NON_RETAIN*.

Functions that call another function, must not pass any local variables as actual parameters to formal parameters which have been declared in a *VAR_OUTPUT* section in the called function.

Debugging and monitoring statements and values within functions is not supported. You therefore cannot set any breakpoints within a function body or monitor values in the Watch view or by using data tips.