18. Macro Reference

This chapter describes the syntax, programming methods and usage of macro commands.

18.1. Overview 18-2
18.2. Instructions to use the Macro Editor 18-2
18.3. Configuration 18-7
18.4. Syntax 18-8
18.5. Statement 18-14
18.6. Function Blocks 18-19
18.7. Built-In Function Block 18-22
18.8. How to Create and Execute a Macro 18-87
18.9. User Defined Macro Function 18-91
18.10. Some Notes about Using the Macro 18-104
18.11. Use the Free Protocol to Control a Device 18-104
18.12. Compiler Error Message 18-109
18.13. Sample Macro Code 18-115
18.14. Macro TRACE Function 18-120
18.15. Example of String Operation Functions 18-125
18.16. Macro Password Protection 18-132
18.17. Reading / Writing CANbus Address Using Variable 18-133

18.1. Overview

Macros provide the additional functionality your application may need. Macros are automated sequences of commands that are executed at run-time. Macros allow you to perform tasks such as complex scaling operations, string handling, and user interactions with your projects. This chapter describes syntax, usage, and programming methods of macro commands.

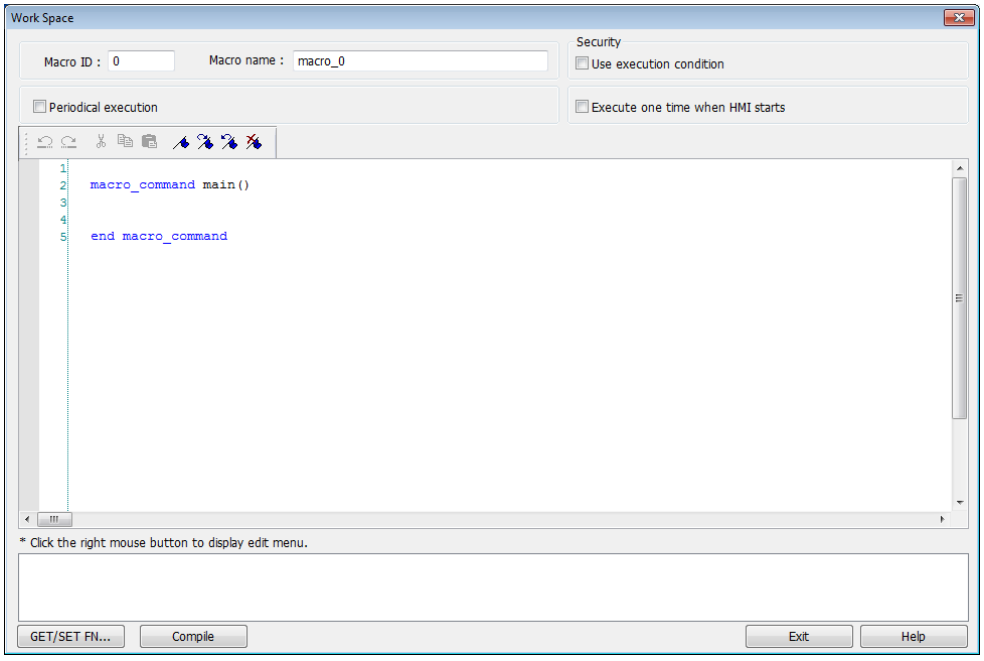
18.2. Instructions to use the Macro Editor

Macro editor provides the following functions:

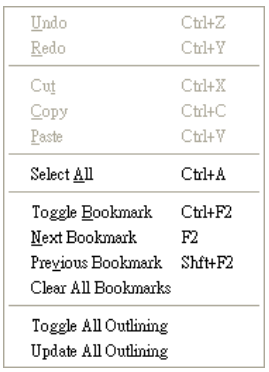
- Display line number
- Undo / Redo
- Cut / Copy / Paste
- Select All
- Toggle Bookmark / Previous Bookmark / Next Bookmark / Clear All Bookmarks
- Toggle All Outlining
- Security -> Use execution condition
- Periodical execution
- Execute one time when HMI starts

The instructions in the following part show you how to use these functions.

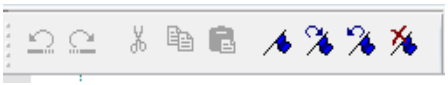
1. Open the macro editor; you'll see the line numbers displayed on the left-hand side of the edit area.



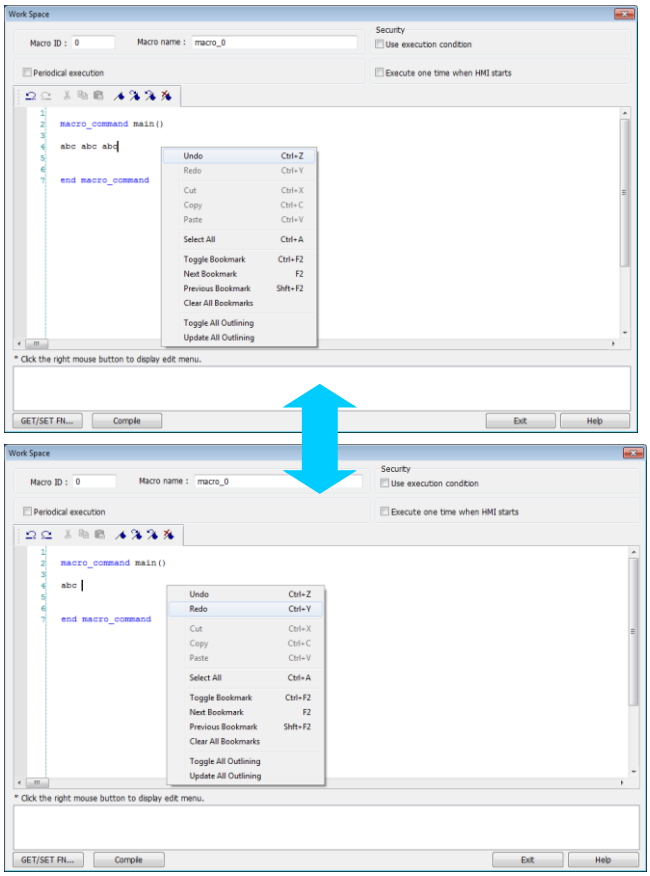
2. Right click on the edit area to open the pop-up menu as shown in the following figure. Disabled operations are colored grey, which indicates that it is not possible to use that function in the current status of the editor. For example, you should select some text to enable the copy function, otherwise it will be disabled. Keyboard shortcuts are also shown.



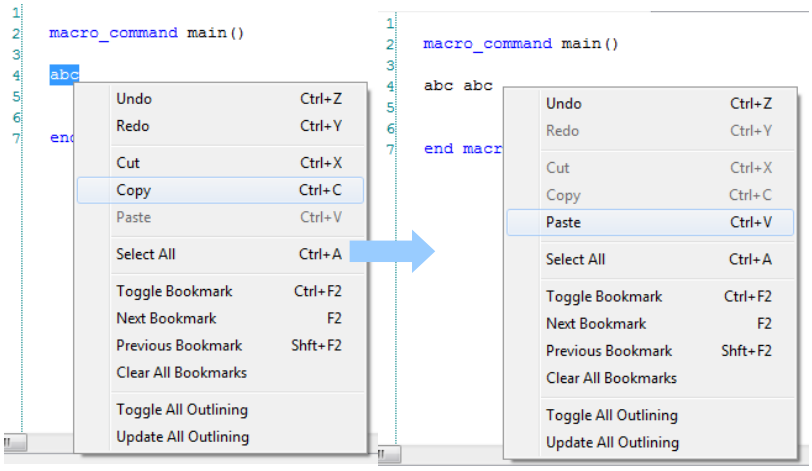
3. The toolbar provides [Undo], [Redo], [Cut], [Copy], [Paste], [Toggle Bookmark], [Next Bookmark], [Previous Bookmark] and [Clear All Bookmarks] buttons.



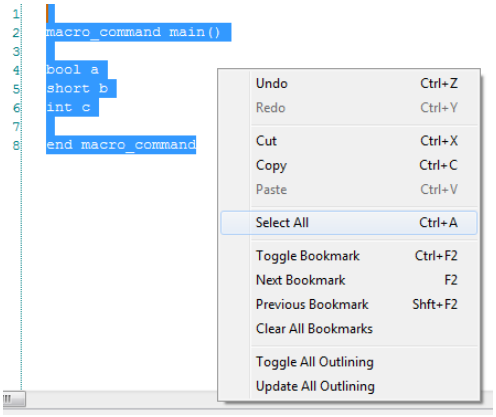
4. Any modification will enable the [Undo] function. [Redo] function will be enabled after the undo action is used. To perform the undo/redo, right click to select the item or use the keyboard shortcuts. (Undo: Ctrl+Z, Redo: Ctrl+Y).



5. Select a word in the editor to enable the [Cut] and [Copy] function. After [Cut] or [Copy] is performed, [Paste] function is enabled.

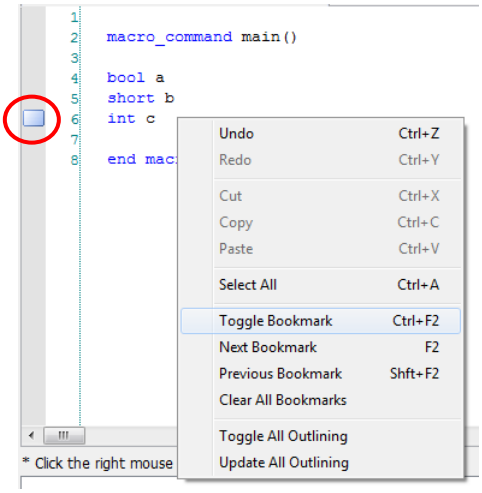


6. Use [Select All] to include all the content in the edit area.

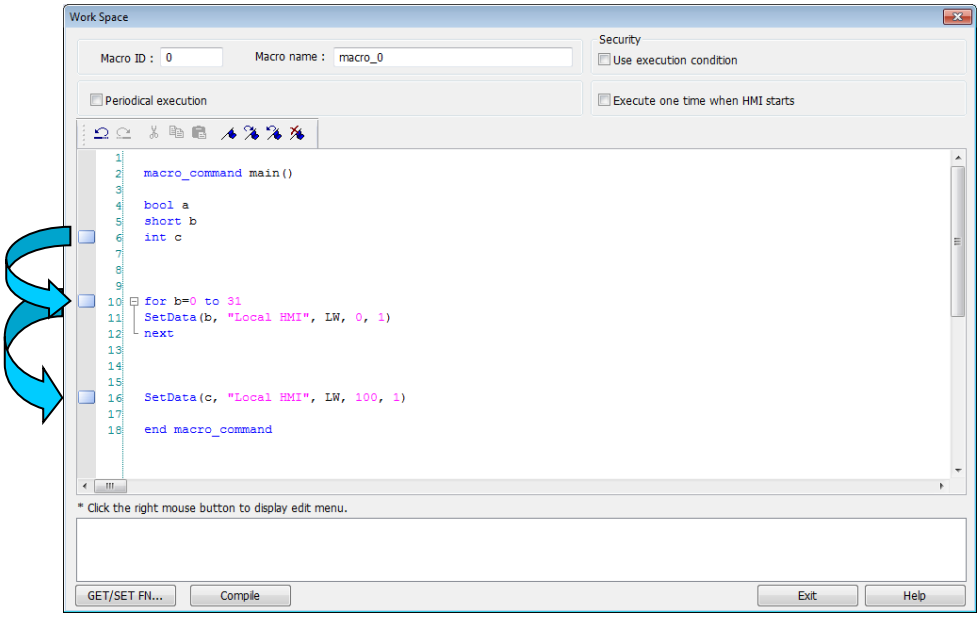


7. If the macro is too long, use bookmarks to manage and read the code with ease. The following illustration shows how it works.

- Move your cursor to the position in the edit area where to insert a bookmark. Right click, select [Toggle Bookmark]. There will be a blue little square that represents a bookmark on the left hand side of edit area.

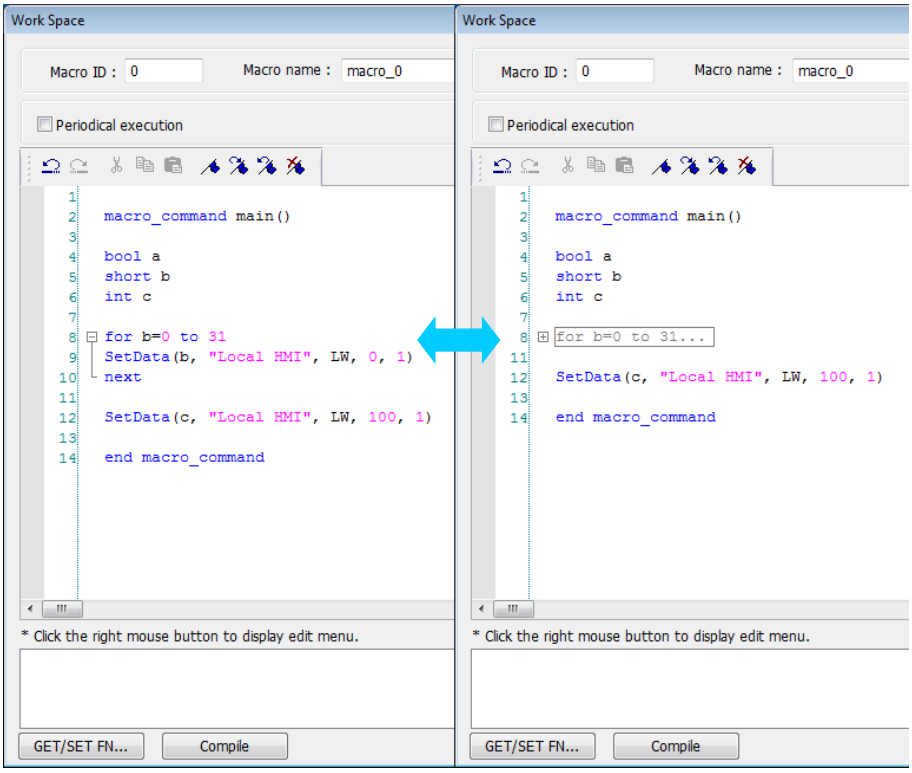


- If there is already a bookmark where the cursor is placed, select [Toggle Bookmark] to close it, otherwise to open it.
- Right click and select [Next Bookmark], the cursor will move to where the next bookmark locates. Selecting [Previous Bookmark] will move the cursor to the previous bookmark.

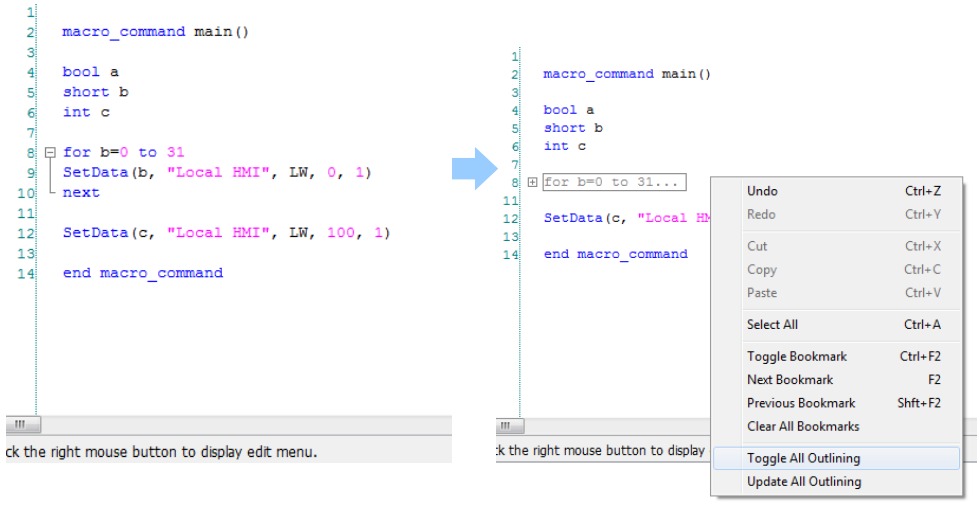


- Selecting [Clear All Bookmarks] will delete all bookmarks.

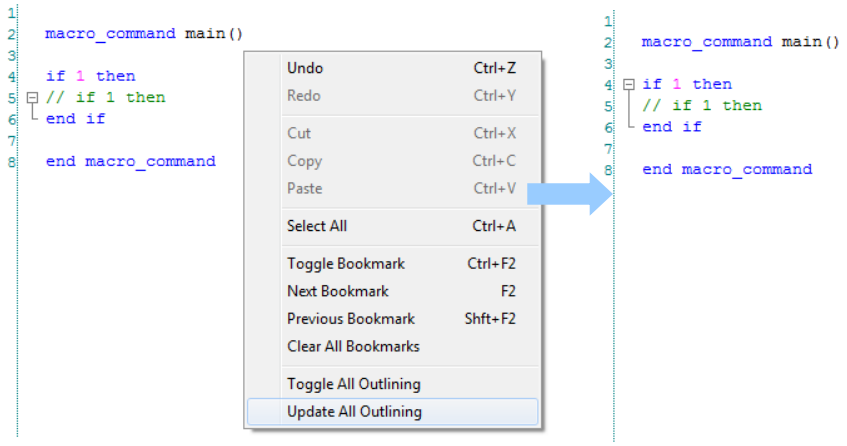
8. Macro editor provides outlining (or code-folding). Outlining will hide macro codes that belong to the same block, and display them as ⊞ . . . . There will be a tree diagram on the left hand side of edit area. Click ⊟ to hide the block or ⊞ to open, as shown in the following figure.



9. Right click to select [Toggle All Outlining] to open all folded macro code blocks.



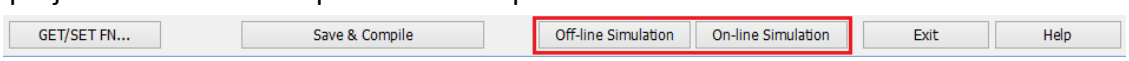
10. Sometimes the outlining might be incorrect since that the keywords are misjudged as shown in the following figure. To solve this problem, right click and select [Update All Outlining].



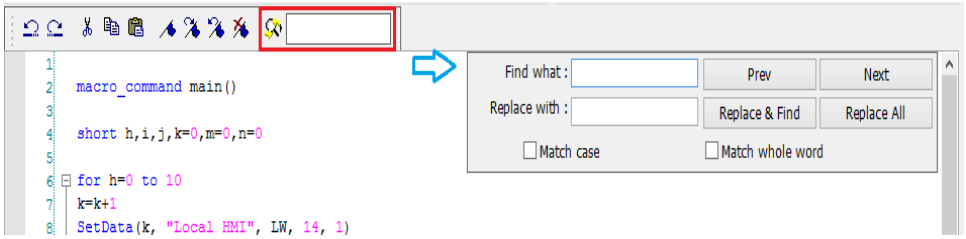
11. The statements enclosed in the following keywords are called a “block” of the macro code:

- Function block: sub – end sub
- Iterative statements:
  - i. for – next
  - ii. while – wend
- Logical statements:
  - i. if – end if
- Selective statements: select case – end select

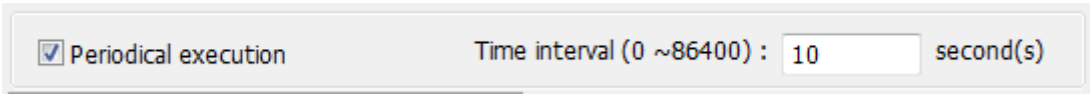
12. The macro editor is not a monopoly window. Returning to the main screen and editing the project with the Work Space window open is allowed.



13. The macro editor provides Find and Replace features.

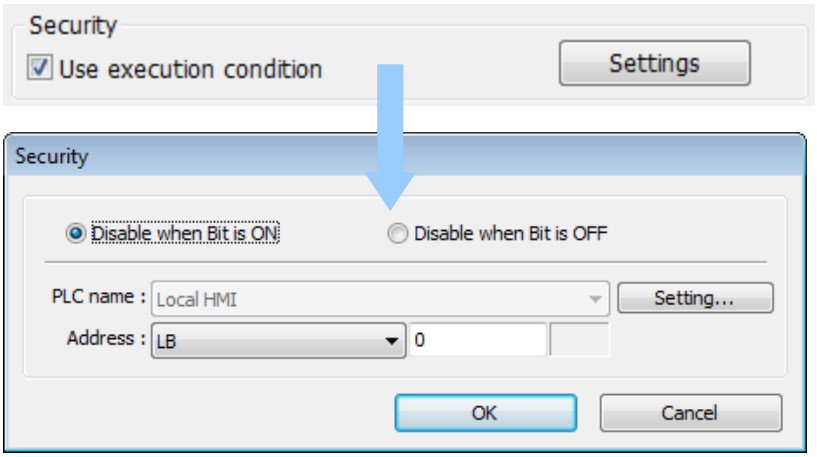


14. When [Periodical execution] is checked, this macro will be triggered periodically.



15. Select [Security] » [Use execution condition] » [Settings] to enable security settings:

- [Disable when Bit is ON]: When Bit is ON, this macro is disabled.
- [Disable when Bit is OFF]: When Bit is OFF, this macro is disabled.



16. Select [Execute one time when HMI starts], this macro will be executed once when HMI starts up.

18.3. Configuration

A macro contains statements. The statements contain constants, variables and operations. The statements are put in a specific order to create the desired output.

A macro has the following structure:

```
Global Variable Declaration          ---------------------------------- Optional

   Sub Function Block Declarations   ---------------------------------- Optional
        Local Variable Declarations
   End Sub

   macro_command main()              ---------------------------------- Required
        Local Variable Declarations
        [Statements]
   end macro_command                 ---------------------------------- Required
```

Macro must have one and only one main function which is the execution start point of macro. The format is:

macro_command main()

end macro_command

Local variables are used within the main macro function or in a defined function block. Its value remains valid only within the specific block.
Global variables are declared before any function blocks and are valid for all functions in the macro. When local variables and global variables have the same declaration of name, only the local variables are valid.

The following example shows a simple macro which includes a variable declaration and a function call.

```
macro_command main()
   short pressure = 10                                  //   local variable declaration
   SetData(pressure, "Allen-Bradley DF1", N7, 0, 1)  //   function calling
end macro_command
```

18.4. Syntax

18.4.1. Constants and Variables

18.4.1.1. Constants

Constants are fixed values and can be directly written into statements. The formats are:

Constant Type	Note	Example
Decimal integer		345, -234, 0, 23456
Hexadecimal	Must begin with 0x	0x3b, 0xffff, 0x237
ASCII	Single character must be enclosed in single quotation marks and a string (group of characters) must be enclosed in double quotation marks. A backslash \ can be used to escape the quotation marks contained in a string. Therefore, to enclose a string containing double quotation marks, please use \" for the double quotation mark in the string.	‘a’, "data", "name"
Boolean		true, false

Here is an example using constants:

```
macro_command main()
short A, B        //   A and B are variables
A = 1234
B = 0x12          //   1234 and 0x12 are constants
end macro_command
```

18.4.1.2. Variables

Variables are names that represent information. The information can be changed as the variable is modified by statements.

Naming Rules for Variables

- A variable name must start with an alphabet.
- Variable names longer than 32 characters are not allowed.
- Reserved words cannot be used as variable names.

There are 8 different Variable types, 5 for signed data types and 3 for unsigned data types:

Variable Type	Description	Range
bool (boolean)	1 bit (discrete)	0, 1
char (character)	8 bits (byte)	+127 to -128
short (short integer)	16 bits (word)	+32767 to -32768
int (integer)	32 bits (double word)	+2147483647to -2147483648
float (floating point)	32 bits (double word)	
unsigned char	8 bits (byte)	0 to 255
unsigned short (short	16 bits (word)	0 to 65535

integer)		
unsigned int	32 bits (double word)	0 to 4,294,967,295
long (long integer)	64 bits (four words) (cMT Series only)	+281474976710655 ~ -281474976710655
unsigned long (long integer)	64 bits (four words) (cMT Series only)	0 ~ 281474976710655
double	64 bits (four words) (cMT Series only)	

Declaring Variables

Variables must be declared before being used. To declare a variable, specify the type before the variable name.

Example:

```
int             a
short           b, switch
float           pressure
unsigned short  c
```

Declaring Arrays

Macros support one-dimensional arrays (zero-based index). To declare an array of variables, specify the type and the variable name followed by the number of variables in the array enclosed in brackets “[]”. The length of an array could be 1 to 4096. (Macros only support at most 4096 variables per macro).

Example:

```
int         a[10]
short       b[20], switch[30]
float       pressure[15]
```

The minimum array index is 0 and the maximum is (array size – 1).

Example:

```
char data [100]        //    array size is 100
```

In this case, the minimum of array index is 0 and maximum of array index is 99 (=100-1)

Variable and Array Initialization

There are two ways variables can be initialized:

- By statement using the assignment operator (=)

Example:

```
int a
```

float b[3]

a = 10

b[0] = 1

- During declaration

char a = '5', b = 9

The declaration of arrays is a special case. The entire array can be initialized during declaration by enclosing comma separated values inside curly brackets "{}".

Example:

float data[4] = {11, 22, 33, 44} // now data[0] is 11, data[1] is 22….

18.4.2. Operators

Operators are used to designate how data is manipulated and calculated.

Operator	Description	Example
=	Assignment operator	pressure = 10

Arithmetic Operators	Description	Example
+	Addition	A = B + C
-	Subtraction	A = B – C
*	Multiplication	A = B * C
/	Division	A = B / C
%	Modulo division (return remainder)	A = B % 5

By default, integer numbers (1, 2,3..etc) are considered having integer data type; therefore, when division is carried out involving two integer numbers where the result should have decimal point, the decimal part will be removed. To avoid this, add .0 (1.0, 2.0, 3.0...etc) behind the dividend or the divisor to turn it into a floating point number calculation.

Examples:

A = 3 / 2 = 1 » 3 and 2 are both integers; therefore the result is an integer.

B = 3 / 2.0 = 1.5 » 3 is an integer whereas 2.0 is a floating point number, therefore the result is a floating point number.

C = 3.0 / 2 = 1.5 » 3.0 is a floating point number whereas 2 is an integer, therefore the result is a floating point number.

Comparison Operators	Description	Example
<	Less than	if A < 10 then B = 5
<=	Less than or equal to	if A <= 10 then B = 5
>	Greater than	if A > 10 then B = 5
>=	Greater than or equal to	if A >= 10 then B = 5
==	Equal to	if A == 10 then B = 5

<>	Not equal to	if A <> 10 then B = 5

Logic Operators	Description	Example
and	Logical AND	if A < 10 and B > 5 then C = 10
or	Logical OR	if A >= 10 or B > 5 then C = 10
xor	Logical Exclusive OR	if A xor 256 then B = 5
not	Logical NOT	if not A then B = 5

Shift and bitwise operators are used to manipulate bits of signed/unsigned character and integer variables. The priority of these operators is from left to right within the statement.

Shift Operators	Description	Example
<<	Shifts the bits in a bitset to the left a specified number of positions	A = B << 8
>>	Shifts the bits in a bitset to the right a specified number of positions	A = B >> 8

Bitwise Operators	Description	Example
&	Bitwise AND	A = B & 0xf
|	Bitwise OR	A = B | C
^	Bitwise XOR	A = B ^ C
~	One's complement	A = ~B

Priority of All Operators

The overall priority of all operations from highest to lowest is as follows:

1. Operations within parenthesis are carried out first
2. Arithmetic operations
3. Shift and Bitwise operations
4. Comparison operations
5. Logic operations
6. Assignment

Reserved Keywords

The following keywords are reserved for system. These keywords cannot be used as variable, array, or function names.

+, -, *, /, %, >=, >, <=, <, <>, ==, and, or, xor, not, <<, >>,=, &, |, ^, ~

exit, macro_command, for, to, down, step, next, return, bool, short, int, char, float, void, if, then, else, break, continue, set, sub, end, while, wend, true, false
SQRT, CUBERT, LOG, LOG10, SIN, COS, TAN, COT, SEC, CSC, ASIN, ACOS, ATAN, BIN2BCD, BCD2BIN, DEC2ASCII, FLOAT2ASCII, HEX2ASCII, ASCII2DEC, ASCII2FLOAT, ASCII2HEX, FILL, RAND, DELAY, SWAPB, SWAPW, LOBYTE, HIBYTE, LOWORD, HIWORD, GETBIT, SETBITON, SETBITOFF, INVBIT, ADDSUM, XORSUM, CRC, CRC8, INPORT, OUTPORT, POW, GetCnvTagArrayIndex, GetError, GetData, GetDataEx, SetData, SetDataEx, SetRTS, GetCTS, Beep, SYNC_TRIG_MACRO, ASYNC_TRIG_MACRO, TRACE, FindDataSamplingDate, FindDataSamplingIndex, FindEventLogDate, FindEventLogIndex
StringGet, StringGetEx, StringSet, StringSetEx, StringCopy, StringMid, StringDecAsc2Bin, StringBin2DecAsc, StringDecAsc2Float, StringFloat2DecAsc, StringHexAsc2Bin, StringBin2HexAsc, StringLength, StringCat, StringCompare, StringCompareNoCase, StringFind, StringReverseFind, StringFindOneOf, StringIncluding, StringExcluding, StringToUpper, StringToLower, StringToReverse, StringTrimLeft, StringTrimRight, StringInsert, String2Unicode

18.5. Statement

18.5.1. Definition Statement

This covers the declaration of variables and arrays. The formal construction is as follows:

```
type        name
```

This defines a variable with name as “name” and type as “type”.
Example:
int A // define a variable A as an integer

```
type        name[constant]
```

This defines an array variable called “name” with size as “constant” and type as “type”.
Example:
int B[10] // where define a variable B as a one-dimensional array of size 10

18.5.2. Assignment Statement

Assignment statements use the assignment operator to move data from the expression on the right side of the operator to the variable on the left side. An expression is the combination of variables, constants and operators to yield a value.

```
VariableName    Expression
```

Example
A = 2 where a variable A is assigned to 2

18.5.3. Logical Statements

Logical statements perform actions depending on the condition of a boolean expression.
The syntax is as follows:

Single-Line Format

```
If <Condition> then
   [Statements]
else
   [Statements]
end if
```

Example:

```
if a == 2 then
        b = 1
else
        b = 2
end if
```

Block Format

```
If <Condition> then
   [Statements]
else if <Condition-n> then
[Statements]
else
   [Statements]
end if
```

Example:

```
if a == 2 then
        b = 1
else if a == 3 then
        b = 2
else
        b = 3
end if
```

Syntax description

if	Must be used to begin the statement.
<Condition>	Required. This is the controlling statement. It is FALSE when the <Condition> evaluates to 0 and TRUE when it evaluates to non- zero.
then	Must precede the statements to execute if the <Condition> evaluates to TRUE.
[Statements]	It is optional in block format but necessary in single-line format without else. The statement will be executed when the <Condition> is TRUE.
else if	Optional. The else if statement will be executed when the relative <Condition-n> is TRUE.
<Condition-n>	Optional. see <Condition>
else	Optional. The else statement will be executed when <Condition> and <Condition-n> are both FALSE.
end if	Must be used to end an if-then statement.

18.5.4. Selective Statements

The select-case construction can be used like multiple if-else statements and perform selected actions depending on the value of the given variable. When the matched value is found, all the actions below will be executed until a break statement is met. The syntax is as follows:

Format without a Default Case

```
Select Case [variable]
Case [value]
    [Statements]
    break
end Select
```

Example:

```
Select Case A
    Case 1
        b=1
    break
end Select
```

Format with a Default Case (Case else)

```
Select Case [variable]
Case [value]
    [Statements]
    break
    Case else
    [Statements]
    break

end Select
```

Example:

```
Select Case A
    Case 1
        b=1
    break
    Case else
        b=0
    break
end Select
```

Multiple cases in the same block

```
Select Case [variable]
Case [value1]
    [Statements]
Case [value2]
    [Statements]
    break

end Select
```

Example:

```
Select Case A
    Case 1
    break
    Case 2
        b=2
    break
    Case 3
        b=3
    break
end Select
```

Syntax description

Select Case	Must be used to begin the statement.
[variable]	Required. The value of this variable will be compared to the value of each case.
Case else	Optional. It represents the default case. If none of the cases above are matched, the statements under default case will be executed. When a default case is absent, it will skip directly to the end of the select-case statements if there is no matched case.
break	Optional. The statements under the matched case will be executed until the break command is reached. If a break command is absent, it simply keeps on executing next statement until the end command is reached.
end Select	Indicates the end of the select-case statements.

18.5.5. Iterative Statements

Iterative statements control loops and repetitive tasks depending on condition. There are two types of iterative statements.

18.5.5.1. for-next Statements

The for-next statement runs for a fixed number of iterations. A variable is used as a counter to track the progress and test for ending conditions. Use this for fixed execution counts. The syntax is as follows:

```
for [Conunter] = <StartValue> to <EndValue> [step <StepValue>]
    [Statements]
next [Counter]
```

Or

```
for [Conunter] = <StartValue> to <EndValue> [step <StepValue>]
    [Statements]
next [Counter]
```

Example:

```
for a = 0 to 10 step 2
    b = a
next a
```

Syntax description

for	Must be used to begin the statement
[Counter]	Required. This is the controlling statement. The result of evaluating the variable is used as a test of comparison.
<StartValue>	Required. The initial value of [Counter]
to/down	Required. This determines if the <step> increments or decrements the <Counter>. “to” increments <Counter> by <StepValue>. “down” decrements <Counter> by <StepValue>.
<EndValue>	Required. The test point. If the <Counter> is greater than this value, the macro exits the loop.
step	Optional. Specifies that a <StepValue> other than one is to be used.
[StepValue]	Optional. The increment/decrement step of <Counter>. It can be omitted when the value is 1 If [step <StepValue>] are omitted the step value defaults to 1.
[Statements]	Optional. Statements to execute when the evaluation is TRUE. “for-next” loops may be nested.
next	Required.
[Counter]	Optional. This is used when nesting for-next loops.

18.5.5.2. while-wend Statements

The while-wend statement runs for an unknown number of iterations. A variable is used to test for ending conditions. When the condition is TRUE, the statements inside are executed repetitively until the condition becomes FALSE. The syntax is as follows.

```
while <Condition>
        [Statements]
wend
```

Example:

```
while a < 10
    a = a + 10
wend
```

Syntax description

while	Must be used to begin the statement.
continue	Required. This is the controlling statement. When it is TRUE, the loop begins execution. When it is FALSE, the loop terminates.
return [value]	Statements to execute when the evaluation is TRUE.
wend	Indicates the end of the while-end statements.

18.5.5.3. Other Control Commands

break	Used in for-next and while-wend. It skips immediately to the end of the iterative statement.
continue	Used in for-next and while-wend. It ends the current iteration of a loop and starts the next one.
return	The return command inside the main block can force the macro to stop anywhere. It skips immediately to the end of the main block.

18.6. Function Blocks

Function blocks are useful for reducing repetitive codes. It must be defined before use and supports any variable and statement type. A function block could be called by putting its name followed by parameters in parenthesis. After the function block is executed, it returns the value to the caller function where it is used as an assignment value or as a condition. A return type is not required in function definition, which means that a function block does not have to return a value. The parameters can also be ignored in function definition while the function has no need to take any parameters from the caller. The syntax is as follows:

Function definition with return type

```
sub type <name> [(parameters)]
        Local variable declarations
        [Statements]
        [return [value]]
end sub
```

Example:

```
    sub int Add(int x, int y)
                int result
                result = x +y
                return result
            end sub

            macro_command main()
                int a = 10, b = 20, sum
                sum = Add(a, b)
            end macro_command
```

or:

```
    sub int Add()
                int result, x=10, y=20
                result = x +y
                return result
            end sub
            macro_command main()
                int    sum
                sum = Add()
            end macro_command
```

Function definition without return type

```
sub <name> [(parameters)]
        Local variable declarations
        [Statements]
end sub
```

Example:

```
    sub Add(int x, int y)
                int result
                result = x +y
```

```
    end sub

    macro_command main()
        int    a = 10, b = 20
        Add(a, b)
    end macro_command
```

or:

```
  sub Add()
        int result, x=10, y=20
        result = x +y
    end sub

    macro_command main()
        Add()
    end macro_command
```

Syntax description

sub	Must be used to begin the function block
type	Optional. This is the data type of value that the function returns. A function block is not always necessary to return a value.
(parameters)	Optional. The parameters hold values that are passed to the function. The passed parameters must have their type declared in the parameter field and assigned a variable name. For example: sub int MyFunction(int x, int y). x and y would be integers passed to the function. This function is called by a statement that looks similar to this: ret = MyFunction(456, pressure) where “pressure” must be integer according to the definition of function. Notice that the calling statement can pass hard coded values or variables to the function. After this function is executed, an integer values is return to ‘ret’.
Local variable declaration	Variables that are used in the function block must be declared first. This is in addition to passed parameters. In the above example x and y are variables that the function can used. Global variables are also available for use in function block.
[Statements]	Statements to execute
[return [value]]	Optional. Used to return a value to the calling statement. The value can be a constant or a variable. Return also ends function block execution. A function block is not always necessary to return a value, but, when the return type is defined in the beginning of the definition of function, the return command is needed.
end sub	Must be used to end a function block.

18.7. Built-In Function Block

EasyBuilder Pro has many built-in functions for retrieving and transferring data to the PLC, data management and mathematical functions.

18.7.1. Table of Functions

Please click on one of the function names in the table to see its details.

Function Name	Description
PLC Functions	
GetData	Receives data from the PLC.
GetDataEx	Receives data from the PLC and continues executing next command even if there's no response from PLC.
GetError	Gets an error code.
SetData	Sends data to the PLC.
SetDataEx	Sends data to the PLC and continues executing next command even if there's no response from PLC.
Free Protocol Functions	
GetCTS	Gets CTS signal of RS-232.
INPORT	Reads data from a COM port or Ethernet port.
INPORT2	Reads data from a COM port or Ethernet port and then wait until the designated time.
INPORT3	Reads data from a COM port or Ethernet port according to the specified data size.
OUTPORT	Sends out the specified data to a PLC or controller via a COM port or Ethernet port.
PURGE	Clears the input and output buffers associated with the COM port.
SetRTS	Raises or lowers the RTS signal of RS-232.
Process Control Functions	
ASYNC_TRIG_MACRO	Triggers the execution of a macro asynchronously in a running macro.
SYNC_TRIG_MACRO	Triggers the execution of a macro synchronously in a running macro. The current macro will pause until the end of execution of this called macro.
DELAY	Suspends the execution of the current macro for at least the specified interval (*time*).

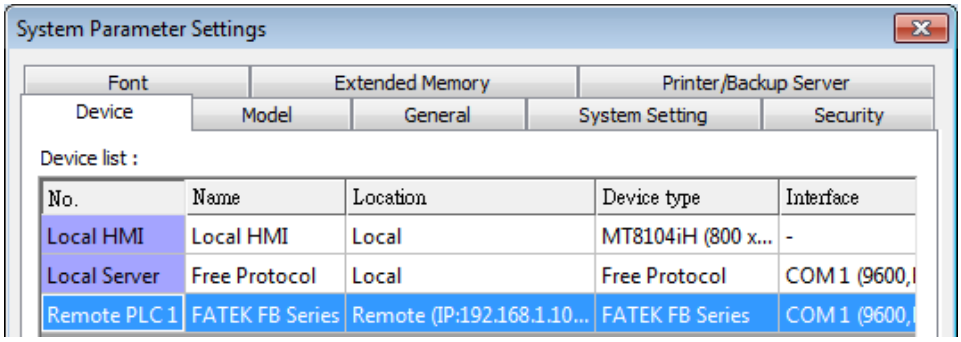
Data Operation Functions	
FILL	Sets array elements to the specified value.
SWAPB	Exchanges the high-byte and low-byte data of a 16-bit (Word).
SWAPW	Exchanges the high-word and low-word data of a 32-bit (DINT).
LOBYTE	Retrieves the low byte of a 16-bit source.
HIBYTE	Retrieves the high byte of a 16-bit source.
LOWORD	Retrieves the low word of a 32-bit source.
HIWORD	Retrieves the high word of a 32-bit source.
INVBIT	Inverts the state of designated bit position of a data source.
SETBITON	Changes the state of designated bit position of a data source to 1.
SETBITOFF	Changes the state of designated bit position of a data source to 0.
GETBIT	Gets the state of designated bit position of a data source.
Data Type Conversion Functions	
ASCII2DEC	Converts an ASCII string to a decimal value.
ASCII2FLOAT	Converts an ASCII string to a float value.
ASCII2HEX	Converts an ASCII string to a hexadecimal value.
BIN2BCD	Converts a binary-type value to a BCD-type value.
BCD2BIN	Converts a BCD-type value to a binary-type value.
DEC2ASCII	Converts a decimal value to an ASCII string.
FLOAT2ASCII	Converts a floating value to an ASCII string.
HEX2ASCII	Converts a hexadecimal value to an ASCII string.
StringDecAsc2Bin	Converts a decimal string to an integer.
StringBin2DecAsc	Converts an integer to a decimal string.
StringDecAsc2Float	Converts a decimal string to floats.
StringFloat2DecAsc	Converts a float to a decimal string.
StringHexAsc2Bin	Converts a hexadecimal string to binary data.
StringBin2HexAsc	Converts binary data to a hexadecimal string.
String Operation Functions	
String2Unicode	Converts all the characters in the source string to Unicode.
StringCat	Appends source string to destination string.
StringCompare	Does a case-sensitive comparison of two strings.
StringCompareNoCase	Does a case-insensitive comparison of two strings.
StringCopy	Copies one string to another.
StringExcluding	Retrieve a substring of the source string that contains characters that are not in the set string.

StringFind	Returns the zero-based index of the first character of substring in the source string that matches the target string.
StringFindOneOf	Returns the zero-based index of the first character in the source string that is also in the target string.
StringGet	Receives data from the PLC.
StringGetEx	Receives data from the PLC and continues executing next command even if there's no response from the PLC.
StringIncluding	Retrieve a substring of the source string that contains characters in the set string, beginning with the first character in the source string and ending when a character is found in the source string that is not in the target string.
StringInsert	Inserts a string in a specific location within the destination string content.
StringLength	Obtains the length of a string.
StringMid	Retrieves a character sequence from the specified offset of the source string.
StringReverseFind	Returns the position of the last occurrence of target string in the source string.
StringSet	Sends data to the PLC.
StringSetEx	Sends data to the PLC and continues executing next command even if there's no response from the PLC.
StringToUpper	Converts all the characters in the source string to uppercase characters.
StringToLower	Converts all the characters in the source string to lowercase characters.
StringToReverse	Reverses the characters in the source string
StringTrimLeft	Trims the leading specified characters in the set buffer from the source string.
StringTrimRight	Trims the trailing specified characters in the set buffer from the source string.
Mathematics Functions	
SQRT	Calculates the square root of source.
CUBERT	Calculates the cube root of source.
POW	Calculates the power of source.
SIN	Calculates the sine of source.
COS	Calculates the cosine of source.
TAN	Calculates the tangent of source.

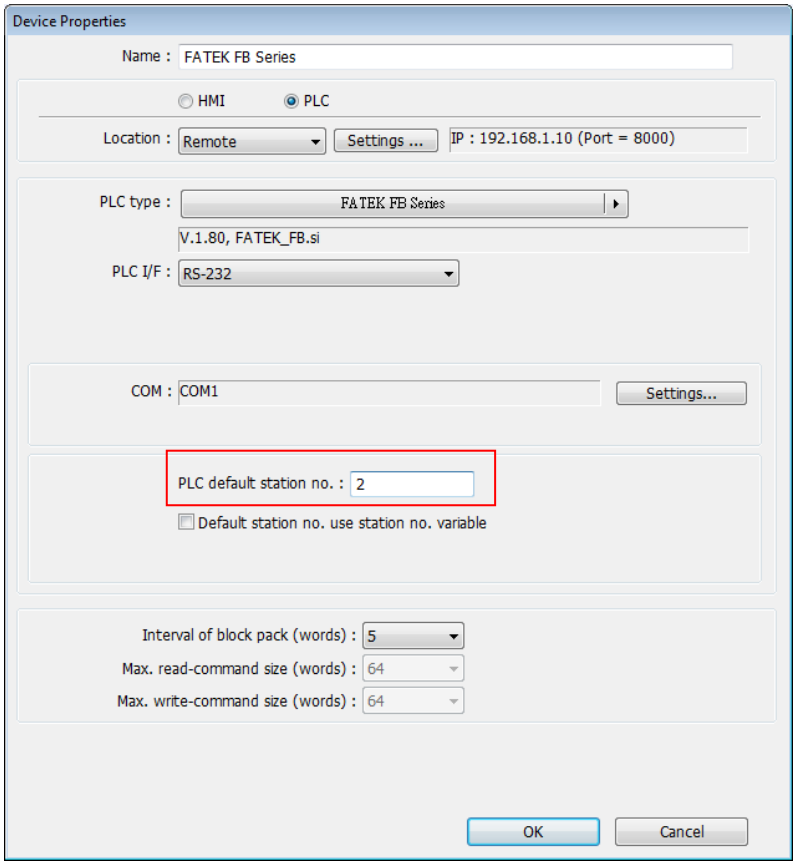
COT	Calculates the cotangent of source.
SEC	Calculates the secant of source
CSC	Calculates the cosecant of source.
ASIN	Calculates the arc sine of source.
ACOS	Calculates the arc cosine of source.
ATAN	Calculates the arc tangent of source.
LOG	Calculates the natural logarithm of a number.
LOG10	Calculates the base-10 logarithm of a number.
RAND	Calculates a random integer.
Statistics Functions	
AVERAGE	Gets the average value from array.
HARMEAN	Gets the harmonic mean value from array.
MAX	Gets the maximum value from array.
MEDIAN	Gets the median value from array.
MIN	Gets the minimum value from array.
STDEVP	Gets the standard deviation value from array.
STDEVS	Gets the sample standard deviation value from array.
Recipe Database Functions	
RecipeGetData	Gets recipe Data.
RecipeQuery	Queries recipe data.
RecipeQueryGetData	Gets the data in the query result obtained by RecipeQuery.
RecipeQueryGetRecordID	Gets the record ID numbers of those records gained by RecipeQuery.
RecipeSetData	Writes data to recipe database.
Data / Event Log Functions	
FindDataSamplingDate	Finds the date of the specified data sampling file.
FindDataSamplingIndex	Finds the file index of the specified data sampling file.
FindEventLogDate	Finds the date of the specified event log file.
FindEventLogIndex	Finds the file index of the specified event log file.
Checksum Functions	
ADDSUM	Adds up the elements of an array to generate a checksum.
XORSUM	Uses XOR to calculate the checksum.
BCC	Same as XORSUM.
CRC	Calculates 16-bit CRC of the variables to generate a checksum.
CRC8	Calculates 8-bit CRC of the variables to generate a checksum.
Miscellaneous Functions	
Beep	Plays beep sound.

Buzzer	Turns ON / OFF the buzzer.
TRACE	Prints out the current value of variables during run-time of macro for debugging.
GetCnvTagArrayIndex	When an user-defined conversion tag uses array, the [Read conversion] subroutine can get the relative array index before doing conversion.

18.7.2. PLC

Name	GetData
Syntax	GetData(*read_data*[*start*], *device_name, device_type, address_offset, data_count*) or GetData(*read_data, device_name, device_type, address_offset, 1*)
Description	Receives data from the PLC. Data is stored into *read_data*[*start*]~ *read_data*[*start + data_count - 1*]. *data_count* is the amount of received data. In general, *read_data* is an array, but if *data_count* is 1, *read_data* can be an array or an ordinary variable. Below are two methods to read one word data from PLC. macro_command main() short read_data_1[2], read_data_2 GetData(read_data_1[0], "FATEK KB Series", RT, 5, 1) GetData(read_data_2, "FATEK KB Series", RT, 5, 1) end macro_command *Device_name* is the PLC name enclosed in the double quotation marks (“) and this name has been defined in the device list of system parameters as follows (see FATEK KB Series):  *Device_type* is the device type and encoding method (binary or BCD) of the PLC data. For example, if *device_type* is LW_BIN, it means the register is LW and the encoding method is binary. If use BIN encoding method, "_BIN" can be ignored. If *device_type* is LW_BCD, it means the register is LW and the encoding method is BCD. *Address_offset* is the address offset in the PLC. For example, GetData(read_data_1[0], "FATEK KB Series", RT, 5, 1) represents that the address offset is 5.

If *address_offset* uses the format –"N#AAAAA", N indicates that PLC's station number is N. AAAAA represents the address offset. This format is used while multiple PLCs or controllers are connected to a single serial port. For example, GetData(read_data_1[0], "FATEK KB Series", RT, 2#5, 1) represents that the PLC's station number is 2. If GetData() uses the default station number defined in the device list as follows, it is not necessary to define station number in *address_offset*.



The number of registers actually read from depends on both the type of the *read_data* variable and the value of the number of *data_count*.

type of read_data	data_count	actual number of 16-bit register read
char (8-bit)	1	1
char (8-bit)	2	1
bool (8-bit)	1	1
bool (8-bit)	2	1
short (16-bit)	1	1
short (16-bit)	2	2
int (32-bit)	1	2
int (32-bit)	2	4
float (32-bit)	1	2
float (32-bit)	2	4

When a GetData() is executed using a 32-bit data type (int or float), the function will automatically convert the data. For example,

	macro_command main()
	float f
	GetData(f, "MODBUS", 6x, 2, 1) // f will contain a floating point value
	end macro_command

Example	macro_command main() bool a bool b_array[30] char c char c_array[20] short s short s_array[50] int i int i_array[10] float f float f_array[15]double g[10] // get the state of LB2 to the variable a GetData(a, "Local HMI", LB, 2, 1) // get 30 states of LB0 ~ LB29 to the variables b_array[0] ~ b_array[29] GetData(b_array[0], "Local HMI", LB, 0, 30) // get lower byte of LW-0 to the variable c // note that char is 1 byte, and a LW address occupies 2 bytes (1 word). Reading the first byte in a word register will get the lower byte of the word. // Ex: when the value in LW-0 is 0x0201, then variable c will read 0x01 GetData(c, "Local HMI", LW, 0, 1) // get data of LW1 ~ LW10 to the c_array[0] ~ c_array[19] GetData(c_array[0], "Local HMI", LB, 0, 20) // get one word from LW-2 to the variable s GetData(s, "Local HMI", LW, 2, 1) // get 50 words from LW-0 ~ LW-49 to the variables s_array[0] ~ s_array[49] GetData(s_array[0], "Local HMI", LW, 0, 50) // get 2 words from LW-6 ~ LW-7 to the variable e // Ex: When value in LW-6 is 0x0002, in LW-7 is 0x0001, then i will read 0x00010002(65538) // note that int occupies 2 words (32-bit) GetData(i, "Local HMI", LW, 6, 1) // get 20 words (10 integer values) from LW-0 ~ LW-19 to variables i_array[0] ~ i_array[9], note that type of i_array[10] is int.

	GetData(i_array[0], "Local HMI", LW, 0, 10)
	
	// get data from LW-10 ~ LW-11 to the variable f
	// note that type of variable f is float.
	GetData(f, "Local HMI", LW, 10, 1)
	
	// get 30 words (15 float variables) from LW-0 ~ LW-29 to variables f_array[0] ~ f_array[14], note that type of f_array[15] is float.
	// note that float occupies 2 words (32-bit)
	GetData(f_array[0], "Local HMI", LW, 0, 15)
	
	end macro_command

Name	GetDataEx
Syntax	GetDataEx(*read_data*[*start*], *device_name, device_type, address_offset, data_count*) or GetDataEx(*read_data, device_name, device_type, address_offset, 1*)
Description	Receives data from the PLC and continues executing next command even if there's no response from this device. Descriptions of *read_data, device_name, device_type, address_offset* and *data_count* are the same as GetData.

Example	macro_command main()
	bool a
	bool b
	bool b_array[30]
	char c
	char c_array[20]
	short s
	short s_array[50]
	int i
	int i_array[10]
	float f
	float f_array[15]
	
	// get the state of LB2 to the variable a
	GetDataEX(a, "Local HMI", LB, 2, 1)
	
	// get 30 states of LB0 ~ LB29 to the variables b_array[0] ~ b_array[29]
	GetDataEX(b_array[0], "Local HMI", LB, 0, 30)
	
	// get lower byte of LW-0 to the variable c
	// note that char is 1 byte, and a LW address occupies 2 bytes (1 word). Reading the first byte in a word register will get the lower byte of the word.
	// Ex: when the value in LW-0 is 0x0201, then variable c will read 0x01

	GetDataEX(c, "Local HMI", LW, 0, 1)
	
	// get data of LW1 ~ LW10 to the c_array[0] ~ c_array[19]
	GetDataEX(c_array[0], "Local HMI", LB, 0, 20)
	
	// get one word from LW-2 to the variable s
	GetDataEX(s, "Local HMI", LW, 2, 1)
	
	// get 50 words from LW-0 ~ LW-49 to the variables s_array[0] ~ s_array[49]
	GetDataEX(s_array[0], "Local HMI", LW, 0, 50)
	
	// get 2 words from LW-6 ~ LW-7 to the variable e
	// Ex: When value in LW-6 is 0x0002, in LW-7 is 0x0001, then i will read 0x00010002(65538)
	// note that int occupies 2 words (32-bit)
	GetDataEX(i, "Local HMI", LW, 6, 1)
	
	// get 20 words (10 integer values) from LW-0 ~ LW-19 to variables i_array[0] ~ i_array[9], note that type of i_array[10] is int.
	GetDataEX(i_array[0], "Local HMI", LW, 0, 10)
	
	// get data from LW-10 ~ LW-11 to the variable f
	// note that type of variable f is float.
	GetDataEX(f, "Local HMI", LW, 10, 1)
	
	// get 30 words (15 float variables) from LW-0 ~ LW-29 to variables f_array[0] ~ f_array[14], note that type of f_array[15] is float.
	// note that float occupies 2 words (32-bit)
	GetDataEX(f_array[0], "Local HMI", LW, 0, 15)
	
	end macro_command

Name	GetError
Syntax	GetError (*err*)
Description	Gets an error code.

Example	macro_command main()
	short err
	char byData[10]
	
	GetDataEx(byData[0], "MODBUS RTU", 4x, 1, 10)// read 10 bytes
	
	// if err is equal to 0, it is successful to execute GetDataEx()
	GetErr(err)// save an error code to err
	
	end macro_command

	Error code: 0: Normal 1: GetDataEx error 2: SetDataEx error

Name	SetData
Syntax	SetData(*send_data*[*start*], *device_name, device_type, address_offset, data_count*) or SetData(*send_data, device_name, device_type, address_offset, 1*)
Description	Sends data to the PLC. Data is defined in *send_data*[*start*]~ *send_data*[*start + data_count - 1*]. *data_count* is the amount of sent data. In general, *send_data* is an array, but if *data_count* is 1, *send_data* can be an array or an ordinary variable. Below are two methods to send one word data. macro_command main() short send_data_1[2] = { 5, 6}, send_data_2 = 5 SetData(send_data_1[0], "FATEK KB Series", RT, 5, 1) SetData(send_data_2, "FATEK KB Series", RT, 5, 1) end macro_command *device_name* is the PLC name enclosed in the double quotation marks (") and this name has been defined in the device list of system parameters. *device_type* is the device type and encoding method (binary or BCD) of the PLC data. For example, if *device_type* is LW_BIN, it means the register is LW and the encoding method is binary. If use BIN encoding method, "_BIN" can be ignored. If *device_type* is LW_BCD, it means the register is LW and the encoding method is BCD. *address_offset* is the address offset in the PLC. For example, SetData(read_data_1[0], "FATEK KB Series", RT, 5, 1) represents that the address offset is 5. If *address_offset* uses the format –"N#AAAAA", N indicates that PLC's station number is N. AAAAA represents the address offset. This format is used while multiple PLCs or controllers are connected to a single serial port. For example, SetData(read_data_1[0], "FATEK KB Series", RT, 2#5, 1) represents that the PLC's station number is 2. If SetData () uses the default station number defined in the device list, it is not necessary to define station number in *address_offset*. The number of registers actually sends to depends on both the type of the *send_data* variable and the value of the number of *data_count.*

type of *read_data*	*data_count*	actual number of 16-bit register send
char (8-bit)	1	1
char (8-bit)	2	1
bool (8-bit)	1	1
bool (8-bit)	2	1
short (16-bit)	1	1
short (16-bit)	2	2
int (32-bit)	1	2
int (32-bit)	2	4
float (32-bit)	1	2
float (32-bit)	2	4

When a SetData() is executed using a 32-bit data type (int or float), the function will automatically send int-format or float-format data to the device. For example,

```
macro_command main()
float f = 2.6
SetData(f, "MODBUS", 6x, 2, 1)      // will send a floating point value to the device
end macro_command
```

Example

```
macro_command main()
int i
bool a = true
bool b[30]
short c = false
short d[50]
int e = 5
int f[10]

for i = 0 to 29
b[i] = true
next i

for i = 0 to 49
d[i] = i * 2
next i

for i = 0 to 9
f [i] = i * 3
next i

//  set the state of LB2
SetData(a, "Local HMI", LB, 2, 1)
```

	// set the states of LB0 ~ LB29 SetData(b[0], "Local HMI", LB, 0, 30) // set the value of LW-2 SetData(c, "Local HMI", LW, 2, 1) // set the values of LW-0 ~ LW-49 SetData(d[0], "Local HMI", LW, 0, 50) // set the values of LW-6 ~ LW-7, note that the type of e is int SetData(e, "Local HMI", LW, 6, 1) // set the values of LW-0 ~ LW-19 // 10 integers equal to 20 words, since each integer value occupies 2 words. SetData(f[0], "Local HMI", LW, 0, 10) end macro_command

Name	SetDataEx
Syntax	SetDataEx (*send_data*[*start*]*, device_name, device_type, address_offset, data_count*) or SetDataEx (*send_data, device_name, device_type, address_offset, 1*)
Description	Sends data to the PLC and continues executing next command even if there's no response from this device. Descriptions of *send_data, device_name, device_type, address_offset* and *data_count* are the same as SetData.
Example	macro_command main() int i bool a = true bool b[30] short c = false short d[50] int e = 5 int f[10] for i = 0 to 29 b[i] = true next i for i = 0 to 49 d[i] = i * 2 next i

	for i = 0 to 9
	f [i] = i * 3
	next i
	
	// set the state of LB2
	SetDataEx (a, "Local HMI", LB, 2, 1)
	
	// set the states of LB0 ~ LB29
	SetDataEx (b[0], "Local HMI", LB, 0, 30)
	
	// set the value of LW-2
	SetDataEx (c, "Local HMI", LW, 2, 1)
	
	// set the values of LW-0 ~ LW-49
	SetDataEx (d[0], "Local HMI", LW, 0, 50)
	
	// set the values of LW-6 ~ LW-7, note that the type of e is int
	SetDataEx (e, "Local HMI", LW, 6, 1)
	
	// set the values of LW-0 ~ LW-19
	// 10 integers equal to 20 words, since each integer value occupies 2 words.
	SetDataEx (f[0], "Local HMI", LW, 0, 10)
	
	end macro_command

18.7.3. Free Protocol

Name	GetCTS
Syntax	GetCTS(*com_port, result*)
Description	Gets CTS state for RS232. *com_port* refers to the COM port number. It can be either a variable or a constant. *result* is used for receiving the CTS signal. It must be a variable. This command receives CTS signal and stores the received data in the *result* variable. When the CTS signal is pulled high, it writes 1 to *result*, otherwise, it writes 0.

Example	macro_command main()
	char com_port=1
	char result
	
	GetCTS(com_port, result) // get CTS signal of COM1
	
	GetCTS (1, result) // get CTS signal of COM1
	
	end macro_command

Name	INPORT
Syntax	INPORT(*read_data*[*start*], *device_name, read_count, return_value*)
Description	Reads data from a COM port or the Ethernet port. The data is stored to *read_data*[*start*]~ *read_data*[*start + read_count - 1*]. *device_name* is the name of a device defined in the device table and the device must be a “Free Protocol”-type device. *read_count* is the required amount of reading and can be a constant or a variable. If the function is used successfully to get sufficient data, *return_value* is 1, otherwise is 0.
Example	Below is an example of executing an action of reading holding registers of a MODBUS device. // Read Holding Registers macro_command main() char command[32], response[32] short address, checksum short read_no, return_value, read_data[2] FILL(command[0], 0, 32)// command initialization FILL(response[0], 0, 32) command[0] = 0x1// station no command[1] = 0x3// function code : Read Holding Registers address = 0 HIBYTE(address, command[2]) LOBYTE(address, command[3]) read_no = 2// read 2 words (4x_1 and 4x_2) HIBYTE(read_no, command[4]) LOBYTE(read_no, command[5]) CRC(command[0], checksum, 6) LOBYTE(checksum, command[6]) HIBYTE(checksum, command[7]) // send out a ‘Read Holding Registers” command OUTPORT(command[0], "MODBUS RTU Device", 8) // read responses for a ‘Read Holding Registers” command INPORT(response[0], "MODBUS RTU Device", 9, return_value)

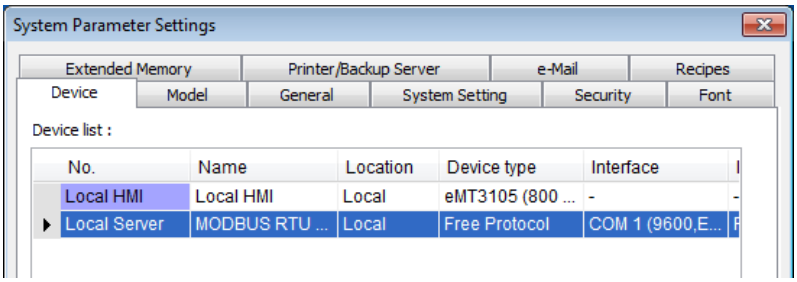
	if return_value > 0 then
	read_data[0] = response[4] + (response[3] << 8)// data in 4x_1
	read_data[1] = response[6] + (response[5] << 8)// data in 4x_2
	
	SetData(read_data[0], "Local HMI", LW, 100, 2)
	end if
	
	end macro_command

Name	INPORT2
Syntax	INPORT2(*response*[*start*], *device_name, receive_len, wait_time*)
Description	Reads data from a COM port or the Ethernet port. The data read will be saved in response. The description of *device_name* is the same as OUTPORT. *receive_len* stores the length of the data received, this must be a variable. *receive_len* total length can't exceed the size of response. *wait_time* (in millisecond) can be a constant or variable. After the data is read, if there's no upcoming data during the designated time interval, the function returns.

Example	macro_command main()
	
	short wResponse[6], receive_len, wait_time=20
	
	INPORT2(wResponse[0], "Free Protocol", receive_len, wait_time)
	// wait_time unit : millisecond
	
	if receive_len > 0 then
	SetData(wResponse[0], "Local HMI", LW, 0, 6)
	// set responses to LW0
	end if
	
	end macro_command

Name	INPORT3
Syntax	INPORT3(*response*[*start*], *device_name, read_count, receive_len*)
Description	Reads data from a communication port (COM Port or Ethernet Port). The data read will be saved in response. The amount of data to be read can be specified. The data that is not read yet will be stored in HMI buffer memory for the next read operation, in order to prevent losing data. The description of *device_name* is the same as OUTPORT. *read_count* stores the length of the data read each time. *receive_len* stores the length of the data received, this must be a variable. *receive_len* total length can't exceed the size of response.
Example	macro_command main() short wResponse[6], receive_len

	INPORT3(wResponse[0], "Free Protocol", 6, receive_len) // read 6 words
	
	if receive_len >= 6 then
	
	SetData(wResponse[0], "Local HMI", LW, 0, 6) // set responses to LW0
	
	end if
	
	end macro_command

Name	OUTPORT
Syntax	OUTPORT(*source*[*start*], *device_name, data_count*)
Description	Sends out the specified data from source[*start*] to source[*start + data_count -1*] to PLC via a COM port or an Ethernet port. *device_name* is the name of a device defined in the device table and the device must be a “Free Protocol”-type device. *data_count* is the amount of sent data and can be a constant or a variable.
Example	To use an OUTPORT function, a “Free Protocol” device must be created first as follows:  The device is named ”MODBUS RTU Device”. The port attribute depends on the setting of this device. (the current setting is “19200,E, 8, 1”) Below is an example of executing an action of writing single coil (SET ON) to a MODBUS device. macro_command main() char command[32] short address, checksum FILL(command[0], 0, 32)// command initialization command[0] = 0x1// station no command[1] = 0x5// function code : Write Single Coil address = 0 HIBYTE(address, command[2]) LOBYTE(address, command[3])

	command[4] = 0xff// force bit on command[5] = 0 CRC(command[0], checksum, 6) LOBYTE(checksum, command[6]) HIBYTE(checksum, command[7]) // send out a “Write Single Coil” command OUTPORT(command[0], "MODBUS RTU Device", 8) end macro_command

Name	PURGE
Syntax	PURGE (*com_port*)
Description	*com_port* refers to the COM port number which ranges from 1 to 3. It can be either a variable or a constant. This function is used to clear the input and output buffers associated with the COM port.
Example	macro_command main() int com_port=3 PURGE (com_port) PURGE (1) end macro_command

Name	SetRTS
Syntax	SetRTS(*com_port, source*)
Description	Sets RTS state for RS232. *com_port* refers to the COM port number. It can be either a variable or a constant. *source* can be either a variable or a constant. This command raise RTS signal while the value of source is greater than 0 and lower RTS signal while the value of *source* equals to 0.

Example	macro_command main()
	char com_port=1
	char value=1
	
	SetRTS(com_port, value) // raise RTS signal of COM1 while value>0
	
	SetRTS(1, 0) // lower RTS signal of COM1
	
	end macro_command

18.7.4. Process Control

Name	ASYNC_TRIG_MACRO
Syntax	ASYNC_TRIG_MACRO (*macro_id or name*)
Description	Triggers the execution of a macro asynchronously (use *macro_id or macro name* to designate this macro) in a running macro. The current macro will continue executing the following instructions after triggering the designated macro; in other words, the two macros will be active simultaneously. *macro_id* can be a constant or a variable.

Example	macro_command main()
	char ON = 1, OFF = 0
	
	SetData(ON, "Local HMI", LB, 0, 1)
	
	ASYNC_TRIG_MACRO(5)// call a macro (its ID is 5)
	
	ASYNC_TRIG_MACRO("macro_1") // call a macro (its name is macro_1)
	
	SetData(OFF, "Local HMI", LB, 0, 1)
	
	end macro_command

Name	DELAY
Syntax	DELAY(*time*)
Description	Suspends the execution of the current macro for at least the specified interval (*time*). The unit of *time* is millisecond. *time* can be a constant or a variable.

Example	macro_command main()
	int time == 500
	
	DELAY(100)// delay 100 ms
	DELAY(time)// delay 500 ms
	
	end macro_command

Name	SYNC_TRIG_MACRO
Syntax	SYNC_TRIG_MACRO(*macro_id or name*)
Description	Triggers the execution of a macro synchronously (use *macro_id* or macro name to designate this macro) in a running macro. The current macro will pause until the end of execution of this called macro. *macro_id* can be a constant or a variable.
Example	macro_command main() char ON = 1, OFF = 0

	SetData(ON, "Local HMI", LB, 0, 1)
	
	SYNC_TRIG_MACRO(5) // call a macro (its ID is 5)
	
	SYNC_TRIG_MACRO("macro_1") // call a macro (its name is macro_1)
	
	SetData(OFF, "Local HMI", LB, 0, 1)
	
	end macro_command

18.7.5. Data Operation

Name	FILL
Syntax	FILL(*source[start], preset, count*)
Description	Sets array elements from '*source[start]*' to '*source[start + count – 1]*' to the specified value (*preset*). *source* and *start* must be a variable, and *preset* can be a constant or variable.
Example	macro_command main() char result[4] char preset FILL(result[0], 0x30, 4) // result[0] is 0x30, result[1] is 0x30, , result[2] is 0x30, , result[3] is 0x30 preset = 0x31 FILL(result[0], preset, 2) // result[0] is 0x31, result[1] is 0x31 end macro_command

Name	SWAPB
Syntax	SWAPB(*source, result*)
Description	Exchanges the high-byte and low-byte data of a 16-bit *source* into *result*. *source* can be a constant or a variable. *result* must be a variable.

Example	macro_command main()
	short source, result
	
	SWAPB(0x5678, result)// result is 0x7856
	
	source = 0x123
	SWAPB(source, result)// result is 0x2301
	
	end macro_command

Name	SWAPW
Syntax	SWAPW(*source, result*)
Description	Exchanges the high-word and low-word data of a 32-bit *source* into *result*. *source* can be a constant or a variable. *result* must be a variable.

Example	macro_command main()
	int source, result
	
	SWAPW (0x12345678, result)// result is 0x56781234
	
	source = 0x12345
	SWAPW (source, result)// result is 0x23450001
	
	end macro_command

Name	LOBYTE
Syntax	LOBYTE(*source, result*)
Description	Retrieves the low byte of a 16-bit *source* into *result*. *source* can be a constant or a variable. *result* must be a variable.

Example	macro_command main()
	short source, result
	
	LOBYTE(0x1234, result)// result is 0x34
	
	source = 0x123
	LOBYTE(source, result)// result is 0x23
	
	end macro_command

Name	HIBYTE
Syntax	HIBYTE(*source, result*)
Description	Retrieves the high byte of a 16-bit *source* into *result*. *source* can be a constant or a variable. *result* must be a variable.

Example	macro_command main()
	short source, result
	
	HIBYTE(0x1234, result)// result is 0x12
	
	source = 0x123
	HIBYTE(source, result)// result is 0x01
	
	end macro_command

Name	LOWORD
Syntax	LOWORD(*source, result*)
Description	Retrieves the low word of a 32-bit source into result. *source* can be a constant or a variable. *result* must be a variable.

Example	macro_command main()
	int source, result
	
	LOWORD(0x12345678, result)// result is 0x5678
	
	source = 0x12345
	LOWORD(source, result)// result is 0x2345
	
	end macro_command

Name	HIWORD
Syntax	HIWORD(*source, result*)
Description	Retrieves the high word of a 32-bit source into result. *source* can be a constant or a variable. *result* must be a variable.

Example	macro_command main()
	int source, result
	
	HIWORD(0x12345678, result)// result is 0x1234
	
	source = 0x12345
	HIWORD(source, result)// result is 0x0001
	
	end macro_command

Name	INVBIT
Syntax	INVBIT(*source, result, bit_pos*)
Description	Inverts the state of designated bit position of a data (*source*), and puts changed data into *result*. *source* and *bit_pos* can be a constant or a variable. *result* must be a variable.

Example	macro_command main()
	int source, result
	short bit_pos
	
	INVBIT(4, result, 1)// result = 6
	
	source = 6
	bit_pos = 1
	INVBIT(source, result, bit_pos)// result = 4
	
	end macro_command

Name	SETBITON
Syntax	SETBITON(*source, result, bit_pos*)
Description	Changes the state of designated bit position of a data (*source*) to 1, and puts changed data into *result*. *source* and *bit_pos* can be a constant or a variable. *result* must be a variable.
Example	macro_command main() int source, result short bit_pos SETBITON(1, result, 3)// result is 9 source = 0

	bit_pos = 2 SETBITON (source, result, bit_pos)// result is 4 end macro_command

Name	SETBITOFF
Syntax	SETBITOFF(*source, result, bit_pos*)
Description	Changes the state of designated bit position of a data (*source*) to 0, and puts changed data into *result*. *source* and *bit_pos* can be a constant or a variable. *result* must be a variable.

Example	macro_command main()
	int source, result
	short bit_pos
	
	SETBITOFF(9, result, 3)// result is 1
	
	source = 4
	bit_pos = 2
	SETBITOFF(source, result, bit_pos)// result is 0
	
	end macro_command

Name	GETBIT
Syntax	GETBIT(*source, result, bit_pos*)
Description	Gets the state of designated bit position of a data (*source*) into *result*. *result* value will be 0 or 1. *source* and *bit_pos* can be a constant or a variable. *result* must be a variable.

Example	macro_command main()
	int source, result
	short bit_pos
	
	GETBIT(9, result, 3)// result is 1
	
	source = 4
	bit_pos = 2
	GETBIT(source, result, bit_pos)// result is 1
	
	end macro_command

18.7.6. Data Type Conversion

Name	ASCII2DEC
Syntax	ASCII2DEC(*source[start], result, len*)
Description	Transforms a string (*source*) into a decimal value saved to a variable (*result*). The length of the string is *len*. The first character of the string is *source[start]*. *source* and *len* can be a constant or a variable. *result* must be a variable. *start* must be a constant.
Example	macro_command main() char source[4] short result source[0] = '5' source[1] = '6' source[2] = '7' source[3] = '8' ASCII2DEC(source[0], result, 4) // result is 5678 end macro_command

Name	ASCII2FLOAT
Syntax	ASCII2FLOAT(*source[start], result, len*)
Description	Transforms a string (*source*) into a float value saved to a variable (*result*). The length of the string is *len*. The first character of the string is *source[start]*. *source* and *len* can be a constant or a variable. *result* must be a variable. *start* must be a constant.
Example	macro_command main() char source[4] float result source[0] = '5' source[1] = '6' source[2] = '.' source[3] = '8' ASCII2FLOAT (source[0], result, 4) // result is 56.8 end macro_command

Name	ASCII2HEX
Syntax	ASCII2HEX (*source[start], result, len*)
Description	Transforms a string (*source*) into a hexadecimal value saved to a variable (*result*). The length of the string is *len*. The first character of the string is *source[start]*.

	source and *len* can be a constant or a variable. *result* must be a variable. *start* must be a constant.
Example	macro_command main() char source[4] short result source[0] = '5' source[1] = '6' source[2] = '7' source[3] = '8' ASCII2HEX (source[0], result, 4) // result is 0x5678 end macro_command

Name	BIN2BCD
Syntax	BIN2BCD(*source, result*)
Description	Transforms a binary-type value (*source*) into a BCD-type value (*result*). *source* can be a constant or a variable. *result* must be a variable.

Example	macro_command main()
	
	short source, result
	
	BIN2BCD(1234, result)// result is 0x1234
	
	source = 5678
	BIN2BCD(source, result)// result is 0x5678
	
	end macro_command

Name	BCD2BIN
Syntax	BCD2BIN(*source, result*)
Description	Transforms a BCD-type value (*source*) into a binary-type value (*result*). source can be a constant or a variable. result must be a variable.

Example	macro_command main()
	
	short source, result
	
	BCD2BIN(0x1234, result)// result is 1234
	
	source = 0x5678
	BCD2BIN(source, result)// result is 5678
	
	end macro_command

Name	DEC2ASCII
Syntax	DEC2ASCII(*source, result[start], len*)
Description	Transforms a decimal value (*source*) into an ASCII string and saves it to an array (*result*). *len* represents the length of the string and the unit of length depends on result's type., i.e. if result's type is "char" (the size is byte), the length of the string is (byte * *len*). If result's type is "short" (the size is word), the length of the string is (word * *len*), and so on. The first character is put into *result[start]*, the second character is put into *result[start + 1]*, and the last character is put into *result[start + (len -1)]*. *source* and *len* can be a constant or a variable. *result* must be a variable. *start* must be a constant.

Example	macro_command main()
	short source
	char result1[4]
	short result2[4]
	char result3[6]
	source = 5678
	
	DEC2ASCII(source, result1[0], 4)
	// result1[0] is '5', result1[1] is '6', result1[2] is '7', result1[3] is '8'
	// the length of the string (result1) is 4 bytes(= 1 * 4)
	DEC2ASCII(source, result2[0], 4)
	// result2[0] is '5', result2[1] is '6', result2[2] is '7', result2[3] is '8'
	// the length of the string (result2) is 8 bytes(= 2 * 4)
	
	source=-123
	DEC2ASCII(source, result3[0], 6)
	// result1[0] is '-', result1[1] is '0', result1[2] is '0', result1[3] is '1'
	// result1[4] is '2', result1[5] is '3'
	// the length of the string (result1) is 6 bytes(= 1 * 6)
	
	end macro_command

Name	FLOAT2ASCII
Syntax	FLOAT2ASCII(*source, result[start], len*)
Description	Transforms a floating value (*source*) into ASCII string saved to an array (*result*). *len* represents the length of the string and the unit of length depends on result's type., i.e. if result's type is "char" (the size is byte), the length of the string is (byte * *len*). If result's type is "short" (the size is word), the length of the string is (word * *len*), and so on. *source* and len can be a constant or a variable. *result* must be a variable. start must be a constant.
Example	macro_command main()

	float source char result[4] source = 56.8 FLOAT2ASCII (source, result[0], 4) // result[0] is '5', result[1] is '6', result[2] is '.', result[3] is '8' end macro_command

Name	HEX2ASCII
Syntax	HEX2ASCII(*source, result[start], len*)
Description	Transforms a hexadecimal value (*source*) into ASCII string saved to an array (*result*). *len* represents the length of the string and the unit of length depends on result's type., i.e. if result's type is "char" (the size is byte), the length of the string is (byte * *len*). If result's type is "short" (the size is word), the length of the string is (word * *len*), and so on. *source* and *len* can be a constant or a variable. *result* must be a variable. *start* must be a constant.
Example	macro_command main() short source char result[4] source = 0x5678 HEX2ASCII (source, result[0], 4) // result[0] is '5', result[1] is '6', result[2] is '7', result[3] is '8' end macro_command

Name	StringDecAsc2Bin
Syntax	success = StringDecAsc2Bin(source[start], destination) or success = StringDecAsc2Bin("source", destination)
Description	This function converts a decimal string to an integer. It converts the decimal string in source parameter into an integer, and stores it in the destination variable. The source string parameter accepts both static string (in the form: "source") and char array (in the form: source[start]). Destination must be a variable, to store the result of conversion. This function returns a Boolean indicating whether the process is successfully done or not. If successful, it returns true, otherwise it returns false. The string can only contain these characters: +, -, and 0 to 9. If the string contains other characters, it returns false. The success field is optional.
Example	macro_command main()

	char src1[5]="12345"
	int result1
	bool success1
	success1 = StringDecAsc2Bin(src1[0], result1)
	// success1=true, result1 is 12345
	
	char src2[5] = "-6789"
	short result2
	bool success2
	success2 = StringDecAsc2Bin(src2[0], result2)
	// success2 = true，result2 is -6789
	
	char result3
	bool success3
	success3 = StringDecAsc2Bin("32768", result3)
	// success3=true, but the result exceeds the data range of result3
	
	char src4[2]="4b"
	char result4
	bool success4
	success4 = StringDecAsc2Bin (src4[0], result4)
	// success4=false, because src4 contains characters other than '+' or '-' and '0' to '9'
	
	end macro_command

Name	StringBin2DecAsc
Syntax	success = StringBin2DecAsc (source, destination[start])
Description	This function converts an integer to a decimal string. It converts the integer in source parameter into a decimal string, and stores it in the destination buffer. Source can be either a constant or a variable. Destination must be an one-dimensional char array, to store the result of conversion. This function returns a Boolean indicating whether the process is successfully done or not. If successful, it returns true, otherwise it returns false. If the length of decimal string after conversion exceeds the size of destination buffer, it returns false. The success field is optional.
Example	macro_command main() int src1 = 2147483647 char dest1[20] bool success1 success1 = StringBin2DecAsc(src1, dest1[0]) // success1=true, dest1="2147483647"

	short src2 = 0x3c
	char dest2[20]
	bool success2
	success2 = StringBin2DecAsc(src2, dest2[0])
	// success2=true, dest2="60"
	
	int src3 = 2147483647
	char dest3[5]
	bool success3
	success3 = StringBin2DecAsc(src3, dest3[0])
	// success3=false, dest3 remains the same.
	
	end macro_command

Name	StringDecAsc2Float
Syntax	success = StringDecAsc2Float (source[start], destination) or success = StringDecAsc2Float ("source", destination)
Description	This function converts a decimal string to floats. It converts the decimal string in source parameter into float, and stores it in the destination variable. The source string parameter accepts both static string (in the form: "source") and char array (in the form: source[start]). Destination must be a variable, to store the result of conversion. This function returns a Boolean indicating whether the process is successfully done or not. If successful, it returns true, otherwise it returns false. If the source string contains characters other than '0' to '9' or '.', it returns false. The success field is optional.

Example	macro_command main()
	char src1[10]="12.345"
	float result1
	bool success1
	success1 = StringDecAsc2Float(src1[0], result1)
	// success1=true, result1 is 12.345
	
	float result2
	bool success2
	success2 = StringDecAsc2Float("1.234567890", result2)
	// success2=true, but the result exceeds the data range of result2, which
	// might result in loss of precision
	
	char src3[2]="4b"
	float result3
	bool success3
	success3 = StringDecAsc2Float(src3[0], result3)
	// success3=false, because src3 contains characters other than '0' to '9' or

	// ‘’ end macro_command

Name	StringFloat2DecAsc
Syntax	success = StringFloat2DecAsc(source, destination[start])
Description	This function converts a float to a decimal string. It converts the float in source parameter into a decimal string, and stores it in the destination buffer. Source can be either a constant or a variable. Destination must be an one-dimensional char array, to store the result of conversion. This function returns a Boolean indicating whether the process is successfully done or not. If successful, it returns true, otherwise it returns false. If the length of decimal string after conversion exceeds the size of destination buffer, it returns false. The success field is optional.

Example	macro_command main()
	float src1 = 1.2345
	char dest1[20]
	bool success1
	success1 = StringFloat2DecAsc(src1, dest1[0])
	// success1=true, dest1="1.2345"
	
	float src2 = 1.23456789
	char dest2 [20]
	bool success2
	success2 = StringFloat2DecAsc(src2, dest2 [0])
	// success2=true, but it might lose precision
	
	float src3 = 1.2345
	char dest3[5]
	bool success3
	success3 = StringFloat2DecAsc(src3, dest3 [0])
	// success3=false, dest3 remains the same.
	
	end macro_command

Name	StringHexAsc2Bin
Syntax	success = StringHexAsc2Bin (source[start], destination) or success = StringHexAsc2Bin ("source", destination)
Description	This function converts a hexadecimal string to binary data. It converts the hexadecimal string in source parameter into binary data, and stores it in the destination variable. The source string parameter accepts both static string (in the form: "source") and char array (in the form: source[start]).

	Destination must be a variable, to store the result of conversion. This function returns a Boolean indicating whether the process is successfully done or not. If successful, it returns true, otherwise it returns false. If the source string contains characters other than '0' to '9', 'a' to 'f' or 'A' to 'F', it returns false. The success field is optional.

Example	macro_command main()
	char src1[5]="0x3c"
	int result1
	bool success1
	success1 = StringHexAsc2Bin(src1[0], result1)
	// success1=true, result1 is 3c
	
	short result2
	bool success2
	success2 = StringDecAsc2Bin("1a2b3c4d", result2)
	// success2=true, result2=3c4d.The result exceeds the data range of
	// result2
	
	char src3[2]="4g"
	char result3
	bool success3
	success3 = StringDecAsc2Bin (src3[0], result3)
	// success3=false, because src3 contains characters other than '0' to '9'
	// , 'a' to 'f' or 'A' to 'F'
	
	end macro_command

Name	StringBin2HexAsc
Syntax	success = StringBin2HexAsc (source, destination[start])
Description	This function converts binary data to a hexadecimal string. It converts the binary data in source parameter into a hexadecimal string, and stores it in the destination buffer. Source can be either a constant or a variable. Destination must be an one-dimensional char array, to store the result of conversion. This function returns a Boolean indicating whether the process is successfully done or not. If successful, it returns true, otherwise it returns false. If the length of hexadecimal string after conversion exceeds the size of destination buffer, it returns false. The success field is optional. Please note that this function cannot convert negative values.
Example	macro_command main() int src1 = 20 char dest1[20]

```
bool success1
success1 = StringBin2HexAsc(src1, dest1[0])
// success1=true, dest1="14"

short src2 = 0x3c
char dest2[20]
bool success2
success2 = StringBin2HexAsc(src2, dest2[0])
// success2=true, dest2="3c"

int src3 = 0x1a2b3c4d
char dest3[6]
bool success3
success3 = StringBin2HexAsc(src3, dest3[0])
// success3=false, dest3 remains the same.

end macro_command
```

Click the icon to download the demo project. Please confirm your internet connection before downloading the demo project.

18.7.7. String Operation

Name	String2Unicode
Syntax	result = String2Unicode("source", destination[start])
Description	Converts all the characters in the source string to Unicode and stores the result in the destination buffer. The length of result string after conversion will be stored to result. Source must be a constant but not a variable.

Example	macro_command main()
	
	char dest[20]
	int result
	result = String2Unicode("abcde", dest[0]) // "result" will be set to 10.
	result = String2Unicode("abcdefghijklmno", dest[0]) // "result" will be set to 20.
	// "result" will be the length of converted Unicode string
	
	end macro_command

Name	StringCat
Syntax	success = StringCat (source[start], destination[start]) or success = StringCat ("source", destination[start])
Description	This function appends source string to destination string. It adds the contents of source string to the last of the contents of destination string.

	The source string parameter accepts both static string (in the form: "source") and char array (in the form: source[start]). Destination must be an one-dimensional char array. This function returns a Boolean indicating whether the process is successfully done or not. If successful, it returns true, otherwise it returns false. If the length of result string after concatenation exceeds the max. size of destination buffer, it returns false. The success field is optional.

Example	macro_command main()
	char src1[20]="abcdefghij"
	char dest1[20]="1234567890"
	bool success1
	success1= StringCat(src1[0], dest1[0])
	// success1=true, dest1="123456790abcdefghij"
	
	char dest2 [10]="1234567890"
	bool success2
	success2= StringCat("abcde", dest2 [0])
	// success2=false, dest2 remains the same.
	
	char src3[20]="abcdefghij"
	char dest3[20]
	bool success3
	success3= StringCat(src3[0], dest3[15])
	// success3=false, dest3 remains the same.
	
	end macro_command

Name	StringCompare
Syntax	ret = StringCompare (str1[start], str2[start]) ret = StringCompare ("string1", str2[start]) ret = StringCompare (str1[start], "string2") ret = StringCompare ("string1", "string2")
Description	Does a case-sensitive comparison of two strings. The two string parameters accept both static string (in the form: "string1") and char array (in the form: str1[start]). This function returns a Boolean indicating the result of comparison. If two strings are identical, it returns true. Otherwise it returns false. The ret field is optional.
Example	macro_command main() char a1[20]="abcde" char b1[20]="ABCDE" bool ret1 ret1= StringCompare(a1[0], b1[0]) // ret1=false

	char a2[20]="abcde"
	char b2[20]="abcde"
	bool ret2
	ret2= StringCompare(a2[0], b2[0])
	// ret2=true
	
	char a3 [20]="abcde"
	char b3[20]="abcdefg"
	bool ret3
	ret3= StringCompare(a3[0], b3[0])
	// ret3=false
	
	end macro_command

Name	StringCompareNoCase
Syntax	ret = StringCompareNoCase(str1[start], str2[start]) ret = StringCompareNoCase("string1", str2[start]) ret = StringCompareNoCase(str1[start], "string2") ret = StringCompareNoCase("string1", "string2")
Description	Does a case-insensitive comparison of two strings. The two string parameters accept both static string (in the form: "string1") and char array (in the form: str1[start]). This function returns a Boolean indicating the result of comparison. If two strings are identical, it returns true. Otherwise it returns false. The ret field is optional.

Example	macro_command main()
	char a1[20]="abcde"
	char b1[20]="ABCDE"
	bool ret1
	ret1= StringCompareNoCase(a1[0], b1[0])
	// ret1=true
	
	char a2[20]="abcde"
	char b2[20]="abcde"
	bool ret2
	ret2= StringCompareNoCase(a2[0], b2[0])
	// ret2=true
	
	char a3 [20]="abcde"
	char b3[20]="abcdefg"
	bool ret3
	ret3= StringCompareNoCase(a3[0], b3[0])
	// ret3=false

	end macro_command

Name	StringCopy
Syntax	success = StringCopy ("source", destination[start]) or success = StringCopy (source[start], destination[start])
Description	Copies one string to another. This function copies a static string (which is enclosed in quotes) or a string that is stored in an array to the destination buffer. The source string parameter accepts both static string (in the form: "source") and char array (in the form: source[start]). destination[start] must be an one-dimensional char array. This function returns a Boolean indicating whether the process is successfully done or not. If successful, it returns true, otherwise it returns false. If the length of source string exceeds the max. size of destination buffer, it returns false and the content of destination remains the same. The success field is optional.

Example	macro_command main()
	char src1[5]="abcde"
	char dest1[5]
	bool success1
	success1 = StringCopy(src1[0], dest1[0])
	// success1=true, dest1="abcde"
	
	char dest2[5]
	bool success2
	success2 = StringCopy("12345", dest2[0])
	// success2=true, dest2="12345"
	
	char src3[10]="abcdefghij"
	char dest3[5]
	bool success3
	success3 = StringCopy(src3[0], dest3[0])
	// success3=false, dest3 remains the same.
	
	char src4[10]="abcdefghij"
	char dest4[5]
	bool success4
	success4 = StringCopy(src4[5], dest4[0])
	// success4=true, dest4="fghij"
	
	end macro_command

Name	StringExcluding
Syntax	success = StringExcluding (source[start], set[start], destination[start]) success = StringExcluding ("source", set[start], destination[start]) success = StringExcluding (source[start], "set", destination[start]) success = StringExcluding ("source", "set", destination[start])
Description	Retrieves a substring of the source string that contains characters that are not in the set string, beginning with the first character in the source string and ending when a character is found in the source string that is also in the target string. The source string and set string parameters accept both static string (in the form: "source") and char array (in the form: source[start]). This function returns a Boolean indicating whether the process is successfully done or not. If successful, it returns true, otherwise it returns false. If the length of retrieved substring exceeds the size of destination buffer, it returns false.

Example	macro_command main()
	char src1[20]="cabbageabc"
	char set1[20]="ge"
	char dest1[20]
	bool success1
	success1 = StringExcluding(src1[0], set1[0], dest1[0])
	// success1=true, dest1="cabba"
	
	char src2[20]="cabbage"
	char dest2[20]
	bool success2
	success2 = StringExcluding(src2[0], "abc", dest2[0])
	// success2=true, dest2=""
	
	char set3[20]="ge"
	char dest3[4]
	bool success3
	success3 = StringExcluding("cabbage", set3[0], dest3[0])
	// success3=false, dest3 remains the same.
	
	end macro_command

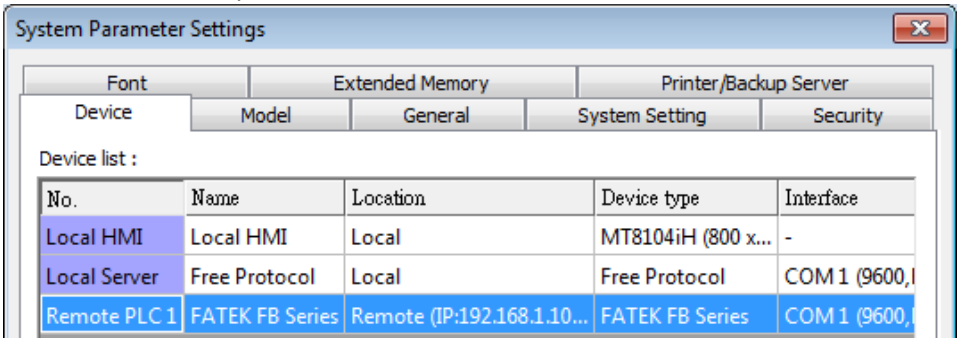
Name	StringFind
Syntax	position = StringFind (source[start], target[start]) position = StringFind ("source", target[start]) position = StringFind (source[start], "target") position = StringFind ("source", "target")
Description	Returns the position of the first occurrence of target string in the source string. The two string parameters accept both static string (in the form: “source”) and

	char array (in the form: source[start]). This function returns the zero-based index of the first character of substring in the source string that matches the target string. Notice that the entire sequence of characters to find must be matched. If there is no matched substring, it returns -1.

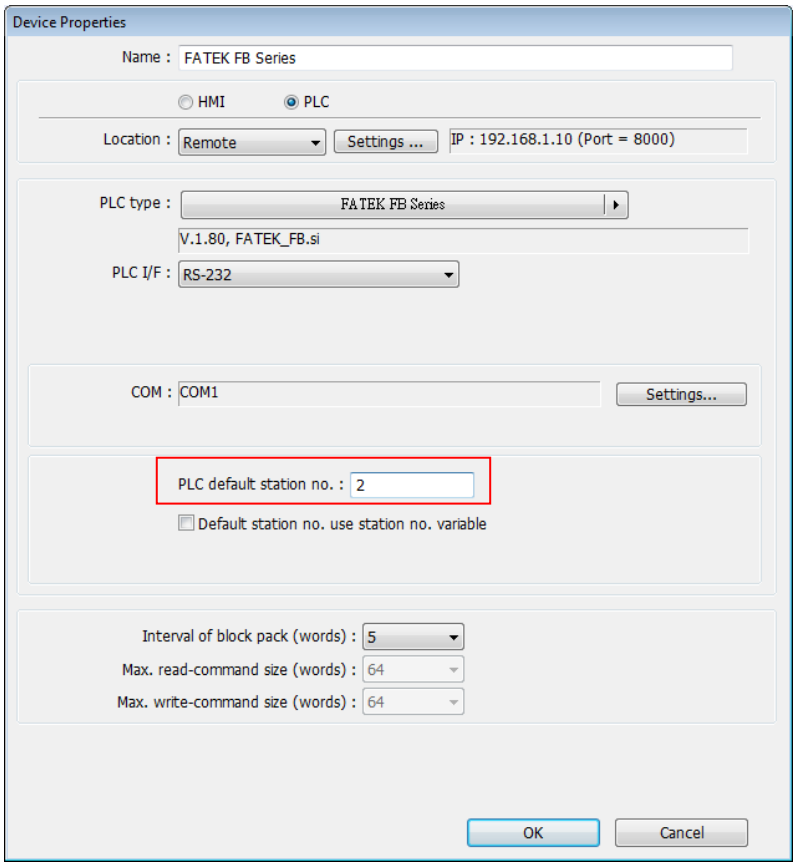
Example	macro_command main()
	char src1[20]="abcde"
	char target1[20]="cd"
	short pos1
	pos1= StringFind(src1[0], target1[0])
	// pos1=2
	
	char target2[20]="ce"
	short pos2
	pos2= StringFind("abcde", target2[0])
	// pos2=-1
	
	char src3[20]="abcde"
	short pos3
	pos3= StringFind(src3[3], "cd")
	// pos3=-1
	
	end macro_command

Name	StringFindOneOf
Syntax	position = StringFindOneOf (source[start], target[start]) position = StringFindOneOf ("source", target[start]) position = StringFindOneOf (source[start], "target") position = StringFindOneOf ("source", "target")
Description	Returns the position of the first character in the source string that matches any character contained in the target string. The two string parameters accept both static string (in the form: "source") and char array (in the form: source[start]). This function returns the zero-based index of the first character in the source string that is also in the target string. If there is no match, it returns -1.

Example	macro_command main()
	char src1[20]="abcdeabcde"
	char target1[20]="sdf"
	short pos1
	pos1= StringFindOneOf(src1[0], target1[0])
	// pos1=3
	
	char src2[20]="abcdeabcde"
	short pos2
	pos2= StringFindOneOf(src2[1], "agi")
	// pos2=4
	
	char target3 [20]="bus"
	short pos3
	pos3= StringFindOneOf("abcdeabcde", target3[1])
	// pos3=-1
	
	end macro_command

Name	StringGet
Syntax	StringGet(*read_data*[*start*], *device_name, device_type, address_offset, data_count)*
Description	Receives data from the PLC. The String data is stored into *read_data*[*start*]~ *read_data*[*start + data_count - 1*]. *read_data* must be a one-dimensional char array. *Data_count* is the number of received characters, it can be either a constant or a variable. *Device_name* is the PLC name enclosed in the double quotation marks (") and this name has been defined in the device list of system parameters as follows (see FATEK KB Series):  Device_type is the device type and encoding method (binary or BCD) of the PLC data. For example, if device_type is LW_BIN, it means the register is LW and the encoding method is binary. If use BIN encoding method, "_BIN" can be ignored. If device_type is LW_BCD, it means the register is LW and the encoding method is BCD. Address_offset is the address offset in the PLC. For example, StringGet(read_data_1[0], "FATEK KB Series", RT, 5, 1) represents

that the address offset is 5.
If address_offset uses the format –"N#AAAAA", N indicates that PLC's station number is N. AAAAA represents the address offset. This format is used while multiple PLCs or controllers are connected to a single serial port. For example, StringGet(read_data_1[0], "FATEK KB Series", RT, 2#5, 1) represents that the PLC's station number is 2. If StringGet() uses the default station number defined in the device list as follows, it is not necessary to define station number in address_offset.



The number of registers actually read from depends on the value of the number of *data_count* since that the *read_data* is restricted to char array.

type of read_data	data_count	actual number of 16-bit register read
char (8-bit)	1	1
char (8-bit)	2	1

1 WORD register(16-bit) equals to the size of 2 ASCII characters. According to the above table, reading 2 ASCII characters is actually reading the content of one 16-bit register.

Example

```
macro_command main()
char str1[20]

//   read 10 words from LW-0~LW-9 to the variables str1[0] to str1[19]
//   since that 1 word can store 2 ASCII characters, reading 20 ASCII
//   characters is actually reading 10 words of register
```

	StringGet(str1[0], "Local HMI", LW, 0, 20) end macro_command

Name	StringGetEx
Syntax	StringGetEx (*read_data*[*start*], *device_name, device_type, address_offset, data_count*)
Description	Receives data from the PLC and continues executing next command even if there's no response from this device. Descriptions of *read_data, device_name, device_type, address_offset* and *data_count* are the same as GetData.

Example	macro_command main()
	char str1[20]
	short test=0
	
	// macro will continue executing test = 1 even if the MODBUS device is
	// not responding
	StringGetEx(str1[0], "MODBUS RTU", 4x, 0, 20)
	test = 1
	
	// macro will not continue executing test = 2 until MODBUS device responds
	StringGet(str1[0], "MODBUS RTU", 4x, 0, 20)
	test = 2
	
	end macro_command

Name	StringIncluding
Syntax	success = StringIncluding (source[start], set[start], destination[start]) success = StringIncluding ("source", set[start], destination[start]) success = StringIncluding (source[start], "set", destination[start]) success = StringIncluding ("source", "set", destination[start])
Description	Retrieves a substring of the source string that contains characters in the set string, beginning with the first character in the source string and ending when a character is found in the source string that is not in the target string. The source string and set string parameters accept both static string (in the form: "source") and char array (in the form: source[start]). This function returns a Boolean indicating whether the process is successfully done or not. If successful, it returns true, otherwise it returns false. If the length of retrieved substring exceeds the size of destination buffer, it returns false.

Example	macro_command main()
	char src1[20]="cabbageabc"
	char set1[20]="abc"
	char dest1[20]
	bool success1
	success1 = StringIncluding(src1[0], set1[0], dest1[0])
	// success1=true, dest1="cabba"
	
	char src2[20]="gecabba"
	char dest2[20]
	bool success2
	success2 = StringIncluding(src2[0], "abc", dest2[0])
	// success2=true, dest2=""
	
	char set3[20]="abc"
	char dest3[4]
	bool success3
	success3 = StringIncluding("cabbage", set3[0], dest3[0])
	// success3=false, dest3 remains the same.
	
	end macro_command

Name	StringInsert
Syntax	success = StringInsert (pos, insert[start], destination[start]) success = StringInsert (pos, "insert", destination[start]) success = StringInsert (pos, insert[start], length, destination[start]) success = StringInsert (pos, "insert", length, destination[start])
Description	Inserts a string in a specific location within the destination string content. The insert location is specified by the pos parameter. The insert string parameter accepts both static string (in the form: “source”) and char array (in the form: source[start]). The number of characters to insert can be specified by the length parameter. This function returns a Boolean indicating whether the process is successfully done or not. If successful, it returns true, otherwise it returns false. If the length of string after insertion exceeds the size of destination buffer, it returns false.

Example	macro_command main()
	
	char str1[20]="but the question is"
	char str2[10]=", that is"
	char dest[40]="to be or not to be"
	bool success
	
	success = StringInsert(18, str1[3], 13, dest[0])
	// success=true, dest="to be or not to be the question"
	
	success = StringInsert(18, str2[0], dest[0])
	// success=true, dest="to be or not to be, that is the question"
	
	success = StringInsert(0, "Hamlet:", dest[0])
	// success=false, dest remains the same.
	
	end macro_command

Name	StringLength
Syntax	length = StringLength (source[start]) or length = StringLength ("source")
Description	Obtains the length of a string. It returns the length of source string and stores it in the length field on the left-hand side of '=' operator. The source string parameter accepts both static string (in the form: "source") and char array (in the form: source[start]). The return value of this function indicates the length of the source string.

Example	macro_command main()
	char src1[20]="abcde"
	int length1
	length1= StringLength(src1[0])
	// length1=5
	
	char src2[20]={'a', 'b', 'c', 'd', 'e'}
	int length2
	length2= StringLength(src2[0])
	// length2=5
	
	char src3[20]="abcdefghij"
	int length3
	length3= StringLength(src3 [2])
	// length3=8
	
	end macro_command

Name	StringMid
Syntax	success = StringMid (source[start], count, destination[start]) or success = StringMid ("string", start, count, destination[start])
Description	Retrieves a character sequence from the specified offset of the source string and stores it in the destination buffer. The source string parameter accepts both static string (in the form: "source") and char array (in the form: source[start]). For source[start], the start offset of the substring is specified by the index value. For static source string(“source”), the second parameter(start) specifies the start offset of the substring. The count parameter specifies the length of substring being retrieved. Destination must be an one-dimensional char array, to store the retrieved substring. This function returns a Boolean indicating whether the process is successfully done or not. If successful, it returns true, otherwise it returns false. If the length of retrieved substring exceeds the size of destination buffer, it returns false. The success field is optional.

Example	macro_command main()
	char src1[20]="abcdefghijklmnopqrst"
	char dest1[20]
	bool success1
	success1 = StringMid(src1[5], 6, dest1[0])
	// success1=true, dest1="fghijk"
	
	char src2[20]="abcdefghijklmnopqrst"
	char dest2[5]
	bool success2
	success2 = StringMid(src2[5], 6, dest2[0])
	// success2=false, dest2 remains the same.
	
	char dest3[20]="12345678901234567890"
	bool success3
	success3 = StringMid("abcdefghijklmnopqrst", 5, 5, dest3[15])
	// success3= true, dest3="123456789012345fghij"
	
	end macro_command

Name	StringReverseFind
Syntax	position = StringReverseFind (source[start], target[start]) position = StringReverseFind ("source", target[start]) position = StringReverseFind (source[start], "target") position = StringReverseFind ("source", "target")
Description	Returns the position of the last occurrence of target string in the source string. The two string parameters accept both static string (in the form: "source") and

	char array (in the form: source[start]). This function returns the zero-based index of the first character of substring in the source string that matches the target string. Notice that the entire sequence of characters to find must be matched. If there exists multiple substrings that matches the target string, function will return the position of the last matched substring. If there is no matched substring, it returns -1.

Example	macro_command main()
	char src1[20]="abcdeabcde"
	char target1[20]="cd"
	short pos1
	pos1= StringReverseFind(src1[0], target1[0])
	// pos1=7
	
	char target2[20]="ce"
	short pos2
	pos2= StringReverseFind("abcdeabcde", target2[0])
	// pos2=-1
	
	char src3[20]="abcdeabcde"
	short pos3
	pos3= StringReverseFind(src3[6], "ab")
	// pos3=-1
	
	end macro_command

Name	StringSet
Syntax	StringSet(*send_data*[*start*], *device_name, device_type, address_offset, data_count*)
Description	Sends data to the PLC. Data is defined in *send_data*[*start*]~ *send_data*[*start + data_count - 1*]. send_data must be a one-dimensional char array. *data_count* is the number of sent characters, it can be either a constant or a variable. *device_name* is the PLC name enclosed in the double quotation marks (") and this name has been defined in the device list of system parameters. *device_type* is the device type and encoding method (binary or BCD) of the PLC data. For example, if *device_type* is LW_BIN, it means the register is LW and the encoding method is binary. If use BIN encoding method, "_BIN" can be ignored. If *device_type* is LW_BCD, it means the register is LW and the encoding method is BCD. *address_offset* is the address offset in the PLC. For example, StringSet(read_data_1[0], "FATEK KB Series", RT, 5, 1) represents that the address offset is 5. If *address_offset* uses the format –"N#AAAAA", N indicates that PLC's station number is N. AAAAA represents the address offset. This format is used while multiple PLCs or controllers are connected to a single serial port. For example,

StringSet(read_data_1[0], "FATEK KB Series", RT, 2#5, 1) represents that the PLC's station number is 2. If SetData () uses the default station number defined in the device list, it is not necessary to define station number in *address_offset*.

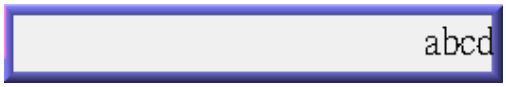
The number of registers actually sends to depends on the value of the number of *data_count*, since that *send_data* is restricted to char array.

type of read_data	data_count	actual number of 16-bit register send
char (8-bit)	1	1
char (8-bit)	2	1

1 WORD register(16-bit) equals to the size of 2 ASCII characters. According to the above table, sending 2 ASCII characters is actually writing to one 16-bit register. The ASCII characters are stored into the WORD register from low byte to high byte. While using the ASCII Display object to display the string data stored in the registers, *data_count* must be a multiple of 2 in order to display full string content. For example:

```
macro_command main()
char src1[10]="abcde"
StringSet(src1[0], "Local HMI", LW, 0, 5)
end macro_command
```

The ASCII Display object shows:



If *data_count* is an even number that is greater than or equal to the length of the string, the content of string can be completely shown:

```
macro_command main()
char src1[10]="abcde"
StringSet(src1[0], "Local HMI", LW, 0, 6)
end macro_command
```

Example

```
macro_command main()

char str1[10]="abcde"

//   Send 3 words to LW-0~LW-2
//   Data are being sent until the end of string is reached.
```

	// Even though the value of data_count is larger than the length of string // , the function will automatically stop. StringSet(str1[0], "Local HMI", LW, 0, 10) end macro_command

Name	StringSetEx
Syntax	StringSetEx (*send_data*[*start*]*, device_name, device_type, address_offset, data_count*)
Description	Sends data to the PLC and continues executing next command even if there's no response from this device. Descriptions of *send_data, device_name, device_type, address_offset* and *data_count* are the same as StringSet.

Example	macro_command main()
	char str1[20]="abcde"
	short test=0
	
	// macro will continue executing test = 1 even if the MODBUS device is
	// not responding
	StringSetEx(str1[0], "MODBUS RTU", 4x, 0, 20)
	test = 1
	
	// macro will not continue executing test = 2 until MODBUS device responds
	StringSet(str1[0], "MODBUS RTU", 4x, 0, 20)
	test = 2
	
	end macro_command

Name	StringToUpper
Syntax	success = StringToUpper (source[start], destination[start]) success = StringToUpper ("source", destination[start])
Description	Converts all the characters in the source string to uppercase characters and stores the result in the destination buffer. The source string parameter accepts both static string (in the form: "source") and char array (in the form: source[start]). This function returns a Boolean indicating whether the process is successfully done or not. If successful, it returns true, otherwise it returns false. If the length of result string after conversion exceeds the size of destination buffer, it returns false.

Example	macro_command main()
	char src1[20]="aBcDe"
	char dest1[20]
	bool success1
	success1 = StringToUpper(src1[0], dest1[0])
	// success1=true, dest1="ABCDE"
	
	char dest2[4]
	bool success2
	success2 = StringToUpper("aBcDe", dest2[0])
	// success2=false, dest2 remains the same.
	
	end macro_command

Name	StringToLower
Syntax	success = StringToLower (source[start], destination[start]) success = StringToLower ("source", destination[start])
Description	Converts all the characters in the source string to lowercase characters and stores the result in the destination buffer. The source string parameter accepts both static string (in the form: "source") and char array (in the form: source[start]). This function returns a Boolean indicating whether the process is successfully done or not. If successful, it returns true, otherwise it returns false. If the length of result string after conversion exceeds the size of destination buffer, it returns false.

Example	macro_command main()
	char src1[20]="aBcDe"
	char dest1[20]
	bool success1
	success1 = StringToLower(src1[0], dest1[0])
	// success1=true, dest1="abcde"
	
	char dest2[4]
	bool success2
	success2 = StringToLower("aBcDe", dest2[0])
	// success2=false, dest2 remains the same.
	
	end macro_command

Name	StringToReverse
Syntax	success = StringToReverse (source[start], destination[start]) success = StringToReverse ("source", destination[start])
Description	Reverses the characters in the source string and stores it in the destination buffer.

	The source string parameter accepts both static string (in the form: "source") and char array (in the form: source[start]). This function returns a Boolean indicating whether the process is successfully done or not. If successful, it returns true, otherwise it returns false. If the length of reversed string exceeds the size of destination buffer, it returns false.

Example	macro_command main()
	char src1[20]="abcde"
	char dest1[20]
	bool success1
	success1 = StringToReverse(src1[0], dest1[0])
	// success1=true, dest1="edcba"
	
	char dest2[4]
	bool success2
	success2 = StringToReverse("abcde", dest2[0])
	// success2=false, dest2 remains the same.
	
	end macro_command

Name	StringTrimLeft
Syntax	success = StringTrimLeft (source[start], set[start], destination[start]) success = StringTrimLeft ("source", set[start], destination[start]) success = StringTrimLeft (source[start], "set", destination[start]) success = StringTrimLeft ("source", "set", destination[start])
Description	Trims the leading specified characters in the set buffer from the source string. The source string and set string parameters accept both static string (in the form: "source") and char array (in the form: source[start]). This function returns a Boolean indicating whether the process is successfully done or not. If successful, it returns true, otherwise it returns false. If the length of trimmed string exceeds the size of destination buffer, it returns false.

Example	macro_command main()
	char src1[20]= "# *a*#bc"
	char set1[20]="# *"
	char dest1[20]
	bool success1
	success1 = StringTrimLeft (src1[0], set1[0], dest1[0])
	// success1=true, dest1="a*#bc"
	
	char set2[20]={'#', ' ', '*'}
	char dest2[4]
	bool success2
	success2 = StringTrimLeft ("# *a*#bc", set2[0], dest2[0])
	// success2=false, dest2 remains the same.
	
	char src3[20]="abc *#"

	char dest3[20] bool success3 success3 = StringTrimLeft (src3[0], "# *", dest3[0]) // success3=true, dest3="abc *#" end macro_command

Name	StringTrimRight
Syntax	success = StringTrimRight (source[start], set[start], destination[start]) success = StringTrimRight ("source", set[start], destination[start]) success = StringTrimRight (source[start], "set", destination[start]) success = StringTrimRight ("source", "set", destination[start])
Description	Trims the trailing specified characters in the set buffer from the source string. The source string and set string parameters accept both static string (in the form: "source") and char array (in the form: source[start]). This function returns a Boolean indicating whether the process is successfully done or not. If successful, it returns true, otherwise it returns false. If the length of trimmed string exceeds the size of destination buffer, it returns false.

Example	macro_command main()
	char src1[20]= "# *a*#bc# * "
	char set1[20]="# *"
	char dest1[20]
	bool success1
	success1 = StringTrimRight(src1[0], set1[0], dest1[0])
	// success1=true, dest1="# *a*#bc"
	
	char set2[20]={'#', ' ', '*'}
	char dest2[20]
	bool success2
	success2 = StringTrimRight("# *a*#bc", set2[0], dest2[0])
	// success2=true, dest2="# *a*#bc"
	
	char src3[20]="ab**c *#"
	char dest3[4]
	bool success3
	success3 = StringTrimRight(src3[0], "# *", dest3[0])
	// success3=false, dest3 remains the same.

	end macro_command

Click the icon to download the demo project. Please confirm your internet connection before downloading the demo project.

18.7.8. Mathematics

Name	SQRT
Syntax	SQRT(*source, result*)
Description	Calculates the square root of *source* and stores the result into *result*. *source* can be a constant or a variable. *result* must be a variable. *source* must be a nonnegative value.
Example	macro_command main() float source, result SQRT(15, result) source = 9.0 SQRT(source, result)// result is 3.0 end macro_command

Name	CUBERT
Syntax	CUBERT(*source, result*)
Description	Calculates the cube root of source and stores the result into *result*. *source* can be a constant or a variable. *result* must be a variable. *source* must be a nonnegative value.

Example	macro_command main()
	float source, result
	
	CUBERT (27, result) // result is 3.0
	
	source = 27.0
	CUBERT(source, result)// result is 3.0
	
	end macro_command

Name	POW
Syntax	POW(*source1, source2, result*)
Description	Calculates *source1* to the power of *source2*. *source1* and *source2* can be a constant or a variable. *result* must be a variable. *source1* and *source2* must be a nonnegative value.
Example	macro_command main() float y, result

	y = 0.5 POW (25, y, result) // result = 5 end macro_command

Name	SIN
Syntax	SIN(*source, result*)
Description	Calculates the sine of *source* (degree) into *result*. *source* can be a constant or a variable. *result* must be a variable.

Example	macro_command main()
	float source, result
	
	SIN(90, result)// result is 1
	
	source = 30
	SIN(source, result)// result is 0.5
	
	end macro_command

Name	COS
Syntax	COS(*source, result*)
Description	Calculates the cosine of *source* (degree) into *result*. *source* can be a constant or a variable. *result* must be a variable.

Example	macro_command main()
	float source, result
	
	COS(90, result)// result is 0
	
	source = 60
	COS(source, result)// result is 0.5
	
	end macro_command

Name	TAN
Syntax	TAN(*source, result*)
Description	Calculates the tangent of *source* (degree) into *result*. *source* can be a constant or a variable. *result* must be a variable.

Example	macro_command main()
	float source, result
	
	TAN(45, result)// result is 1
	
	source = 60
	TAN(source, result)// result is 1.732

	end macro_command

Name	COT
Syntax	COT(*source, result*)
Description	Calculates the cotangent of *source* (degree) into *result*. *source* can be a constant or a variable. *result* must be a variable.

Example	macro_command main()
	float source, result
	
	COT(45, result)// result is 1
	
	source = 60
	COT(source, result)// result is 0.5774
	
	end macro_command

Name	SEC
Syntax	SEC(*source, result*)
Description	Calculates the secant of *source* (degree) into *result*. *source* can be a constant or a variable. *result* must be a variable.

Example	macro_command main()
	float source, result
	
	SEC(45, result)// result is 1.414
	
	source = 60
	SEC(source, result)// if source is 60, result is 2
	
	end macro_command

Name	CSC
Syntax	CSC(*source, result*)
Description	Calculates the cosecant of *source* (degree) into *result*. *source* can be a constant or a variable. *result* must be a variable.

Example	macro_command main()
	float source, result
	
	CSC(45, result)// result is 1.414
	
	source = 30
	CSC(source, result)// result is 2
	
	end macro_command

Name	ASIN
Syntax	ASIN(*source, result*)
Description	Calculates the arc sine of *source* into *result* (degree). *source* can be a constant or a variable. *result* must be a variable.

Example	macro_command main()
	float source, result
	
	ASIN(0.8660, result)// result is 60
	
	source = 0.5
	ASIN(source, result)// result is 30
	
	end macro_command

Name	ACOS
Syntax	ACOS(*source, result*)
Description	Calculates the arc cosine of *source* into *result*. *source* can be a constant or a variable. *result* must be a variable.

Example	macro_command main()
	float source, result
	
	ACOS(0.8660, result)// result is 30
	
	source = 0.5
	ACOS(source, result)// result is 60
	
	end macro_command

Name	ATAN
Syntax	ATAN(*source, result*)
Description	Calculates the arc tangent of *source* into *result*. *source* can be a constant or a variable. *result* must be a variable.

Example	macro_command main()
	float source, result
	
	ATAN(1, result)// result is 45
	
	source = 1.732
	ATAN(source, result)// result is 60
	
	end macro_command

Name	LOG
Syntax	LOG (*source, result*)
Description	Calculates the natural logarithm of a number and saves into *result*. *source* can be either a variable or a constant. *result* must be a variable.
Example	macro_command main() float source = 100, result LOG (source, result)// result is approximately 4.6052 end macro_command

Name	LOG10
Syntax	LOG10(*source, result*)
Description	Calculates the base-10 logarithm of a number and saves into *result*. *source* can be either a variable or a constant. *result* must be a variable.
Example	macro_command main() float source = 100, result LOG10 (source, result) // result is 2 end macro_command

Name	RAND
Syntax	RAND(*result*)
Description	Calculates a random integer and saves into *result*. (Range: 0 ~ 32767) *result* must be a variable.
Example	macro_command main() short result RAND (result) //result is not a fixed value when executes macro every time end macro_command

Click the icon to download the demo project. Please confirm your internet connection before downloading the demo project.

18.7.9. Statistics

Name	AVERAGE
Syntax	AVERAGE(*source[start], result, count*)
Description	Gets the average value from array.
Example	int data[5] = {1, 2, 3, 4, 5} float result AVERAGE(data[0], result, 5) // result is equal to 3

	AVERAGE(data[2], result, 3) // result is equal to 4

Name	HARMEAN
Syntax	HARMEAN(*source[start], result, count*)
Description	Gets the harmonic mean value from array.
Example	int data[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} float result HARMEAN(data[0], result, 10) // result is equal to 3.414

Name	MAX
Syntax	MAX(*source[start], result, count*)
Description	Gets the maximum value from array.

Example	int data[5] = {1, 2, 3, 4, 5}
	int result
	
	MAX(data[0], result, 5) // result is equal to 5
	MAX(data[1], result, 3) // result is equal to 4

Name	MEDIAN
Syntax	MEDIAN(*source[start], result, count*)
Description	Gets the median value from array.
Example	int data[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} float result MEDIAN(data[0], result, 10) // result is equal to 5.5

Name	MIN
Syntax	MIN(*source[start], result, count*)
Description	Gets the minimum value from array.

Example	int data[5] = {1, 2, 3, 4, 5}
	int result
	
	MIN(data[0], result, 5) // result is equal to 1
	MIN(data[1], result, 3) // result is equal to 2

Name	STDEVP
Syntax	STDEVP(*source[start], result, count*)
Description	Gets the standard deviation value from array.
Example	int data[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} float result STDEVP(data[0], result, 10) // result is equal to 2.872

Name	STDEVS
Syntax	STDEVS(*source[start], result, count*)
Description	Gets the sample standard deviation value from array.
Example	int data[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} float result STDEVS(data[0], result, 10) // result is equal to 3.027

18.7.10. Recipe Database

Name	RecipeGetData
Syntax	RecipeGetData(*destination, recipe_address, record_ID*)
Description	Gets Recipe Data. The gained data will be stored in *destination*, and must be a variable. *recipe_address* consists of recipe name and item name: "recipe_name.item_name". record_ID specifies the ID number of the record in recipe being gained.

Example	macro_command main()
	int data=0
	char str[20]
	int recordID
	bool result
	
	recordID = 0
	result = RecipeGetData(data, "TypeA.item_weight", recordID)
	// From recipe "TypeA" get the data of the item "item_weight" in record 0.
	
	recordID = 1
	result = RecipeGetData(str[0], "TypeB.item_name", recordID)
	  // From recipe "TypeB" get the data of the item "item_name" in record 1.
	
	end macro_command

Name	RecipeQuery
Syntax	RecipeQuery (*SQL_command, destination*)
Description	Uses SQL statement to query recipe data. The number of records of query result will be stored in the *destination*. This must be a variable. SQL command can be static string or char array. Example: RecipeQuery("SELECT * FROM TypeA", destination) or RecipeQuery(sql[0], destination) SQL statement must start with "SELECT * FROM" followed by recipe name and query condition.
Example	macro_command main() int total_row=0 char sql[100]="SELECT * FROM TypeB"

	short var bool result result = RecipeQuery("SELECT * FROM TypeA", total_row) // Query Recipe "TypeA". Store the number of records of query result in total_row. result = RecipeQuery(sql[0], total_row) // Query Recipe "TypeB". Store the number of records of query result in total_row. result = RecipeQuery("SELECT * FROM Recipe WHERE Item >%(var)", total_row) // Query "Recipe", where "Item" is larger than var. Store the number of records of query result in total_row. end macro_command

Name	RecipeQueryGetData
Syntax	RecipeQueryGetData (*destination, recipe_address, result_row_no*)
Description	Gets the data in the query result obtained by RecipeQuery. This function must be called after calling RecipeQuery, and specify the same recipe name in *recipe_address* as RecipeQuery. *result_row_no* specifies the sequence row number in query result

Example	macro_command main()
	
	int data=0
	int total_row=0
	int row_number=0
	bool result_query
	bool result_data
	
	result_query = RecipeQuery("SELECT * FROM TypeA", total_row)
	// Query Recipe "TypeA". Store the number of records of query result in total_row.
	if (result_query) then
	for row_number=0 to total_row-1
	result_data = RecipeQueryGetData(data, "TypeA.item_weight", row_number)
	next row_number
	end if
	
	end macro_command

Name	RecipeQueryGetRecordID
Syntax	RecipeQueryGetRecordID (*destination, result_row_no*)
Description	Gets the record ID numbers of those records gained by RecipeQuery. This function must be called after calling RecipeQuery. *result_row_no* specifies the sequence row number in query result, and write

	the obtained record ID to destination.

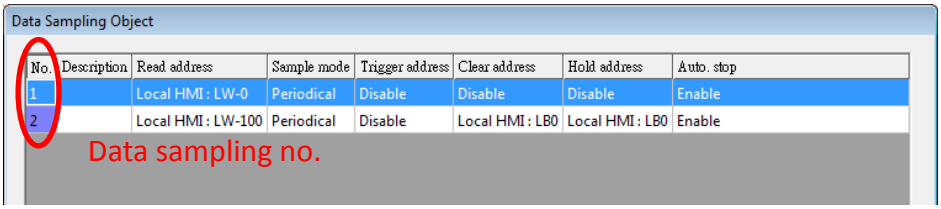
Example	macro_command main()
	
	int recordID=0
	int total_row=0
	int row_number=0
	bool result_query
	bool result_id
	
	result_query = RecipeQuery("SELECT * FROM TypeA", total_row)
	// Query Recipe "TypeA". Store the number of records of query result in total_row.
	if (result_query) then
	for row_number=0 to total_row-1
	result_id = RecipeQueryGetRecordID(recordID, row_number)
	next row_number
	end if
	
	end macro_command

Name	RecipeSetData
Syntax	RecipeSetData(*source*, *recipe address*, *record_ID*)
Description	Writes data to recipe. If success, returns true, else, returns false. *recipe_address* consists of recipe name and item name: "recipe_name.item_name". *record_ID* specifies the ID number of the record in recipe being modified.

Example	macro_command main()
	
	int data=99
	char str[20]="abc"
	int recordID
	bool result
	
	recordID = 0
	result = RecipeSetData(data, "TypeA.item_weight", recordID)
	// set data to recipe "TypeA", where item name is "item_weight" and the record ID is 0.
	
	recordID = 1
	result = RecipeSetData(str[0], "TypeB.item_name", recordID)
	// set data to recipe "TypeB", where item name is "item_name" and the record ID is 1.
	
	end macro_command

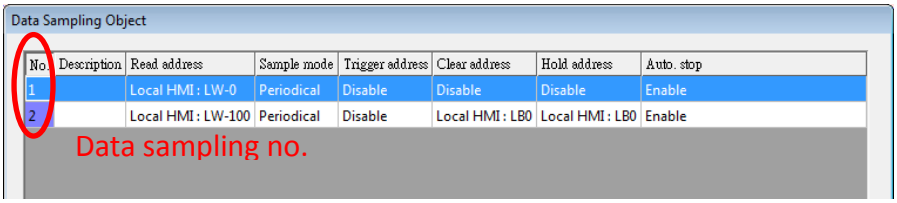
Click the icon to download the demo project. Please confirm your internet connection before downloading the demo project.

18.7.11. Data/Event Log

Name	FindDataSamplingDate
Syntax	return_value = FindDataSamplingDate *(data_log_number, index, year, month, day*) or FindDataSamplingDate (*data_log_number, index, year, month, day*)
Description	A query function for finding the date of specified data sampling file according to the data sampling no. and the file index. The date is stored into year, month and day respectively in the format of YYYY, MM and DD.  The directory of saved data: [Storage location]\[filename]\yyyymmdd.dtl. The data sampling files under the same directory are sorted according to the file name and are indexed starting from 0. The most recently saved file has the smallest file index number. For example, if there are four data sampling files as follows: 20101210.dtl 20101230.dtl 20110110.dtl 20110111.dtl The file index are: 20101210.dtl -> index is 3 20101230.dtl -> index is 2 20110110.dtl -> index is 1 20110111.dtl -> index is 0 *return_value* equals to 1 if referred data sampling file is successfully found, otherwise it equals to 0. *data_log_number* and *index* can be constant or variable. *year, month, day* and *return_value* must be variable. *return_value* is optional.

Example	macro_command main()
	short data_log_number = 1, index = 2, year, month, day
	short success
	
	// if there exists a data sampling file named 20101230.dtl, with data sampling //
	number 1 and file index 2.
	// the result after execution: success == 1, year == 2010, month == 12 and //day
	== 30
	success = FindDataSamplingDate(data_log_number, index, year, month, day)
	
	end macro_command

Click the icon to download the demo project. Please confirm your internet connection before downloading the demo project.

Name	FindDataSamplingIndex
Syntax	return_value = FindDataSamplingIndex (*data_log_number, year, month, day, index*) or FindDataSamplingIndex (*data_log_number, year, month, day, index*)
Description	A query function for finding the file index of specified data sampling file according to the data sampling no. and the date. The file index is stored into index. year, month and day are in the format of YYYY, MM and DD respectively.  The directory of saved data: [Storage location]\[filename]\yyyymmdd.dtl. The data sampling files under the same directory are sorted according to the file name and are indexed starting from 0. The most recently saved file has the smallest file index number. For example, if there are four data sampling files as follows: 20101210.dtl 20101230.dtl 20110110.dtl 20110111.dtl The file index are: 20101210.dtl -> index is 3 20101230.dtl -> index is 2 20110110.dtl -> index is 1 20110111.dtl -> index is 0 *return_value* equals to 1 if referred data sampling file is successfully found, otherwise it equals to 0. *data_log_number, year, month* and *day* can be constant or variable. *index* and *return_value* must be variable. *return_value* is optional.

Example	macro_command main()
	short data_log_number = 1, year = 2010, month = 12, day = 10, index
	short success
	
	// if there exists a data sampling file named 20101210.dtl, with data sampling //
	number 1 and file index 2.
	// the result after execution: success == 1 and index == 2
	success = FindDataSamplingIndex (data_log_number, year, month, day, index)
	
	end macro_command

Click the icon to download the demo project. Please confirm your internet connection before downloading the demo project.

Name	FindEventLogDate
Syntax	return_value = FindEventLogDate (*index, year, month, day*) or

	FindEventLogDate (*index, year, month, day*)
Description	A query function for finding the date of specified event log file according to file index. The date is stored into year, month and day respectively in the format of YYYY, MM and DD. The event log files stored in the designated position (such as HMI memory storage or external memory device) are sorted according to the file name and are indexed starting from 0. The most recently saved file has the smallest file index number. For example, if there are four event log files as follows: EL_20101210.evt EL_20101230.evt EL_20110110.evt EL_20110111.evt The file index are: EL_20101210.evt -> index is 3 EL_20101230.evt -> index is 2 EL_20110110.evt -> index is 1 EL_20110111.evt -> index is 0 *return_value* equals to 1 if referred data sampling file is successfully found, otherwise it equals to 0. *index* can be constant or variable. *year, month, day* and *return_value* must be variable. *return_value* is optional.

Example	macro_command main()
	short index = 1, year, month, day
	short success
	
	// if there exists an event log file named EL_20101230.evt，with index 1
	// the result after execution: success == 1, year == 2010, month == 12, day //== 30
	success = FindEventLogDate (index, year, month, day)
	
	end macro_command

Click the icon to download the demo project. Please confirm your internet connection before downloading the demo project.

Name	FindEventLogIndex
Syntax	return_value = FindEventLogIndex (*year, month, day, index*) or FindEventLogIndex (*year, month, day, index*)
Description	A query function for finding the file index of specified event log file according to date. The file index is stored into index. year, month and day are in the format of YYYY, MM and DD respectively. The event log files stored in the designated position (such as HMI memory storage or external memory device) are sorted according to the file name and are indexed starting from 0. The most recently saved file has the smallest file index number. For example, if there are four event log files as follows: EL_20101210.evt EL_20101230.evt EL_20110110.evt EL_20110111.evt

	The file index are: EL_20101210.evt -> index is 3 EL_20101230.evt -> index is 2 EL_20110110.evt -> index is 1 EL_20110111.evt -> index is 0 *return_value* equals to 1 if referred data sampling file is successfully found, otherwise it equals to 0. *index* can be constant or variable. *year, month, day* and *return_value* must be variable. *return_value* is optional.

Example	macro_command main()
	short year = 2010, month = 12, day = 10, index
	short success
	
	// if there exists an event log file named EL_20101210.evt, with index 2
	// the result after execution: success == 1, index == 2
	success = FindEventLogIndex (year, month, day, index)
	
	end macro_command

Click the icon to download the demo project. Please confirm your internet connection before downloading the demo project.

18.7.12. Checksum

Name	ADDSUM
Syntax	ADDSUM(*source*[*start*], *result, data_count*)
Description	Adds up the elements of an array (*source*) from *source[start]* to *source*[*start + data_count - 1*] to generate a checksum. Puts in the checksum into *result*. *result* must be a variable. *data_count* is the amount of the accumulated elements and can be a constant or a variable.
Example	macro_command main() char data[5] short checksum data[0] = 0x1 data[1] = 0x2 data[2] = 0x3 data[3] = 0x4 data[4] = 0x5 ADDSUM(data[0], checksum, 5)// checksum is 0xf end macro_command

Name	XORSUM
Syntax	XORSUM(*source*[*start*], *result, data_count*)
Description	Uses XOR to calculate the checksum from *source*[*start*] to *source*[*start +*

	data_count - 1]. Puts the checksum into *result*. *result* must be a variable. *data_count* is the amount of the calculated elements of the array and can be a constant or a variable.
Example	macro_command main() char data[5] = {0x1, 0x2, 0x3, 0x4, 0x5} short checksum XORSUM(data[0], checksum, 5)// checksum is 0x1 end macro_command

Name	BCC
Syntax	BCC(*source[start], result, data_count*)
Description	Same as XORSUM.
Example	macro_command main() char data[5] = {0x1, 0x2, 0x3, 0x4, 0x5} char checksum BCC(data[0], checksum, 5) // checksum is 0x1 end macro_command

Name	CRC
Syntax	CRC(*source*[*start*], *result, data_count*)
Description	Calculates 16-bit CRC of the variables from *source*[*start*] to source[*start + data_count - 1*]. Puts in the 16-bit CRC into *result*. *result* must be a variable. *data_count* is the amount of the calculated elements of the array and can be a constant or a variable.
Example	macro_command main() char data[5] = {0x1, 0x2, 0x3, 0x4, 0x5} short checksum CRC(data[0], checksum, 5) // checksum is 0xbb2a, 16-bit CRC end macro_command

Name	CRC8
Syntax	CRC8(*source*[*start*], *result, data_count*)
Description	Calculates 8-bit CRC of the variables from *source*[*start*] to source[*start + data_count - 1*]. Puts in the 8-bit CRC into *result*. *result* must be a variable. *data_count* is the amount of the calculated elements of the array and can be a constant or a variable.
Example	macro_command main() char source[5] = {1, 2, 3, 4, 5} short CRC8_result

	CRC8(source[0], CRC8_result, 5) // CRC8_result = 188 end macro_command

18.7.13. Miscellaneous

Name	Beep
Syntax	Beep ()
Description	Plays beep sound. This command plays a beep sound with frequency of 800 hertz and duration of 30 milliseconds.
Example	macro_command main() Beep() end macro_command

Name	Buzzer
Syntax	Buzzer ()
Description	Turns ON / OFF the buzzer.

Example	macro_command main()
	
	char on = 1, off = 0
	Buzzer(on) // turn on the buzzer
	
	DELAY(1000) // delay 1 second
	
	Buzzer(off) // turn off the buzzer
	
	DELAY(500) // delay 500ms
	
	Buzzer(1) // turn on the buzzer
	
	DELAY(1000) // delay 1 second
	
	Buzzer(0) // turn off the buzzer
	
	end macro_command

Name	TRACE
Syntax	TRACE(*format, argument*)
Description	Use this function to send specified string to the EasyDiagnoser / cMT Diagnoser. Users can print out the current value of variables during run-time of macro for debugging. When TRACE encounters the first format specification (if any), it converts the

	value of the first argument after format and outputs it accordingly. format refers to the format control of output string. A format specification, which consists of optional (in [ ]) and required fields (in red font), has the following form: %[flags] [width] [.precision] type Each field of the format specification is described as below: *flags* (optional): - : Aligns left. When the value has fewer characters than the specified width, it will be padded with spaces on the left. + : Precedes the result with a plus or minus sign (+ or -) *width* (optional): A nonnegative decimal integer controlling the minimum number of characters printed. *precision* (optional): A nonnegative decimal integer which specifies the precision and the number of characters to be printed. *type*: C or c : specifies a single-byte character d : signed decimal integer i : signed decimal integer o : unsigned octal integer u : unsigned decimal integer X or x : unsigned hexadecimal integer lld : signed long integer (64-bit) (cMT Series only) llu : unsigned long integer (64-bit) (cMT Series only) f : signed floating-point value llf : double-precision floating-point value E or e : Scientific notation in the form "[–]d.dddd **e** [sign]ddd" , where d is a single decimal digit, dddd is one or more decimal digits, ddd is exactly three decimal digits, and sign is + or –. The length of output string is limited to 256 characters. Extra characters will be ignored. The *argument* part is optional. One format specification converts exactly one argument.

Example	macro_command main()
	char c1 = 'a'
	short s1 = 32767
	float f1 = 1.234567
	
	TRACE("The results are") // output: The results are
	TRACE("c1 = %c, s1 = %d, f1 = %f", c1, s1, f1)
	// output: c1 = a, s1 = 32767, f1 = 1.234567
	
	end macro_command

Click the icon to download the demo project. Please confirm your internet connection before downloading the demo project.

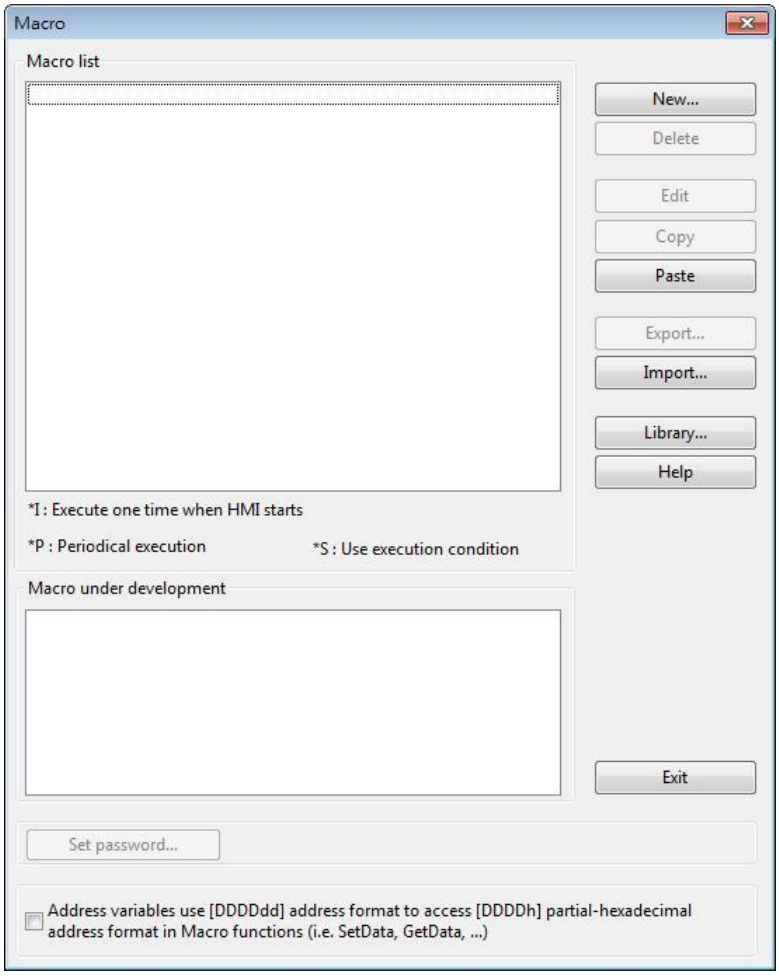
Name	GetCnvTagArrayIndex
Syntax	GetCnvTagArrayIndex(*array_index*)
Description	When a user-defined conversion tag uses array, the GetCnvTagArrayIndex() function of [Read conversion] subroutine can get the relative array index before doing conversion.
Example	Sub short newfun(short param) Int index GetCnvTagArrayIndex(index) If index is 2, the third data record in the array will be converted. return param end sub

18.8. How to Create and Execute a Macro

18.8.1. How to Create a Macro

Please follow the steps below to create a macro.

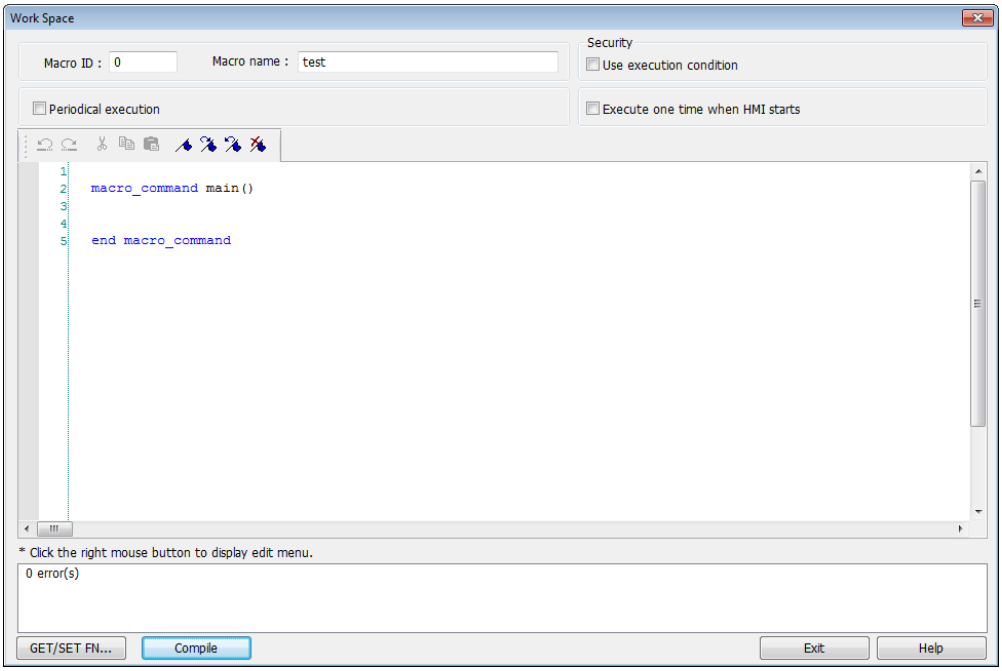
1. Click [Project] » [Macro] to open Macro Manager dialog box.



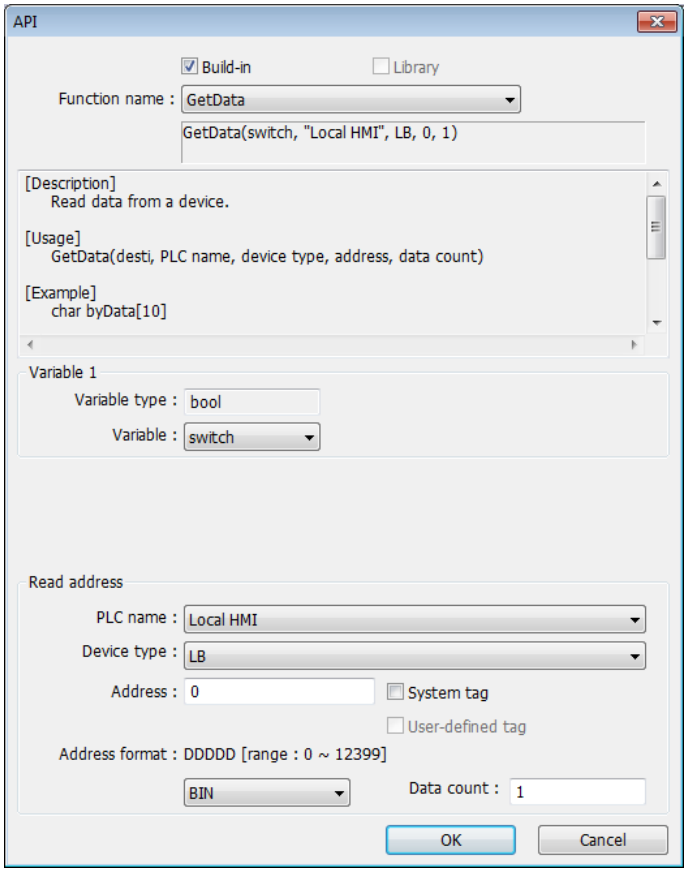
In Macro Manager, all macros compiled successfully are displayed in “Macro list”, and all macros under development or cannot be compiled are displayed in “Macro under development”. The following is a description of the various buttons.

Setting	Description
New	Opens a blank “WorkSpace” editor for creating a new macro.
Delete	Deletes the selected macro.
Edit	Opens the “WorkSpace” editor, and loads the selected macro.
Copy	Copies the selected macro into the clipboard.
Paste	Pastes the macro in the clipboard into the list, and creates a new name for the macro.
Export	Save the selected macro as *.edm file.
Import	Import an *.edm file to the project.
Library	Open Macro Function Library managing dialog.

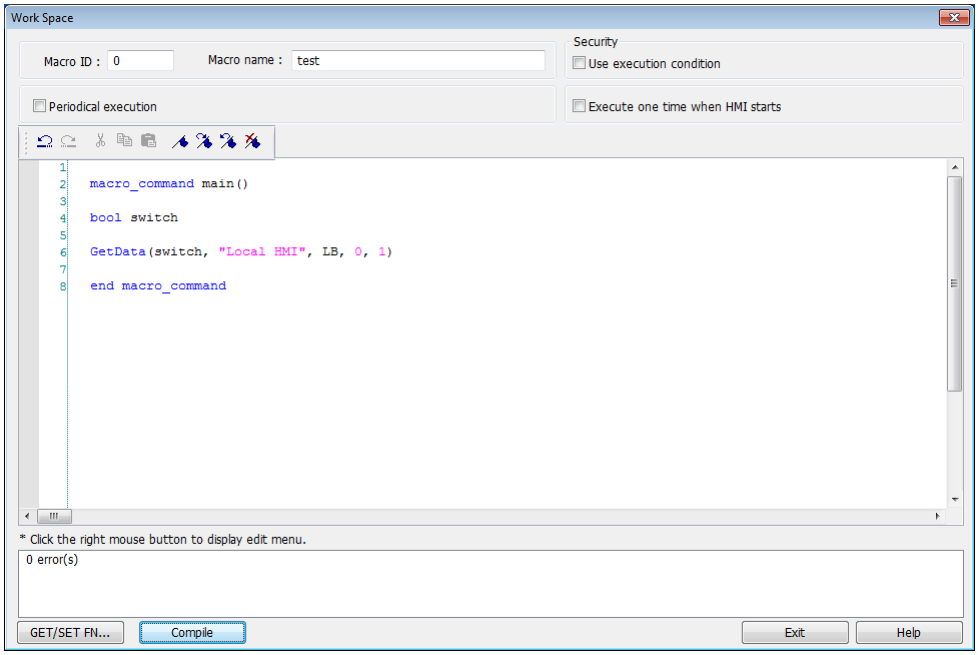
2. Press the [New] button to create an empty macro and open the macro editor. Every macro has a unique number defined at [Macro ID], and must have a macro name, otherwise an error will appear while compiling.



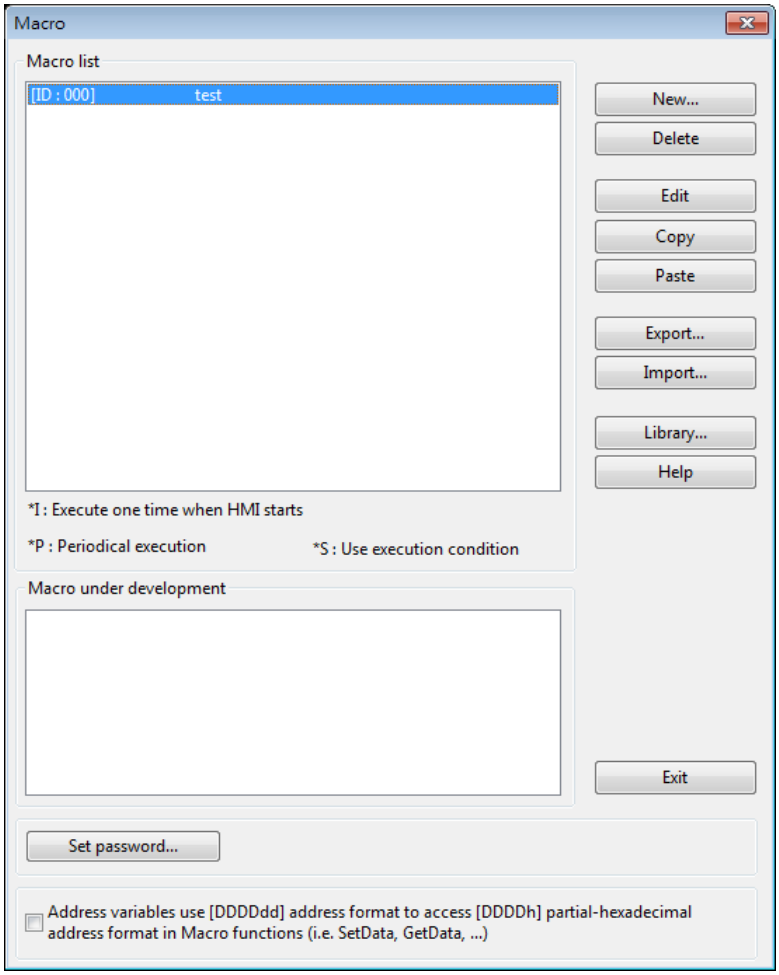
3. Design your macro. To use built-in functions (like SetData() or Getdata()), press [Get/Set FN…] button to open API dialog box and select the function and set essential parameters.



4. After the completion of a new macro, press [Compile] button to compile the macro.



5. If there is no error, press [Exit] button and a new macro “macro_test” will be in “Macro list”.



18.8.2. Execute a Macro

There are several ways to execute a macro.

- Use a PLC Control object

1. Open [PLC Control] and add one PLC Control object with the [Type of control] as [Execute macro program].
2. Select the macro in [Macro name]. Choose a bit and select a trigger condition to trigger the macro. In order to guarantee that the macro will run only once, consider latching the trigger bit, and then resetting the trigger condition within the macro.
3. Use a [Set Bit] or Toggle Switch object to change the bit to activate the macro.

- Use a [Set Bit] or Toggle Switch object

1. On the [General] tab of the [Set Bit] or [Toggle Switch] dialog box, select the [Execute Macro] option.
2. Select the macro to execute. The macro will be executed one time when the button is activated.

- Use a Function Key object

1. On the [General] tab of the [Function Key] dialog, select the [Execute Macro] option.
2. Select the macro to execute. The macro will execute one time when the button is activated.

- In macro editor, use

1. [Periodical Execution]: Macro will be triggered periodically.
2. [Execute one time when HMI starts]: Macro will be executed once HMI starts.

- In Window Settings, Macro group box

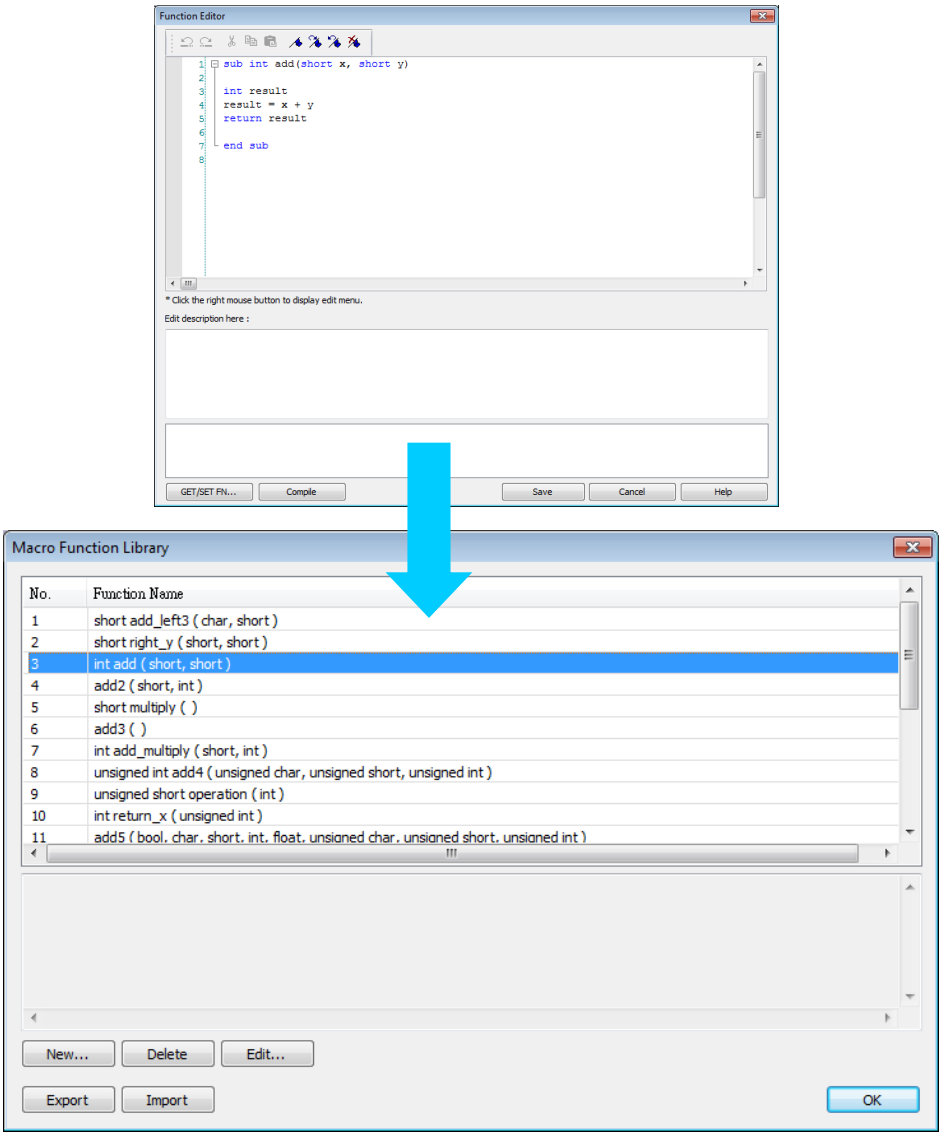
1. [Open]: When the window opens, run the selected macro once.
2. [Cycle]: When the window opens, run the selected macro every 0.5 second.
3. [Close]: When the window closes, run the selected macro once.

Click the icon to watch the demonstration film. Please confirm your internet connection before playing the film.

18.9. User Defined Macro Function

When editing Macro, to save time of defining functions, user may search for the needed from built-in Macro Function Library. However, certain functions, though frequently used, may not be found there. In this case, user may define the needed function and save it for future use. Next time when the same function is required, the saved functions can be called from [Macro

Function Library] for easier editing. Additionally, [Macro Function Library] greatly enhances the portability of user-defined functions. Before building a function please check the built-in functions or online function library to see if it exists.



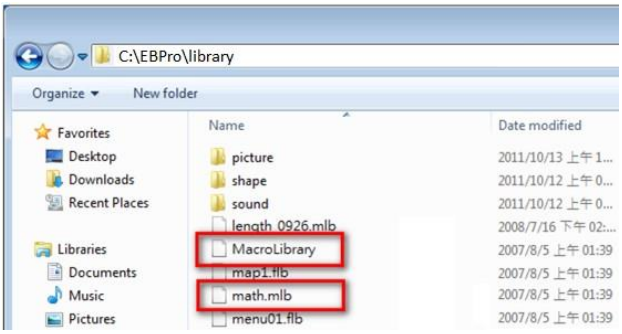
18.9.1. Import Function Library File

Open a project in HMI programming software, the default Function Library File will be read automatically and the function information will be loaded in. At this moment if a user-defined function is called, the relevant .mlb file must be imported first.

1. Default Function Library File Name: MacroLibrary (without filename extension)
2. Function Library Directory: HMI programming software installation directory\library (folder)
3. \library (folder) contains two types of function library files:
   Without filename extension: MacroLibrary, the Default Function Library for HMI programming software to read at the beginning.

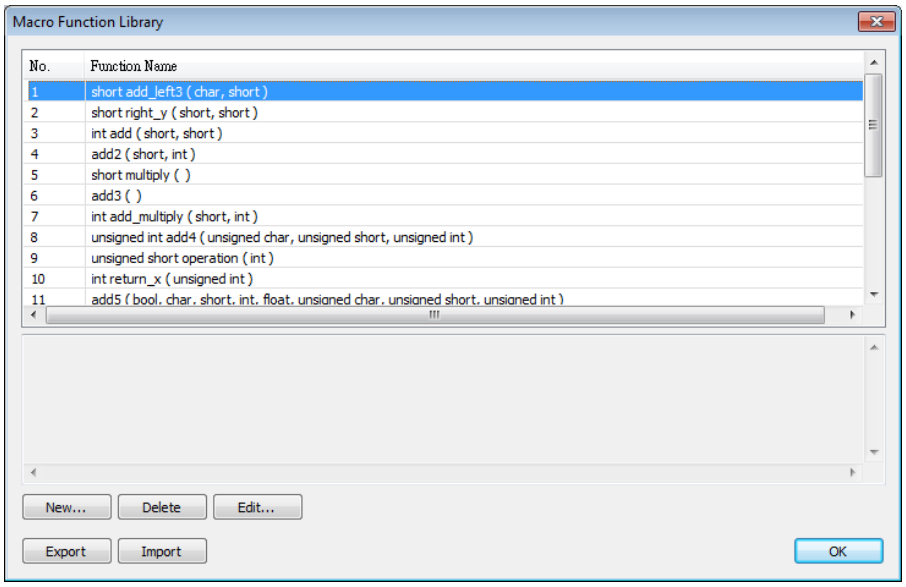
With filename extension (.mlb): Such as ”math.mlb”. The files to be read / written when users import / export. These files are portable and can be called from the folder when needed.

4. When opening HMI programming software, only the functions in Default Function Library will be loaded in, to use functions in .mlb files, please import them first.

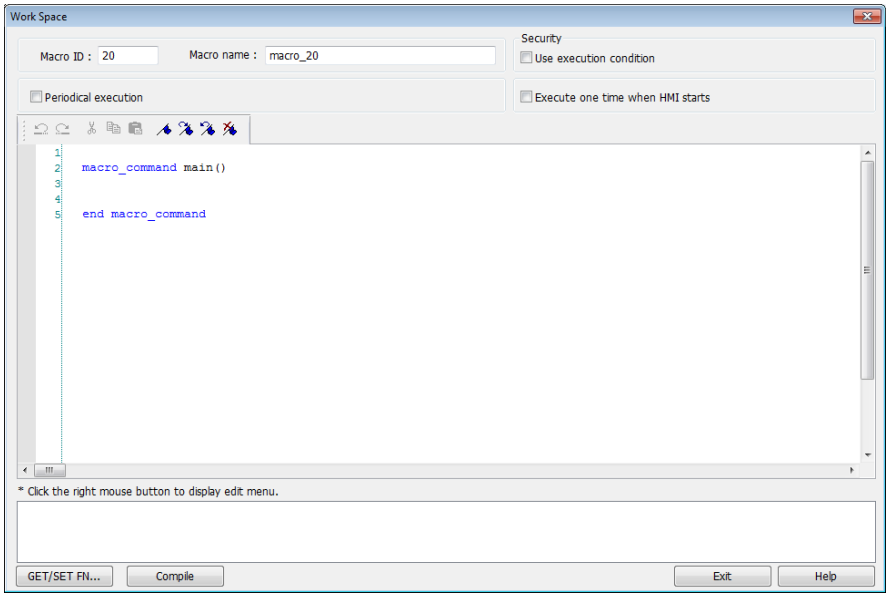


18.9.2. How to Use Macro Function Library

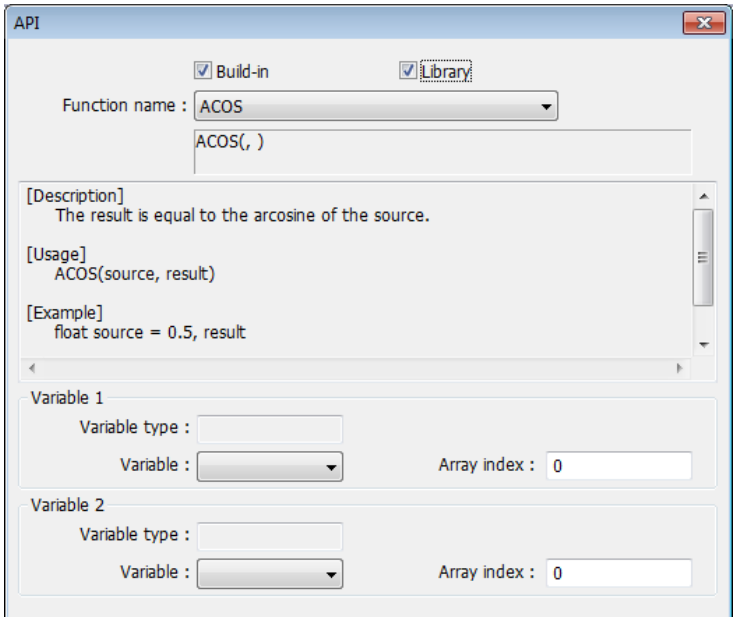
1. Select the function directly from Macro Function Library.



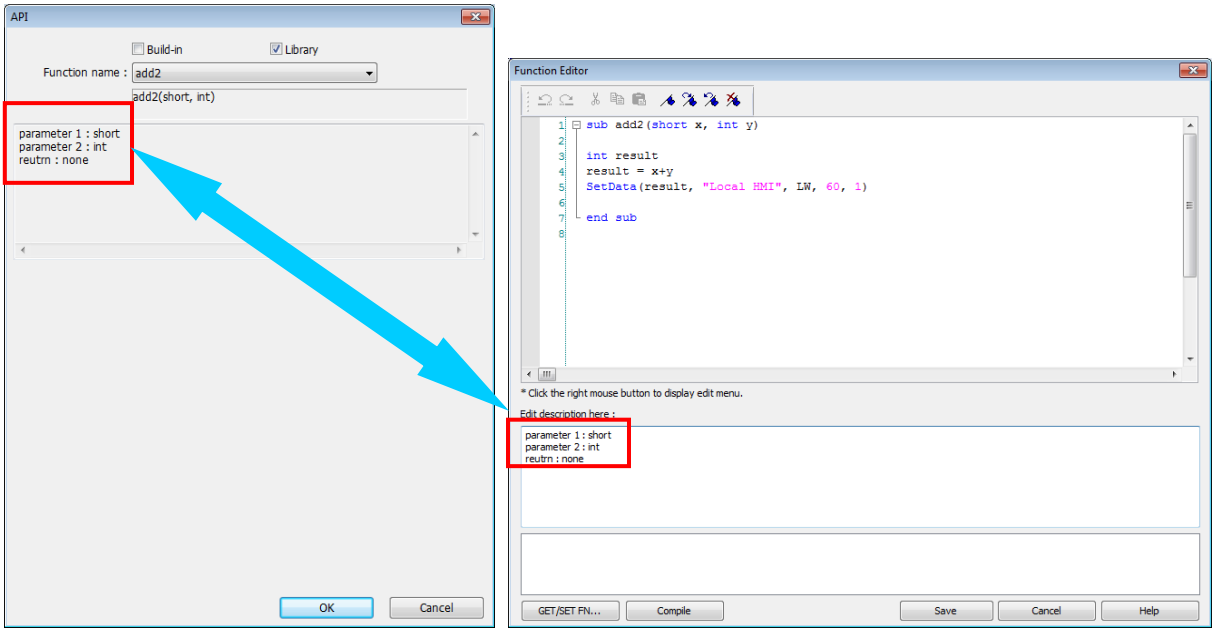
2. In WorkSpace click [GET/SET FN…] to open API dialog box.



3. At least check one from [Library] or [Build-in] and select the function to be used.



4. The description displayed in API dialog box is the same as written in Function Editor.



5. Select the function to be used, fill in the corresponding variables according to the data type.

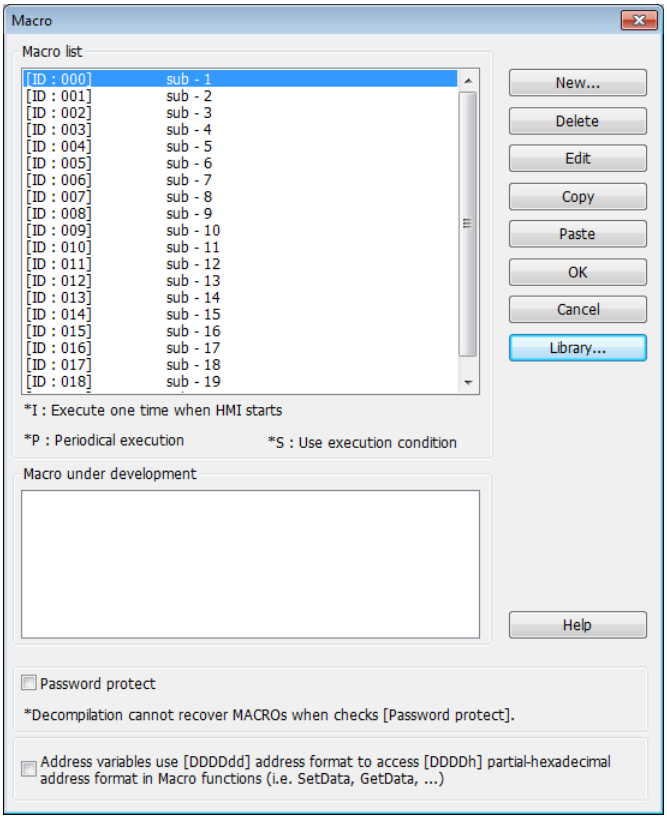
```
1
2  macro_command main()
3
4  short a
5  int b,result
6
7  add2(short, int)
8
9  end macro_command
```

```
1
2  macro_command main()
3
4  short a
5  int b,result
6
7  result = add2(a, b)
8
9  end macro_command
```

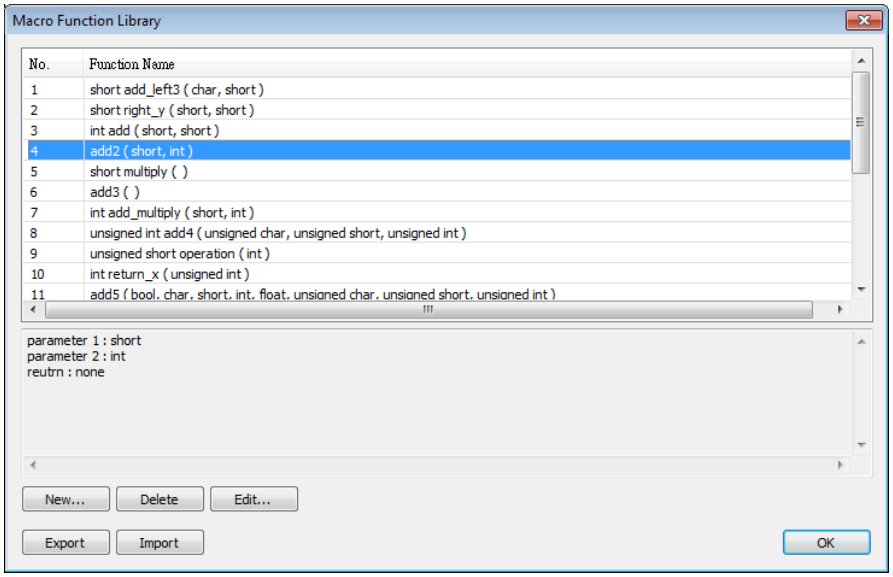
6. Upon completion of the steps above, user-defined functions can be used freely without defining the same functions repeatedly.

18.9.3. Function Library Management Interface

1. Open macro management dialog, click [Library] to open [Macro Function Library] dialog box.



2. A list of functions is shown. When the project is opened, the software will load all the functions in the Macro Function Library.



3. Each listed function has the following format:

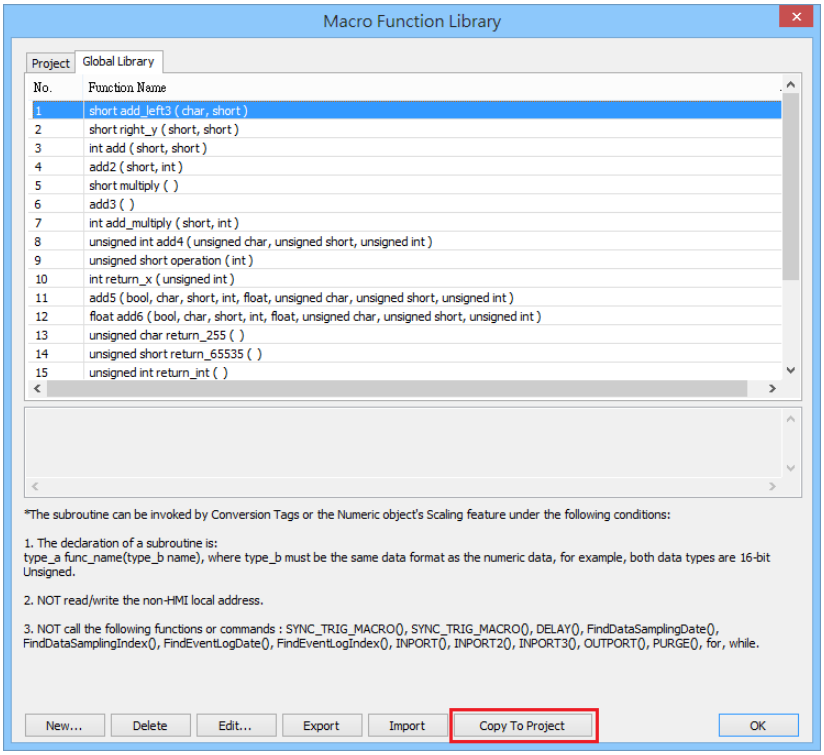
return_type function_name (*parameter_type1*, …, *parameter_typeN*)

return_type indicates the type of the return value. If this value does not exist, this column will be omitted. function_name indicates the name of the function. “N” in *parameter_typeN* stands for the number of parameter types. If this function does not

need any parameter, this column will be omitted.

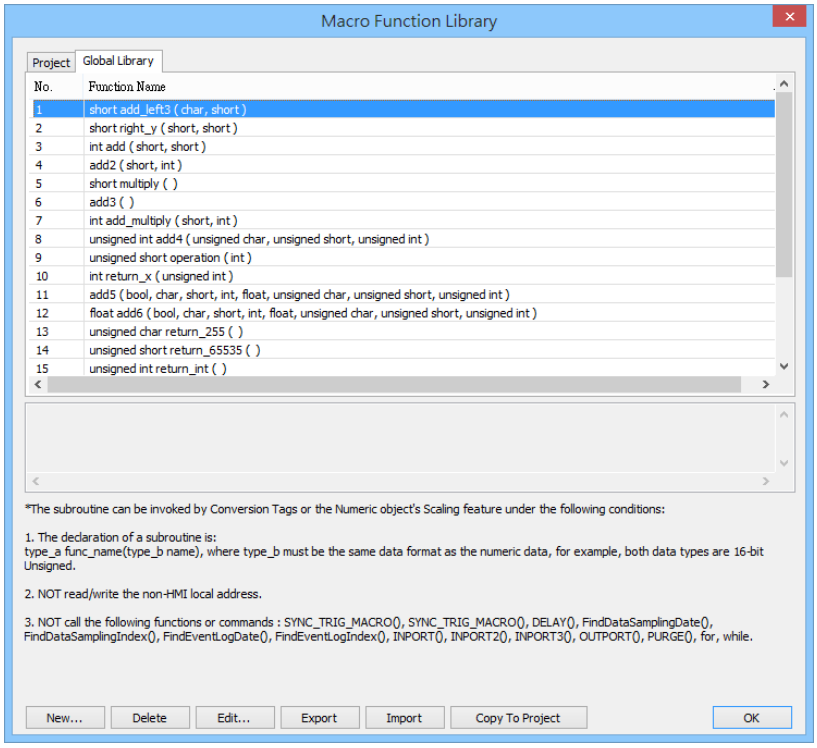
```
1  sub int ADD(int a, int b)
2    int ret
3    ret = a+b
4    return ret
5  end sub
6
```

4. Macro function can be embedded in the project file. Select the function and then click [Copy To Project], then you can find this function in [Project] tab. When opening the project on another computer, this function can still be used. When compiling the project, the .exob file will included the functions that are used. Please note that decompiling the project will only produce the macro commands that are used.

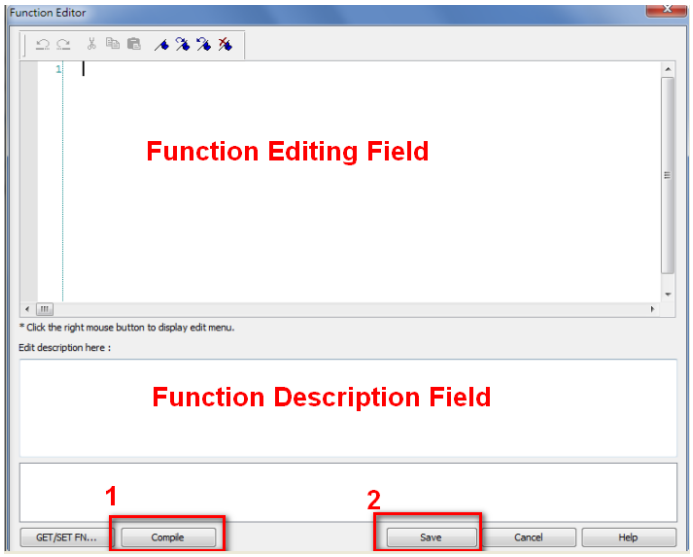


18.9.3.1. Create a Function

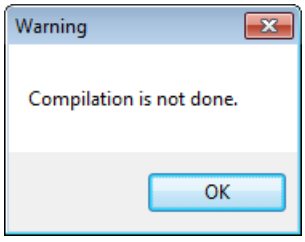
1. Click [New] to enter Function Editor.



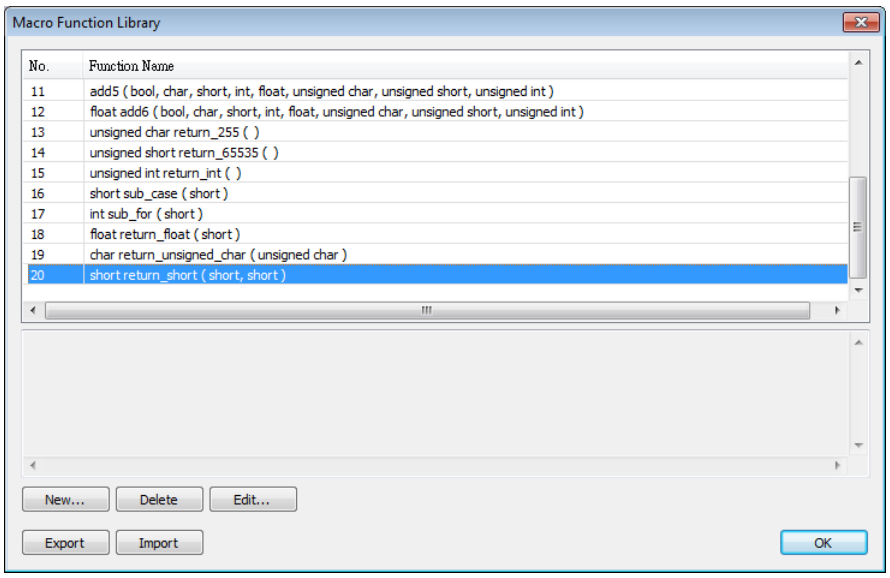
2. Edit function in Function Editor.



3. Edit the function description to describe what the specification is, how to use … etc.
4. After editing, click [Compile] and [Save] to save this function to the Library. Otherwise, a warning is shown.



5. Successfully add a function into Macro Function Library.

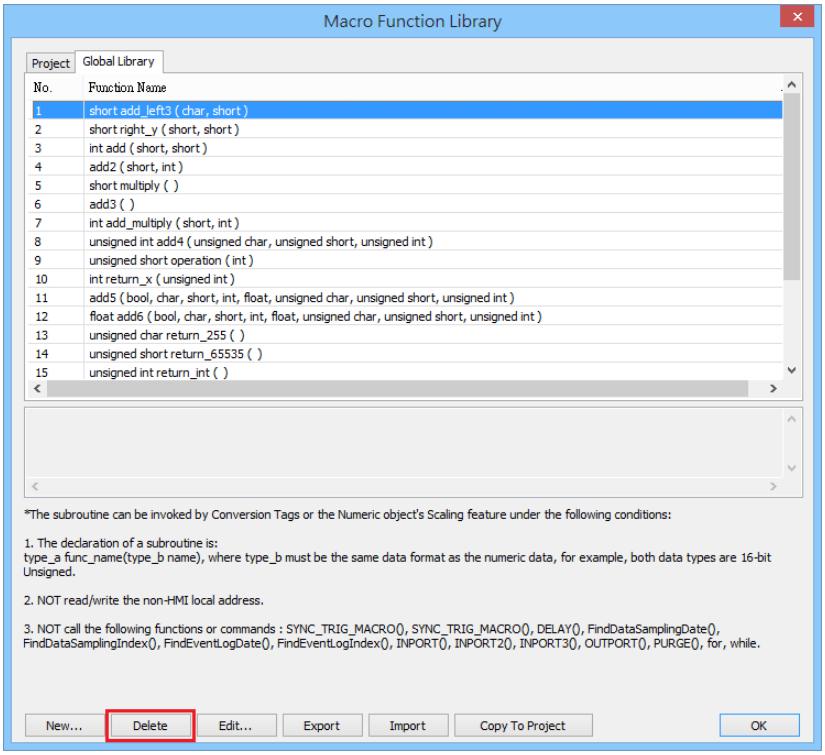


Note

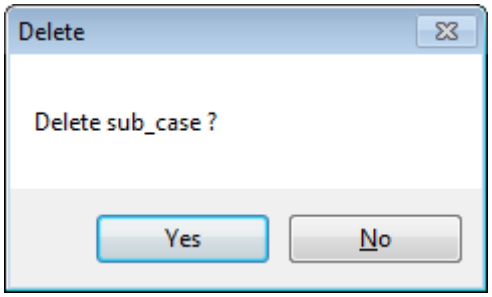
- The total size of data type can be declared in a function is 4096 bytes.
- Function name must only contain alphanumeric characters, and cannot start with a number.

18.9.3.2. Delete a Function

1. In function list select the function to be deleted and click [Delete].

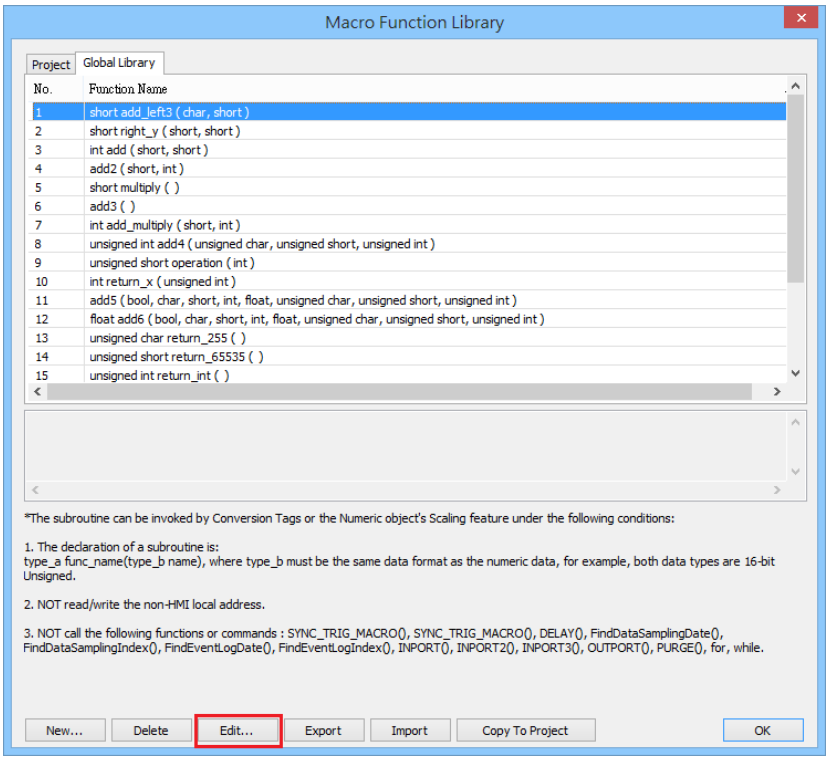


2. Click [Yes] to confirm, [No] to cancel the deletion. Click [Yes] to delete MAX_SHORT function.

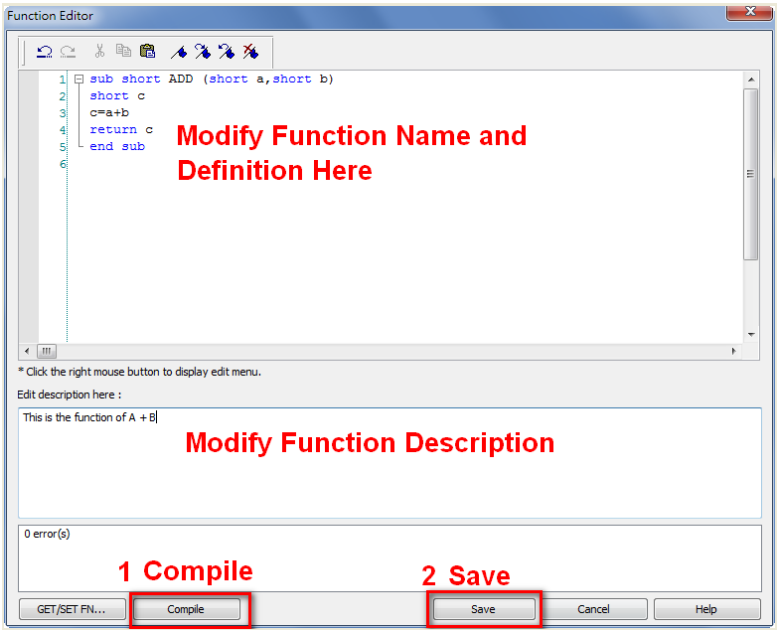


18.9.3.3. Modify a Function

1. Users can modify the functions exist in the Library.
2. Select a function to modify by clicking [Edit] to enter Function Editor.



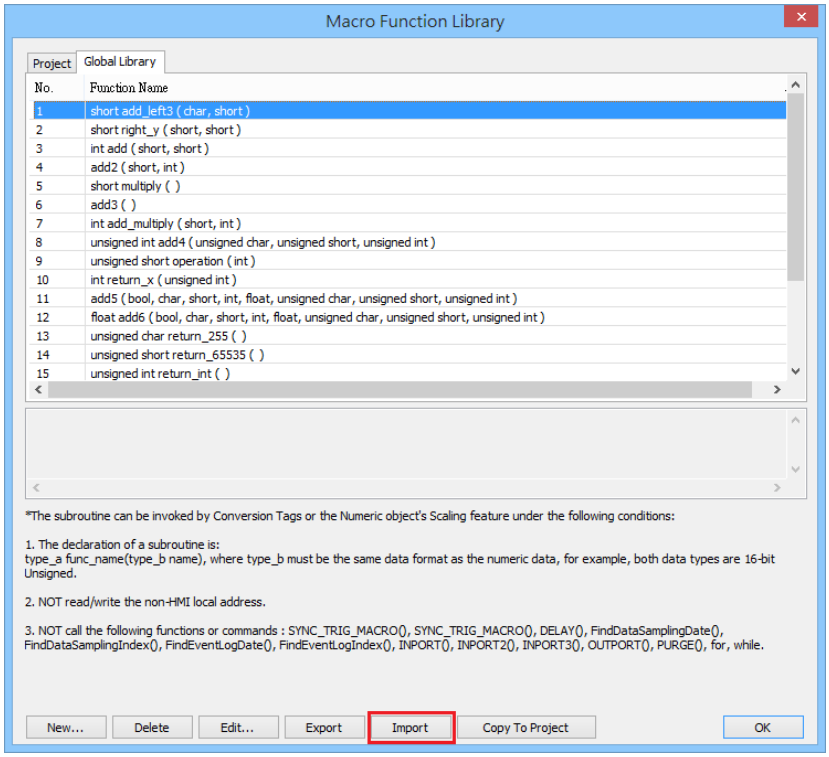
3. Double click the function to be modified can also enter Function Editor.



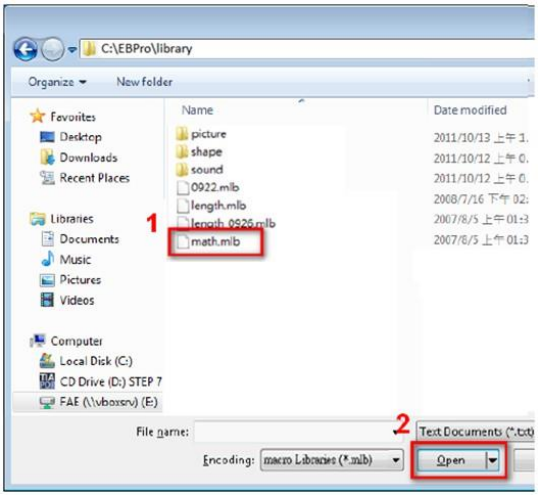
4. After modifying, [Compile] then [Save] before leaving.

18.9.3.4. Import a Function

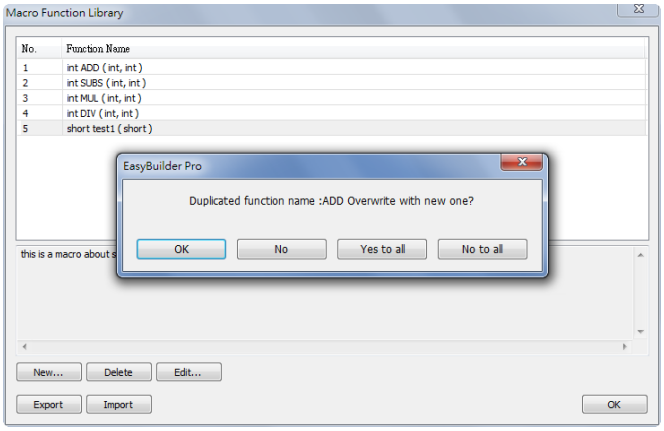
1. Functions can be imported using an external .mlb file.



2. For example, import a function library “math.mlb” which contains a function “test1”. Click [Open].



3. When importing a function which already exists in the Library, a confirmation pop-up will be shown. The buttons are:



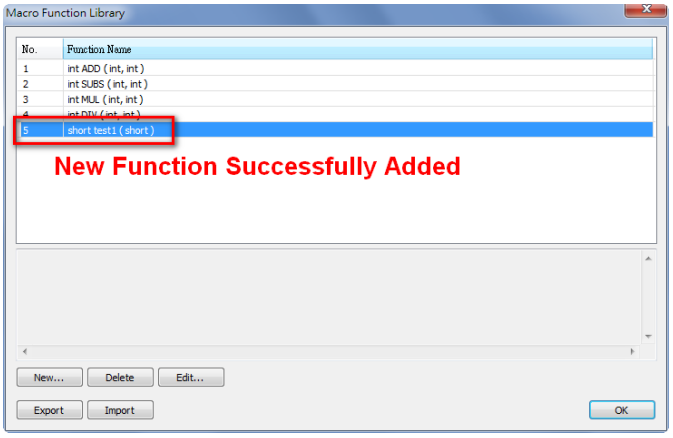
[OK]: Overwrite the existing function with the imported one.

[NO]: Cancel the importing of the function with the same name.

[Yes to all]: Overwrite using all the imported functions with the same name.

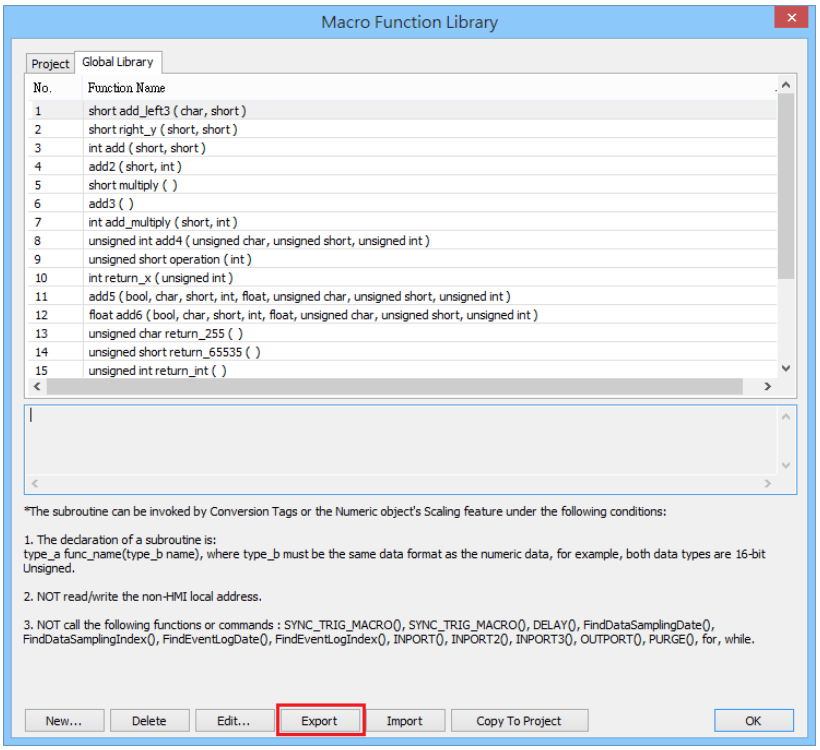
[No to all]: Cancel the importing of all the functions with the same name.

4. The imported functions will be saved in Default Function Library, so if “math.mlb” file is deleted, “test1” will still exist in the Library, even restarting EasyBuilder Pro.

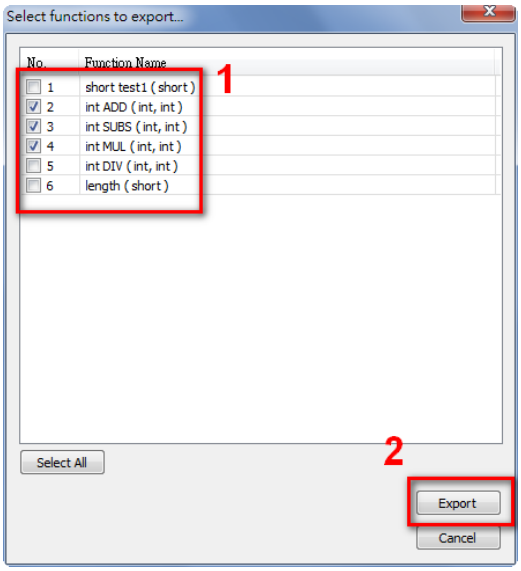


18.9.3.5. Export a Function

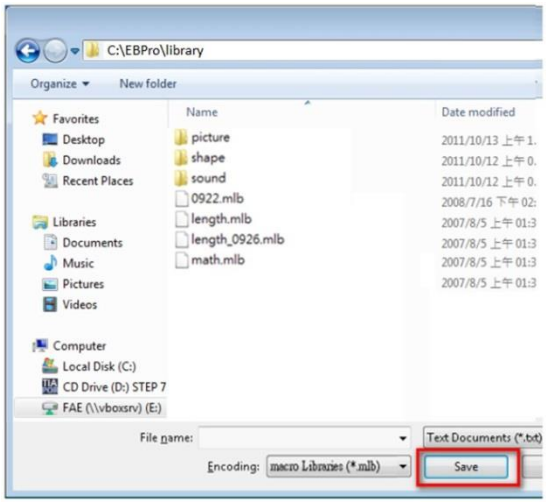
1. Export the function from Function Library and save as .mlb file. Click [Export].



2. Select the function to be exported, and click [Export].



3. A “math.mlb” file can be found under export directory. This file contains 4 functions: ADD, SUBS, MUL, and DIV.
4. The exported .mlb file can be imported on another PC. Open HMI programming software, import, then the functions in this file can be used.



18.10. Some Notes about Using the Macro

1. The maximum storage space of local variables in a macro is 4K bytes. So the maximum array size of different variable types are as follows:

char	a[4096]
bool	b[4096]
short	c[2048]
int	d[1024]
float	e[1024]
long	f[512]
double	g[512]

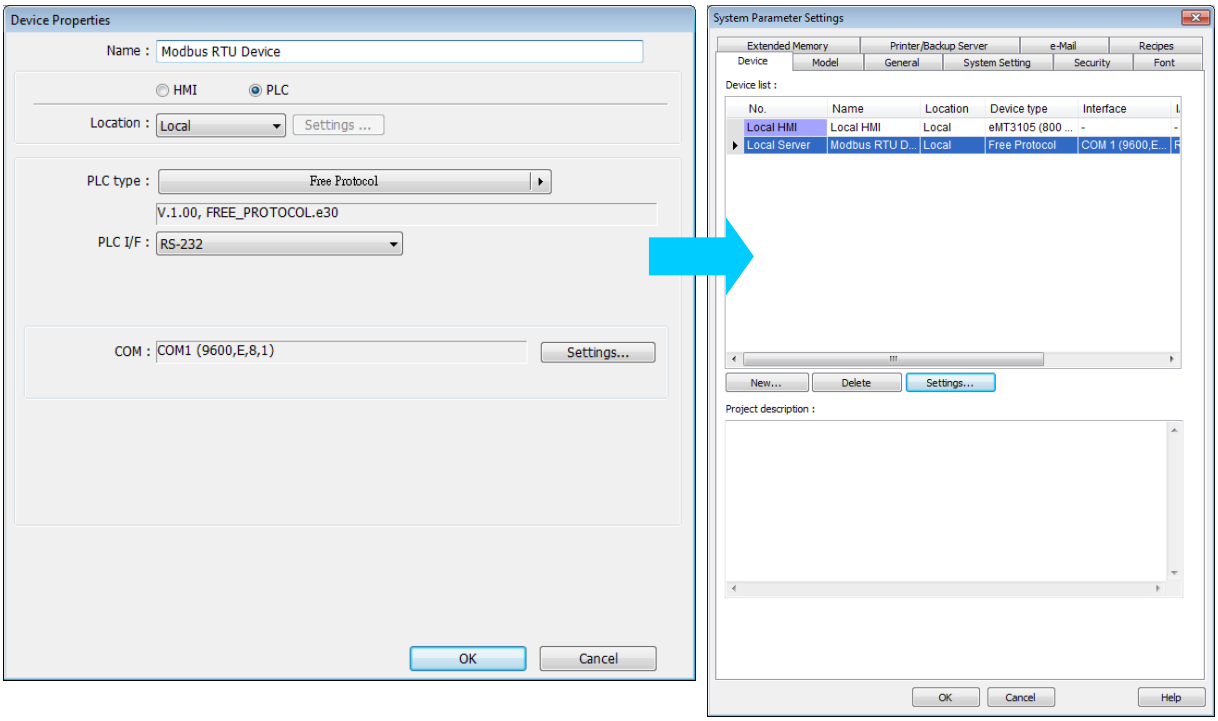
2. A maximum of 255 macros are allowed in an EasyBuilder Pro project.
3. A macro may cause the HMI unresponsive. Possible reasons are:
   - A macro contains an infinite loop with no PLC communication.
   - The size of an array exceeds the storage space in a macro.
4. The PLC communication speed affects the running time for the macro to execute. Also, too many macros may slow down the communication between HMI and PLC.

18.11. Use the Free Protocol to Control a Device

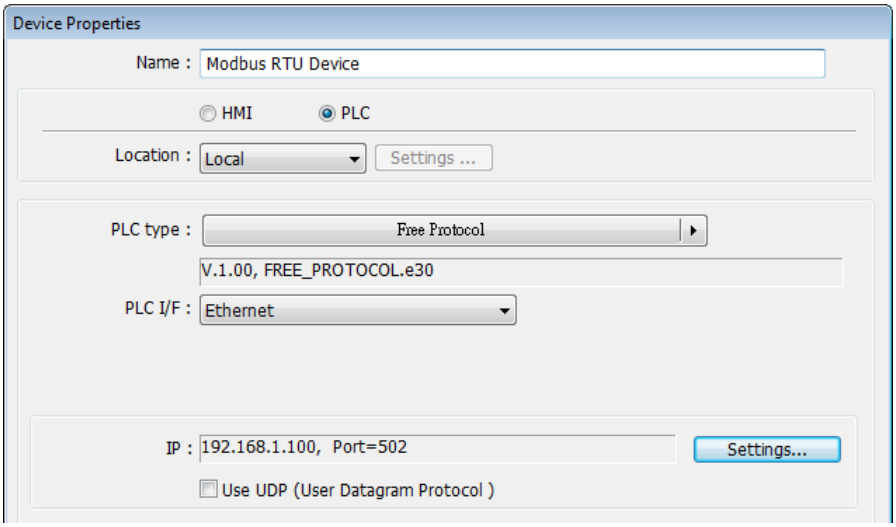
If EasyBuilder Pro does not provide a driver for a specific device, users can use OUTPORT and INPORT built-in functions to control the device. The data sent by OUTPORT and INPORT must follow the communication protocol of the device. The following example explains how to use these two functions to control a MODBUS RTU device.

1. First, create a new device in the device table. The device type of the new device is set to

"Free Protocol" and named with "MODBUS RTU device" as follows:



2. The interface of the device (PLC I/F) uses [RS-232]. If a MODBUS TCP/IP device is connected, the interface should be [Ethernet] with correct IP and port number as follows:



Suppose that the HMI will read the data of 4x_1 and 4x_2 on the device. First, utilize OUTPORT to send out a read request to the device. The format of OUTPORT is:

OUTPORT(command[start], device_name, cmd_count)

Since "MODBUS RTU device" is a MODBUS RTU device, the read request must follow MODBUS RTU protocol. The request uses"Reading Holding Registers (0x03)" command to read data. The following picture displays the content of the command. (The items of the station number (byte 0) and the last two bytes (CRC) are ignored).

Request

Function code	1 Byte	0x03
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Registers	2 Bytes	1 to 125 (0x7D)

Response

Function code	1 Byte	0x03
Byte count	1 Byte	2 x N*
Register value	N* x 2 Bytes	

*N = Quantity of Registers

Error

Error code	1 Byte	0x83
Exception code	1 Byte	01 or 02 or 03 or 04

Depending on the protocol, the content of a read command as follows (The total is 8 bytes):

command[0]: station number (BYTE 0)
command[1]: function code (BYTE 1)
command[2]: high byte of starting address (BYTE 2)
command[3]: low byte of starting address (BYTE 3)
command[4]: high byte of quantity of registers (BYTE 4)
command[5]: low byte of quantity of registers (BYTE 5)
command[6]: low byte of 16-bit CRC (BYTE 6)
command[7]: high byte of 16-bit CRC (BYTE 7)

So a read request is designed as follows:

```
char command[32]
short address, checksum

FILL(command[0], 0, 32)    //    initialize command[0]~command[31] to 0

command[0] = 0x1    //    station number
command[1] = 0x3    //    read holding registers (function code    is 0x3)

address =    //    starting address (4x_1) is 0
HIBYTE(address, command[2])
LOBYTE(address, command[3])

read_no = 2    //    the total words of rading is 2 words
HIBYTE(read_no, command[4])
LOBYTE(read_no, command[5])

CRC(command[0], checksum, 6)    //    calculate 16-bit CRC

LOBYTE(checksum, command[6])
HIBYTE(checksum, command[7])
```

Lastly, use OUPORT to send out this read request to PLC.

```
OUTPORT(command[0], “MODBUS RTU Device”, 8)    //    send read request
```

After sending out the request, use INPORT to get the response from PLC. Depending on the protocol, the content of the response is as follows (the total byte is 9):

response[0]: station number (BYTE 0)
response[1]: function code (BYTE 1)
response[2]: byte count (BYTE 2)
response[3]: high byte of 4x_1 (BYTE 3)
response[4]: low byte of 4x_1 (BYTE 4)
response[5]: high byte of 4x_2 (BYTE 5)
response[6]: high byte of 4x_2 (BYTE 6)
response[7]: low byte of 16-bit CRC (BYTE 7)
response[8]: high byte of 16-bit CRC (BYTE 8)

The format of INPORT is:

```
INPORT(response[0], “MODBUS RTU Device”, 9, return_value)    //    read reponse
```

Where the real read count is restored to the variable return_value (unit is byte). If return_value is 0, it means reading fails in executing INPORT.

According to the MODBUS RTU protocol specification, the correct response[1] must be equal to 0x03. After getting correct response, calculate the data of 4x_1 and 4x_2 and put in the data into LW-100 and LW-101 of HMI.

```
If (return_value) >0 and response[1] == 0x3) then
   read_data[0] = response[4] + (response[3] << 8) // 4x_1
   read_data[1] = response[6] + (response[5] << 8) // 4x_2

   SetData(read_data[0], “Local HMI”, LW, 100, 2)
endif
```

The complete macro is as follows:

```
// Read Holding Registers
macro_command main()

   char command[32], response[32]
   short address, checksum
   short read_no, return_value, read_data[2], i

   FILL(command[0], 0, 32)//    initialize command[0]~command[31] to 0
   FILL(response[0], 0, 32)

   command[0] = 0x1//    station number
   command[1] = 0x3//    read holding registers (function code is 0x3)

   address = 0
   address =    0//    starting address (4x_1) is 0
   HIBYTE(address, command[2])
   LOBYTE(address, command[3])

   read_no = 2/    the total words of reading is 2 words
   HIBYTE(read_no, command[4])
   LOBYTE(read_no, command[5])

   CRC(command[0], checksum, 6)//    calculate 16-bit CRC

   LOBYTE(checksum, command[6])
   HIBYTE(checksum, command[7])

   OUTPORT(command[0], "MODBUS RTU Device", 8 )//    send request
   INPORT(response[0], "MODBUS RTU Device", 9, return_value)//    read response

   if (return_value > 0 and response[1] == 0x3) then
      read_data[0] = response[4] + (response[3] << 8)//    4x_1
      read_data[1] = response[6] + (response[5] << 8)//    4x_2

      SetData(read_data[0], "Local HMI", LW, 100, 2)
   end if

   end macro_command
```

The following example explains how to design a request to set the status of 0x_1. The request uses ”Write Single Coil(0x5)” command.

Request

Function code	1 Byte	**0x05**
Output Address	2 Bytes	0x0000 to 0xFFFF
Output Value	2 Bytes	0x0000 or 0xFF00

Response

Function code	1 Byte	**0x05**
Output Address	2 Bytes	0x0000 to 0xFFFF
Output Value	2 Bytes	0x0000 or 0xFF00

Error

Error code	1 Byte	**0x85**
Exception code	1 Byte	01 or 02 or 03 or 04

The complete macro is as follows:

```
//    Write Single Coil (ON)
macro_command main()

char command[32], response[32]
short address, checksum
short i, return_value

FILL(command[0], 0, 32)//    initialize command[0]~ command[31] to 0
FILL(response[0], 0, 32)

command[0] = 0x1//    station number
command[1] = 0x5//    function code : write single coil

address = 0
HIBYTE(address, command[2])
LOBYTE(address, command[3])

command[4] = 0xff//    force 0x_1 on
command[5] = 0

CRC(command[0], checksum, 6)

LOBYTE(checksum, command[6])
HIBYTE(checksum, command[7])

OUTPORT(command[0], "MODBUS RTU Device", 8)//    send request
INPORT(response[0], "MODBUS RTU Device", 8, return_value)//    read response

end macro_command
```

Click the icon to download the demo project. Please confirm your internet connection before downloading the demo project.

18.12. Compiler Error Message

- Error Message Format

error C# : error description
(# is the error message number)
Example: error C37 : undeclared identifier : i
When there are compile errors, the description of the error can be found by the compiler error message number.

- Error Description

(C1) syntax error：'identifier'
There are many possibilities to cause compiler error.

For example:
```
macro_command main()
char i, 123xyz    //    this is an unsupported variable name
end macro_command
```

(C2) 'identifier' used without having been initialized
Macro must define the size of an array during declaration.

For example:
```
macro_command main()
char i
int g[i]    //    i must be a numeric constant
end macro_command
```

(C3) redefinition error : 'identifier'
The name of variable and function within its scope must be unique.

For example:
```
macro_command main()
int g[10]，g    //    error
end macro_command
```

(C4) function name error : 'identifier'
Reserved keywords and constant cannot be the name of a function

For example：
```
sub int if()    // error
```

(C5) parentheses have not come in pairs
Statement missing “(“ or “)”
For example：
macro_command main) // missing “(“

(C6) illegal expression without matching ‘if’
Missing expression in “if” statement

(C7) illegal expression (no ‘then’) without matching ‘if’
Missing “then” in “if” statement

(C8) illegal expression (no ‘end if’)
Missing “end if”

(C9) illegal ‘end if’ without matching ‘if’
Unfinished “If’ statement before “End If”

(C10) illegal ‘else’
The format of “if” statement is :
if [logic expression] then
[else [if [logic expression] then]]

end if

Any format other than this format will cause a compile error.

(C17) illegal expression (no 'for') without matching ‘next’
“for” statement error : missing “for” before “next”

(C18) illegal variable type (not integer or char)
Should be integer or char variable

(C19) variable type error
Missing assign statement

(C20) must be keyword ‘to’ or ‘down’
Missing keyword “to” or “down”

(C21) illegal expression (no 'next')
The format of “for” statement is:
for [variable] = [initial value] to [end value] [step]

next [variable]

Any format other than this format will cause a compile error.

(C22) ‘wend’ statement contains no ‘while’
“While” statement error : missing “while” before “Wend”

(C23) illegal expression without matching ‘wend’
The format of “While” statement is :

while [logic expression]

wend

Any format other than this format will cause a compile error.

(C24) syntax error : ‘break’
“break” statement can only be used in “for”, “while” statement.

(C25) syntax error : ‘continue’
“continue” statement can only be used in “for” statement, or “while” statement.

(C26) syntax error
Error in expression.

(C27) syntax error
The mismatch of an operation object in expression can cause a compile error.

For example :
macro_command main( )
int a, b
for a = 0 to 2
b = 4 + xyz // illegal : xyz is undefined
next a

end macro_command

(C28) must be ‘macro_command’
There must be ’macro_command’

(C29) must be key word ‘sub’
The format of function declaration is:

sub [data type] function_name(…)
……..
end sub

For example::
sub int pow(int exp)
…….
end sub

format other than this format will cause a compile error.

(C30) number of parameters is incorrect
Mismatch of the number of parameters

(C31) parameter type is incorrect
Mismatch of data type of parameter. When a function is called, the data type and the number of parameters should match the declaration of function, otherwise it will cause a compile error.

(C32) variable is incorrect
The parameters of a function must be equivalent to the arguments passing to a function to avoid compile error.

(C33) function name : undeclared function

(C34) expected constant expression
Illegal array index format.

(C35) invalid array declaration

(C36) array index error

(C37) undeclared identifier : i 'identifier'
Any variable or function should be declared before use.

(C38) un-supported PLC data address
The parameter of GetData(…) , SetData(…) should be legal PLC address. If the address is illegal, this error message will be shown.

(C39) 'idenifier' must be integer, char or constant
The format of array is:
Declaration: array_name[constant] (constant is the size of the array)
Usage: array_name[integer, character or constant]
Any format other than this format will cause a compile error.

(C40) execution syntax should not exist before variable declaration or constant definition

For example :

```
macro_command main(    )
int a, b
for a = 0 To 2
    b = 4 + a
int h , k    //    illegal – definitions must occur before any statements or expressions
             //    for example, b = 4 + a
next a
end macro_command
```

(C41) float variables cannot be contained in shift calculation

(C42) function must return a value

(C43) function should not return a value

(C44) float variables cannot be contained in calculation

(C45) PLC address error

(C46) array size overflow (max. 4k)

(C47) macro command entry function is not only one

(C48) macro command entry function must be only one
The only one main entrance of macro is :
macro_command function_name( )
end macro_command

(C49) an extended addressee's station number must be between 0 and 255

For example :
SetData(bits[0] , "PLC 1", LB , 300#123, 100)
// illegal : 300#123 means the station number is 300, but the maximum is 255

(C50) an invalid PLC name
PLC name is not defined in the device list of system parameters.

(C51) macro command do not control a remote device
A macro can only control a local machine.

For example :
SetData(bits[0] , "PLC 1", LB , 300#123, 100)
"PLC 1" is connected with the remote HMI ,so it cannot work.

18.13. Sample Macro Code

- "for" statement and other expressions (arithmetic, bitwise shift, logic and comparison)

```
macro_command main()
int a[10], b[10], i

b[0] = (400 + 400 << 2) / 401
b[1] = 22 *2 - 30 % 7
b[2] = 111 >> 2
b[3] = 403 > 9 + 3 >= 9 + 3 < 4 + 3 <= 8 + 8 == 8
b[4] = not 8 + 1 and 2 + 1 or 0 + 1 xor 2
b[5] = 405 and 3 and not 0
b[6] = 8 & 4 + 4 & 4 + 8 | 4 + 8 ^ 4
b[7] = 6 – (~4)
b[8] = 0x11
```

```
b[9] = 409

for i = 0 to 4 step 1
  if (a[0] == 400) then
        GetData(a[0], "Device 1", 4x, 0,9)
        GetData(b[0],”Device 1", 4x, 11,10)
  end If
  next i
  end macro_command
```

- “while”, “if” and “break” statements

```
macro_command main()
int b[10], i
i = 5
while i == 5 - 20 % 3
      GetData(b[1], "Device 1", 4x, 11, 1)

      if b[1] == 100 then
            break
      end if
wend
end macro_command
```

- Global variables and function call

```
char g
sub int fun(int j, int k)
      int y

      SetData(j, "Local HMI", LB, 14, 1)
      GetData(y, "Local HMI", LB, 15, 1)
      g = y

      return y
end Sub

macro_command main()
      int a, b, i
```

```
a = 2
b = 3
    i = fun(a, b)
    SetData(i, "Local HMI", LB, 16, 1)
end macro_command
```

- “if” statement

```
macro_command main()
    int k[10], j

    for j = 0 to 10
        k[j] = j
    next j

    if k[0] == 0 then
    SetData(k[1], "Device 1", 4x, 0, 1)
    end if

if k[0] == 0 then
        SetData(k[1], "Device 1", 4x, 0, 1)
    else
    SetData(k[2], "Device 1", 4x, 0, 1)
end if

    if k[0] == 0 then
        SetData(k[1], "Device 1", 4x, 1, 1)
    else if k[2] == 1 then
    SetData(k[3], "Device 1", 4x, 2, 1)
end If

    if k[0] == 0 then
    SetData(k[1], "Device 1", 4x, 3, 1)
else if k[2] == 2 then
    SetData(k[3], "Device 1", 4x, 4, 1)
else
    SetData(k[4], "Device 1", 4x, 5, 1)
end If
end macro_command
```

- “while” and “wend” statements

```
macro_command main()
char i = 0
int a[13], b[14], c = 4848

b[0] = 13

while b[0]
        a[i] = 20 + i * 10

    if a[i] == 120 then
        c =200
            break
    end if

    i = i + 1
wend

SetData(c, "Device 1", 4x, 2, 1)
end macro_command
```

- “break” and “continue” statements

```
macro_command main()
char i = 0
int a[13], b[14], c = 4848

b[0] = 13

while b[0]
        a[i] = 20 + i * 10

        if a[i] == 120 then
        c =200
        i = i + 1
            continue
        end if
```

```
        i = i + 1

        if c == 200 then
            SetData(c, "Device 1", 4x, 2, 1)
        break
            end if
    wend
    end macro_command
```

- Array

```
    macro_command main()
    int a[25], b[25], i

    b[0] = 13

    for i = 0 to b[0] step 1
        a[i] = 20 + i * 10
    next i

    SetData(a[0], "Device 1", 4x, 0, 13)
    end macro_command
```

- Syntax for placing quotation marks in a string applies to variable declaration and function's argument.

```
    macro_command main()
    char data[40]= "\"Note\" "

    StringCopy("This is a \"test\" for weintek", data[7])
    //The string contains "Note" This is a "test" for weintek
    end macro_command
```

18.14. Macro TRACE Function

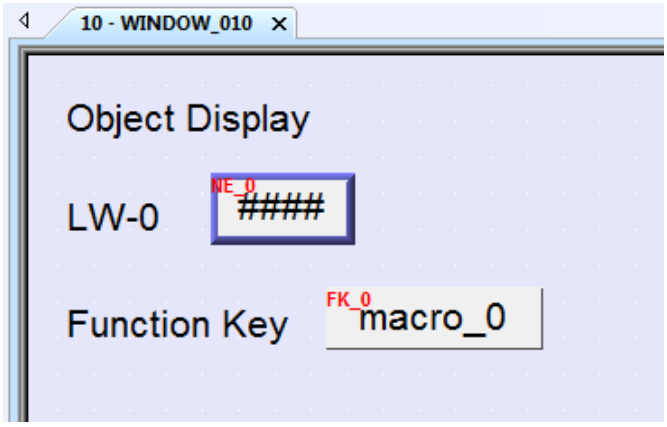
TRACE function can be used with EasyDiagnoser / cMT Diagnoser to show the current content of the variables. For users of cMT series, a better, more straightforward way would be to use the Macro Debugger in the cMT Diagnoser for debugging. The use of TRACE function is not required.

The following example explains how to use TRACE function in macro and then use EasyDiagnoser for monitoring.

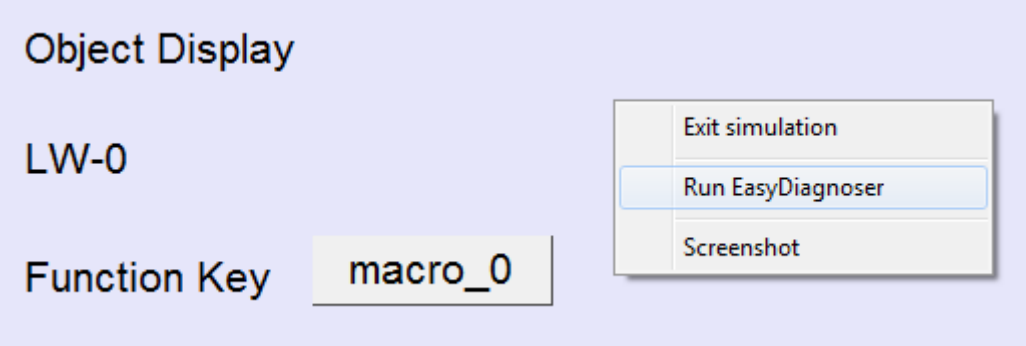
1. First of all, add a new macro “macro_0” in the project, and in “macro_0” add TRACE (“LW = %d”, a). “%d” indicates display current value of LW in decimal format. The content of “macro_0” is as follows:

```
1
2   macro_command main()
3
4   short a
5   GetData(a, "Local HMI", LW, 0, 1)
6   a=a+1
7   SetData(a, "Local HMI", LW, 0, 1)
8   TRACE ("LW0 = %d" , a)
9
10  end macro_command
```

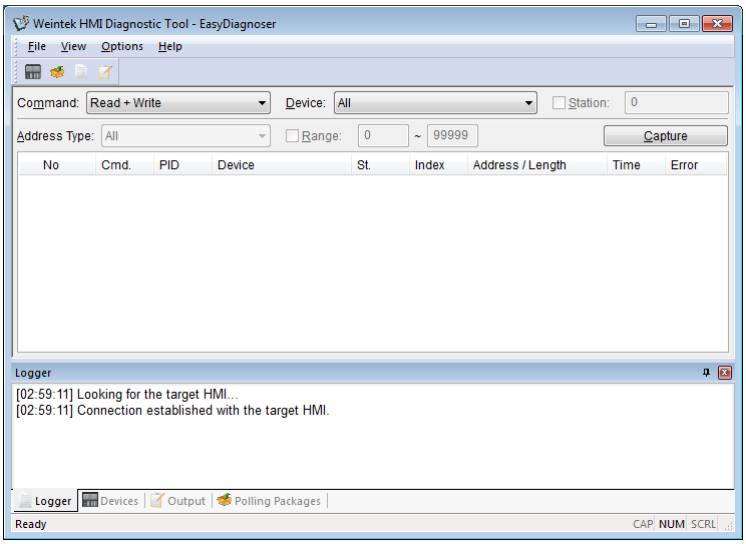
2. Secondly, add a Numeric Display object and a Function Key object in window no. 10 of the project. The Function Key object is used to execute macro_0.



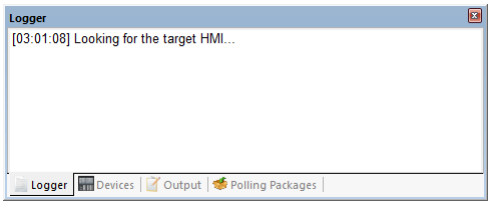
3. Lastly, compile the project and execute [Off-line simulation] or [On-line simulation].
4. When processing simulation on PC, right click and select “Run EasyDiagnoser” in the pop-up menu.



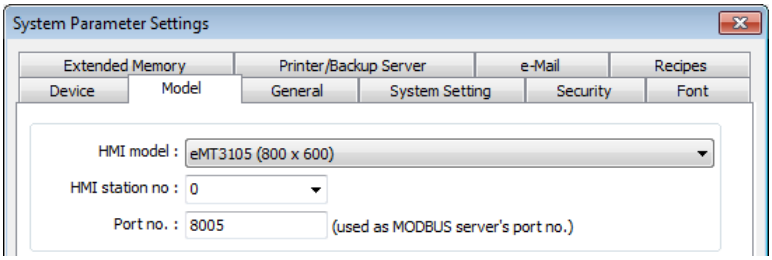
5. Afterwards, EasyDiagnoser will be started. [Logger] window displays whether EasyDiagnoser is able to connect with the HMI to be watched or not. [Output] window displays the output of the TRACE function. The illustration below shows that EasyDiagnoser succeeds in connecting with HMI.



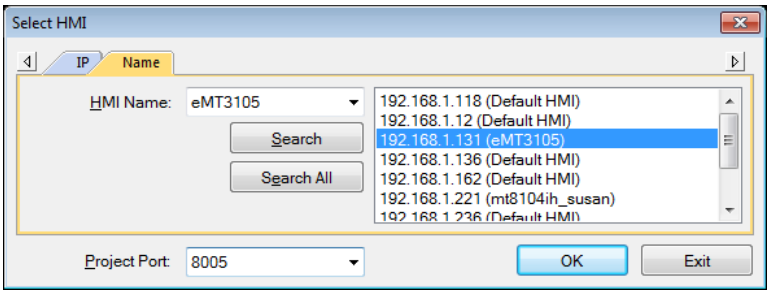
When EasyDiagnoser is not able to connect with HMI, [Logger] window displays content as shown in the following figure:



6. The possible reason of not being able to get connection with HMI can be failure in executing simulation on PC. Another reason is that the Port No. used in project for simulation on PC is incorrect (or occupied by system). Please change Port No. as shown, compile project then do simulation again.



7. In EasyDiagnoser, the Port No. should be set the same as the Port No. in the project.



The three consecutive ports of the project port no. are preserved for HMI communication. In the setting above as an example, Port No. is set as 8005. Port 8005, 8006 and 8007 should be reserved. In this case when executing simulation on PC, please make sure that these ports are not occupied by other programs.

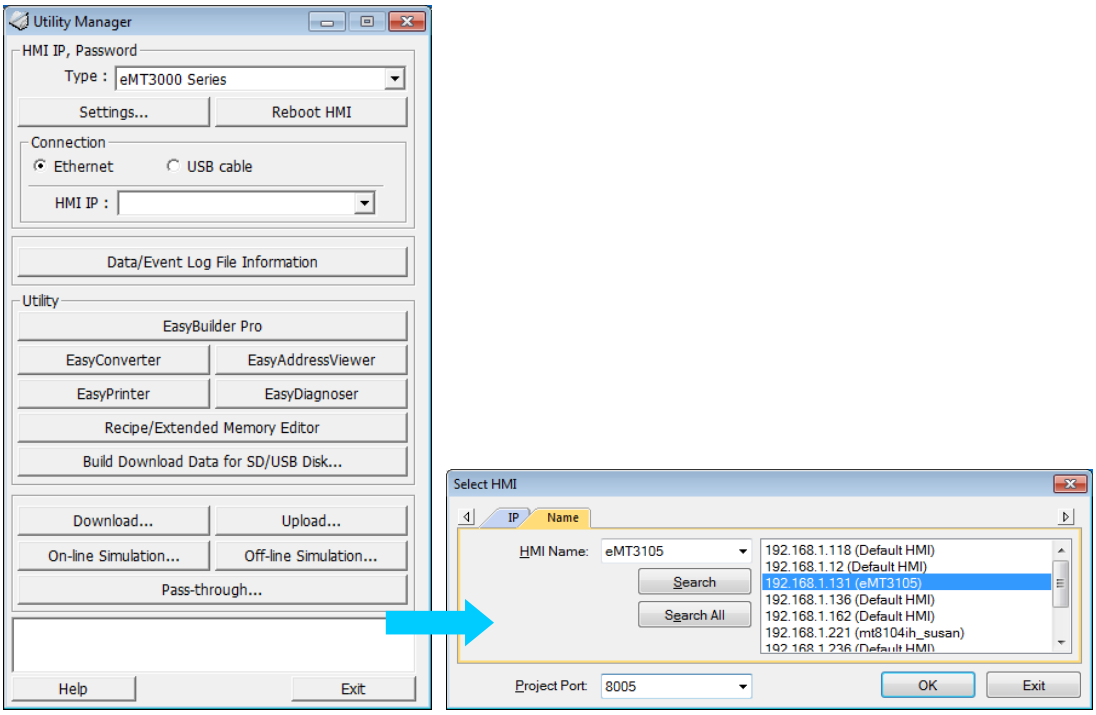
TRACE Syntax List

Name	TRACE
Syntax	TRACE(format, argument)
Description	Use this function to send specified string to the EasyDiagnoser / cMT Diagnoser. Users can print out the current value of variables during run-time of macro for debugging. When TRACE encounters the first format specification (if any), it converts the value of the first argument after format and outputs it accordingly. format refers to the format control of output string. A format specification, which consists of optional (in [ ]) and required fields (in red font), has the following form: %[flags] [width] [.precision] type Each field of the format specification is described as below: *flags* (optional): - : Aligns left. When the value has fewer characters than the specified width, it will be padded with spaces on the left. + : Precedes the result with a plus or minus sign (+ or -) *width* (optional): A nonnegative decimal integer controlling the minimum number of characters printed. *precision* (optional): A nonnegative decimal integer which specifies the precision and the number of characters to be printed. *type*: C or c : specifies a single-byte character d : signed decimal integer

	i : signed decimal integer o : unsigned octal integer u : unsigned decimal integer X or x : unsigned hexadecimal integer lld : signed long integer (64-bit) (cMT Series only) llu : unsigned long integer (64-bit) (cMT Series only) f : signed floating-point value llf : double-precision floating-point value E or e : Scientific notation in the form "[–]d.dddd **e** [sign]ddd" , where d is a single decimal digit, dddd is one or more decimal digits, ddd is exactly three decimal digits, and sign is + or –. The length of output string is limited to 256 characters. Extra characters will be ignored. The *argument* part is optional. One format specification converts exactly one argument.

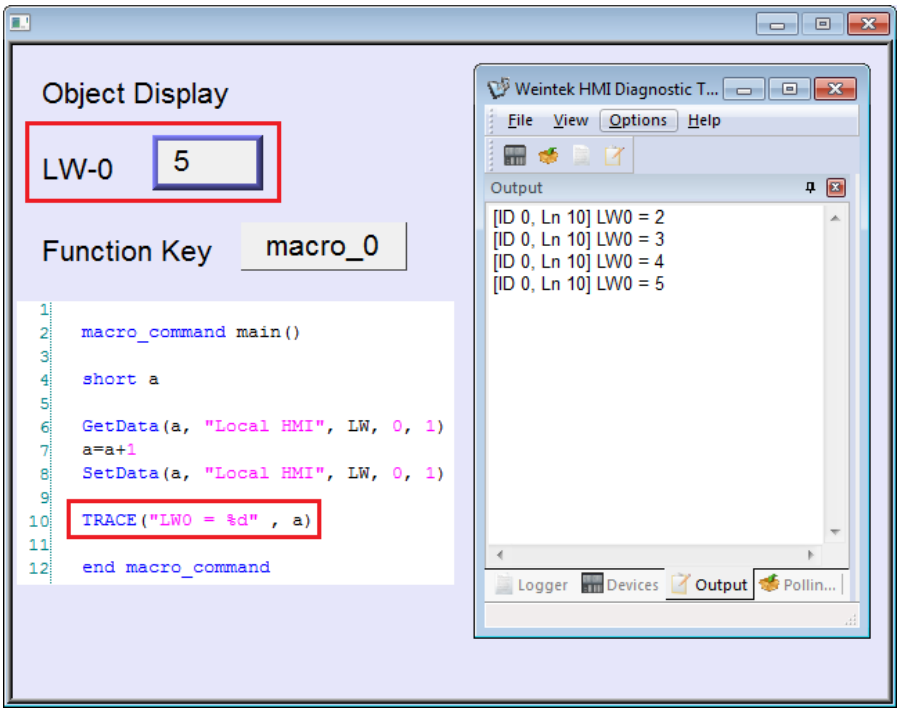
Example	macro_command main()
	char c1 = ' a'
	short s1 = 32767
	float f1 = 1.234567
	
	TRACE("The results are") // output: The results are
	TRACE("c1 = %c, s1 = %d, f1 = %f", c1, s1, f1)
	// output: c1 = a, s1 = 32767, f1 = 1.234567
	
	end macro_command

8. Use LB-9059 to disable MACRO TRACE function (when ON). When set ON, the output message of TRACE won't be sent to EasyDiagnoser.
9. Users can directly execute EasyDiagnoser.exe from Utility Manager. In Utility Manager, current HMI on line will be listed; users can simply select the HMI to be watched. Please note that Project Port should be the same as Port No. used in project file.



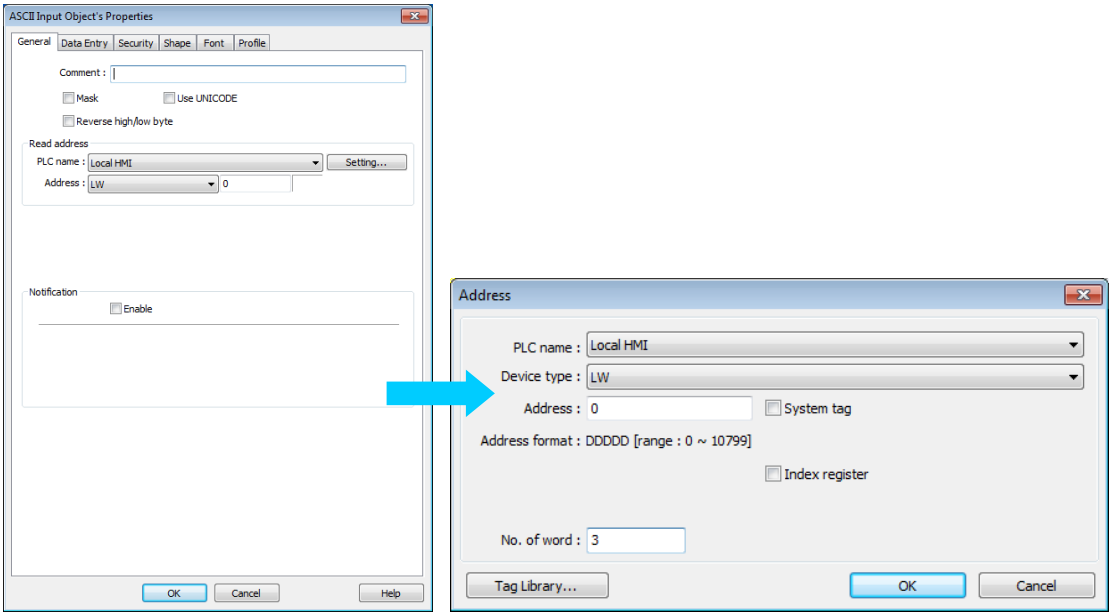
10. Download the project to HMI and start the project. If EasyDiagnoser is unable to get connection with the HMI to be watched, it is possible that HMI power is not ON, or Port No. is incorrect. This may cause EasyDiagnoser to connect then disconnect with HMI continuously. Please check the Port No. in EasyDiagnoser settings.

11. When EasyDiagnoser succeeds in connecting with HMI, simply execute macro_0, [Output] window will then display the output of the TRACE function.



18.15. Example of String Operation Functions

String operation functions are added to macro to provide a convenient way to operate strings. The term “string” means a sequence of ASCII characters, and each of them occupies 1 byte. The sequence of characters can be stored into 16-bit registers with least significant byte first. For example, create an ASCII Input object and setup as follows:



Run simulation and input “abcdef”:

The string “abcdef” is stored in LW-0~LW-2 as follows (LB represents low byte and HB represents high byte):

	HB	LB
LW0	'B'	'A'
LW1	'D'	'C'
LW2	'F'	'E'
LW3		
LW4		
LW5		

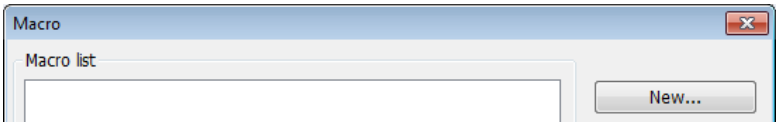
The ASCII Input object reads 1 word (2 bytes) at a time as described in the previous chapter. Suppose an ASCII Input object is set to read 3 words as shown in the above example, it can actually read at most 6 ASCII characters since that one ASCII character occupies 1 byte.

In order to demonstrate the powerful usage of string operation functions, the following examples will show you step by step how to create executable project files using the new

functions; starts from creating a macro, ends in executing simulation.

1. To read (or write) a string from a device:

Create a new macro:



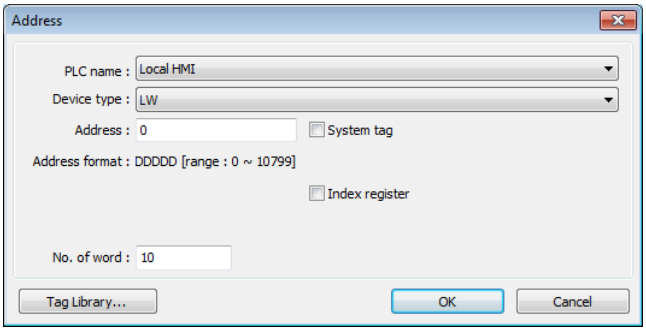
Edit the content:

```
1
2   macro_command main()
3
4   char str[20]
5
6   StringGet(str[0], "Local HMI", LW, 0, 20)
7   StringSet(str[0], "Local HMI", LW, 50, 20)
8
9   end macro_command
```

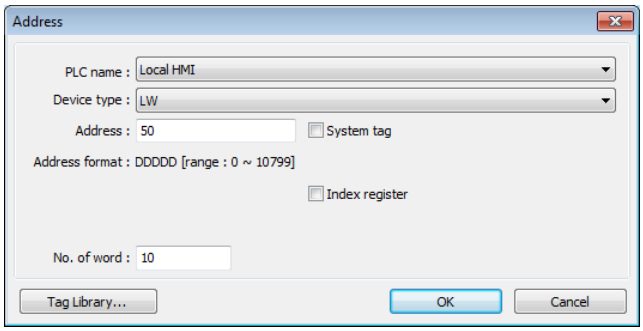
The first function “StringGet” is used to read a string from LW-0~LW-19, and store it into the str array. The second function “StringSet” is used to output the content of str array.

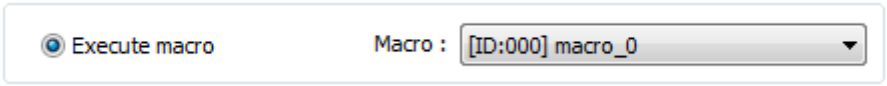
Add one ASCII Input object and one Function Key object in window 10 of the project. The settings of these objects are shown as below. Function Key object is used to execute macro_0.

ASCII Input object:



Function Key object:





Lastly, use [Compile] to compile the project and execute [Off-line simulation] or [On-line simulation]. Follow the steps below to operate the executing project:

Step 1. Input string.

Step 2. Press “GO” button.

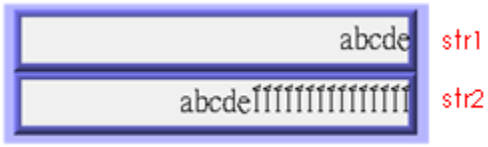
Step 3. Output string.

2. Initialization of a string.

Create a new macro and edit the content:

```
1
2   macro_command main()
3
4   char str1[20]="abcde"
5   char str2[20]={'a','b','c','d','e'}
6
7   StringSet(str1[0], "Local HMI", LW, 0, 20)
8   StringSet(str2[0], "Local HMI", LW, 50, 20)
9
10  end macro_command
```

The data enclosed in double quotation mark (" ") is viewed as a string. str1 is initialized as a string while str2 is initialized as a char array. The following snapshot of simulation shows the difference between str1 and str2 using two ASCII Input objects.



Macro compiler will add a terminating null character ('\0') at the end of a string. The function "StringSet" will send each character of str1 to registers until a null character is reached. The extra characters following the null character will be ignored even if the data count is set to a larger value than the length of string.

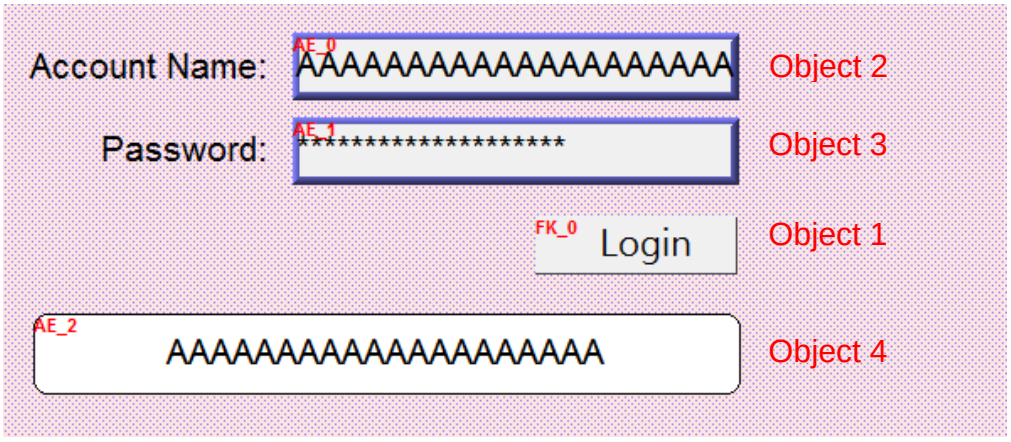
On the contrary, macro compiler will not add a terminating null character ('\0') at the end of a char array. The actual number of characters of str2 being sent to registers depends on the value of data count that is passed to the "StringSet" function.

3. A simple login page.

Create a new macro and edit the content, for example, Macro [ID:001] macro_1.

```
macro_command main()

char name[20]="admin"
char password[20]="123456"
char name_input[20], password_input[20]
char message_success[40]="Success! Access Accepted."
char message_fail[40]="Fail! Access Denied."
char message_clear[40]
bool name_match=false, password_match=false

StringGet(name_input[0], "Local HMI", LW, 0, 20)
StringGet(password_input[0], "Local HMI", LW, 50, 20)

name_match = StringCompare(name_input[0], name[0])
password_match = StringCompare(password_input[0], password[0])

FILL(message_clear[0], 0x20, 40) //FILL with white space
StringSet(message_clear[0], "Local HMI", LW, 100, 40)

if (name_match==true and password_match==true) then
    StringSet(message_success[0], "Local HMI", LW, 100, 40)
else
    StringSet(message_fail[0], "Local HMI", LW, 100, 40)
end if

end macro_command
```

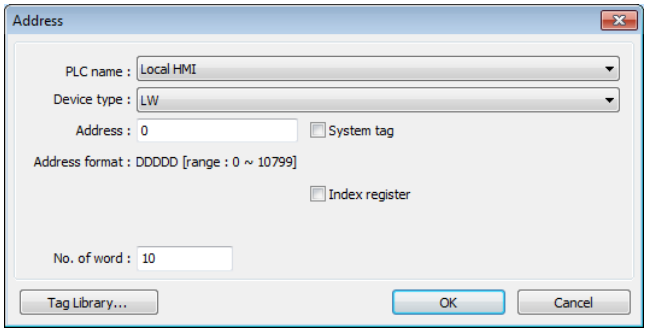
The first two “StringGet” functions will read the strings input by users and store them into arrays named name_input and password_input separately. Use the function “StringCompare” to check if the input account name and password are matched. If the account name is matched, name_match is set true; if the password is matched, password_match is set true. If both name_match and password_match are true, output the string “Success! Access Accepted.”. Otherwise, output the string “Fail! Access Denied.”. Add ASCII Input and Function Key objects in window 10 of the project. The settings of these objects are shown as below. Function Key object is used to execute macro_1.



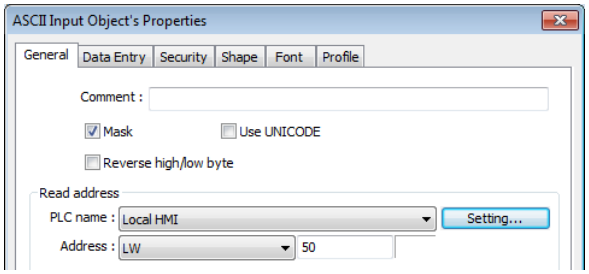
Object 1: Function Key

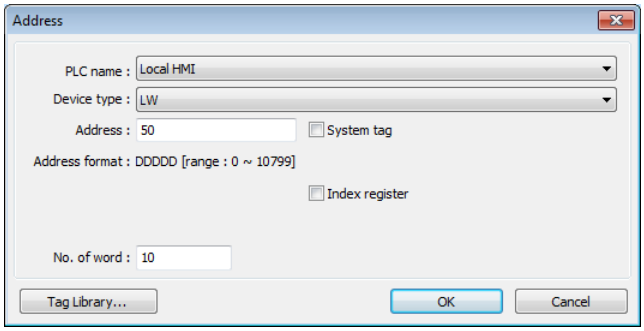
Select [Execute macro] and Macro: [ID:000] macro_1.

Object 2: ASCII Input

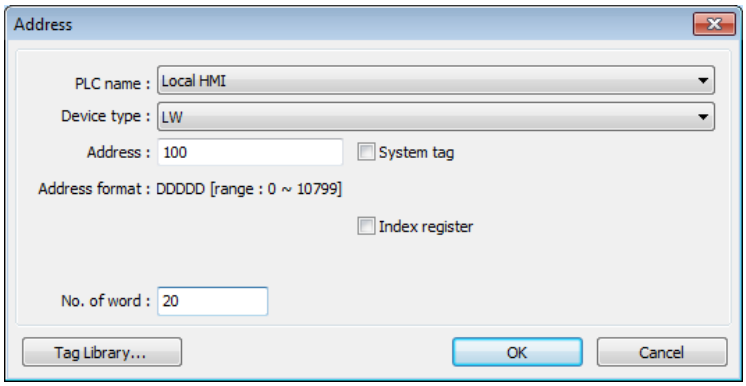


Object 3: ASCII Input



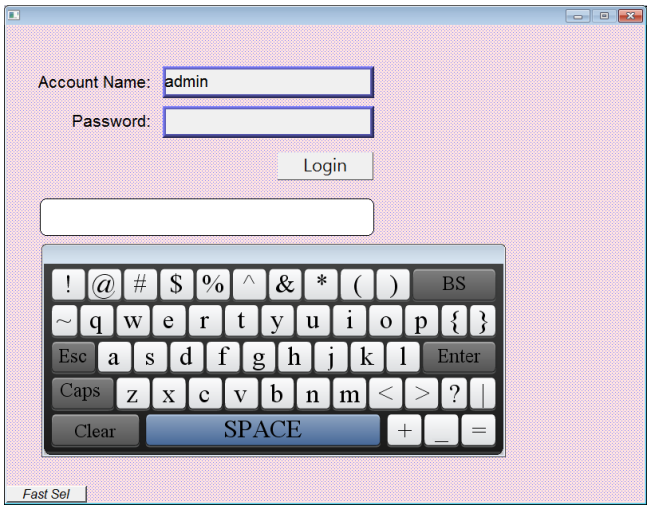


Object 4: ASCII Display

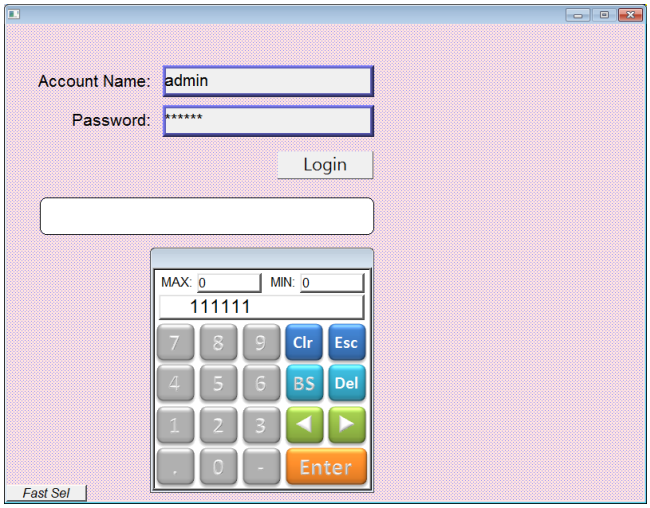


Lastly, use [Compile] to compile the project and execute [Off-line simulation] or [On-line simulation]. Follow the steps below to operate the executing project:

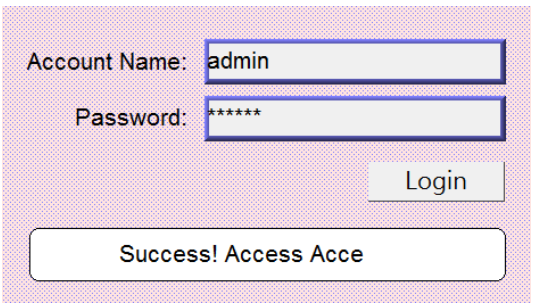
Step 1. Enter account name.

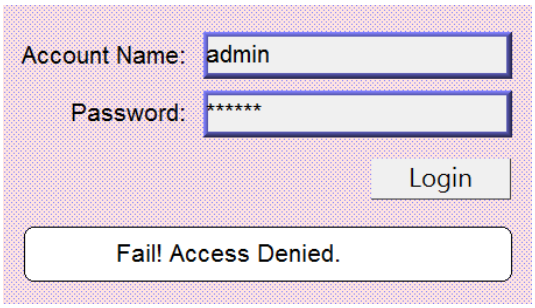


Step 2. Enter password and press [Login] button.



Step 3. Login succeeded or failed.

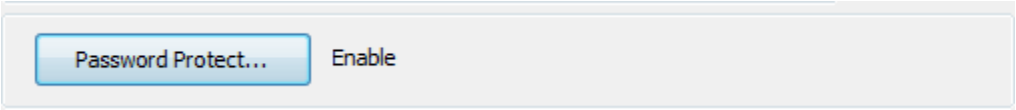




18.16. Macro Password Protection

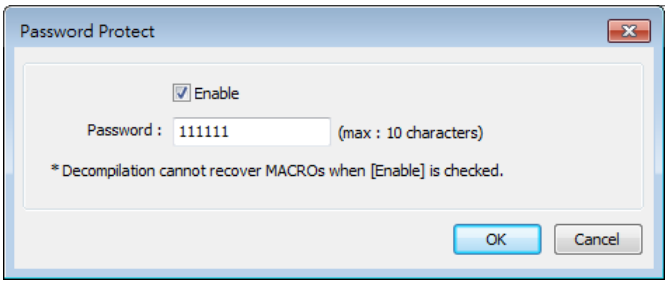
A password can be set to protect all the macros in the list, or an individual macro.

Protecting all macros:

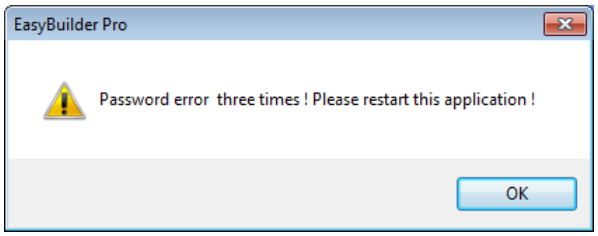


In Macro Manager window there's the [Password Protect…] button, click it and then click [Enable] to set a password less than or equals to 10 characters (support ASCII character only, e.g. "a$#*hFds").

After setting the password, users will have to enter correct password when opening Macro Manager.



EasyBuilder Pro should be rebooted for typing the password again after 3 incorrect attempts.

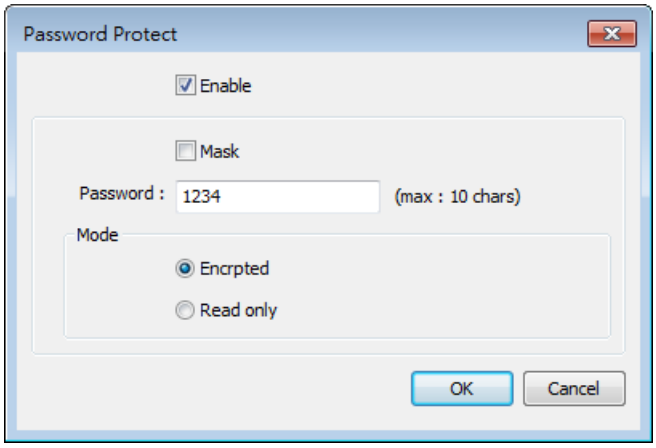


When macro is password protected, de-compilation of EXOB file will not be able to restore macro contents. The macro content will be shown as below:

```
macro_command main()

//  macro is password protected

end macro_command
```

Protecting individual macro:

In the Work Space for editing an indivisual macro, click the [Password Protect…] button and then click [Enable] to set a password less than or equals to 10 characters (support ASCII character only, e.g. "a$#*hFds"). [Encrypted] and [Read only] modes work as follows.



[Encrypted]

Encrypt the macro content. Entering macro editing window will require password.

EasyBuilder Pro should be rebooted for typing the password again after 3 incorrect attempts opening the same macro.

(The number of allowable incorrect attempts may vary between macros.)

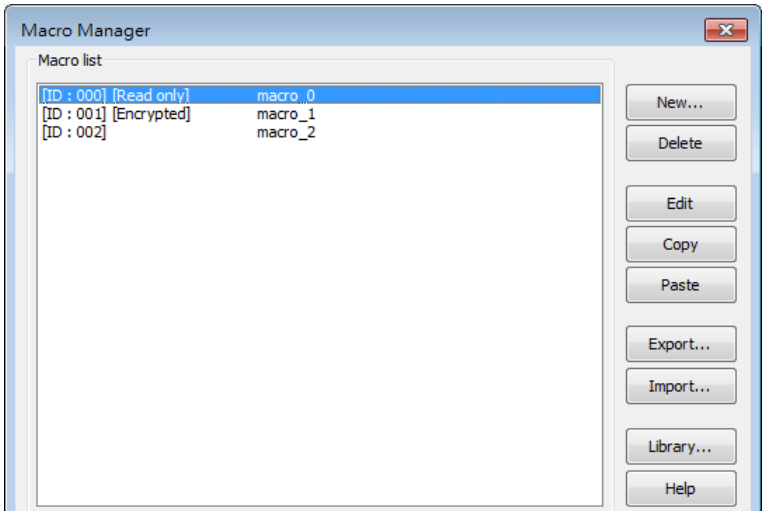
[Read-only]

The user can only view the content of the macro and will not be able to edit it.

With this mode selected, macro editing window can be opened directly from Macro Manager; however, a password is required after clicking [Password Protect…] button.

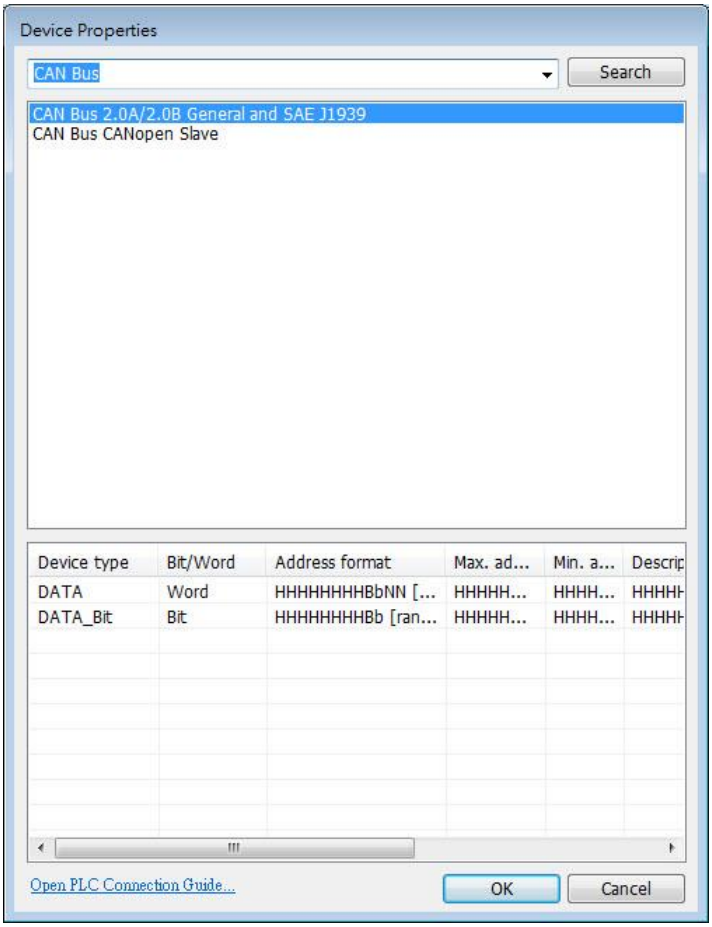
EasyBuilder Pro should be rebooted for typing the password again after 3 incorrect attempts.

In the macro list, the selected mode for each macro is shown.



18.17. Reading / Writing CANbus Address Using Variable

In “CAN Bus 2.0A/2.0B General and SAE J1939” driver, two device types can be found: DATA and DATA_Bit, and the formats of these device types are shown in the following window.



Device Type & Address Format	Description
DATA HHHHHHHHBbNN	H: ID B: Byte position(1~8) b: Bit position (1~8) NN: Bit number(1~64)
DATA_Bit HHHHHHHHBb	H: ID B: Byte position(1~8) b: Bit position(1~8)

The ID is represented in hexadecimal while the position and number are represented in decimal, please see the usage below.

Examples:

Variable is **not** used:

short f

GetData(f, “CAN Device”, DATA, 4e55108, 1)

GetData(f, “CAN Device”, DATA, 4e65108, 1

Variable is used:
short f
unsigned int address = **0x**4e55108
GetData(f, “CAN Device”, DATA, address, 1)
address = address + 0x10000// == 0x4e65108
GetData(f, “CAN Device”, DATA, address, 1)

Please note that:

1. Declare variable as “Unsigned int” and use hexadecimal to represent address.
   Since the size of Unsigned int is 4 bytes and Bb, NN take 1 byte respectively, when using a variable for address parameter to read/write DATA_Bit device type, the format will change to HHHHHHBb (Max. ID: 0xffffff), and when using a variable for address parameter to read/write DATA device type, the format will change to HHHHBbNN (Max. ID: 0xffff).