PDHonline Course E334 (3 PDH)

Structured Text Programming

Instructor: Anthony K. Ho, PE

2020

PDH Online | PDH Center

5272 Meadow Estates Drive
Fairfax, VA 22030-6658
Phone: 703-988-0088
www.PDHonline.com

An Approved Continuing Education Provider

Table of Contents

1. Background and Benefits of Structured Text

Structured Text (ST) is a high level textual programming language that is syntactically similar to Pascal. It is developed and published by IEC in the IEC 61131-3 international standard in 1993, which is aimed to standardize programming languages for programmable logic controllers (PLC). ST is very flexible and intuitive for writing control algorithms. The language, which is as efficient as ladder logic, uses typical operations such as logical branching, multiple branching, and loops. ST programs can be created in any text editor and they resemble sentences, making them easy to program, debug, test, and understand. Due to its highly-structured nature, ST is ideal for tasks requiring complex math, algorithms, or decision-making.

In recent years, an increasing number of industrial users are requesting the use of ST over other programming languages for their industrial process needs. The reason is simple: most manufacturing companies prefer vendor independent platform in their plants. When circumstances arise, a control system can be switched from one vendor to another with minimal effort. It is because most common PLC manufacturers have already adopted ST. ST can also be written to run on hardware and software PLC platforms, making it one of the most universal text based languages.

Besides its universal nature, ST offers these benefits to engineers and programmers:

- Engineers and programmers trained in computer languages can learn to program ST without difficulty.
- Tag Structure UDT (User Defined Tags), local variables, symbols, and ST programming conventions make the programs easy to understand.
- Programs can be created in any text editor.
- Programs run as fast and efficient as ladder logic.
- State Machine can easily be emulated with CASE statement.
- ST can be connected to other IEC 61131-3 languages, such as Ladder Diagram (LD) and Sequential Function Chart (SFC).
- A running program or part of a running program can be changed without stopping the PLC (online change).
- Many build-in functions are available.

2. Comment

Comment is very essential part of any programming languages. Comments in the code serve to explain the flow and purpose of the code so that its understanding and maintenance become easier. There are two types of comment in ST:

```
//  Line Comment
(* This is a Block
   Comment *)
```

Line command can only be placed on a line, on its own or after some code. Comment placed between the set (* and *) is called block comment and therefore multiple lines of comments are allowed. However, nested comments are not allowed.

3. Data Type

Similar to other programming languages, the IEC 61131-3 standard provides many different types of data in ST programs, both simple and derived data types.

3.1. Simple Data Type

The following are the simple data types provided by the standard:

- Integer
- Floating Point
- DateTime
- String
- Bit String

3.1.1. Integer

Data Type	Description	Lower Range	Upper Range	Example
SINT	Short Integer	−128	127	100
INT	Integer	−32768	32767	20000
DINT	Double Integer	-2^{31}	$2^{31} - 1$	2^{18}
LINT	Long Integer	-2^{63}	$2^{63} - 1$	-2^{10}
USINT	Unsigned Short Integer	0	255	22
UINT	Unsigned Integer	0	$2^{16} - 1$	3300

UDINT	Unsigned Double Integer	0	$2^{32} - 1$	2^{20}
ULINT	Unsigned Long Integer	0	$2^{64} - 1$	2^{55}

If a variable is assigned a value or variable that is out of its permitted range, an overflow runtime error may occur.

3.1.2. Floating Point

Data Type	Description	Lower Range	Upper Range	Example
REAL	Real Number	-10^{38}	10^{38}	-10^{25}
LREAL	Long Real Number	-10^{308}	10^{308}	10^{185}

3.1.3. DateTime

Data Type	Description	Example
TIME	The Duration of Time	TIME#16d3h38m4s120ms
DATE	Calendar Date	DATE#2005-06-15
TIME_OF_DAY	Time of Day	TIME_OF_DAY#21:45:30.92
DATE_AND_TIME	Date and Time of Day	DATE_AND_TIME#2005-06-15-12:30:00

In the above example, the keywords can be abbreviated to T#, D#, TOD#, and DAT#, respectively.

3.1.4. String

Data Type	Description	Example
STRING	Character String	'This is a string'

Note that the maximum length of STRING is 254 characters.

3.1.5. Bit String

Data Type	Description	Example
BOOL	Bit String of 1 Bit	1
BYTE	Bit String of 8 Bits	01110001
WORD	Bit String of 16 Bits	0110111011001000

DWORD	Bit String of 32 Bits	01101111111111101101101011110110
LWORD	Bit String of 64 Bits	0110111111111110110110111111111110110110

3.2. Derived Data Type

Aside from using the simple data types, programmers always find it necessary to construct sophisticated data types; therefore, ST provides programmer the ability to build new and custom data types. A new data type is constructed by enclosing its definition within the keywords TYPE and END_TYPE. The definition consists of the name of the data type under construction, followed by a colon, and then by the kind of derived data type to be constructed. There are four kinds of derived data types:

- Structured Data Type
- Enumerated Data Type
- Sub-Ranges Data Type
- Array Data Type

3.2.1. Structured Data Type

A structured data type is a composite data type, constructed by enclosing the elements of the composition within the keywords STRUCT and END_STRUCT. Each element consists of its name followed by a colon and then by its data type. A structured data type itself can contain one or more structured data types. For example:

```
TYPE Valve :
   STRUCT
      DisplayColor : ValveColor;
      CurrentState : ValveState;
      Pressure : REAL;
   END_STRUCT;
END_TYPE
```

In the example above, the ValveState data type is a structured data type itself and ValveColor is an enumerated data type as illustrated below.

3.2.2. Enumerated Data Type

An enumerated data type is composed by enclosing the elements of the enumeration in parentheses. Each element consists of a different name. For example:

```
TYPE ValveColor :
 (Red, Yellow, Green);
END_TYPE
```

All derived data types can be assigned initial default values. The defaults are included in the type definition. For example:

```
TYPE ValveColor :
 (Red, Yellow, Green) := Red;
END_TYPE
```

3.2.3. Sub-Ranges Data Type

A sub-ranges data type is composed by limiting the range of another data type, usually integers. Limited versions of a data type consist of the name of the data type to be restricted, followed by a lower and an upper bound separated by two dots and enclosed in parentheses. For example:

```
TYPE PressureRange :
   INT(-100..+500);
END_TYPE
```

The sub-ranges data type can be used to ensure variable declared with the data type will not be assigned a value out of its range. Runtime error will occur when out of range data is assigned to the variable.

3.2.3. Array Data Type

An array data type is composed by the keyword ARRAY, followed by a multi-dimensional vector of ranges of indices enclosed by square brackets, and then by the keyword OF, followed by the data type (simple or derived) that is going to be contained in the array. A multi-dimensional vector of ranges of indices consists of a comma-delimited list of ranges of indices. A range of indices consists of a lower and upper bound of the range, both integers, separated by two dots. For example:

```
TYPE MeteringSkid :
   ARRAY[1..5, 1..10] OF Valve;
END_TYPE
```

The above example creates a 2-dimensional array, a total of 50 elements in the array. Although it is dependent on memory limitation of the running machine, you should try to keep the total number of elements in the array below 65536. Only values of same data types can be stored in an array. Also, array can be zero based, if ARRAY[0..5] is declared, 6 array elements will be available.

4. Variable Declaration

ST programs allow named variables to be defined. This is similar to the use of symbols when programming in ladder logic. Variable names must be begun with a letter, after that they can include combinations of letters, numbers, and some symbols such as '_'. Variable names are not case sensitive and can include any combination of upper and lower case letters. However, they must not have the same name as predefined functions, or user defined functions. For example, a variable name ARRAY will be invalid. Also, direct memory access variable can be used, such as I:001/02 and 40001. The following keywords can be used to declare variables:

Declaration	Description
VAR	Local or Internal Variable Declaration
VAR_GLOBAL	Global Variable Declaration
VAR_INPUT	Input Variable Declaration
VAR_OUTPUT	Output Variable Declaration
VAR_IN_OUT	Input and Output Variable Declaration
VAR_ACCESS	Direct Access Variable Declaration
VAR_EXTERNAL	External Variable Declaration
VAR_TEMP	Temporary Variable Declaration
AT	Assign Memory Location to Variable
CONSTANT	Value Cannot Be Changed
RETAIN	Retain After Power Failure
END_VAR	End of Variable Declaration

The above keywords define the scope of the variables. The VAR_INPUT, VAR_OUTPUT and VAR_IN_OUT declarations are used for variables that are passed as arguments to the program or function. The RETAIN declaration is used to retain a variable value, even when the PLC power has been cycled.
The following example uses some of the keywords mentioned above.

```
INTERFACE
```

```
   VAR_GLOBAL RETAIN  // Global Battery-Backed Variable
      Area : REAL;
      TimePeriod : TIME := t#50ms; // 50 ms As Initial Value
   END_VAR
   PROGRAM DemoProgram;  // Declare the program
END_INTERFACE

FUNCTION_BLOCK Pressure : REAL
   VAR_INPUT
      Force1, Force2 : REAL;
   END_VAR
   Pressure := (Force1 + Force2) / Area;
END_FUNCTION_BLOCK

IMPLEMENTATION
   PROGRAM DemoProgram
      VAR  // Local Variable
         FlipFlop : BOOL;  // Flag
         AT %QL100 : LWORD;  // Output
      END_VAR
      (* Program Body *)
   END_PROGRAM
END_ IMPLEMENTATION
```

Although ST can run on PC, its primary running platform is PLC. Therefore, there is no file I/O with ST programming. ST program can read from and write to stored memory through PLC register as described in the above example using the keyword AT and then the register address.

5. Program Structure

As you have seen in the previous example, ST provides the programmer a structured way to position the code. Variable and program declarations are first placed within the keywords INTERFACE and END_INTERFACE. After that, functions or function blocks are defined. IMPLEMENTATION and END_IMPLEMENTATION keywords are then used to include the program definition:

```
INTERFACE
   VAR_GLOBAL
      ...
   END_VAR
   PROGRAM ProgramName;
END_INTERFACE
```

```
FUNCTION_BLOCK FunctionName
   ...
END_FUNCTION_BLOCK

IMPLEMENTATION
   PROGRAM ProgramName
      ...
   END_PROGRAM
END_ IMPLEMENTATION
```

6. Assignment Statement

Statements are software instructions for various purposes such as assignment statement, calling function block, iterative processing, conditional evaluation, etc. Among those, assignment statement is used the most; it can change the value stored in a variable or the value returned by a function. The general syntax of assignment statements takes the form:

```
VarA := VarB;
```

VarA is a variable to which a value is assigned using variable VarB which can also be an expression or a literal constant. The value obtained by evaluation of VarB is assigned to the variable VarA by this statement and replaces the previous value of VarA. Value of VarB will not be changed after the assignment. Care must be taken to ensure that the data type of VarA is the same as that of VarB; otherwise, runtime error may occur. Take a look at the following examples.

```
MotorSpeed := 22.8;
```

This statement assigns a literal constant value of 22.8 to a variable MotorSpeed. It is assumed that the variable MotorSpeed has been declared as a REAL data type variable in the declaration section of the program.

```
Counter := Counter + 1;
```

The above statement assigns a value to an Integer variable Counter, which is incremented by 1 (add 1 to Counter's previous value and assign back to Counter.)

```
Pressure[4] := Force / Area;
```

This statement replaces the 4th element of the array variable Pressure by the value evaluated by the expression Force / Area where Force and Area are two other variables having some predetermined values. (Note that if Area has a value of zero, division by zero error will result during runtime.) The values of Force and Area remain unchanged after the assignment statement.

```
RadianA := SIN(AngleA);
```

This is an example of using an arithmetic function (a sine function) for assigning a value to variable RadianA.

```
TemperatureArray1 := TemperatureArray2;
```

This statement causes the value of array variable TempeatureArray2 to be assigned to another array variable TempeatureArray1. The value of TempeatureArray2 remains unchanged after the assignment. It is important to make sure that the two array variables must be of the same data type and size, for example REAL and 100, respectively.

Assume the following array variable CoeffArray contains 30 integers:

```
CoeffArray := 15(1), 10(2), 5(3);
```

The above statement will assign 1 to the first 15 elements, 2 to the next 10 elements, and so forth. This can be used to conveniently initialize array variable.

7. Expression

Expressions are part of statements where the values of one or several variables are manipulated using arithmetic or Boolean operations to produce a single value. This value is then assigned to another variable, which should be of the same data type as the result of the evaluation. Runtime error may occur if the data type of the result of the expression evaluation and that of the variable to be assigned do not match. An expression is placed on the right hand side of an assignment statement and the variable to be assigned on the left hand side. For example:

```
VarA := VarB + VarC;
```

The expression VarB + VarC is evaluated first and then assigned to variable VarA. All three variables must use the same data type. If the data types are not the same,

casting function such as INT_TO_DINT can be used to force the data type conversion, for example:

```
VarA := INT_TO_DINT(VarB + VarC);
```

7.1. Evaluating Expression

An expression is evaluated using the order of operator precedence. Operator precedence refers to the order in which operators are used or evaluated. An operator with higher precedence will be used or evaluated before an operator with lower precedence. In the case that operators in two different parts having equal precedence, they are evaluated from left to right. Below are the rules for expression evaluation:

A. All parenthesized sub-expressions are evaluated fist. Nested parenthesized sub-expressions are evaluated inside out, with the innermost sub-expression evaluated first.
B. The operator precedence rule: operators in the same sub-expression are evaluated in the order from level 1 through level 4 as shown in the table below, with level 1 being evaluated first.
C. The left associative rule: operators in the same sub-expression and at the same precedence level (such as + and -) are evaluated left to right.

The table lists the operators from highest to lowest order of precedence in which an expression will be evaluated.

Operator	Description	Example
( )	Parentheses Expression	X := (A + B) * C;
Build-In or User Defined Function ( )	Function Evaluation. Build-In Functions such as SQRT, TOD, FRD, NOT, NEG, LN, LOG, DEG, RAD, SIN, COS, TAN, ASN, ACS, ATN	Avg1 := Avg_REAL(14.0, 15.0);
**	Exponent	X := A ** 3;
-	Negation	A := -A;
NOT	Boolean Complement	C := NOT D;
*	Multiplication	A := B * C;
/	Division	A := B / C;
MOD	Modulus	A := B MOD C;

+	Addition	A := B + C;
-	Subtraction	A := B - C;
<, <=, =, <>, >=, >	Comparison	VarA := VarB >= VarC + 10;
AND, &	Boolean AND (Logical)	VarA := VarB AND VarC;
XOR	Boolean Exclusive OR (Logical)	VarA := VarB XOR VarC;
OR	Boolean OR (Logical)	VarA := VarB OR VarC;

As you can see, the NOT operator has the highest precedence, followed by the multiplicative operators (including AND), the additive operators (including OR), and finally the relational operators. Because the comparison operators have the lowest precedence, you should generally use them with parentheses to prevent unintentional results.

Let's look at an example:

```
X := A + C / 2 + SQRT(A + B) - C * 3;
```

Assume the following values of A = 30, B = 6, and C = 4. The expression is evaluated in this order:

(A + B) = (30 + 6) = 36
SQRT(36) = 6
C / 2 = 4 / 2 = 2
C * 3 = 4 * 3 = 12
X = 30 + 2 + 6 - 12 = 26

Therefore, X is assigned the value 26. If we add some parentheses to the above expression, the result will change. For example:

```
X := (A + C) / 2 + (SQRT(A + B) – C) * 3;
```

Now the evaluation will be done in this order:

(A + C) = (30 + 4) = 34
(A + B) = (30 + 6) = 36
SQRT(36) = 6
(6 - C) = (6 - 4) = 2
34 / 2 = 17
2 * 3 = 6

17 + 6 = 23

The result becomes 23. This example illustrates how expression evaluation is affected by the change in precedence obtained by introducing set of parentheses. In the case of Boolean expressions, similar rules of precedence apply. However, Boolean expression is evaluated only up to a point that is necessary to determine the result. For example:

```
X := A AND B AND C;
```

If the value of A is FALSE, the expression is evaluated as FALSE right away since the evaluation of the B and C will not affect the final result.

8. Function

Function is one of the basic building blocks of a program. It composes a sequence of statements, with zero or more input values and one or more output values for exchanging data. The advantage of using function is that a sequence of statements can be reused many times without the need to rewrite the same and tedious code in the program, which could help to reduce the size of program and syntax errors. A function always produces the same output value(s) for the same set of input values. Variable declaration keywords can be used to declare variables within a function. ST supports two types of function: function and function block. Function is a block with function value for extension of the basic PLC operation set. It is a logic block with no static data which means that all local variables lose their value when you exit the function and the variables are reinitialized the next time you call the function. Function block, on the other hand, is a block with input and output variables and is a code block with static data. Since function block has memory, its output parameters can be accessed at any time and from any point in the user program. Local variables retain their values between calls. Therefore, function block is used more frequently by programmer.

To use function or function block, you must create its definition before calling it. The following is an example of a function block which calculates the average pressure of two valves. Two input arguments of REAL type are declared in the function block and the computed result is passed back to the function name.

```
FUNCTION_BLOCK Valve_Average : REAL
  VAR_INPUT
    Valve1, Valve2 : REAL;
  END_VAR
```

```
    Valve_Average := (Valve1 + Valve2) / 2;
END_FUNCTION_BLOCK
```

Not only that the code can be reused many times, but different variables can be passed to the function block and result from the computation is returned. The following example demonstrates different ways to call the function block. Note that the three ways of calling the function block below will yield the same output.

```
Avg1 := Valve_Average(Valve1 := 187.3, Valve2 := 176.9);
Avg2 := Valve_Average(187.3, 176.9);
ValveX := 187.3;
ValveY := 176.9;
Avg3 := Valve_Average(ValveX, ValveY);
```

The function call itself can behave as a variable; therefore, the following is valid.

```
Total := Total + Valve_Average(Valve1, Valve2);
```

The table below lists the variable type allowed in PROGRAM, FUNCTION_BLOCK, and FUNCTION.

Variable Type	Allowed In:		
	PROGRAM	FUNCTION_BLOCK	FUNCTION
VAR	Yes	Yes	Yes
VAR_INPUT	Yes	Yes	Yes
VAR_OUTPUT	Yes	Yes	No
VAR_IN_OUT	Yes	Yes	No
VAR_EXTERNAL	Yes	Yes	No
VAR_GLOBAL	Yes	No	No
VAR_ACCESS	Yes	No	No

Note:

- PROGRAM may call FUNCTION and FUNCTION_BLOCK, but not the other way around.
- FUNCTION may call FUNCTION.
- FUNCTION_BLOCK may call FUNCTION_BLOCK.
- FUNCTION_BLOCK may call FUNCTION, but not the other way around.

9. Conditional Statement

ST provides statements that decide a particular course of action depending on a set of conditions. We will examine different types of conditional statements. IF-THEN-ELSE statement is very common and it is often used when there are a few conditions. When there are many conditions, CASE statement can be used to improve readability of the program.

9.1. IF-THEN-ELSE Statement

The general format of this statement is as follows:

```
IF <Condition> THEN
   <Statement>
ELSE
   <Statement>
END_IF;
```

Note that the ELSE portion of the statement is optional. The following example tests the status of the pump and the pressure reading to determine the states of the open and closed latches.

```
IF (Pump_Status = TRUE) AND (Pressure > Set_Point) THEN
   Open_Latch := TRUE;
   Closed_Latch := FALSE;
ELSE
   Open_Latch := FALSE;
   Closed_Latch := TRUE;
END_IF;
```

If the conditions following the IF keyword are satisfied, then the value of variable Open_Latch will be set to TRUE and Closed_Latch set to FALSE. Otherwise, Open_Latch will be set to FALSE and Closed_Latch set to TRUE. These statements can be nested within each other to form more complex conditions. The format of nested conditions is as follows:

```
IF <Condition 1> THEN
   <Statement 1>
   IF <Condition 2> THEN
      <Statement 2>
   ELSE
      <Statement 3>
   END_IF
ELSE
```

```
   <Statement 4>
END_IF;
```

In the above conditional statements, Statement 1 will be executed if Condition 1 is satisfied. After executing Statement 1, Condition 2 is checked and if true, Statement 2 will be executed; otherwise, Statement 3 is executed. If Condition 1 is not satisfied, then Statement 4 will be executed instead. You can add an ELSIF keyword to check more conditions in a conditional statement:

```
IF <Condition 1> THEN
   <Statement 1>
ELSIF <Condition 2> THEN
   <Statement 2>
ELSE
   <Statement 3>
END_IF;
```

Statement 1 will be executed if Condition 1 is satisfied. If not, Condition 2 is checked and if satisfied Statement 2 will be executed. If none of the above conditions are satisfied, then Statement 3 will be executed. You can see that if there are a lot of conditions to check, the IF-THEN-ELSE statement can get cumbersome and become hard to read. CASE statement can be considered to replace IF statement to increase readability of the code.

9.2. CASE Statement

CASE is a type of conditional statement where certain actions are carried out depending on the value of a variable or expression. The CASE construct has the general form:

```
CASE <Expression> OF
   <Selector Value 1> : <Statement 1>
   <Selector Value 2> : <Statement 2>
   ...
ELSE
   <Statements...>
END_CASE;
```

For example, the speed of the conveyor will be determined by the position of the manual switch:

```
CASE Switch_Position OF
```

```
   1, 2 : Conveyor_Speed := 100;
   3    : Conveyor_Speed := 200;
   4..6 : Conveyor_Speed := 300;
ELSE
   Conveyor_Speed := 0;
END_CASE;
```

CASE statement can also be used with enumerated variables. For example:

```
TYPE
   Position : (Slow, Normal, Fast);
END_TYPE
VAR
   Switch_Position : Position;
END_VAR
CASE Switch_Position OF
   Slow   : Conveyor_Speed := 100;
   Normal : Conveyor_Speed := 200;
   Fast   : Conveyor_Speed := 300;
ELSE
   Conveyor_Speed := 0;
END_CASE;
```

10. Iteration statement

Iteration statement causes a group of statements executed repeatedly based on the value of an integer constant or variable used as a counter to decide how many times the statements will be executed. It is also possible to do the iteration based on a Boolean logic being satisfied. Care must be taken to ensure that the execution does not cause an infinite loop. The following statements offer different methods for iteration.

10.1. FOR-DO Statement

This construct causes an iteration to be performed based on the value of an integer constant or variable of type INT, SINT or DINT. The general format of FOR-DO iteration construct is as follows.

```
FOR <Initial Value> TO <Final Value> BY <Increment> DO
   <Statements...>
END_FOR;
```

Note that in this statement BY <Increment> is optional, if omitted, the increment is assumed to be 1. The statements will be iterated and executed until the value of the counter variable reaches the final value. The counter variable check is made each time before executing the statements.
Consider the following example:

```
For I := 1 TO 9 BY 2 DO
   Motor_Fault[I] := FALSE;  // Assign FALSE to Array Element
END_FOR;
```

This example will assign FALSE to positions 1, 3, 5, 7, and 9 of the array variable MOTOR_FAULT. It is important that the value of the integer variable I is not changed or affected in any ways in the statements inside the loop as it may cause unpredictable behavior.

10.2. WHILE-DO Statement

This construct causes a group of statements to be executed repeatedly while a particular Boolean expression remains TRUE. Since the check is performed before the statements are executed, the program execution goes out of the loop when the expression becomes FALSE. The general format of WHILE-DO iteration construct is as follows.

```
WHILE <Boolean Expression> DO
   <Statements...>
END_WHILE;
```

Consider the following example:

```
J := 1;  // Initialize Iteration Variable
WHILE J <= 10 DO
   Total := Total + Valve_Average(Valve[J], Valve[J + 1]);
   J := J + 2;  // Increment Iteration Variable
END_WHILE;
```

This example will add the average of valve array elements 1 and 2 to Total until 9 and 10. It is important that the value of the iteration variable J must be modified inside the while loop so that the iteration will eventually be finished; otherwise, infinite loop will occur and it may cause fault in the controller.

10.3. REPEAT-UNTIL Statement

This is similar to the WHILE-DO construct except that the check is made after the statements are executed. The general format of REPEAT-UNTIL iteration construct is as follows.

```
REPEAT
   <Statements...>
UNTIL <Boolean Expression>
END_REPEAT;
```

The following example will yield the same result as the previous example.

```
J := 1;  // Initialize Iteration Variable
REPEAT
   Total := Total + Valve_Average(Valve[J], Valve[J + 1]);
   J := J + 2;  // Increment Iteration Variable
UNTIL J > 10
END_REPEAT;
```

11. Termination Statement

ST provides programmers two termination statements to exit and terminate execution of statements.

11.1. EXIT Statement

It is sometimes necessary to terminate the execution of an iteration routine on the occurrence of a certain event without completing the full sequence of the iteration. In such cases, the EXIT statement can be used to go out of the loop. The statement can only be used within an iteration. For example:

```
WHILE Tank_Level <= Max_Level DO
   Valve_Open := TRUE;
   IF Sensor_Fault = TRUE THEN
      Valve_Open := FALSE;
      EXIT;
   END_IF;
END_WHILE;
```

While the tank fluid level is less than the maximum value, the valve will be kept open. However, when there is a fault on the sensor, the valve will be closed and

the iteration terminates. Note that if this WHILE-DO statement is nested with another WHILE-DO statement, EXIT will only exit out of this inner loop.

11.2. RETURN Statement

RETURN statement is similar to EXIT but is used inside function or function block only. The statement causes the execution of a function to stop at the point where RETURN is called. It provides an early exit for function or function block, usually as the result of the evaluation of an IF statement. For example:

```
FUNCTION_BLOCK Valve_Control
   VAR_INPUT
      Tank_Level, Max_Level : REAL;
   END_VAR
   VAR_OUTPUT
      Valve_Open : BOOL;
      Out_Msg : STRING;
   END VAR
   IF Tank_Level > Max_Level THEN
      Valve_Open := FALSE;
      Out_Msg := 'Tank Level is Full';
      RETURN;
   ELSE
      Valve_Open := TRUE;
   END_IF;
   IF Tank_Level > Max_Level * 0.8 THEN
      Out_Msg := 'Tank Level is High';
   ELSE
      Out_Msg := 'Tank Level is Normal';
   END_IF;
END_FUNCTION_BLOCK
```

In this example, function block Valve_Control executes to open or close the valve according to the tank level. If the tank level is greater than the maximum level, valve is closed and tank is full message are returned and the function block is terminated immediately. If that is not the case, then the tank level is further checked to determine the appropriate message.

12. Best Practice for the Language

Although engineers and programmers are free to use the language to program in their own styles, good practices should be followed to increase the readability throughout the program. Upper or lower case and indentation can improve

program's readability. Other programming techniques can actually prevent potential runtime errors.

12.1. Variable Name

Unlike C or C++, ST is not case sensitive; however for clarity purpose and for you or someone else reading your code later, it is best that you keep the case consistent throughout your program. For example, upper case can be used for all expressions and title case can be used for all variables. A variable MotorSpeed can be used to represent motor speed. Use of underscore is also encouraged to separate many words, for example, Initial_Motor_Frequency can be used. Be certain that you use title case or underscore in a consistent manner. Also, abbreviation for long name can be used, such as StdDev for standard deviation. Consider the following two set of statements:

```
While Tanklevel <= MAXLEVEL do
   Valve_Open := true;
   IF SensorFault = TRUE then
      valve_open := False;
      exit;
   END_IF;
end_while;

WHILE Tank_Level <= Max_Level DO
   Valve_Open := TRUE;
   IF Sensor_Fault = TRUE THEN
      Valve_Open := FALSE;
      EXIT;
   END_IF;
END_WHILE;
```

The second set of statements is a lot easier to read than the first, all declared variables can easily be differentiated from the reserved keywords. Therefore, careful planning before writing your program is critical.

It is also important to note that meaningful names be chosen for variables. If you want to represent the status of pump 141, use Pump_Status_141, instead of Status141. Typically, I, J, and K are used as iterative variables and array indices, and Temp1 and Temp2, etc. are used as temporary variables. These variables, however, should not be extensively used and must be used consistently. Some programmers also prefer to attach the type of the variable as the prefix of the variable. For example, Int_Number_of_Run and Real_Initial_Temperature.

12.2. Indentation

Using correct indentation increases readability of your code. On the other hand, using no or incorrect can make the code very hard to read. Care must be taken to use consistent indentation throughout your program. A lot of ST programming interfaces allow you to set the length of the tab key, for example two spaces, so that you can quickly and correctly indent your code.

12.3. Avoid Spaghetti Code

Spaghetti code is a pejorative term for source code which has a complex and tangled control structure, especially one using many GOTOs, exceptions, threads, or other unstructured branching constructs. You should only use the GOTO statement in special circumstances (for example, for troubleshooting). Also, extended use of EXIT can cause unexpected spaghetti code. Therefore, always try to avoid EXIT and use other conditional statements such as IF and CASE to replace the branching construct. On the other hand, EXIT can be useful when performing debugging of your code, but care must be taken when you finalize the code.

12.4. Use Function

The most basic way of gaining organizational control is the use of subroutines. By dividing one big program into several smaller ones, each with a specific task to perform, the programmer can write one clean main program that calls these subroutines as needed. This is also quite efficient in that subroutines represent a form of reusable code such that different values can be computed at different times using the same piece of code.

12.5. Scope of Variable

Very often, you will have to decide whether to use local or global variables. Using global variables is convenient since you only have to declare them once. However, a very bad side effect is inadvertently changing the value of the variables in other part of the program and this problem is often difficult to debug. Using local variables wherever possible is therefore recommended.

12.6. Other Suggestions

There are a few other programming tips when writing your ST programs:

- Replace nested and complicated IF-THEN statement with CASE statement.
- Avoid infinite loop. Infinite loop can fault a processor.
- Do not use more than 3 nested loops.
- Do not use more than 3-dimensional array.
- Do not create too many unnecessary array elements. Because they are easy to create, programmers tend to create too many of them and use up a lot of system resource.
- If you often forget the operator precedence rules, you should use parentheses to make the order of evaluation explicit.