Mechatronics Engineering
ENMT201 Introduction to Mechatronics

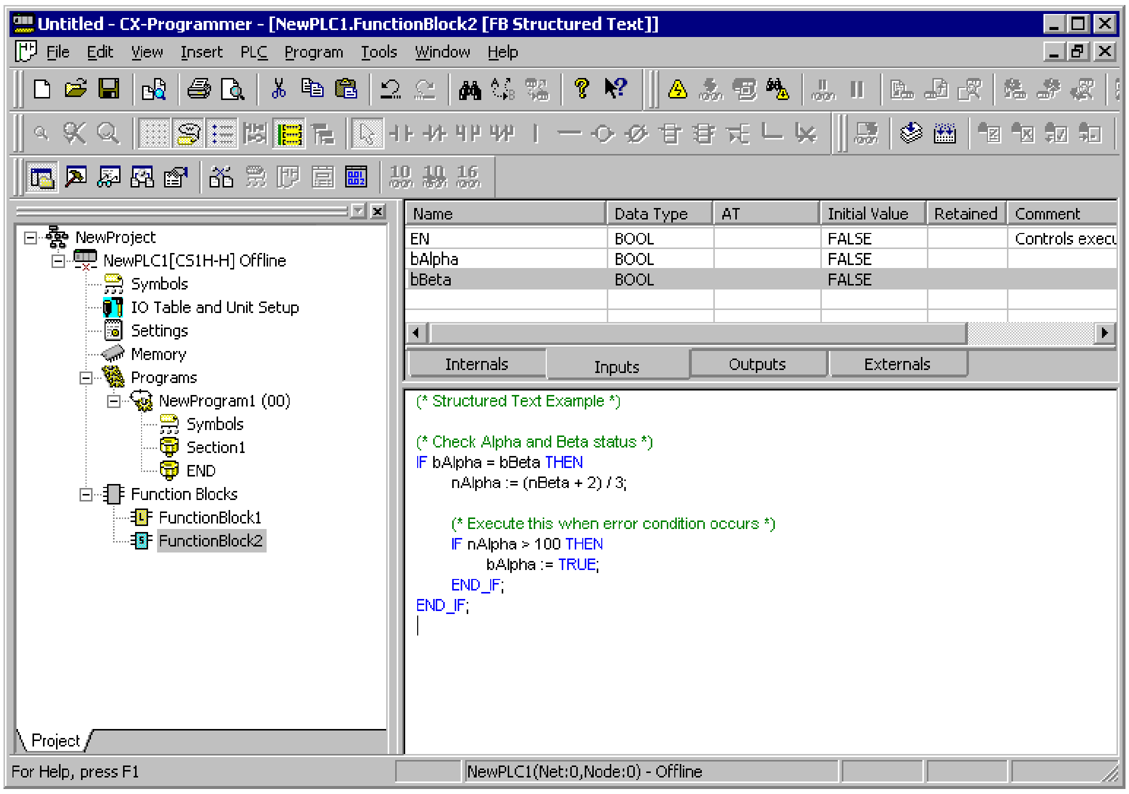
PLC Controlled Elevator

LAB 4 TUTORIAL

Contents:

1. The Structured Text (ST) Language

Function Block bodies can be written using the IEC61131-3 Structured Text language, a language very similar to PASCAL. The Structured Text Editor is a free format text editor component that uses color to improve readability of code. The image below shows an example of a Function Block written in the Structured Text Editor (the Structured Text Editor is displayed in the lower right corner of the image).



Name	Data Type	AT	Initial Value	Retained	Comment
EN	BOOL		FALSE		Controls execu
bAlpha	BOOL		FALSE		
bBeta	BOOL		FALSE		

```
(* Structured Text Example *)

(* Check Alpha and Beta status *)
IF bAlpha = bBeta THEN
	nAlpha := (nBeta + 2) / 3;

	(* Execute this when error condition occurs *)
	IF nAlpha > 100 THEN
		bAlpha := TRUE;
	END_IF;
END_IF;
```

2. Commands Supported in ST

The Structured Text Editor supports the following functionality in CX-Programmer. All functionality provided within the Structured Text Editor is compliant with the IEC61131-3 standards.

Structured Text Commands (Keywords)

- TRUE, FALSE.
- IF, THEN, ELSE, ELSIF, END_IF.
- DO, WHILE, END_WHILE.
- REPEAT, UNTIL, END_REPEAT.
- FOR, TO, BY, DO, END_FOR.
- CASE, OF, END_CASE.
- EXIT, RETURN.

Operators

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)
- Parenthesis (brackets)
- Array Indexing (square brackets [ ])
- Assignment Operator (:=)
- Less Than Comparison Operator (<)
- Less Than or Equal To Comparison Operator (<=)
- Greater Than Comparison Operator (>)
- Greater Than or Equal To Comparison Operator (>=)
- Equals Comparison Operator (=)
- Is Not Equal To Comparison Operator (<>)
- Bitwise AND (AND or &)
- Bitwise OR (OR)
- Exclusive OR (XOR)
- NOT (NOT)
- Exponentiation (**)

3. IF Statement

IF *expression1* THEN *statement-list1*

[ELSIF *expression2* THEN *statement-list2*]

[ELSE *statement-list3*]

END_IF;

The *expression1 and expression2* expressions must each evaluate to a boolean value. The *statement-list* is a list of several simple statements e.g. a:=a+1; b:=3+c; etc...

The **IF** keyword executes *statement-list1* if *expression1* is true; if **ELSIF** is present and *expression1* is false and *expression2* is true, it executes *statement-list2*; if **ELSE** is present and *expression1 or expression2* is false, it executes *statement-list3*. After executing *statement-list1*, *statement-list2 or statement-list3*, control passes to the next statement after the **END_IF**.

There can be several **ELSIF** statements within an **IF** Statement, but only one **ELSE** statement.

IF statements can be nested within other **IF** statements.

Example 1

```
IF a < 10 THEN
  b := TRUE;
  c := 100;
ELSIF a  > 20 THEN
  b := TRUE;
  c := 200;
ELSE
  b := FALSE;
  c := 300;
END_IF;
```

In this example, if the variable "a" is less than ten, then the variable "b" will be assigned the value of true (1), and the variable "c" will be assigned the value of 100. Control will then be passed to the program steps following the END_IF clause.

If the variable "a" is equal to or greater than 10 then control is passed to the ELSE_IF clause, and if the variable "a" is greater than 20, variable "b" will be assigned the value of true (1), and the variable "c" will be assigned the value of 200. Control will then be passed to the program steps following the END_IF clause.

If the variable "a" is between the values of 10 and 20 (i.e. both of the previous conditions IF and ELSE_IF were false) then control is passed to the ELSE clause, and the variable "b" will be assigned the value of false (0), and the variable "c" will be assigned the value of 300. Control will then be passed to the program steps following the END_IF clause.

Example 2

```
IF a THEN
  b :=  TRUE;
END_IF;
```

4. WHILE Statement

```
WHILE expression DO
   statement-list;
END_WHILE;
```

The **WHILE** *expression* must evaluate to a boolean value. The *statement-list* is a list of several simple statements.

The **WHILE** keyword repeatedly executes the *statement-list* while the *expression* is true. When the *expression* becomes false control passes to the next statement after the **END_WHILE**.

Example

```
WHILE a < 10 DO
  a := a + 1;
  b := b * 2.0;
END_WHILE;
```

In this example, the WHILE **expression** will be evaluated and if true (i.e. variable "a" is less than 10) then the **statement-list** (a := a + 1; and b := b * 2.0;) will be executed. After execution of the **statement-list** control will pass back to the start of the WHILE **expression**. This process is repeated while variable "a" is less than 10. When the variable "a" is greater than or equal to 10 then the **statement-list** will not be executed and control will pass to the program steps following the END_WHILE clause.

5. REPEAT Statement

REPEAT

  ***statement-list*;**

UNTIL ***expression***

END_REPEAT**;**

The **REPEAT** *expression* must evaluate to a boolean value. The *statement-list* is a list of several simple statements.

The **REPEAT** keyword repeatedly executes the *statement-list* while the *expression* is false. When the *expression* becomes true control passes to the next statement after **END_REPEAT**.

Example

```
REPEAT
   a := a + 1;
  b := b * 2.0;
UNTIL a > 10
END_REPEAT;
```

In this example, the statement-list (a := a + 1; and b := b * 2.0;) will be executed. After execution of the statement-list the UNTIL expression is evaluated and if false (i.e. variable "a" is less than or equal to 10) then control will pass back to the start of the REPEAT expression and the statement-list will be executed again. This process is repeated while the UNTIL expression equates to false. When the UNTIL expression equates to true (i.e. variable "a" is greater than 10) then control will pass to the program steps following the END_REPEAT clause.

6. FOR Statement

FOR control variable := integer expression1 **TO** integer expression2 [**BY** integer expression3] **DO**

***statement-list*;**

END_FOR;

The **FOR** *control variable* must be of an integer variable type. The **FOR** *integer expressions* must evaluate to the same integer variable type as the *control variable*. The *statement-list* is a list of several simple statements.

The FOR keyword repeatedly executes the *statement-list* while the *control variable* is within the range of *integer expression1* to *integer expression2*. If the **BY** is present then the *control variable* will be incremented by *integer expression3* otherwise by default it is incremented by one. The *control variable* is incremented after every executed call of the *statement-list*. When the *control variable* is no longer in the range *integer expression1* to *integer expression2,* control passes to the next statement after the **END_FOR**.

FOR statements can be nested within other FOR statements.

Example

```
FOR a := 1 TO 10 DO

  b := b + a;

END_FOR;
```

In this example, the FOR expression will initially be evaluated and variable "a" will be initialised with the value 1. The value of variable "a" will then be compared with the 'TO' value of the FOR statement and if it is less than or equal to 10 then the statement-list (i.e. b := b + a;) will be executed. Variable "a" will then be incremented by 1 and control will pass back to the start of the FOR statement. Variable "a" will again be compared with the 'TO' value and if it is less than or equal to 10 then the statement-list will be executed again. This process is repeated until the value of variable "a" is greater than 10 and then control will pass to the program steps following the END_FOR clause.

7. CASE Statement

CASE expression **OF**

***case label1* [, *case label2*] [.. *case label*3] : *statement-list1*;**

[ELSE

***statement-list2*]**

END_CASE;

The **CASE** *expression* must evaluate to an integer value. The *statement-list* is a list of several simple statements. The *case labels must be valid literal integer values e.g. 0, 1,+100, -2 etc..*

The **CASE** keyword evaluates the *expression* and executes the relevant *statement-list* associated with a *case label* whose value matches the initial *expression.* Control then passes to the next statement after the **END_CASE**. If no match occurs within the previous case labels and an **ELSE** command is present the statement-list associated with the **ELSE** keyword is executed. If the **ELSE** keyword is not present, control passes to the next statement after the **END_CASE**.

There can be several different *case labels* statements (and associated *statement-list)* within a **CASE** statement but only one **ELSE** statement.

The **,** operator is used to list multiple *case labels* associated with the same *statement-list.*

The **..** operator denotes a range case label. If the **CASE** *expression* is within that range then the associated *statement-list* is executed e.g. case label of *1..10* : a:=a+1; would execute the a:=a+1 if the **CASE** *expression* is greater or equal to one and less than ten.

Example

```
 CASE a OF

   2 :   b := 1;

   5 :   c := 1.0;

END_CASE;
```

In this example, the CASE statement will be evaluated and then compared with each of the CASE statement comparison values (i.e. 2 and 5 in this example).

If the value of variable "a" equals 2 then that statement-list will be executed (i.e. b := 1;). Control will then pass to the program steps following the END_CASE clause.

If the value of variable "a" equals 5 then that statement-list will be executed (i.e. c := 1.0;). Control will then pass to the program steps following the END_CASE clause.

If the value of variable "a" does not match any of the CASE statement comparison values then control will pass to the program steps following the END_CASE clause.

8. EXIT Statement

```
 WHILE expression DO

   statement-list1;

   EXIT;

END_WHILE;

statement-list2;

 REPEAT

   statement-list1;

   EXIT;
```

UNTIL ***expression***

END_REPEAT;

statement-list2;

FOR *control variable* := *integer expression1* TO *integer expression2* [BY *integer expression3*] DO

***statement-list1*;**

EXIT;

END_FOR;

statement-list2;

The *statement-list* is a list of several simple statements.

The EXIT keyword discontinues the repetitive loop execution to go to the next statement, and can only be used in repetitive statements (WHILE, REPEAT, FOR statements). When the EXIT keyword is executed after s*tatement-list1* in the repetitive loop, the control passes to *statement-list2* immediately.

Example

```
 WHILE a DO
  IF c = TRUE THEN
       b:=0;EXIT;
  END_IF;
  b:=b+1;
 IF b>10 THEN
    a:=FALSE;
 END_IF;
END_WHILE;
d:=1;
```

If the first IF expression is true (i.e. variable "c" is true), the statement-list (b := 0; and EXIT;) is executed during the execution of the WHILE loop. After the execution of the EXIT keyword, the WHILE loop is discontinued and the control passes to the next statement (d : = 1;) after the END_WHILE clause.

9. RETURN Statement

statement-list1;

RETURN;

statement-list2;

The *statement-list* is a list of several simple statements.

The **RETURN** keyword breaks off the execution of the inside of the Function Block after *statement-list1*, and then the control returns to the program which calls the Function Block without executing *statement-list2*.

Example

```
 IF a_1 * b > 100 THEN

   c := TRUE; RETURN;

END_IF;

IF a_2 * (b + 10) > 100 THEN

    c := TRUE; RETURN;

IF a_3 * (b + 20) > 100 THEN

    c := TRUE;

END_IF;
```

If the first or second IF statement is true (i.e. "a_1 * b" is larger than 100, or "a_2 * (b + 10)" is larger than 100), the statement (c := TRUE; and RETURN;) is executed. The execution of the RETURN keyword breaks off the execution of the inside of the Function Block and the control returns to the program which calls the Function Block.

10. ST Conversion Instructions

Throughout a Structured Text expression, the data types must be consistent to be validated correctly (e.g. **A := B + C**; where A, B and C are INT data types). However, it is possible to use different data types within a Structured Text expression by using appropriate 'Conversion Instructions' to convert the necessary parts of the expression.

Conversion functions are in the format **<input data type>_TO_<output data type>** where the 'input data type' is the current data type of that part of the expression and 'output data type' is the required data type (e.g. if C is a WORD in the example then the correct syntax would be **A := B + WORD_TO_INT(C)**). **INT_TO_WORD** converts integer to word.