# HMC series controller

*Basic Programming Instructions Manual*


Thank you very much for purchasing our HMC series products.

This manual describes the use and maintenance of the HMC series controller, basic programming instruction, etc. Please read this manual carefully before installing, wiring, using, maintaining, and checking the product.

Please keep this manual in a safe place and deliver it to the end user.

# Statement

The contents of this user manual are subject to change without prior notice.

If you find any suspicion, error, or omission in the content of this user manual, please contact us to change it.

If there are any error or missing pages in this user manual, we will replace them for you.

HMC series controller programming basic instruction manual

Publisher ： Guangzhou Auctech Automation Technology Ltd

Headquarters Office ： Hongshi Business Building, 11 Kehua Road, SCI-TECH Industry Park, Taihe Town, Baiyun District, Guangzhou, CHINA

# Change Log

| Revision | Change Information | Originator | Date |
|---|---|---|---|
| V1.0 | New Release | mxh | 2021-05 |
| V1.1 | Organize content formatting and adjust table formatting | czm | 2022-09 |
| V1.2 | Change of HMC Series Controller Related Manual Form and Notes | czm | 2023-02 |
| V1.3 | Change the template and font of the entire manual | czm | 2023-06 |
| V1.4 | Change company address and related information | czm | 2023-09 |

HMC Series Controller Related Manuals

The following table shows the information, please select the manual according to your needs

| No.# | Manual Name | Description |
|---|---|---|
| 1 | HMC Series Controller and IO Unit Selection Manual | To understand the basic functions of controller products |
| 2 | HMC Series Controller Software Getting Started Manual | Software acquisition, installation, getting started tutorial |
| 3 | HMC S3 Series Controller User Manual | Explanation on the basic usage of S3 series controllers, etc. |
| 4 | HMC S4 Series Controller User's Manual | Explanation on the basic use of S4 series controllers, etc. |
| 5 | HMC G300 Series Controller User Manual | About the basic usage and functions of the G3-6 series controllers and other operating instructions |
| 6 | HMC series controller basic programming instruction manual (this book) | Understanding of the concepts and functions of basic controller programming instructions |
| 7 | HMC series controller motion control command manual | Understanding of basic concepts and functions of motion control commands |

*Note: All the above information can be found on the official website: www.auctech.com.cn

# Table of Contents

# Section 1 Safety Precautions

## ■ Safety Commands

- Please read and follow these safety precautions when installing, operating, or maintaining the product.
- For personal and equipment safety, please follow all safety precautions described in the markings and manuals on the product when installing, operating, and maintaining the product.
- The "Caution", "Warning" and "Danger" items in the manual do not represent all safety precautions to be observed, but only in addition to all other safety precautions.
- This product should be used in an environment that meets design specifications, otherwise it may cause a malfunction due to failure to comply with the relevant safety precautions.
- The product quality warranty does not cover abnormal function or damage to parts caused by the regulations.
- We will not bear any legal responsibility for personal safety accidents and property damage caused by illegal operation of the product.

| Security Level Definition | |
|---|---|
| Danger | Indicates a potentially hazardous situation which, if not avoided,could result in death or serious injury. Additionally, there maybesevere property damage. |
| Caution | If not used in accordance with the regulations, may cause fires, serious personal injury, or even death! |
| Warning | Failure to use in accordance with the regulations may result in moderate personal injury or minor injury, as well as the occurrence of equipment damage! |

| When products arrive and are stored | |
|---|---|
| Warning | ● If the product and product accessories are damaged when opening the box, please do not install them and contact our company or your supplier immediately.<br>● Check carefully whether the arriving product and the ordered product model match, and whether the product and product accessories are included. |

| | |
|---|---|
| Caution | ● Do not stack multiple of this product on top of each other as this may cause injury or malfunction.<br>● Do not store in places exposed to direct sunlight, places where the ambient temperature exceeds the temperature conditions for storage, places where the relative humidity exceeds the humidity condition for storage, places where there is a large temperature difference, places where there is high condensation, places near corrosive gases, places where there are flammable gases, places where there is a large amount of dust, dirt, salt or metal dust, places where water, oil or medicine drip, places where vibration or shock can affect the main body of product; otherwise it can lead to fire, Electric shock or machine damage.<br>● Do not hold the cable or motor shaft for weight holding, as this may result in injury or malfunction. |

| When designing the system | |
|---|---|
| Danger | ● If the rated load of current is exceeded or the load is short-circuited for a long period of time resulting in over-current, the product may start smoking or catch fire. safety devices such as fuses, or circuit breakers should be set externally. |
| Warning | ● Be sure to design safety circuits to ensure that the product system will still work safely if the external power supply is lost, or the product fails.<br>● For safe operation of the equipment, please design external protection |

| | |
|---|---|
| | circuits and safety mechanisms for output signals related to major accidents. |
| Caution | ● Be sure to install emergency brake circuits, protection circuits, interlock circuits for forward and reverse operation, and position upper and lower limit interlock switches to prevent damage to the machine in the external circuit of the product.<br>● The product may shut down all outputs after detecting abnormalities in its own system; when part of the controller circuit fails, it may cause its output to be uncontrolled. To ensure normal operation, a suitable external control circuit needs to be designed.<br>● If the output unit such as relay or transistor of the product is damaged, the output will not be controlled to the ON or OFF state.<br>● The product is designed to be used in indoor, overvoltage class II electrical environments, and its power system level should have lightning protection devices to ensure that lightning overvoltage is not applied to the product's power input or signal input, control output and other ports to avoid damage to equipment. |

| **When the product is installed** | |
|---|---|
| Danger | ● Only maintenance professionals with adequate electrical knowledge and training related to electrical equipment should install this product.<br>● For the product with open equipment, please install in the control cabinet with door lock (product cabinet shell protection > IP20), only operators with sufficient electrical knowledge and training related to electrical equipment can open the product cabinet. |
| Warning | ● When disassembling the product, the external power supply used for the system must be completely disconnected before performing the operation. Failure to disconnect all power supplies may result in electric shock or product failure and malfunction.<br>● While dissembling the product, the power and the power indicator must be turned off for at least 5 minutes, before disassembling the driver. Otherwise, the residual voltage may cause electric shock.<br>● Do not use the product in the following places: places with dust, oil fumes, conductive dust, corrosive gases, combustible gases; places exposed to high temperature, condensation, wind, and rain; places with vibration and shock. Electric shock, fire, and misuse can also cause damage and deterioration of the product! |
| Caution | ● Avoid metal shavings and wire tips falling into the ventilation holes of the product during installation, this may cause fire, malfunction, and misoperation.<br>● After installation, ensure that there is no foreign matter on the ventilation surfaces, otherwise it may lead to poor heat dissipation and cause fire, malfunction and misoperation.<br>● When installing, make a tight connection to the respective connector and lock the product connection hook firmly. If the products are not installed properly, it may lead to misoperation, malfunction and dislodgement. |

| **When wiring products** | |
|---|---|
| Danger | ● Only maintenance professionals with adequate electrical knowledge and training related to electrical equipment should perform the wiring of this product. |
| Warning | ● During wiring operations, the external supply power used by the system must be completely disconnected before operation. Failure to disconnect all of them may result in electric shock or equipment malfunction or misoperation.<br>● When powering up and running after the wiring operation, the terminal cover that comes with the product must be installed. Failure to install the terminal cover may result in electric shock. |

| | |
|---|---|
| | ● Check the type of interface to be connected before connecting the cable correctly. If the wrong interface is connected or the wiring is incorrect, it may cause the product or external equipment to malfunction.<br>● The cable terminals should be well insulated to ensure that the insulation distance between the cables is not reduced after the cables are installed to the terminal block. Otherwise, it will lead to electric shock or equipment damage.<br>● Avoid metal shavings and wire tips falling into the ventilation holes of the controller when wiring, which may cause fire, malfunction, and misoperation!<br>● The bolts on the terminal blocks should be tightened within the specified torque range. Untightened terminal bolts may result in short circuit, fire, or malfunction. Over-tightening the bolts may damage the bolts and the product, resulting in dislodgement, short circuit, fire, or false operation. |
| ⚠ Caution | ● The specification and installation method of the external wiring of the equipment should meet the requirements of local power distribution regulations.<br>● To ensure the safety of the equipment and the operator, the equipment needs to be reliably grounded using cables of sufficient wire size.<br>● For connections using connectors and external devices, press fit, crimp, or properly solder using the tool specified by the manufacturer. A poor connection may result in a short circuit, fire, or malfunction.<br>● If the product is labeled to prevent foreign objects from entering the product during wiring, such as the wiring head. Do not remove this label during wiring operations. Before starting system operation, be sure to remove the label to facilitate heat dissipation.<br>● Please do not bundle the control and communication cables with the main circuit or power supply cables, etc. The alignment should be more than 100mm apart, otherwise the noise may lead to misoperation.<br>● For applications with serious interference, please use shielded cables for input or output of high frequency signals to improve the system's anti-interference capability. |

| **Before powering on the product** | |
|---|---|
| ⚠ Danger | ● Before powering on, please make sure the product is well installed, wired firmly and the motor unit is allowed to restart.<br>● Before powering on, please confirm that the power supply meets the product requirements to avoid causing damage to the product or starting a fire.<br>● It is strictly forbidden to open the product cabinet door or product protective cover, touch any terminals of the product, disassemble any device or parts of the product in the energized state, otherwise there is a risk of electric shock.<br>● Make sure that no one is around the product, the motor, or the machinery before powering it on, as this may result in injury or death! |
| ⚠ Warning | ● After the wiring operation and parameter setting are completed, please conduct a test run of the machine to confirm that it can operate safely, otherwise it may lead to injury or equipment damage!<br>● Before powering on, please make sure that the rated voltage of the product is the same as the power supply voltage. If the power supply voltage is used incorrectly, there is a risk of fire! |

| When operating and maintaining | |
|---|---|
| Danger | ● Only maintenance professionals with adequate electrical knowledge and training on electrical equipment can perform the operation and maintenance of the products.<br>● Do not touch the terminals when the power is on, as this may cause electric shock or malfunction.<br>● When the motor or equipment is running, please never touch its rotating parts, otherwise it may lead to serious personal safety accidents. |
| Warning | ● When cleaning the product or retightening the bolts on the terminal block or the connector mounting bolts, the external supply power used by the system must be completely disconnected. Failure to do so may result in electric shock.<br>● When disassembling the product or connecting or removing the communication cable, the external supply power used by the system must be completely disconnected first. Failure to disconnect all of them may result in electric shock or false operation.<br>● While dissembling the product, the power and the power indicator must be turned off for at least 5 minutes, before disassembling the driver. Otherwise, the residual voltage may cause electric shock. |
| Caution | ● For online modification, forced output, RUN, STOP, etc., you must read the user's manual and confirm its safety before performing the relevant operations.<br>● Be sure to disconnect the power before loading and unloading expansion cards, modules, and other components! |

| When the product is scrapped | |
|---|---|
| Caution | ● Please dispose of them as industrial waste; when disposing of batteries, do so separately according to the ordinances established by each region to avoid property damage or human injury!<br>● End-of-life products should be treated and recycled in accordance with industrial waste treatment standards to avoid polluting the environment. |

# Section 2 Basic Instructions

In a programmable controller, the commands and combinations of commands that enable the CPU to perform a certain operation or achieve a certain function are called instructions, and the collection of instructions is called the instruction system. The instruction system is the bridge between programmable controller hardware and software and is the basis of programmable controller programming. This section will introduce the bit logic instructions, timer instructions, counter instructions, data processing instructions, operation instructions, and data conversion instructions.

## 2. 1 List of basic instructions

| Total type | Category | Instruction | Name | Function Summary |
|---|---|---|---|---|
| Bit Logic Instructions | Basic Logic Instructions | AND | And | Bit ‘and’, Boolean 'and' |
| | | OR | Or | Bit 'or', Boolean 'or' |
| | | NOT | Not | Bit 'not', Boolean 'not' |
| | | XOR | Exclusive Or | By-bit 'iso-or', Boolean 'iso-or' two |
| | Set priority and reset priority flip-flop instructions | SR | Set as priority trigger | Set bistable flip-flop, set priority |
| | | RS | Reset Priority Trigger | Reset bistable flip-flop, reset priority |
| | Edge detection command | R_TRIG | Rising edge trigger | For detecting rising edges |
| | | F_TRIG | Falling edge triggering | For detecting falling edge |
| Timer commands | Timer | TP | Pulse Timer | Pulse Timing |
| | | TON | Power-on delay timer | Power on delay timing |
| | | TOF | Power failure delay timer | Power failure delay timing |
| | | RTC | Real Time Clock | Starts at the given time and returns the current date and time |
| Counter command | Counter command | CTU | Increase counter | Counting |
| | | CTD | Less Counter | Subtractive count |
| | | CTUD | Increase/decrease counter | Count up/down |
| Data processing instructions | Select operation command | SEL | Two-choice command | Select one of the two input data as output |
| | | MAX | Take the maximum value | Maximum value function |
| | | MIN | Take the minimum value | Minimum value function |
| | | LIMIT | Restricted values | Limit value output |
| | | MUX | Multiple Choice One | Multiplexer operation |
| | Compare commands | = | Equivalent | When the first operand is equal to the second operand, the Boolean operator returns TRUE |
| | | <> | No Equivalent | When the first operand is not equal to the second operand, the Boolean operator returns TRUE |
| | | > | Greater than | When the first operand is greater than the second operand, the Boolean operator returns TRUE |
| | | >= | Greater than or equal to | When the first operand is greater than or equal to the second operand, the Boolean operator returns TRUE |
| | | < | Less than | When the first operand is smaller than the second operand, the Boolean operator returns TRUE |
| | | <= | Less than or equal to | When the first operand is less than or equal to the second operand, the Boolean operator returns TRUE |
| | Shift command | SHL | Shift left by position | The operand is shifted to the left by bit, the left shift out bit is |

| Total type | Category | | Instruction | Name | Function Summary |
|---|---|---|---|---|---|
| | | | | | not processed, and the right empty bit is automatically filled with 0 |
| | | | SHR | Right shift by position | The operand is shifted to the right by bit, the right shift out bit is not processed, and the left empty bit is automatically filled with 0 |
| | | | ROL | Cyclic left shift | The operand is cyclically shifted left by bit, and the bit shifted out on the left is directly added to the lowest bit on the right |
| | | | ROR | Circular right shift | The operand is cyclically shifted right by bit, and the bit shifted out on the right is directly added to the highest bit on the left. |
| Transport count Refers to make | Assignment Instructions | | := | Assignment | Assigning the value of a constant or variable to another variable |
| | Arithmetic operations Instruction | | + | add up | Addition instruction, two (or more) variables or constants are added together |
| | | | - | phase minus | Subtraction instruction, two variables or constants are subtracted from each other |
| | | | × | multiply | Multiply instruction, two (or more) variables or constants are multiplied together |
| | | | / | phase division | Division instruction, divide two variables or constants |
| Transport count Refers to make | Arithmetic operations Instruction | | MOD | Remainder | The remainder of dividing two variables or constants, which is an integer data |
| | Mathematical operations commands | Basic Instructions | ABS | Absolute value command | This function instruction is used to calculate the absolute value of a number, and has no relationship with the positive or negative sign of the number |
| | | | SQR | Square root command | Square root of a non-negative real number |
| | | | EXP | Exponential | Returns the power of e (the base of the natural logarithm), e being a constant of 2.71828 |
| | | | LN | Natural logarithm instruction | Returns the natural logarithm of a number |
| | | | LOG | Common logarithmic instructions | Returns the logarithm of the base 10 |
| | | | SIN | Sine command | Sine function |
| | | | COS | Cosine command | cosine function |
| | | | ACOS | Inverse cosine command | Cosine radian (inverse cosine function) |
| | | | ASIN | Chord command anyway | Sine of arc (inverse sine function) |
| | | | TAN | Tangent command | tangent function |
| | | | ATAN | Cut command anyway | Tangent radian (inverse tangent function) |
| | | Expansion Instructions | ATAN2 | Angle of the point X | Azimuth from the origin to the point (x,y), i.e. the angle with the x-axis |
| | | | Ceil | Rounding up | Rounding up to integer |
| | | | CRC16Check | 16-bit CRC check | 16-bit CRC check |
| | | | DegToRad | Angle to arc | Angle to arc |
| | | | Floor | Rounding down | Rounding down to the nearest integer |
| | | | Fmod | Floating point data modulus | Floating point data modulus |

| Total type | Category | | Instruction | Name | Function Summary |
|---|---|---|---|---|---|
| | | | ParamPeriodLimits | Periodic value processing, automatic modulus, negative values to positive | Periodic value processing, automatic modulus, negative values to positive |
| | | | PeriodLimit | Periodic value processing | Periodic value processing |
| Transport count Refers to make | Mathematical operations commands | Expansion Instructions | RadToDeg | Radian to Degree | Radian to Angle degree |
| | | | Round | Float rounding | Rounding of float numbers, with decimal places reserved |
| | | | Saturation | Saturation function | Saturation function with upper and lower limits of 1, -1 |
| | | | Signum | Positive and negative sign functions | Positive and negative number function, positive number return 1.0, negative number return -1.0, zero return 0. |
| | | | Statistics_N_Lreal | Find the average of the last N data | The average of the last N times data, less than N times to take the average of all recent data, earlier than N times data will be cleared |
| | | | Angel | Find the angle between the three points | Find the angle of three points using the dot product principle |
| | | | Cross | Find the cross product | Find the cross product |
| | | | Dot | Find vector dot product | Find the value of the vector op point multiplied by the vector oq |
| | | | INV | Find the inverse matrix of matrix a | Inverse matrix of nth-order real matrix a by the all-choice principal Gaussian-approximation elimination method |
| | | | InvertGaussJordan | Find the inverse matrix of matrix a | Inverse matrix of nth-order real matrix a by the all-choice principal Gaussian-approximation elimination method |
| | | | InvM | Find the generalized inverse matrix | Find the generalized inverse matrix |
| | | | MatrixAdd | Find matrix plus | Find matrix plus |
| | | | MatrixMultiplyN | Find matrix multiplication | Find matrix multiplication |
| | | | MatrixSub | Find the matrix minus | Find the matrix minus |
| | | | MatrixTranp | Matrix transpose | Matrix transpose |
| | | | Norm | Find the norm of the matrix | Find the matrix parametrization |
| | Address operation instructions | | SIZEOF | Data Type Size | Perform this function to determine the number of bytes required for the given data type |
| | | | ADR | Address Operators | Get the memory address of the input variable and output |
| | | | BITADR | Bit Address Operators | Returns the bit address information offset of the allocated variable |
| Data conversion instructions | Data type conversion instructions | | BCD_TO_BYTE | BCD to BYTE | BCD to BYTE |
| | | | BCD_TO_DWORD | BCD to DWORD | BCD to DWORD |
| | | | BCD_TO_INT | BCD to INT | BCD to INT |
| | | | BCD_TO_WORD | BCD to WORD | BCD to WORD |
| | | | BYTE_TO_BCD | BYTE to BCD | BYTE to BCD |
| | | | DWORD_TO_BCD | DWORD to BCD | DWORD to BCD |
| | | | INT_TO_BCD | INT to BCD | INT to BCD |
| | | | WORD_TO_BCD | WORD to BCD | WORD to BCD |

| Total type | Category | | Instruction | Name | Function Summary |
|---|---|---|---|---|---|
| | | | BOOL_TO_INT | BOOL to INT | BOOL to INT |
| | | | BOOL_TO_STRING | BOOL to STRING | BOOL to STRING |
| | | | BOOL_TO_TIME | BOOL to TIME | BOOL to TIME |
| | | | BOOL_TO_TOD | BOOL to TOD | BOOL to TOD |
| | | | BOOL_TO_DATA | BOOL to DATA | BOOL to DATA |
| | | | BOOL_TO_DT | BOOL to DT | BOOL to DTime |
| | | | BYTE_TO_BOOL | BYTE to BOOL | BYTE to BOOL |
| | | | BYTE_TO_INT | BYTE to INT | BYTE to INT |
| | | | BYTE_TO_TIME | BYTE to TIME | BYTE to TIME |
| | | | BYTE_TO_DT | BYTE to DT | BYTE to DTime |
| Data conversion instructions | Data type conversion instructions | | BYTE_TO_REAL | BYTE to REAL | BYTE to REAL |
| | | | BYTE_TO_STRING | BYTE to STRING | BYTE to STRING |
| | | | WORD_TO_USINT | WORD to USINT | WORD to USINT |
| | | | WORD_TO_TIME | WORD to TIME | WORD to TIME |
| | | | WORD_TO_DT | WORD to DT | WORD to DTime |
| | | | REAL_TO_INT | REAL to INT | REAL to INT |
| | | | TIME_TO_STRING | TIME to STRING | TIME to STRING |
| | | | TIME_TO_DWORD | TIME to DWORD | TIME to DWORD |
| | | | DT_TO_BYTE | DT to BYTE | DT to BYTE |
| | | | DATA_TO_INT | DATA to INT | DATA to INT |
| Ladder diagram (LD)/function block (FBD) instruction | Connecting elements | | Contacts | Normally open contacts | Normally open contacts |
| | | | | Normally closed contacts | Normally closed contacts |
| | | | | Insert right contact | Insert right contact |
| | | | | Insert parallel lower normally open contact | Insert parallel lower normally open contact |
| | | | | Insert parallel lower normally closed contact | Insert parallel lower normally closed contact |
| | | | | Inserted in parallel with the upper normally open contact | Inserted in parallel with the upper normally open contact |
| | | | Coils | Coils | Coils |
| | | | | Positioning coil | Positioning coil |
| | | | | Reset | Reset |
| Ladder diagram (LD)/function block (FBD) instruction | Connecting elements | | Functions/ Function Blocks | Insert operator block | Insert operator block |
| | | | | Insert Empty Block | Insert empty Block |
| | | | | Insertion of an operator block with EN/ENO | Insertion of an operator block with EN/ENO |
| | | | | Insert with EN/EN0 | Insert with EN/EN0 |
| Structured Text (ST) Instructions | Command statements | Assignment Statements | := | Statement assignment | Assign the data on the right to the data on the left |
| | | Function and function block control statements | Function block/function name (); | For function/function block control statements | Function block and function calling format |
| | | Select statement | IF | IF Selection | Judgment statement, meet the condition to execute. If |

| Total type | Category | | Instruction | Name | Function Summary |
|---|---|---|---|---|---|
| | | | | | condition then, end_if end |
| | | | CASE | CASE Selection | Select statement. Case select condition of, end_case end |
| | | Iterative Statements | FOR | FOR loop | For loop, e.g. for I :=0 to 100 by 1 |
| | | | WHILE | WHILE loop | While(condition) {}, satisfies the condition and keeps executing, but does not exit |
| | | | REPEAT | REPEAT loop | REPAEAT<command>UNTIL< Boolean expression>END_REPEAT. |
| | | Jump statements | EXIT | Exit | Exit current loop |
| | | | CONTINUE | Continue | Continue to implement |
| | | | JMP | Jump | Jump Instructions |
| | | Return statement | RETURN | Back | Returns the result of the function. Or exit to another location |
| | | NULL | NULL statement | Empty statements | Judgment condition is null, return null |
| String command | String command | Basic Instructions | CONCAT | Concat | Concatenate two strings |
| | | | DELETE | Delete | Remove multiple characters from a string |
| | | | FIND | Find | Search for the position of a partial string in a string |
| String command | String command | Basic Instructions | INSERT | Insert | Inserting a string into another string at a specific location |
| | | | LEFT | Left of string | Returns the number of specific characters in the string starting from the left |
| | | | LEN | Length of string | Returns the number of characters in the string |
| | | | MID | Number of specific characters at specific locations | Returns the number of characters in a string from a specific position |
| | | | REPLACE | Replace string | Replace a specific number of characters of a string with another string |
| | | | RIGHT | Right of string | Returns the number of specific characters in the string starting from the right |
| | | Expansion Instructions | ByteToHexString | byte to hexadecimal string | Byte to hexadecimal string, output String |
| | | | HexToByte | Convert hexadecimal string to byte value | Hexadecimal string to byte value |
| | | | HexToDword | Hexadecimal string to Dword value | Hexadecimal string to Dword long character value |
| | | | IsHex | Byte to hexadecimal string conversion | Byte to hexadecimal string conversion |
| | | | Split | Split String | Split a string by a character, use a string(255) array to receive the split string related instructions |
| | | | StringToWS | String to WString | String to WString |
| | | | Trim | Remove the beginning and end of the string or other characters | Removes the beginning and end of the string or other characters. When the function succeeds, it returns the string with the first and last part of the string removed |
| | | | Concat_p | String Splicing | |
| String command | String command | Expansion Instructions | Delete_p | Delete String | Delete characters of the specified length from a string |
| | | | Find_p | Find String | Find the string from the source string and return the position |
| | | | IsSpace_p | Determine the input string | Determine if the input |

| Total type | Category | | Instruction | Name | Function Summary |
|---|---|---|---|---|---|
| | | | | | character is a carriage, return or tab, etc. |
| | | | Left_p | Get the string on the left | Get the specified number of characters from the left side of the string to the target string, starting from the first character |
| | | | Len_p | String length | Get the length of the string |
| | | | Mid_p | Get a specific string | Get the specified number of characters from the source string at the specified position to the target string |
| | | | Right_p | Get the string on the right side | Get the specified number of characters from the right side of the source string to the target string |
| | | | ToLower_p | String to lowercase | Convert English characters in a string to lowercase |
| | | | ToUpper_p | String to uppercase | Convert English characters in a string to uppercase |
| | | | Trim_p | Delete whitespace characters from both ends of a string | Whitespace characters are removed from both ends of a string. The whitespace characters in this context are all whitespace characters (space, tab, no-breakspace, etc.) and all line terminator characters (e.g. LF, CR, etc.) |
| Time/ Moment command | Get system time command | | GetCycleTimeMS | Get Task Period | Get system time in ms |
| | | | GetCycleTimeS | Get Task Period | Get system time in s |
| | Get time type command | | GetTimeDT | Get time type local time | Get time type local time |
| | Get string type command | | GetTimeS | Get the local time of the string type | Get the local time of the string type |
| File manipulation commands | File Operations Instruction | | CopyFile | Copy file | Copy the file to the specified path |
| | | | CreateFile | Create file | Create a file to the specified path |
| | | | DeleteFile | Delete file | Specify a path to delete a file |
| | | | GetFileList | Get file list | Get file list (return file name only) function return value for the number of valid files obtained |
| | | | GetFileList2 | Get file list | The return value of the Get File List (with file information) function is the number of valid files obtained |
| | | | HasFile | Check the file | Check if the file exists |
| | | | ReadFile | Read file | Read file function returns the number of bytes read from the file Note: If the read is Chinese characters need to use the following function to convert |
| | | | ReadFileS | Read a line of data from a file | Read a line of data from a file, the return value is the number of bytes read |
| | | | ReadFileWS | Read a line of data from a file | Read a line of data from a file, the return value is the number of bytes read |
| | | | WriteFile | Write data to file | Write data to file |
| EtherCAT communication commands | SDO Operation Instruction | | ETC_CO_SdoReadDWord | Read SDO parameters | Read SDO parameters, read servo parameters |
| | | | ETC_CO_SdoWriteDWord | Write SDO parameters | Write SDO parameters, change servo parameters |
| | Reset Module | | MC_ Reset ETCSlave | Reset ECT unit | Reset ECT unit |
| | | | MC_Reset | Resetting shaft and drive | Resetting shaft and drive |

| Total type | Category | Instruction | Name | Function Summary |
|---|---|---|---|---|
| | | Drive | | |
| | | MC_ ResetMaster | Resetting the master | Reset Master EtherCAT_Master_SoftMotion |

# Section 3 Basic Descriptions

## 3. 1 Bit Logic Instructions

The bit logic instructions handle the logical changes of Boolean values "1" and "0." CodeSys provides bit logic instructions including basic logic operations, set/reset priority flip-flops and rising/falling edge detection instructions, as listed in Table 3-1.

| Bit Logic Instructions | Graphical language | Textualized Language | Description |
|---|---|---|---|
| Bit Logic Instructions | | AND | And |
| | | OR | Or |
| | | NOT | Not |
| | | XOR | Exclusive OR |
| | | SR | Set as priority trigger |
| | | RS | Reset Priority Trigger |
| | | R_TRIG | Rising edge trigger |
| | | F_TRIG | Falling edge triggering |

Table 3-1 Table of graphical and textual instructions for bit logic instructions

## 3. 1. 1 Basic logic instructions

The basic bit logic instructions include "AND", "OR", "NOT", and "XOR". In CodeSys, the functions can be divided into bitwise logic operations and Boolean logic operations.

- Bitwise logic operation performs Boolean logic operation on the corresponding bits of two integer data one by one and returns the compatible integer result.
- Boolean logic operation performs a logical operation on two Boolean type data.

### 3. 1. 1. 1 "AND" by bit

Function: The "AND" instruction compares the corresponding bits of two integers. When the corresponding bits of two numbers are 1, the corresponding result bit is "1"; when the corresponding bits of two integers are "0" or one of them is "0", the corresponding result bit is "0". When the corresponding bit of two integers is "0" or one of them is "0", the corresponding result bit is "0". The "with" logic is shown in Table 3 -2.

| Input 1 | Input 2 | Results |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Table 3-2 "AND" instruction logic relationship table

[Example 3.1] Create a POU, declare two integer variables iVar1 and iVar2, assign 100 and 200 to them respectively, and perform bitwise operations on these two variables, output the result save to iResult, the variable declaration is shown below, and the CFC programming code is shown in Figure 3.1.

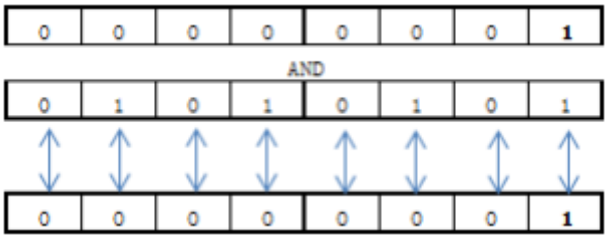



Figure 3.1 Example of the "AND" logic instruction program by bit

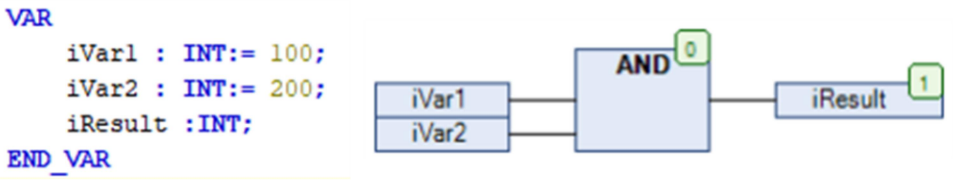



Figure 3.2 By bit “AND” operation CFC program diagram

The decimal number 1 corresponds to the binary number 00000001, and the decimal number 85 corresponds to the binary number 01010101. According to the definition of the operation by bit and operation, each individual bit is "AND" one by one, result is 00000001, which is the decimal value 1, as shown in Figure 3.2.

3. 1. 1. 2 Boolean "AND"

The Boolean "AND" operation is used to calculate the "AND" result of two Boolean expressions. When the result of both Boolean expressions is true, it returns true, and if one of them is false, it returns false.

[Example 3.2] Create a POU, use the Boolean "AND" operation, determine the return value of the operation, the program is shown in Figure 3.3.

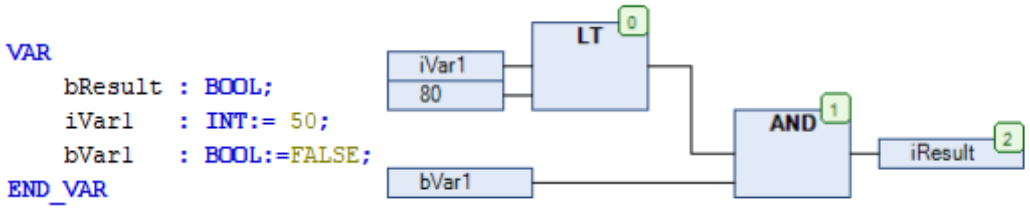



Figure 3.3 Boolean “AND” arithmetic CFC program diagram

Since 50 is indeed less than 80, the condition in the first half of the program is true, but the default value of bVar is FALSE, so the result of 0 and 1 is 0. The program runs with the result that bResult is FALSE.

3. 1. 1. 3 "OR" by bit

The "OR" by bit instruction compares the corresponding bits of two integers. When one of the corresponding bits of two numbers is "1" or both are "1", the corresponding result bit is returned as "1". When the corresponding bit of two integers is "0", the corresponding result is "0". The "OR" logic is shown in Table 3 -3.

[Example 3.3] Create a POU, perform bitwise "or" operations on the variables iVar1 and iVar2, and perform bitwise or operations on these two variables, and output the result save to iResult, the specific implementation program is shown in Figure 3.4

| Input 1 | Input 2 | Results |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Table 3-3 "OR" instruction logic relationship table

Procedure:

```
        VAR
    iVar1:INT:=1.
    iVar2:INT:=85.
    iResult:INT.
END_VAR
```

The final running result of the program is 85.

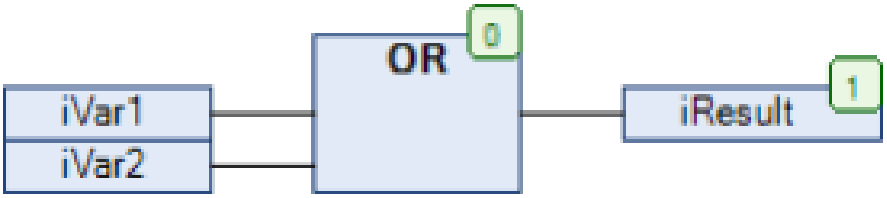



Figure 3.4 Program diagram of by-bit “OR” operation CFC

3. 1. 1. 4 Boolean "OR"

The Boolean "OR" instruction is used to calculate the "or" result of two Boolean expressions. When one of the two Boolean expressions returns true, the result is true; when the result of both Boolean expressions is false, the result is false.

[Example 3.4] Create a POU, use the Boolean "or" operation, determine the return value of the operation, the program is shown in Figure 3.5.

Procedure:

```
VAR
    bResult:BOOL.
    bVar1:BOOL.
    iVar1:INT:=30.
END_VAR
```

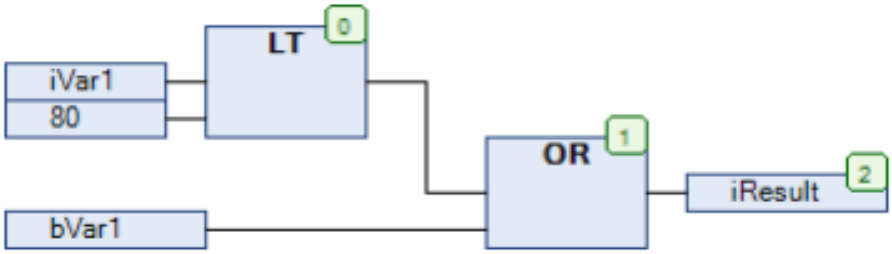



Fig. 3.5 Boolean “OR” operation CFC program diagram

Since the initial value of iVar1 is 30, iVar1<80 is true, and the initial value of bVar1 is "0", so it is false, and the final "or" logic result of one true and one false can be seen according to Table 3-3, which is true. Therefore, the condition on the right side of the equation is true, and the result of the program is that bResult is TRUE.

3. 1. 1. 5 "NOT" by bit

Function: Invert the logical string, change the current value from "0" to "1", or from "1" to "0 ". The by-bit "NOT" instruction is to change the value of the quantities or constants are taken off one by one. The logic is shown in Table 3-4.

[Example 3.5] Create a POU, use the bitwise "non-" operation, determine the return value of the operation, the program is shown in Figure 3.6.

| Input | Results |
|---|---|
| 0 | 1 |
| 1 | 0 |

Table 3-4 "NOT" instruction logic relationship table

Procedure:

```
VAR
byVar1: BYTE:= 1.
byVar2: BYTE.
END_VAR
```

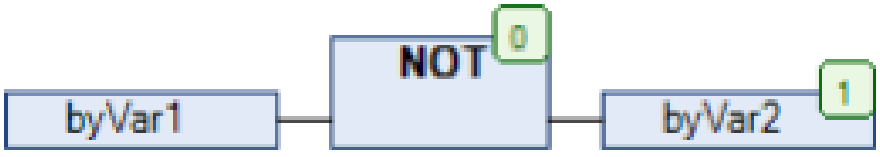


Figure 3.6 Program diagram of “NOT” by bit CFC

Since the value of byVar1 is 1, converting it to binary will result in 00000001, and after bitwise inversion, the result will be 111111110. The final output is 254.

### 3. 1. 1. 6 Boolean "NOT"

The Boolean "NOT" instruction is used to calculate the result of a single Boolean expression. When the input is true, the result is false. When the input is false, the result is true.

[Example 3.6] Create a POU, use the Boolean "NOT" operation, determine the return value of the operation, the specific code is as follows.

Procedure:

```
VAR
bResult: BOOL.
bVar1: BOOL.
iVar1:INT:=30.
END_VAR
```

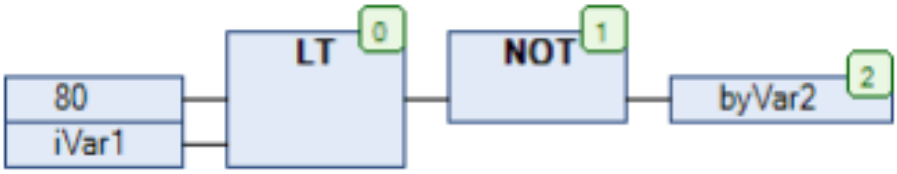



Figure 3.7 Boolean “NOT” operating CFC program diagram

Since the proposition 80<30 is false, the result obtained after using the “NOT” instruction to invert this Boolean expression is true, so finally the result of bResult is True, as shown in Figure 3-7.

### 3. 1. 1. 7 Bitwise "XOR”

Function: The "XOR" instruction compares the corresponding bits of two integers by bit. When the corresponding bit of two integers is a "1" and the other is a "0", the corresponding result bit is "1". When the corresponding bits of two integers are both "1" or both "0", the corresponding result bit is "0".

[Example 3.7] Create a POU to perform bitwise "XOR" operations on variables iVar1 and iVar2, and output the result.

Procedure:

```
VAR
iVar1:INT:=1.
iVar2:INT:=85.
iResult: INT.
END_VAR
```

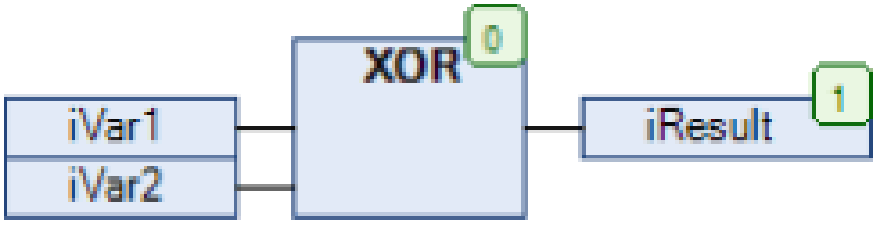



Figure 3.8 Program diagram of CFC by bit “XOR” operation

The 1 in decimal corresponds to the binary number 00000001, and the 85 in decimal corresponds to the binary number 0101.The result is 84 according to the definition of the by-bit "iso-or" operation instruction.

According to bitwise operation. The output is "1" only when the input state of one contact is "1" and the input state of the other contact is "0", if the state of both contacts is If the two contact states are "1" or "0" at the same time, the output will be "0". The timing diagram is shown in 3.9.

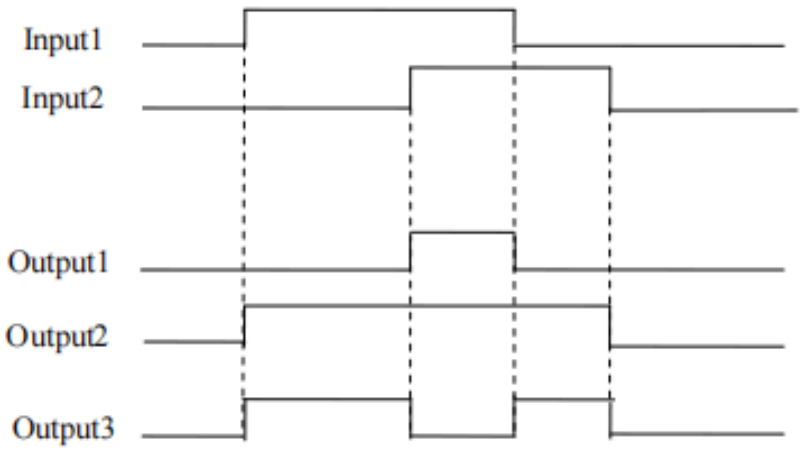



Figure 3.9 Timing diagram of basic logic instructions

[Example 3.8] When decorating a bedroom, you usually choose to install a dual control switch panel. For example, when entering the bedroom press the switch at the door IX0.1 to turn on the lamps, go to bed and do not want to get up, there is also a switch IX0.2 at the head of the bed to control the bedroom lights QX0.1. The switch at the door of the bedroom and the switch at the head of the bed at the same time can independently switch the bedroom lights, please use bit logic instructions to achieve this function.

Procedure:


Figure 3.10 Example of the use of “XOR” instruction

[Example 3.9] A device works with three fans to cool the heat. When the equipment is in operation, the three fans rotate normally, the equipment cooling status indicator is always on; when any two fans rotate, the equipment cooling status indicator flashes at a frequency of 2Hz; when only one fan rotates, the equipment cooling status indicator flashes at a frequency of 0.5Hz; if all three fans do not rotate, the equipment cooling status indicator is not on. I/O address assignment the control program is shown in Figure 3.11.

| Address | Description | Address | Description |
|---|---|---|---|
| %IX0.0 | No.1 fan feedback signal | %IX0.0 | Equipment cooling status indicator |
| %IX0.1 | No.2 fan feedback signal | %MX0.0 | 0.5Hz pulse blink signal |
| %IX0.2 | No.3 fan feedback signal | %MX0.1 | 2Hz pulse blink signal |

The procedure is as follows:

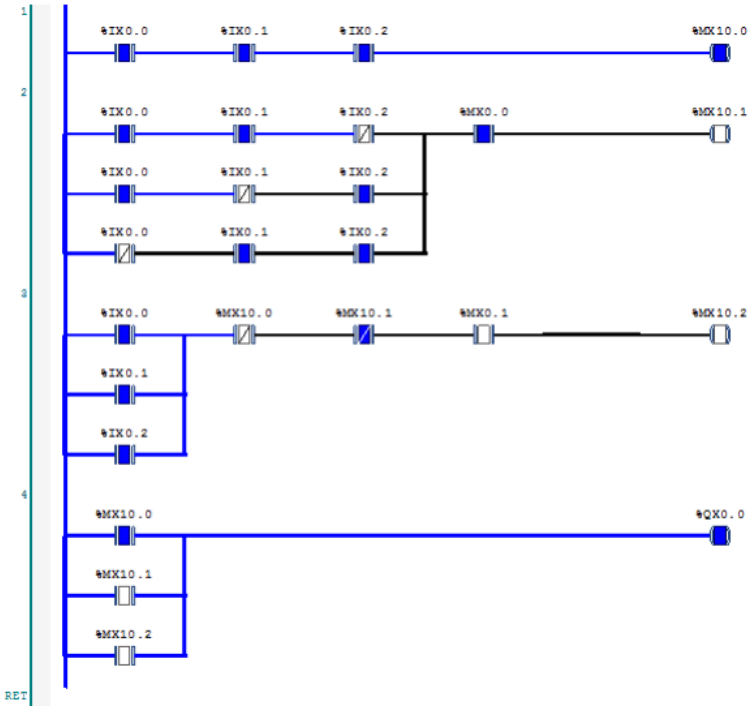



Figure 3.11 Example of fan cooling control program

### 3. 1. 1. 8 Boolean "XOR"

The Boolean "XOR" instruction is used to calculate the result of two Boolean expressions. Only when one of the expressions is true and the other is false, the result returned by the expression is true; when the result of both expressions is true or both are false, the result returned is false.

[Example 3.10] Create a POU and use the Boolean "XOR" instruction to determine whether the return value is TRUE or FALSE.

The specific procedure is shown in Figure 3.12.

VAR
bResult: BOOL.
bVar1: BOOL.
iVar1: INT:=30.
END_VAR

The program runs with a TRUE result.

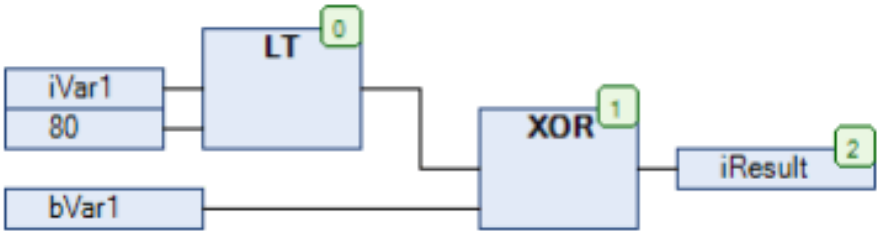



Figure 3.12 Boolean “XOR” operation CFC program diagram

## 3. 1. 2 Placement priority and reset priority flip-flop instructions

In the relay system, several pairs of contacts of a relay act at same time. In the PLC, the instructions are executed one by one, and the instructions are executed in sequence. There are no instructions execution at same time.

Therefore, the set and reset instructions of coil format have priority. the set input and reset input of SR flip -flop and RS flip-flop are in the same instruction, and the set and reset input is executed after whoever is below the instruction input.

The SR flip-flop is a "set priority" flip-flop, when the set signal (SET1) and reset signal (RESET) are 1 at the same time. The flip-flop final state is set state; RS flip-flop is "reset priority" type flip-flop, when the set signal (SET) and reset signal (RESET1) are 1 at the same time, the flip-flop final state is reset state. Set

priority SR and RS reset priority flip-flop instruction parameters are shown in Table 3-6, and the instruction table is shown in 3-7.

| Function Name | FB | ST | Description |
|---|---|---|---|
| SR and RS reset first Priority Triggers |  | SR | Set Priority Trigger |
| |  | RS | Reset Priority Trigger |

Table 3-6 Set Priority SR and RS Reset Priority Trigger Instructions

| Name | Definition | Type data | Description |
|---|---|---|---|
| SET1 | Input Parameters | BOOL | Set priority command |
| SET | Input Parameters | BOOL | Set command |
| RESET1 | Input Parameters | BOOL | Reset priority command |
| RESET | Input Parameters | BOOL | Reset command |
| QT1 | Input Parameters | BOOL | Output |

Table 3-7 Reset Priority SR and RS Reset Trigger Instruction Parameters

3. 1. 2. 1 Set Priority Flip-flop SR

Function: Set bistable flip-flop, set priority. Logic relationship: Q1=(NOTRESETANDQ1)ORSET1 where SET1 is the set signal and RESET is the reset signal.

Syntax: When SET1 is "1", regardless of whether RESET is "1", Q1 output is "1"; when SET1 is "0 ", if Q1 output is "1", once RESET is "1", Q1 output will be reset to "0" immediately. If Q1 output is "0", no matter RESET is "1" or "0", Q1 output will remain as "0 ". The timing diagram is shown in Figure 3.13 (a), and the corresponding state table is shown in Figure 3.13 (b).

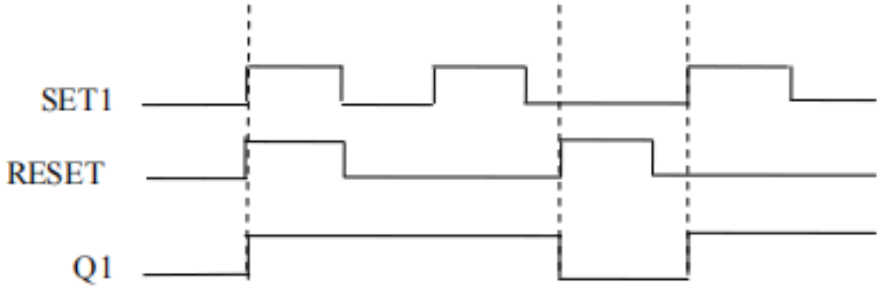



Figure 3.13a) Timing diagram of SR set priority flip-flop

| SET1 | RESET | Q1 Output |
|---|---|---|
| 0 | 0 | Keep the original state |
| 1 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 1 | 1 |

Figure 3.13b) SR Set Priority Flip-flop Status Table

Example 3.11] A system needs a stop signal and requires the system to be shut down immediately after a fault occurs. The output signal to control the equipment shutdown is bStopMachine. Otherwise, it can operate normally.

| Variable Name | Description |
|---|---|
| bRun | System operation |
| bError | System failure |
| bStopMachine | Shutdown command |

Table 3-8 Variable assignment table

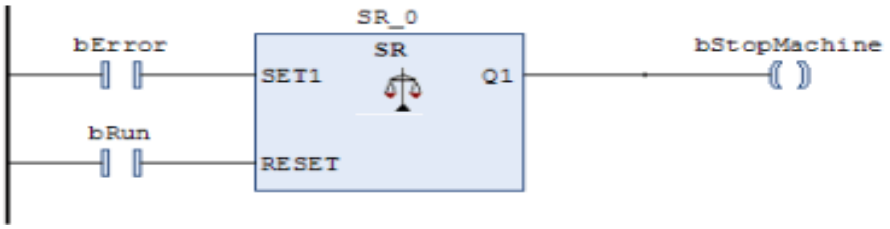



Figure 3.14 Example of SR reset priority flip-flop program

The device's run signal is bRun, and bError will be set to "1" when any fault occurs in the system. The specific variable assignment table is shown in Table 3-8. The program as shown in Figure 3.14. Since bError has a higher priority than bRun, bError needs to correspond to the set priority, and it only makes sense for bRun to be ON when there is no fault.

3. 1. 2. 2 Reset Priority Flip-flop RS

Function: Reset bistable flip-flop, reset priority. Logic relationship: Q1=NOTRESET1AND(Q1ORSET) where SET is the set signal and RESET1 is the reset signal.

Syntax: When RESET1 is "1", the output of Q1 is "0" regardless of whether SET is "1" or not; when RESET1 is "0", if Q1

Once SET is "1", the Q1 output is immediately set to "1". If the Q1 output is "1", the Q output remains "1" regardless of whether SET is "1" or "0". ". The timing diagram is shown in Figure 3.15 (a), and the corresponding state table is shown in Figure 3.15 (b).

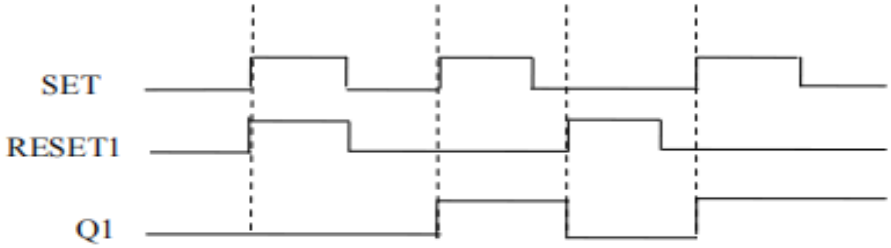



Figure 3.15a) RS Reset Priority Flip-Flop Timing Diagram

| SET1 | RESET1 | Q1 Output |
|---|---|---|
| 0 | 0 | Keep the original state |
| 1 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 1 | 0 |

Figure 3.15b) RS reset priority flip-flop status table

[Example 3.12] To control the motor forward and reverse, the direction of rotation is switched by buttons S1 and S2. When the direction is confirmed, press the HALT button and the motor starts to run, please use the RS reset priority flip-flop to implement this function. The state of the direction selection button corresponding to the program is S1 for the clockwise running direction of the motor and S2 for the counterclockwise direction. The program statement and the implementation procedure are shown below.

```
PROGRAM PLC_PRG
VAR_INPUT
    S1 AT %IX0.1: BOOL;//motor clockwise button
    S2 AT %IX0.2: BOOL;//motor counterclockwise button
    HALT AT %IX0.0: BOOL;//motor Halt button
END_VAR
VAR
    RS_1,RS_2: RS;//reset priority flip-flop
    END_VAR
    VAR_OUTPUT
    K1 AT%QX0.0:BOOL;//execute clockwise
    K2 AT%QX0.1:BOOL;//Run counterclockwise
END_VAR
```

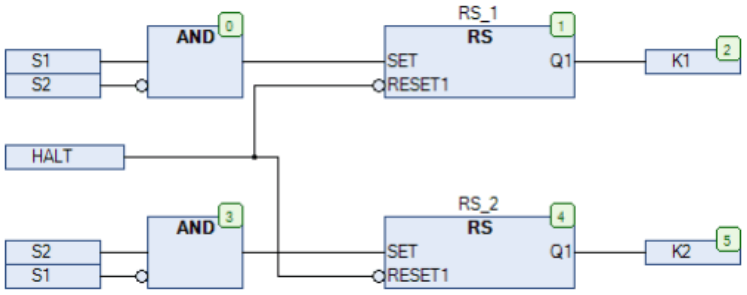



Figure 3.16 RS reset priority for forward and reverse motor control

When the direction S1 is selected, after pressing the HALT button, the K1 contactor is operated, and vice versa, after the S2 button, the HALT button is pressed and the K2 contactor is operated, thus making the motor run clockwise, the program is shown in Figure 3.16.

In practical applications, for the safety of equipment and personnel, it is often necessary to achieve interlocking functions, such as the direction of rotation control buttons S1 and S2 need to achieve the same time only one of the two can be ON, if the two are ON at the same time, it is necessary to stop immediately. To achieve the above function, the corresponding program needs to be modified, and its program variable declaration is consistent with Example 3-12, and the specific implementation procedure is shown in Figure 3.17.

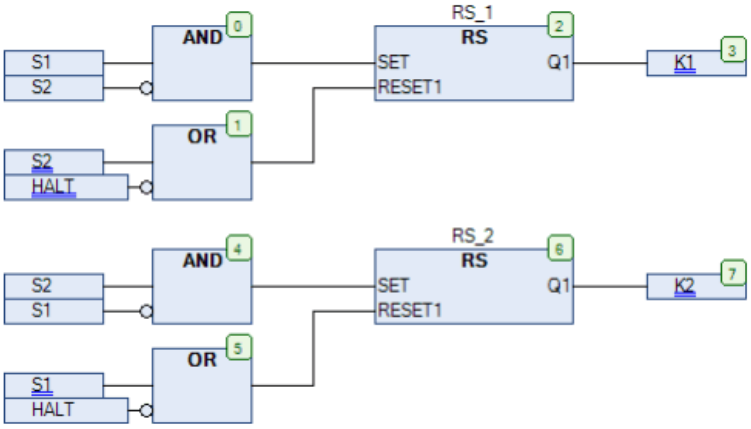
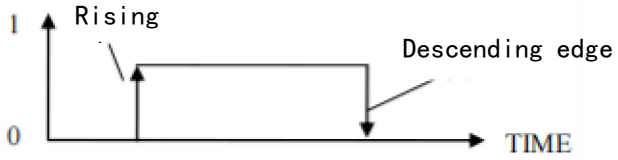


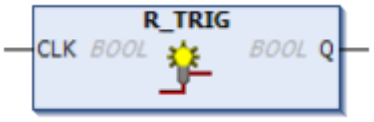

Figure 3.17 RS reset priority motor forward and reverse control with interlock function

The above procedure effectively realizes the interlocking of the program, and once S1 and S2 are ON at the same time, the output will be stopped immediately to protect equipment and operator.

## 3. 1. 3 Edge detection command

The edge detection instruction is used to detect the change in the rising edge (signal from 0 ---->1) and falling edge (signal from 1 ---->0) of the BOOL signal, as shown in Figure 3.18. The signal state is compared with its state in the previous scan cycle in each scan cycle, and if it is different, it indicates a jump edge. Therefore, the signal state in the previous cycle must be stored so that it can be compared with the new signal state. The edge detection command table is detailed in Table 3-9.




| Function Name | Graphical language | Textualized Language | Description |
|---|---|---|---|
| R_TRIG |  | R_TRIG | Rising edge detection |

| | | | |
|---|---|---|---|
| F_TRIG |  | F_TRIG | Falling edge detection |

Table 3-9 Edge detection command

The edge detection command parameters are shown in Figure 3.18 Edge Signal, Table 3-10.

| Name | Definition | Data Type | Description |
|---|---|---|---|
| CLK | Input Variables | BOOL | Detected signal input |
| Q | Output Variables | BOOL | Trigger status output |

Table 3-10 Edge detection command parameters

### 3. 1. 3. 1 Rising edge detection R_TRIG

Function: Used to detect the rising edge.

Syntax: When CLK changes from "0" to "1", the rising edge detector starts to start, Q output first from "1" and then the output becomes "0 ", which lasts for one PLC operation cycle; if CLK is kept as "1" or "0", Q output is kept as "0". Collect the rising edge of the bInput signal, the program as shown in Figure 3.19 (a), the timing diagram as shown in Figure 3.19 (b).

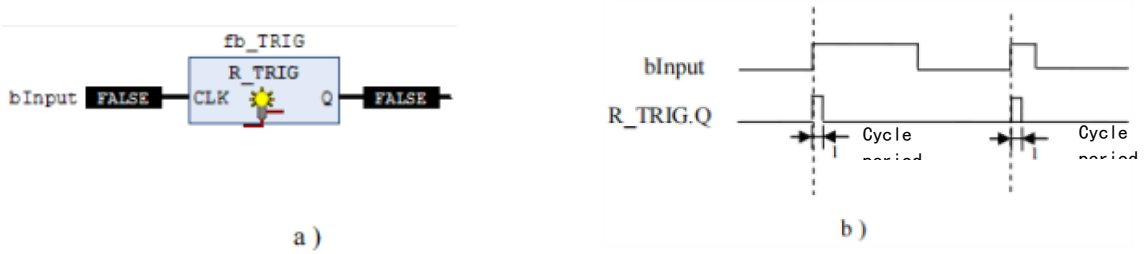



Figure 3.19 Rising edge trigger program a) Rising edge trigger program b) Rising edge trigger timing diagram

[Example 3.13] In industrial projects, the alarm display is often used. The rising edge trigger instruction is used to detect the alarm signal source, and the alarm display is controlled by the set reset function, the variable assignment table is shown in Table 3-11, and the program is shown in Figure 3.20.

| Variable Name | Description | Variable Name | Description |
|---|---|---|---|
| bAlarm1 | Alarm signal source 1 | bAlarm4 | Alarm signal source 4 |
| bAlarm2 | Alarm signal source 2 | bReset | Reset button |
| bAlarm3 | Alarm signal source 3 | bTowerLightRed | Alarm red display light |

The program variables are declared as follows:

```
PROGRAMPLC_PRG
VAR
RS_0:RS.
bAlarm1,bAlarm2,bAlarm3,bAlarm4:BOOL.
R_TRIG_0:R_TRIG.
R_TRIG_1:R_TRIG.
bReset:BOOL.
bTowerLightRed:BOOL.
END_VAR
```

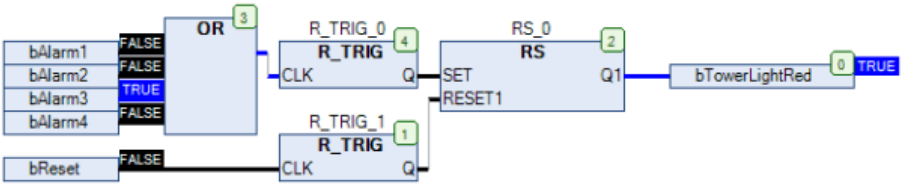



Figure 3.20 Example of alarm display reset CFC program

### 3. 1. 3. 2 Falling edge detection F_TRIG

Function: Used to detect the falling edge.

Syntax: When CLK changes from "1" to "0", the falling edge detector starts, Q output first from "1" and then the output becomes "0 ", which lasts for one PLC operation cycle; if CLK is kept as "1" or "0", Q output is

kept as "0".

[Example 3.14] The falling edge of bInput signal is captured, and when bInput changes from True to False, the function block F_TRIG.Q will give the corresponding output according to the falling edge trigger event, and the output time is maintained in one cycle. The program is shown in Figure 3.21

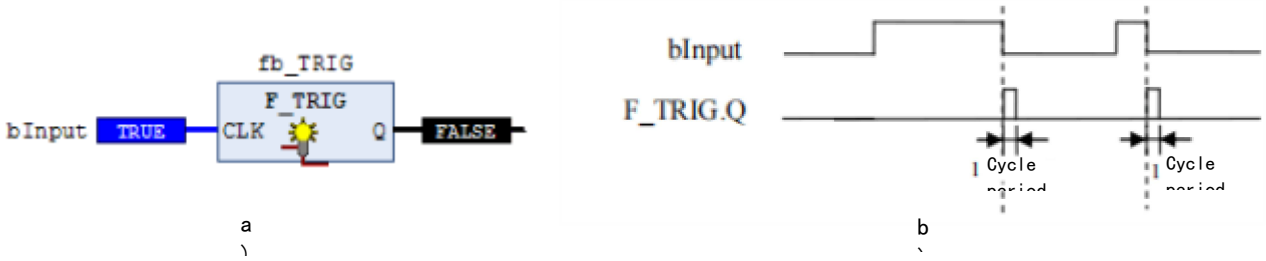



Figure 3.21 Rising edge trigger program

a) Falling edge triggering procedure b) Falling edge triggering timing diagram

# 3. 2 Timer Instructions

## 3. 2. 1 Timer

The timer adopts IEC61131-3 standard timer, which is divided into pulse timer TP, power-on delay timer TON, power-off delay timer TOF and real-time clock RTC. graphical and textual instruction table of all timer instructions is shown in Table 3-12, and timer instruction parameter table is detailed in 3-13.

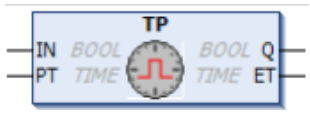
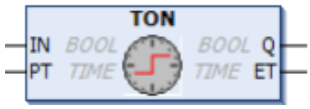
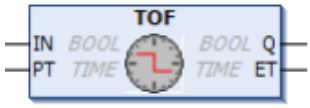
| Timer commands | Graphical language | Textualized Language | Description |
|---|---|---|---|
| Timer |   | TP | Pulse Timer |
| |   | TON | Power-on delay timer |
| **Timer commands** | **Graphical language** | **Textualized Language** | **Description** |
| |   | TOF | Power failure delay timer |
| |   | RTC | Real Time Clock |

Table 3-12 Graphical and text-based instruction table for timer instructions

| Variable Name | Description | Variable Name | Description |
|---|---|---|---|
| IN | Input Variables | BOOL | Start input |
| PT | Input Variables | TIME | Delay time |
| Q | Output Variables | BOOL | Timer output |
| ET | Output Variables | TIME | Current Timing Time |

Table 3-13 Timer instruction parameters

The timer timing diagram is shown in detail in Figure 3.22.

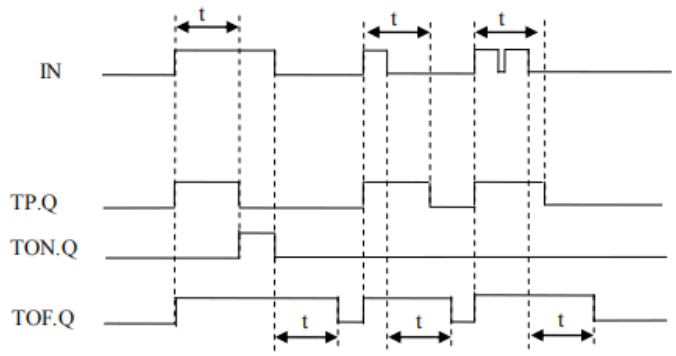


Figure 3.22 Timer characteristics timing diagram

3. 2. 1. 1 Pulse Timer TP

Function: Pulse timing.

Syntax: When the input IN of the timer changes from "0" to "1", the timer is started. of the output signal is "1". The output ET provides the timing time for the output Q. The timing starts from T#0s and ends at the set PT time. When the PT time is reached, ET will keep the timing time until IN becomes "0". If the input IN has become "0" before the PT timing time is reached, enter ET to program T#0s, the PT timing moment. To reset the timer, simply set PT=T#0s.

[Example 3.15] Use the pulse timer TP to make a blinking program with an “ON” keep time of 1s and an “OFF” keep time of five seconds. Two TP timers are used in the program, and the input to control the timer ON to keep 1s is taken as inverse by the output signal of the OFF timer as control. Variable assignment table is shown in Table 3-14, and the program is shown in Figure 3.23

| Variable Name | Description |
|---|---|
| bTowerLightGreen | Indicator light output |

Table 3-14 Variable assignment table

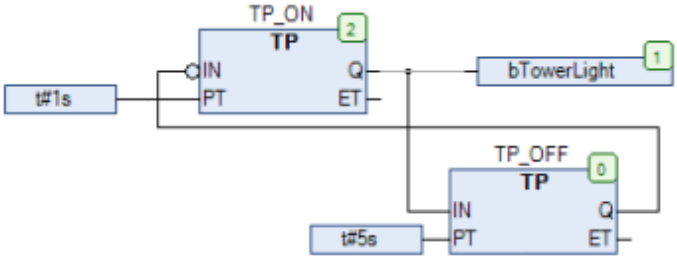


Figure 3.23TP timer program

3. 2. 1. 2 Power-on delay timer TON

Function: Power-on delay timer. When the input IN of the timer changes from "0" to "1", the timer will start, and when the timing time PT is reached and the signal IN of the input is always maintained at "1", the output signal is “1”. If the input IN signal changes from "1" to "0" before the timer reaches the timer time, the timer will be reset, and the next timer is restarted on the rising edge of IN signal.

The output ET provides a timing time that starts at T#0s and ends at the set PT time, when the PT is reached, ET will hold the timing time until IN becomes "0". If the input IN becomes "0" before the PT timing time is reached, the output ET immediately becomes T#0s. To restart the timer, you can set PT=T#0s or set IN=FALSE.

[Example 3.16] Two motors M1 and M2 are required to start M1 when the start button DI_bStart is pressed, and M2 starts 20 seconds later; when it is necessary to stop, press the stop button DI_bStop, and M2 stops, and M1 stops 10 seconds later. Each motor has overload protection, and when either motor is overloaded, both motors stop at the same time. Variable assignment table is shown in Table 3-15.

When the start button DI_bStart is pressed, it immediately sets DO_KM_M1 to "1" and triggers the M2 motor start delay through the self-locking signal, and after reaching 20s, the timer M2_StartDelay. Q is set to "1", and M2 is started by this signal.

Motor. When it needs to stop, press the stop button DI_bStop, set the intermediate variable bStopTemp to "1", and stop the M2 motor immediately, and start the M1 motor stop delay timer M1_StopDelay through this intermediate variable, and M1 will stop after reaching the set time of 10s. The program is shown in Figure 3.23

| | |
|---|---|
| DI_bStop | Stop Button |
| DI_bFuse1 | M1 overload protection |
| DI_bFuse2 | M2 overload protection |
| DO_KM_M1 | Start M1 |
| DO_KM_M2 | Start M2 |

Table 3-15 Variable assignment table

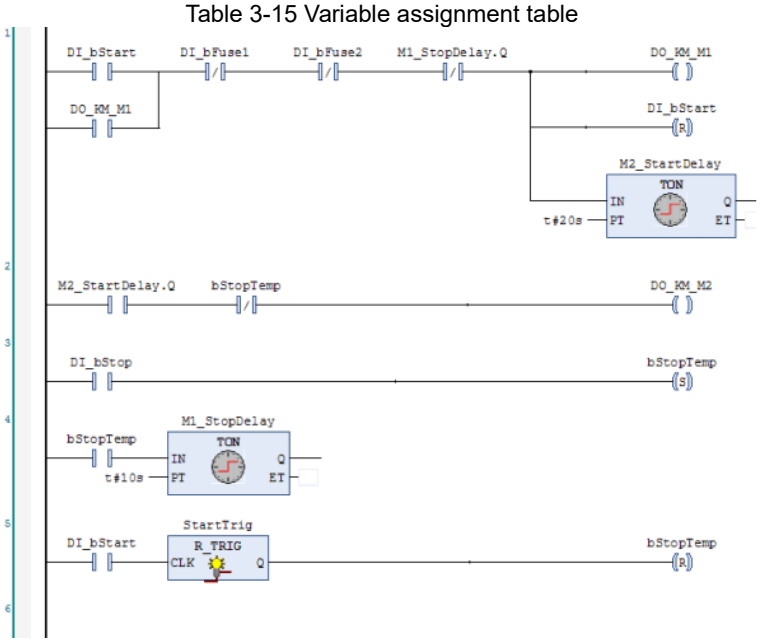



Figure 3.23 Example of motor delay start procedure

### 3. 2. 1. 3 Power-off delay timer TOF

Function: Power-off delay timing. When the input IN of the timer changes from "0" to "1", the Q output signal of the timer is "1", and the start input of the timer changes to When the timer input becomes "0", the timer starts. If the timer is running, its output Q is always "1", and when the timer time is reached, the output Q is reset, and before the timer time is reached, if the timer input returns to "1", then the timer is reset and the output Q output signal at the output is kept as "1". You can refer to the TOF related timing diagram in Figure 6.x.

The output ET provides the timing time, the delay time starts from T#0s and ends at the set timing time PT. When the PT time is reached, ET will hold the timing time until input IN returns "1". If the input IN changes to "1" before the PT timing time is reached, the output ET immediately changes to T#0s. To reset the timer, you can set PT=T#0s.

[Example 3.17] In the interior light control, when the door is opened, the interior light will be on, even if the door is closed within 10 seconds, the interior light will continue to be on, such control is used in the timer break delay action mode. The following PLC program to achieve this function.

The timing diagram of the program is shown in Figure 3.24. The bDI_Door door lock signal is "1" when the door is open and "0" when it is closed. The program is shown in Figure 3.25, and the variable assignment table is shown in Table 3-16.

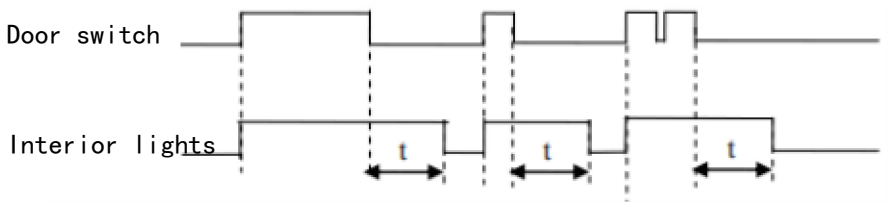



Figure 3.24 Timing diagram of the car interior light control program

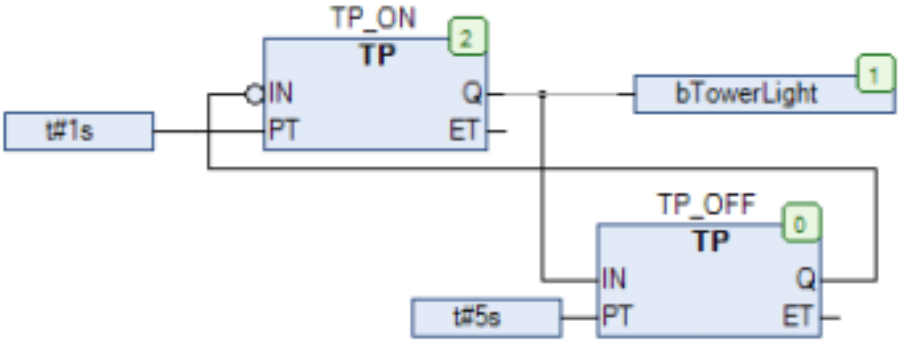



Figure 3.25 TOF Car Door Timer Program

| Variable Name | Description |
|---|---|
| bDI_Door | Door lock signal |
| bDO_IndoorLight | Interior lights |

Table 3-16 Variable Assignment Table

### 3. 2. 1. 4 Real Time Clock RTC

Function: Starts at the given time and returns the current date and time.

Syntax: RTC (EN, PDT, Q, CDT) means when EN is "0", the output variable Q and CDT is "0", related time is DT#1970-01-01-00:00:00. Once EN is "1", the time given by PDT will be set and will be counted in seconds, once EN is TRUE, CDT will be returned. Once EN is reset to FALSE, CDT will be reset to its initial value of DT#1970-01-01-00:00:00. Please note that the PDT time is only valid on the rising edge. the RTC timer parameter table is shown in detail in 3-17.

| Variable Name | Description | Variable Name | Description |
|---|---|---|---|
| EN | Input Variables | BOOL | Start Enable |
| PDT | Input Variables | DATA_AND_TIME | Set the time and date when it will be started |
| Q | Output Variables | BOOL | Status Output |
| CDT | Output Variables | DATA_AND_TIME | Status of current count time and date |

Table 3-17 Standard Timer Instruction Parameters

[Example 3.18] Create a POU, use the RTC instruction, set the initial time for it, and when the bEnable variable is ON, returns the current date and time. The procedure is shown in Figure 3.26.

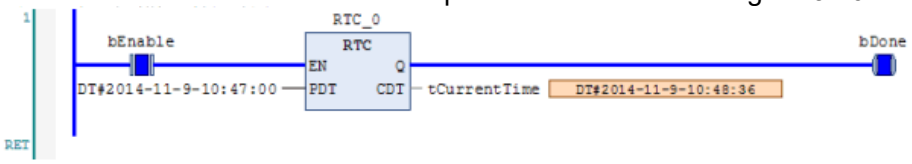



Figure 3.26 Example of RTC instruction application

# 3. 3 Counter Instructions

## 3. 3. 1 Introduction to Counters

The CodeSys standard function library provides add and subtract counting function blocks, and the system provides three function blocks, CTU add counter, CTD subtract counter and CTUD add and subtract counter.

## 3. 3. 2 Counter Instructions

The graphical and textual instruction table of all counter instructions that come with the standard library is shown in Table 3-18, and the counter instruction parameter table is detailed in 3-19.

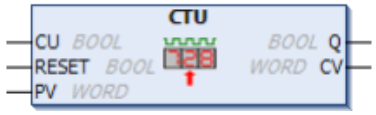
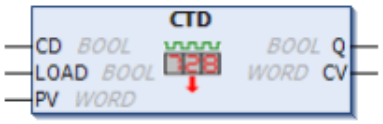
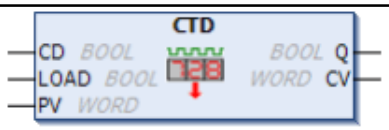
| Counter command | Graphical language | Textualized Language | Description |
|---|---|---|---|
| Counters |  | CTU | Incremental counter |
| |  | CTD | Decremental Counter |
| |  | CTUD | Incremental / Decremental counter |

Table 3-18 Graphical and text-based instruction table for standard counters

| Variable Name | Description | Variable Name | Description |
|---|---|---|---|
| CU | Input Variables | BOOL | Detection of rising edge signal input triggers output CV increment |
| CD | Input Variables | BOOL | Detection of rising edge signal input trigger output CV decrement |
| RESET | Input Variables | BOOL | Reset counter |
| LOAD | Input Variables | BOOL | Load counter |
| Q | Output Variables | BOOL | Output TRUE when CV is incremented to the count limit/0 |
| QU | Output Variables | BOOL | When CV increments to the upper count limit PV, QU outputs TRUE |
| QD | Output Variables | BOOL | When the output CV decreases to 0, the QD outputs TRUE |
| CV | Output Variables | WORD | Current count value |

Table 3-19 Standard Counter Instruction Parameters

### 3. 3. 2. 1 Incremental Counter CTU

When the signal at the counter input CU changes from state "0" to state "1", the current calculated value is added by 1, and the current value is calculated via the output

CV is displayed, and when it is called for the first time (the reset input RESET signal status is "0"), the count at the input PV is the default value, and when the count reaches the upper limit of 32767, the counter will not be increased, and the CU will not work anymore.

When the signal state of reset input RESET is "1", the CV and Q of the counter are "0", and if the state of input RESET is "1", the rising edge will no longer work for the CU will no longer work. When the CV value is greater than or equal to PV, the output Q is "1". At this time, the CV can continue to accumulate, and the output Q continues to be output "1".

An example of the increment counter CTU instruction is shown in Figure 3.27, and the timing diagram is shown in 3.28.

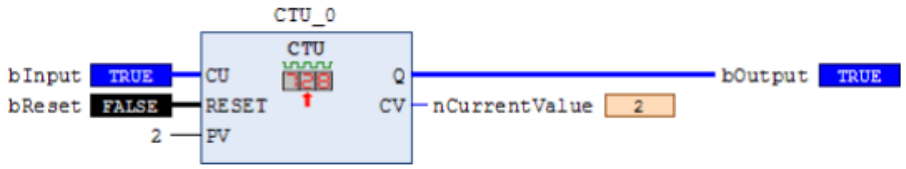



Figure 3.27 Example of using the incremental counter CTU

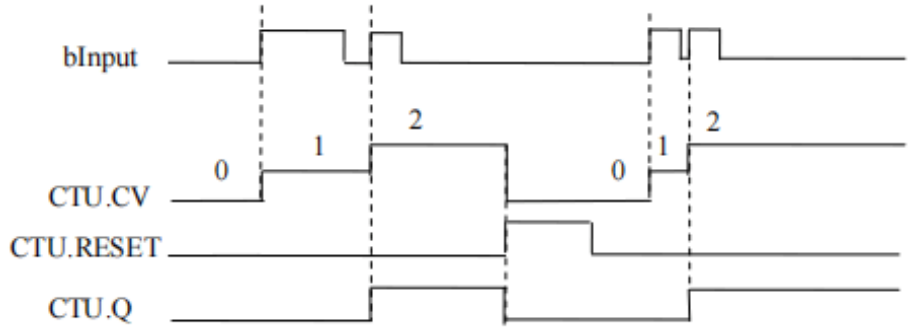



Figure 3.28 Timing diagram of incremental counter CTU

Incremental function blocks. The input variable CU and reset RESET and the output variable Q are of Boolean type, and the input variable PV and the output variable CV are of WORD type.

CV will be initialized to 0 if reset RESET is TRU really. If the CU has a rising edge from FALSE to TRUE, the CV is raised by 1 and Q will return TRUE so that the CV will be greater than or equal to the upper limit PV.

There is a photoelectric sensor on the line, and the signal will be set to "1" when the product passes by, as shown in Figure 3.29 (a). With the counter instruction, the program is used to calculate the total output throughput.

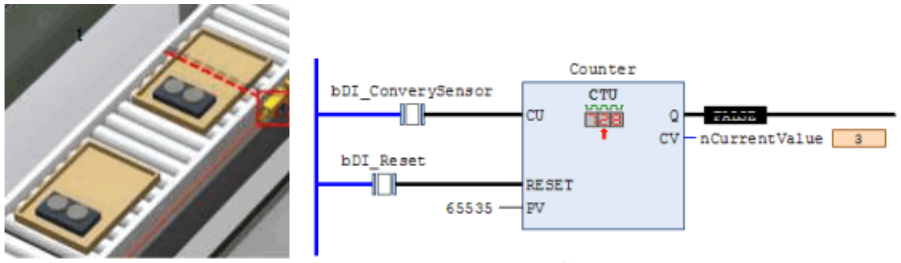



Figure 3.29 Example of using the incremental counter CTU
a) Application schematic b) Program ladder diagram

See Table 3-20 for the variable assignment table. bDI_ConverySensor's sensing status corresponds to "1" when there are products flowing through the pipeline and "0" when there are no products, so this signal can be directly used as the CU input signal of the increment counter CTU. The current real-time value is displayed by nCurrentValue. bDI_Reset is used as a clear signal to clear the current value.

| Variable Name | Description |
|---|---|
| bDI_ConverySensor | Transmission with sensor signal |
| bDI_Reset | Counter reset button |
| nCurrentValue | Total number of current products |

Table 3-20 Variable assignment table

### 3. 3. 2. 2 Decremental Counter CTD

When the CD signal at the input of the counter is changed from "0" to "1", the current count value is reduced by 1 and the current value is displayed on the output side of the CV, the first time it is called (the signal LOAD at the load input needs to be initialized, it needs to be changed from The first time it is called (it is necessary to initialize the LOAD signal at the input, which needs to be changed from "0" to state "1" and then to state "0" before the function block can take effect), the count at the input PV is the default value, and when the count reaches 0, the count value will not be When the count reaches 0, the count value will not decrease and CD will no longer work.

When the load input signal LOAD is "1", the count value is set to the PV default value, and the CD rising edge of the input does not work if the load input signal LOAD is "1". When the CV value is less than or equal to 0, the output Q is "1". The example of subtracting counter CTD instruction is shown in Figure 3.30, and the timing diagram is shown in 3.31.

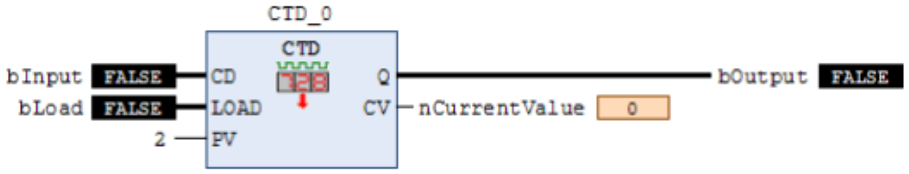



Figure 3.30 Example of the use of the decremental counter CTD

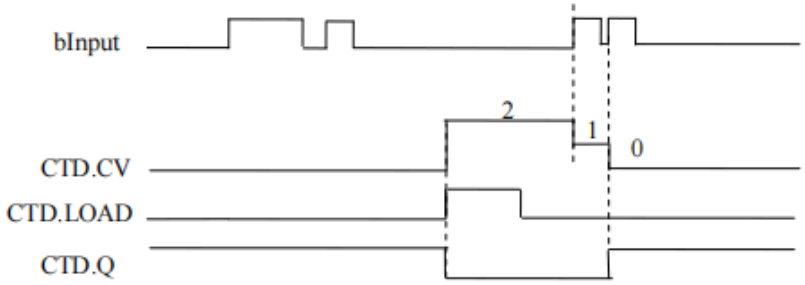



Figure 3.31 Timing diagram of decremental counter CTD

[Example 3.19] A factory produces products every 25 can be filled with a box, after flowing through the assembly line, each full box, you need to output a 3s delay instructions bPackingDone signal, the assembly line is equipped with photoelectric sensors, through the bDI_ConverySensor signal feedback to the PLC. in addition to the need to count the total number of boxes produced that day nPackageQTY.

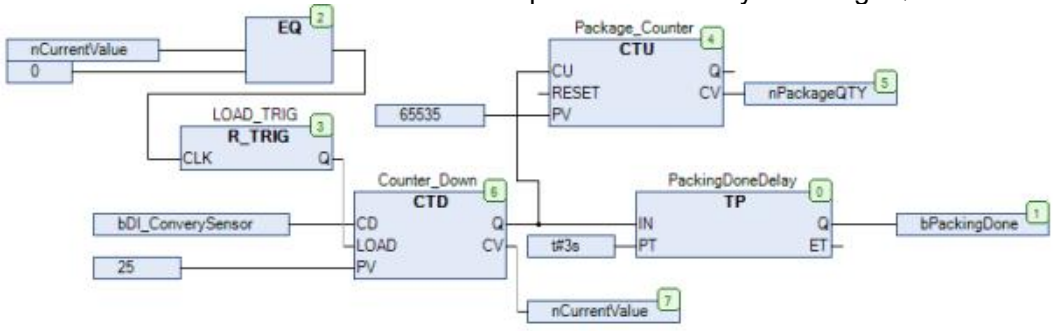



Figure 3.32 Example of the decremental counter CTU program

The variable assignment table is shown in Table 3-21, using the bDI_ConverySensor sensor signal on the transmission line as the CD source signal for the CTD function block, PV is given as 25, and Counter_Down.Q outputs a high-level pulse as the BOOL signal for a full case when it is full of 25 cases. This signal is also used as the CU source signal of the Package_Counter counter to accumulate the number of cases. nPackageQTY is the total number of cases produced for the day. The full box indicator is output by using TP function block to do delay. The program is shown in Figure 3.32.

| Variable Name | Description |
|---|---|
| bDI_ConverySensor | Transmission with sensor signal |
| bPackingDone | Full tank indicator |
| nPackageQTY | Total number of cases produced on that day |

Table 3-21 Variable assignment table

### 3. 3. 2. 3 Incremental / decremental bidirectional counter CTUD

When the CU signal at the plus counter input changes from "0" to "1", the current count value is added by 1 and is displayed on the output CV line. When the state of the CD signal at the minus input changes from "0" to "1", the current count value is subtracted by 1 and displayed on the output CV. If both inputs have rising edges, the current count value will remain unchanged.

When the count value reaches the upper limit value of 32767, the rising edge of the plus count input CU no longer works. Therefore, even if the rising edge of the plus counter input CU appears, the count value does not increase. Similarly, when the count value reaches the lower limit value 0, the decrement input CD will not be in its role, so the count value will not decrease even if the rising edge of the decrement input CD appears.

When the CV value is greater than or equal to the PV value, the output QU is "1". When the CV value is less than or equal to 0, the output QD is "1".

[Example 3.20] Create a POU, use the increment/decrement bidirectional counter CTUD, when bUp has a rising edge signal, the count value increases, bDown has a rising edge signal, the count value decreases. bReset is used for data reset, the specific code is as follows, the program is shown in Figure 3.33.

```
VAR
bUp:BOOL.
bDown:BOOL.
bReset:BOOL.
bLoad:BOOL.
CTUD_0:CTUD.
END_VAR
CTUD_0(CU:=bUp,CD:=bDown,RESET:=bReset,LOAD:=bLoad,PV:=,QU=>,QD=>,CV=>).
```

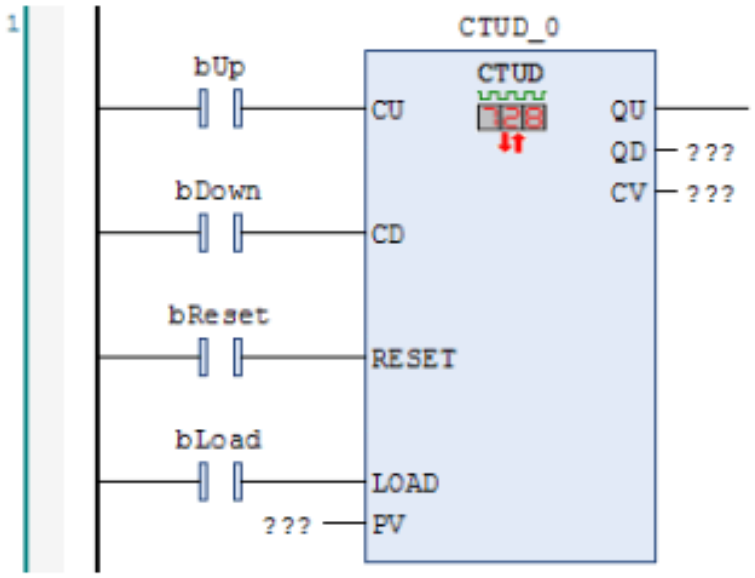



Figure 3.33 Example of using the CTUD increment/decrement counter

[Example 3.21] An automatic warehouse stores a certain kind of goods, up to 6000 boxes, and needs to count the incoming and outgoing goods stored, with more than 1000 boxes of goods, the light L1 lights up; the excess of 5000 boxes of goods, the light L2 lights up. The input signal for incoming goods is bInput, and the corresponding input signal for outgoing goods is bOutput. see the variable assignment table in Table 3-22 for details.

| Variable Name | Description |
|---|---|
| bInput | Stocking |
| bOutput | Shipment |
| nL1Value | Number 1000 |
| nL2Value | Number 5000 |
| L1 | More than 1000 lights |
| L2 | More than 5000 lights |

- Using the CTUD command, when receiving

bInput is connected to the CU signal of the bidirectional counter for value accumulation, and bOutput is connected to the CD signal of the bidirectional counter for value decrement when shipping. The current value is represented by FB_CTUD.CV. If the current count value is greater than nL1Value, L1 is ON, and greater than nL2Value, L2 is ON.

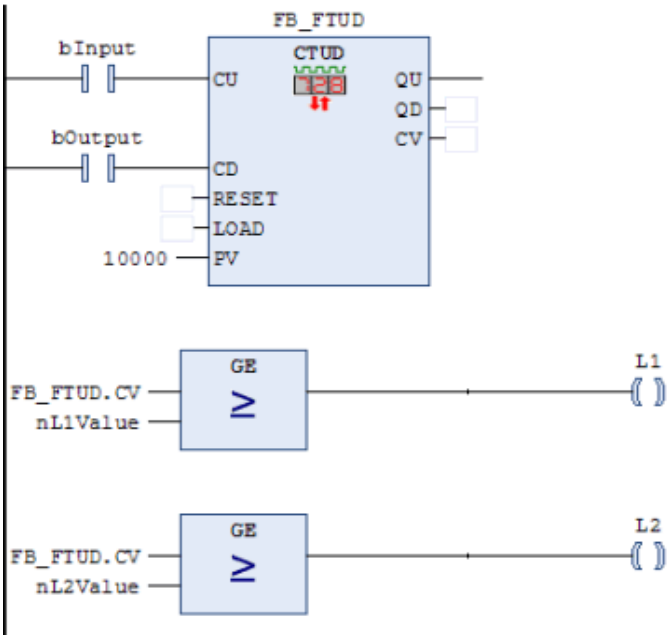



Figure 3.34 Example of automatic warehouse incoming and outgoing counting function

# 3. 4 Data Processing Instructions

The data processing instructions provided in the CodeSys standard function library include selection operation instructions, comparison instructions and shift instructions, etc. The instructions in each section are described in detail below.

## 3. 4. 1 Selecting operation commands

In the actual application often used in some data selection and screening, in the selection operation instructions will introduce several common instructions for the reader in the future practical applications to provide convenience, common selection operation instructions as shown in Table 3-23.

| Select operation command | Graphical language | Textualized Language | Description |
|---|---|---|---|
| Select operation command | SEL<br>??? G<br>??? IN0<br>??? IN1 | SEL | Two-choice-one |
| | MAX<br>???<br>??? | MAX | Take the maximum value |
| | MIN<br>???<br>??? | MIN | Take the minimum value |
| | LIMIT<br>??? MN<br>??? IN<br>??? MX | LIMIT | Limited value |

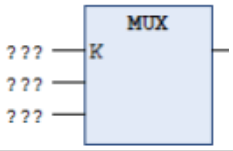
| | | | |
|---|---|---|---|
| |  | MUX | Multiple Choice One |

Table 3-23

3. 4. 1. 1 The two-choice SEL

Function: Select one of the two input data as output by the selector switch, and when the selector switch is FALSE, the output is the first input data, when the selector switch is TRUE, the output is the second data.

Syntax: Its textualized language syntax format is as follows, OUT:=SEL(G,IN0,IN1) The parameter G must be a Boolean variable. If G is FALSE, the result of the return value is IN0, if G is TRUE, the result of the return value is IN1, and its parameter description is detailed in the table 3-24 shown.

| Name | Definition | Data Type | Description |
|---|---|---|---|
| G | Input Variables | BOOL | Input selection bit |
| IN0 | Input Variables | Any type | Input data 0 |
| IN1 | Input Variables | Any type | Input data 1 |
| Return Value | Output Variables | Any type | Output Data |

Table 3-24 Two-choice SEL parameter description

[Example 3.22] Create a POU, when the input value bInput is FALSE, the output is 3, and vice versa, when it is TRUE

The output is 4. The specific implementation procedure is as follows.

```
VAR
iVar1:INT:=3.
iVar2:INT:=4.
iOutVar:INT.
bInput:BOOL.
END_VAR
iOutVar:=SEL(bInput,iVar1,iVar2).
```

3. 4. 1. 2 Taking the maximum value of MAX

Function: Maximum value function. Selects the maximum value among multiple input data as the output.

Syntax: The format of its textualized language syntax is shown below.

OUT:=MAX(IN0,...,INn)

The parameter descriptions are detailed in Table 3-25.

| Name | Definition | Data Type | Description |
|---|---|---|---|
| IN0 | Input Variables | Any type | Input data 0 |
| INn | Input Variables | Any type | Input data n |
| Return Value | Output Variables | Any type | Output Data |

Table 3-25 Explanation of MAX parameters for taking the maximum value

[Example 3.23] Create a POU where the input value of iOutVar is the greater of iVar1 and iVar2, and implement the program as follows.

```
VAR
iVar1:INT:=30.
iVar2:INT:=60.
iOutVar:INT.
END_VAR
iOutVar:=MAX(iVar1,iVar2).
```

The program runs and the output is 60.

3. 4. 1. 3 Taking the minimum value MIN

Function: Minimum value function. Selects the minimum value as the output among multiple input data.

Syntax: The format of its textualized language syntax is shown below.

OUT:=MIN(IN0,...,INn)

IN0,INn and OUT can be of any data type, and their parameter descriptions are shown in Table 3-26.

| Name | Definition | Data Type | Description |
|---|---|---|---|
| IN0 | Input Variables | Any type | Input data 0 |
| INn | Input Variables | Any type | Input data n |
| Return Value | Output Variables | Any type | Output Data |

Table 3-26 Minimum value MIN parameter description

[Example 3.24] Create a POU where the input value of iOutVar is the smaller of iVar1 and iVar2, and implement the program as follows.

```
VAR
iVar1:INT:=30.
iVar2:INT:=60.
iOutVar:INT.
END_VAR
iOutVar:=MIN(iVar1,iVar2).
```

The program runs and the output is 30.

### 3. 4. 1. 4 Limiting value LIMIT

Function: Limit value output. Determines whether the input data is between the minimum and maximum values, and if the input data is between them, the input data is output as the output data. If the input data is greater than the maximum value, the maximum value is used as the output value. If the input data is less than the minimum value, the minimum value is used as the output value.

syntax: its textualized language syntax format is as follows.

OUT:=LIMIT(Min,IN,Max)

IN,Min,Max and the return value can be of any data type, and their parameter descriptions are detailed in Table 3-27.

| Name | Definition | Data Type | Description |
|---|---|---|---|
| Min | Input Variables | Any type | Input data 0 |
| IN | Input Variables | Any type | Input data n |
| Max | Input Variables | Any type | Input data n |
| Return Value | Output Variables | Any type | Output Data |

Table 3-27 Parameter Description

Example 5.25] Create a POU and use the limit value instruction to ensure that the output value is in the range of 30 to 80, regardless of the input value. The specific implementation procedure is as follows.

```
VAR
iVar:INT:=90.
iOutVar:INT.
END_VAR
iOutVar:=limit(30,iVar,80).
```

The minimum input value is 30 and the maximum input value is 80. The actual input value is 90, which is greater than the maximum value, so the final output takes the maximum value of 80 as the output, so the result is 80.

### 3. 4. 1. 5 Multiple Choice MUX

Function: Multiplexer operation. Selects one of multiple input data as output by control number.

syntax: its textualized language syntax format is as follows.

OUT:=MUX(K,IN0,... ,INn)

IN0,... ,INn and the return value can be any variable type. However, K must be BYTE, WORD, DWORD, LWORD, SINT, USINT, INT, UINT, DINT, LINT, ULINT, or UDINT, and MUX selects the Kth data output from the variable group. The parameter descriptions are shown in Table 3-28.

| Name | Definition | Data Type | Description |
|---|---|---|---|
| K | Input Variables | Integer type | Control number |
| IN0 | Input Variables | Any type | Input data 0 |

| | | | |
|---|---|---|---|
| INn | Input Variables | Any type | Input data n |
| Return Value | Output Variables | Any type | Output Data |

Table 3-28 Parameter Description

[Example 3.26] Create a POU and use the multi-select instruction to select the final data to be output according to the input control number iVar.

The specific implementation procedure is as follows.

VAR
iVar:INT:=1.
iOutVar:INT.
END_VAR
iOutVar:=MUX(iVar,30,40,50,60,70,80).

The final output result is 40, because the data sorting is accumulated from the 0th element. If the data is out of range, the most data is output by the last data, such as in example 6.x, the value of iVar is set to 10, the final output is 80. If iVar is -1, the final output is still 80.

## 3. 4. 2 Comparison Instructions

The compare instruction is used to compare two signed or unsigned numbers of the same data type, IN1 and IN2, to determine the operation. The operations involved are: =, > =, < =, >, <, < > In graphical languages, the comparison instruction is programmed as a moving contact, with the comparison parameter and the comparison operator in the middle of the moving contact. When the result of the comparison is true, the moving contact closed; in the text-based language, the comparison instruction can be directly expressed in symbols, when the comparison result is true, the PLC will set the result of the operation to True. comparison instructions graphical and text-based instructions are shown in Table 3-29.

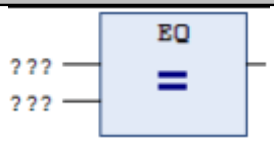
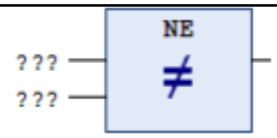
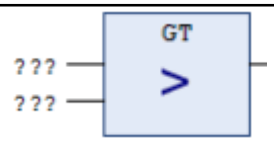
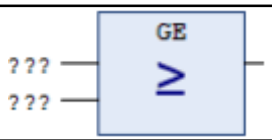
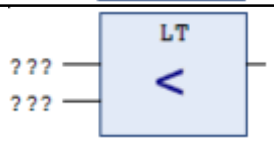
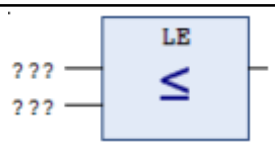
| Compare commands | Graphical language | Textualized Language | Description |
|---|---|---|---|
| Compare commands |  | = | Equivalent |
| |  | <> | No equivalent |
| |  | > | Greater |
| |  | >= | Greater or equal |
| |  | < | Less |
| |  | <= | Less or equal |

Table 3-29 Table of graphical and textual instructions for comparison instructions

### 3. 4. 2. 1 Greater

Function: the Boolean operator returns TRUE when the first operand is greater than the second operand. The variable types are BOOL, BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL, LREAL, TIME, DATE, TIME_OF_DAY, DATE_ AND_TIME and STRING.

Syntax: Its textualized language syntax format is as bResult:=bVar1>bVar2.

[Example 3.27] As the INT type variable his value valid range is -32768 to 32767, the type exists positive and negative, in the actual application often need to determine its sign. If the value is negative, the highest bit of the variable is 1. For example, 32766 can be used as the defining line, once the current value is greater than 32766, both the current value can be judged as negative, to carry out the next action. The declaration part of this judgment procedure is shown below, and the specific procedure is shown in Figure 3.35.

```
PROGRAMPLC_PRG
VAR_INPUT
nValue:INT;//input value
END_VAR
VAR_OUTPUT
bOverFlowAT%QX0.0:BOOL;//Sign bit overflow
END_VAR
```

When the value of nValue is greater than or equal to 32766, bOverFlow will be set to TRUE.

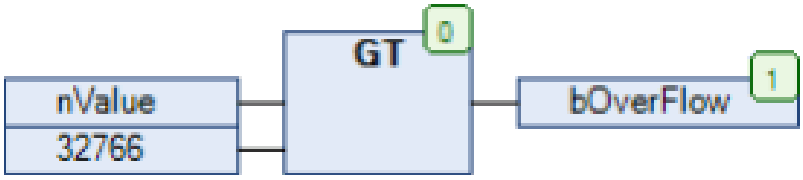



Figure 3.35 Example of a CFC program application larger than the function

### 3. 4. 2. 2 Greater or equal

Function: when the first operand is greater than or equal to the second operand, the Boolean operator returns TRUE.

The variable types are BOOL, BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL, LREAL, TIME, DATE, TIME_OF_DAY, DATE_AND_TIME and STRING.

Syntax: Its textualized language syntax format is bResult:=bVar1>=bVar2.

[Example 3.28] Since the UINT type variable he valid range of values is 0 to 65535, if the value is greater than this range, there will be an overflow. To prevent this phenomenon, an overflow warning can be made in the program, when the value is greater than or equal to 65530, that is, a warning message is given to remind to clear the data to zero. The declaration part of this judgment program is shown below, and the specific program is shown in Figure 3.36.

```
PROGRAMPLC_PRG
VAR_INPUT
nValue:UINT;//input value
END_VAR
VAR_OUTPUT
bOverFlowAT%QX0.0:BOOL;//overflow warning
END_VAR
```

When the value of nValue is greater than or equal to 32766, bOverFlow will be set to TRUE.

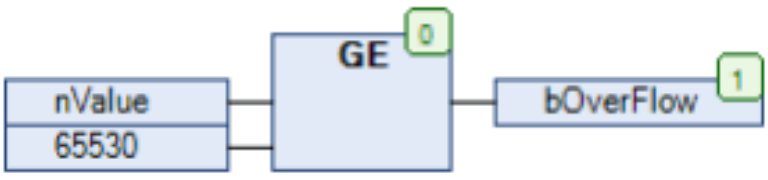



Figure 3.36 Example of a CFC program application with a greater than or equal to function

3. 4. 2. 3 Equivalent

Function: the Boolean operator returns TRUE when the first operand is equal to the second operand, and the operand types are BOOL, BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL, LREAL, TIME, DATE, TIME_OF_DAY, DATE_ AND_TIME and STRING.

Syntax: The format of its textualized language syntax is bResult:=bVar1=bVar2.

[Example 3.29] Judgment of S1 button and S2 button, if the two buttons are in the same state, then K1 output is ON, and vice versa is FALSE. the declaration part of this judgment program is shown below, and the specific program is shown in Figure 3.37.

```
PROGRAMPLC_PRG
VAR_INPUT
S1AT%IX0.0:BOOL;//Input button 1
S2AT%IX0.1:BOOL;//Input button 2
END_VAR
VAR_OUTPUT
K1AT%QX0.0:BOOL;//output indicator
END_VAR
```

Buttons S1 and S2 are two input buttons, and when both are ON or OFF at the same time, K1 has an output ON signal.

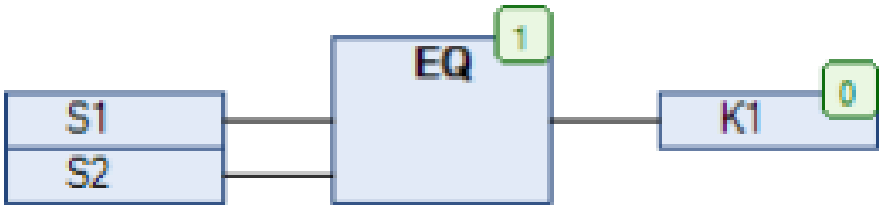



Figure 3.37 Example of a CFC program application equal to a function

3. 4. 2. 4 Less

Function: when the first operand is smaller than the second operand, the Boolean operator returns TRUE. The operand types are BOOL, BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL, LREAL, TIME, DATE, TIME_OF_DAY, DATE_ AND_TIME and STRING.

syntax: its textualized language syntax format is bResult:=bVar1<bVar2.

[Example 3.30] If K1 has an output when the input value nVar1 is smaller than the input value nVar2, then K1 has an output. The declaration part of this judgment program is shown below, and the specific program is shown in Figure 3.38.

```
PROGRAMPLC_PRG
VAR_INPUT
nVar1:WORD;//Input value 1
nVar2:WORD;//Input value 2
END_VAR
VAR_OUTPUT
K1AT%QX0.0:BOOL;//output indication
END_VAR
```

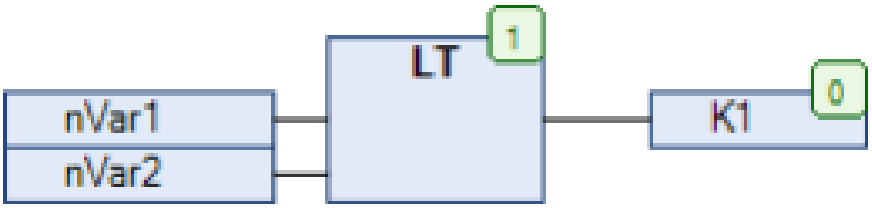



Figure 3.38 Less than function CFC program application example

3. 4. 2. 5 Less or equal

Function: the Boolean operator returns TRUE when the first operand is less than or equal to the second operand. The operand types are BOOL, BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL, LREAL, TIME, DATE, TIME_OF_DAY, and STRING. DATE_AND_TIME and STRING.

Syntax: Its textualized language syntax format is as follows, bResult:=bVar1<=bVar2.

【Example 3.31】byVar1 is a WORD type variable, such as when its actual value exceeds 255, the program outputs a warning K1, the declaration part of this judgment program is shown below, the specific program is shown in Figure 3.39.

```
PROGRAMPLC_PRG
VAR_INPUT
byVar1:WORD;//Input value 1
END_VAR
VAR_OUTPUT
K1AT%QX0.0:BOOL;//output indication
END_VAR
```

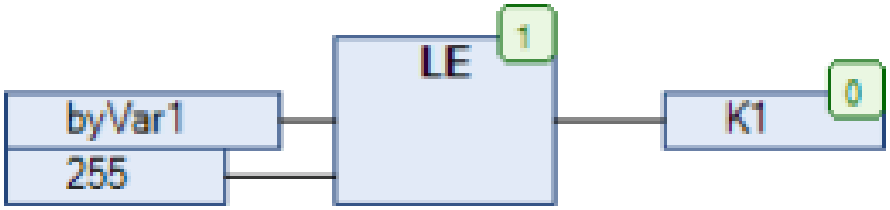



Figure 3.39 Less than or equal to function CFC program application examples

3. 4. 2. 6 Not equal

Function: the Boolean operator returns TRUE when the first operand is not equal to the second operand. The operand types are BOOL, BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL, LREAL, TIME, DATE, TIME_OF_DAY, DATE _AND_TIME and STRING.

Syntax: Its textualized language syntax format is as follows, bResult:=bVar1<>bVar2.

[Example 3.32] tTime is an event type variable, compare it with the fixed value of 1 hour, 20 minutes and 10 seconds, if the current value tTime is not equal to the fixed time, the program outputs the run signal K1, the declaration part of this judgment program is shown below, the specific program is shown in Figure 3.40.

```
PROGRAMPLC_PRG
VAR_INPUT
tTime:TIME;//Input time value 1
END_VAR
VAR_OUTPUT
K1AT%QX0.0:BOOL;//output indication
END_VAR
```

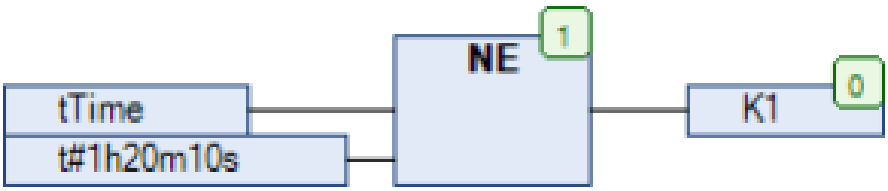



Figure 3.40 Example of a CFC program application that is not equal to a function

Example 3.33] A wind turbine needs to monitor the generator temperature in real time. The program is required to issue a generator overheat warning when the generator temperature is greater than or equal to 90 degrees Celsius and the time is maintained for more than 1min; and a generator under temperature warning when the temperature is less than or equal to 60 degrees Celsius and the time is maintained for more than 1min. The variable assignment table is shown in Table 3-30.

| Variable Name | Description |
|---|---|

| | |
|---|---|
| AI_rGenTemperaturer | Generator temperature |
| bTempHighAlarm | High temperature alarm |
| bTempHighAlarm | Low temperature alarm |

Table 3-30 Variable assignment table

The program compares the input temperature AI_rGenTemperature by means of the GE and LE instructions. The output result is assigned directly to the input IN of the timer. the program is shown in Figure 3.41

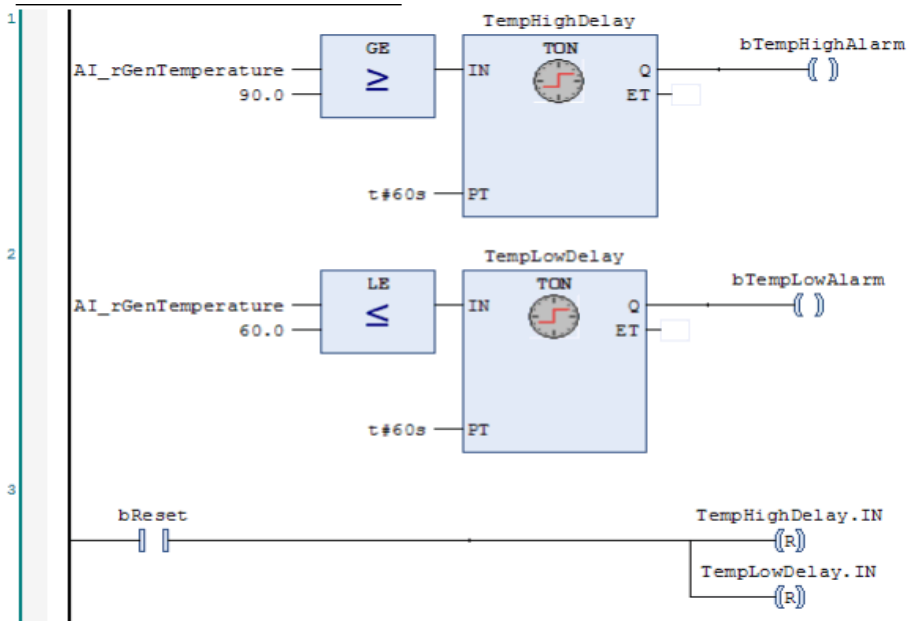



Figure 3.41 Example of generator temperature alarm program

## 3. 4. 3 Shift instruction

Shift operation instructions are frequently used instructions in CodeSys, which are divided into two categories: bitwise shift instructions and circular shift instructions. Its function is to shift all bits of the operand by 1 bit n times in the manner specified by the operation instruction and send the result to the return value. The graphical and textual instruction table of the shift instruction table is detailed in Table 3-31.

| Shift command | Graphical language | Textualized Language | Description |
|---|---|---|---|
| Shift command | SHL ??? ??? | SHL | Shift left by position |
| | SHR ??? ??? | SHR | Shift right by position |
| | ROL ??? ??? | ROL | Cyclic left shift |
| | ROR ??? ??? | ROR | Circular right shift |

Table 3-31 Table of graphical and text-based instructions for shift instructions

### 3. 4. 3. 1 Shift Left by Bit SHL

Function: Shift the operand left by bit, the left shift out bit is not processed, the right empty bit is automatically filled with 0.

Syntax: The instruction can shift the data in the input IN by n bits left, and the output result is assigned to OUT. shifting a binary number by one bit left is equivalent to multiplying the original number by 2. BYTE, Word and DWORD values will be filled to zero if n is greater than the data type width. The format of the textualized language syntax is shown below.

OUT:=SHL(IN,n)

Example 3.34] utilizes the shift-by-bit left instruction to shift the current value of the WORD type input variable wWord1 by 4 bits.

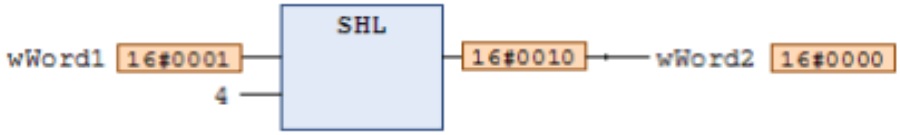



Figure 3.42 Example of a bitwise left shift program

wWord1 is 0001 in hexadecimal, and after shifting 4 bits to the left by bit, the final output is 16#0010. The process is shown in Figure 3.43. The empty bits in the lower 4 bits are filled with zeros.

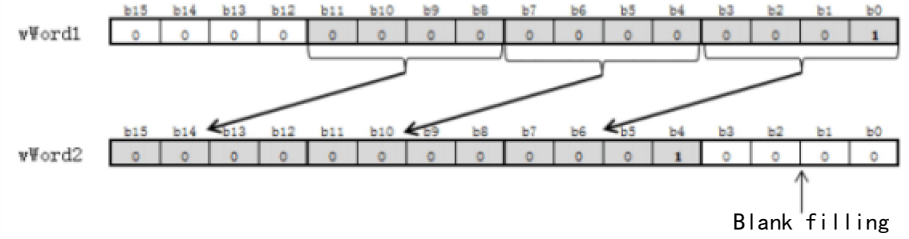



Figure 3.43 Bit-by-bit left shift 4-bit process

The total number of bits in the shift operation is affected by the data type of the input variable. If the input variable is a constant, the data type with the smallest length will be taken. The output variable's data type has no effect on the arithmetic operation, as the following example identifies.

[Example 3.35] Compare the following bitwise left-shift operations on hexadecimal numbers. Although the values of the input variables in byte and word form are equal, erg_byte and erg_word will give different results depending on the data type of the input variable (BYTE or WORD).

```
VAR
in_byte:BYTE:=16#45.
in_word:WORD:=16#45.
erg_byte:BYTE.
erg_word:WORD.
n:BYTE:=2.
END_VAR
erg_byte:=SHL(in_byte,n);(*result is 16#14*)
erg_word:=SHL(in_word;n);(*results in 16#0114*)
```

When bits b6 and b7 of the BYTE type variable overflow after a two-bit left shift, the final data is 14 in hexadecimal. and when bits b6 and b7 of the WORD type variable overflow after a two-bit left shift into bits b8 and b9 of the high byte, the bits will remain reserved, and the final result is 114 in hexadecimal. the process is shown in Figure 3.44.

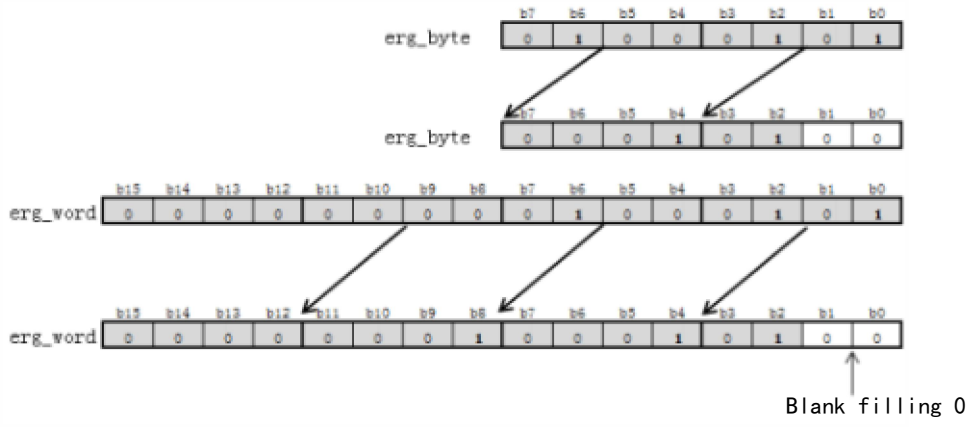



Figure 3.44 Comparison of BYTE and WORD Variables by Bit Left Shift

3. 4. 3. 2 Shift Right by Bit SHR

Function: Shift the operand to the right by bit, the right shifts out bit is not processed, and the left empty bit is automatically filled with 0.

Syntax: The instruction can shift the data in the input IN by n bits to the right, the output result is assigned to OUT, the binary number is shifted one bit to the right equivalent dividing the original number by 2.

If n is greater than the data type width, BYTE, Word and DWORD values will be filled with zero. If a signed data type is used, the arithmetic shift will complement the number by the highest digit.

The format of its textualized language syntax is shown below:

OUT:=SHR(IN,n)

【Example 3.36】Use bitwise right shift instruction to complete the WORD type input variable wWord1 current value right shift 5 bits, the output result assigned to wWord2, the program as shown in 4.45

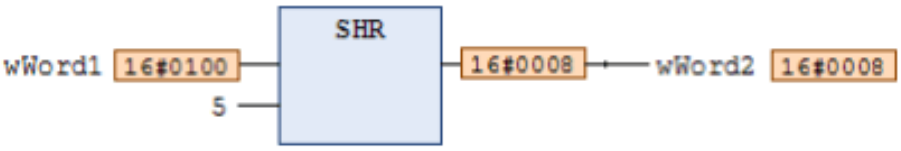



Figure 3.45 Example of a bitwise right shift procedure

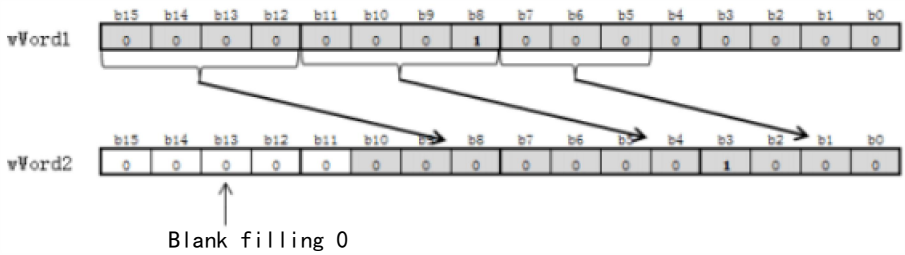



Figure 3.46 The process of shifting 5 bits by bit to the right

As above, wWord1 is 0100 in hexadecimal, and after shifting 5 bits to the right, the final output is 16#0008. Since WORD type variables belong to unsigned data type, valid values range from 0 to 65535, so after shifting 5 bits to the right, there is no sign bit, and the higher 5 bits are complemented by 0. The shifting process is shown in Figure 3.46.

[Example 3.36] is a right shift of unsigned data. If signed integer data is encountered, the higher right shift

needs to be complemented by a sign bit. The following example 4.37 is shown.

[Example 3.37] Use the shift right instruction to shift the current value of the INT type input variable iINT1 by 4 bits and assign the output to iINT2.

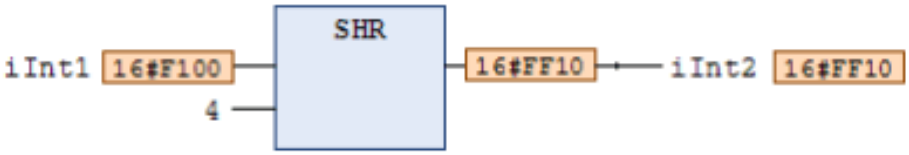



Figure 3.47 Example of a bitwise right shift program with sign bits

As above, because INT is signed bit data, valid values for -32768 ~ 32767, iINT1 for hexadecimal signed data F100, the highest bit b15 for the symbol bit, after shifting 4 bits to the right, the need to fill the data, because the source data symbol bit is 1, so the high 4 bits to fill 4 1, so the program runs the final result of 16 # FF10, the specific shift process as shown in Figure 3.48 The specific shift process is shown in Figure 3.48.

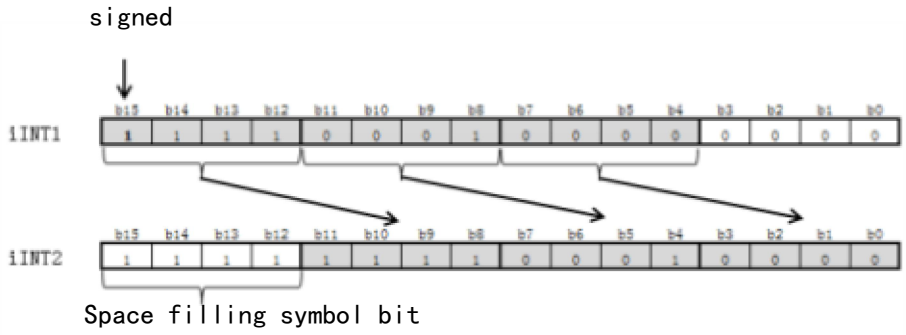



Figure 3.48 Bitwise right shift 4-bit process with sign bit

3. 4. 3. 3 Cyclic left shift ROL

Function: The operand is cyclically shifted left by bit, and the bit shifted out on the left is directly added to the lowest bit on the right.

Syntax: Allowed data types: BYTE, WORD, DWORD. Use this instruction to shift the entire contents of input IN cyclically left bit by bit, and the empty bit is filled with the signal state of the shifted bit. The input parameter n provides the value indicating the number of bits to be cyclically shifted, and OUT is the result of the cyclic shift operation. Its textualized language syntax format is shown below.

OUT:=ROL(IN,n)

Example 3.38] Create a POU and try to compare the difference between bitwise left shift and circular left shift. Move the hexadecimal WORD-type variable wWord1 by the same number of bits using two different left shift methods and try to compare the results.

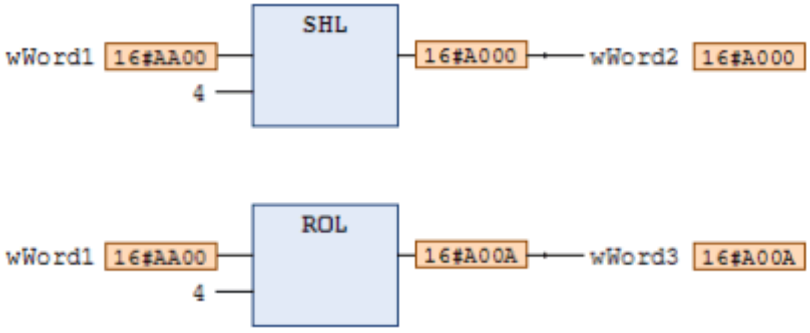



Figure 3.49 Example of a bitwise left shift cyclic left shift comparison program

Through Example 3.38 it is easy to see that the use of circular right shift after the output wWord3 b0 ~ b3 bits is not to fill the empty bit 0, but the input data wWord1 in b12 ~ b15 in 1010 to b0 ~ b3 bits, the detailed process is shown in 3.50 illustration.

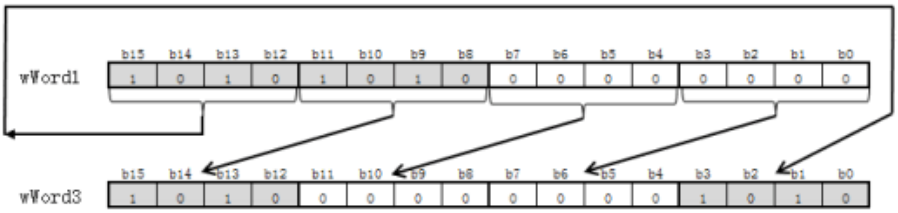



Figure 3.50 Example of cyclic left shift 4-bit program

The total number of bits in a circular shift instruction is also affected by the data type of the input variable. If the input variable is a constant, the data type with the smallest length will be taken. The data type of the output variable has no effect on the arithmetic operation, as can be seen in the following example.

[Example 3.39] Compare the following operations of cyclic left shift of hexadecimal numbers. Although the values of the input variables in the form of byte and word are equal, erg_byte and erg_word will give different results depending on the data type of the input variable (BYTE or WORD).

```
VAR
in_byte:BYTE:=16#45.
in_word:WORD:=16#45.
erg_byte:BYTE.
erg_word:WORD.
n:BYTE:=2.
END_VAR
erg_byte:=ROL(in_byte,n);(*results in 16#15*)
erg_word:=ROL(in_word,n);(*results in 16#0114*)
```

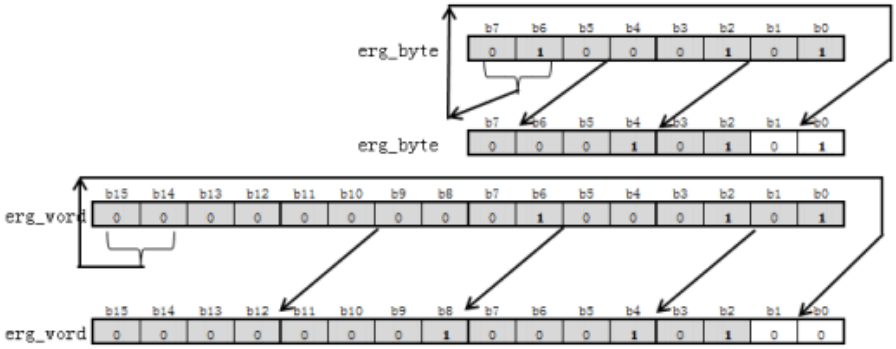



Figure 3.51 Comparison of BYTE and WORD variables cyclic left shift

As shown in Figure 3.51, when bits b6 and b7 of the BYTE variable are shifted by 2 bits left to bits b0 and b1 of the output data, the final data is 15 in hexadecimal, while when bits b6 and b7 of the WORD variable are shifted by 2 bits left to bits b8 and b9 of the output data, bits b14 and b15 of the original data are shifted to bits b0 and b1 of the output data after the left shift. Therefore, the result is 114 in hexadecimal.

### 3. 4. 3. 4 Cyclic Right Shift ROR

Function: The operand is cyclically shifted to the right by bit, and the bit shifted out on the right is directly added to the highest bit on the left.

Syntax: Allowed data types: BYTE, WORD, DWORD. use this instruction to cycle the entire contents of the input IN shifted right bit by bit, and the empty bits are filled with the signal state of the shifted bit. The input parameter n provides a numerical value indicating the number of bits shifted in the loop, and OUT is the result of the loop shift operation. The textualized language syntax format is shown below.

```
OUT:=ROR(IN,n)
```

【Example 3.40】Use the loop by bit right shift instruction to complete the WORD type input variable wWord1 current value loop right shift 5 bits, the output result assigned to wWord2, the program as shown in 4.52.

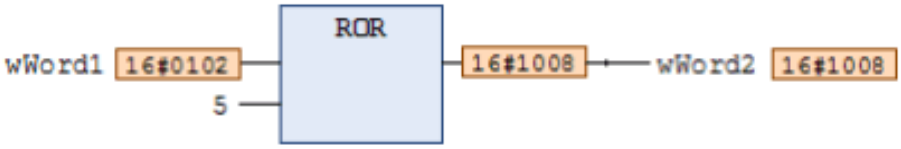



Figure 3.52 Example of a cyclic shift-right program

The result of the operation of the program for the hexadecimal 1008, the program will be the original wWord1 low 5 bits b0 ~ b4 to wWord2 in b11 ~ b15, its program shift process as shown in Figure 3.53.

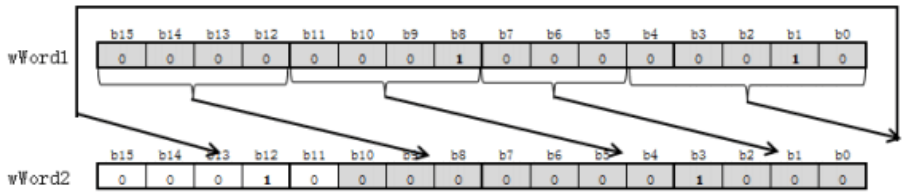



Figure 3.53 Cyclic shift right by 5 bits process

[Example 3.41] Using the cyclic right-shift instruction, design a 16-bit flashing program that flashes to the right one by one at a frequency of 2s. Through the output of the power-on delay timer every 2 seconds to trigger the instruction loop right shift instruction, its program as shown in Figure 3.54.

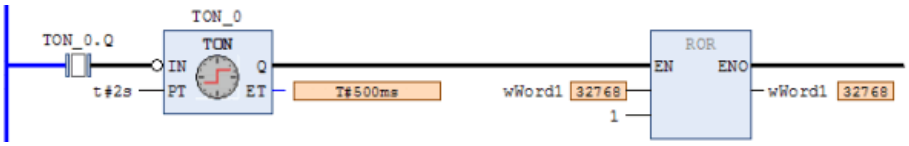



Figure 3.54 Cyclic right shift flashing light program

# 3. 5 Computing Instructions

A computing instruction performs an operation on an operand and produces the result of the operation. A computing instruction is a special symbol that deals with data operations, and the combination of data variables with an operation instruction forms a complete program operation statement. This section provides a detailed description of the commonly used operation instructions in CodeSys, including assignment instructions, arithmetic operations, mathematical operations, and address operations.

## 3. 5. 1 Assignment Instructions

The assignment instruction is the most used instruction in CodeSys. In practice, he implements the function of transferring data from one variable to another variable.

### 3. 5. 1. 1 Assignment instruction MOVE

Function: Assign the value of a constant or variable to another variable, and is the most common command in CodeSys, assignment means

The graphical and textual command tables of the order form are detailed in Table 3-32.

| Shift command | Graphical language | Textualized Language | Description |
|---|---|---|---|
| Shift command | MOVE<br>??? | := | Assignment |

Table 3-32 Table of graphical and text-based instructions for assignment instructions

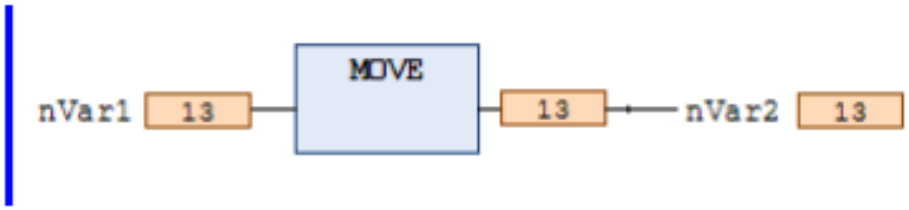



Figure 3.55 Example of MOVE instruction program

Example 3.42] Create a POU to assign the data in the WORD variable nVar1 to nVar2, and implement the program as follows. If you use the textualization instruction, the code in Example 3.42 is: nVar2:=nVar1.

## 3. 5. 2 Arithmetic operations instructions

The +, -, *, /, and MOD instructions are all arithmetic instructions for adding, subtracting, multiplying, dividing, and modulo.

These arithmetic operations instructions are explained in detail below. Details of their instructions are shown in Table 3-33.

| Arithmetic operations instructions | Graphical language | Textualized Language | Description |
|---|---|---|---|
| Arithmetic operations instructions | ADD ??? ??? + | + | add up |
| | SUB ??? ??? − | - | minus |
| | MUL ??? ??? × | * | multiply |
| | DIV ??? ??? / | / | integer division |
| | MOD ??? ??? | MOD | modulus |

Table 3-33 Table of graphical and text-based instructions for arithmetic operations

### 3. 5. 2. 1 Additive operations ADD

Function: Addition instruction, two (or more) variables or constants are added together. Time variables can also be added, and the result is another a time variable.

Syntax: The instruction can do the addition operation from the value of input variable IN0 to the value of INn and assign the result to OUT.

Addition operation support the following variable types: BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, (L)REAL, TIME, and constant. Its textualized language syntax format is shown below.

OUT:=IN0+...+INn

[Example 3.43] Create a POU, declare two integer variables iVar1 and iVar2, and assign iVar1 to 2014,

and then make the value of iVar2 the value of iVar1 and iVar1 after adding them together, as follows.

```
VAR
iVar1:INT: = 2014.
iVar2:INT.
END_VAR

iVar2:=iVar1+iVar1.
```

The result of running the program is iVar2 equals 4028.

Example 3.44] In practical engineering, it is often necessary to record the number of operations. Using the ST programming language, when the number accumulates to 10, the accumulation variable is cleared to zero. The following is the program implementation in ST language. The rising edge triggers the function block to accumulate the added number iCounter.

```
VAR
bCalStart:BOOL.
FB_StartTrigR_TRIG:R_TRIG.
iCounter:word.
END_VAR

FB_StartTrigR_TRIG(CLK:=bCalStart).
IFFB_StartTrigR_TRIG.QTHEN
iCounter:=iCounter+1.
END_IF
IFiCounter=10THEN
iCounter:=0.
END_IF
```

Notes: Time-type variables can also be used with the addition function, where two TIME variables are added to get a new time. Example: t#45s+ t#50s=t#1m35s. The selected output data type should be able to store the output result, otherwise it may cause data error.

### 3. 5. 2. 2 Subtraction Operations SUB

Function: Subtraction instruction, two variables or constants are subtracted from each other.

Syntax: The instruction can subtract the value of IN1 from the input variable IN0 and assign the result to OUT. The subtraction instruction supports the following variable types, BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL, (L)REAL, TIME, and constant. The format of its textualized language syntax is shown below.

```
OUT:=IN0-IN1
```

[Example 3.45] Create a POU, declare two floating-point variables rVar1 and rVar2, and assign them the values 3.14 and 10 respectively.

```
VAR
rVar1:REAL:=3.14.
rVar2:REAL:=10.
rResult:REAL.
END_VAR
rResult:=rVar2-rVar1.
```

The program runs with rResult equal to 6.86.

Notes: TIME-type variables can also be used with the subtraction function, where two TIME variables are subtracted to get a new time.

Example: t#1m35s - t#50s = t#45s, but the time result cannot have negative values.

TOD type variables can also use the subtraction function, where two TOD types are subtracted to get a new TIME type data.

Example: TOD#45:40:30-TOD#22:30:20=T#1390m10s0ms, but the time result cannot have a negative value.

### 3. 5. 2. 3 Multiplication operations MUL

Function: Multiplication instruction, two (or more) variables or constants are multiplied together.

Syntax: The instruction can multiply the value of the input variable IN0 up to INn and assign its product to OUT. the multiplication instruction supports the following variable types, BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, and

UDINT, (L)REAL, TIME, TOD, and constants. The format of its textualized language syntax is shown below.

OUT:=IN0*...*INn

[Example 3.46] Create a POU, declare two plastic variables iVar1 and iVar2, and assign them to 10 and 2 respectively, and then declare an integer variable iResult so that the result is the product of iVar1 and iVar2.

```
VAR
iVar1:INT:=10.
iVar2:INT:=2.
iResult:INT.
END_VAR
iResult:=iVar1*iVar2.
```

The program runs with the result that iResult is equal to 20.

### 3. 5. 2. 4 Integer Division Operations DIV

Function: Division instruction, two variables or constants are divided by each other.

Syntax: The instruction can divide the input variable IN0 by the value of IN1 and assign the resulting quotient value to OUT. the division operation instruction supports the following variable types, BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL, LREAL, and constant. Its textualized language syntax format is shown below.

OUT:=IN0/IN1

[Example 3.47] Create a POU, declare two integer variables iVar1 and iVar2, and assign the values to 10 and 2 respectively, then declare an integer variable iResult so that its value is the value obtained by dividing iVar1 by iVar2.

```
VAR
iVar1:INT:=10.
iVar2:INT:=2.
iResult:INT.
END_VAR

iResult:=iVar1/iVar2.
```

The program runs with the result that iResult is equal to 5.

Caution:

When using the DIV instruction in your project, you can use the CheckDivByte, CheckDivWord, CheckDivDWord and CheckDivReal instructions to check if the divisor is zero, avoiding the divisor to be zero

### 3. 5. 2. 5 Modulo Operations MOD

Function: The variable or constant is divided by the remainder, and the result is the remainder after dividing the two numbers, which is an integer data.

Syntax: The remainder instruction MOD instruction assigns the remainder of the input variable IN0 divided by IN1 to OUT, which is usually used to create an equation with a remainder in a specific range. The remainder instruction supports the following variable types, BYTE,

WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL, LREAL, and constants. The format of its textualized language syntax is shown below.

OUT:=IN0 MOD IN1.

Example 3.48] Create a POU, declare two integer variables iVar1 and iVar2, and assign them to 44 and 9 respectively, and then declare an integer variable iResult, so that its value is the value of iVar1 and iVar2 after the remainder operation.

```
VAR
iVar1:INT:=44.
iVar2:INT:=9.
iResult:INT.
END_VAR
```

iResult:=iVar1 MOD iVar2.
The program runs with the result that iResult is equal to 8.

## 3. 5. 3 Mathematical operation commands

### 3. 5. 3. 1 Basic commands

Mathematical operation instructions include trigonometric operation instructions, advanced arithmetic instructions, and the related instruction table is shown in Table 3-34.

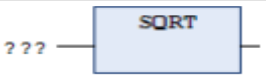
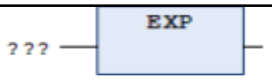
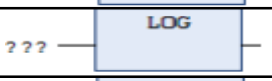
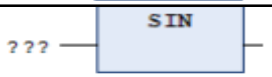
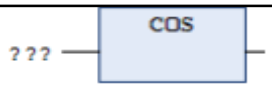
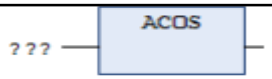
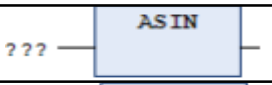
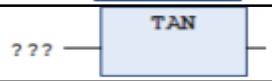
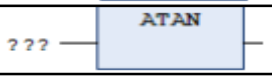
| Mathematical operations commands | Graphical language | Textualized Language | Description |
|---|---|---|---|
| Mathematical operations commands |  | ABS | Absolute value |
| |  | SQR | Square root |
| |  | EXP | Exponential |
| |  | LN | Natural logarithm |
| |  | LOG | Logarithmic |
| |  | SIN | Sine |
| |  | COS | Cosine |
| |  | ACOS | Inverse cosine |
| |  | ASIN | Inverse sine |
| |  | TAN | Tangent |
| |  | ATAN | Inverse tangent |

Table 3-34 Table of graphical and textual instructions of mathematical operation instructions

1. Absolute ABS[FUN]

Function: This function instruction is used to calculate the absolute value of a number and has no relationship with the positive or negative sign number sign.

Syntax: The absolute value operation instruction supports the following variable types, BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL, LREAL and constants. Its textualized language syntax format is shown below.

OUT:=ABS(IN).

[Example 3.49] Example of ABS function.

```
VAR
iVar1:INT:=-44.
iResult:INT.
END_VAR
iResult:=abs(iVar1).
```

The program runs with the result that iResult is equal to 44.

2. Square root SQRT[FUN]

Function: The square root of a non-negative real number.

Syntax: The input variable IN can be BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL, LREAL and constants, but the output must be of type REAL or LREAL. The format of its textualized language syntax is shown below.

OUT:=SQRT(IN).

[Example 3.50] Example of SQRT function.
VAR
rVar1:REAL:=16.
rResult:REAL.
END_VAR

rResult:=SQRT(rVar1).
The program runs with the result that rResult is equal to 4.

3. Exponential function EXP[FUN]

Function: Returns the power of e (the base of the natural logarithm), which is a number with constant 2.71828.

Syntax: The input variable IN can be BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL, LREAL and constants, but the output must be of type REAL or LREAL. The format of its textualized language syntax is shown below.

OUT:=EXP(IN).

[Example 3.51] EXP function example.
VAR
rVar1:REAL:=2.
rResult:REAL.
END_VAR
rResult:=EXP(rVar1).
The result of running the program is rResult equal to 7.389056.

4. Natural logarithm LN [FUN]

Function: Returns the natural logarithm of a number. The natural logarithm is underlain by the constant term e(2.71828182845904).

Syntax: The input variable IN can be BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL, LREAL and constants, but the output must be of type REAL or LREAL. The format of its textualized language syntax is shown below.

OUT:=LN(IN).
[Example 3.52] Example of LN function.
VAR
rVar1:REAL:=45.
rResult:REAL.
END_VAR

rResult:=LN(rVar1).
The program runs with the result that rResult is equal to 3.80666.

5. Logarithmic of a base 10 LOG[FUN]

Function: Returns the logarithm of a number with base 10.

Syntax: The input variable IN can be BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL, LREAL and constants, but the output must be of type REAL or LREAL. The format of its textualized language syntax is shown below.

OUT:=LOG(IN).
[Example 3.53] Example of LOG function.
VAR
rVar1:REAL:=314.5.
rResult:REAL.
END_VAR

rResult:=LOG(rVar1).
The program runs with the result that rResult is equal to 2.49762.

6. Sine function SIN[FUN]

Function: Sine function.

Syntax: The input variable IN can be BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT,

UDINT, REAL, LREAL and constants, but the output must be of type REAL or LREAL. The format of its textualized language syntax is shown below.

OUT:=SIN(IN).

【Example 3.54】SIN function example.

```
VAR
rVar1:REAL:=0.5.
rResult:REAL.
END_VAR
rResult:=SIN(rVar1).
```

The program runs with the result that rResult is equal to 0.479426.

[Example 3.55] The following program converts the sine of an angle by using the arithmetic instruction. Before using the trigonometric instruction, convert the angle value to radian value, and then use the SIN instruction to find the sine value. The program is shown in Figure 3.56.

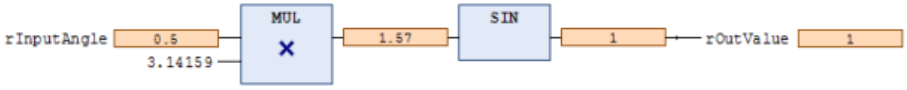



Figure 3.56 Sine SIN instruction program example

7. Cosine function COS[FUN]

Function: Cosine function.

Syntax: The input variable IN can be BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL, LREAL and constants, but the output must be of type REAL or LREAL. The format of its textualized language syntax is shown below.

OUT:=COS(IN).

[Example 3.56] COS function example.

```
VAR
rVar1:REAL:=0.5.
rResult:REAL.
END_VAR

rResult:=COS(rVar1).
```

The result of running the program is rResult equal to 0.877583.

8. Inverse cosine function ACOS[FUN]

Function: Cosine radian (inverse cosine function).

Syntax: The input variable IN can be BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL, LREAL and constants, but the output must be of type REAL or LREAL. The format of its textualized language syntax is shown below.

OUT:=ACOS(IN).

[Example 3.57] Example of ACOS function.

```
VAR
rVar1:REAL:=0.5.
rResult:REAL.
END_VAR

rResult:=ACOS(rVar1).
```

The program runs with the result that rResult is equal to 1.0472.

9. Inverse Sine function ASIN[FUN] anyway

Function: Sine radian (inverse sine function).

Syntax: The input variable IN can be BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL, LREAL and constants, but the output must be of type REAL or LREAL. The format of its textualized language syntax is shown below.

OUT:=ASIN(IN).

[Example 3.58] Example of ASIN function.

```
VAR
rVar1:REAL:=0.5.
rResult:REAL.
```

END_VAR

rResult:=ASIN(rVar1).
The result of running the program is rResult equal to 0.523599.

10. Tangent function TAN[FUN]
Function: The tangent function.
Syntax: The input variable IN can be BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL, LREAL and constants, but the output must be of type REAL or LREAL. The format of its textualized language syntax is shown below.
OUT:=TAN(IN).
[Example 3.59] Example of TAN function.
VAR
rVar1:REAL:=0.5.
rResult:REAL.
END_VAR

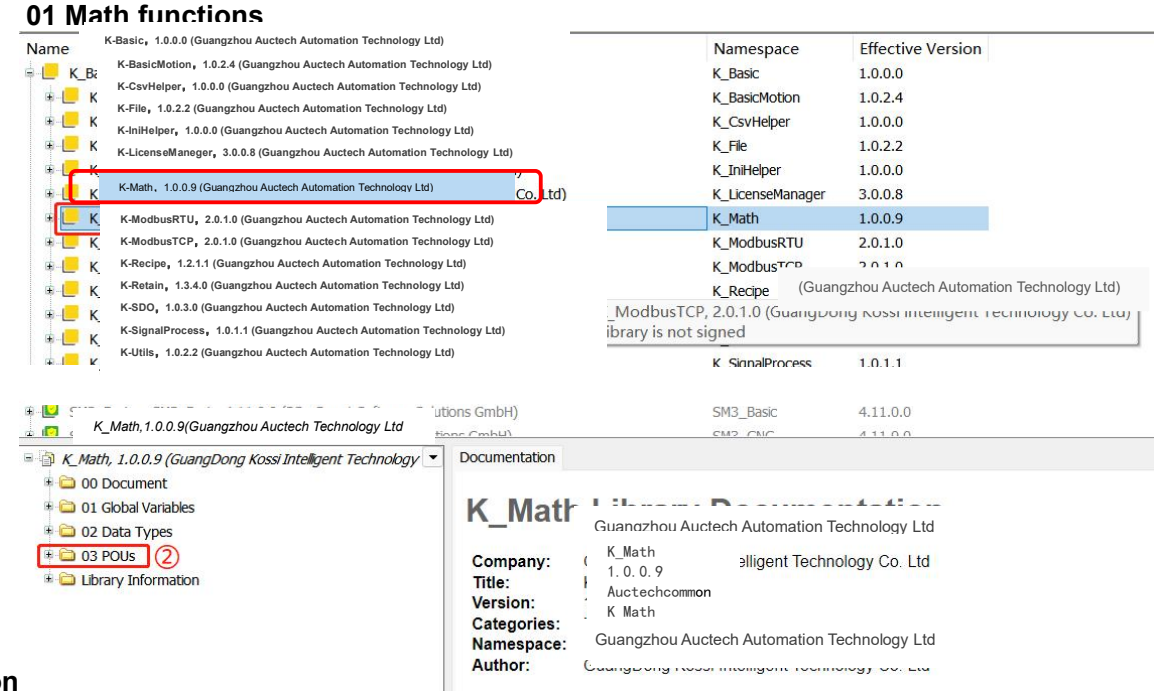
rResult:=TAN(rVar1).
The program runs with the result that rResult is equal to 0.546302.

11. Inverse tangent function ATAN[FUN]
Function: The tangent radian (inverse tangent function).
Syntax: The input variable IN can be BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL, LREAL and constants, but the output must be of type REAL or LREAL. The format of its textualized language syntax is shown below.
OUT:=ATAN(IN).
[Example 3.60] Example of ATAN function.
VAR
rVar1:REAL:=0.5.
rResult:REAL.
END_VAR

rResult:=ATAN(rVar1).
The program runs with the result that rResult is equal to 0.463648.

3. 5. 3. 2 Expansion Instructions

- **01 Math functions**




**section**

- **ATAN2 [FUN]**

Azimuth from the origin to the point (x,y), the angle with the x-axis

1) Instruction

| Instruction | Graphical language | Textualized Language |
|---|---|---|
| ATAN2 |  | ATAN2(x:=,y:=). |



2) Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | ATAN2 | REAL | |
| Input | x | REAL | Floating point value representing the x-axis coordinates |
| | y | REAL | Floating point value representing the y-axis coordinate |

3) Example

- **Ceil [FUN]**

Rounding up

1) Instruction

| Instruction | Graphical language | Textualized Language |
|---|---|---|
| Ceil |  | Ceil(X:=). |



2) Variables

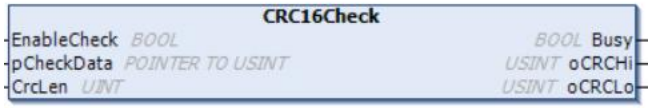
| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | Ceil | DINT | |
| Input | X | LREAL | |

3) Example

- **CRC16Check(FB)**

Hexadecimal CRC check

1) Instruction

| Instruction | Graphical language | Textualized Language |
|---|---|---|
| CRC16Check |  | CRC16Check(EnableCheck:=,<br>pCheckData:=.<br>CrcLen:=). |

2) Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Input | EnableCheck | BOOL | Enable the function to make the function works |
| | pCheckData | POINTTOUSINT | |
| | CrcLen | UINT | Crc length |
| Out | Busy | BOOL | |
| | oCRCHi | USINT | High 8 bits |
| | oCRCLo | USINT | Lower 8 bits |

3) Example

- **DegToRad(FUN)**

Angle to arc

1) Instruction

| Instruction | Graphical language | Textualized Language |
|---|---|---|
| DegToRad |  | DegToRad(deg:=). |

2) Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | DegToDad | REAL | |
| Input | deg | REAL | Angle values in degrees |

3) Example

- **Floor(FUN)**

Rounding down

1) Instructions

| Instruction | Graphical language | Textualized Language |
|---|---|---|
| Floor |  | Floor(x:=). |

2) Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | Floor | DINT | |
| Input | x | LREAL | |

3) Example

- **Fmod(FUN)**

Floating data modulus

1) Instructions

| Instruction | Graphical language | Textualized Language |
|---|---|---|
| Fmod |  | Fmod(x:=, m:=). |

2) Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | Floor | DINT | |

| Scope | Name | Type | Comment |
|---|---|---|---|
| Input | x | LREAL | Divisor |
| | m | LREAL | Divide the number, an invalid input m=0 will return 0 |

3) Example

- **ParamPeriodLimits(FB)**

Periodic value processing, automatic modulo, negative values to positive

1) Instructions

| Instruction | Graphical language | Textualized Language |
|---|---|---|
| ParamPeriodLimits |  | ParamPeriodLimits(<br>rIn:=, Period:=,<br>Scale:=,ScaledOutput=>). |

2) Variables

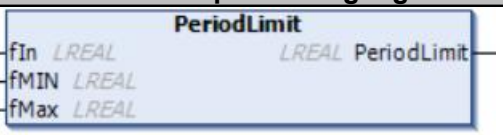
| Scope | Name | Type | Init | Comment |
|---|---|---|---|---|
| Inp | rIn | REAL | | Set value, beyond Period automatically modified |
| | Period | REAL | 360 | Take the number of cycles |
| | Scale | REAL | 1 | Convert the scale, rIn processing result value *Scale for ScaleOutput output |
| Output | ScaledOutput | REAL | | |

3) Example

- **PeriodLimit(FUN)**

Periodic value processing

1) Instructions

| Instruction | Graphical language | Textualized Language |
|---|---|---|
| PeriodLimit |  | PeriodLimit(<br>fIn:=, the<br>fMIN:=, the<br>fMax:=). |

2) Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | PeriodLimit | LREAL | |
| Input | fIn | LREAL | Value to be processed |
| | fMIN | LREAL | Cycle Minimum |
| | fMax | LREAL | Periodic Maximum |

3) Example

(*Example declaration*) fPeriodValue1: LREAL.
(*Example result is 5*) fPeriodValue1 := PeriodLimit(fIn:=365,fMIN:=0,fMax:=360).
(*Example declaration*) fPeriodValue1 :LREAL.
(*Example result is 90*) fPeriodValue2 := PeriodLimit(fIn:=-20,fMIN:=-10,fMax:=100).

- **RadToDeg(FUN)**

Radian to Angle degree

1) Instructions

| Instruction | Graphical language | Textualized Language |
|---|---|---|
| RadToDeg |  | RadToDeg(rad:=). |

2) Variables

| Scope | Name | Type | Comment |
|---|---|---|---|

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | RadToDeg | REAL | |
| Input | rad | REAL | Angle values in radians |

3) Example

- **Round(FUN)**

Rounding of float numbers, with decimal places reserved

1) Instructions

| Instruction | Graphical language | Textualized Language |
|---|---|---|
| Round |  | Round(In:=, N:=). |

2) Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | Round | LREAL | |
| Input | In | REAL | Values to be processed |
| | N | INT | Number of decimal places to be retained, up to 8 decimal places |

3) Example

Note

Supports up to 8 decimal places

(*Example declaration*) fValue:LREAL.

(*Example result is 120.305*) fValue:=Round(In:=120.305443,N:=3).

- **Saturation(FUN)**

Saturation function with upper and lower limits of 1, -1

1) Instructions

| Instruction | Graphical language | Textualized Language |
|---|---|---|
| Saturation |  | Saturation(value:=). |

2) Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | Saturation | REAL | |
| Input | Value | REAL | |

3) Example

- **Signum(FUN)**

Positive and negative function, positive number return 1.0, negative number return -1.0, 0 return 0.

1) Instructions

| Instruction | Graphical language | Textualized Language |
|---|---|---|
| Signum |  | Signum(value:=). |

2) Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | Signum | REAL | |
| Input | Value | REAL | |

3) Example

- **Statistic_N_Lreal(FB)**

The average of the last N times data, less than N times to take the average of all recent data, earlier than N times data will be cleared.

1) Instructions

| Instruction | Graphical language | Textualized Language |
|---|---|---|
| Statistics_ N_Lreal |  | Statistics_N_Lreal(Enale:=. lrInput:=,N:=,lrMin=>,lrMax=>,lrAverage=>,xOverrun=> ). |

2) Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Input | Enable | BOOL | Start function block, TRUE is valid, reset when FALSE |
| | lrInput | REAL | Input Value |
| | N | UINT | Find the average |
| Output | lrMin | Lreal | Minimum value |
| | lrMax | Lreal | Maximum value |
| | lrAverage | Lreal | The return value is the average of the last N input values |
| | xOverrun | Bool | Invalid input data overflow |

**02 Matrix**

- **Angel (FUN)**

Find the angle of three points using the dots product principle

1) Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|
| Angel | Find the angle between the three points |  | Angel(o:=,a:=, b:=, m:=). |

2) Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | Angel | Lreal | |
| Input | o | POINTERTOLREAL | Origin |
| | a | POINTERTOLREAL | p |
| | b | POINTERTOLREAL | q |
| | m | UINT | Dimension |

3) Example

- **Cross (FUN)**

1) Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|
| Cross | |  | Cross(a:=, b:=). |

2) Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | Cross | ARRAY[0..2]OFLREAL | |

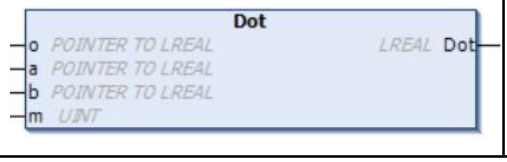
| Scope | Name | Type | Comment |
|---|---|---|---|
| Input | a | ARRAY[0..2]OFLREAL | |
| | b | ARRAY[0..2]OFLREAL | |

3) Example

- **Dot (FUN)**

Find the value of the vector op point multiplied by the vector oq

1) Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|
| Dot | Find the value of the vector op point multiplied by the vector oq |  | Dot(o:=,a:=, b:=,m:=). |

2) Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | Dot | LREAL | |
| | o | POINTEROFLREAL | Origin |
| | a | POINTEROFLREAL | p |
| | b | POINTEROFLREAL | q |
| | m | UINT | Dimension |

3) Example

- **INV(FUN)**

Inverse matrix of nth-order real matrix a by the all-choice principal Gaussian-approximation elimination method

1) Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|
| INV | Find the inverse matrix |  | INV(a:=, n:=). |

2) Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | INV | LREAL | The function returns the shaping token, 0:A singular. Otherwise: normal |
| | a | POINTEROFLREAL | Store the original matrix and its inverse matrix on return. |
| | n | UINT | p |

3) Example

- **InvertGaussJordan(FUN)**

1) Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|
| InvertGaussJordan | |  | InvertGaussJordan(elements:=, numColumns:=). |

2) Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | InvertGaussJordan | BOOL | |
| Input | elements | POINTEROFLREAL | |
| | numColumns | UINT | Number of columns of the matrix |

3) Example

- **invM(FB)**

1) Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|
| InvM | |  | InvM(numRows:=,<br>numColumns:=,<br>Elements:=.<br>Invelement:=). |

2) Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Input | NumRows | INT | |
| | NumColumns | INT | |
| | elements | POINTEROFLREAL | Matrix requiring inverse |
| | numColumns | POINTEROFLREAL | Inverse matrix |

3) Example

- **MatrixAdd(FUN)**

1) Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|
| MatrixAdd | |  | MatrixAdd(T1:=, T2:=,<br>m:=.<br>n:=, the<br>Tout:=). |

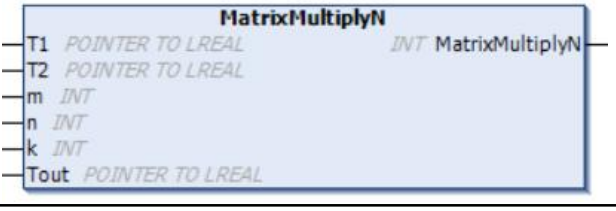
2) Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | MatrixAdd | INT | |
| Input | T1 | POINTERTOLREAL | Matrix to the left of the minus sign |
| | T2 | POINTERTOLREAL | Matrix to the right of the minus sign |
| | m | UINT | Number of matrix rows |
| | n | UINT | Matrix column number |
| | Tout | POINTERTOLREAL | Output Matrix |

3) Example

- **MatrixMultiplyN(FUN)**

1) Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|
| MatrixMultiplyN | |  | MatrixMultiplyN(T1:=,<br>T2:=,<br>m:=.<br>k:=,.<br>Tout:=). |

2) Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | MatrixMultiplyN | INT | |
| Input | T1 | POINTERTOLREAL | Matrix to the left of the multiplication sign |
| | T2 | POINTERTOLREAL | Matrix to the right of the multiplication sign |
| | m | INT | Number of rows of matrix T1 |
| | n | INT | Matrix T1 columns, matrix T2 rows |

| Scope | Name | Type | Comment |
|---|---|---|---|
| | k | INT | Number of columns of matrix T2 |
| | Tout | POINTERTOLREAL | Output Matrix |

3) Example

- **MatrixSub(FUN)**

1) Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|
| MatrixSub | |  | MatrixSub(T1:=, the<br>T2:=,.<br>m:=.<br>n:=, the<br>Tout:=). |

2) Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | MatrixSub | INT | |
| Input | T1 | POINTERTOLREAL | Matrix to the left of the minus sign |
| | T2 | POINTERTOLREAL | Matrix to the right of the minus sign |
| | m | UINT | Number of matrix rows |
| | n | UINT | Matrix column number |
| | Tout | POINTERTOLREAL | Output Matrix |

3) Example

- **MatrixTranp(FUN)**

Find the transpose of a matrix

1) Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|
| MatrixTranp | Find the transpose of a matrix |  | MatrixTranp(T1:=, the<br>m:=.<br>n:=, the<br>Tout:=). |

2) Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | MatrixSub | INT | |
| Input | T1 | POINTERTOLREAL | Matrix to be transposed |
| | m | INT | Number of rows of input matrix |
| | n | INT | Enter the number of columns of the matrix |
| | Tout | POINTERTOLREAL | Output Matrix |

3) Example

- **Norm(FUN)**

1) Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|
| Norm | |  | Norm(v:=). |

2) Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | Norm | LREAL | |
| Input | v | ARRAY[0..2]OFLREAL | The array represents a vector x,y,z |

## 3. 5. 4 Address operation instructions

In practical applications, there are many cases involving memory address instructions, such as taking the first address of an array in memory and needing to know how many bytes the array occupies in memory and other related information, the instructions involved are listed in Table 3-35, and will be introduced in this section one by one.

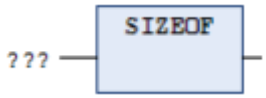
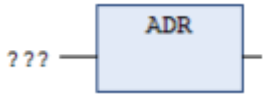
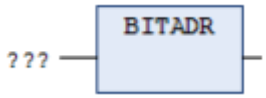
| Address operation instructions | Graphical language | Textualized Language | Description |
|---|---|---|---|
| Address operation instructions |  | SIZEOF | Data Type Size |
| |  | ADR | Address Operators |
| |  | BITADR | Bit Address Operators |

Table 3-35 Graphical and text-based instruction table of address operation instructions

1.Data type size SIZEOF

Function: Performs this function to determine the number of bytes required for the given data type. Simply put its function is to return the number of memory bytes occupied by an object or type.

Syntax: The return value of SIZEOF is an unsigned value, the return value of the type will be used to find the size of the variable IN0.

The OUT output value is in bytes and IN0 can be any data type. Its textualized language syntax format is shown below. The type of the return value is an implicit data type, and it will be determined by the actual data value, as shown in Table 3-36.

OUT:=SIZEOF(IN0);

| Return value X of SIZEOF | Implicit data types |
|---|---|
| 0<=sizeofx<256 | USINT |
| 256<=sizeofx<65536 | UINT |
| 65536<=sizeofx<4294967296 | UDINT |
| 4294967296<=sizeofx | ULINT |

Table 3-36 Return data types of SIZEOF

Example 3.61] An example of using the SIZEOF instruction to fetch the memory size occupied by an array is shown in the following program.

Example of ST language: Example of IL language:

```
VAR
arr1:ARRAY[0..4]OFINT.
var1:INT.
END_VAR

var1:=SIZEOF(arr1)
```

```
LD      arr1
SIZEOF
ST      Var1
```

.

The program assigns the result to var1, and finally var1 equals 10, because the arr1 array consists of 5 INT integer elements, and the result unit of SIZEOF is BYTE, so the final program runs as a total of 10 BYTE, which means that arr1 occupies 10 bytes of memory.

2. Address operator ADR

Function: Obtains the memory address of an input variable and outputs it. The address can be used as a pointer within the program or passed to a function as a pointer.

Syntax: The ADR operator whose return value is a DWORD address variable, IN0 can be of any data type. Its textualized language syntax format is shown below.

-OUT:=ADR(IN0).

The return value of ADR is only the memory address of the variable. The length of the data that can be stored in the memory address is 1 BYTE, and the contents of the corresponding address can be extracted by

the content operator "^". "The implementation procedure is shown below.

pt:=ADR(var_int1).

var_int2:=pt^.

Example 3.62] An example of using the ADR instruction to fetch an array is shown in the following program.

Example of ST language: Example of IL language:

VAR

arr1:ARRAY[0..4]OFINT.

dwVar:DWORD.

END_VAR

dwVar:=ADR(arr1).

```
LD      bVar
ADR
ST      dwVar
```

Example 3.63] An example of using the ADR instruction to fetch an array is shown in the following program.

Example of ST language:

VAR

arr1:ARRAY[0..4]OFINT.

dwVar:DWORD.

END_VAR

dwVar:=ADR(arr1).

Example 3.64] An example of using the ADR instruction to fetch an array is shown in the following program.

Example of ST language:

pt:POINTERTOINT.

var_int1:INT.

var_int2:INT.

pt:=ADR(var_int1).

var_int2:=pt^.

3. Bit address operator BITADR

Function: Returns the bit address information offset of the allocated variable.

Syntax: The BITADR operator whose return value is a DWORD address variable, IN0 can be of any data type. Its textualized language syntax format is shown below.

OUT:=BITADR(IN0).

The return value of ADR is only the memory address of the variable. The length of the data that can be stored in this memory address is 1 BYTE. The contents of the corresponding address can be extracted by the content operator "^". "The implementation procedure is shown below, and BITADR returns the bit offset address as a DWORD variable type. Note that the offset value depends on whether the option type address is available from the target system. the DWORD maximum value defines the memory area as shown in Table 3-37:

| Address area | Start Address | Description |
|---|---|---|
| Memory | 16x40000000 | %M |
| Input | 16x40000000 | %I |
| Output | 16x40000000 | %Q |

Table 3-37 BITADR Offset Addresses for Each Address Area

Example 3.65] Example of using the BITADR instruction to fetch bit address information.

Example of ST language: Example of IL language:

VAR

var1AT%IX2.3:BOOL.

bitoffset:DWORD.

END_VAR

bitoffset:=BITADR(var1).

The result is 80000013 in hexadecimal, the "2" in %IX2.3 is 2 Bytes, the ".3" is the 4th Bit, and the "2" is the 4th Bit.

So, its address is equal to 2*8+4=20. convert 20 in decimal to 14 in hexadecimal. and because it corresponds to the first place of the I area

The address is stored from 80000000, so it is not difficult to understand that the actual address of 14 in hexadecimal is 16#80000013.

The schematic diagram is shown in Figure 3.57. Figure 3.57 Schematic diagram of BITADR

```
LD      Var1
BITADR
ST      bitoffset
```

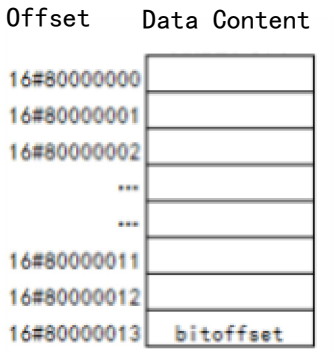



Figure 3.57 Schematic diagram of BITADR

# 3. 6 Data Conversion Instructions

## 3. 6. 1 Data type conversion instructions

Syntax: <TYPE1>_TO_<TYPE2>

Implicit conversion of "larger" data types to "smaller" data types is strictly prohibited, as information may be lost when converting from a larger to a smaller data type.

If the value being converted is outside the storage range of the target data type, the high byte of this number will be ignored. Example: Converting INT type to BYTE type or converting DINT type to WORD type.

In the conversion of <TYPE>_TO_STRING, the string is generated from the left side. If the length of the defined string is less than the length of <TYPE>, the right part will be truncated.

1) BCD code and integer data interconversion

BCD (BinaryCodedDecimal...BCD) is a 4-bit binary number to represent the value of each digit of a decimal number in parallel. For example, the BCD data 0000000000101010111 (343) is used to represent the decimal number "157" in the BIN data as shown in Figure 3.58.

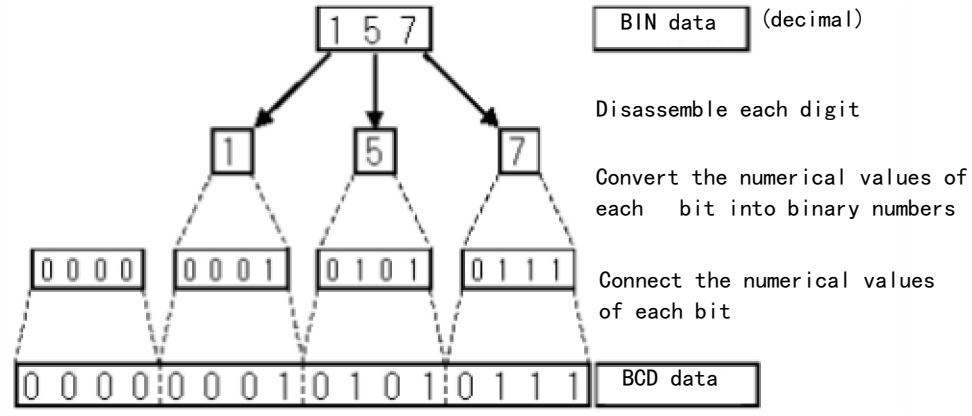



Figure 3.58 BCD example illustration

When the BCD data is stored in 16-bit memory, it can handle values from 0 to 9999 (the maximum value of 4 bits). The weights of the individual bits are shown in Figure 3.59 below.

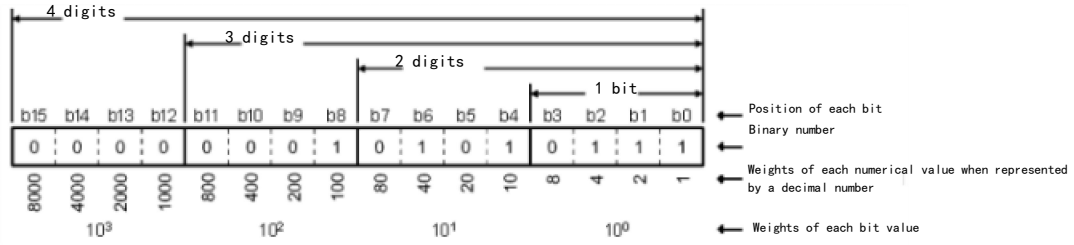



Figure 3.59 Weighting of each value of BCD in decimal

CodeSys uses BCD code and integer interconversion instructions detailed in Table 3-38. Before using this conversion function, you need to add util.library.

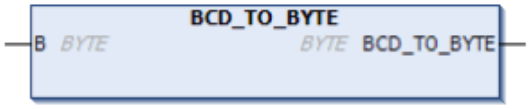
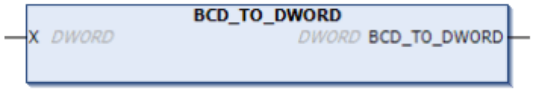
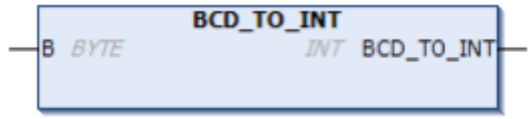
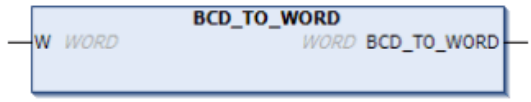
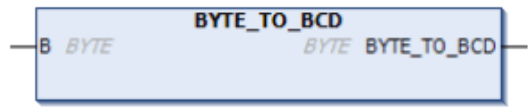
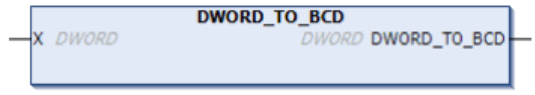
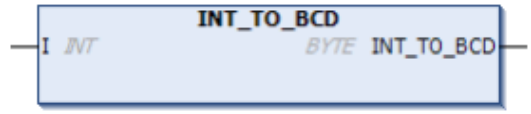
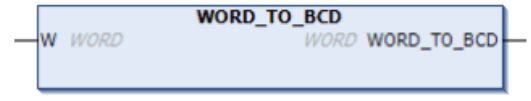
| Conversion Instructions | Graphical language | Textualized Language | Description |
|---|---|---|---|
| BCD Code with the whole Number of Mutual Conversion Instruction |  | BCD_TO_BYTE | BCD to BYTE |
| |  | BCD_TO_DWORD | BCD to DWORD |
| |  | BCD_TO_INT | BCD to INT |
| |  | BCD_TO_WORD | BCD to WORD |
| |  | BYTE_TO_BCD | BYTE to BCD |
| |  | DWORD_TO_BCD | DWORD to BCD |
| |  | INT_TO_BCD | INT to BCD |
| |  | WORD_TO_BCD | WORD to BCD |

Table 3-38 BCD code and integer interconversion instructions

[Example 3.66] Using ST programming language, convert BCD code 73 to integer data.
i:=BCD_TO_INT(73).

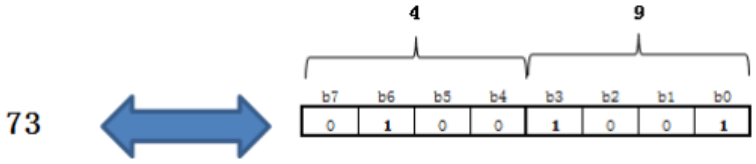



Figure 3.60 Example of BCD to INT application

As shown in Figure 3.60 above, the conversion is performed using the BCD_TO_INT instruction, and since the result of converting 73 to binary is 01001001, the result of converting it to BCD is 49.

[Example 3.67] Using ST programming language, convert integer data 73 to BCD code.

i:=INT_TO_BCD(73).

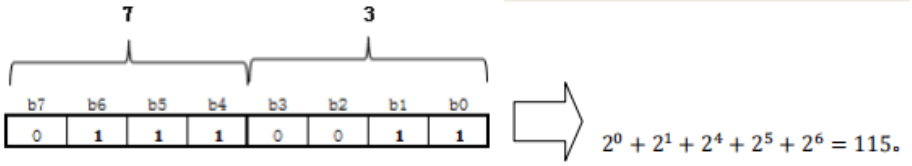



Figure 3.61 Example of INT to BCD application

As shown in Figure 3.61 above, the INT_TO_BCD instruction is used to perform the conversion. The result of the program after converting 73 in decimal to BCD code is 01110011, so the final BCD code decimal representation results in 115.

【Example 3.68】 Industrial control often encounters data setting and data display, when it is usually necessary to achieve through BCD code and integer data conversion. As shown in Figure 3.62, the user enters the data 1942 using the numeric input button, and the number needs to be added within the program to calculate, add 752, and display the result in the 7-segment code data display window.

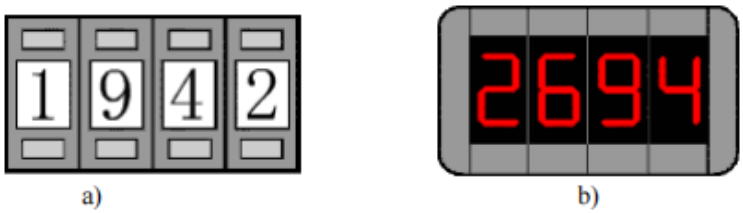



Figure 3.62 Input signal and output display

a) Input signal of the digital input switch b) Output signal of the 7-segment display (digital display)

The program is shown in Figure 3.63, the valid area is the segment program in red box, the ladder diagram sentence 1 and sentence 3 for easy understanding, the actual application does not want this conversion instruction.

When the user keys in 1942 by dialing the code, the actual BCD data received in the program is 6466, so it is necessary to set the

6466 converts to actual integer data for logical operations. Since the output of BCD_TO_INT is a BYTE type variable, the data will be overflowed if 6466 is used for conversion, so here it is necessary to use the WORD type output instruction BCD_TO_WORD to restore its BCD data to integer data. After the conversion can be carried out in the program of normal logic operations, here the addition operation, restore the valid data, through the operation of the results of the conversion to 7-segment display can recognize the final output of BCD data.

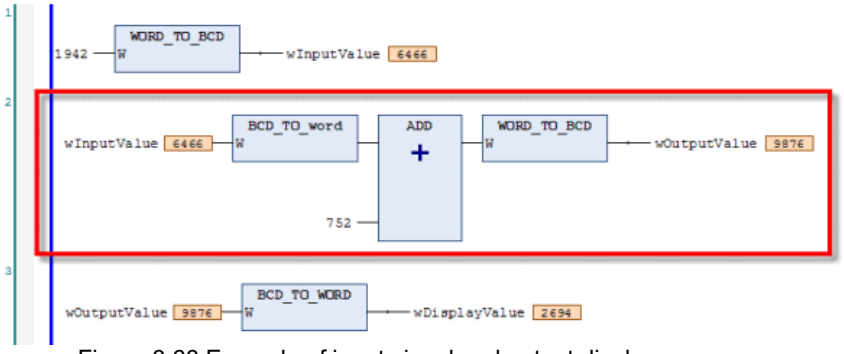



Figure 3.63 Example of input signal and output display program

2) BOOL_TO_<TYPE> Boolean type conversion data

Function: Converts a Boolean data type to other data types.

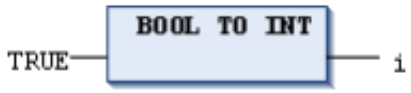
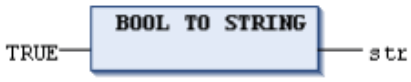
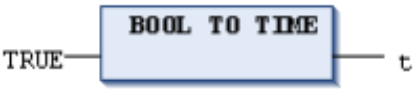
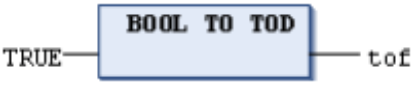
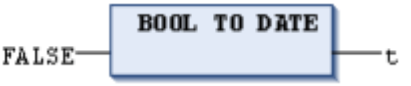
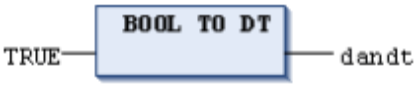
Supported data types: BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL, TIME, DATE, TOD, DT, and STRING.

a. When the output is of numeric type: If the input is TRUE, the output is 1. If the input is FALSE, the output is 0.

b. When the output is a string type: If the input is TRUE, the string 'TRUE' is output. If the input is FALSE, then

The output is the string 'FALSE'.

[Example 3.69] BOOL_TO_<TYPE> example is shown in Table 3-39 below.

| Conversion Instructions | Graphical language | Textualized Language | Results |
|---|---|---|---|
| BOOL_TO |  | i:=BOOL_TO_INT(TRUE) | 1 |
| |  | str:=BOOL_TO_STRING (TRUE) | 'TRUE' |
| |  | t:=BOOL_TO_IIME (TRUE) | T#1ms |
| |  | tof:=BOOL_TO_TOD (TRUE) | TOD#00:00:00.001 |
| |  | dat:=BOOL_TO_DATE(FALSE) | D#1970 |
| |  | dandt:=BOOL_TO_DT (TRUE) | DT#1970-01-01-00:00:01 |

3) BYTE_TO_<TYPE> Byte type conversion data

Function: Convert byte type to other data type.

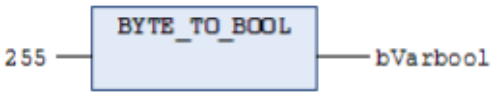
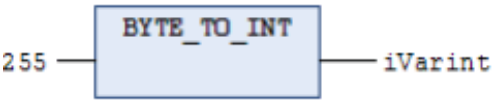
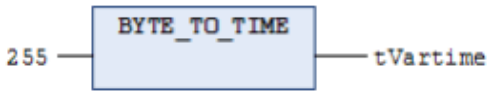
Supported data types: BOOL, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL, TIME, DATE, TOD, DT, and STRING.

a. When the output is BOOL: TRUE if the input is not equal to 0. FALSE if the input is equal to 0.

b. When the output is TIME or TOD: the input will be converted in millisecond values.

c. When the output is DATE or DT: the input will be converted in seconds value.

[Example 3.70] BYTE_TO_<TYPE> example is shown in Table 3-40 below.

| Conversion Instructions | Graphical language | Textualized Language | Results |
|---|---|---|---|
| BYTE_TO |  | bVarbool:=BYTE_TO_BOOL(255). | TRUE |
| |  | iVarint:=BYTE_TO_INT (255). | 255 |
| |  | tVartime:=BYTE_TO_TIME(255). | T#255ms |

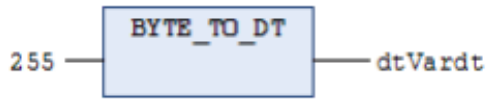
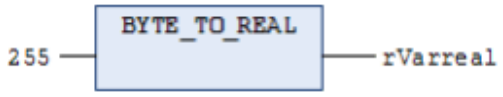
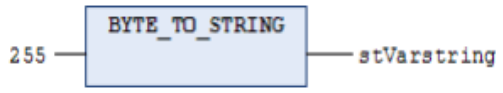
| | | | |
|---|---|---|---|
| |  | dtVardt:=BYTE_TO_DT<br>(255). | DT#1970-01-01-00:04:15 |
| |  | rVarreal:=BYTE_TO_REAL(255). | 255 |
| |  | stVarstring:=BYTE_TO_STRING(255). | '255' |

Table 3-40 Example of BYTE_TO_<TYPE> conversion ladder instruction

4) <integer data>_TO_<TYPE> integer type conversion instruction

Function: Converts integer type data to other data types.

Supported data types: BOOL, BYTE, SINT, WORD, DWORD, USINT, INT, UINT, DINT, UDINT, REAL, TIME, DATE, TOD, DT, and STRING.

a. When the output is BOOL: TRUE if the input is not equal to 0. FALSE if the input is equal to 0.
b. When the output is TIME or TOD, the input will be converted in millisecond values.
c. When the output is DATE or DT: the input will be converted in seconds value.

[Example 3.71] Since there are many integer types and the conversion process is similar, WORD_TO_<TYPE> is used as an example, as shown in Table 3-41.

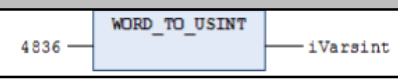
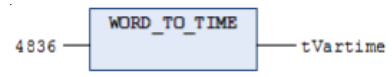
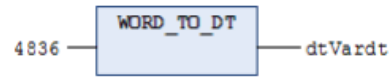
| Conversion Instructions | Graphical language | Textualized Language | Results |
|---|---|---|---|
| <shaping data>_TO |  | iVarsint:=WORD_TO_USINT(4836). | 255 |
| |  | tVartime:=WORD_TO_TIME(4836). | T#4s863ms |
| |  | dtVardt:=BYTE_TO_DT(4836). | DT#1970-01-01-01:21:03 |

Table 3-41 Example of <integer data>_TO_<TYPE> conversion ladder instruction

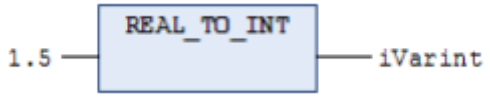
5) REAL_TO_<TYPE> real type conversion instruction

The function converts a floating-point number to other types of data. When converting floating point numbers to other types of data, the value is first rounded to an integer value, and then converted to the new quantity type.

Supported data types: BOOL, BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL, TIME, DATE, TOD, DT, and STRING.

a. When the output is BOOL: TRUE if the input is not equal to 0. FALSE if the input is equal to 0.
b. When the output is TIME or TOD: the input will be converted in millisecond values.
c. When the output is DATE or DT: the input will be converted in seconds value.

[Example 3.72] REAL_TO_<TYPE> example is shown in Table 3-42 below.

| Conversion Instructions | Graphical language | Textualized Language | Results |
|---|---|---|---|
| REAL_TO |  | iVarsint:=REAL_TO_INT(1.5). | 2 |

6) TIME_TO_<TYPE> time type conversion instruction

Function: Converts time-based data to other types of data, where time is stored internally in milliseconds as DWORD type (for TIME_OF_DAY variable starting from 00:00 am).

Supported data types: BOOL, BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL, TIME, DATE, TOD, DT, and STRING.

When the output is BOOL: TRUE if the input is not equal to 0. FALSE if the input is equal to 0.

[Example 3.73] The TIME_TO_<TYPE> example is shown in Table 3-43 below.

| Conversion Instructions | Graphical language | Textualized Language | Results |
|---|---|---|---|

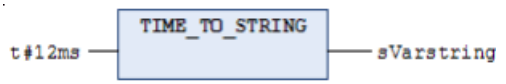
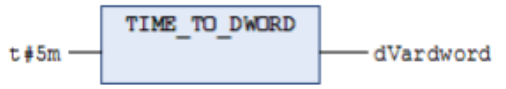
| | | | |
|---|---|---|---|
| TIME_TO |  | sVarstring:= TIME_TO_STRING(t#12ms ). | 'T#12ms' |
| |  | dVardword:= TIME_TO_DWORD(t#5m). | 300000 |

Table 3-43 Examples of TIME_TO_<TYPE> Conversion Ladder Instructions

7) DATE_TO_<TYPE> Date Type Conversion Directive

Function: Convert date type data to other type data, date is stored internally in seconds, time starts from Jan. 1, 1970.

Supported data types: BOOL, BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL, TIME, DATE, TOD, DT, and STRING.

When the output is BOOL: TRUE if the input is not equal to 0. FALSE if the input is equal to 0.

[Example 3.74] DATE_TO_<TYPE> example is shown in Table 3-44 below.

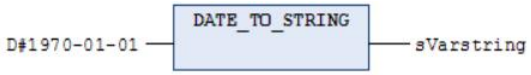
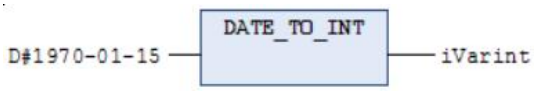
| Conversion Instructions | Graphical language | Textualized Language | Results |
|---|---|---|---|
| DATE_TO |  | sVarstring:= DATE_TO_STRING(D #1970 -01-01). | 'D#1970 -01-01' |
| |  | iVarint:= DATE_TO_INT(D#197 0- 01-01). | 29952 |

Table 3-44 Example of DATE_TO_<TYPE> conversion ladder command

8) DT_TO_<TYPE> Date Time Type Conversion Instruction

Function: Converts date-time type data to other types of data. The date is stored internally in seconds, and the time starts from January 1, 1970.

Supported data types: BOOL, BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL, TIME, DATE, TOD, DT, and STRING.

When the output is BOOL: TRUE if the input is not equal to 0. FALSE if the input is equal to 0.

[Example 3.75] DT_TO_<TYPE> example is shown in Table 3-45 below.

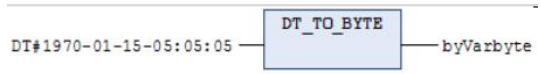
| Conversion Instructions | Graphical language | Textualized Language | Results |
|---|---|---|---|
| DT_TO |  | byVarbyte:= DT_TO_BYTE(DT#197 0-01-15-05:05:05). | 129 |
| |  | sVarstr:=DT_TO_STRI NG(DT#1998-02- 13-05:05:06). | 'DT#1998- 02-13- 05:05:06' |

Table 3-45 Example of DT_TO_<TYPE> conversion ladder instruction

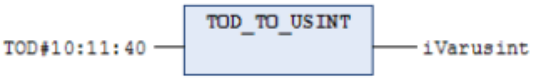
9) TOD_TO_<TYPE> time type conversion command

Function: Converts time-based data to other types of data, and dates are converted internally in milliseconds.

Supported data types: BOOL, BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL, TIME, DATE, TOD, DT, and STRING.

When the output is BOOL: TRUE if the input is not equal to 0. FALSE if the input is equal to 0.

[Example 3.75] TOD_TO_<TYPE> example is shown in Table 3-46 below.

| Conversion Instructions | Graphical language | Textualized Language | Results |
|---|---|---|---|
| |  | iVarusint:=TOD_TO_ USINT(T OD#10:11:40). | 96 |

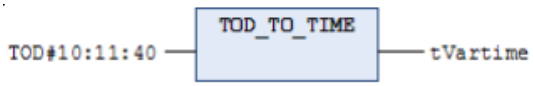
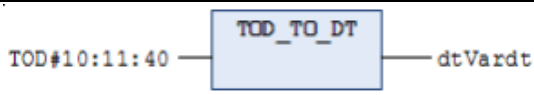
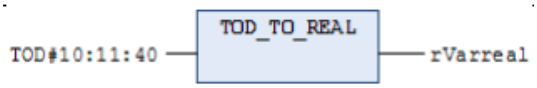
| | | | |
|---|---|---|---|
| TOD_TO |  | tVartime:=TOD_TO_TIME(TOD#10:11:40). | T#611m40s0ms |
| |  | dtVardt:=TOD_TO_DT(TOD#10:11:40). | DT#1970-01-01-10:11:40 |
| |  | rVarreal:=TOD_TO_REAL(TOD#10:11:40) | 3.67e+007 |

Table 3-46 Example of TOD_TO_<TYPE> conversion ladder instruction

10) STRING_TO_<TYPE> character type conversion instruction

Function: Convert a string to other type data, the string type variable must contain a valid target variable value, otherwise the conversion result is 0.

Supported data types: BOOL, BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL, TIME, DATE, TOD, DT, and STRING.

[Example 3.76] Example of STRING_TO_<TYPE> is shown in Table 3-47 below.

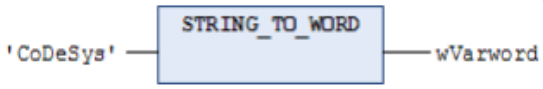
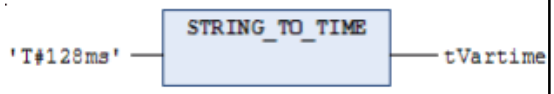
| Conversion Instructions | Graphical language | Textualized Language | Results |
|---|---|---|---|
| STRING_TO |  | wVarword:=STRING_TO_WORD('CoDeSys'). | 0 |
| |  | tVartime:=STRING_TO_TIME('T#128ms'). | T#128ms |

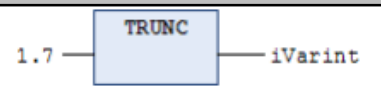
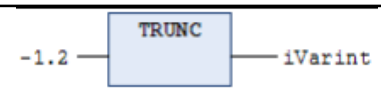
Table 3-47 Example of STRING_TO_<TYPE> conversion ladder instruction

11) rounding TRUNC

Function: Truncate the decimal part of the data and keep only the integer part.

Support data type: input is REAL type, output is INT, WORD, DWORD type.

Example 3.77] Example of TRUNC rounding instruction is shown in Table 3-48 below.

| Conversion Instructions | Graphical language | Textualized Language | Results |
|---|---|---|---|
| TRUNC |  | iVarint: = TRUNC (1.7) | 1 |
| |  | iVarint: = TRUNC (-1.2) | -1 |

Caution:

1. There is a possibility of losing information when going from a larger data type to a smaller data type.
2. This instruction only intercepts the integer part. If you want to round up to the nearest whole number, you can use the REAL_TO_INT instruction.

# 3. 7 Ladder Diagram (LD)/Function Block (FBD)

## 3. 7. 1 Introduction to the ladder/function block diagram programming language

Two graphic programming languages are defined in the IEC 61131-3 standard. They are the LadderDiagram (LD) programming language and the FunctionBlockDiagram (FBD) programming language. The LadderDiagram programming language uses a series of ladder steps to form a ladder diagram that represents the relationship between variables in an industrial control logic system. The Function Block Diagram programming language represents the ontological part of the program organization unit with a series

of connections of function blocks.

1) Ladder diagram (LD)

Ladder diagram originated from the United States, it is based on a graphic representation of the relay logic, is the most widely used in PLC programming a graphical language. The ladder program has two vertical power rails on the left and right sides. The power rail on the left nominally provides energy for the power flow from left to right along the horizontal ladder through each contact, function, function block, coil, etc., and the power flow ends at the power rail on the right. Each contact represents the state of a Boolean variable, each coil represents the state of an actual device, and the function or function block corresponds to a standard library in IEC 1131-3 or to a user-created function or function block.

Ladder diagram is the most widely used programming language in China, it is also one of the three graphic programming languages of IEC1131-3, ladder diagram is the most used traditional PLC graphic programming language, also known as the first programming language of PLC. According to the state and logic relationship of each contact in the ladder diagram, to find out the state of the programming element corresponding to each coil in the diagram is called the logic solution of the ladder diagram.

Some programming components in the ladder diagram follow the name of relays, such as coils, contacts, etc., but they are not real physical relays, but some storage units (soft relays), each soft relay and PLC memory in the image register of a storage unit corresponding to. If the memory unit is "TRUE" state, it means that the corresponding soft relay's coil in the ladder diagram is "energized", its normally open contacts are on, normally closed contacts are off, and this state is called the soft relay's "TRUE " or "ON" state of the soft relay. If the memory cell is in the "FALSE" state, the state of the coil and contacts of the corresponding soft relay is the opposite of the above, and the soft relay is in the "FALSE" or "OFF" state. These "soft relays" are often referred to as programming elements in use.

2) Functional Block Diagram (FBD)

Function block diagrams are used to characterize the behavior of functions, functional blocks, and programs, and to characterize the behavior of steps, actions, and transitions in sequential functional flow diagrams. Function block diagrams are very similar to signal flow diagrams in electronic circuit diagrams, which in a program can be seen as a flow of information between two process elements. Function block diagrams are commonly used in the field of process control.

The function block is represented by a rectangular block, and each function block has no less than one input on the left side and no less than one output on the right side. The type of name of the function block is usually written inside the block, but the name of the function block instance is usually written in the upper part of the block, and the input and output names of the function block are written in the corresponding places of the input and output points inside the block.

3) Program execution order

The execution process of ladder diagram and function block diagram is similar, both are executed in the order of left to right and top to bottom, as in Figure 3.7shown.

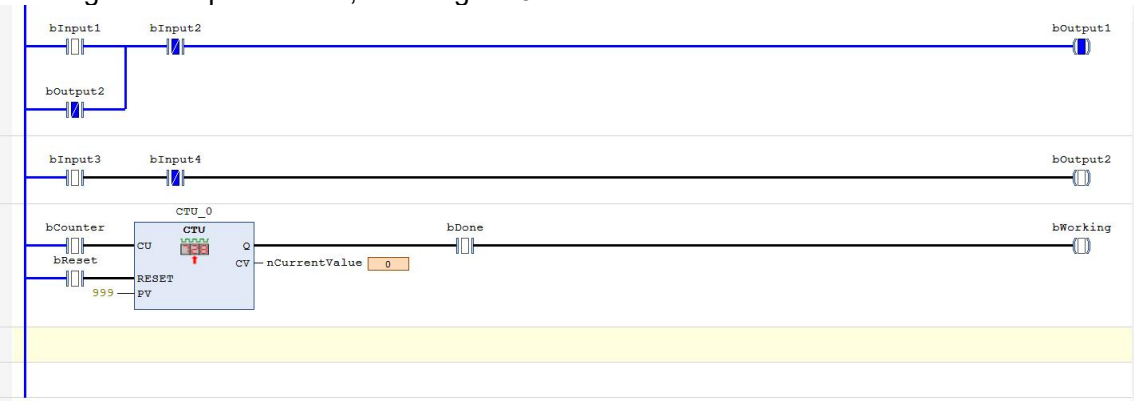



Figure 3.7 Program execution sequence

- **Execution process**

a. busbar

Ladder diagrams use a network structure, and the network of a ladder diagram is bounded by the left bus. When analyzing the logical relationships of ladder diagrams, to use a relay circuit diagram analysis, imagine

that there is a positive left and negative right DC supply voltage between the left and right buses (left and right buses), and that there is "energy flow" from left to right between the buses. The right bus is not shown.

b. section

A section is the smallest unit in the ladder network structure. The network of related logic starting from an input condition and ending at a coil is called a section. In the editor, sections are arranged vertically. In CodeSys, each section is represented by a series of section numbers on the left side, containing input and output instructions, consisting of logical equations, arithmetic expressions, programs, jumps, returns or function block call instructions.

To insert a section, you can use the command Insert section or drag it from the toolbox. The elements contained in a section can all using the drag and drop in the editor to copy or move.

When executing the ladder diagram, start from the section with the smallest label, determine the state of each element from left to right, and determine its state of the right connection element. The elements is executed one by one to the right, and the result of the operation execution is output by the execution control element. Then the execution process of the next section is carried out. Figure 3.7 shows the execution process of the ladder diagram.

c. Energy flow diagram

The bold blue line on the left in 4.7 is the energy flow, which can be interpreted as a hypothetical "conceptual current" or "PowerFlow" flowing from left to right, in the same direction as the logical operations performed in the user program. Energy flow can only flow from left to right. Using the concept of energy flow can help us better understand and analyze ladder diagrams.

d. branch

When there are branches in the ladder diagram, the state of each graphical element is also analyzed according to the top-to-bottom, left-to-right order of execution. For vertically connected elements the state of their right connected elements is determined according to the above-mentioned regulations, so that the evaluation process is executed one by one from left to right and from top to bottom. In ladder diagrams, the solving of values without feedback paths is not very clear. All its external input values with these relevant contacts must be evaluated before each ladder stage.

- **Execution control**

a. Jump and return

When the jump condition is met, the program jumps to the section marked by Label and starts execution until the execution of that part of the program reaches RETURN, when it returns to the original section and continues execution. The structure diagram is shown in 3.8.

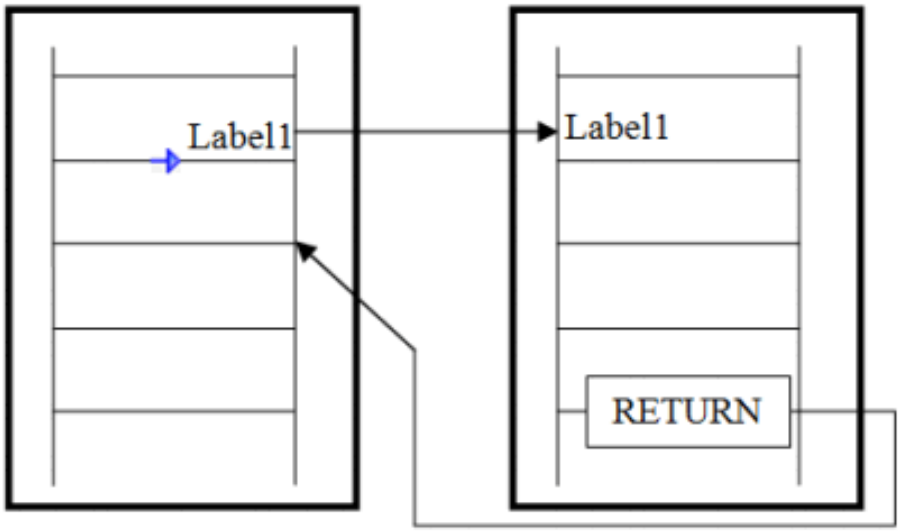



Figure 3.8 Jump instruction execution process

When the program executes to Label1 on the left side of Figure 3.8, the program starts to execute a jump and jumps directly to the right side of Figure 3.8, finds the segment labeled with Label1 and starts to execute the next program until the program runs to RETURN, then the execution of the jump program is completed and returns to the main program loop on the left side of the figure.

The jump and return instructions in CodeSys using the ladder are as follows, as shown in Example 3.18.

[Example 3.18] Example of executing a program using a jump instruction.

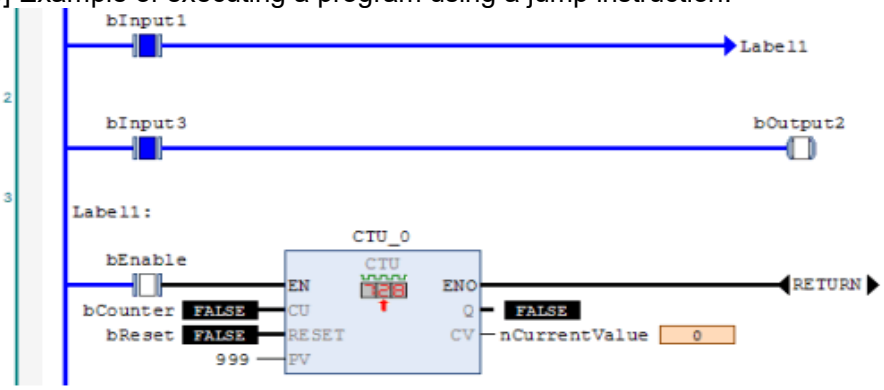



Figure 3.9 Jump instruction execution

As shown in Figure 3.9, when bInput1 is set to TRUE, the main program executes the jump statement and according to label Label1, the program jumps to the Label1 segment in Section 3. It is easy to see from Figure 3.9 that even though bInput3 in Section 2 is set to ON, bOutput2 is never set to TRUE because the program skips the statement directly. bOutput2 is only TRUE when bInput1 is FALSE and bInput3 is TRUE.

## 3. 7. 2 Connecting elements

The ladder diagram language in IEC1131-3 is a reasonable absorption and reference to the ladder diagram language of each PLC manufacturer, and each graphic symbol in the language is basically the same as that of each PLC manufacturer, Figure 3.10 shows the ladder diagram editor view. the main graphic symbols of IEC61131-3 include:

a. Basic connection class: power cabinet first, connecting elements.
b. ①Contact category: Normally open contact, normally closed contact, positive conversion readout contact, negative conversion contact.
c. ②Coils: general coils, inverse coils, set (latch) coils, reset de-lock coils, hold coils, set hold coils
d. Coil, reset hold coil, positive changeover readout coil, negative changeover readout coil.
e. ③Functions and function blocks: Include standard functions and function blocks as well as user-defined function blocks.

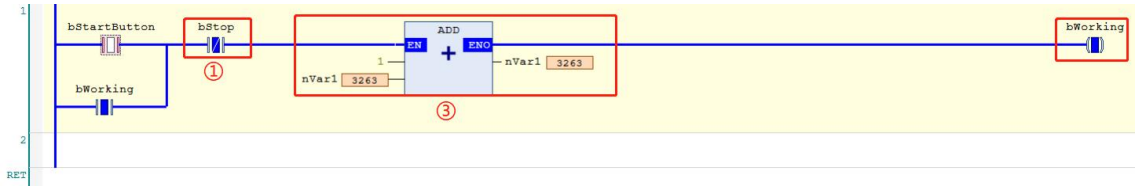



Figure 3.10 Ladder Diagram Editor

- **Line element**

1) Power rail line (bus bar)

The graphical element of the ladder power rail (PowerRail) is also called the bus bar. Its graphical representation is located on the left side of the ladder diagram and can also be

It is called the left power bus. Figure 3.11 shows a graphical representation of the left busbar.

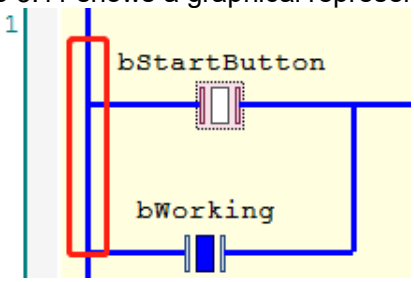



Figure 3.11 Graphical representation of the left bus bar

2) Connection cable

In a ladder diagram, the graphical symbols are connected by connecting lines, and the graphical symbols of connecting lines are horizontal and vertical lines, which are the most basic elements forming the ladder diagram. Figure 3.12's a) and b) are the graphic representations of horizontal and vertical connecting lines.

a ) b )

a) Horizontal connection line b) Vertical connection line

Figure 3.12 Graphical representation of connection lines

3) Rules for passing connected elements:

a. The state of the connected elements is passed from left to right, enabling the flow of energy. The transfer of state observes the following rules:

b. The connection element connected to the left power rail line, whose state is TRUE at any moment, which indicates that the left power rail line is starting point of the energy flow. The right power rail line is like the zero potential in the electrical diagram.

c. The horizontally connected element is represented by a horizontal line, and the horizontally connected element passes the state of the graphic element to the graphic element to its right.A vertically connected element is always connected to one or more horizontally connected elements, i.e., it is joined by one or more horizontally connected elements in each side is composed of intersecting with the vertical line. The state of the vertically connected element is based on the state of each left-hand horizontally connected element connected to itor arithmetic representation.

Therefore, the state of vertically connected elements is determined according to the following rules:

a. If the state of all connected horizontally connected elements on the left is FALSE, the state of this vertically connected element is FALSE.

b. If the status of one or more horizontally connected elements on the left is TRUE, the status of that vertically connected element is TRUE.

c. The state of a vertically connected element is passed to all horizontally connected elements connected to it, but not to the left of itelement of all horizontally connected elements.

[Example 3.19] Example of connecting elements and state passing.

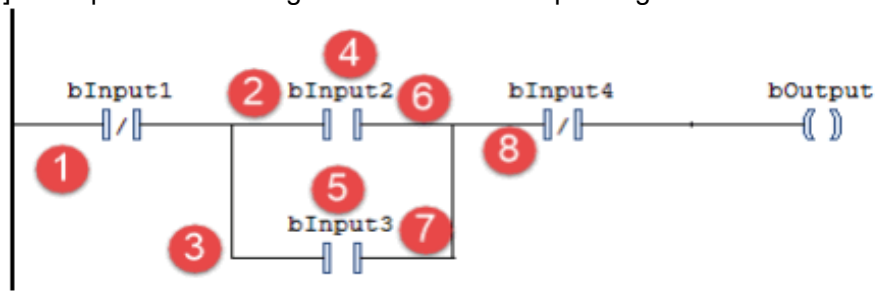



Figure 3.13 Example of connected elements and states

Figure 3-13 shows an example of connected elements and their states. Connection element 1 is connected to the left power rail line and its status is TRUE.

a. Connected element 2 is connected to connected element 1 and its state is passed from connected element 1 and, therefore, its state is TRUE.

b. Connected element 3 is the vertically connected element which is connected to the horizontally connected element 1 with the status TRUE.

c. Connection elements 2 and 3 are passed 4, 5, respectively, and since the variables bInput2 and bInput3 corresponding to graphical elements 4 and 5 are normally open contacts, respectively, the states of connection elements 6 and 7 become TRUE by the passing of the graphical elements.

Since all the states to the left of connected element 8 are TRUE, the state of connected element 8 is TRUE.

d. The input and output data types of the connection elements must be the same. In the standard, the data types of graphical elements such as contacts and coils are not limited to Boolean types. Therefore, connecting elements with the same input and output data types is necessary to ensure correct state transfer.

- **Section**

Sections are the basic entities of LD and FBD. In the LD/FBD editor, sections are arranged in numerical order. Each section starts with a section number on the left side and has a structure consisting of logical or

arithmetic expressions, procedures, functions, function block calls, and jump or return instructions. The schematic diagram of the sections is shown in Figure 3.14 shaded in red with the sequential numbering.


Figure 3.14 Section view

1) Section comment

A section can also be assigned a title, a comment, and a marker. The availability of the title and comment fields can be turned on or off via the Tools-->Options-->FBD, LD, and IL Editors dialog box, as shown in Figure 3.15. This is shown in Figure 3.15.

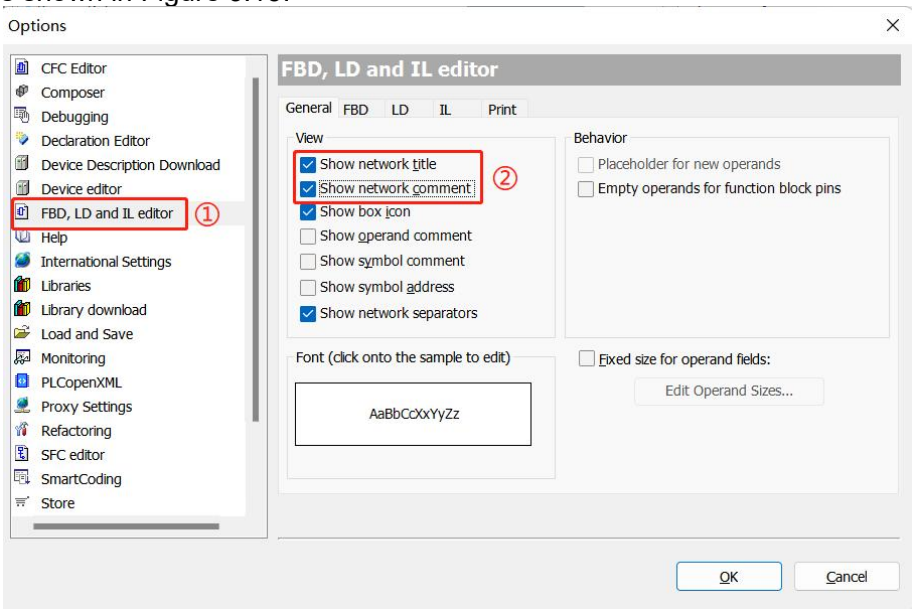



Figure 3.15 Section header comments and marker functions

If the above option is enabled, the user can open an edit area for the header by clicking with the mouse below the upper boundary of the section. Entering a comment opens the corresponding edit area under the header area. Comments can be multi-line. Line feeds can be achieved by using the Enter key, and comment text entry is terminated by [Ctrl]+[Enter]. Figure 3.16 shows the view of section header comments and section notes.

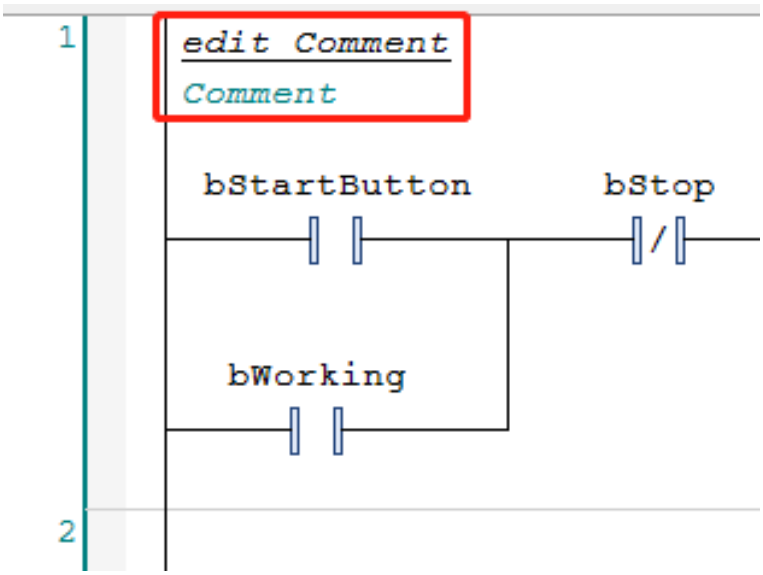



Figure 3.16 Section heading notes and section notes

2) Section title comment

The section can also be set to "comment state" by "Toggle section comment state", after which the section will be displayed as a comment and will not be execution.

3) Section branching

Create "subsections" by inserting " branch " in the toolbox, as in Figure 3.17 where the branching function is used.

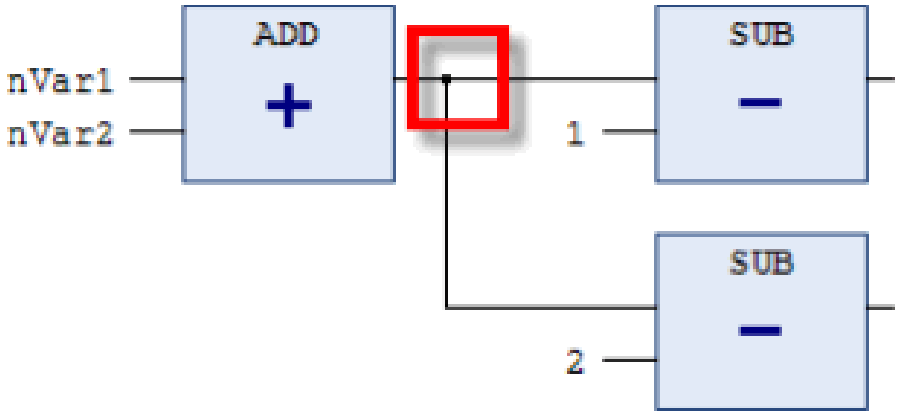



Figure 3.17 Creating subsections by branching function

- **Label**

A tag is an optional identifier and can be addressed when a jump is defined. It can contain any character.

Under the section area, each FBD, LD or IL section has a text input area to define a marker. The marker is an optional identifier for the section that can be addressed when defining a jump, and it can contain characters in any order.

1) Use in FBD

Right-click in the margin of the section and select "Insert Label", as shown in Figure 3.18 in 1, then Label: will pop up in 2, and the user can edit it.

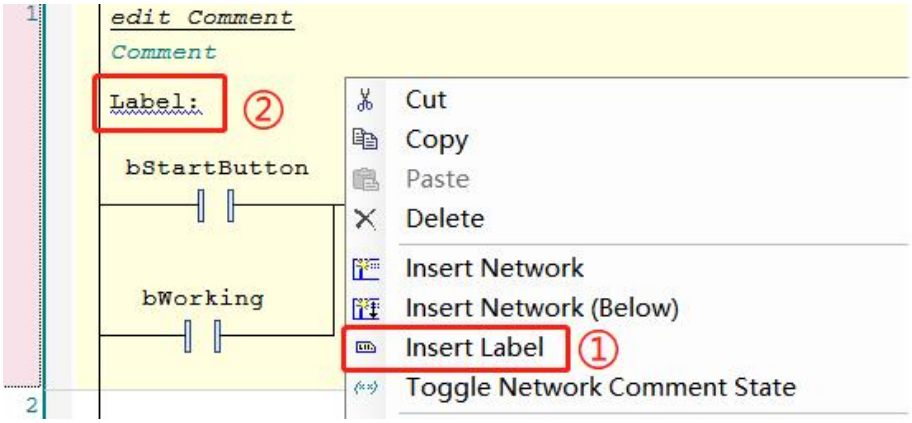



Figure 3.18 Adding a marker to a section

- **Contacts**

1) Contact type

Contacts are the graphical elements of a ladder diagram. The contact (Contact) of a ladder diagram follows the contact terminology of electrical logic diagrams and is used to represent a change of state of a Boolean variable. A contact is a ladder diagram element that passes a state to the horizontally connected element to its right.

The contacts can be divided into NormallyOpenContact (NO) and NormallyClosedContact (NC). Normally open contact means under normal operating conditions, the contact is open, and its state is FALSE. normally closed contact means under normal operating conditions, the contact is closed, and its state is True. table 3-8 lists the common contact graphic symbols and descriptions in CodeSys ladder diagram.

| Type | Graphical symbols | Description |
|---|---|---|
| Normally open contacts | | If the contact corresponds to the current Boolean variable value of True, the contact is sucked, and if the state of the connected element on the left side of the contact is True, the state TRUE is passed to the right side of the contact, making the state of the connected element on the right side True. conversely, when the Boolean variable value is False, the state of the connected element on the right side is FALSE. |
| Normally closed contacts | | If this contact corresponds to the current Boolean variable value of False, the normally closed contact is in the sucking state, and if the state of the connected element on the left side of the contact is True, the state True is passed to the right side of the contact, making the state of the connected element on the right side True. conversely, when the Boolean variable value is True, the contact is broken, and the state of the connected element on the right side is False. |
| Insert right contact | | Multiple contacts in series can be made, and the contact is inserted on the right side. The last contact can transmit True only when multiple contacts connected in series are all in the engaged state. |
| Inserted under parallel often Open contact | | Parallel connection of multiple contacts is possible, and normally open contacts are inserted in parallel on the lower side of the contacts. Of the two parallel contacts<br>Only one contact needs to be True, then the parallel line transmits True. |
| Inserted under parallel often Closed contacts | | Parallel connection of multiple contacts is possible, and normally closed contacts are inserted in parallel on the lower side of the contact. The normally closed contact is closed by default. If the contact corresponds to the current Boolean variable value of False, when the state of the connected element on the left side is True, the parallel contact transmits True on the right side. |

| Type | Graphical symbols | Description |
|---|---|---|
| Inserted in parallel on often Open Contact |  | Parallel connection of multiple contacts is possible, and normally open contacts are inserted in parallel on the upper side of the contacts. Only one of the two parallel contacts needs to be True, then the parallel line transmits True. |

Table 3-8 Graphical symbols and descriptions of contact elements

2) State transfer rules

According to the state of the contact and the state of the left connection element to which the contact is connected, the following rules determine the graphic symbol on its right sideStatus.

a. When the state of the graphical element on the left side of the contact is TRUE, only then can its state be passed to the graphical element on the right side of the exit point, according to the following principle for transmission:

❖ If the state of the contact is TRUE, the state of the graphical element to the right of the contact is TRUE.

❖ If the state of the contact is FALSE, the state of the graphical element to the right of the contact is FALSE

b. When the state of the graphical element on the left side of the contact is FALSE, the state of the contact cannot be passed to the right graphic element of the contact, the status of the right graphic element is FALSE.

c. from FALSE-->TRUE in the contact left side of the left side of the graph changes, its relevant variables also from FALSE-->TRUE, then the

The right graphical state of the contact goes from FALSE --> TRUE and remains for one cycle before changing to FALSE, then this is called rising edge trigger.

d. from TRUE --> FALSE in the contact left side of the graph left change, its relevant variables also from TRUE --> FALSE, then the right side of the contact graph state from TRUE --> FALSE, and hold a cycle, and then change to TRUE, then this is called falling edge trigger.

- **Coil**

1) Coil type

Coils are the graphical elements of ladder diagrams. Coils in ladder diagrams follow the coil terminology used in electrical logic diagrams to represent Boolean-type variables of state change.

According to the different characteristics of the coils, they can be divided into instantaneous coils and latching coils, and latching coils are divided into setting coils and resetting coils. Table 4-9 lists the common coil graphic symbols and descriptions in CodeSys ladder diagrams.

| Type | Graphical symbols | Description |
|---|---|---|
| Coils |  | The state of the left connected element is passed to the relevant Boolean variable and the right connected element. If the state of the left connected element of the coil is TRUE, the Boolean variable of the coil is TRUE, and vice versa the coil is FALSE. |
| Positioning coil |  | There is an S in the coil. When the state of the left connected element is TRUE, the Boolean variable for that coil is set and held until reset by Reset of the coil. |
| Reset coil |  | The coil has an R. When the state of the left connected element is TRUE, the Boolean variable of this coil is reset and held until set by Set of the coil. |

Table 3-9 Graphical symbols and descriptions of coil elements

2) Coil transfer rules

A coil is a ladder element that passes the state of the left-hand horizontally or vertically connected element unchanged to its right-hand horizontally connected element. During the transfer, the state of the relevant variables and direct addresses of the left-hand connection are stored in the appropriate Boolean variables. Conversely, an inverted coil is a ladder element whose left-hand horizontally or vertically connected element's state is inverted and passed to the ladder element passed to its right-hand horizontally connected element.

The place and reset coils hold the state of the left side horizontally connected element from FALSE to

TRUE and from TRUE to FALSE for one seek cycle at the instant and pass its left side horizontally connected element state to the right side horizontally connected element at other times.

The rising-edge and falling-edge coils hold the variables of the coils in question for one seek cycle at the moments when the state of their left-hand horizontal connection element goes from FALSE to TRUE and from TRUE to FALSE and pass the state of their left-hand horizontal connection element to their right-hand connection horizontal element at other times.

It is not formulated that only one element can be linked on the right side, so the user can expand on the right side for the purpose of simplifying the program. For example, other coils can be linked in parallel on the right side, as shown in Example 3.20.

[Example 3.20] Transfer of coil state.

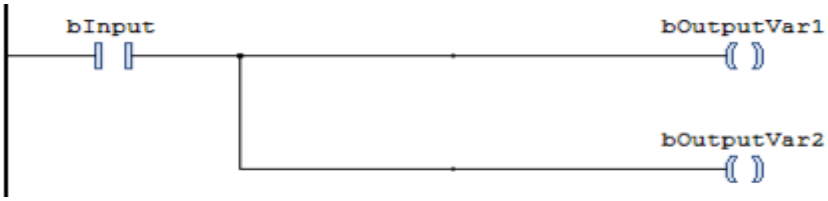



Figure 3.19 Transfer of coil states

Figure 3.19 shows the transfer process of the coil state. In the figure, when the contact bInput is closed, the state of the connection element to the right of its contact is also TRUE, and it is connected to the coils bOutputVar1 and bOutputVar2 after the horizontal and vertical connection elements, respectively, and its state is also set to TRUE.

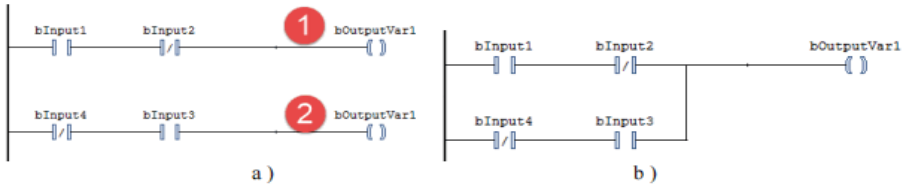
3) Dual Coil

A humbucker is when the same coil is used twice or more in a user program, a phenomenon known as humbucker output.

Figure 3.20 (a) has two coils with output variable "bOutputVar1". In the same scan cycle, the logic operation results of the two coils may be opposite, i.e., one coil of variable bOutputVar1 may be "energized " and the other one may be "powered off". For the variable bOutputVar1 control, it is the state of the last bOutputVar1 coil that really makes a difference.

The on/off state of the coil of bOutputVar1, in addition to acting on the external load, may also, through its contacts, influence the program.

The state of the variables of the Therefore, the phenomenon of humbucker output should be generally avoided, and the humbucker problem should be solved by using the parallel connection adopted in Figure 3.20 (b) as much as possible.




a) Dual Coil b) Avoid Dual Coil

Figure 3.20 Humbucker example

Such a dual coil output is allowed if only the logic operation corresponding to one of the coils can be executed in the same scan cycle The Humbucker output is allowed in the following 3 cases:

a. In two program segments with opposite judgment conditions (e.g., an automatic program and a manual program), a dual coil output is allowed. Coils of the same variable can appear once in each of the two program segments. The PLC executes only one coil output instruction of the dual coil element in the program segment being processed.

b. In two subroutines with opposite call conditions (e.g., automatic, and manual programs), the phenomenon of double coils is allowed. The same variable coil can appear once in each of the two subroutines. Instructions in a subroutine are executed only when that subroutine is called, and not when it is not called.

c. To avoid double-coiled outputs, the set/reset instruction can be used multiple times for the same variable.

- **function and function block calls**

If you want to implement the function or function block calls will use the operation block, the operation block can represent all the POU, including the function block, the functions and even programs are included. Function blocks such as timers, counters, etc. can be inserted in sections of FBD, LD. The operation blocks can have any input and any output. The graphical symbols of functions and function blocks are described in detail in Table 3-10.

Together with the contacts and coils, the user can also insert function blocks and programs. In the network, they must have an input with a Boolean valuein and an output and can be used in the same position as a contact, i.e., on the left side of the LD network.

| Type | Graphical symbols | Description |
|---|---|---|
| Fast insertion operation | | Insert a function or function block and select the function and function block you want to use by mouse according to the pop-up dialog box. Suitable for those who are not familiar with functions and function blocks. |
| Insertion of air transport is considered fast | | Insert a rectangular block directly in the "???" where you can directly enter the function or function block name, for users who are familiar with functions and function blocks. |
| Insert with EN/ENO The computing is fast | | Executes a function or function block and allows state to be passed downstream only when EN is True. Suitable for use by those less familiar with functions and function blocks. |
| Insert with EN/ENO The air transport is considered fast | | Insert rectangular block with EN/ENO and enter the function or function block name directly in the "???" where the function or function block name is entered directly. Only when EN is True, the function or function block is executed, and the state is allowed to be passed downstream. Suitable for users who are familiar with functions and function blocks. |

Table 3-10 Function and function block graphic symbols and descriptions

The ladder programming language supports function and function block calls. When function and function block calls are made, the following matters need to be noted:

1) In a ladder diagram, functions and function blocks are represented by a rectangular box. Functions can have multiple input parameters but only one return parameters. Function blocks can have multiple input parameters and multiple output parameters.

2) The input column is on the left side of the rectangular box, and the output column is on the right side of the rectangular box.

3) The names of the function and function blocks are displayed in the upper middle of the box, and the function blocks need to be instantiated, and the instances are listed in the upper-middle part of the box outside. Use the instance name of the function block as its unique identifier in the project.

4) In order to ensure that the energy flow can be passed through the function or function block, each called function or function block should have at least one input and output parameter. For the connected function block to execute, at least one Boolean input should be connected to the vertical left power rail line via the horizontal ladder.

5) When the function block is called, the actual parameter value can be filled in directly at the external connection line of the function block for that internal form parameter variable name.

[Example 3.21] Function block calls the setting of real parameters.

Figure 3.21 calls the TON delay ON function block, TON_1 is the instance name of the function block TON after instantiation. The input form parameter PT of the function block is set to t#5s. The output form parameter Q and ET, when the output form parameter such as ET in the example is not needed, the variables can be left unconnected.

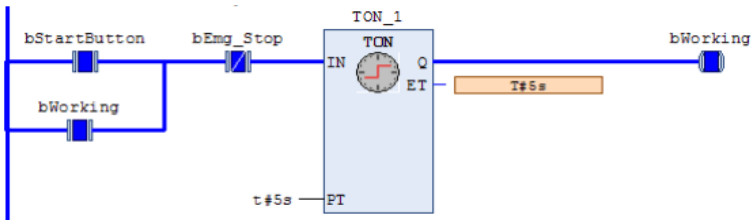


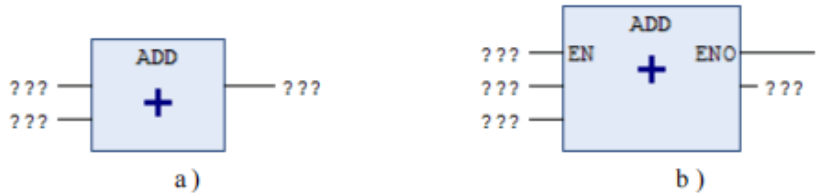

Figure 3.21 Setting of real parameters when calling function blocks

As you can see, the output Q of function block TON is connected to the coil bWorking. indicates that bWorking is True when the contact bStartButton is True and bEmg_Stop is False for more than 5s. bWorking is False when bEmg_Stop is disconnected, i.e., True.

6) If there are no dedicated input and output parameters for EN and ENO, the functions and function blocks are executed automatically, and the status is passed downstream. In Example 3.22, the function block

with EN and ENO is called.

In the toolbox you can choose to insert a standard operator block " " or insert a function block with EN/ENO " ". Copy or move by drag and drop in the editor. Figures 3.22 in a) and b) show a schematic comparison of a standard operator block and an operator block with EN/ENO.




a)Standard arithmetic block b) Function block with EN/ENO

Figure 3.22 Comparison of the two operation blocks in FBD

In Figure 3.22, a) the function block is executed directly if the front-end conditions are met, while in b) the function block is only executed when EN is TRUE; otherwise, the function block will not be executed by the program even if all the front-end conditions are satisfied. If the input signal of EN in (b) is set to the constant "TRUE", the functions of (a) and (b) are the same.

[Example 3.22] Calling a function block with EN and ENO.

Figure 3.23 shows a function block with EN and ENO. The Boolean input bEnable is used for the start of the counter function block CTU_0 and bWorking is used as a status variable signal for this function block to be enabled.

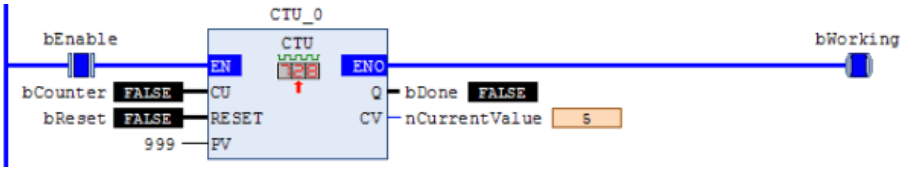



Figure 3.23 Calling a function block with EN and ENO

When bCounter has a rising edge trigger signal, the shape reference output variable CV is calculated by adding 1.

- ❖ When EN is False, the operation defined by the function block ontology is not executed and the value of ENO is False accordingly.
- ❖ When the value of ENO is True, it means that the function block is being executed.

7) Distribution

The allocation function can be interpreted as the assignment of input/output to the arithmetic block. In the toolbox you can select the insert "" tool, drag and drop it into the editing area of the program. At this time the budget block in the editing area corresponding to the input and output interface will appear at the small gray diamond pattern, the reader can directly drag and drop it to the interface. After the insertion, the text string "????" can be replaced by the name of the variable to be assigned, or you can use the button to call "Input Assistant", and then the assignment of the variables to the input/output interface of the computing block is completed. The assignment view is shown in Figure 3.24.

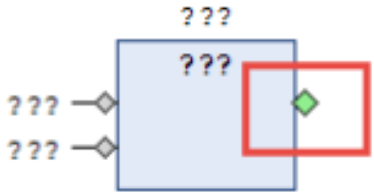



Figure 3.24 Distribution view

- **Jump execution**

1) Jump execution control element

The Jump execution control element is represented by a Boolean signal line terminating with a double arrow. The jump signal line starts with a Boolean variable, the Boolean output of a function or function block, or the energy flow line of a ladder diagram. Jumps are divided into conditional and unconditional jumps.

When a jump signal starts with a Boolean variable, function or function block output, the jump is a conditional jump. The jump occurs only when the program control executes to the jump signal line for a specific network marker and that Boolean value is TRUE.

An unconditional jump is unconditional if the jump signal line starts at the left power rail line of the ladder diagram. In the function block diagram programming language, the jump is also unconditional if it starts at the Boolean constant 1. The jump control element graphics are listed in Table 3-11.

| Execution Control Type | | Graphical symbols for execution control elements | Description |
|---|---|---|---|
| Unconditional Jump | LD Language | TRUE → Label | Jump directly and unconditionally to Label |
| | FBD Language | → Label | |
| Conditional Jump | LD Language | bInput → Label | Condition jumps to Label when bInput is 1 |
| | FBD Language | bInput → Label | |
| Article Jump Return | LD Language | bInput → RETURN | Conditional jump returns when bInput is 1 |
| | FBD Language | bInput → RETURN | |

2) Jump to the target

In a program organization unit, a jump target is a marker within that program organization unit where the jump occurs. It indicates that the program will start execution from that target after the jump occurs.

3) Jump back

Jump return (Return) is divided into two categories: conditional jump return and unconditional jump return.

Applicable to conditional return from functions, function blocks, when the Boolean input returned by the conditional jump is TRUE, program execution will jump and returns to the calling entity. When the Boolean input is FALSE, program execution will continue in the normal way.

The unconditional jump return is provided by the physical end of the function or function block. As shown in Table 3-11, linking the RETURN statement directly to the left track line indicates an unconditional return.

4) Jump to the execution of the configuration

Insert "  " in the toolbox, after inserting, the automatically entered "????", the marker of the jump target is replaced.

You can enter the target's mark directly or use the input assistant to select it by clicking "  " browse key, as shown in Figure 3.25, the system will automatically filter the available markers for users to choose.

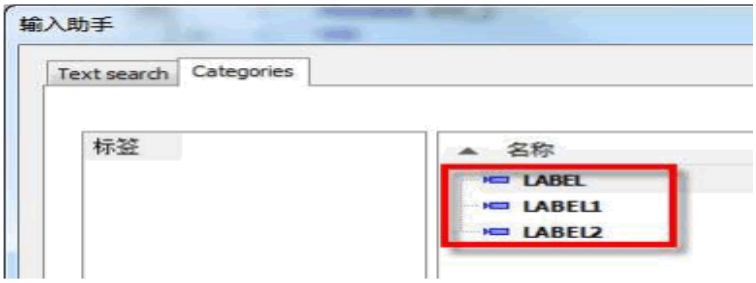



Figure 3.25 Jump to input assistant

[Example 3.23] Example of jump statement.

In the cylinder control, the outgoing signal of the control cylinder solenoid valve is bExtrent. If no feedback signal bExtrented_Sensor1 is received from the outgoing sensor within 5s after the outgoing signal bExtrent is sent, the alarm program Alarm will be jumped to, and the variable declaration and program are as follows.

```
PROGRAMPLC_PRG
VAR
    bExtrent:BOOL.
    bExtrented_Sensor1:BOOL.
    fb_TON:ton.
END_VAR
```

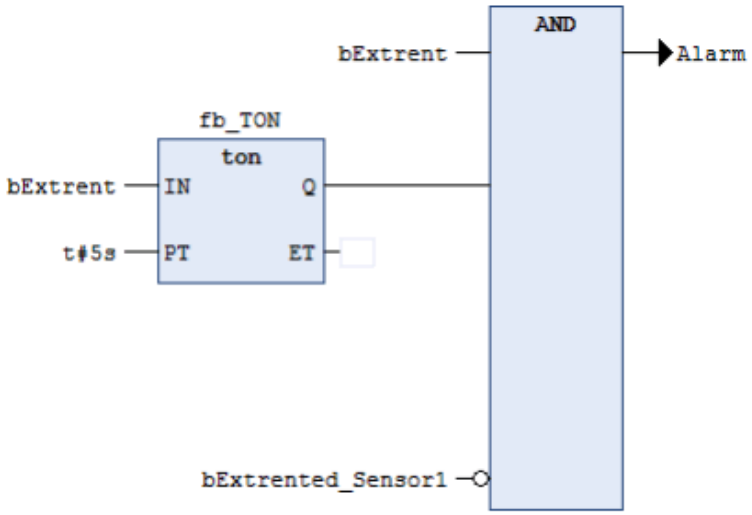



Figure 3.26 Example program of jump statement

Figure 3.26 shows the sample program of the jump statement. Eventually, when the output signal Q of the fb_TON function block and the bExtrent signal are satisfied at the same time, the output signal Alarm with the logic is set to TRUE.

## 3. 7. 3 Application examples

[Example 3.24] Blinking signal light.

Control requirements

A signal blinking light system is formed using a timer and a logic function. The line output can make the signal light ON and OFF at a certain period.

Programming

The program implements the alternation of bLamp and bLamp1

ON and OFF to achieve the control requirements of the flashing beacon. The program is shown in Figure 3.27

The ladder diagram shown is implemented.

The user can use t_SetValue to set the ON and OFF switching time, which is set to 500 milliseconds as follows, with the specific variation

The quantities are defined as shown below:

```
PROGRAMPLC_PRG
VAR
    fb_TON:ton;//TimeDelay
    t_SetValue:TIME:=t#500ms.          //SetTime
    bLampAT%QX0.0:BOOL.        //Output0
    bLamp1AT%QX0.1:BOOL.       //Output1
END_VAR
```

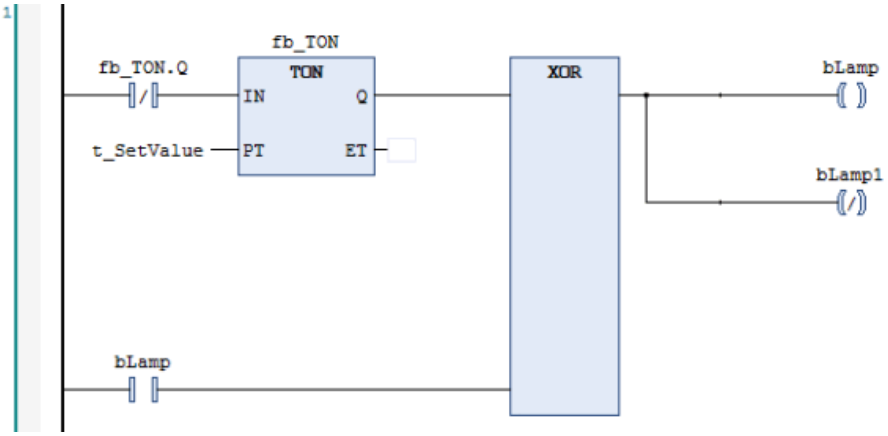



Figure 3.27 Ladder program for flashing signal light

The output effect is shown in Figure 3.28. The output curves of bLamp and bLamp1 are exactly opposite, and the time of their state switching is exactly 1s.

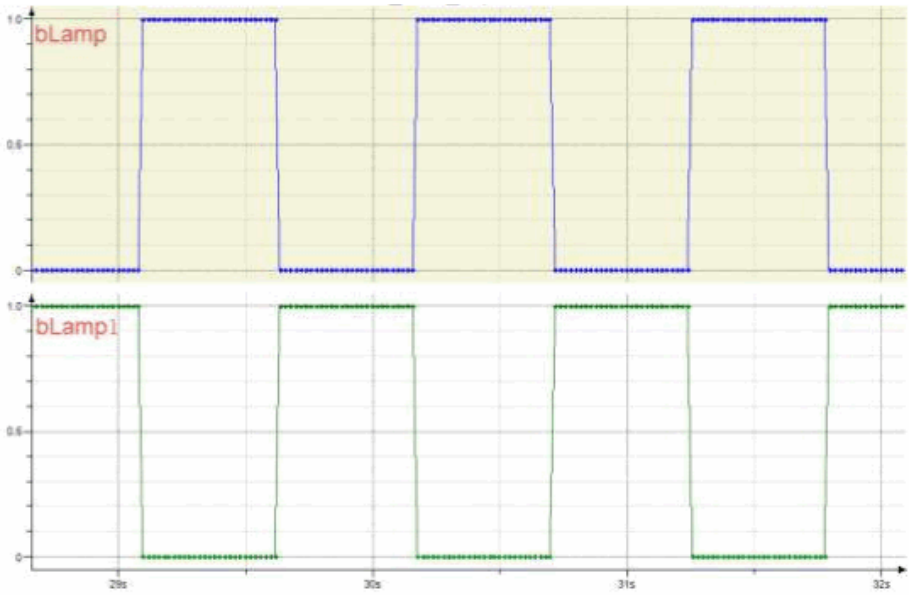



Figure 3.28 Output graph of flashing beacon

[Example 3.25] pH control system.

1) Control requirements

a. In the process of wastewater treatment or fermentation, pH control is often required. pH control system has non-linearity and time lag of the controlled object, so non-linearity and time lag compensation control scheme are commonly used. However, the following control strategy can also be used in simple control schemes: when the pH measurement exceeds the set acidity value, wait for a certain time, and then add alkaline liquid for a certain time. When the pH value exceeds the set value, the contact PHH closes, and vice versa, when it is less than the set value, the alkaline valve is bValves1. The control scheme is "look and adjust".

b. When the pH control is in the linear region, it can be assumed that the change of pH in the control process shows linearity characteristics. The addition of alkaline solution or acid solution to neutralization is performed, the pH change is linear. Usually, the linearity holds when the upper set value and the lower set value are small.

c. Set the time required from pH change to during fermentation as t, and the time from pH change to after alkali addition as t2, then the delay time can be set as t1=t/2, and the time for alkali addition control valve to open as t2.

d. The actual pH control is set at SP=()/2. Reducing the difference between and helps to improve the control accuracy.

e. The start condition of the alkaline addition valve bValves1 is the arrival of the set time of timer t1, therefore, the program uses t1.Q as the start conditionThe stop condition for the alkaline addition valve bValves1 is the set time to t2, so the program uses t2.Q as the stop condition.

f. The start condition of timer t1 is that the pH reaches the set value SP, therefore, the rising edge of contact PHH is used to trigger the fb_Trigger function block and its signal is temporarily stored with the RS function block, and the start condition of timer t2 is that the timing of timer t1 is reached.

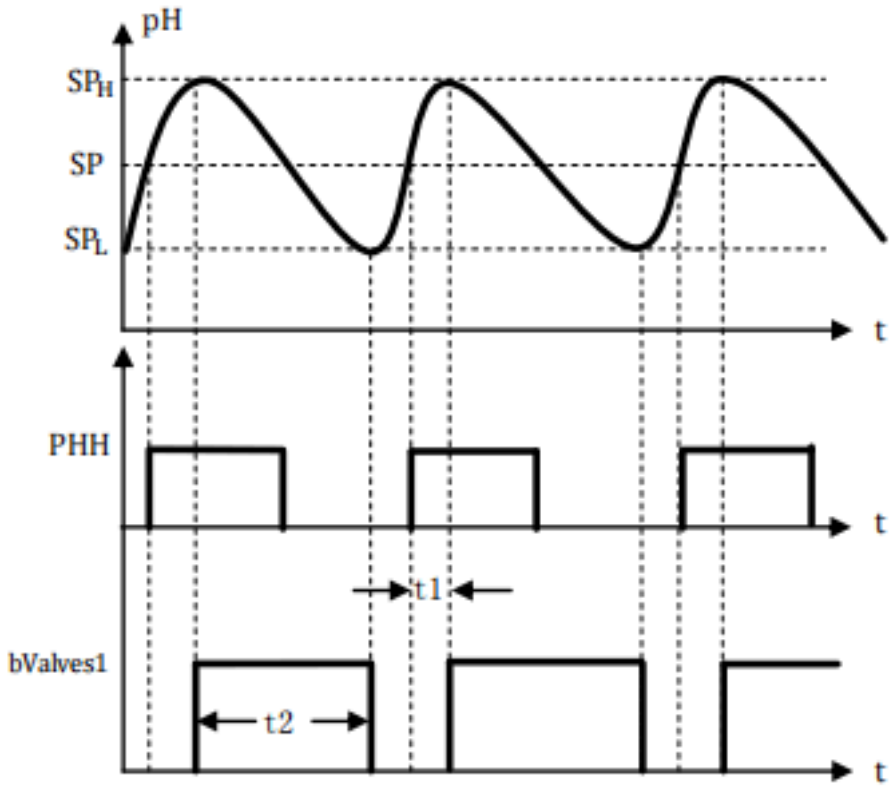



Figure 3.29 pH control signal wavefor

2 ) Programming

According to the above control requirements, the pH control program is written using the ladder programming language, and its variable declaration and program are shown in Figure 3.30 shown. Two separate timers are used in the program.

```
PROGRAMPLC_PRG
VAR
    t1,t2:ton;//timer t1, t2
    PHH:BOOL;//Exceed set value signal
    bValves1AT%QX0.0:BOOL;//add lye valve
    fb_R_Trig:R_Trig.
    fb_RS_0,fb_RS_1:RS.
END_VAR
```

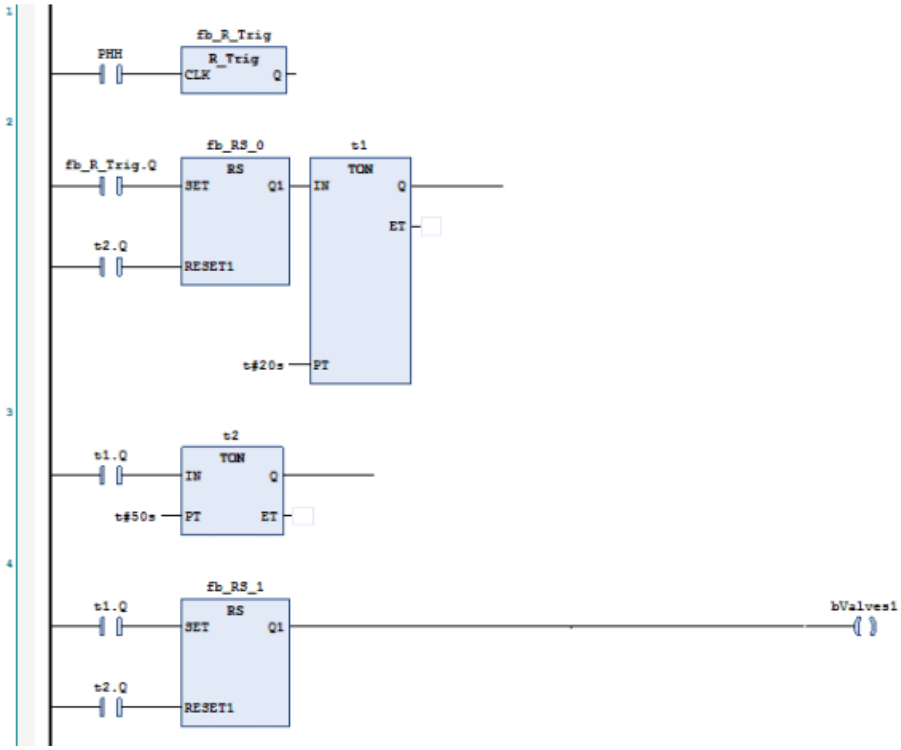



Figure 3.30 pH control ladder program

In the above figure, the program sets the time of timer t1 to 20s and the time of timer t2 to 50s. When the time of t1 arrives, it immediately triggers the t2 timer and opens the alkaline valve to reduce the acidity, and closes the alkaline valve when the t2 timing time arrives.

# 3. 8 Structured Text (ST)

## 3. 8. 1 Introduction to Structured Text Programming Languages

### 3. 8. 1. 1 Introduction

Structured Text (ST) is a high-level text language that can be used to describe the behavior of functions, functional blocks, and programs, and to describe the behavior of steps, actions, and transitions in sequential functional flowcharts.

Structured text programming language is a high-level language, like Pascal, a language developed especially for industrial control applications and one of the most used in CodeSys. For those familiar with advanced computer language development, structured text language is even easier to learn and use, enabling functions such as selection, iteration, and jump statements. In addition, structured text languages are easy to read and understand, especially when annotated with identifiers and annotations that have real meaning. In complex control systems, structured text can greatly reduce their code volume and make complex system problems simple, with the disadvantage that debugging is not intuitive, and compilation is relatively slow. A view of structured text is shown in Figure 3.31.

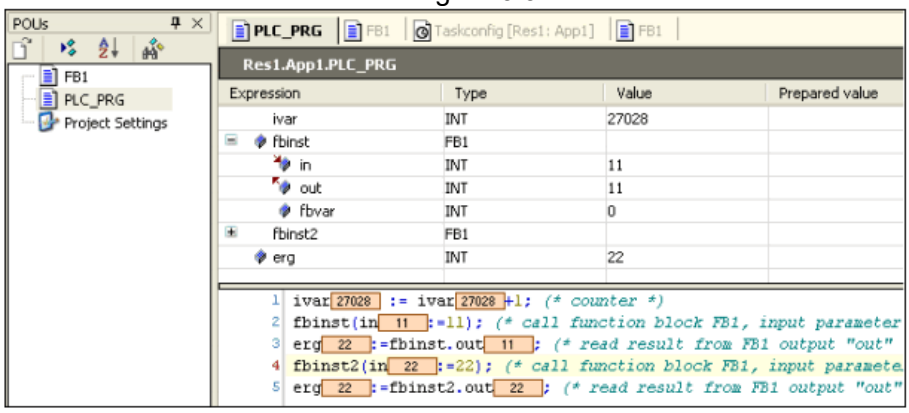



Figure 3.31 Structured text view

### 3. 8. 1. 2 Program execution sequence

The order of program execution using structured text is based on the "line number", starting from top to bottom, as shown in Figure 3.32.

At the beginning of each cycle, the program line with the smaller line number is executed first.

```
1  MC_Jog_Axis_1(
2      Axis:= Axis_1,
3      JogForward:= HMI_gbJogAxis_1_Fwd,
4      JogBackward:= HMI_gbJogAxis_1_Bwd,
5      Velocity:= HMI_rJogAxis_1_Vel,
6      Acceleration:= MC_Jog_Axis_1.Velocity*10,
7      Deceleration:= MC_Jog_Axis_1.Velocity*10,
8      Jerk:= MC_Jog_Axis_1.Velocity*100,
9      Busy=> ,
10     CommandAborted=> ,
11     Error=> ,
12     ErrorId=> );
13
```

Figure 3.32 Structured text program execution sequence

### 3. 8. 1. 3 Expression execution order

An expression includes an operator and an operand. The operand operates according to the rules specified by the operator, gets the result, and returns it. Operands can be variables, constants, register addresses, functions, etc.

[Example 3.26] Expression example.

a+b+c.
3.14*R*R.
ABS(-10)+var1.

If there are several operators in an expression, the operators are executed in the agreed priority order: the operators with higher priority are executed first, and then the operators with lower priority are executed sequentially. If there are operators with the same priority in the expression, they are executed from left to right in the written order. The operator priorities are shown in Table 3-12

| Operators | Symbols | Priority |
|---|---|---|
| Parentheses | () | Highest |
| Function calls | Functionname(Parameterlist) |  |
| power search | EXPT | |
| fetch the opposite | NOT | |
| Multiplication<br>Division<br>Mold pickup | *<br>/<br>MOD | |
| Addition<br>Subtraction | +<br>- | |
| Compare | <,>,<=,>= | |
| equal to<br>Not equal to | =<br><> | |
| Logic and | AND | |
| Logical Iso-or | XOR | |
| Logical or | OR | Minimum |

Table 3-12 Priority of Operators

## 3. 8. 2 Instruction Statements

There are five main types of structured text statement tables, namely assignment statements, function and function block control statements, selection statements, iteration (loop) statements, and jump statements. Table 3-13 lists all the statements used in structured text.

| Instruction Type | Command statements | Examples |
|---|---|---|
| Assignment Statements | := | bFan:=TRUE. |
| Function and function block control statements | Function block / function call name (). | |
| Select statement | IF | IF<Boolean expression>THEN<br><statement content>.<br>END_IF |
| Select statement | CASE | CASE<Condition variable>OF<br><value1>:<statement content1>.<br>...<br><value n>:<statement content n>.<br>ELSE<br><ELSE statement content>.<br>END_CASE. |
| Iterative Statements | FOR | FOR<variable>:=<initial value>TO<target value>{BY<step Long>}DO<br><statement content<br>END_FOR. |
| | WHILE | WHILE<Boolean expression><br><statement content>.<br>END_WHILE. |
| | REPEAT | REPEAT<br><statement content |

| Instruction Type | Command statements | Examples |
|---|---|---|
| | | UNTIL<br><Boolean expressions<br>END_REPEAT. |
| Jump statements | EXIT | EXIT. |
| | CONTINUE | CONTINUE. |
| | JMP | <identifier>.<br>...<br>JMP<Identifier>. |
| Return statement | RETURN | RETURN. |
| NULL statement | . | |

Table 3-13 Structured Text

3. 8. 2. 1 Assignment statement

1) Format and function

Assignment statement is one of the most commonly used statement in structured text and serves to assign the value generated by the expression on its right side to the operand (variable or address) on the left side, using the ":=" representation.

The specific format is as follows:

<variable>:=<expression>.

[Example 3.27] Assign values to two boolean variables separately, with bFan set to TRUE and bHeater set to FALSE.

```
VAR
    bFan:BOOL.
    bHeater:BOOL.
END_VAR

bFan:=TRUE.
bHeater:=FALSE.
```

This is achieved by using the ":=" assignment statement.

2) Cautions in use

Matching of data types. If the data types on both sides of the assignment operator are different, the data type conversion function should be called. For example.

rVar1 is Real type and iVar1 is Int integer type. When iVar1 is assigned to rVar1, the conversion function of INT_TO_REAL should be called. Example: rVar1:=INT_TO_REAL(iVar1).

a. There can be more than one statement in a line, for example, arrData[1]:=3; arrData[2]:=12; the two instructions can be written in one line.

[Example 3.28] There can be multiple data in a line. arrData1[i]:=iDataInLine1;arrData2[j]:=iDataInLine2.

When a function is called, the function return value is assigned as the value of the expression, which should be the latest result of the evaluation.

[Example 3.29] The return value of a function call is used as the value of an expression.

```
Str1:=INSERT(IN1:='CoDe',IN2:='Sys',P:=2).
```

3) Function and function block control statement

Function and function block control statement are used to call functions and function blocks.

4) Function Control Statement

The function call assigns the return value directly to the variable as the value of the expression. For example, rVar1:=SIN(rData1); statement in which the call

Use the sine function SIN and assign the return value to the variable rVar1. The statement format is as follows:

Variable:= function name (parameter list).

[Example 3.30] Example of function control statement.

```
rResult:=ADD(rData1,rData2);//Use ADD function to assign the result of rData1 plus rData2 to the variable rResult.
```

5) Function block control statement

Function block calls are made by instantiating the name of the function block, such as Timer as the instance name of the TON function block, and the specific grid

The formula is as follows:

Function block instance name:(function block parameter).

If you need to call a function block in ST, you can directly enter the instance name of the function block and give the function block in the subsequent parentheses

Each parameter is assigned a value or variable, and the parameters are separated by commas; function block calls are terminated by semicolons.

For example, calling the function block TON timer in structured text, assuming its instance name is TON1, is implemented as in Figure 3.33

shown.

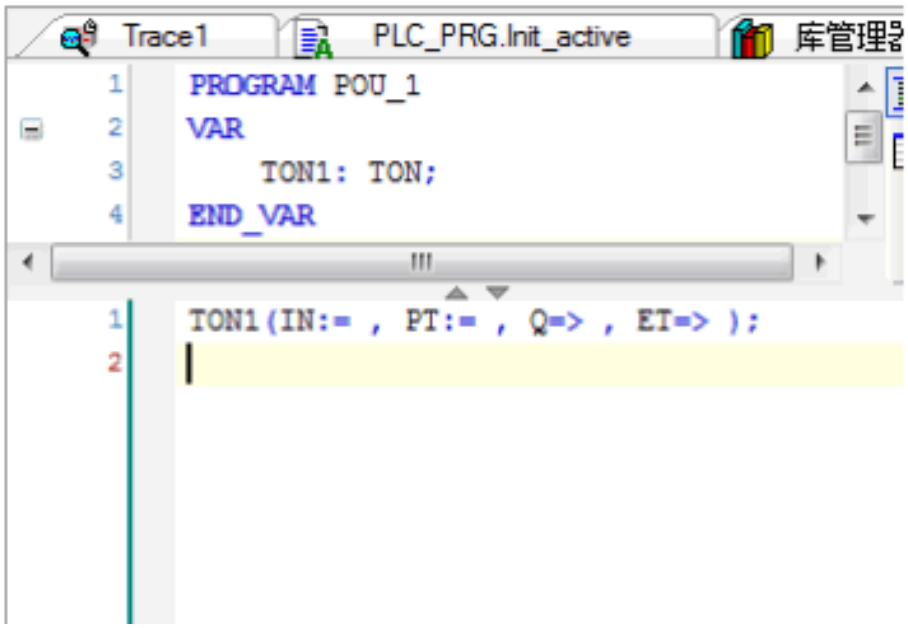



Figure 3.33 Structured text call function block

3. 8. 2. 2 Select statement

A select statement is a statement that determines the execution of its composition by selecting an expression based on a specified condition. In terms of broad categories, they can be divided into IF and CASE two categories.

1) IF statement

The basic format for implementing a single-branch selection structure with an IF statement is as follows.

IF<Boolean expression>THEN

<statement content>.

END_IF

If the above format is used, the statement content is executed only when the <Boolean expression> is TRUE, otherwise the <statement content> of the IF statement is not executed. The statement content can be a statement or can be an empty statement or can be juxtaposed with multiple statements. The flowchart of the statement expression execution is shown in Figure 3.34.

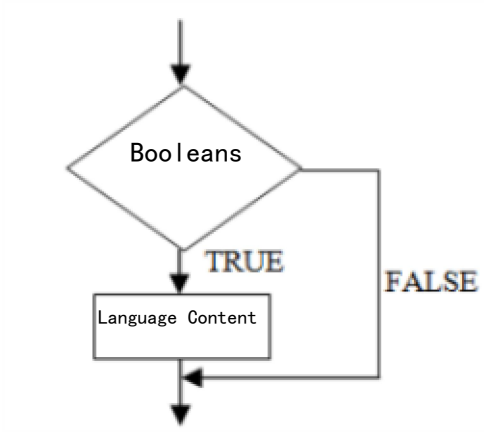



Figure 3.34 Flow chart of simple IF statement execution

Example 3.31] Use PLC to determine whether the current temperature exceeds 60 degrees Celsius, and if it does, always turn on the fan for heat dissipation.

```
VAR
    nTemp:BYTE;(*Current temperature status signal*)
    bFan:BOOL;(*fan switch control signal*)
END_VAR

nTemp:=80.
IFnTemp>60THEN
    bFan:=TRUE.
END_IF
```

2) IF...ELSE statement

The basic format for implementing a two-branch selection mechanism with an IF statement is as follows:

```
IF<Boolean expression>THEN
    <statement content1>.
ELSE
    <statement content2>.
END_IF
```

As the above expression first determines the value inside <Boolean expression>, if it is TRUE, then execute <statement content1>, if it is FALSE then execute <statement content 2>, and the program execution flow chart is shown in Figure 3.35.

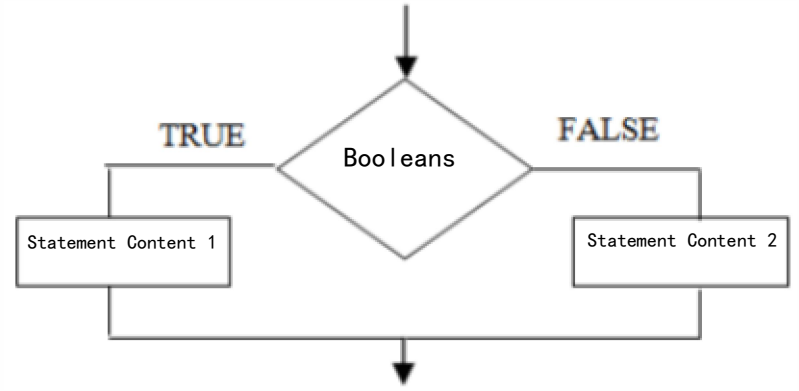



Figure 3.35 Flowchart of IF...ELSE statement execution

[Example 3.32] Use PLC to determine when the temperature is less than to 20 degrees Celsius, turn on the heating equipment, otherwise (temperature greater than or equal to 20 degrees Celsius) heating equipment disconnected state.

```
VAR
    nTemp:BYTE;(*Current temperature status signal*)
    bHeating:BOOL;(*Heater switch control signal*)
```

```
    END_VAR

    IFnTemp<20THEN
    bHeating:=TRUE.
ELSE
    bHeating:=FALSE.
END_IF
```

When the program has more than one conditional judgment formula, then it needs another nested IF...ELSE statement, i.e., multi-branch selection structure, with the following basic format.

```
IF<Boolean expression1>THEN
    IF<Boolean expression2>THEN
        <statement content1>.
ELSE
    <statement content2>.
END_IF
ELSE
    <statement content3>.
END_IF
```

As above, there is an IF...ELSE statement placed in IF...ELSE to achieve nesting, and the following is an example to illustrate the use of nesting.

As the above expression first determines the value within <Boolean expression 1>, and if it is TRUE, then continues to determine the value of <Boolean expression 2>. If the value of <Boolean expression 1> is FALSE, <statement 3> is executed, and it returns to <Boolean expression 2> for judgment. If <Boolean expression 2> is TRUE, <statement 1> is executed, and if not, <statement 2> is executed.

【Example 3.33】When the device enters the automatic mode, if the actual temperature is greater than 50 degrees Celsius, the fan will be turned on and the heater will be turned off, and when it is less than or equal to 50 degrees Celsius, the fan will be turned off and the heater will be turned on, such as in the manual mode, the heater fan will not act.

```
VAR
    bAutoMode:BOOL;(*hand/auto mode status signal*)
    nTemp:BYTE;(*Current temperature status signal*)
    bFan:BOOL;(*fan switch control signal*)
    bHeating:BOOL;(*Heater switch control signal*)
END_VAR

IFbAutoMode=TRUETHEN
    IFnTemp>50THEN
        bFan:=TRUE.
        bHeating:=FALSE.
    ELSE
        bFan:=FALSE.
        bHeating:=TRUE.
    END_IF
    ELSE
        bFan:=FALSE.
        bHeating:=FALSE.
    END_IF
```

3) IF...ELSIF...ELSE statements

In addition, the multi-branch selection structure can be presented in the following way. The specific format is as follows

```
IF<Boolean expression1>THEN
<statement content1>.
ELSIF<Boolean expression2>THEN
<statement content2>.
ELSIF<Boolean expression3>THEN
<statement content3>.
...
...
```

ELSE
<statement content n>.
END_IF

If the expression <Boolean expression 1> is TRUE, then only the instruction <statement content 1> is executed, and no other instruction is executed. Otherwise, the judgment starts from expression <Boolean expression 2> until one of the Boolean expressions is TRUE, and then the contents of the statement corresponding to this Boolean expression are executed. If none of the Boolean expressions is TRUE, then only instruction <statement n> is executed, and the program execution flow chart is shown in Figure 3.36.

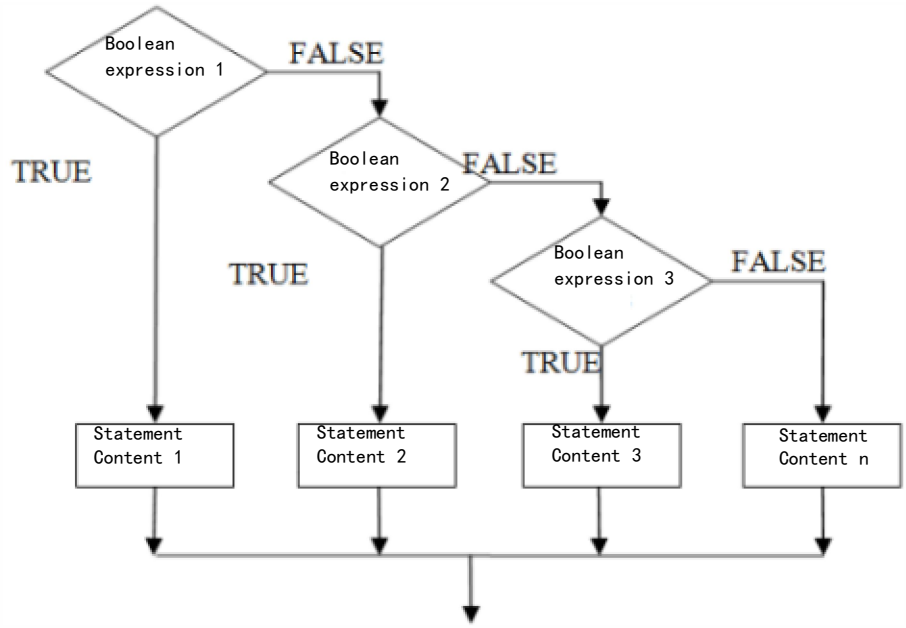



Figure 3.36 Flowchart of IF...ELSIF...ELSE statement execution

1) CASE statement

The CASE statement is a multi-branch selection statement that causes the program to select a branch for execution from among multiple branches based on the value of the expression, in the following basic format.

CASE<Condition variable>OF
<value1>:<statement content1>.
<value2>:<statement content2>.
<value3,value4,value5>:<statement content3>.
<value 6... Value 10>:<statement content 4>.
...
<value n>:<statement content n>.
ELSE
<ELSE statement content>.
END_CASE.

The CASE statement is executed according to the following pattern:

a. If the value of <condition variable> is <value i>, the instruction <statement content i> is executed.

b. If <condition variable> does not have any specified value, the instruction <ELSE statement content> is executed.

c. If several values of a condition variable are required to execute the same instruction, then several values can be written one after another and separated by commas. In this way, the common instruction is executed, as in the fourth line of the above program.

d. If you need the conditional variables to execute the same instructions within a certain range, you can separate them by writing the initial and final values with two points. In this way, the common instruction is executed as in the fifth line of the above program.

[Example 3.34] When the current state is 1 or 5, device 1 runs and device 3 stops; when the state is 2, device 2 stops and device 3 runs; if the current state is between 10 and 20, device 1 and device 3 both run, and in other cases, device 1, 2, and 3 are required to stop, and the specific code to achieve this is as follows:

VAR
nDevice1,nDevice2,nDevice3:BOOL;(*Device1..3 switch control signal*)

```
    nState:BYTE;(*Current state signal*)
END_VAR
CASEnStateOF
1,5.
    nDevice1:=TRUE.
    nDevice3:=FALSE.
2.
    nDevice2:=FALSE.
    nDevice3:=TRUE.
10..20.
    nDevice1:=TRUE.
    nDevice3:=TRUE.
ELSE
    nDevice1:=FALSE.
    nDevice2:=FALSE.
    nDevice3:=FALSE.
END_CASE.
```

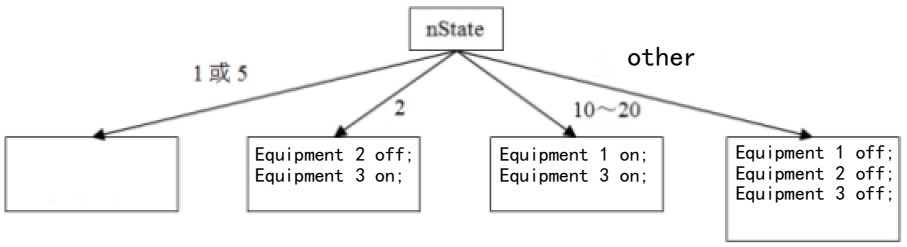



Figure 3.37 Flowchart of CASE statement example

The CASE statement flowchart is shown in Figure 3.37, where device 1 is on and device 3 is off when nState is 1 or 5;

nState is 2 when device 2 is off and device 3 is on;

nState is 10 to 20 when device 1 is off and device 3 is on;

In other cases, device 1 is off, device 2 is off, and device 3 is off.

2) Iterative statement

Iteration statements are mainly used for programs that are executed repeatedly. In CodeSys, the common iteration statements are FOR, REPEAT and the WHILE statement, which is explained in detail below.

❖ FOR loop

The FOR loop statement is used to compute an initialization sequence. When a condition is TRUE, the nested statements are repeated and a sequence of iterative expressions is computed, and the loop is terminated if it is FALSE.

FOR<variable>:=<initial value>TO<target value>{BY<step>}DO

    <statement content

    END_FOR.

❖ The order of execution of the FOR loop is as follows:

a. Calculating whether the <variable> is within the range of <initial value> and <target value>.
b. Execute <statement content> when <variable> is less than <target value>.
c. When <variable> is greater than <target value>, <statement content> is not executed.
d. Each time <statement content> is executed, <variable> always increases its value by the specified step size. The step size can be any integer value.

If no step is specified, the default value is 1. Exit the loop when <variable> is greater than <target value>.

In a sense, the principle of FOR cycle is like a photocopier, where the number of copies to be copied is first preset on the photocopier, and here is the condition of the cycle, and when the condition is satisfied, the number of copies is equal to the set number of copies, the copying stops.

The FOR loop is the most common type of loop statement. The FOR loop embodies a function that specifies number of iterations, one after the other. But other loop functions can also be implemented due to different ways of writing the code. Here is an example to demonstrate how to use a FOR loop.

[Example 3.35] Use a FOR loop to implement the fifth power of 2 calculation.

```
VAR
    Counter:BYTE;(*loop counter*)
    Var1:WORD;(*output result*)
```

END_VAR

FORCounter:=1TO5BY1DO
Var1:=Var1*2.
END_FOR.
Suppose the initial value of Var1 is 1, then the value of Var1 at the end of the loop is 32.
Caution:
If <target value> is equal to the limit value of <variable>, it will enter a dead loop. Suppose the type of the counter variable Counter in [Example 5.X] is SINT (-128 to 127), and the controller will enter a dead loop if the <target value> is set to 127. Therefore, you cannot set the limit value for <Target value>.

❖ WHILE loop

The WHILE loop is used in a similar way to the FOR loop. The difference between the two is that the end condition of a WHILE loop can be any logical expression. That is, you can specify a condition and when that condition is met, the loop is executed in the following format.

WHILE<Boolean expression>
<statement content>.
END_WHILE.
The WHILE loop is executed in the following order:

a. Computes the return value of a <Boolean expression>.
b. When the value of <Boolean expression> is TRUE, the <statement content> is repeated.
c. When the initial value of <Boolean expression> is FALSE, then the instruction <statement content> will not be executed and jumps to the end of the WHILE statement. The flowchart is shown in Figure 3.38

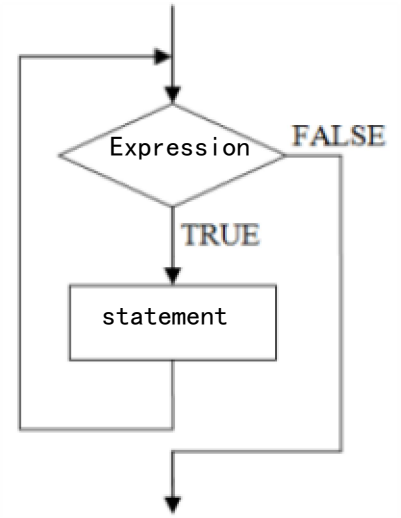



Figure 3.38 Flowchart of WHILE statement

Caution:
If the value of <Boolean expression> is always TRUE, then a dead loop will be generated and should be avoided. Dead loops can be avoided by changing the condition of the loop instruction. For example, use an incrementable or decrementable counter to avoid dead loops.

The WHILE statement controls a motor as in a project when the "Start" button is pressed (when the Boolean expression is TRUE).

The motor keeps rotating, and when the "Stop" button is pressed (when the Boolean expression is FALSE), the motor stops immediately. The following is an example of how to use the WHILE loop.

[Example 3.36] The program inside the loop is always executed as long as the counter does not go to zero.

VAR
Counter:BYTE;(*Counter*)
Var1:WORD.
END_VAR

WHILECounter<>0DO
Var1:=Var1*2.

```
    Counter:=Counter-1.
END_WHILE
```

In a certain sense, WHILE loops are more powerful than FOR loops, since WHILE loops do not need to know the number of loops before they are executed. Therefore, in some cases, it is sufficient to use only these two loops. However, if the number of loops is clearly known, then the FOR loop is better because the FOR loop avoids creating dead loops.

❖ REPEAT loop

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

The specific format is as follows.

```
REPEAT
    <statement content
UNTIL
    <Boolean expressions
END_REPEAT.
```

❖ The REPEAT loop is executed in the following order:

a. Executes <statement content> when the value of <Boolean expression> is FALSE.
b. Stops the execution of <statement content> when the value of <Boolean expression> is TRUE.
c. After the first execution of <statement content>, if the value of <Boolean expression> is TRUE, then <statement content> is executed only once.

Caution:

If the value of <Boolean expression> is always TRUE, then a dead loop will be generated and should be avoided. This can be done by changing the loop

The conditions in the instruction section are used to avoid dead loops. For example, the use of incrementable and decrementable counters to avoid dead loops.

The following is an example that demonstrates how to use the REPEAT loop.

[Example 3.37] Example of REPEAT loop, when the counter is 0, then stop the loop.

```
VAR
    Counter:BYTE.
END_VAR

REPEAT
Counter:=Counter+1.
UNTIL
Counter=0
END_REPEAT.
```

The result of this example is that each program cycle enters this REPEAT loop and Counter is BYTE (0-255), which means that 256 self-additive calculations are performed in each cycle.

Because of the previously mentioned "this means that the loop is executed at least once regardless of the end condition", Counter is first set to 1 whenever this REPEAT statement is entered, and the Counter:=Counter+1 instruction is executed 256 times in each cycle until the Counter variable is accumulated until it overflows to 0, jumping out of the loop. It is added to the overflow again, and so on and so forth.

❖ Jump statement

1) EXIT statement

If the EXIT instruction is used in the FOR, WHILE and REPEAT loops, the inner loop stops immediately, regardless of the end condition, in the following format.

```
    EXIT.
```

[Example 3.38] Use the EXIT instruction to avoid dividing by zero when using iterative statements.

```
        FOR Counter:=1 TO 5 BY 1 DO
            INT1:=INT1/2.
IF INT1=0 THEN
    EXIT;(*Avoid program division by zero*)
END_IF
    Var1:=Var1/INT1.
    END_FOR
```

When INT1 equals 0, the FOR loop ends.

2) CONTINUE statement

This instruction is an extension of the IEC 61131-3 standard. The CONTINUE instruction can be used in three types of loops: FOR, WHILE and REPEAT.

The CONTINUE statement breaks the loop, ignoring the code that follows it and starting a new loop directly. When multiple loops are nested, the CONTINUE statement can only cause the loop statement that directly contains it to start a new loop, as follows.

CONTINUE.

[Example 3.39] Use the CONTINUE instruction to avoid dividing by zero when using iterative statements.

```
VAR
    Counter:BYTE;(*loop counter*)
    INT1,Var1:INT;(*intermediate variable*)
Erg:INT;(*output result*)
END_VAR

FORCounter:=1TO5BY1DO
    INT1:=INT1/2.
IFINT1=0THEN
    CONTINUE;(*to avoid dividing by zero*)
END_IF

Var1:=Var1/INT1;(*execute only if INT1 is not equal to 0*)
END_FOR.
Erg:=Var1.
```

3) JMP statement

Jump statement, the jump instruction can be used to jump unconditionally to a line of code that uses the jump good marker, in the following format.

```
<identifier>.
.
JMP<Identifier>.
```

<identifier> can be any identifier that is placed at the beginning of a program line. the JMP instruction is followed by the jump destination, which is a predefined identifier. When the JMP instruction is executed, it will jump to the program line corresponding to the identifier.

Note: You must avoid creating dead loops, which can be combined with the use of IF conditional control jump instructions.

[Example 3.40] Use JMP statement to implement counter to loop in the range of 0..10.

```
VAR
    nCounter:BYTE.
END_VAR

Label1:nCounter:=0.
Label2:nCounter:=nCounter+1.

IFnCounter<10THEN
    JMPLabel2.
ELSE
    JMPLabel1.
END_IF
```

Label1 and Label2 in the above example are labels, not variables, so there is no need to make variable declarations in the program.

The IF statement determines whether the counter is in the range of 0-10, if it is, then the statement JMPLabel2 is executed and the program will jump to to Label2 in the next cycle and execute the program nCounter:=nCounter+1 to self-add 1 to the counter, and vice versa, it will jump to Label1 and execute nCounter:=0 to counter to zero.

The functionality in this example can also be achieved by using FOR, WHILE or REPEAT loops. In general, the use of JMP jump directives should be avoided, as this reduces the readability and reliability of the code.

4) RETURN instruction

The RETURN instruction is a return instruction to exit the Program Organization Unit (POU) in the following format.

RETURN.

[Example 3.41] Use the IF statement as a judgment to terminate the execution of this program immediately when the condition is satisfied.

```
VAR
    nCounter:BYTE.
    bSwitch:BOOL;(*switch signal*)
END_VAR

IFbSwitch=TRUETHEN
RETURN.
END_IF.
nCounter:=nCounter+1.
```

When bSwitch is FALSE, nCounter always performs self-adding 1. If bSwitch is TRUE, nCounter keeps the value of the previous cycle and immediately exits this Program Organization Unit (POU).

7、Empty statement

That is, nothing is executed.

The specific format is as follows.

.

8、Comments

Comments are a very important part of a program, making it more readable without affecting the execution of the program. In comments can be added anywhere in the declaration section or the execution section of the ST editor.

In the ST language, there are two methods of annotation:

1) Multi-line comments start with (* and end with *). This comment method allows multi-line comments, as shown in Figure 3.39 of a).

2) Single line comment starts with "//" and continues to the end of the line. This is the method for single line comments, as shown in Figure 3.39, b)

```
(*
    bOperationActive:=FALSE;
    bOrderActive:=FALSE;
    bRecipeActive:=FALSE;
    bInfoActive:=FALSE;
    bServiceActive:=FALSE;
    bSimulationActive:=FALSE;
*)
    IF iMainAreaIndex = 0 THEN
        bOperationActive:=TRUE;
    ELSIF iMainAreaIndex = 1 THEN
        bOrderActive:=TRUE;
```

a)

```
// gesture handling:
// only when mouseup was done
IF xRight AND bDragCanStart = FALSE THEN
    xRight := FALSE;
    IF iMainAreaIndex < MAX_MODULES-1 THEN
        iMainAreaIndex := iMainAreaIndex + 1;
        bIndexChanged := TRUE;
    END_IF
```

b)

a) Multi-line comments b) Single line comments

Figure 3.39 Structured text language annotation

## 3. 8. 3 Application examples

### 3. 8. 3. 1 Control requirements

[Example 3.24] Hysteresis function block FB_Hystersis.

This function block has three input signals, the current real-time value input signal, the comparison set value input signal and the deviation value input signal. In addition, an output value is required. when the output is TRUE, the output switches to FALSE only when the input signal IN1 is smaller than VAL-HYS. when the output signal is FALSE, the output switches to FALSE only when the input signal IN1 is larger than VAL+HYS.

The output is switched to TRUE only when the output is switched to TRUE.

The input and output variables of the function block FB_Hystersis are defined as follows.

```
FUNCTION_BLOCKFB_Hysteresis
VAR_INPUT
    IN1:REAL;//Input signal
    VAL:REAL;//Compare signals
    HYS:REAL;//Hysteresis deviation signal
```

END_VAR

VAR_OUTPUT
Q:BOOL.
END_VAR
The schematic diagrams of the hysteresis process and the functional block graphics are shown in Figure 3.40 for a) and b), respectively.

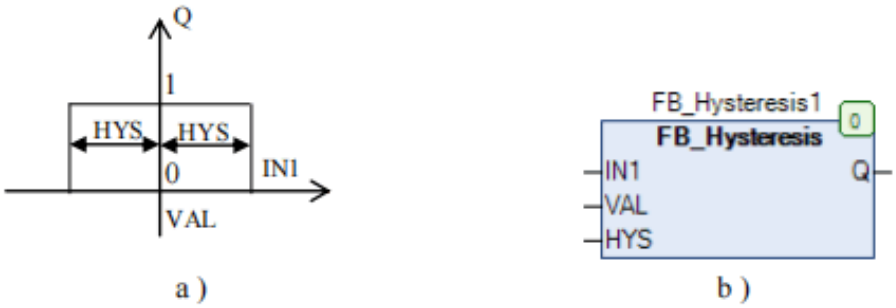



Figure 3.40 Hysteresis function block
Hysteresis process schematic b) Functional block graphic schematic

3. 8. 3. 2 Function block programming

The procedure of the function block body for determining the input signal is as follows.
IFQTHEN
IFIN1<(VAL-HYS)THEN
Q:=FALSE;//IN1 decreases
END_IF

ELSIFIN1>(VAL+HYS)THEN
Q:=TRUE;//IN1 increase
END_IF

3. 8. 3. 3 Function Block Application

The FB_Hysteresis function block can be used for bit signal control, where IN1 links the process variable rActuallyValue, VAL links the process set value rSetValue, and rTolerance is the desired control deviation, and the declarative part of the program is as follows:
PROGRAMPOU
VAR
fbHysteresis:FB_Hysteresis;//fbHysteresis is an instance of the FB_Hysteresis function block
rActuallyValue:REAL;//Actually measured value
rSetValue:REAL;//process set value
rTolerance:REAL;//deviation setting value
bOutputAT%QX0.0:BOOL;//bit signal output
END_VAR
- The ontology of the program is as follows:

fbHysteresis(IN1:=rActuallyValue,VAL:=rSetValue,HYS:=rTolerance,Q=>bOutput).
The program section as above can also be represented by the following program, and the result is the same.
fbHysteresis(IN1:=rActuallyValue,VAL:=rSetValue,HYS:=rTolerance).
bOutput:=fbHysteresis.Q.

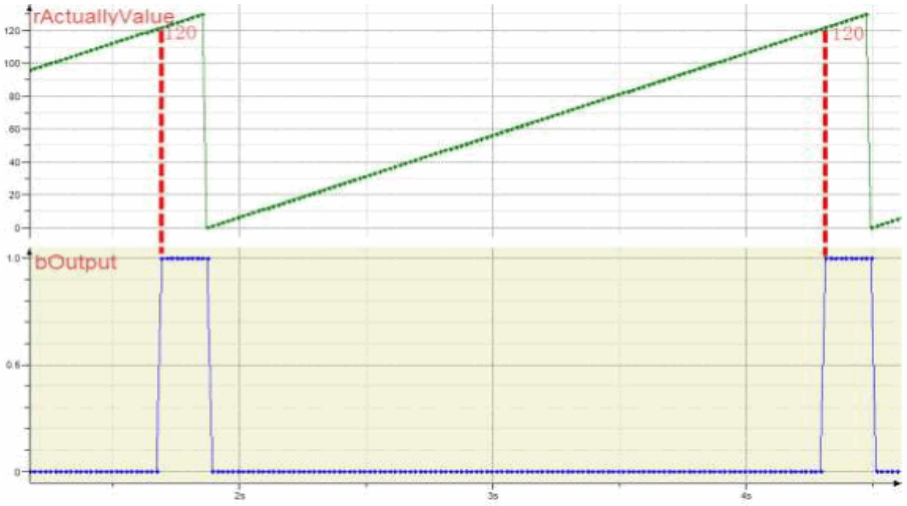



Figure 3.41 Hysteresis function block program operation results

Figure 3.41 shows the results of the actual program run, where rSetValue is set to 100 and rTolerance is set to 20. When the value of rActuallyValue is incremented from 0 to 120, the bOutput signal is set to TRUE, and then when rActuallyValue drops to 0, bOutput also changes to FALSE, and theoretically when it drops to 80, bOutput changes to FALSE.

The function block FB_Delay is a time lag function block, which is different from the FB_Hystersis hysteresis function block. The time at which the output signal lags the input signal in time is called time lag. The controlled object of the production process is often described by a first-order filtering link plus time lag. Only the time lag function block is introduced here, and the first-order filtering is not introduced much.

The transfer function of the time lag link is

$$Y(s) = e^{-st}X(s) \quad (4\text{-}1)$$

Assuming that the sampling period is, the discretization gives, after

$$Y(k) = X(k - N) \quad (4\text{-}2)$$

where X is the input signal of the time lag link; Y is the output signal of the time lag link. Let the sampling period used for discretization be that

The ratio of the time lag to the sampling period is the number of lag taps N.

### 3. 8. 3. 4 Variable declaration for function block FB_Delay

The program uses an array to store the input signal, and the array stores the sampling data at different moments, cell 1 stores the moment 1× the sampled values, and the i-th cell stores the sampled values at moment i×. The integer value of the ratio of the time lag time to the sampling period is N (the fractional part of N is removed and denoted by N). Therefore, if the input signal is stored in cell N at a certain moment, the time -lagged output signal should be output from storage cell 1.

```
FUNCTION_BLOCKFB_Delay
VAR_INPUT
    IN:REAL;//Input signal
    bAuto:BOOL;//Automatic manual flag signal
    tCycleTime:TIME;//sampling period
    tDelayTime:TIME;//time lag time
END_VAR
VAR_OUTPUT
    rOutValue:REAL;//the output after the time lag link processing
END_VAR
VAR
    N:INT;//lagged beat number
    arrValue:ARRAY[0..2047]OFREAL;//first-in-first-out array stack
    i:INT;// subscript of the array for input
    j:INT;// subscript of the array, used for output
    fbTrig:R_TRIG;//convert auto signal to pulse
    fbTon:TON.
END_VAR
```

When the above input and output parameters are filled in, the function block diagram can be called through the graphical programming language to see the effect schematic in Figure 3.42.

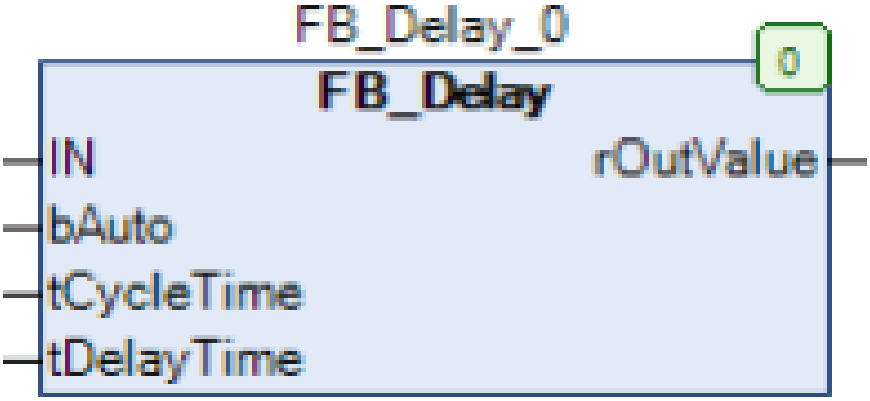



Figure 3.42 Graphical diagram of the FB_Delay function block

3. 8. 3. 5 Program ontology of function block FB_Delay

```
N:=TIME_TO_INT(tDelayTime)/TIME_TO_INT(tCycleTime).
fbTrig(CLK:=bAuto).

IFfbTrig.QTHEN
    i:=N.
    j:=0.
END_IF
fbTon(IN:=NOTfbTon.Q,PT:=tCycleTime).

IFfbTon.QANDbAutoTHEN
    i:=(i+1)MOD2000.
    arrValue[i]:=i.
    j:=(j+1)MOD2000.
    rOutValue:=arrValue[j].
END_IF
```

The function block ontology uses two subscript windows to manage the access and output of the input and output signals. The input signal data is stored in the i-subscript address of the array X with an initial value equal to the number of lag taps. The output signal is stored in the j subscript address of the array X, and the initial value of the output is equal to 0. The modulo method is used to determine the address of each storage and output, and the original address is added by 1 after each operation. The next execution operation is guaranteed to store the input of that time and the previous N input signals as the output of that time.

The number of array memory cells determines the time lag size and sampling period. The larger the time lag and the smaller the sampling period, the more memory cells are required. Generally, the number of lag taps N can be larger than the total number of memory cells according to the size of the application.

In the example, the number of lag beats N is required to be less than 2000 (the length of the array is 2048). In addition, the storage cells of the array start at address 0. The actual application starts at address 0. Figure 3.43 shows the relationship between the input window and the output window.

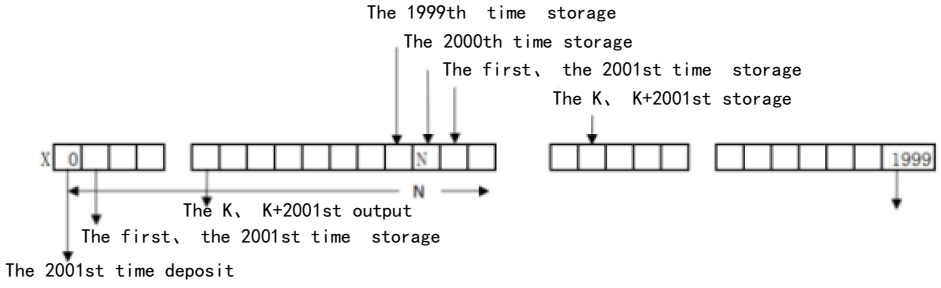



Figure 3.43 Relationship diagram of input/output windows

3. 8. 3. 6 Notes on using function FB_Delay

The hysteresis beat number N is related to the time lag and sampling period, and the program uses the signal for switching the running state to the automatic state as the pulse signal for the initial value setting.

This function block can be combined with a first-order filtering link for simulating actual production processes and conducting control system simulation studies.

[Example 3.26] Calculate the maximum, minimum, and average values.

In some industrial controls, it is often necessary to calculate the average, maximum and minimum values of several measured values. Such applications are implemented below using a structured text programming language.

1) Control requirements

There are 32 points in a furnace where the temperature values need to be measured, and the maximum, minimum and average values of these 32 points are calculated separately.

2) Program writing

The program defines the maximum value, minimum value, cumulative sum, and average value, respectively, and the specific variables are defined as shown below:

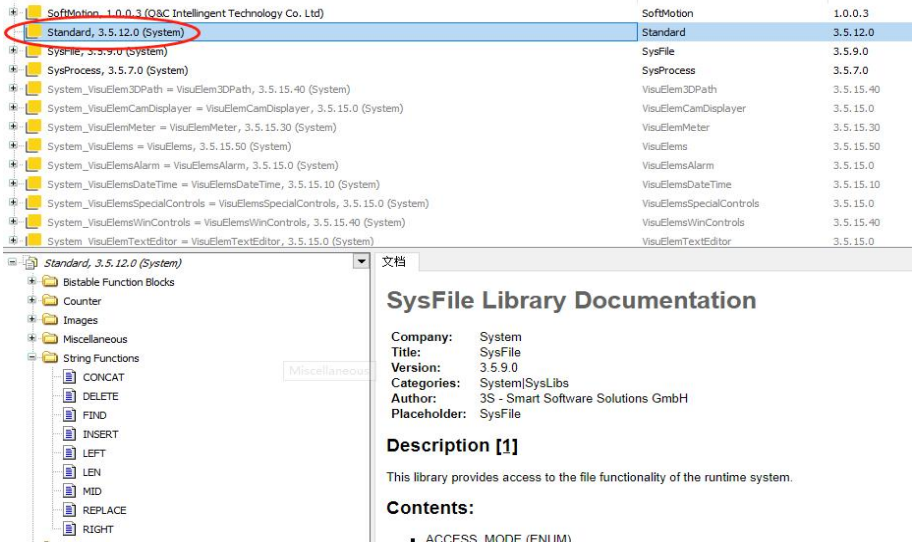
```
PROGRAMPLC_PRG
VAR
    rMaxValue:REAL;//maximum value
    rMinValue:REAL;//minimum value
     rSumValue:LREAL;// sum
     rAvgValue:REAL;//average
     arrInputBufferAT%IW100:ARRAY[1..32]OFREAL;//Input source data
     i:INT.
END_VAR
The main procedure is as follows.
rSumValue:=0.
FORi:=1TO32BY1DO
rSumValue:=REAL_TO_LREAL(arrInputBuffer[i])+rSumValue.
    IFarrInputBuffer[i]>rMaxValueTHEN
        rMaxValue:=arrInputBuffer[i].
    END_IF
    IFarrInputBuffer[i]<rMinValueTHEN
        rMinValue:=arrInputBuffer[i].
    END_IF
END_FOR.
rAvgValue:=rSumValue/32.
```

Use the FOR...DO statement to scan all input channels and calculate the average, maximum and minimum values, in addition to calculating the sum.

# 3. 9 String commands

## 3. 9. 1 Basic Instructions

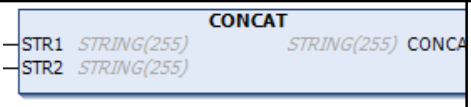
String handling functions, as shown below So, a new project will automatically install the Standard library.



- **CONCAT[FUN]**

Concatenate two strings

1) Command format

| Instruction | Name | FB/FUN | Graphical representation | ST Performance |
|---|---|---|---|---|
| CONCAT | String* concate nation | FUN |  | Out:=concat(STR1,STR2) |

2) Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | CONCAT | STRING(255) | Serial string, up to 255 characters. If the result does not fit this 255 bytes, it will be truncated by default. No error is generated. |
| Input | STR1 | STRING(255) | String 1 to be concatenated, maximum 255 characters |
| | STR2 | STRING(255) | String 2 to be concatenated, maximum 255 characters |

3) Examples

Concatenate two strings

Concatenate (STR1, STR2) means: concatenate STR1 and STR2 to a single string STR1STR2.

(*Exampledeclaration*)VARSTRING1:STRING.

(*ExampleinST,resultis'SUSIWILLI'*)VARSTRING1:=CONCAT('SUSI','WILLI').

- **DELETE[FUN]**

Remove multiple characters from a string

1) Command format

| Instruction | Name | FB/FUN | Graphical representation | ST Performance |
|---|---|---|---|---|
| DELETE | String * Delete | FUN |  | Out:=DELETE(STR,LEN,POS. |

2) Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | DELETE | STRING(255) | The remaining string after deletion |
| Input | STR | STRING(255) | The string to be modified |
| | LEN | INT | Length of the part of the string to be deleted, number of characters |
| | POS | INT | position in STR, after which the deletion starts. Counting from the left, starting from 1 |

3) Example

Deleting multiple characters from a string Delete (STR, LEN, POS) means remove the LEN characters from STR and start with the character in the POS position. POS=0 or POS=1, both resolve the first character.

Caution:

Unfortunately, the current implementation is incorrect for case Pos=0. Due to compatibility reasons, it is not possible to change the implementation.

If Pos=0 is used, the parameter LEN will be reduced by one parameter internally!

It is usually recommended to use values within the range specified in IEC 61131-3. the minimum value of Pos is specified as 1.

(*Exampledeclaration*)VARSTRING1:STRING.

(*ExampleinST,resultis'SUSI'*)VARSTRING1:=DELETE('SUXYSI',2,3).

- **FIND [FUN]**

Search for the position of a partial string in a string.

1) Command format

| Instruction | Name | FB/FUN | Graphical representation | ST Performance |
|---|---|---|---|---|
| FIND | String* Search | FUN |  | Out:=FIND(STR1,STR2). |

2) Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | FIND | INT | The position of the character where STR2 first occurred in STR1. If no occurrence is found, the result is 0 |
| Input | STR1 | STRING(255) | Search for STR2 strings |
| | STR2 | STRING(255) | String whose position is searched in STR1 |

3) Example

Search for the position of a partial string in a string.

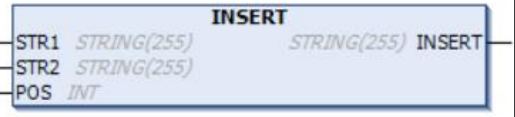
FIND (STR1, STR2) means: Find the position of the first character in STR1 where STR2 first appears. If STR2 is not found in STR1, it is: = 0.

(*Exampledeclaration*)arINT1:INT.
(*ExampleinST,resultis'4'*)arINT1:=FIND('abcdef','de').

- **INSERT [FUN]**

Insert a string into another string at a specific location

1) Command format

| Instruction | Name | FB/FUN | Graphical representation | ST Performance |
|---|---|---|---|---|
| INSERT | String* insertion | FUN |  | Out:=DELETE(STR1,STR2,POS). |

2) Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | INSERT | STRING(255) | Return string |
| Input | STR1 | STRING(255) | Insert the string of STR2 |
| | STR2 | STRING(255) | The string will be inserted into STR1 |
| | POS | INT | Insert position. If POS is 255 or <0>, the result is equal to STR1<br>0: insert before the first character<br>1: Insert after the first character. |

3) Example

Insert a string into another string at a specific location

Insertion (STR1, STR2, POS) means: insert STR2 into STR1 after position POS.

(*Exampledeclaration*)
VarSTRING1:STRING.
(*ExampleinST,resultis'SUXYSI'*)
VarSTRING1:=INSERT('SUSI','XY',2).

- **LEFT [FUN]**

Returns the number of specific characters in the string starting from the left

1) Command format

| Instructio | Name | FB/FUN | Graphical representation | ST |
|---|---|---|---|---|

| n | | | | Performance |
|---|---|---|---|---|
| LEFT | String* left returns the number of specific characters | FUN |  | Out:=LEFT(STR,SIZE). |

2) Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | LEFT | STRING(255) | Generated string |
| Input | STR | STRING(255) | String to be analyzed |
| | SIZE | INT | Number of characters |

3) Example

Returns the number of specific characters in the string starting from the left

Left side (STR, size) means: return the first size character from the left side in the string STR

```
VarSTRING1:STRING.
(*ExampleinST,resultis'SUS'*)
VarSTRING1:=LEFT('SUSI',3).
```

- **LEN [FUN]**

Returns the number of characters in the string

1) Command format

| Instruction | Name | FB/FUN | Graphical representation | ST Performance |
|---|---|---|---|---|
| LEN | String*Number of strings | FUN |  | Out:=LEN(STR). |

2) Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | LEN | INT | Length of string STR |
| Input | STR | STRING(255) | String to be analyzed |

3) Example

Returns the number of characters in the string

```
(*Exampledeclaration*)
VarINT1:INT.
(*ExampleinST,resultis'4'*)
VarINT1:=LEN('SUSI').
```

- **MID [FUN]**

Returns the number of characters in a string from a specific position

1) Command format

| Instruction | Name | FB/FUN | Graphical representation | ST Performance |
|---|---|---|---|---|
| MID | String* returns a specific string at a specific position | FUN |  | Out:=MID(STR,LEN,POS). |

2) Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | MID | STRING(255) | Partial string STR |
| Input | STR | STRING(255) | String to be analyzed |
| | LEN | INT | Number of characters, counting from the left |
| | POS | INT | Partial string start position |

3) Example

Returns the number of characters in a string from a specific position

MID (STR, LEN, POS) means: retrieve the LEN character from the STR string, starting with the character at position POS.

```
(*Exampledeclaration*)
VarSTRING1:STRING.
(*ExampleinST,resultis'US'*)
VarSTRING1:=MID('SUSI',2,2).
```

- **REPLACE [FUN]**

Replace a specific number of characters of a string with another string

1) Command format

| Instruction | Name | FB/FUN | Graphical representation | ST Performance |
|---|---|---|---|---|
| REPLACE | String* replaces a specific number of strings | FUN |  | Out:=replace( STR1,STR2, L, P). |

2) Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | REPLACE | STRING(255) | Generated string |
| Input | STR1 | STRING(255) | Replace a partial string |
| | STR2 | STRING(255) | Replace the STR1 part of the string |
| | L | INT | Number of characters, counting from left |
| | P | INT | The start position of the character to be replaced. p=1 or p=0 both resolve the first character |

3) Example

Replace a specific number of characters of a string with another string

Replace (STR1, STR2, L, P) means: replace the L character in STR1 with STR2, starting from the character in position P.

POS=0 or POS=1, both resolve the first character.

```
(*Exampledeclaration*)
VarSTRING1:STRING.
(*ExampleinST,resultis'SKYSI'*)
VarSTRING1:=REPLACE('SUXYSI','K',2,2).
```

- **RIGHT[FUN]**

Returns the number of specific characters in the string starting from the right

1) Command format

| Instruction | Name | FB/FUN | Graphical representation | ST Performance |
|---|---|---|---|---|
| RIGHT | The right side of the string * returns a specific number of characters | FUN |  | Out:=right(ST R1,SISE). |

2) Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | RIGHT | STRING(255) | Generated string |
| Input | STR | STRING(255) | String to be analyzed |
| | SIZE | INT | Number of strings |

3) Example

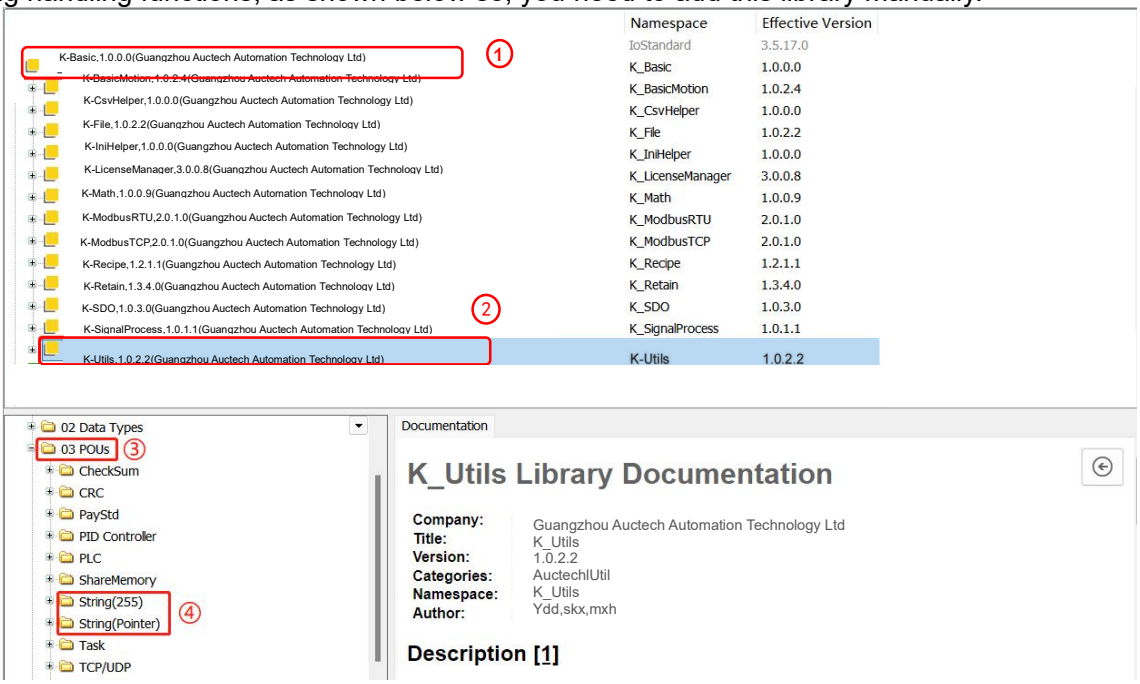
Returns the number of specific characters in the string starting from the right

Right side (STR, size) means: return the first size character from the right side in the string STR

```
    (*Exampledeclaration*)
VarSTRING1:STRING.
    (*ExampleinST,resultis'USI'*)
VarSTRING1:=RIGHT('SUSI',3).
```

## 3. 9. 2 Expansion Instructions

String handling functions, as shown below so, you need to add this library manually.




### 3. 9. 2. 1 Converted String Instructions

- **ByteToHexString [FUN]**

byte to hexadecimal string

1) Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|
| ByteToHex String | byte to hexadecimal string |  | Out:=ByteToHexString(in) |



2) Variables

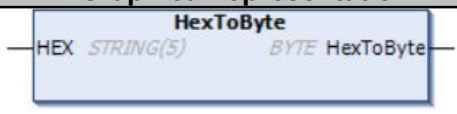
| Scope | Name | Type |
|---|---|---|
| Return | ByteToHexString | STRING(3) |
| Input | in | BYTE |

3) Example.

- **HexToByte [FUN]**

Convert hexadecimal string to byte value

1) Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|
| ByteToByte | Hexadecimal string to byte |  | Out:=HexToByte(in) |

2） Variables

| Scope | Name | Type |
|---|---|---|
| Return | HexToByte | BYTE |
| Input | HEX | STRING(5) |

3） Example

- **HexToDword [FUN]**

Hexadecimal string to Dword value

1） Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|
| ByteToDword | byte to Dword |  | Out:=ByteToByte(in) |

2） Variables

| Scope | Name | Type |
|---|---|---|
| Return | HexToDword | DWORD |
| Input | Hex | STRING(20) |

3） Example

- **IsHex [FUN]**

Byte to hexadecimal string conversion

1） Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|
| IsHex | byte to hex string conversion |  | Out:=IsHex(in) |

2） Variables

| Scope | Name | Type |
|---|---|---|
| Return | IsHex | BOOL |
| Input | IN | BYTE |

3） Example

- **Split [FUN]**

Split a string by a character, use a string(255) array to receive the split string related instructions

1） Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|
| Split | Split String |  | split(src:=,pdest:,separator:=,idestSize:=,num=>) |

2） Variables

| Scope | Name | Type | Document |
|---|---|---|---|
| Return | IsHex | BOOL | |
| Input | Src | BYTE | The string to be split |
| | Pdest | Pointertostring(255) | The first address of the array that receives the string (the address of buf) |
| | Separator | string | Specify the split character |
| | IdestSize | int | The length of the array receiving the string |

| Scope | Name | Type | Document |
|---|---|---|---|
| Output | Num | int | Number of strings after splitting |
| | split | bool | |

3） Example

```
(*declaration*)VAR
STR:STRING(255):='1.2.3.4.5.6.7.8.9.10.11.12'.
dst:ARRAY[0..12]OFSTRING(255);END_VAR
(*execution*)STR:='1.2.3.4.5.6.7.8.9.10.11.12';split(src:=STR,pdest:=ADR(dst[0]),separator:='.'
,idestSize:=12,num=>).
```

- **StringToWS [FUN]**

Use the specified character to split the string, not splitting returns ''

1） Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|
| StringToWS | Split String |  | StringToWS(str). |

2） Variables

| Scope | Name | Type |
|---|---|---|
| Return | StringToWS | WSTRING |
| Input | str | STRING |

3） Example

Use the specified character to split the string, not splitting returns ''

```
(*declaration*)VAR
STR:STRING(255):='1.2.3.4.5.6.7.8.9.10.11.12'.
index:INT.
STR2:STRING(255):='';END_VAR
(*execution*)index:=0;whileSTR2<>''do//Calltosplitonecharacteratatime
STR2:=strtok(src:=STR,delim:='.') .
index:=index+1.

//Theresultisasfollows
//indexSTR2STR
//0'1''2.3.4.5.6.7.8.9.10.11.12'
//1'2''3.4.5.6.7.8.9.10.11.12'
//2'3''4.5.6.7.8.9.10.11.12'
//...
//...
//...
//...
//...
//10'11''12'
//11'12'''END_WHILE
```

- **Trim [FUN]**

Removes the beginning and end of the string or other characters. The function executes successfully and returns the string with the first part of the string removed and the

Trailing string

1） Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|
| Trim | Remove the beginning and end of the string or other characters |  | Trim(str). |

2） Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | Trim | STRING(255) | |
| Input | str | STRING(255) | Delete from the beginning and end of a string |

3） Example

Removes the beginning and end of the string or other characters. When the function succeeds, it returns the string with the first and last part of the string removed

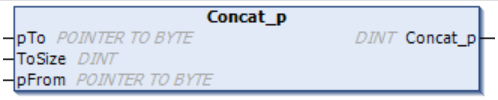
```
(*declaration*)VAR
STR:STRING(255):='test1.2.3.4.5.trim'.
STR2:STRING(255):='test21.2.3.4.5.trim2';END_VAR
(*execution*)STR:=Trim(STR);//'test1.2.3.4.5.trim'---------
>'test1.2.3.4.5.trim'STR2:=Trim(STR2);//'test21.2.3.4.5.trim2'------ --->'test21.2.3.4.5.trim2'
```

### 3. 9. 2. 2 Pointer-type String Instructions

- **Concat_p [FUN]**

String Splicing Function

1） Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|
| Concat_p | String Splicing |  | Concat_p (pto:=,toSize:=,pFrom) . |

2） Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | Concat_p | DINT | Returns the number of splices, -1 means the number of characters is overflowing |
| Input | pTo | POINTERTOBYTE | To stitch the target string pointer address |
| | ToSize | DINT | Target string size, number of bytes |
| | pFrom | POINTERTOBYTE | Splice source string pointer address |

3） Example

```
(*declaration*)VAR
To:STRING(1000).
From:STRING(255):='Helloworld'.
index:int;END_VAR
(*execution*)index:=0;to:='';WHILEConcat_p(adr(to),sizeof(to),adr(From))<>-1do
index:=index+1.

//indexTo
//1'Helloworld'
//2'HelloworldHelloworld'
//...
//...
//...
//n'HelloworldHelloworld'
//KnowthatthereturnvalueIs-1END_WHILE
```

- **Delete_p [FUN]**

Delete characters of the specified length from a string

1） Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|

| | | | |
|---|---|---|---|
| Delete_p | Delete String |  | Delete_p (pSrc:=,SrcSize:=,LEN:=, Pos:=). |

2） Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | Delete_p | DINT | |
| Input | pTo | POINTERTOBYTE | Source string address |
| | SrcSize | DINT | Source string size, byte count |
| | Len | DINT | Delete string length |
| | Pos | DINT | Delete the starting position of the string, starting from 1 |

3） Example

```
(*declaration*)VAR
str:STRING:='12345678910';END_VAR
(*execution*)Delete_p(adr(str),sizeof(str),3,6);//str='12345610'
```

- **Find_p[FUN]**

Find the string from the source string and return the position

1） Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|
| Find_p | Find String |  | Find_p (pSrc:=,SrcSize:=,Str:=). |

2） Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | Find_p | DINT | |
| Input | pSrc | POINTERTOBYTE | Source string address |
| | SrcSize | DINT | Source string size, byte count |
| | Str | STRING(255) | |

3） Example

Find the string from the source string and return the position

```
(*declaration*)VAR
str:STRING:='12345678910'.
iFindIndex:dint;END_VAR
(*execution*)iFindIndex:=Find_p(adr(str),sizeof(str),'6');//iFindIndex=6
```

- **IsSpace_p [FUN]**

Determine if the input character is a /carriage return/tab, etc.

1） Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|
| IsSpace_p | Determine the input string |  | IsSpace_p (str:=). |

2） Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | IsSpace_p | BOOL | |

| Scope | Name | Type | Comment |
|---|---|---|---|
| Input | Str | BYTE | Single character |

3） Example

```
(*declaration*)VAR
bspace:byte:=32;//''
btab:byte:=9;//\t.
bnewline:byte:=10;//\n
breturn:byte:=13;//\r.
bA:byte:=65;//A.
bYes:bool;END_VAR
(*execution*)
ifIsSpace_p(bspace)then
bYes:=TRUE;end_if
ifIsSpace_p(bnewline)then
bYes:=TRUE;end_if
ifIsSpace_p(btab)then
bYes:=TRUE;end_if
ifIsSpace_p(breturn)then
bYes:=TRUE;end_if
ifIsSpace_p(bA)then
bYes:=TRUE;ELSE
bYes:=FALSE;end_if
```

- **Left_p [FUN]**

Fetch the specified number of characters from the left side of the string to the target string, starting from the first character

1） Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|
| Left_p | Take out the string on the left |  | Left_p (pSrc:=, the SrcSize: =, the pDest:=. DestSize:=Count:=). |

2） Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | Left_p | DINT | Return -1 means the number of characters overflowed |
| Input | pSrc | POINTERTOBYTE | Source string address |
| | SrcSize | DINT | Source string size, byte count |
| | pDest | POINTERTOBYTE | Destination Address |
| | DestSize | DINT | Target string size |
| | Count | DINT | Number of bytes fetched |

3） Example

```
(*declaration*)VAR
Src:STRING:='123456789'.
Dest:string;END_VAR
(*execution*)
Left_p(ADR(Src),sizeof(Src),adr(Dest),sizeof(Dest),3);//Dest='123'
```

- **Len_p [FUN]**

Find the length of the string

1） Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|

| | | | |
|---|---|---|---|
| Len_p | String length |  | Len_p (pStr:=). |

2） Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | LEN_p | DINT | |
| Input | pStr | POINTERTOBYTE | String address |

3） Example

```
(*declaration*)VAR
Str:STRING:='123456789'.
iLen:dint;END_VAR
(*execution*)
iLen:=Len_p(ADR(Str));//iLen=9
```

- **Mid_p [FUN]**

Retrieve the specified number of characters from the source string at the specified position to the target string

1） Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|
| Mid_p | Fetch a specific string |  | Len_p (pStr:=). |

2） Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | Mid_p | DINT | Return -1 means the number of characters overflowed |
| Input | pSrc | POINTERTOBYTE | Source string address |
| | SrcSize | DINT | Source string size, byte count |
| | pDest | POINTERTOBYTE | Destination Address |
| | DestSize | DINT | Target string size |
| | Position | DINT | StartpositionFORtheparttialSTRING |
| | Length | DINT | NumberOFcharacters,countedFROMtheleft |

3） Example

```
(*declaration*)VAR
Src:STRING:='123456789'.
Dest:STRING;END_VAR
(*execution*)
Mid_p(ADR(Src),sizeof(Src),adr(Dest),sizeof(Dest),5,2);//Dest='67'
```

- **Right_p [FUN]**

Retrieve the specified number of characters from the right side of the source string to the target string

1） Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|

| | | | |
|---|---|---|---|
| \|Right_p | Take out the string on the right side |  | Right_p (pSrc:=, the SrcSize: =, the pDest:=. DestSize:=Count:=) . |

2） Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | Right_p | DINT | Return -1 means the number of characters overflowed |
| Input | pSrc | POINTERTOBYTE | Source string address |
| | SrcSize | DINT | Source string size, byte count |
| | pDest | POINTERTOBYTE | Destination Address |
| | DestSize | DINT | Target string size |
| | Count | DINT | Number of bytes fetched |

3） Example

```
(*declaration*)VAR
Src:STRING:='123456789'.
Dest:string;END_VAR
(*execution*)
Right_p(ADR(Src),sizeof(Src),adr(Dest),sizeof(Dest),3);//Dest='789'
```

- **ToLower_p [FUN]**

Convert English characters in a string to lowercase

1） Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|
| ToLower_p | String to lowercase |  | ToLower_p (pStr:=,). |

2） Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | ToLower_p | DINT | |
| Input | pStr | POINTERTOBYTE | String address |

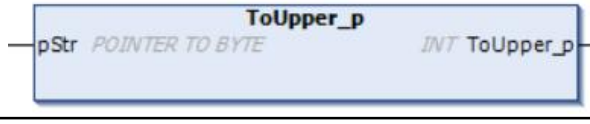
3） Example

```
(*declaration*)VAR
Str:STRING:='GuangDongQ&CIntelligentTechnologyCo,Ltd';END_VAR
(*execution*)
ToLower_p(ADR(Str));//Str='guangdongq&cintelligenttechnologyco,ltd'
```

- **ToUpper_p[FUN]**

Convert English characters in a string to uppercase

1） Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|
| ToUpper_p | String to uppercase |  | ToUpper_p (pStr:=,). |

2） Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | ToUpper_p | INT | |
| Input | pStr | POINTERTOBYTE | String address |

3） Example

```
(*declaration*)VAR
Str:STRING:='GuangDongQ&CIntelligentTechnologyCo,Ltd';END_VAR
(*execution*)
ToLower_p(ADR(Str));//Str='GUANGDONGQ&CINTELLIGENTTECHNOLOGYCO,LTD'
```

- **Trim_p [FUN]**

Whitespace characters are removed from both ends of a string. The whitespace characters in this context are all whitespace characters (space, tab, no-breakspace, etc.) and all line terminator characters (e.g. LF, CR, etc.)

1） Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|
| Trim_p | Delete whitespace characters from both ends of a string |  | Trim_p (Str:=,). |

2） Variables

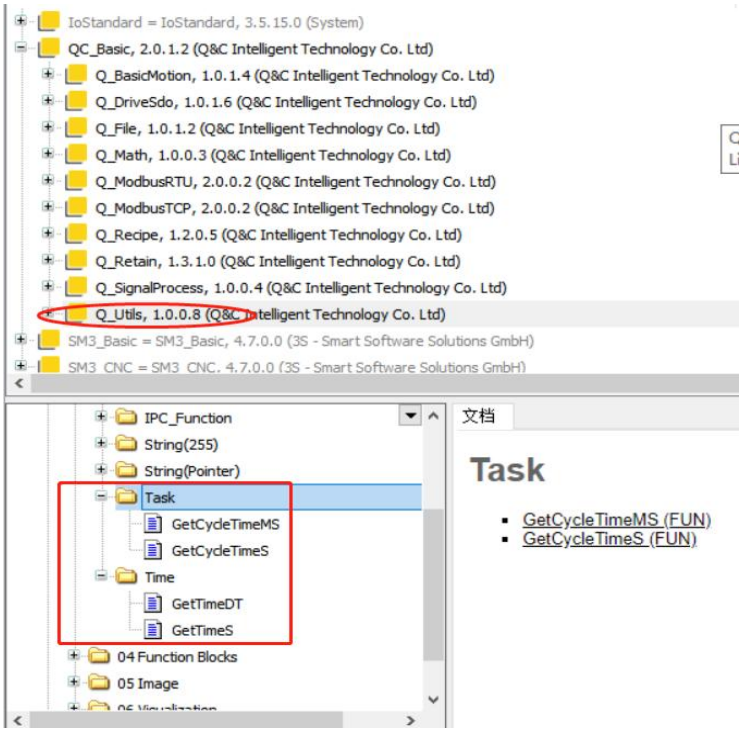
| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | ToUpper_p | INT | |
| Input | Str | POINTERTOBYTE | String address |

3） Example

```
(*declaration*)VAR
STR:STRING(255):='test1.2.3.4.5.trim'.
STR2:STRING(255):='test21.2.3.4.5.trim2';END_VAR
(*execution*)Trim_p(ADR(STR));//'test1.2.3.4.5.trim'---------
>'test1.2.3.4.5.trim'Trim_p(ADR(STR2));//'test21.2.3.4.5.trim2'--- ------>'test21.2.3.4.5.trim2'
```

# 3. 10 Time/Moment command

## 3. 10. 1 Library Manager



- **GetCycleTimeMS(FUN)**

Get the current task period, in ms (million second)

1） Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|
| GetCycleTimeMS | Get the current task period in ms |  | GetCycleTimeMS(). |

2） Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | GetCycleTimeMS | REAL | |

3） Example

- **GetCycleTimeS(FUN)**

Get the current task period, in s (second)

1） Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|
| GetCycle TimeS | Get the current task period in s |  | GetCycleTimeS( ). |

2） Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | GetCycleTimeS | REAL | |

3） Example

- **GetTimeDT(FUN)**

Get the local time for the time data type

1） Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|
| GetTimeDT | Get the local time for the time data type |  | GetTimeDT(). |

2） Related instructions

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | GetTimeDT | DATE_AND_TIME | |

3） Example

- **GetTimeS(FUN)**

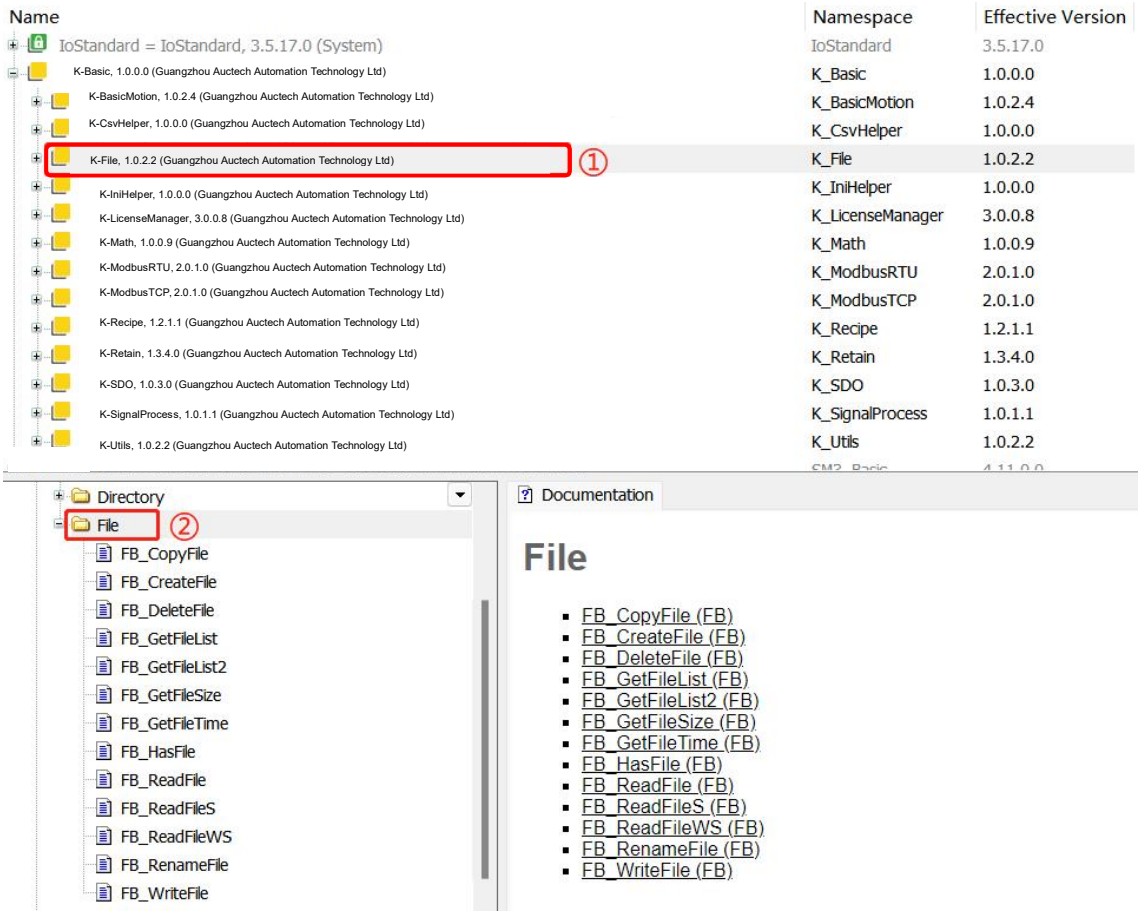
Get the local time of the string type

1） Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|
| GetTimeS | Get the local time of the string type |  | GetTimeS(). |

# 3. 11 File manipulation commands

## 3. 11. 1 Library Manager




- **CopyFile [FUN]**

Copy the file to the specified file path

1） Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|
| CopyFile | Copy files | CopyFile<br>sDestFileName STRING(255)<br>sSourceFileName STRING(255)<br>pulCopied POINTER TO __XWORD<br>SysFile.RTS_IEC_RESULT CopyFile | CopyFile(sDestFileName:=,sSourceFileName:=,pulCopied:=). |

2） Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | CopyFile | SysFile.RTS_IEC_RESULT | |
| Input | sDestFileName | STRING(255) | Destination file path name, the file name can contain an absolute or relative path to the file. Path entries |

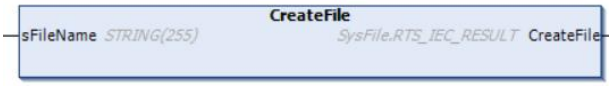
| Scope | Name | Type | Comment |
|---|---|---|---|
| | | | must be slash (/) separated, not backslash (\) separated! |
| | sSourceFileName | STRING(255) | The source file path name, the file name can contain the absolute or relative path to the file. Path entries must be slash (/) separated, not backslash (\) separated! |
| | pulCopied | POTNTERTO_XWORD | Returns the number of bytes copied |

3） Example

- **CreateFile [FUN]**

Create file

1） Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|
| CreateFile | Create file |  | CreateFile(sFileName:=). |

2） Variables

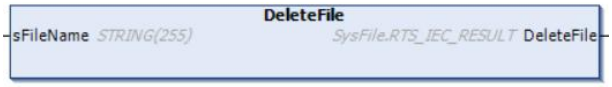
| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | CreateFile | SysFile.RTS_IEC_RESULT | |
| Input | sFileName | STRING(255) | File names can contain absolute or relative paths to files. Path entries must be slash (/) separated, not backslash (\) separated! |

3） Example

- **DeleteFile [FUN]**

Delete files

1） Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|
| DeleteFile | Delete files |  | DeleteFile(sFileName:=). |

2） Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | CreateFile | SysFile.RTS_IEC_RESULT | |
| Input | sFileName | STRING(255) | File names can contain absolute or relative paths to files. Path entries must be slash (/) separated, not backslash (\) separated! |

3） Example

- **GetFileList [FUN]**

Get file list (return file name only) function return value for the number of valid files obtained

1） Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|
| GetFileList | Get the text list |  | GetFileList(sPath:=,pList:=,pListFullName:=,ListSize:=,sSuffixName:=). |

2） Variables

| Scope | Name | Type | Comment |
|---|---|---|---|

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | GetFileList | UINT | |
| Input | sPath | STRING(255) | File paths, which can contain absolute or relative paths to files |
| | pList | POINTERTOSTRING(80) | File name list address |
| | pListFullName | POINTERTOSTRING(80) | Full name list address of the file |
| | ListSize | _XWORD | File name list byte size |
| | sSuffixName | STRING | Suffix name with ',', e.g. '.txt', case-insensitive |

3） Example

- **GetFileList2 [FUN]**

The return value of the Get File List (with file information) function is the number of valid files obtained

1） Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|
| Get FileList2 | Get the text list |  | GetFileList2(sPath:=,pList:=,Size:=,sSuffixName:=). |

2） Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | GetFileList | UINT | |
| Input | sPath | STRING | File paths, which can contain absolute or relative paths to files |
| | pList | POINTERTOTFileInfo | File name list address |
| | Size | _XWORD | File name list byte size |
| | sSuffixName | STRING | Suffix name with ',', e.g. '.txt', case-insensitive |

3） Example

- **HasFile [FUN]**

Check if the file exists

1） Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|
| HasFile | Check if the file exists |  | HasFile(sFileName:=). |

2） Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | HasFile | BOOL | |
| Input | sFileName | STRING(255) | The file name, which can contain the absolute or relative path to the file. Path entries must be split by a slash (/) and not separated by a backslash (\). |

3） Example

- **ReadFile [FUN]**

Read file function returns the number of bytes read from the file Note: If the read is Chinese characters need to use the following function to convert

1） Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|

| | | | |
|---|---|---|---|
| ReadFile | Read files |  | ReadFile(sFileName:=,). |

2） Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | sFileName | _XWORD | |
| Input | sFileName | STRING(255) | The file name, which can contain the absolute or relative path to the file. Path entries must be split by a slash (/) and not separated by a backslash (\). |

3） Example

- **ReadFileS [FUN]**

Read a line of data from a file, the return value is the number of bytes read

1） Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|
| ReadFileS | Reads a line of data from a file. |  | ReadFileS(str:=,sFileName:=, bResetRead:=). |

2） Variables

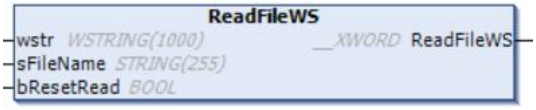
| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | ReadFileS | _XWORD | |
| In_Out | str | STRING(1000) | Read up buffer |
| Input | sFileName | STRING(255) | The file name, which can contain the absolute or relative path to the file. Path entries must be split by a slash (/) and not separated by a backslash (\). |
| | bResetRead | BOOL | Reset |

3） Example

- **ReadFileWS [FUN]**

Read a line of data from a file, the return value is the number of bytes read

1） Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|
| ReadFileWS | Reads a line of data from a file. |  | ReadFileWS(wstr:=,sFileName:=,bResetRead:=). |

2） Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | ReadFileWS | _XWORD | |
| In_Out | wstr | WSTRING(1000) | Read up buffer |
| Input | sFileName | STRING(255) | The file name, which can contain the absolute or relative path to the file. Path entries must be split by a slash (/) and not separated by a backslash (\). |
| | bResetRead | BOOL | Reset |

3） Example

- **WriteFile [FUN]**

Write data to file

1） Instructions

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|

| | | | |
|---|---|---|---|
| WriteFile | Write data to file |  | WriteFile(sFileName:= ,pbyBuffer:=, ulSize:=, WriteMode:=). |

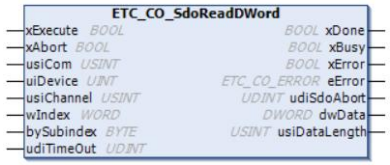
2） Variables

| Scope | Name | Type | Comment |
|---|---|---|---|
| Return | WriteFile | _XWORD | |
| Input | sFileName | STRING(255) | The file name, which can contain the absolute or relative path to the file. Path entries must be split by a slash (/) and not separated by a backslash (\). |
| | pbyBuffer | _XWORD | Address where the data is located, use ADR() to take the address |
| | ulSize | UDINT | Data length, number of bytes or string length |
| | WriteMode | EWriteMode | Write mode |

# 3. 12 EtherCAT Communication Commands

## 3. 12. 1 Read SDO parameter ETC_CO_SdoReadDword(FB)

1） Command format

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|
| ETC_CO_ SdoReadDword | Read SDO parameters |  | ETC_CO_SdoReadDword(x Execute:=,xAbort:=,usiCom: =, uiDevice:=. usiChannel:=. wIndex:=. bySubindex:=. udiTimeOut:=). |

2） Variables

a. Input

| Name | Data Type | Description |
|---|---|---|
| xExecute | BOOL | Rising edge: start reading slave parameters.<br>To release the internal channel again later, the instance must be called at least once by xExecute: = FALSE. |
| xAbort | BOOL | TRUE: the current read process is aborted. |
| usiCom | USINT | EtherCAT master number: If only one EtherCAT master is used, usiCom is always 1. If multiple masters are used, 1 specifies the first one, 2 the second one and so on. |
| uiDevice | UInt | The physical address of the slave.<br>If the auto-configuration mode is de-activated in the master, it is possible to provide the slave with its own address. This address must be specified here.<br>The current slave address can be checked in the Slave dialog in the Device Editor of the EtherCAT Address Area. |
| usiChannel | USINT | Reserved for future expansion |
| wIndex | WORD | Index of the parameters in the object directory. |
| bySubindex | BYTE | Subindex of parameters in the object directory. |
| udiTimeOut | UDINT | Definition of watchdog time, in milliseconds.<br>If the reading of the parameter has not been completed by the time this time expires, an error message is output. |
| pBuffer | CAA_PVOID | Pointer to the data buffer, which stores data after successful transfer of parameters |

| Name | Data Type | Description |
|---|---|---|
| szSize | CAA_SIZE | The size of the data buffer (pBuffer), in bytes. |

b. Output

| Name | Data Type | Description |
|---|---|---|
| xDone | BOOL | TRUE: no error in reading the completion parameters. |
| xBusy | BOOL | TRUE: reading is not complete. |
| xError | BOOL | TRUE: an error occurred during reading. |
| eError | ETC_CO_ERROR | Information about the cause of the error displayed by xError, such as ETC_CO_TIMEOUT on timeout |
| udiSdoAbort | UDINT | If an error occurs in the device, this output will provide more information about it. |
| szDataRead | CAA_SIZE | Number of bytes to read; max szSize(input). |

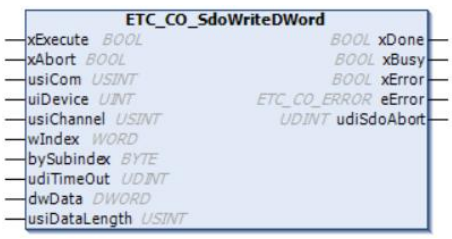
c. ENUMETC_CO_ERROR

| | | |
|---|---|---|
| ETC_CO_NO_ERROR | 0 | No errors |
| ETC_CO_FIRST_ERROR | 5750 | The cause of the error is stored in the output udiSdoAbort |
| ETC_CO_OTHER_ERROR | 5751 | Can't find the main site |
| ETC_CO_DATA_OVERFLOW | 5752 | ETC_CO_Expedited and size > 4 |
| ETC_CO_TIME_OUT | 5753 | Exceeding the time limit |
| ETC_CO_FIRST_MF | 5770 | Not used |
| ETC_CO_LAST_ERROR | 5799 | Not used |

3） Example

```
fb_SdoRead        :ARRAY[1..PARAM_NUMS]OFETC_CO_SdoReadDWord;//Sdo read module

// Program section
fb_SdoRead[i](
        xExecute:=SdoParam[i].bRead,//trigger signal
        xAbort:=,//Module interrupt signal, no input, default is false
        usiCom        :=1,//number of devices 1
uiDevice  := uiDevice,// the first driver of the slave module address is 1001, the second
1002, and so on
        usiChannel    :=,//Channel without input
        wIndex        :=SdoParam[i].wIndex,//index address, such as 16#603FErrorCode
        bySubindex    :=SdoParam[i].bySubindex,//wordindex
        udiTimeOut    :=5000000,//timeout time, unit us
        xDone         =>.
        xBusy=>.
        xError=>.
        eError=>.
        udiSdoAbort=>.
        dwData        =>SdoParam[i].wActValue,//the output value, dword type
        usiDataLength=>);4.12.2 Write SDO parameters ETC_CO_SdoWriteDword(FB)
```

4） Command format

| Instruction | Name | Graphical representation | ST Performance |
|---|---|---|---|
| ETC_CO_ SdoWriteDword | Write SDO parameters |  | ETC_CO_SdoWriteDword( xExecute:=. xAbort:=, the usiCom:=, the uiDevice:=. usiChannel:=. wIndex:=. bySubindex:=. udiTimeOut:=. dwData:=. usiDataLength:=). |

5） Variables

a. Input

| Name | Data Type | Description |
|---|---|---|
| xExecute | BOOL | Rising edge: Start reading slave parameters.<br>To release the internal channel again later, the instance must be called at least once by xExecute: = FALSE. |
| xAbort | BOOL | TRUE: the current writing process is aborted. |
| usiCom | USINT | EtherCAT master number: If only one EtherCAT master is used, usiCom is always 1. If multiple masters are used, "1" means the first one, "2" means the second one, and so on. |
| uiDevice | UInt | The physical address of the slave.<br>If the auto-configuration mode is de-activated in the master, it is possible to provide the slave with its own address. This address must be specified here<br>If the auto-configuration mode is activated, the first slave address is always 1001. The current slave address is always located in the Slave tab of the slave in the EtherCAT address area. |
| usiChannel | USINT | Reserved for future expansion |
| wIndex | WORD | Index of the parameters in the object directory. |
| bySubindex | Bytes | Subindex of parameters in the object directory. |
| udiTimeOut | UDINT | Definition of monitoring time, in milliseconds.<br>If the writing of the parameter has not been completed by the time this time expires, an error message is output. |
| dwData | dword | Contains the data to be written. |
| usiDataLength | USINT | The number of bytes to be written (1,2,4). |

b. Output

| Name | Data Type | Description |
|---|---|---|
| xDone | BOOL | TRUE: the writing of the parameter is complete and there are no errors. |
| xBusy | BOOL | TRUE: writing is not yet complete. |
| xError | BOOL | TRUE: an error occurred during writing. |
| eError | ETC_CO_ ERROR | Information about the cause of the error displayed by xError, such as ETC_CO_TIMEOUT on timeout. |
| udiSdoAbort | UDINT | If an error occurs in the device, this output will provide more information about it |

c. ENUMETC_CO_MODE

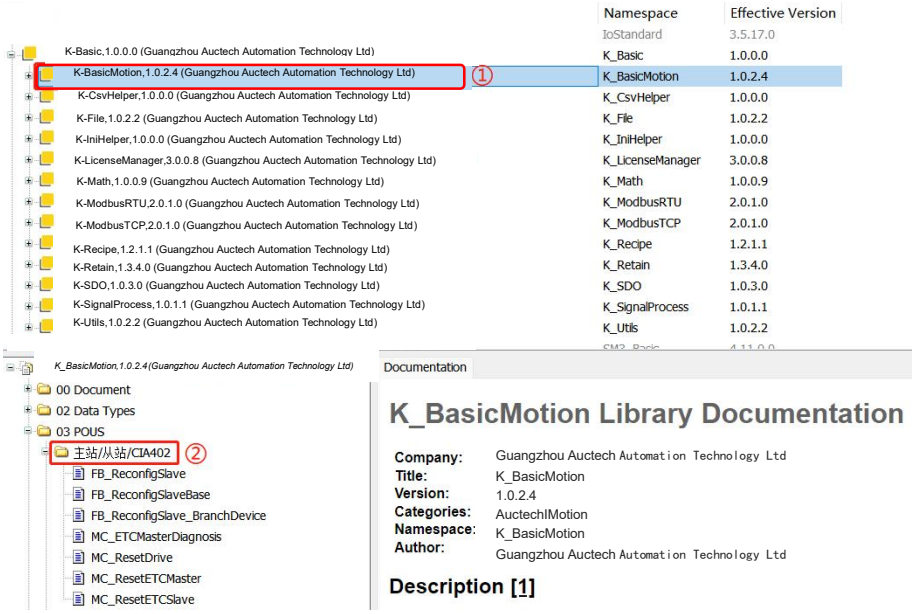
| | | |
|---|---|---|
| AUTO | 0 | Automatic client selection mode |
| EXPEDITED | 1 | Client use acceleration protocol |
| SEGMENTED | 2 | Client uses segmentation protocol |

6） Example

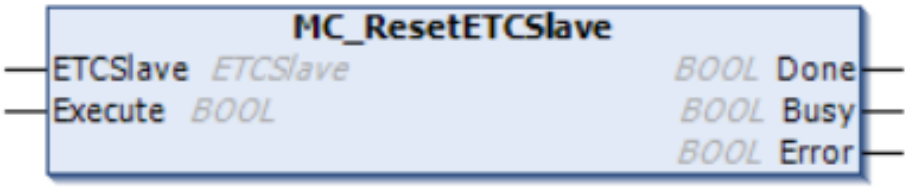
```
fb_SdoWrite        :ARRAY[1..PARAM_NUMS]OFETC_CO_SdoWriteDWord;//Sdo write module

fb_SdoWrite[i](
        xExecute        :=SdoParam[i].bWrite,//trigger signal
        xAbort:=,//Module interrupt signal, no input, default is false
        usiCom          :=1,//number of devices 1
        uiDevice        := uiDevice,// the first driver of the slave module address is 1001, the second
1002, and so on
        usiChannel:=,//Channel without input
        wIndex          :=SdoParam[i].wIndex,//index address, such as 16#603FErrorCode
        bySubindex      :=SdoParam[i].bySubindex,//wordindex
        udiTimeOut      :=5000000,//timeout time, unit us
        dwData          :=SdoParam[i].wSetValue.
        usiDataLength   := 2.
        xDone=>.
        xBusy=>.
```

xError=>.
eError=>.
udiSdoAbort=>).

## 3. 12. 2 Reset Module




### 3. 12. 2. 1 ETC module communication reset function block MC_ResetETCSlave




### 3. 12. 2. 2 Axis and drive reset function block MC_ResetDrive


### 3. 12. 2. 3 EtherCAT bus reset function block MC_ResetMaster

Reversion: V1.4

# HMC

AUCTECH AUTOMATION

Guangzhou Auctech Automation Technology Ltd

Hongshi Business Building, SCI-TECH Industry Park, Baiyun District, Guangzhou city, PRC

| Fax | Web | Mailbox |
| --- | --- | --- |
| +86 020 8489 8493 | www.auctech.com.cn | info@auctech.com.cn |