Macro Reference Manual for EZwarePlus

Your Industrial Control Solutions Source

www.maplesystems.com

For use with the following:

- EZwarePlus Programming Software

Maple Systems, Inc. | 808 134th St. SW, Suite 120, Everett, WA 98204 | 425.745.3229

COPYRIGHT NOTICE

This manual is a publication of Maple Systems, Inc., and is provided for use by its customers only. The contents of the manual are copyrighted by Maple Systems, Inc.; reproduction in whole or in part, for use other than in support of Maple Systems equipment, is prohibited without the specific written permission of Maple Systems.

WARRANTY

Warranty Statements are included with each unit at the time of purchase and are available at www.maplesystems.com.

TECHNICAL SUPPORT

If you need assistance, please contact Maple Systems:

- Phone: 425-745-3229
- Email: support@maplesystems.com
- Web: http://www.maplesystems.com

Table of Contents

COPYRIGHT NOTICE 2
WARRANTY 2
TECHNICAL SUPPORT 2
1. Overview 5
2. The Macro Editor 6
Line numbers 7
Bookmarks 7
Outlines 9
Undo/Redo 9
Cut/Copy/Paste 10
Select All 10
Find/Replace 11
Security (Use execution condition) 11
Periodical execution 12
Execute one time when HMI starts 12
3. Configuration 13
4. Syntax 16
Constants and Variables 16
Declaring Arrays 17
Variable and Array Initialization 18
Operators 18
Priority of All Operators 20
Reserved Keywords 20
5. Statements 21
Declaration Statement 21
Assignment Statement 21
Logical Statement 21
Selective Statement 22
Iterative Statement 24
For-next Statements 24
While-wend Statements 25
Other Control Commands 25
6. Function Blocks 26
7. Built-in Function Blocks 28
Mathematical Functions 28
Data Transformation 33
Data Manipulation 38
Bit Transformation 40

String Operation Functions 56
Recipe Query Functions 78
Miscellaneous 81
8. How to Create and Execute a Macro 89
How to Create a Macro 89
How to Execute a Macro 92
PLC Control Object 92
Set Bit or Toggle Switch Object 94
Function Key Object 95
Macro Work Space 96
9. User-Defined Macro Functions 97
Using the Macro Function Library 97
Creating a New Function 100
Using a Macro Function in a Numeric Object 101
Using a Macro Function in the Address Tag Library 102
Some Notes About Using Macros 103
Deleting a Macro Function 103
Modifying a Macro Function 103
Importing a Macro Function 104
Exporting a Macro Function 105
10. Using the Free Protocol to Control a Device 106
11. Compiler Error Message 111
Error message format 111
Error Description 111
12. Sample Macro Code 117
for statement 117
while, if, and break statements 117
Global variables and function call 118
If statement 118
While and wend statements 119
Break and continue statements 119
Array 120
13. Macro TRACE Function 121
Trace Syntax 122
Connecting EasyDiagnoser to an HMI 123
14. Macro Password Protection 125

1. Overview

Macros provide additional functionality for performing tasks that are not supported in a standard object or feature in the HMI. Macros are automated sequences of commands that are executed at run-time. Macros allow you to perform tasks such as moving data, complex scaling operations, string handling, conditional logic, and other user interactions with your projects.

The macro language combines elements of C, C++, and a little of its own style. Previous experience in programming with C or C++ would be helpful when developing macros. It is not the scope of this document to explain concepts of high-level programming or basic concepts such as variable data types, arrays, conditional logic, and variable passing. This document will discuss syntax, usage, and command structures that are required in building macros.

Macros are triggered by setting a bit or by pressing a Function Key configured to “Execute macro.” The bit is specified in the PLC Control object list (Objects menu > PLC Control > New > Type of Control > Execute macro program). The bit may be a Local Bit (LB) or a PLC bit. Turn the bit ON to initiate the macro. The macro will run its sequence of events and will exit when finished. The bit that called the macro into execution should be cleared by the macro before it exits or the macro will start again. This could form an endless loop, which can tax the CPU and cause a slow response time in the HMI.

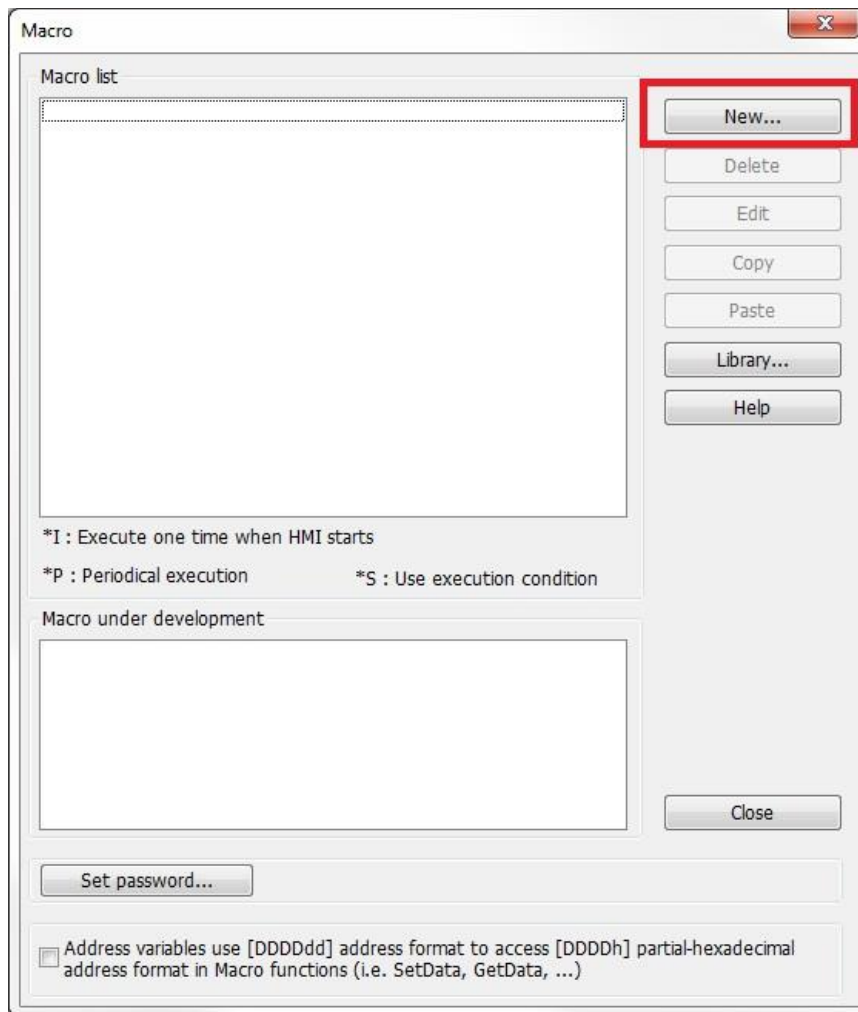
2. The Macro Editor

Use the Macro Editor (Work Space) to create and edit your macros.

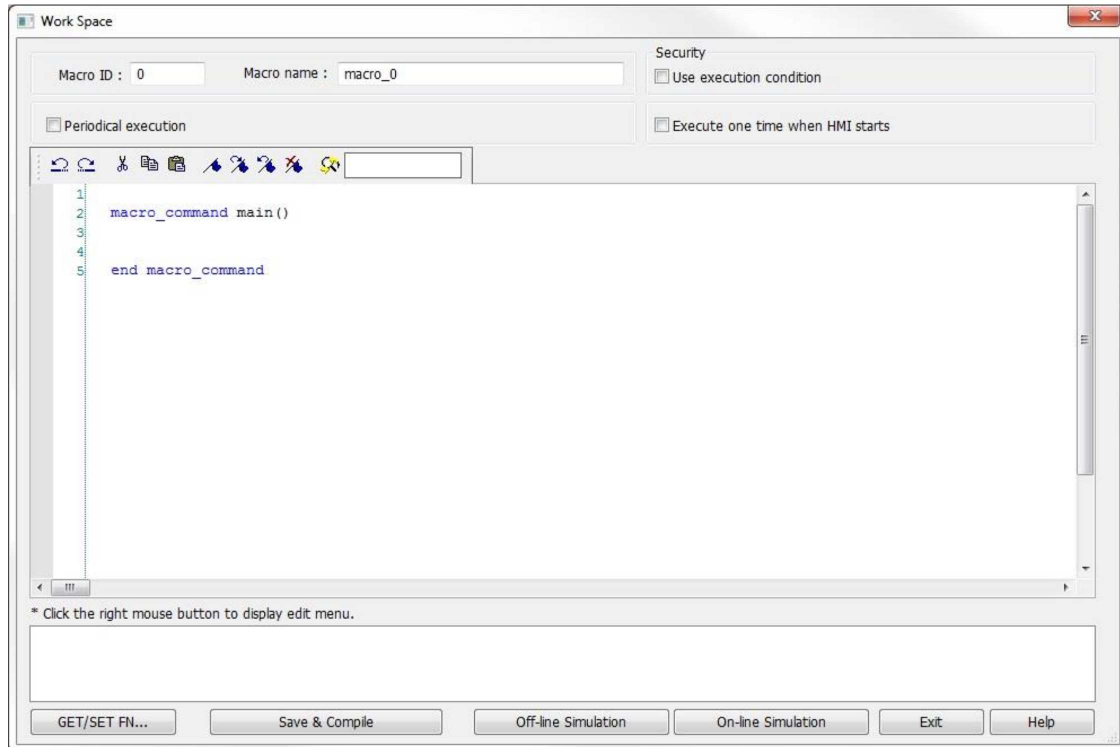
To open the Macro Editor:

1. Click the “Tools” menu in EZwarePlus and select “Macro.” This opens the Macro List window.

2. Click the “New” button to open the Macro Editor (Work Space).



3. The Macro ID number is automatically assigned when the macro is created but can be changed to reorder the macros in the list. The programmer can give a name to the macro or simply use the default name (for example, macro_0).



The macro work space provides the following editing and execution functions:

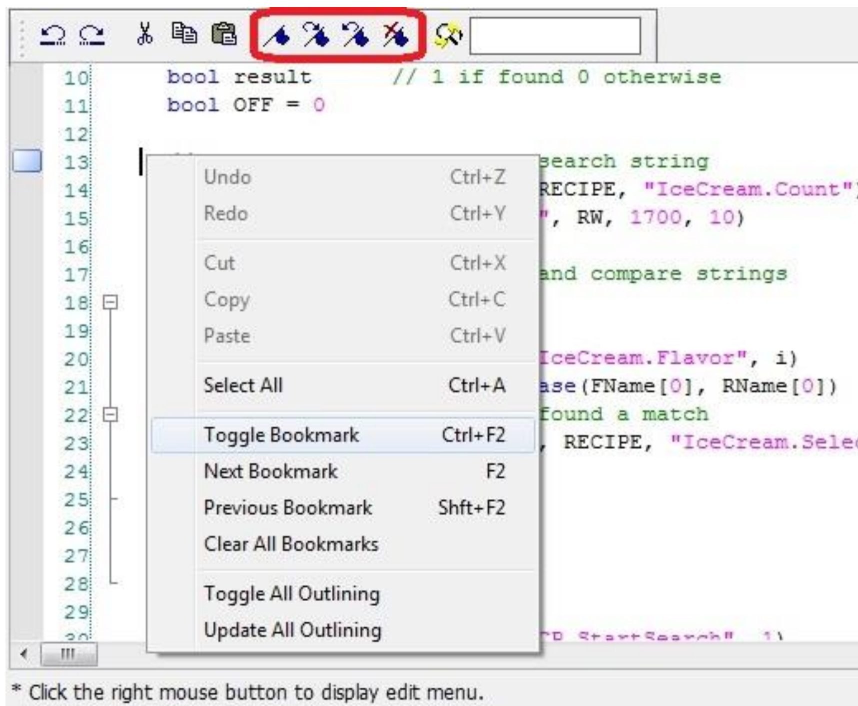
- Display line number
- Bookmarks
- Outlines
- Undo/Redo
- Cut/Copy/Paste
- Select All
- Find/Replace
- Security (Use execution condition)
- Periodical execution
- Execute one time when HMI starts

Line numbers

Line numbers are automatically created as your macro grows in size. Numbered lines are useful for keeping track of sections in your macro.

Bookmarks

Use bookmarks to quickly jump from one area of your macro to another. Move the cursor to the beginning of a section or block of the macro, right-click and select “Toggle Bookmark” (or click the “Toggle Bookmark” icon in the Edit toolbar).



This will place a blue square to the left of the line number indicating a bookmark. The “Toggle Bookmark” function is used to add a new bookmark or remove an existing bookmark.

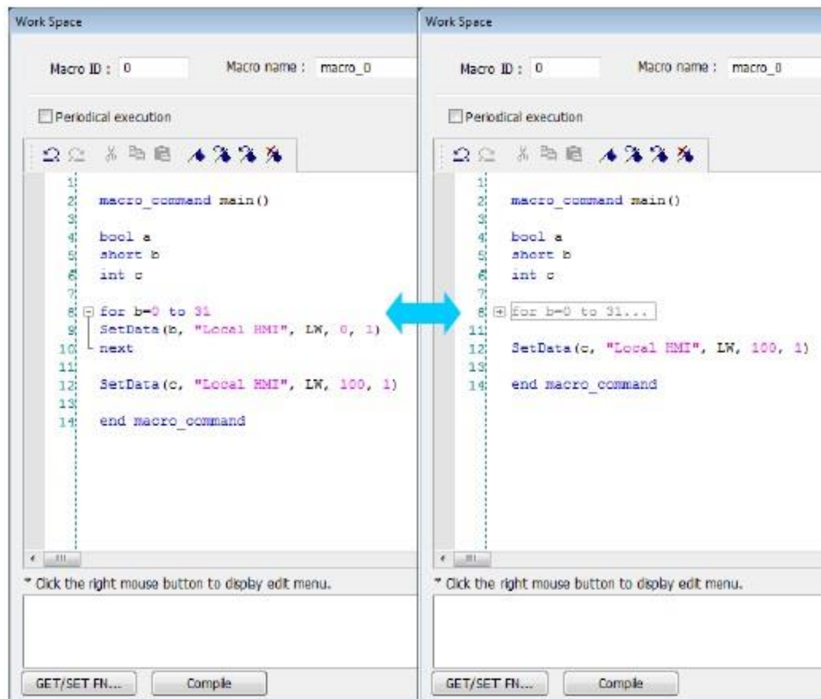
```
1   //RCP_FindRecipe - Searches through the recipe data base to find the string
2   //                 entered in the search box and, if found, select the desired recipe
3   macro_command main()
4
5      //declarations
6      short i = 0
7      short count      // number of recipes
8      char RName[20]   // Buffer for recipe name
9      char FName[20]   // Buffer for search string
10     bool result      // 1 if found 0 otherwise
11     bool OFF = 0
12
13     //Get number of recipes and search string
14     GetData(count, "Local HMI", RECIPE, "IceCream.Count")
15     GetData(FName[0], "Local HMI", RW, 1700, 10)
16
```

Right-click and select “Next Bookmark” (or select “Next Bookmark” in the Edit toolbar) to jump to the next bookmark in the macro editor. Select “Previous Bookmark” to go back to a previous bookmark in the macro editor.

Select “Clear All Bookmarks” to delete all the bookmarks in the macro.

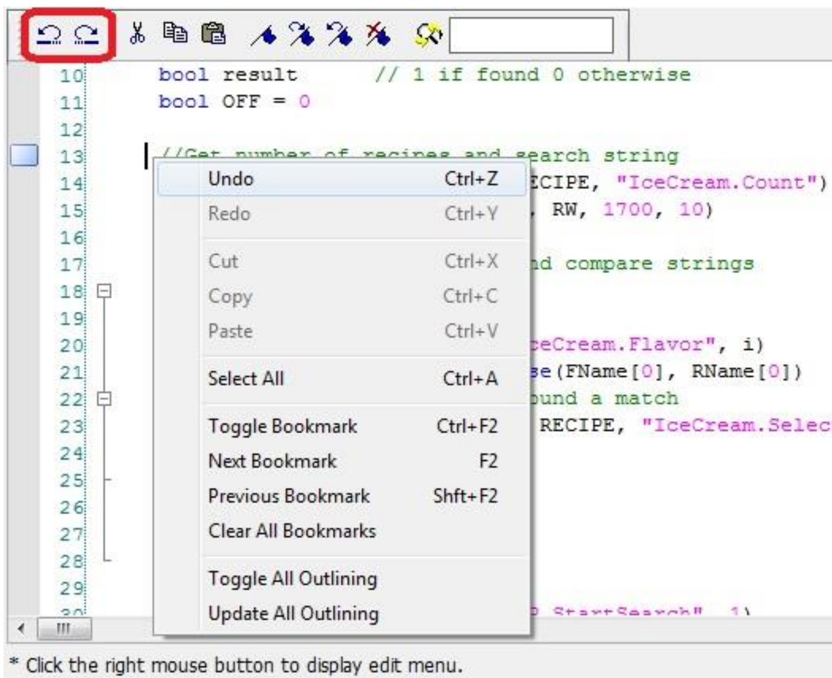
Outlines

Outlining is used to hide sections of the macro code that belong in the same block. Outlines are automatically created for functions such as if/then statements, for, and while loops.



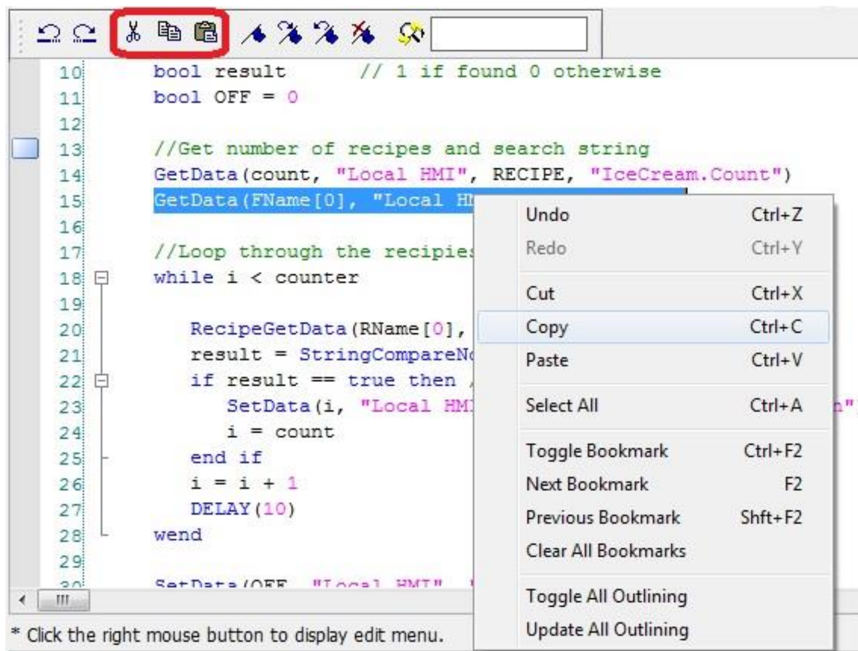
Undo/Redo

Any modification in the macro work space can be undone or redone using the Undo and Redo functions. Right-click in the macro work space to access the edit menu and select Undo or Redo (or click the Undo or Redo icons in the Edit toolbar).



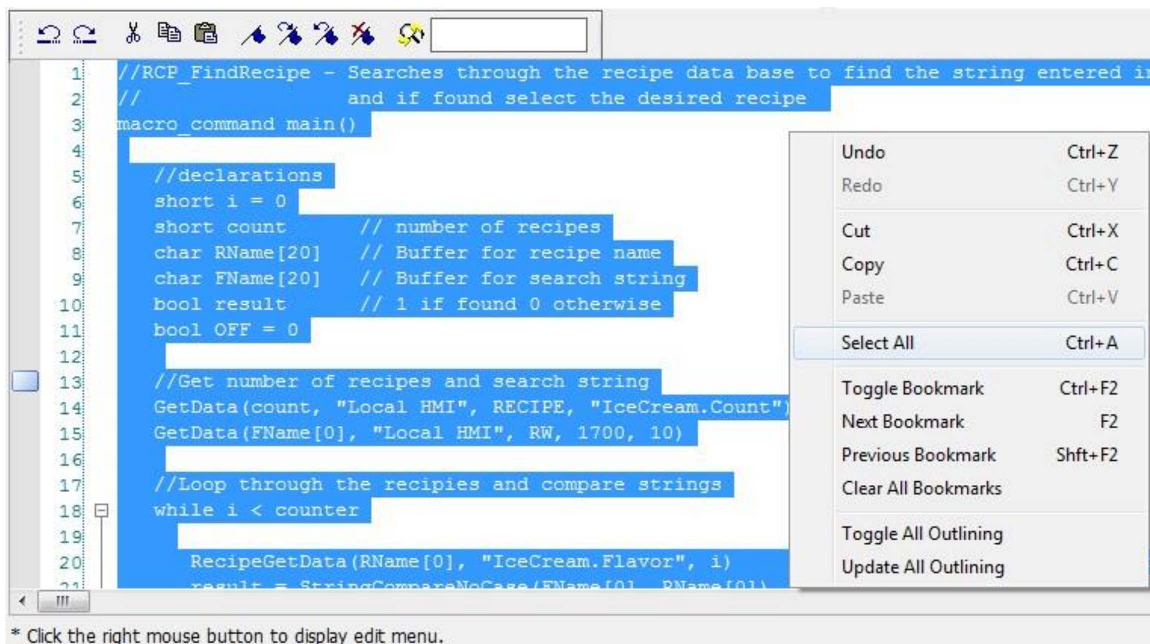
Cut/Copy/Paste

Text can be selected in the macro work space and copied and pasted to another location within the macro. Right-click in the macro work space to access the edit menu and select Copy or Cut to copy the selected text to the clipboard, and Paste to paste the text into a new location.



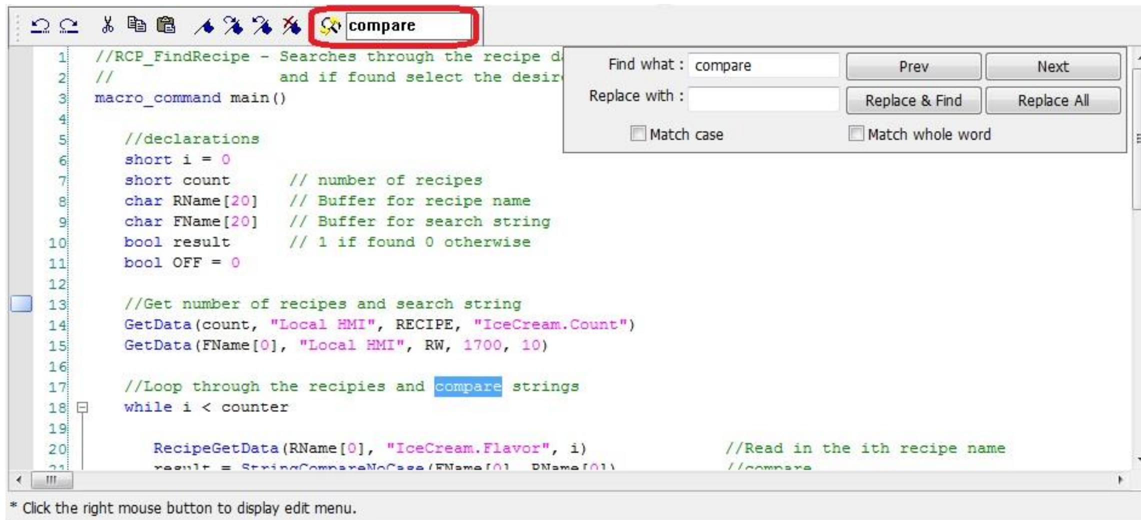
Select All

Right-click in the macro work space to access the edit menu and select Select All (or Ctrl+A) to select all the text in the macro work space.



Find/Replace

You can search a macro for keywords or text strings by entering the text in the Find/Replace field and clicking the Find/Replace button. In the Find/Replace dialog window, click the “Next” and “Prev” buttons to search forward and backwards through the macro. If you wish to replace the selected text, enter the new text in the “Replace with” field.



Click the Find/Replace button again to close the Find/Replace dialog window.

Security (Use execution condition)

The security feature allows you to use a bit to enable/disable the macro. With “Use execution condition” checked, click the “Settings” button to assign the control bit. Select “Disable when Bit is ON” or “Disable when Bit is OFF” to determine which state will disable the macro.

Periodical execution

Select “Periodical execution” to execute the macro periodically at a predetermined time interval.

Periodical execution Time interval (0 ~ 864000) : 10 x 100ms

Execute one time when HMI starts

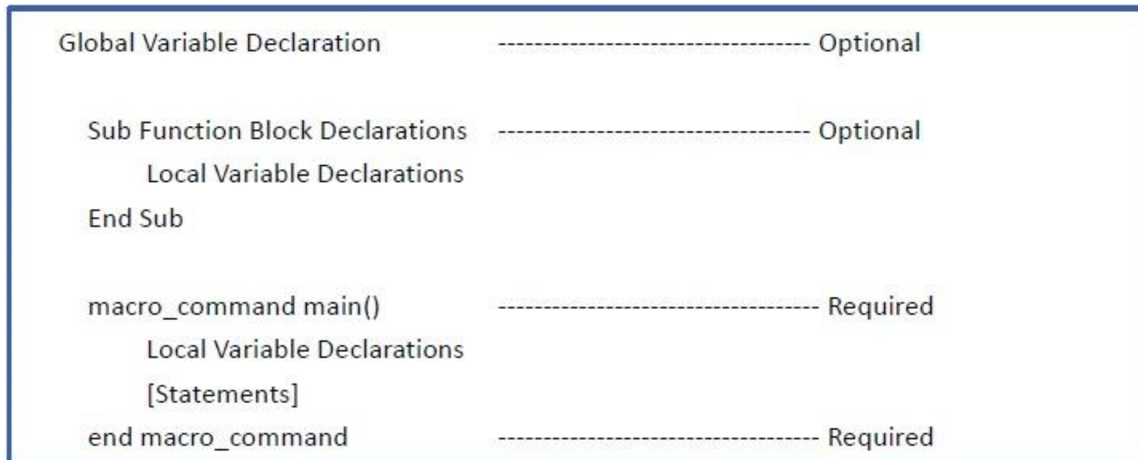
With this option selected, the macro will be executed once when the HMI starts up. The macro can be triggered again using the normal methods of setting a bit or using a Function Key configured to “Execute a macro.”

Execute one time when HMI starts

3. Configuration

A macro contains statements, which are comprised of constants, variables, and functions. The statements are put in a specific order to create the desired output.

A macro has the following structure:



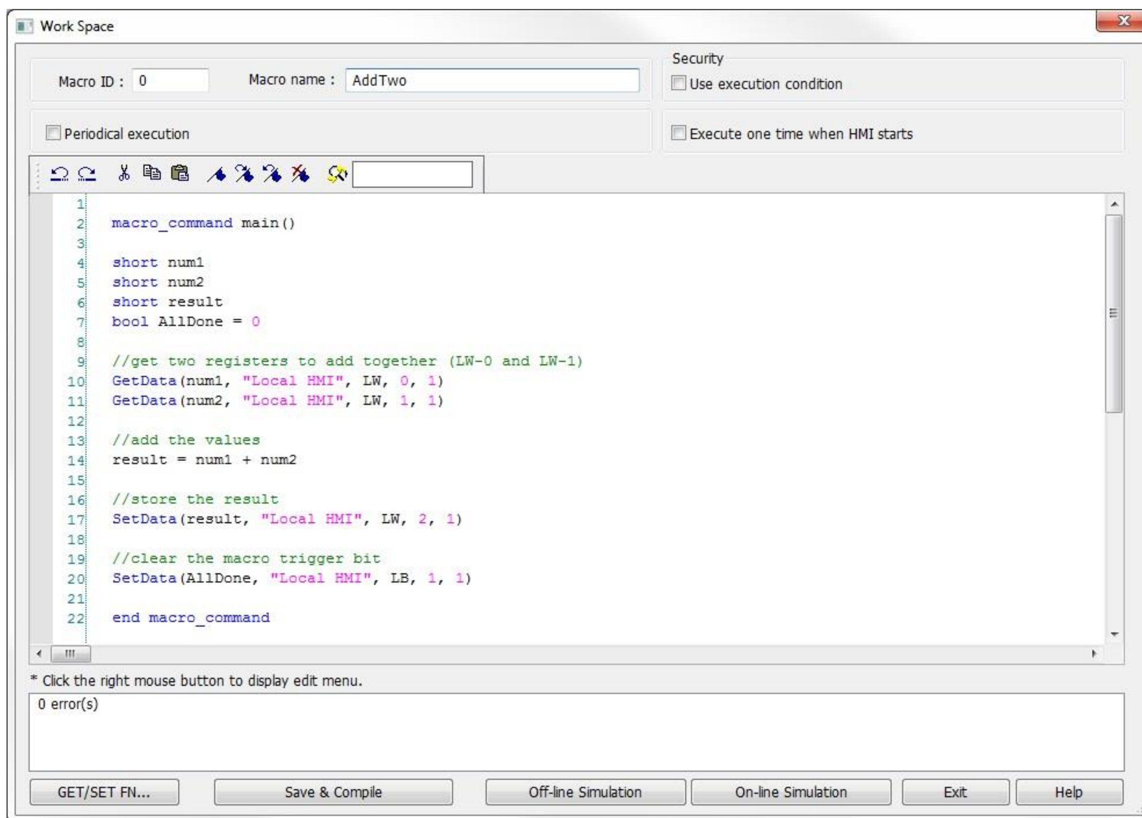
A macro can have only one main function, which is the start point for the execution of the macro.

```
macro_command main()


end macro_command
```

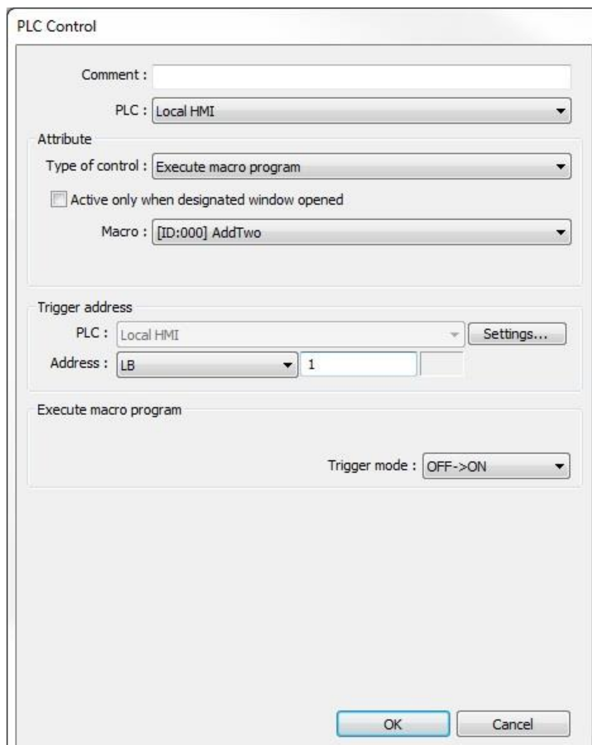
Local variables are used within the main macro function or in a defined function block. Its value remains valid only within the specified block.

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 name, only the local variables are valid.

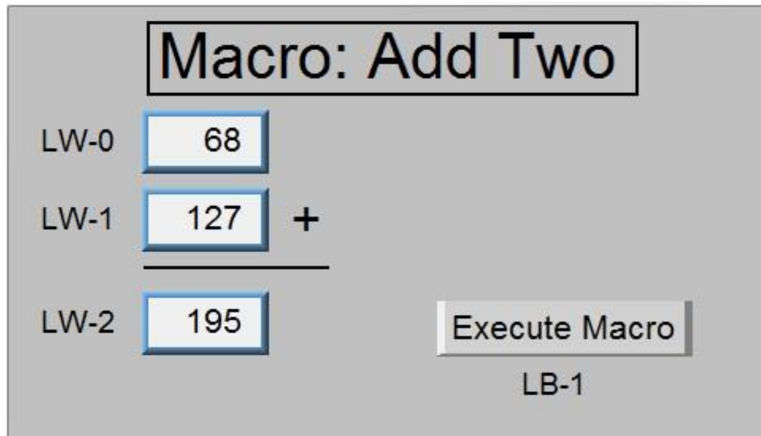
The following example shows a simple macro that reads two local word registers, adds them together, and stores the result into another register.



The macro is named “AddTwo” and the bit that executes the macro is LB-1. A PLC Control object was created with the “Type of control” set to “Execute macro program.” LB-1 was selected as the Trigger address and the macro “AddTwo” was selected. Here are the settings for the PLC Control object:



Project example of “AddTwo” macro:



The operator enters a number in the Numeric object LW-0 and LW-1, then presses the “Execute Macro” button. This turns ON LB-1and triggers the PLC Control object to execute the macro “AddTwo.” The macro adds the values in LW-0 and LW-1, writes the result to LW-2, and clears LB-1 so that the macro can be triggered again.

4. Syntax

Constants and Variables

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 by double quotation marks.	‘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
```

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

- 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 eight different variable types; five for signed data types and three for unsigned data types:

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

unsigned int	32 bits (double word)	0 to +4,294,967,295
float (floating point – IEEE 754)	32 bits (double word)	N/A

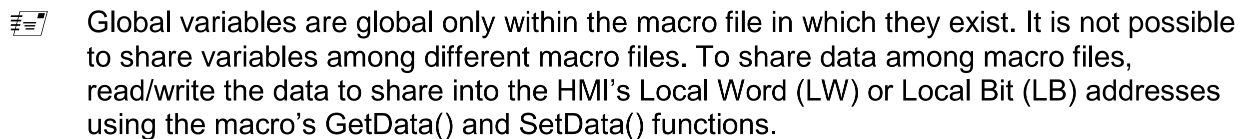
Variables must be declared before being used. To declare a variable, specify the variable type before the name of the variable.

Example:

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

The declarations are listed as the first part of the macro. Any declarations contained within the macro function are considered “Local” and any variable outside the macro function is considered “Global.” Local variables are only seen within the function they are declared in. Global variables retain their values and can be globally used by all functions.

Note: If a local variable has the same name as a global variable, the compiler will use local variable in the function instead of the global variable.



Global variables are global only within the macro file in which they exist. It is not possible to share variables among different macro files. To share data among macro files, read/write the data to share into the HMI’s Local Word (LW) or Local Bit (LB) addresses using the macro’s GetData() and SetData() functions.

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 can range from 1 to 4096 (a macro can support up to 4096 variables per macro).

Example:

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

The array index ranges from 0 to (array size - 1).

Example:

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

In this case, the array index ranges from 0 to 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
  Example:
  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 “{ }”

Examples:
float data[4] = {11, 22, 33, 44} // now data[0] is 11, data[1] is 22, data[2] is 33, and data[3] is 44.
char LetterArray[6] = “MYWORD”

The initial values are assigned in order from left to right starting at array index = 0.

Operators

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

Arithmetic Operators	Description	Example
=	Assignment operator	pressure = 10
+	Addition	A = B + C
-	Subtraction	A = B - C
*	Multiplication	A = B * C
/	Division	A = B / C
%	Modulo division (returns remainder)	A = B % 5

By default, integer numbers (1, 2, 3, etc.) are considered having an integer data type. Therefore, when division is carried out involving two integer numbers where the result should have a decimal point, the decimal part will be removed. To avoid this, add a decimal to the integer (1.0, 2.0, 3.0, etc.). This will turn it into a floating point number for the calculation and the result will also be floating point.

Example:
A = 3 / 2 results in 1 (3 and 2 are both integers, therefore the result is an integer)
B = 3 / 2.0 results in 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 results in 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 B = 256 then C = 5
not	Logical NOT	if not A then B = 5

Shift operators 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
<<	Shift the bits in a bitset to the left a specified number of positions	A = B << 8
>>	Shift the bits in a bitset to the right a specified number of positions	A = B >> 8

<table>
<tr><th>Bitwise Operators</th><th>Description</th><th>Example</th></tr>
<tr><td>&</td><td>Bitwise AND</td><td>A = B & 0xf</td></tr>
<tr><td>|</td><td>Bitwise OR</td><td>A = B | C</td></tr>
<tr><td>^</td><td>Bitwise XOR</td><td>A = B ^ C</td></tr>
<tr><td>~</td><td>Ones’ complement</td><td>A = ~ B</td></tr>
</table>

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 use. 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, INPORT, OUTPORT, POW, 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

5. Statements

There are different styles of statements including declaration statements, assignment statements, logical or conditional statements, selective statements, and iterative statements (for-next and while-wend).

An *expression* combines one or more of the following: constants, variables, arrays, functions and operators. Expressions can be logical or mathematical equations.

When the value of a Boolean expression is zero, it means FALSE.
When the value of a Boolean expression is not zero, it means TRUE.

Declaration Statement

A declaration (definition) statement declares a variable with a name and type.

Format:
type name

Examples:
int A // define a variable A as an integer

short MyVars[10] // format: *type* name[constant]; define an array variable MyVars and the [constant] is the number of elements in the array

Assignment Statement

An assignment statement transfers (assigns) an expression to a variable.

Format:
variable = expression

Example:
MyVar = 10

MyVar = x + y //assigns the result of the expression to a variable

Logical Statement

Logical statements perform actions depending on the condition of a Boolean expression. It processes a logical expression and branches depending on the result.

Single-Line Format:
if <condition> then
[statements]
else
[statements]
end if

Examples:
if a == 2 then
b = 1
else
b = 2
end if

```
if (x == 0 or x == 1) and y == 0 then
z = 10
else
z = 5
end if
```

Note that there is a difference between the conditional equality notes by two equal signs '==' (read as 'is equal to') and the assignment equality noted by only one equal sign ('=').

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 the block format but necessary in single-line format without else. The statements 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.

Selective Statement

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.

Format without default case:

```
select case [variable]
case [value]
[statements]
break
end select
```

Example:

```
select case A
case 1
b = 1
break
end Select
```

Format with default case:

```
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
```

Format with multiple cases in the same block:

```
select case [variable]
case [value 1]
[statements]
case [value 2]
[statements]
break
end select
```

Example:

```
select case A
case 1
b = 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 statements until the end command is reached.
end select	Indicates the end of the select-case statements.

Iterative Statement

Iterative statements control loops and repetitive tasks depending on condition. There are two types of iterative statements, for-next statements and wend-while statements.

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.

Format:
for [counter] = <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.
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>] is 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.

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.

Format:
while <condition>
[statements]
wend

Example:
while a < 10
a = a + 10
wend

Syntax description

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

Other Control Commands

break	Used in 'for-next' and 'while-wend' statements. It skips immediately to the end of the iterative statement.
continue	Used in 'for-next' and 'while-wend' statements. 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.

6. Function Blocks

Function blocks are useful for reducing repetitive codes. It must be defined before use and is therefore placed before the macro_command main() function. It 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 the function definition, which means that a function block does not have to return a value. The parameters can also be ignored in the function definition while the function has no need to take any parameters from the caller.

Format with return type:

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

Examples:

```
sub int Add(int x, int y)  //defines a sub function called Add
int result                 // declares the variable 'result' to be returned to the main function
result = x + y
return result
end sub

macro_command main()
int a = 10, b = 20, sum
sum = Add(a,b)             //calls the sub function Add
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
```

Format without return type:

```
sub <name> [(parameters)]
local variable declarations
[statements]
end sub
```

Examples:

```
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 the 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, in 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 an integer according to the definition of the function. Notice that the calling statement can pass hard coded values or variables to the function. After this function is executed, an integer value is returned 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 use. Global variables are also available for use in the 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 the 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 the function, the return command is required.
end sub	Must be used to end a function block.

7. Built-in Function Blocks

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

Mathematical Functions

Name	SQRT
Syntax	SQRT(*source, result*)
Description	Calculate the square root of *source* and store the result in *result*. *Source* can be a constant or a variable. *Result* must be a variable. *Source* must be a non-negative value.

Example	macro_command main()
	
	float source, result
	
	SQRT(16, result) // result is 4.0
	
	source = 9.0
	SQRT(source, result) // result is 3.0
	
	end macro_command

Name	CUBERT
Syntax	CUBERT(*source, result*)
Description	Calculate the cube root of *source* and store the result in *result*. *Source* can be a constant or a variable. *Result* must be a variable. *Source* must be a non-negative value.

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

Name	POW
Syntax	POW(*source1, source2, result*)
Description	Calculate *source1* to the power of *source2* and store the result in *result*. *Source1* and *source2* can be a constant or a variable. *Result* must be a variable. *Source1* and *source2* must be a non-negative value.
Example	macro_command main() float y, result y=0.5 POW(25, y, result) // result is 5.0 end macro_command

Name	SIN
Syntax	SIN(*source, result*)
Description	Calculate the sine of *source* (in degrees) and store the result in *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 1.0
	
	source = 30
	SIN(source, result) // result is 0.5
	
	end macro_command

Name	COS
Syntax	COS(*source, result*)
Description	Calculate the cosine of *source* and store the result in *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	Calculate the tangent of *source* (in degrees) and store the result in *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 1.0
	
	source = 60
	TAN(source, result) // result is 1.732
	
	end macro_command

Name	COT
Syntax	COT(*source, result*)
Description	Calculate the cotangent of *source* (in degrees) and store the result in *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 1.0
	
	source = 60
	COT(source, result) // result is 0.5774
	
	end macro_command

Name	SEC
Syntax	SEC(*source, result*)
Description	Calculate the secant of *source* (in degrees) and store the result in *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) // result is 2.0
	
	end macro_command

Name	CSC
Syntax	CSC(*source, result*)
Description	Calculate the cosecant of *source* (in degrees) and store the result in *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 2.0
	
	end macro_command

Name	ASIN
Syntax	ASIN(*source, result*)
Description	Calculate the arc sine of *source* (in degrees) and store the result in *result*. *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	Calculate the arc cosine of *source* (in degrees) and store the result in *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
	TAN(source, result) //result is 60
	
	end macro_command

Name	ATAN
Syntax	ATAN(*source, result*)
Description	Calculate the arc tangent of *source* (in degrees) and store the result in *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
	TAN(source, result) // result is 60
	
	end macro_command

Name	LOG
Syntax	LOG(*source, result*)
Description	Calculate the natural logarithm of *source* and store the result in *result*. *Source* can be a constant or a variable. *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	Calculate the base-10 logarithm of *source* and store the result in *result*. *Source* can be a constant or a variable. *Result* must be a variable.
Example	macro_command main() float source = 100, result LOG10(source, result) //result is 2.0 end macro_command

Name	RAND
Syntax	RAND(*result*)
Description	Calculate a random integer and store the result in *result*. *Result* must be a variable.
Example	macro_command main() short result RAND(result) // result will vary each time the macro is executed end macro_command

Data Transformation

Name	BIN2BCD
Syntax	BIN2BCD(*source, result*)
Description	Transform 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	Transform 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	Transform a decimal value (*source*) into an ASCII string and save it to an array (*result*). *Len* represents the length of the string and the unit of length depends on the result's type. For example, if the result type is "char" (where the size is one byte), the length of the string is (byte * *len*). If the result type is "short" (where the size is one 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', result1[1] is '6', result1[2] is '7', result1[3] is '8'
	// the length of the string *result2* is 8 bytes (2*4)
	
	source = -123
	DEC2ASCII(source, result3[0], 6)
	// result3[0] is '-', result3[1] is '0', result3[2] is '0', result3[3] is '1', result3[4] is '2',
	// result3[5] is '3'
	// the length of the string *result3* is 6 bytes (1*6)
	
	end macro_command

Name	HEX2ASCII
Syntax	HEX2ASCII(*source, result[start], len*)
Description	Transform a hexadecimal value (*source*) into an ASCII string and save it to an array (*result*). *Len* represents the length of the string and the unit of length depends on the result's type. For example, if the result type is "char" (where the size is one byte), the length of the string is (byte * *len*). If the result type is "short" (where the size is one 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 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'
	// the length of the string *result* is 4 bytes (1*4)
	
	end macro_command

Name	FLOAT2ASCII
Syntax	FLOAT2ASCII(*source, result[start], len*)
Description	Transform a floating point value (*source*) into an ASCII string and save it to an array (*result*). *Len* represents the length of the string and the unit of length depends on the result's type. For example, if the result type is "char" (where the size is one byte), the length of the string is (byte * *len*). If the result type is "short" (where the size is one 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() 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’
	// the length of the string *result* is 4 bytes (1*4)
	
	end macro_command

Name	ASCII2DEC
Syntax	ASCII2DEC(*source[start], result, len*)
Description	Transform a string (*source*) into a decimal value and save it 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	ASCII2HEX
Syntax	ASCII2HEX(*source[start], result, len*)
Description	Transform a string (*source*) into a hexadecimal value and save it 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	ASCII2FLOAT
Syntax	ASCII2FLOAT(*source[start], result, len*)
Description	Transform a string (*source*) into a floating point value and save it 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] = ‘.’ source[3] = ‘8’ ASCII2FLOAT(source[0], result, 4) // result is 56.8 end macro_command

Data Manipulation

Name	FILL
Syntax	FILL(*source[start], preset, count*)
Description	Sets the first count elements of an array (*source*) to a specified value (*preset*). *Source* and *start* must be a variable. *Preset* can be a constant or a variable.

Example	macro_command main()
	
	char result[4]
	short 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* and save it 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* and save it 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* and save it 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* and save it 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	LOWORD
Syntax	LOWORD(*source, result*)
Description	Retrieves the low word of a 32-bit *source* and save it 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 low word of a 32-bit *source* and save it 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

Bit Transformation

Name	GETBIT
Syntax	GETBIT(*source, result, bit_pos*)
Description	Get the state of the designated bit position (*bit_pos*) of a word (*source*) and save it 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

Name	SETBITON
Syntax	SETBITON(*source, result, bit_pos*)
Description	Change the state of the designated bit position (*bit_pos*) of a word (*source*) to 1 and save it 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	Change the state of the designated bit position (*bit_pos*) of a word (*source*) to 0 and save it 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	INVBIT
Syntax	INVBIT(*source, result, bit_pos*)
Description	Inverts the state of the designated bit position (*bit_pos*) of a word (*source*) and save it 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 is 6
	
	source = 6
	bit_pos = 1
	INVBIT(source, result, bit_pos) // result is 4
	
	end macro_command

Communication

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 milliseconds. *Time* can be a constant or a variable.

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

Name	ADDSUM
Syntax	ADDSUM(*source[start], result, data_count*)
Description	Add up the elements of an array (*source*) from *source[start]* to *source[start+(data_count-1)]* to generate a checksum. Puts the checksum into *result*. *Result* must be a variable. *Data_count* is the number of 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

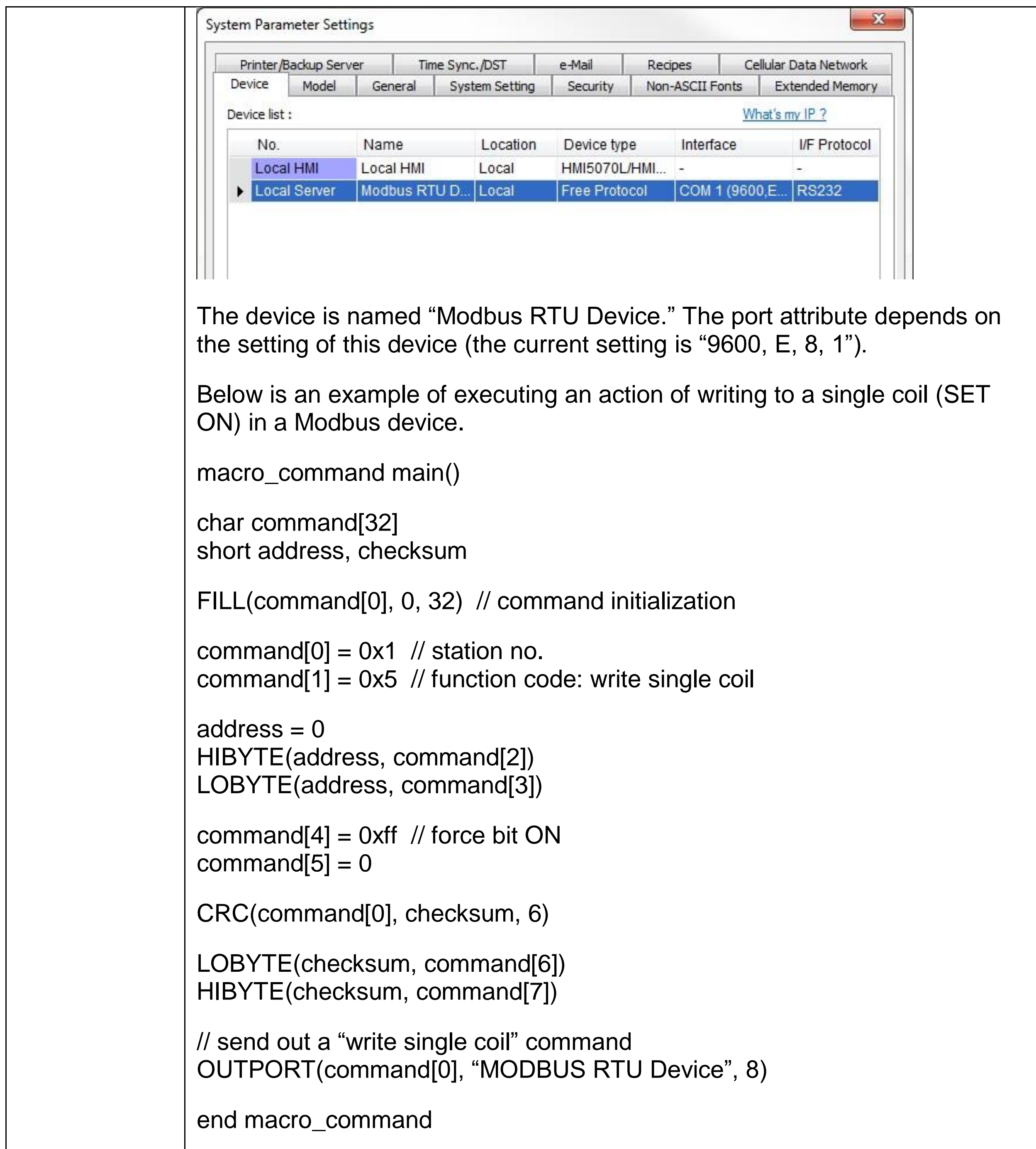
Name	XORSUM
Syntax	XORSUM(*source[start], result, data_count*)
Description	Uses an exclusion method to calculate the checksum from *source[start]* to *source[start+(data_count-1)]*. Puts the checksum into *result*. *Result* must be a variable. *Data_count* is the number of calculated elements of the array and can be a constant or a variable.
Example	macro_command main() char source[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	Uses an XOR method 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 number of calculated elements of the array and can be a constant or a variable.
Example	macro_command main() char source[5] = {0x1, 0x2, 0x3, 0x4, 0x5} short checksum BCC(source[0], checksum, 5) // checksum is 0x1 end macro_command

Name	CRC
Syntax	CRC(*source[start], result, data_count*)
Description	Calculates a 16-bit CRC of the variables from *source[start]* to *source[start+(data_count-1)]*. Puts the 16-bit CRC into *result*. *Result* must be a variable. *Data_count* is the number of calculated elements of the array and can be a constant or a variable.
Example	macro_command main() char source[5] = {0x1, 0x2, 0x3, 0x4, 0x5} short 16bit_CRC CRC(source[0], 16bit_CRC, 5) // 16bit_CRC is 0xbb2a 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+(count-1)]* to the PLC via a COM port or 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 in the Device list as follows:



The device is named “Modbus RTU Device.” The port attribute depends on the setting of this device (the current setting is “9600, E, 8, 1”).

Below is an example of executing an action of writing to a single coil (SET ON) in 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	INPORT
Syntax	INPORT(*read_data[start], device_name, read_count, return_value*)
Description	Reads data from a COM port or Ethernet port. The data is stored to *read_data[start]* to *read_data[start+(read_count-1)]*. *Device_name* is the name of the device defined in the Device list and the device must be a “Free Protocol” –type device.

	Read_count is the required amount of data read and can be a constant or a variable. If the function is used successfully to get sufficient data, *return_value* is 1, otherwise it is 0.

Example	Below is an example of executing an action of reading holding registers from 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 the "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 Ethernet port. The data is stored to *response[start]*. *device_name* is the name of the device defined in the Device list and the device must be a “Free Protocol” –type device. *receive_len* is the length of the data received and must be a variable. The total length cannot exceed the size of response. *wait_time* (in milliseconds) can be a constant or a variable. After the data is read, if there is no upcoming data during the designated time interval, the function returns.

Example	macro_command main()
	
	short wResponse[6], receive_len, wait_time=20
	short read_no, return_value, read_data[2]
	
	INPORT2(wResponse[0], “Free Protocol”, receive_len, wait_time)
	
	if receive_len > 0 then
	SetData(wResponse[0], “Local HMI”, LW, 0, 6) // write responses to LW-0
	end if
	
	end macro_command

Name	INPORT3
Syntax	INPORT3(*response[start], device_name, read_count, receive_len*)
Description	Reads data from a COM port or Ethernet port. The data is stored to *response[start]*. 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. *device_name* is the name of the device defined in the Device list and the device must be a “Free Protocol” –type device. *read_count* is the length of the data read each time. *receive_len* is the length of the data received and must be a variable. The total length cannot 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

<table>
<tr><td></td><td>if receive_len >= 6 then
SetData(wResponse[0], “Local HMI”, LW, 0, 6) // write responses to LW-0
end if

end macro_command</td></tr>
</table>

<table>
<tr><td>Name</td><td>GetData</td></tr>
<tr><td>Syntax</td><td>GetData(read_data[start], device_name, device_type, address_offset, data_count)
or
GetData(read_data, device_name, device_type, address_offset, 1)</td></tr>
<tr><td>Description</td><td>Receives data from the PLC. The data is stored to read_data[start] to (read_data[start+data_count-1]).

data_count is the amount of data received. 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 of data from the PLC.

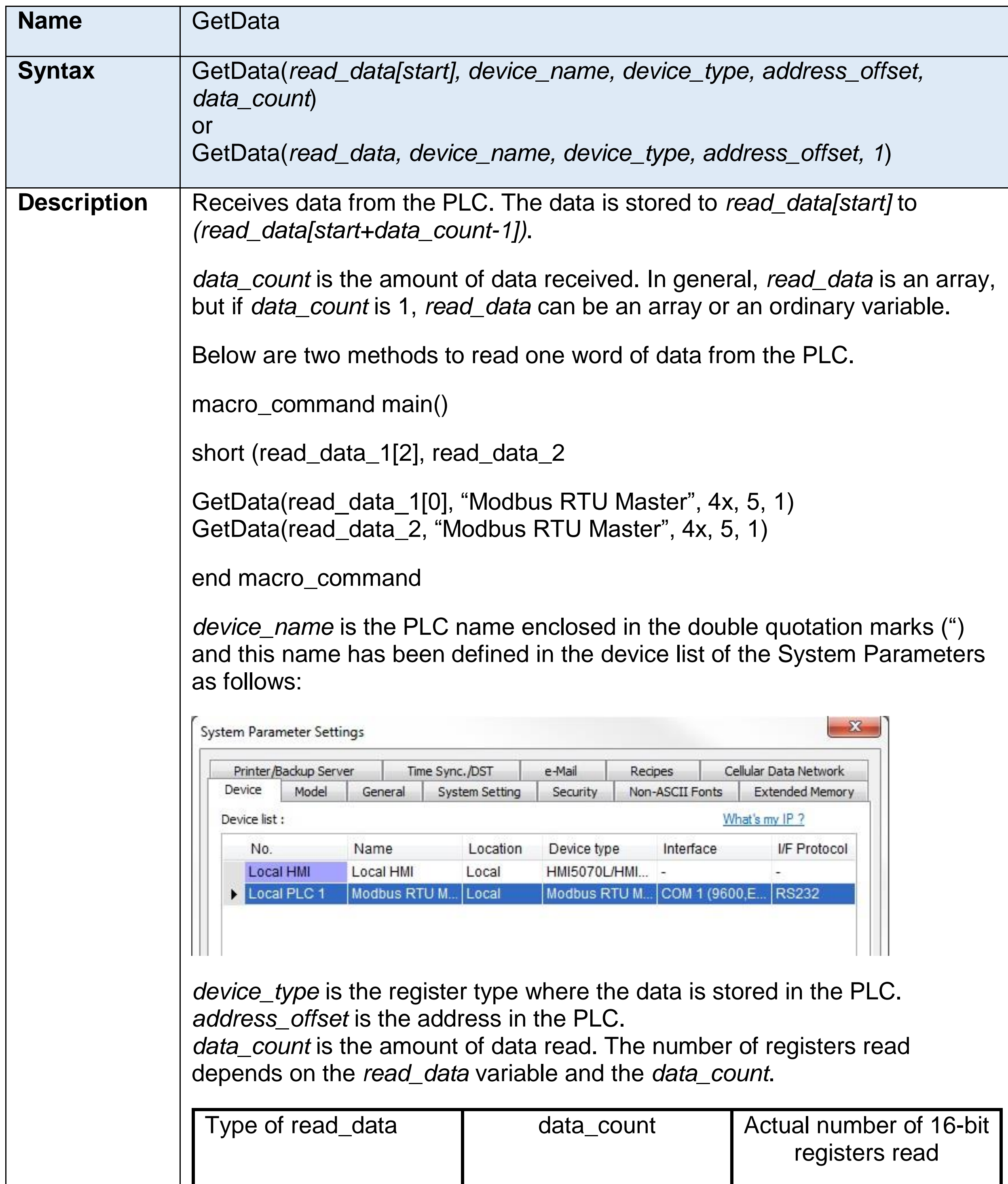
macro_command main()

short (read_data_1[2], read_data_2

GetData(read_data_1[0], “Modbus RTU Master”, 4x, 5, 1)
GetData(read_data_2, “Modbus RTU Master”, 4x, 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 the System Parameters as follows:



device_type is the register type where the data is stored in the PLC.
address_offset is the address in the PLC.
data_count is the amount of data read. The number of registers read depends on the read_data variable and the data_count.

<table><tr><td>Type of read_data</td><td>data_count</td><td>Actual number of 16-bit registers read</td></tr></table></td></tr>
</table>

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() function 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 RTU Master”, 6x, 2, 1)  // f will contain a floating point value

end macro_command
```

Example

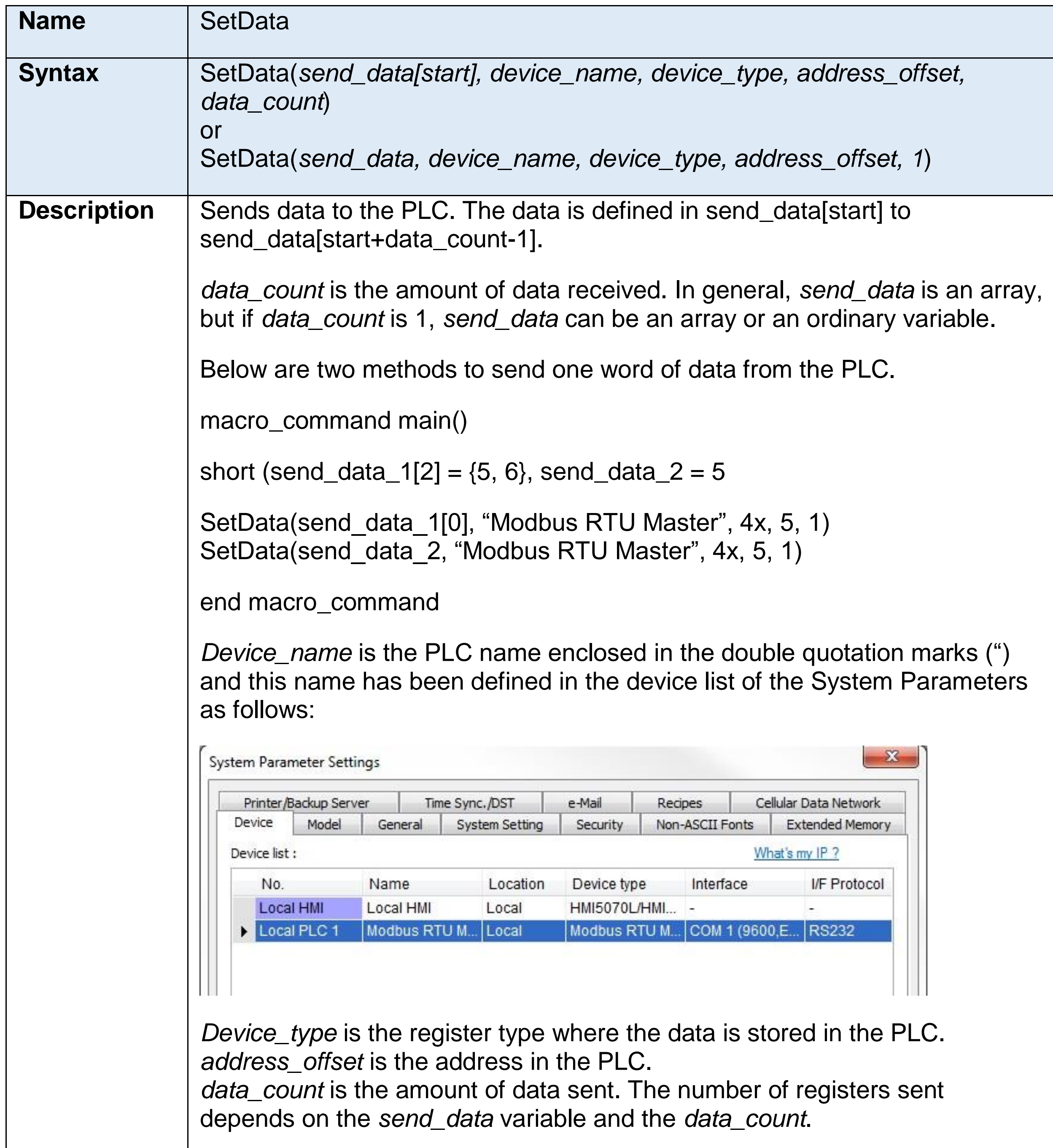
```
macro_command main()

bool a
bool b[30]
short c
short d[50]
int e
int f[10]

// get the state of LB-2 to the variable a
GetData(a, “Local HMI”, LB, 2, 1)

// get 30 states of LB-0 to LB-29 to the variables b[0] to b[29]
GetData(b[0], “Local HMI”, LB, 0, 30)

// get one word from LW-2 to the variable c
GetData(c, “Local HMI”, LW, 2, 1)

// get 50 words from LW-0 to LW-49 to the variables d[0] to d[49]
GetData(d[0], “Local HMI”, LW, 0, 50)
```

	// get 2 words from LW-6 to LW-7 to the variable e
	// note that the type of variable e is int (32-bit)
	GetData(e, "Local HMI", LW, 6, 1)
	
	// get 20 words (10 integer values) from LW-0 to LW-19 to variables f[0] to f[9]
	// note that each integer occupies 2 words
	GetData(f[0], "Local HMI", LW, 0, 10)
	
	// get 2 words from LW-2 to LW-3 to the variable f
	GetData(f, "Local HMI", LW, 2, 1)
	
	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. The data is stored to *read_data[start]* to *(read_data[start+data_count-1])*. The macro will move on to the next command even if there is no response from the PLC. Descriptions of *read_data*, *device_name, device_type, address_offset*, and *data_count* are the same as the GetData function.

Example	macro_command main()
	
	bool a
	bool b[30]
	short c
	short d[50]
	int e
	int f[10]
	
	// get the state of LB-2 to the variable a
	GetData(a, "Local HMI", LB, 2, 1)
	
	// get 30 states of LB-0 to LB-29 to the variables b[0] to b[29]
	GetData(b[0], "Local HMI", LB, 0, 30)
	
	// get one word from LW-2 to the variable c
	GetData(c, "Local HMI", LW, 2, 1)
	
	// get 50 words from LW-0 to LW-49 to the variables d[0] to d[49]
	GetData(d[0], "Local HMI", LW, 0, 50)
	
	// get 2 words from LW-6 to LW-7 to the variable e
	// note that the type of variable e is int (32-bit)
	GetData(e, "Local HMI", LW, 6, 1)

	// get 20 words (10 integer values) from LW-0 to LW-19 to variables f[0] to f[9]
	// note that each integer occupies 2 words
	GetData(f[0], “Local HMI”, LW, 0, 10)
	
	// get 2 words from LW-2 to LW-3 to the variable f
	GetData(f, “Local HMI”, LW, 2, 1)
	
	end macro_command

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. The data is defined in send_data[start] to send_data[start+data_count-1]. *data_count* is the amount of data received. 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 of data from the PLC. macro_command main() short (send_data_1[2] = {5, 6}, send_data_2 = 5 SetData(send_data_1[0], “Modbus RTU Master”, 4x, 5, 1) SetData(send_data_2, “Modbus RTU Master”, 4x, 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 the System Parameters as follows:  *Device_type* is the register type where the data is stored in the PLC. *address_offset* is the address in the PLC. *data_count* is the amount of data sent. The number of registers sent depends on the *send_data* variable and the *data_count*.

Type of read_data	data_count	Actual number of 16-bit registers 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 SetData() function is executed using a 32-bit data type (int or float), the function will automatically send int-formatted or float-formatted data to the device. For example:

```
macro_command main()

float f = 2.6
SetData(f, “Modbus RTU Master”, 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 LB-2
	SetData(a, "Local HMI", LB, 2, 1)
	
	// set 30 states of LB-0 to LB-29
	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 to LW-49
	SetData(d[0], "Local HMI", LW, 0, 50)
	
	// set the values of LW-6 to LW-7
	// note that the type of variable e is int (32-bit)
	SetData(e, "Local HMI", LW, 6, 1)
	
	// set the values of LW-0 to LW-19
	// note that 10 integers are equal to 20 words, since each integer 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. The data is defined in send_data[start] to send_data[start+data_count-1]. The macro will move on to the next command even if there is no response from the PLC. Descriptions of *send_data*, *device_name, device_type, address_offset*, and *data_count* are the same as the SetData function.
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 LB-2
	SetData(a, “Local HMI”, LB, 2, 1)
	
	// set 30 states of LB-0 to LB-29
	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 to LW-49
	SetData(d[0], “Local HMI”, LW, 0, 50)
	
	// set the values of LW-6 to LW-7
	// note that the type of variable e is int (32-bit)
	SetData(e, “Local HMI”, LW, 6, 1)
	
	// set the values of LW-0 to LW-19
	// note that 10 integers are equal to 20 words, since each integer occupies 2 words
	SetData(f[0], “Local HMI”, LW, 0, 10)
	
	end macro_command

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

Example	macro_command main()
	
	short err
	char byData[10]
	
	GetDataEx(byData[0], “Modbus RTU Master”, 4x, 1, 10) // read 10 bytes
	
	GetError(err) // save an error code to err
	// if err is equal to 0, the macro was successful in executing GetDataEx()
	
	end macro_command

Name	PURGE
Syntax	PURGE(*com_port*)
Description	*com_port* refers to the COM port number on the HMI, 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 COM port 3
	PURGE(1) // purge COM port 1
	
	end macro_command

Name	SetRTS
Syntax	SetRTS(*com_port, source*)
Description	Set RTS state for RS232. *com_port* refers to the COM port number on the HMI, which ranges from 1 to 3. It can be either a variable or a constant. This command raises the RTS signal while the value of *source* is greater than 0 and lowers the RTS signal while the value of *source* equals 0.

Example	macro_command main()
	
	char com_port = 1
	char value = 1
	
	SetRTS(com_port, value) // raises RTS signal of COM 1 while *value* > 0
	
	SetRTS(1, 0) // lowers RTS signal of COM 1
	
	end macro_command

Name	GetCTS
Syntax	GetCTS(*com_port, result*)
Description	Get CTS state for RS232. *com_port* refers to the COM port number on the HMI, which ranges from 1 to 3. It can be either a variable or a constant. *result* is used for receiving the CTS signal. It must be a variable.

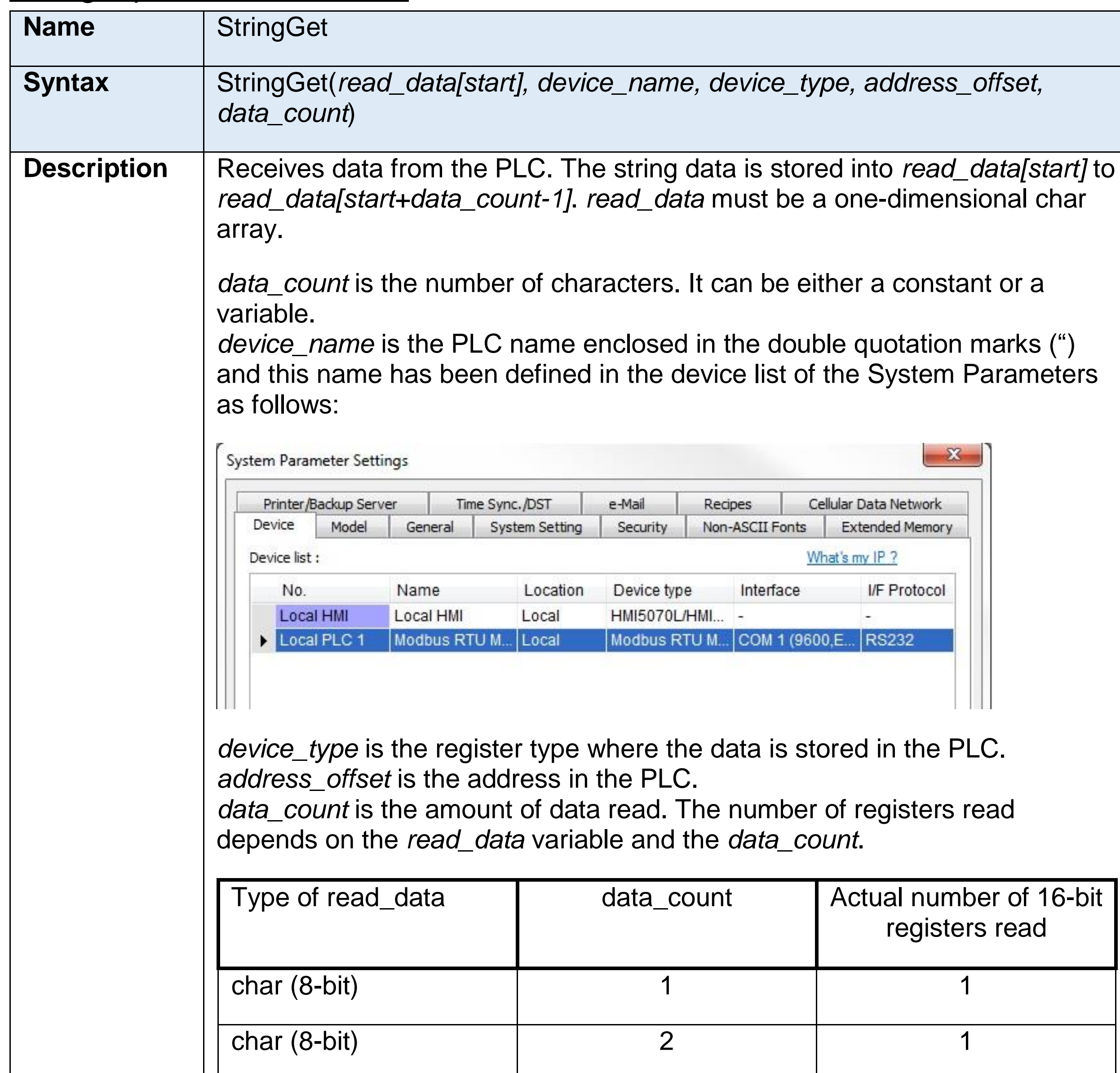
	This command receives the CTS signal and stores the received data in the *result* variable. When the CTS signal is pulled high, it writes a 1 to *result*, otherwise, it writes a 0.

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

String Operation Functions

<table>
<tr><td>Name</td><td>StringGet</td></tr>
<tr><td>Syntax</td><td>StringGet(read_data[start], device_name, device_type, address_offset, data_count)</td></tr>
<tr><td>Description</td><td>Receives data from the PLC. The string data is stored into read_data[start] to read_data[start+data_count-1]. read_data must be a one-dimensional char array.

data_count is the number of 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 the System Parameters as follows:



device_type is the register type where the data is stored in the PLC.
address_offset is the address in the PLC.
data_count is the amount of data read. The number of registers read depends on the read_data variable and the data_count.

<table>
<tr><th>Type of read_data</th><th>data_count</th><th>Actual number of 16-bit registers read</th></tr>
<tr><td>char (8-bit)</td><td>1</td><td>1</td></tr>
<tr><td>char (8-bit)</td><td>2</td><td>1</td></tr>
</table>
</td></tr>
</table>

	One WORD register (16-bit) can contain two ASCII characters. Reading two ASCII characters is equivalent to reading one 16-bit register.

Example	macro_command main()
	
	char str1[20]
	
	// read 10 words from LW-0 to LW-9 to the variables str1[0] to str1[19]
	// since one word can store two ASCII characters, reading 20 ASCII characters is
	// actually reading 10 16-bit registers.
	
	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. The string data is stored into *read_data[start]* to *read_data[start+data_count-1]*. The macro will move on to the next command even if there is no response from the PLC. Descriptions of *read_data, device_name, device_type, address_offset,* and *data_count* are the same as StringGet.

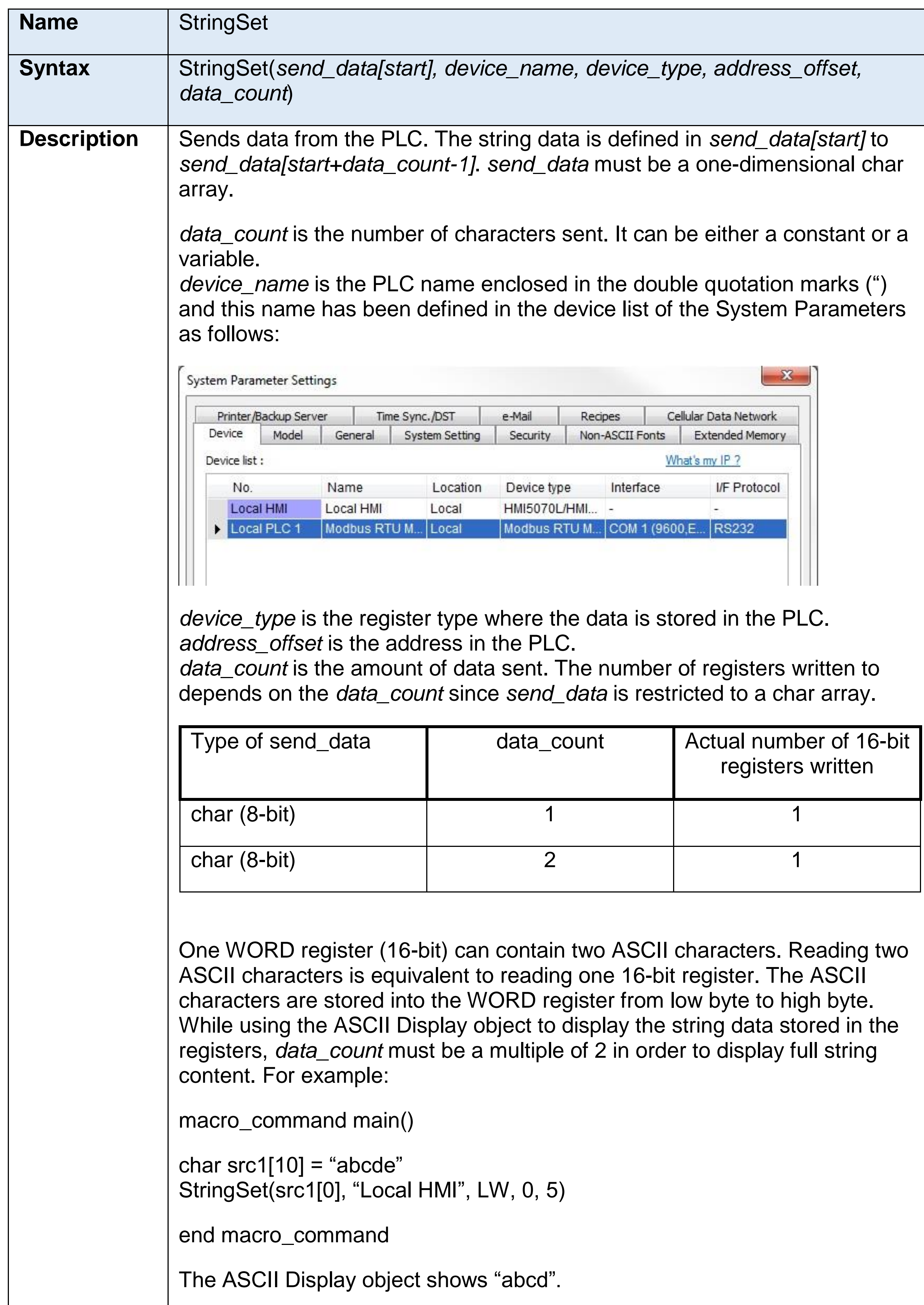
Example	macro_command main()
	
	char str1[20]
	short test = 0
	
	// the macro will continue executing test = 1 even if the Modbus device is not
	// responding.
	
	StringGetEx(str1[0], “Modbus RTU Master”, 4x, 0, 20)
	test = 1
	
	// the macro will not continue executing test = 2 until the Modbus device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

Name	StringSet
Syntax	StringSet(*send_data[start], device_name, device_type, address_offset, data_count*)

Description

Sends data from the PLC. The string data is defined in *send_data[start]* to *send_data[start+data_count-1]*. *send_data* must be a one-dimensional char array.

data_count is the number of characters sent. 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 the System Parameters as follows:



device_type is the register type where the data is stored in the PLC.
address_offset is the address in the PLC.
data_count is the amount of data sent. The number of registers written to depends on the *data_count* since *send_data* is restricted to a char array.

Type of send_data	data_count	Actual number of 16-bit registers written
char (8-bit)	1	1
char (8-bit)	2	1

One WORD register (16-bit) can contain two ASCII characters. Reading two ASCII characters is equivalent to reading 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 “abcd”.

	If data_count is an even number that is greater than or equal to the length of the string, the content of the string can be completely shown: macro_command main() char src1[10] = “abcde” StringSet(src1[0], “Local HMI”, LW, 0, 6) end macro_command The ASCII Display object shows “abcde”.

Example	macro_command main()
	
	char str1[10] = “abcde”
	
	// send 3 words to LW-0 to LW-2
	// data is being sent until the end of the string is reached
	// even though the value of data_count is larger than the length of the 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. The string data is stored into *send_data[start]* to *send_data[start+data_count-1]*. The macro will move on to the next command even if there is no response from the PLC. 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
	
	// the macro will continue executing test = 1 even if the Modbus device is not
	// responding.
	
	StringSetEx(str1[0], “Modbus RTU Master”, 4x, 0, 20)
	test = 1
	
	// the macro will not continue executing test = 2 until the Modbus device responds
	StringSet(str1[0], “Modbus RTU Master”, 4x, 0, 20)
	test = 2
	
	end macro_command

Name	StringCopy
Syntax	StringCopy(“*source”, destination[start]*) or StringCopy(source[start], destination[start])
Description	Copy 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 a static string (in the form: “*source*”) and a char array (in the form: *source[start]*). *destination[start]* must be a one-dimensional char array. This function returns a Boolean indicating whether the process is successful or not. If successful, it returns true, otherwise it returns false. If the length of the *source* string exceeds the maximum size of the destination buffer, it returns false and the content of the *destination* buffer 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	StringDecAsc2Bin
Syntax	Success = StringDecAsc2Bin(source[start], destination) or Success = StringDecAsc2Bin(“source”, destination)
Description	This function converts a decimal string to binary data. It converts the decimal string in the *source* parameter into binary data, and stores it in the *destination* variable. The *source* string parameter accepts both a static string (in the form: “source”) and a char array (in the form: source[start]). *destination* must be a variable, to store the result of the conversion. This function returns a Boolean indicating whether the process is successful or not. If successful, it returns true, otherwise it returns false. If the *source* string contains characters other than ‘0’ to ‘9’, 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)
	// success = true, result1 is 12345
	
	char result2
	bool success2
	success2 = StringDecAsc2Bin(“32768”, result2)
	// success2=true, but the result exceeds the data range of result2
	
	char src3[2] = “4b”
	char result3
	bool success3
	success3 = StringDecAsc2Bin(src3[0], result3)
	// success3 = false, because src3 contains characters other than ‘0’ to ‘9’.
	
	end macro_command

Name	StringBin2DecAsc
Syntax	Success = StringBin2DecAsc(source, destination[start])
Description	This function converts binary data into a decimal string. It converts the binary data in the *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 a one-dimensional char array, to store the result of the conversion. This function returns a Boolean indicating whether the process is successful or not. If successful, it returns true, otherwise it returns false. If the length of the decimal string after conversion exceeds the size of the *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 floating point values. It converts the decimal string in the *source* parameter into floats and stores it in the *destination* variable.

	The *source* string parameter accepts both a static string (in the form: “source”) and a char array (in the form: source[start]). *destination* must be a variable, to store the result of the conversion. This function returns a Boolean indicating whether the process is successful or not. If successful, it returns true, otherwise it returns false. If the *source* string contains characters other than ‘0’ to ‘9’, 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 floating point value into a decimal string. It converts the float in the *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 a one-dimensional char array, to store the result of the conversion. This function returns a Boolean indicating whether the process is successful or not. If successful, it returns true, otherwise it returns false. If the length of

	the decimal string after conversion exceeds the size of the *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 the *source* parameter into binary data, and stores it in the *destination* variable. The *source* string parameter accepts both a static string (in the form: “source”) and a char array (in the form: source[start]). *destination* must be a variable, to store the result of the conversion. This function returns a Boolean indicating whether the process is successful 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)
	// success = true, result1 is 3c
	
	short result2
	bool success2
	success2 = StringJexAsc2Bin(“1a2b3c4d”, result2)
	// success2=true, result2 = 3c4d. The result exceeds the data range of result2.
	
	char src3[2] = “4g”
	char result3
	bool success3
	success3 = StringHexAsc2Bin(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 into a hexadecimal string. It converts the binary data in the *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 a one-dimensional char array, to store the result of the conversion. This function returns a Boolean indicating whether the process is successful or not. If successful, it returns true, otherwise it returns false. If the length of the hexadecimal string after conversion exceeds the size of the destination buffer, it returns false. The *success* field is optional.
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

Name	StringMid
Syntax	Success = StringMid(source[start], count, destination[start]) or Success = StringMid("string", start, count, destination[start])
Description	Retrieve a character sequence from the specified offset of the source string and store 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 *[start]*. For static source string (*source*), the second parameter (*start*) specifies the start offset of the substring. The *count* parameter specifies the length of the substring being retrieved. *Destination* must be a one-dimensional char array to store the retrieved substring. This function returns a Boolean indicating whether the process is successful or not. If successful, it returns true, otherwise it returns false. If the length of the retrieved substring exceeds the size of the *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	StringLength
Syntax	length = StringLength(source[start]) or length = StringLength(“source”)
Description	Obtain the length of a string. It returns the length of the *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]*). For *source[start]*, the start offset of the substring is specified by the index value *[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’}
	in length2
	length2 = StringLength(src2[0])
	// length2 = 5
	
	char src3[20] = “abcdefghij”
	int length3
	length3 = StringLength(src3[2])
	// length3 = 8
	
	end macro_command

Name	StringCat
Syntax	success = StringCat(source[start], destination[start]) or success = StringCat(“source”, destination[start])
Description	Appends the *source* string to the *destination* string. It adds the contents of the *source* string to the end of the contents of the *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 a one-dimensional char array.

	This function returns a Boolean indicating whether the process is successful or not. If successful, it returns true, otherwise it returns false. If the length of the string after concatenation exceeds the size of the *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 = "1234567890abcdefghij"
	
	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	Performs 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 whether the process is successful or not. 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	Performs 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 whether the process is successful or not. 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	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 the 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 a zero-based index of the first character of the 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 matching substring, it returns -1.

Example	macro_command main()
	
	char src1[20] = “abcde”
	char target1[20] = “cd”
	bool pos1
	pos1 = StringFind(src1[0], target1[0])
	// pos1 = 2
	
	char target2[20] = “ce”
	bool pos2
	pos2 = StringFind(“abcde”, target2[0])
	// pos2 = -1
	
	char src3[20] = “abcde”
	bool pos3
	pos3 = StringFind(src3[3], “cd”)
	// pos3 = -1
	
	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 the 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 a zero-based index of the first character of the last occurrence of the 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 matching substring, it returns -1.

Example	macro_command main()
	
	char src1[20] = “abcdeabcde”
	char target1[20] = “cd”
	bool pos1
	pos1 = StringReverseFind(src1[0], target1[0])
	// pos1 = 7
	
	char target2[20] = “ce”
	bool pos2
	pos2 = StringReverseFind(“abcdeabcde”, target2[0])
	// pos2 = -1
	
	char src3[20] = “abcdeabcde”
	bool pos3
	pos3 = StringReverseFind(src3[6], “ab”)
	// 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 a zero-based index of the first character in the *source* string that is also in the *target* string. Notice that the entire sequence of characters to find must be matched. If there is no match, it returns -1.
Example	macro_command main() char src1[20] = “abcdeabcde” char target1[20] = “sdf” bool pos1 pos1 = StringFindOneOf(src1[0], target1[0]) // pos1 = 3 char src2[20] = “abcdeabcde” bool pos2

	pos2 = StringFindOneOf(src2[1], “agi”)
	// pos2 = 4
	
	char target3[20] = “bus”
	bool pos3
	pos3 = StringFindOneOf(“abcdeabcde”, target3[1])
	// pos3 = -1
	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 the 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 *set* 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 successful or not. If successful, it returns true, otherwise it returns false. If the length of the retrieved substring exceeds the size of the *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, dest 3 remains the same
	
	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 *set* 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 successful or not. If successful, it returns true, otherwise it returns false. If the length of the retrieved substring exceeds the size of the *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, dest 3 remains the same
	
	end macro_command

Name	StringToUpper
Syntax	success = StringToUpper(source[start], destination[start]) success = StringToUpper(“source”, destination[start])
Description	Convert all the characters in the source string to uppercase characters and store 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 successful or not. If successful, it returns true, otherwise it returns false. If the length of the *result* string after conversion exceeds the size of the *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	Convert all the characters in the source string to lowercase characters and store 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 successful or not. If successful, it returns true, otherwise it returns false. If the length of the *result* string after conversion exceeds the size of the *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	Reverse the characters in the *source* string and store 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 successful or not. If successful, it returns true, otherwise it returns false. If the length of the reversed string exceeds the size of the *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	Trim the specified characters in the *set* buffer from the leading end of 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 successful or not. If successful, it returns true, otherwise it returns false. If the length of the trimmed string exceeds the size of the *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”, 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	Trim the specified characters in the *set* buffer from the trailing end of 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 successful or not. If successful, it returns true, otherwise it returns false. If the length of the trimmed string exceeds the size of the *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 success4

	success4 = StringTrimRight(src3[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 successful or not. If successful, it returns true, otherwise it returns false. If the length of the string after insertion exceeds the size of the *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

Recipe Query Functions

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

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	Use SQL statements to query recipe data. The number of records from the query result will be stored in the *destination*. This must be a variable. SQL commands can be static string or char array. Example: RecipeQuery(“SELECT * FROM TypeA”, destination) or RecipeQuery(sql[0], destination) A SQL statement must start with “SELECT * FROM” followed by a recipe name and query condition.
Example	macro_command main() int total_row = 0 char sql[100] = “SELECT * FROM TypeB” bool result

	result = RecipeQuery("SELECT * FROM TypeA", total_row)
	// Query recipe "TypeA". Store the number of records from the query result in *total_row*.
	
	result = RecipeQuery(sql[0], total_row)
	// Query recipe "TypeB". Store the number of records from the query result in *total_row*.
	
	end macro_command

Name	RecipeQueryGetData
Syntax	RecipeQueryGetData(destination, recipe_address, result, 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 the *recipe_address* as RecipeQuery. *result_row_no* specifies the sequence row number in the 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 from the 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 recordID numbers of those records obtained by RecipeQuery. This function must be called after calling RecipeQuery. *result_row_no* specifies the sequence row number in the query result, and writes the obtained recordID 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 from the 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 a recipe. *recipe_address* consists of the recipe name and item name: "recipe_name.item_name". *record_ID* specifies the ID number of the record in the recipe being modified. If successful, it returns true, otherwise it returns false.

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 the item name is "item_weight" in record 0.
	
	recordID = 1
	result = RecipeSetData(str[0], "TypeB.item_name", recordID)
	// Set data to recipe "TypeB" where the item name is "item_name" in record 1.
	
	end macro_command

Miscellaneous

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

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

Example	macro_command main()
	
	char ON = 1, OFF = 0
	
	Buzzer(ON) // turns ON the buzzer
	
	DELAY(1000) // delay 1 second
	
	Buzzer(OFF) // turns OFF the buzzer
	
	DELAY(500) // delay 500 milliseconds
	
	Buzzer(1) // turns ON the buzzer
	
	DELAY(1000) // delay 1 second
	
	Buzzer(0) // turns OFF the buzzer
	
	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 whose ID is 5
	
	SYNC_TRIG_MACRO(“macro_1”) // call a macro whose name is macro_1
	
	SetData(OFF, “Local HMI”, LB, 0, 1)
	
	end macro_command

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 whose ID is 5
	
	ASYNC_TRIG_MACRO(“macro_1”) // call a macro whose name is macro_1
	
	SetData(OFF, “Local HMI”, LB, 0, 1)
	
	end macro_command

Name	TRACE
Syntax	TRACE(format, argument)
Description	Use this function to send a specified string to the EasyDiagnoser. Users can print out the current value of variables during run-time of a 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 bold), has the following form:

<table>
<tr><td></td><td>%[flags] [width] [precision] type

Each field of the format specification is described below:

flags (optional):
-
+

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.macro_id can be a constant or a variable.

E or e : Signed value having 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 –.

f : Signed value having the form [–]dddd.dddd, where dddd is one or more decimal digits.

The length of the output string is limited to 256 characters. The extra characters will be ignored.

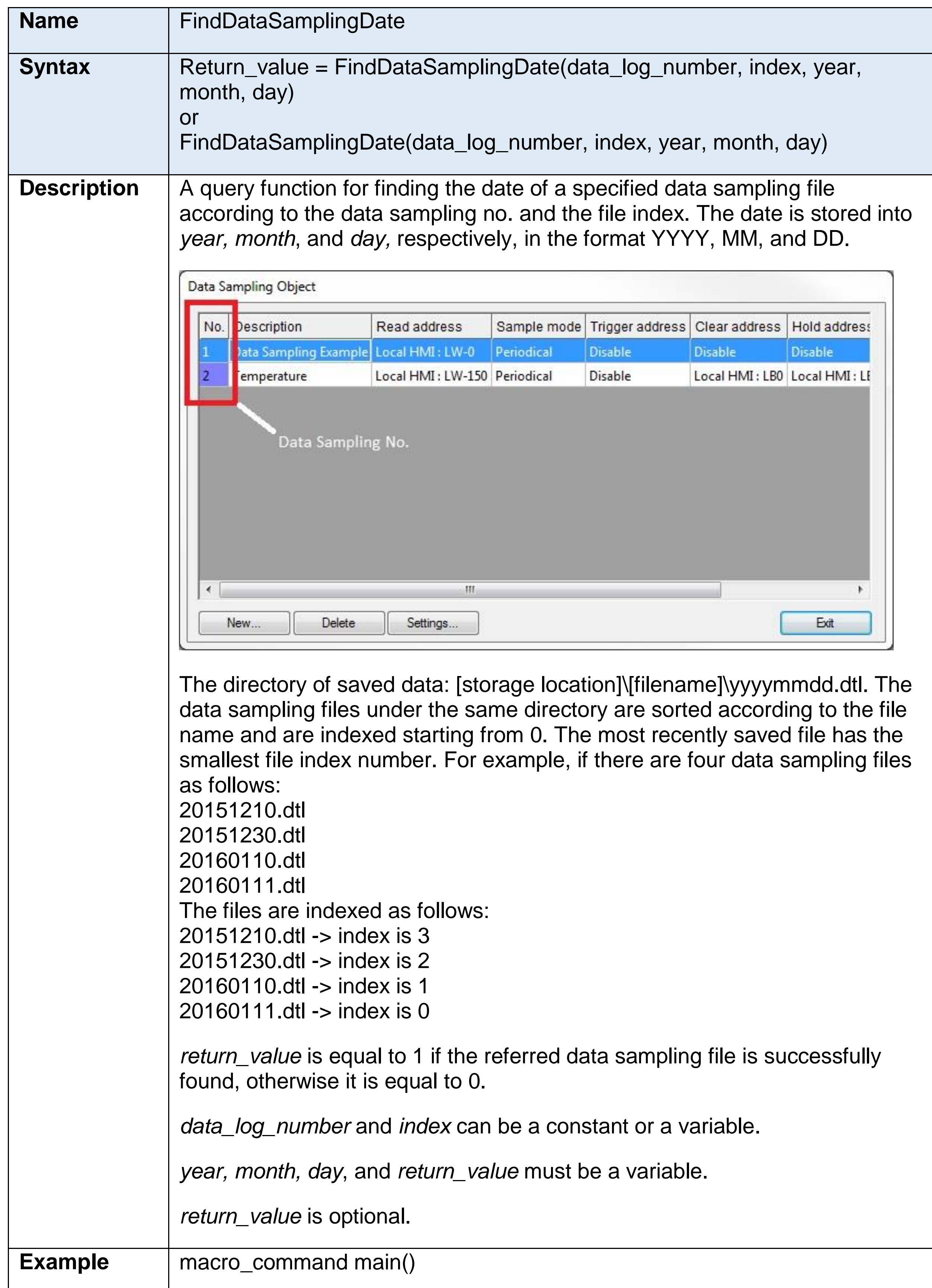
The argument part is optional. One format specification converts exactly one argument.</td></tr>
<tr><td>Example</td><td>macro_command main()

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</td></tr>
</table>

<table>
<tr><td>Name</td><td>FindDataSamplingDate</td></tr>
<tr><td>Syntax</td><td>Return_value = FindDataSamplingDate(data_log_number, index, year, month, day)
or
FindDataSamplingDate(data_log_number, index, year, month, day)</td></tr>
<tr><td>Description</td><td>A query function for finding the date of a 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 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:
20151210.dtl
20151230.dtl
20160110.dtl
20160111.dtl
The files are indexed as follows:
20151210.dtl -> index is 3
20151230.dtl -> index is 2
20160110.dtl -> index is 1
20160111.dtl -> index is 0

return_value is equal to 1 if the referred data sampling file is successfully found, otherwise it is equal to 0.

data_log_number and index can be a constant or a variable.

year, month, day, and return_value must be a variable.

return_value is optional.</td></tr>
<tr><td>Example</td><td>macro_command main()</td></tr>
</table>

	short data_log_number = 1, index = 2, year, month, day
	short success
	
	// if there exists a data sampling file named 20151230.dtl with data sampling number // 1 and file index 2, the result after execution:
	// success == 1, year == 2015, month == 12, and day == 30.
	
	success = FindDataSamplingDate(data_log_number, index, year, month, day)
	
	end macro_command

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 a 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 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: 20151210.dtl 20151230.dtl 20160110.dtl 20160111.dtl The files are indexed as follows: 20151210.dtl -> index is 3

	20151230.dtl -> index is 2 20160110.dtl -> index is 1 20160111.dtl -> index is 0 *return_value* is equal to 1 if the referred data sampling file is successfully found, otherwise it is equal to 0. *data_log_number*, *year, month* and *day* can be a constant or a variable. *index* and *return_value* must be a variable. *return_value* is optional.

Example	macro_command main()
	
	short data_log_number = 1, year = 2015, month = 12, day = 10, index
	short success
	
	// if there exists a data sampling file named 20151230.dtl with data sampling number // 1 and file index 2, the result after execution:
	// success == 1, index == 2.
	
	success = FindDataSamplingIndex(data_log_number, year, month, day, index)
	
	end macro_command

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 a specified event log file according to the file index. The date is stored into *year, month*, and *day* respectively in the format YYYY, MM, and DD. The event log files are 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: 20151210.evt 20151230.evt 20160110.evt 20160111.evt The files are indexed as follows: 20151210.evt -> index is 3 20151230.evt -> index is 2 20160110.evt -> index is 1 20160111.evt -> index is 0

	return_value is equal to 1 if the referred event log file is successfully found, otherwise it is equal to 0. *index* can be a constant or a variable. *year, month, day*, and *return_value* must be a variable. *return_value* is optional.

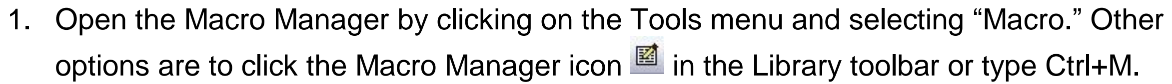
Example	macro_command main()
	
	short index = 1, year, month, day
	short success
	
	// if there exists an event log file named 20151230.evt with file index 1, the result
	// after execution:
	// success == 1, year == 2015, month == 12, and day == 30.
	
	success = FindEventLogDate(index, year, month, day)
	end macro_command

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 a specified event log file according to the date. The file index is stored into *index*. *year, month*, and *day* are in the format 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: 20151210.evt 20151230.evt 20160110.evt 20160111.evt The files are indexed as follows: 20151210.evt -> index is 3 20151230.evt -> index is 2 20160110.evt -> index is 1 20160111.evt -> index is 0 *return_value* is equal to 1 if the referred event log file is successfully found, otherwise it is equal to 0. *index* can be a constant or a variable. *year, month*, *day,* and *return_value* must be a variable.

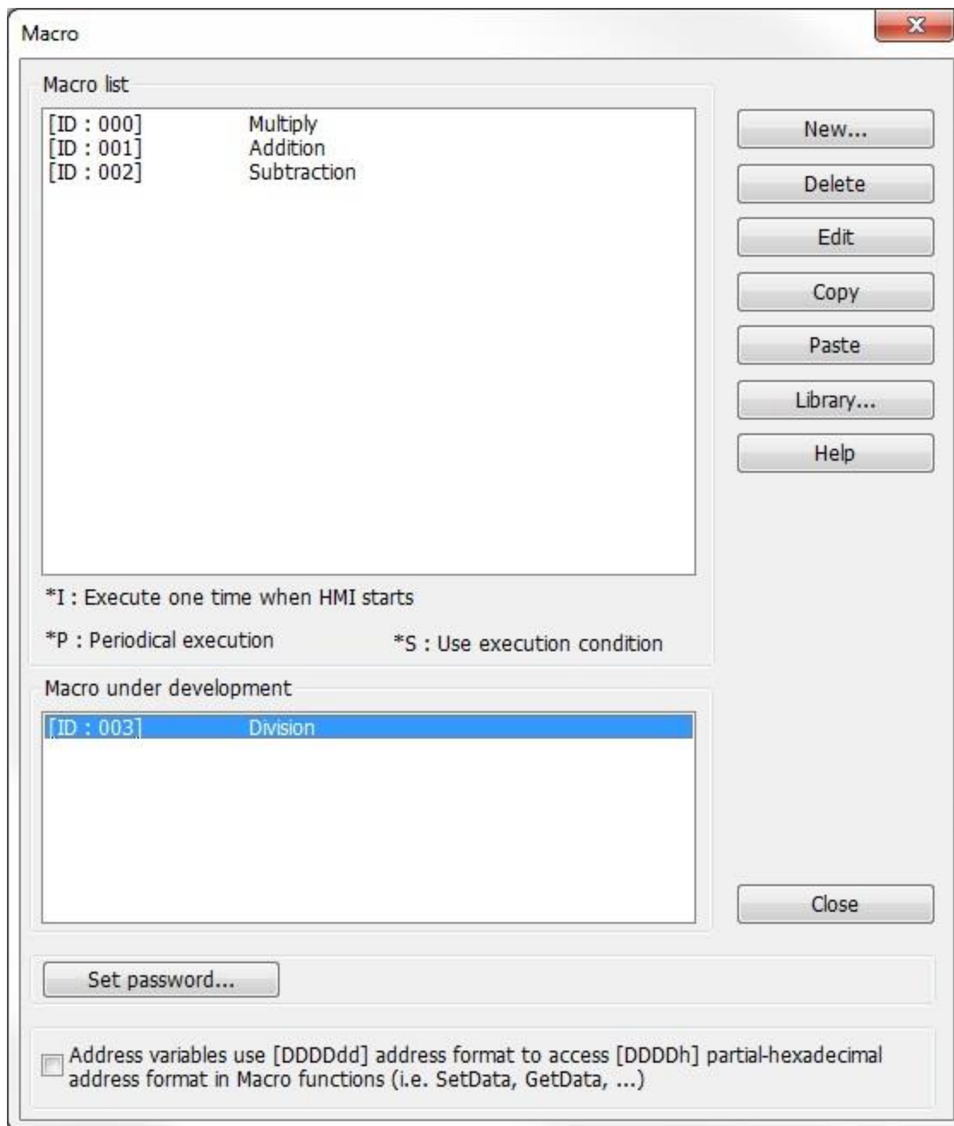
	return_value is optional.
Example	macro_command main() short year = 2015, month = 12, day = 10, index short success // if there exists an event log file named 20151230.evt with data file index 2, the // result after execution: // success == 1, index == 2. success = FindEventLogIndex(year, month, day, index) end macro_command

8. How to Create and Execute a Macro

How to Create a Macro



1. Open the Macro Manager by clicking on the Tools menu and selecting “Macro.” Other options are to click the Macro Manager icon in the Library toolbar or type Ctrl+M.

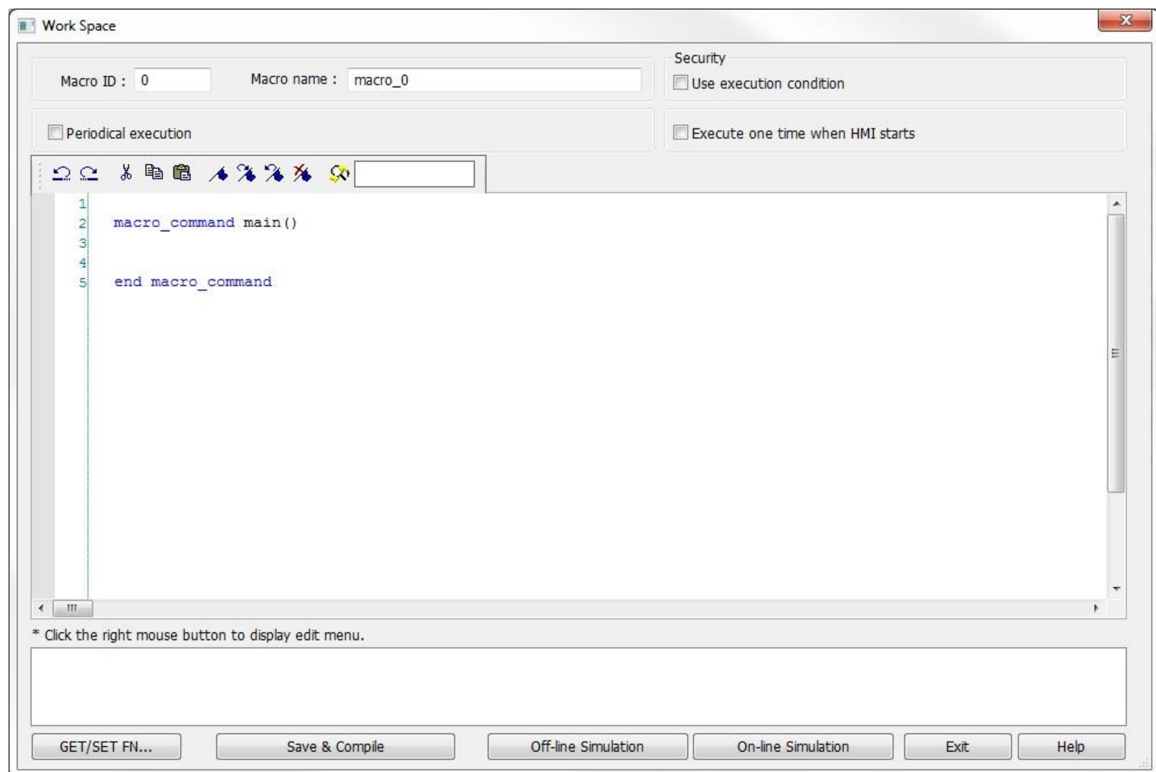


In the Macro Manager, all macros compiled successfully are displayed in the “Macro list” and all macros under development or not compiled are displayed in the “Macro under development” window.

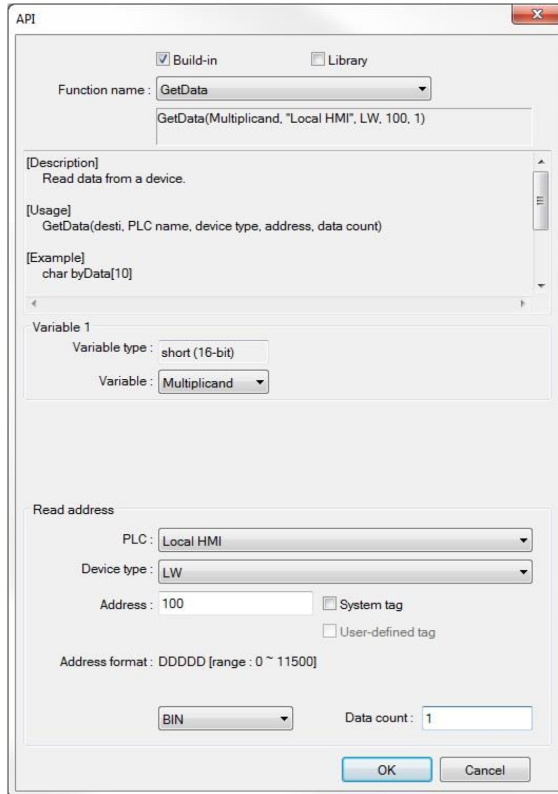
The following table lists the various buttons and their descriptions:

Setting	Description
New	Opens a blank Work Space editor for creating a new macro.
Delete	Deletes the selected macro.
Edit	Opens the Work Space editor and loads the selected macro for editing.
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.
OK	Click this button to save the contents of the edited macros before leaving this dialog.
Cancel	Cancel the edits and leave the macro editing dialog without saving changes.
Library	Opens the Macro Function Library.

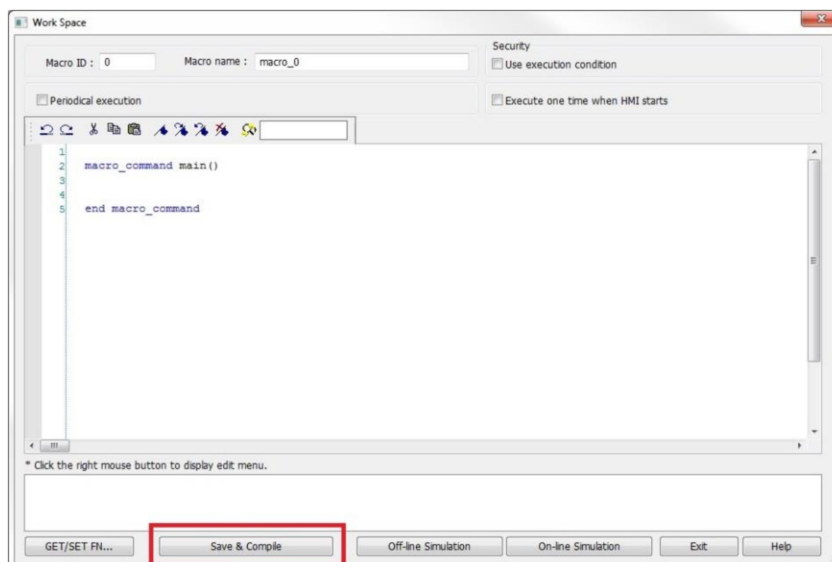
2. Press the “New” button to create an empty macro and open the macro editor (Work Space). Every macro has a unique number assigned as the Macro ID and must have a macro name, otherwise an error will appear when compiling.



3. Design your macro. To use built-in functions like SetData() or GetData(), click the "GET/SET FN…" button to open the API dialog box. Select the function and set the appropriate parameters.



4. Once the macro is completed, click the "Save & Compile" button to compile the macro.



5. If there is no error, click the "Exit" button and the new macro will be added to the "Macro list."

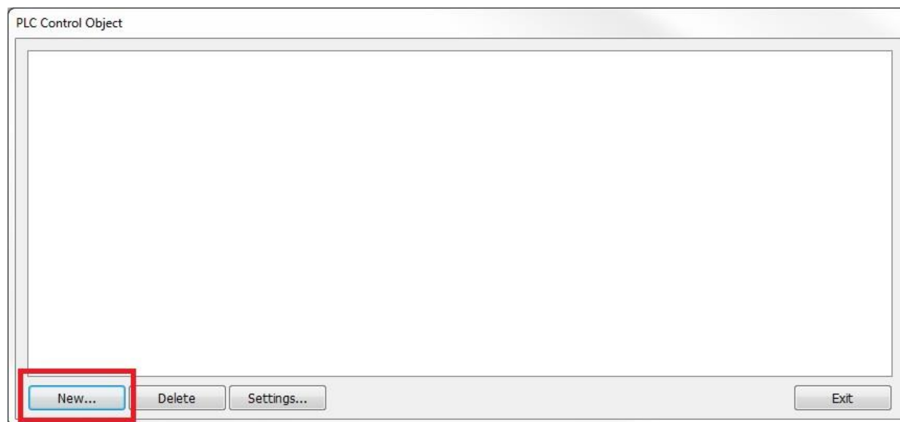
How to Execute a Macro

There are several ways to execute a macro.

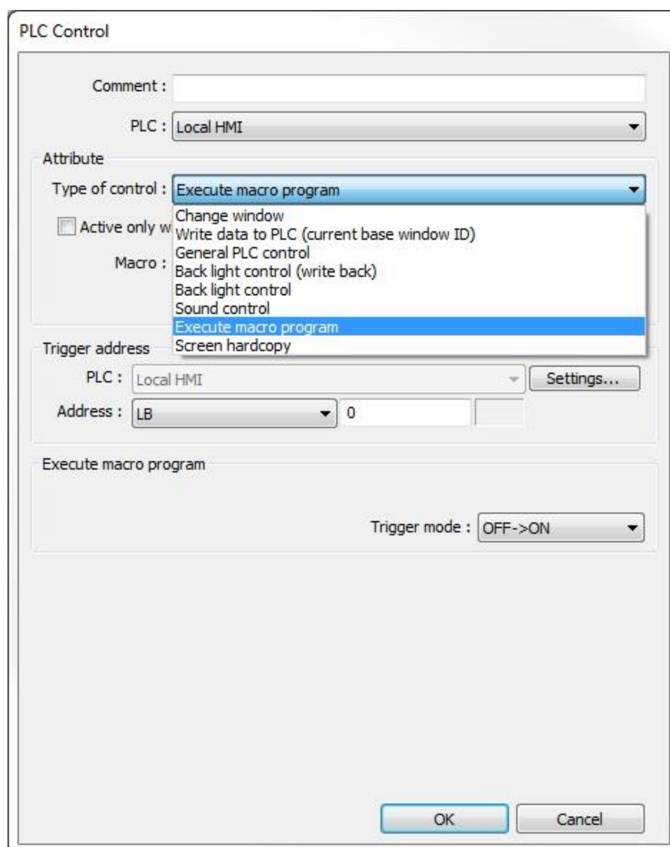
Note: A macro must exist in the Macro list for the "Execute macro" option to become available in the following examples.

PLC Control Object

1. Open the PLC Control object (Objects menu > PLC Control) and click "New."



2. Under "Type of control," select "Execute macro program."

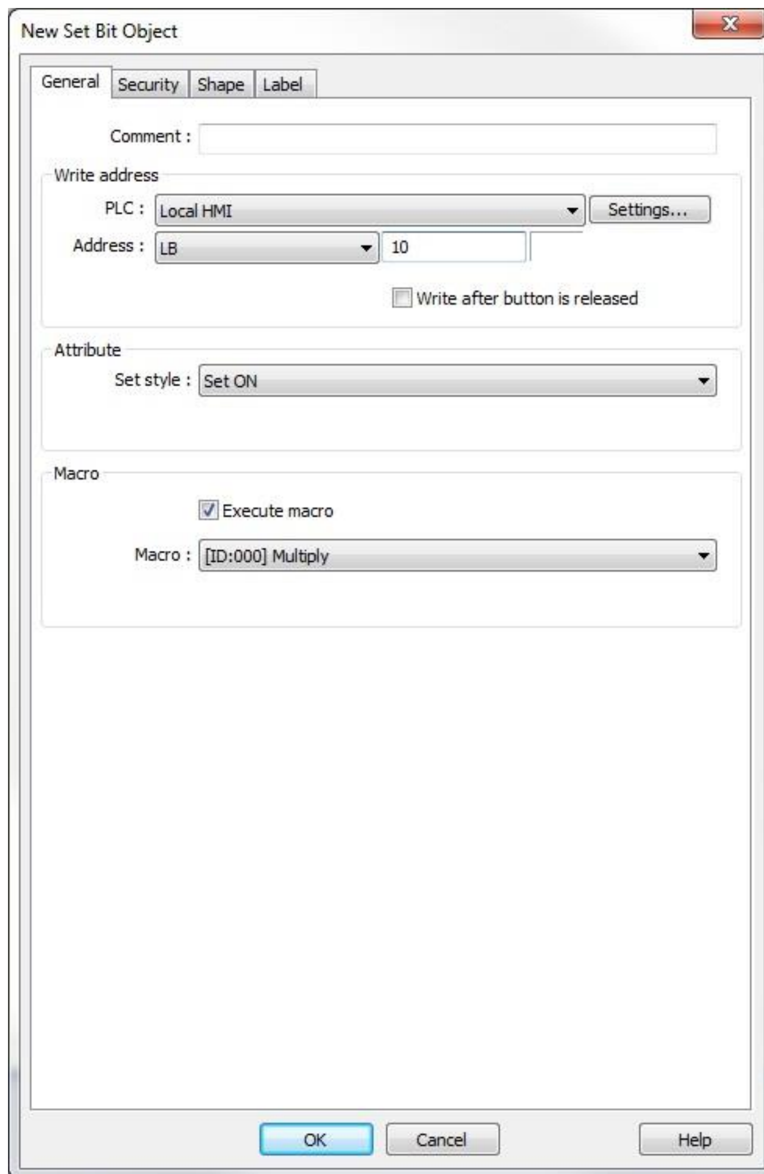


3. Select the macro you wish to execute under “Macro.” Under “Trigger address,” choose a bit and select a trigger condition to trigger the macro. In order to guarantee that the macro will run only once, consider latching the trigger bit and then resetting the trigger condition within the macro.

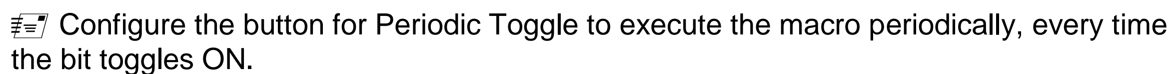
4. Use a Set Bit or Toggle Switch object to trigger the bit and execute the macro.

Set Bit or Toggle Switch Object

1. Select a Set Bit or Toggle Switch object under the Objects > Button menu. On the General tab of the object's Properties window, select the "Execute macro" option.

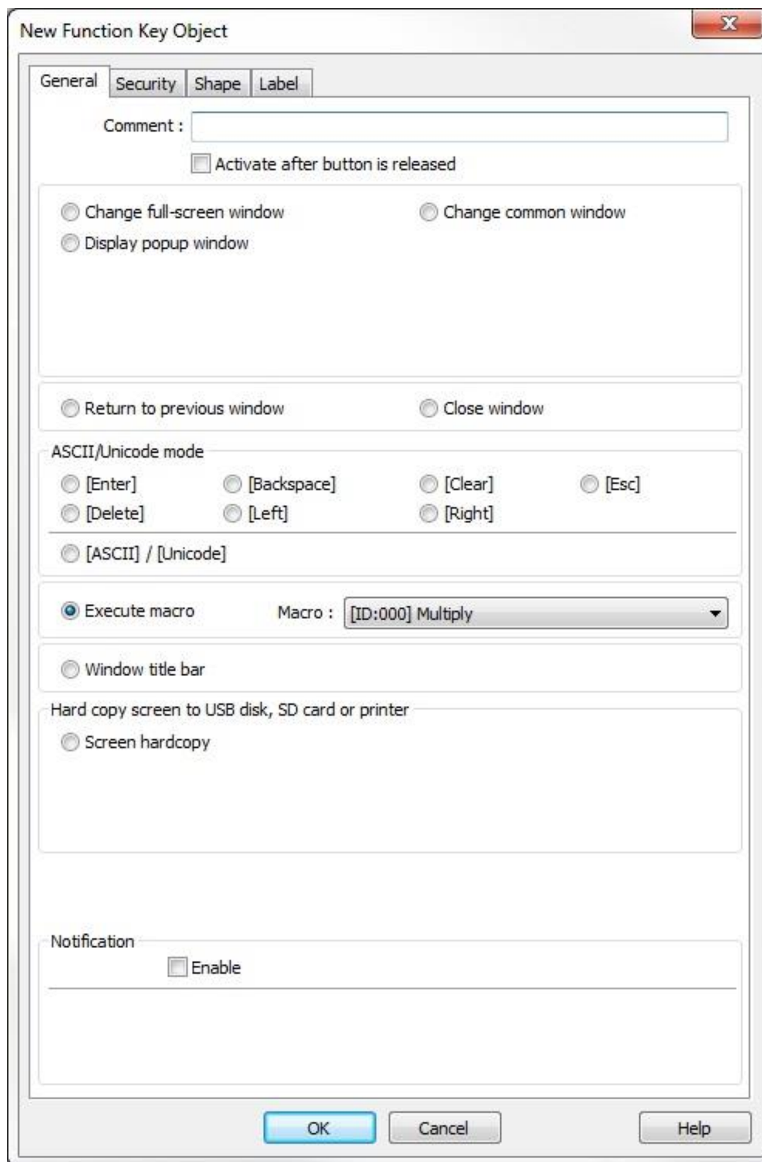


2. Select the macro you wish to execute. The macro will be executed one time when the button is activated.


Configure the button for Periodic Toggle to execute the macro periodically, every time the bit toggles ON.

Function Key Object

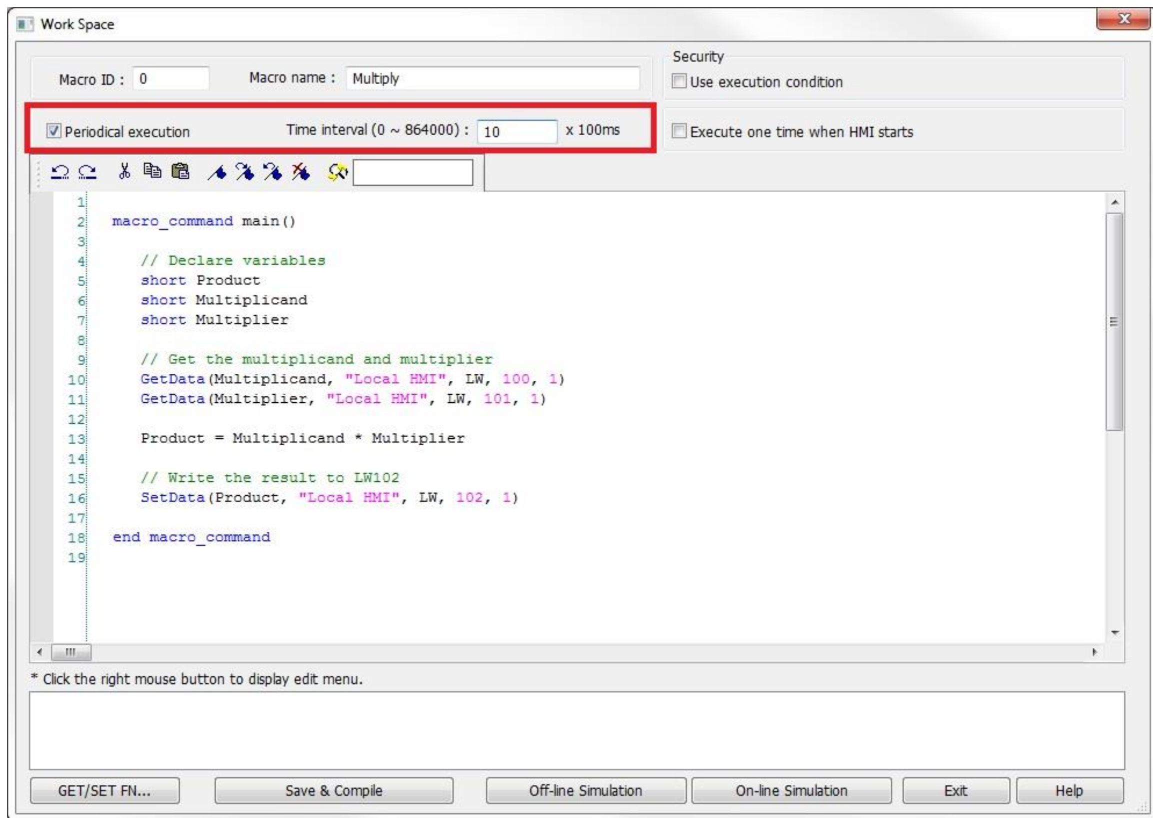
1. Select the Function Key object under the Objects > Button menu.
2. On the General tab of the object's Properties window, select the "Execute macro" option.



3. Select the macro you wish to execute. The macro will be executed one time when the button is activated.

Macro Work Space

1. In the Macro Work Space, select “Periodical Execution” to automatically trigger the macro on a periodic basis. The “Time interval” can be set in increments of 100 ms.



2. In the Macro Work Space, select “Execute one time when HMI starts” to run the macro once when the HMI is first turned on or rebooted.

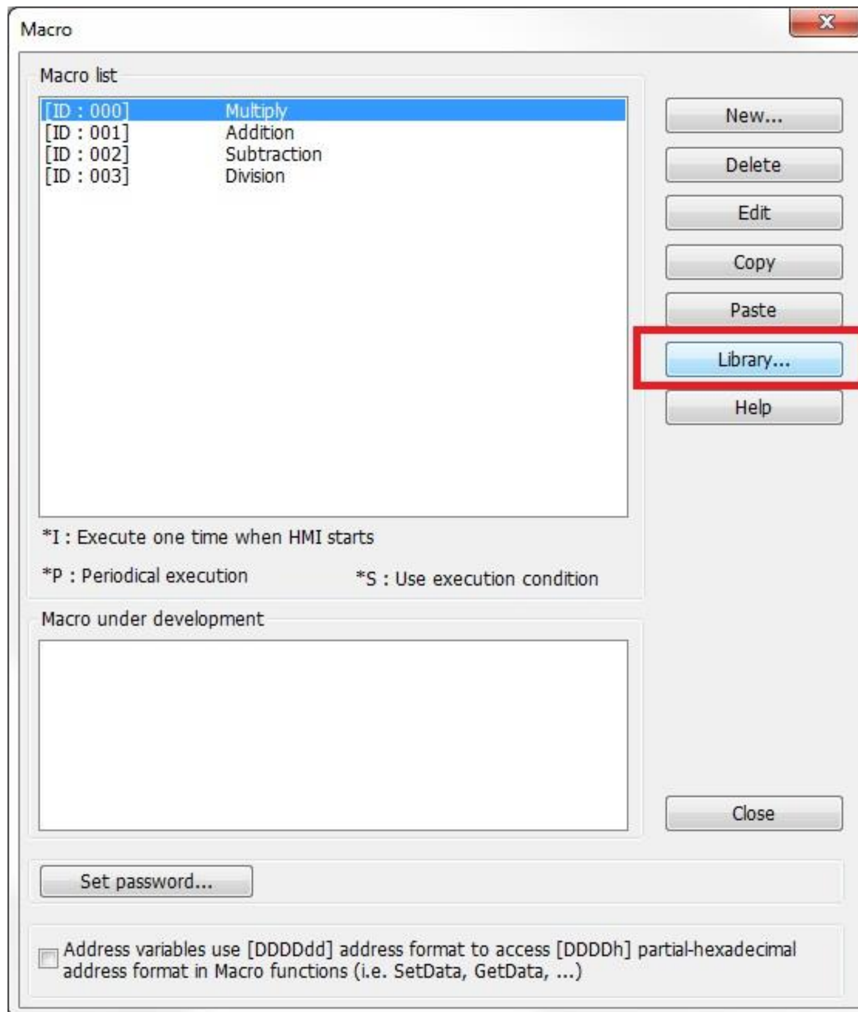
9. User-Defined Macro Functions

Up until this point, we have been referring to built-in macro functions. These functions are the most commonly used functions and can be used to implement most applications. However, there may be some combination of built-in functions that are frequently used from project to project and it would be advantageous to save them as a user-defined macro function. If this is the case, you can define a macro function yourself and save it in the Macro Function Library for future use.

Using the Macro Function Library

When you open a project in EZwarePlus, the default Macro Function Library ("MacroLibrary" without filename extension) is loaded in the Macro Function Library automatically. When you create user-defined macro functions, they are saved in a file with an *.mlb extension. These files can be imported and exported in the Macro Function Library and are stored in the EZPlus/library folder.

1. Open the Macro Manager (Tools menu > Macro) and click the "Library" button to open the Macro Function Library window.



2. A list of functions from the default MacroLibrary file will appear under the “Global Library” tab.

Macro Function Library

Project | Global Library

No.	Function Name
1	short Slav_hibyte (short)
2	short Slav_lobyte (short)
3	short Turkey_hibyte (short)
4	short Turkey_lobyte (short)
5	short CentralEuropean_hibyte (short)
6	short CentralEuropean_lobyte (short)

*The subroutine can be invoked by Conversion Tags or the Numeric object's Scaling feature under the following conditions:

1. The declaration of a subroutine is:
type_a func_name(type_b name), where type_b must be the same data format as the numeric data, for example, both data types are 16-bit Unsigned.

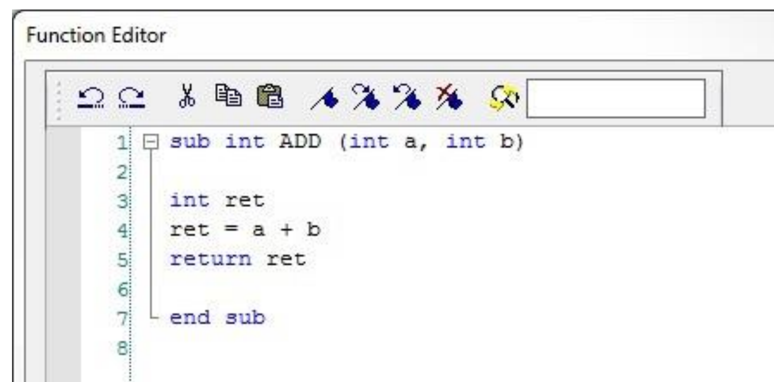
2. NOT read/write the non-HMI local address.

3. NOT call the following functions or commands : SYNC_TRIG_MACRO(), SYNC_TRIG_MACRO(), DELAY(), FindDataSamplingDate(), FindDataSamplingIndex(), FindEventLogDate(), FindEventLogIndex(), INPORT(), INPORT2(), INPORT3(), OUTPORT(), PURGE(), for, while.

New... Delete Edit... Export Import OK

1. Each function has the format:
return_type function_name (*parameter_type1*, …, parameter_typeN)

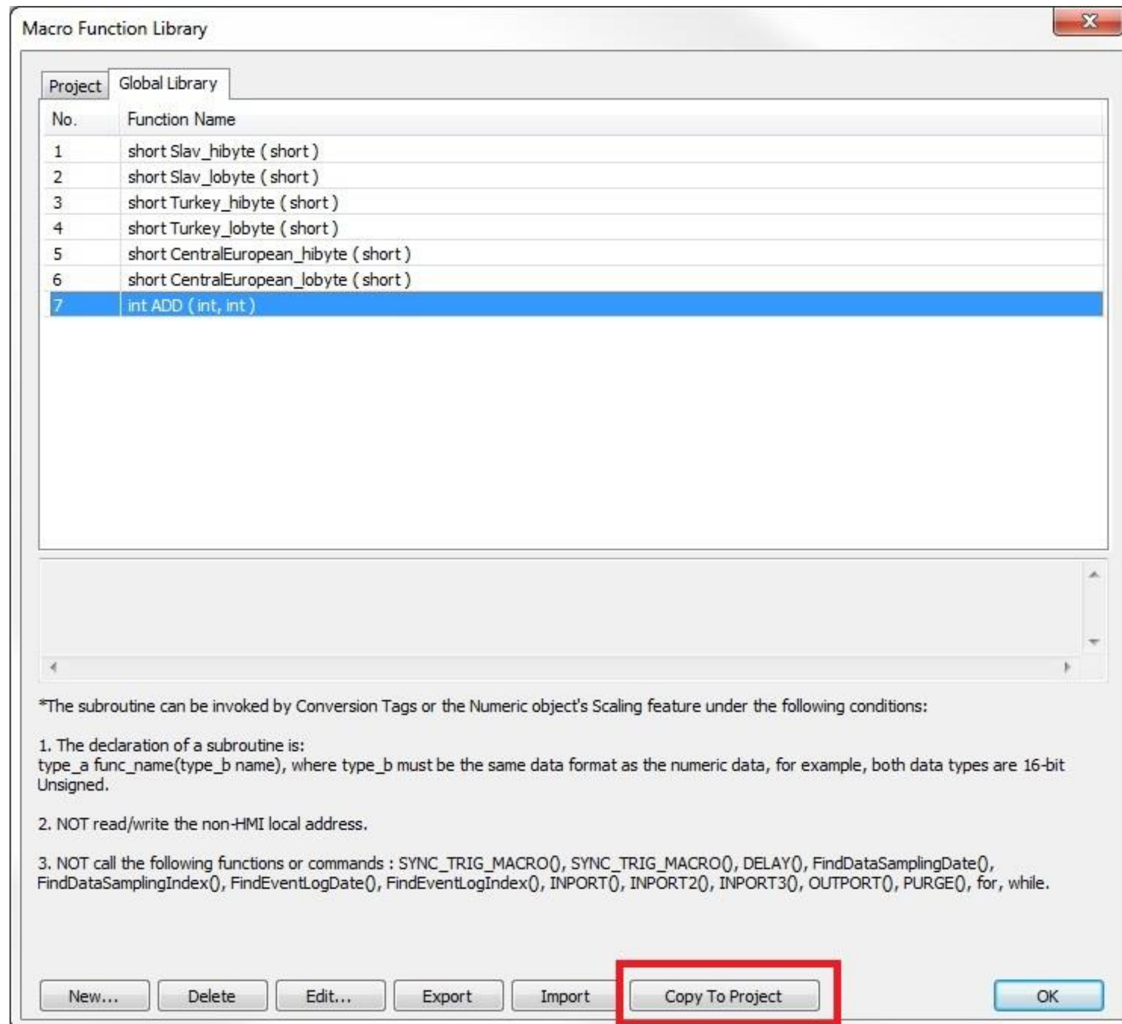
return_type indicates the type of the returned value. If this value does not exist, this column will be omitted. *function_name* indicates the name of the function. *parameter_type* indicates the type for the various parameters used. If the function does not use parameters, this column will be omitted. (N = number of parameters.)



For example, the following function has the Function Name “int ADD (int, int).”

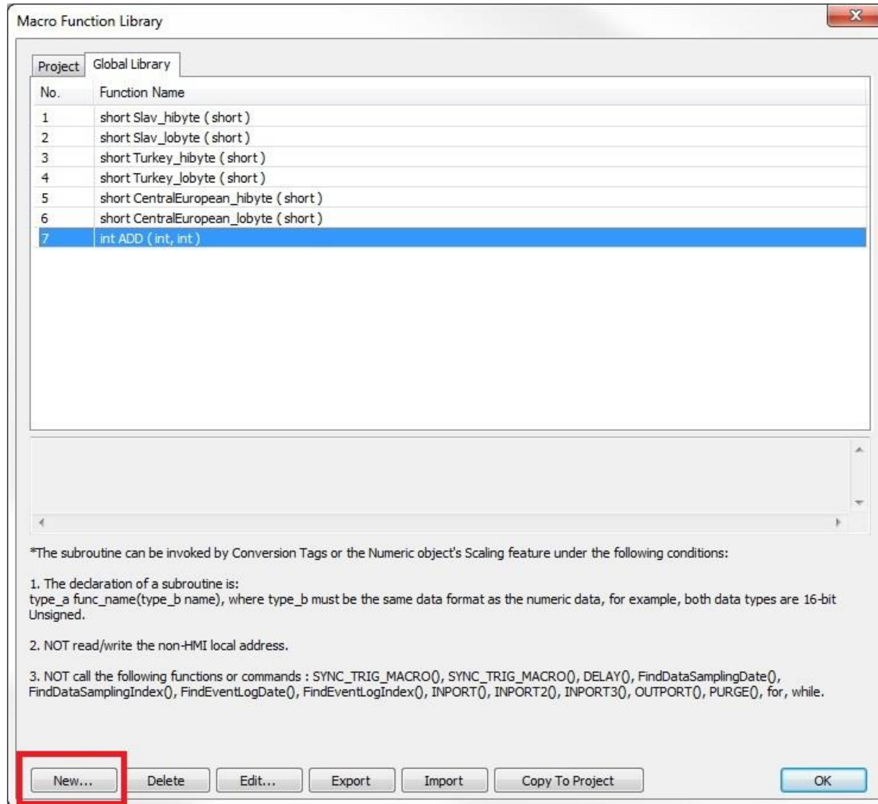
2. The Global Library functions are stored in either the MacroLibrary file or a user-defined *.mlb file in the EZPlus/library folder and are therefore linked to those files. You can embed a macro function in the project file so it no longer requires the link to the library file. Select the function in the Macro Function Library and click “Copy to Project.” This will copy the macro function to the Project tab.

When you open the project on another computer, the macro function can still be used. When compiling the project, the *.exob file will include the macro functions that are used.

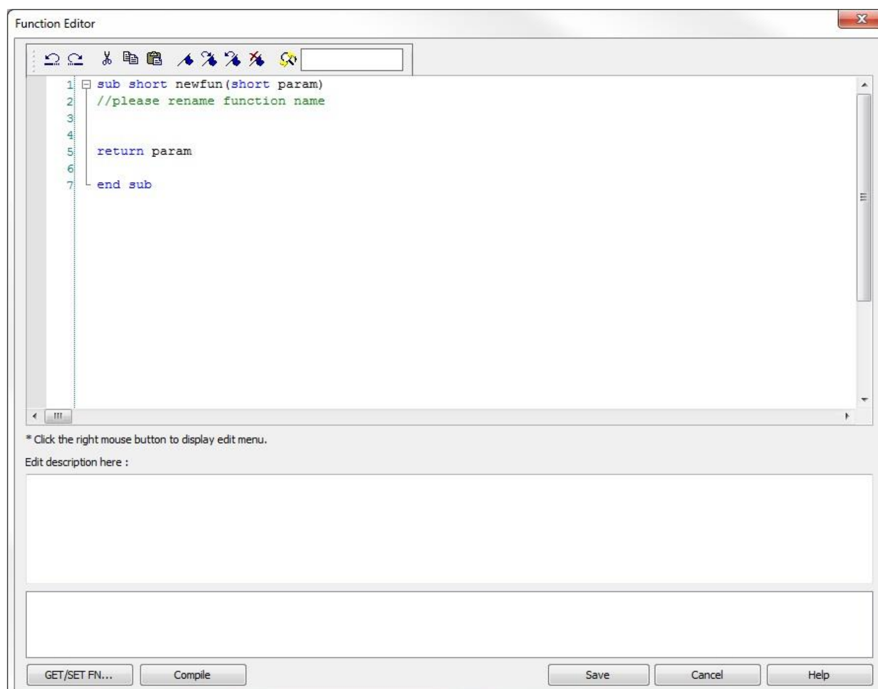


Creating a New Function

1. Open the Macro Function Library and click the “New” button.



2. This opens the Function Editor window. A template of the formatting required for the new macro function appears in the window.

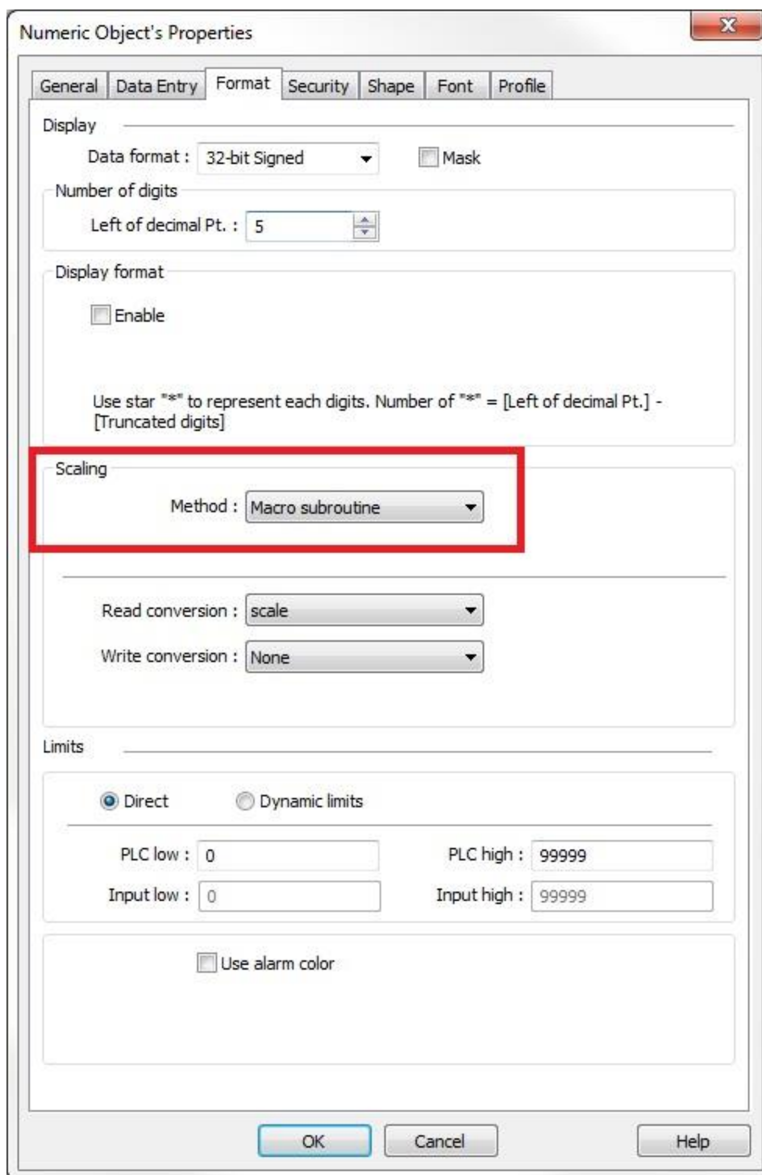


Edit the function according to the required format. You can use the description field at the bottom of the Function Editor to describe what the function does, how to use it, etc.

3. After editing the function, click “Compile.” After compiling successfully, click “Save” to save the function to the library.

Using a Macro Function in a Numeric Object

A macro function can be assigned to a Numeric object on the Numeric Format tab in the Numeric Object’s Properties window. Set the “Scaling Method” to “Macro subroutine.”



When “Allow input” is selected on the General tab, a macro function from the Macro Function Library can be assigned to the Read and Write actions of the Numeric object independently.

The Data format of the Numeric object must be the same as the data type declared in the macro function for the *return_type* and *parameter_type* in order for it to be selected. In the above example, the macro function “scale” uses the data type “int,” which is 32-bit signed, so the Data format for the Numeric object must be 32-bit signed.

Using a Macro Function in the Address Tag Library

A macro function can be assigned to a user-defined tag in the Address Tag Library window. Add a new tag or edit an existing tag by clicking the “Settings” button.

Enable the “Conversion/Calculation” option. A macro function from the Macro Function Library can be assigned to the Read and Write actions of the Address Tag independently. When the tag is assigned to an object, the macro function is applied to the read or write value that is displayed or written.

The Data format of the Address Tag and the Conversion/Calculation must be the same as the data type declared in the macro function for the *return_type* and *parameter_type* in order for it to be selected. In the above example, the macro function “scale” uses the data type “int,” which is 32-bit signed, so the Data format for the Address Tag and the Conversion/Calculation must be 32-bit signed.

Some Notes About Using Macros

1. The total size of the declared data types in a function cannot exceed 4096 bytes.

Data Type	Size
short	2 bytes (16-bit)
int	4 bytes (32-bit)
float	4 bytes (32-bit)
char	1 byte (8-bits)
bool	1 bit
arrays	The corresponding number of bytes for each element, plus another 2 bytes for the index

2. A Function Name must only contain alphanumeric characters and cannot start with a number.

3. A macro may cause the HMI to become unresponsive if:
   a. A macro contains an infinite loop with no PLC communication.
   b. The size of an array exceeds the storage space in a macro.

4. The PLC communication speed affects the running time for the macro.

5. Too many macros may slow down the communication between the HMI and PLC.

6. A maximum of 255 macros are allowed in an EZwarePlus project.

Deleting a Macro Function

1. Select the function to be deleted in the list of functions in the Macro Function Library.

2. Click “Delete.” Confirm by clicking “Yes” or cancel by clicking “No.”

Modifying a Macro Function

1. Select the function to modify in the list of functions in the Macro Function Library.

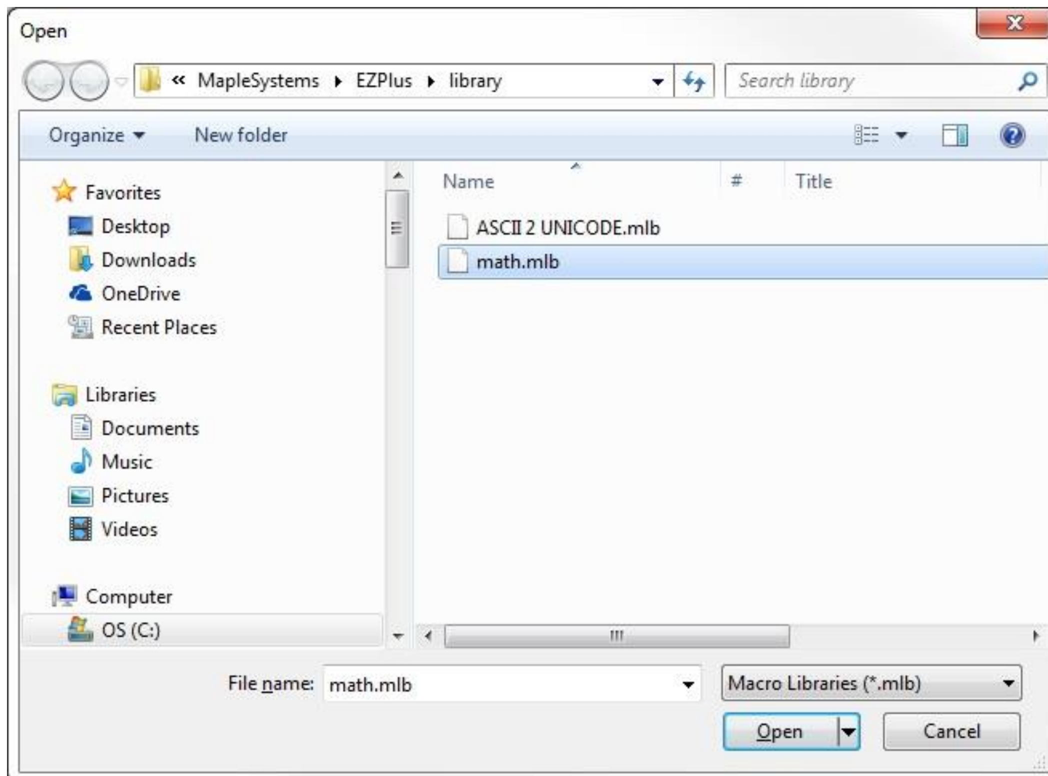
2. Click “Edit” to open the Function Editor (or double-click on the function).

3. Modify the function as desired, then click “Compile” and “Save.”

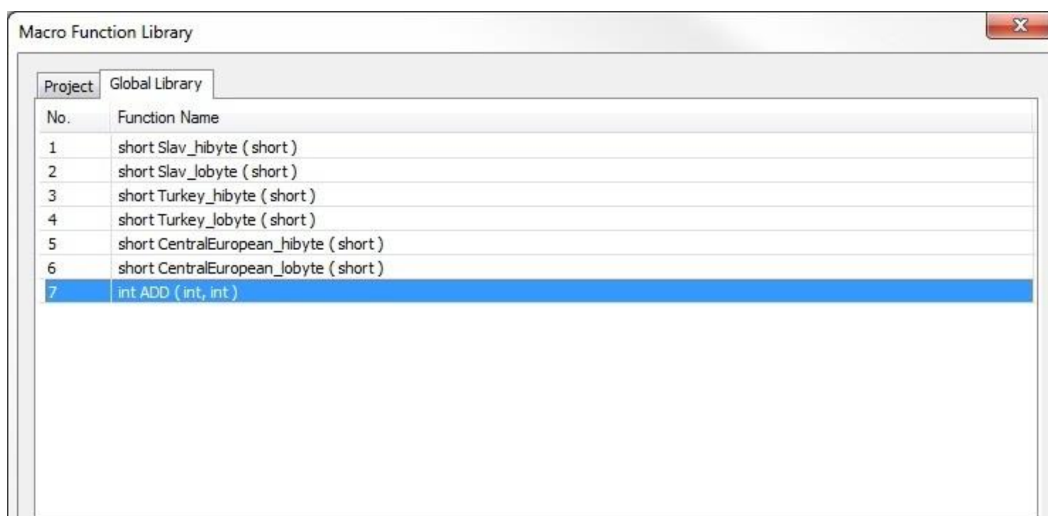
Importing a Macro Function

1. A function can be imported from an external *.mlb file. Click “Import” and select the *.mlb file you wish to import from the EZPlus/library folder.

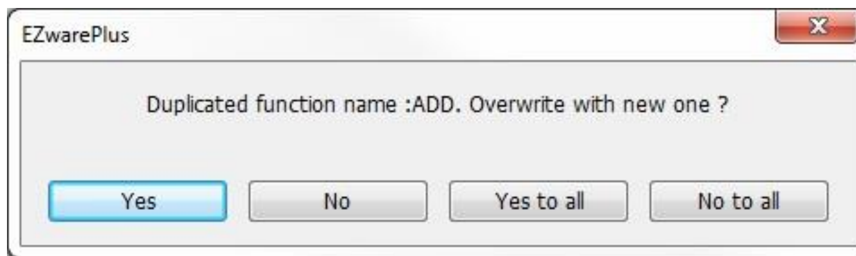
   For example, click “Import” and select a macro function called “math.mlb,” which contains a function “ADD.”



2. Click “Open” and the function ADD appears in the list of functions.



3. If a function already exists in the library, a confirmation window will pop-up.

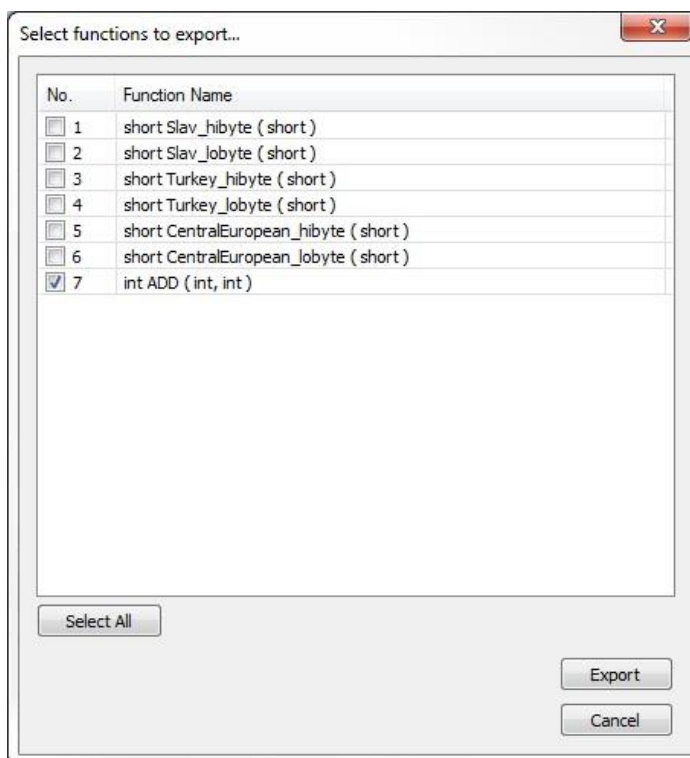


Click “Yes” to overwrite the existing function with the imported one.
Click “No” to cancel importing the function with the same name.
Click “Yes to all” to overwrite all the imported functions with the same name.
Click “No to all” to cancel importing all the functions with the same name.

4. The imported function(s) will be saved in the default MacroLibrary, so if the “math.mlb” file is deleted, the ADD function will still exist in the Macro Library.

Exporting a Macro Function

1. A function can be exported from the Macro Function Library and saved as a *.mlb file. Click “Export” and check boxes for the functions to be exported. Click the “Select All” button to select all the functions.



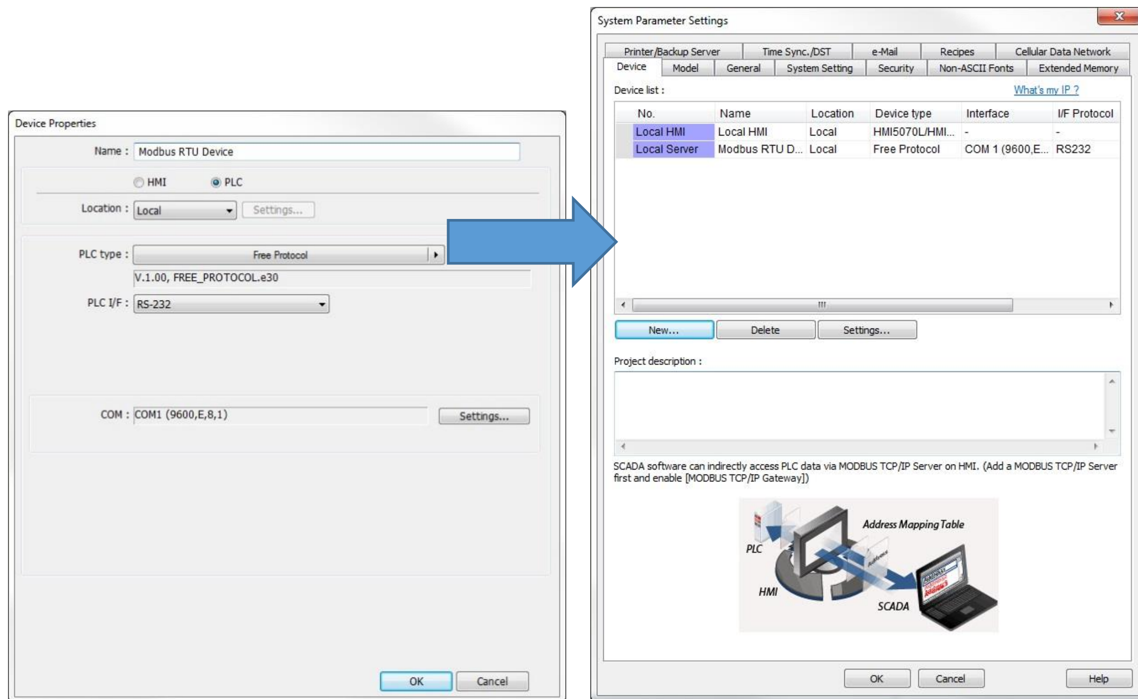
2. Click “Export” and the “Save As” window pops up. Navigate to the EZPlus/library folder and provide a name for the function(s) being exported, then click “Save.”

3. The exported *.mlb file can be imported in another EZwarePlus project or copied to another computer to use in other EZwarePlus projects.

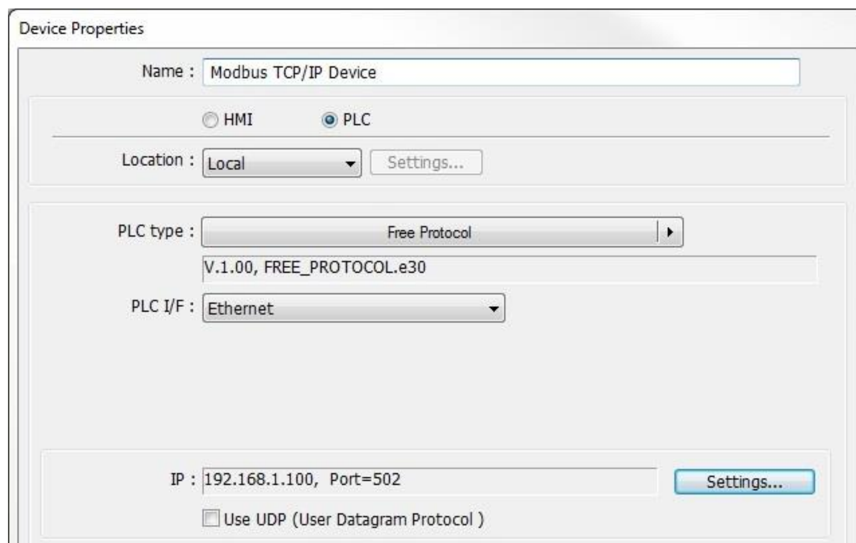
10. Using the Free Protocol to Control a Device

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:



The above example uses RS-232 for the PLC I/F (PLC interface). If the interface of the Modbus device uses Ethernet (Modbus TCP/IP), select Ethernet for the PLC I/F with the correct IP address and port number in the Device Properties:



2. Suppose you want the HMI to read the data from 4x-1 and 4x-2 on the device. First, use the OUTPORT function to send out a read request to the device. The format of OUTPORT is:
   OUTPORT(command[start], device_name, cmd_count)

   Since the device is a MODBUS RTU device, the read request must follow MODBUS RTU protocol. The request uses the "Reading Holding Registers (0x03)" command (function code) 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 is as follows (for a total of 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)

Therefore, a read request is formatted as follows:

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

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

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

address = 0   // starting address (4x-1) is 0
HIBYTE(address, command[2])
LOBYTE(address, command[3])

read_no = 2   // the total number or words read is two words
HIBYTE(read_no, command[4])
LOBYTE(read_no, command[5])

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

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

Lastly, use OUTPORT to send out this read request to the PLC:

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

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

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

The format of INPORT is:

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

…where the actual read count is written to the variable *return_value* (in bytes). If *return_value* is 0, it means the read command failed.

According to the MODBUS RTU protocol specification, the correct response[1] must be equal to 0x03 (function code 0x3). After getting the correct response, calculate the data of 4x-1 and 4x-2 and put the data into LW-100 and LW-101 of the 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)

end if
```

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

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

address = 0   // starting address (4x-1) is 0
HIBYTE(address, command[2])
LOBYTE(address, command[3])

read_no = 2   // the total number or words read is two words
HIBYTE(read_no, command[4])
LOBYTE(read_no, command[5])

CRC(command[0], checksum, 6)   // calculate the 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)” function code.

Request

Function code	1 byte	0x05
Output address	2 bytes	0x0000 to 0xFFFF
Output value	2 bytes	0x0000 to 0xFF00

Response

Function code	1 byte	0x05
Output address	2 bytes	0x0000 to 0xFFFF
Output value	2 bytes	0x0000 to 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] to command[31] to 0
FILL(response[0], 0, 32)

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

address = 0   // starting address (4x-1) is 0
HIBYTE(address, command[2])
LOBYTE(address, command[3])

command[4] = 0xff   // force 0x-1 ON
command[5] = 0

CRC(command[0], checksum, 6)   // calculate the 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

end macro_command
```

11. 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 possible causes for this compiler error

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

(C2) 'identifier' used without having been initialized
The macro must define the size of an array in the declaration/

Example:
```
macro_command main(0)
char i
int g[i]   // i must be a numeric constant
end macro_command
```

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

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

Example:
```
sub int if()   // error
```

(C5) parentheses have not come in pairs
Statement missing "(" or ")"

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 will cause a compiling 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 assignment 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 will cause a compiling 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 will cause a compiling error.

(C24) syntax error: 'break'
"break" statement can only be used in "for"or "while" statement.

(C25) syntax error: 'continue'
"continue" statement can only be used in "for" or "while" statement.

(C26) syntax error
Error in expression.

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

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" at the beginning and end of the macro.

(C29) must be keyword 'sub'
The format of the function declaration is:

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

Example:
```
sub int pow(int exp)
……..
end sub
```

Any format other than this will cause a compiling error.

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

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

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

(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 'indentifier'
Any variable or function should be declared before use.

(C38) unsupported PLC data address
The parameter of GetData() or SetData() should be a legal PLC address. If the address is illegal, this error message will be shown.

(C39) 'identifier' must be an integer, char, or constant
The format of an array is:

Declaration: array_name[constant] (constant is the size of the array)
Usage: array_name[integer, character, or constant]

Any form other than this will cause a compiling error.

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

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

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 the System Parameters.

(C51) macro_command does not control a remote device
A macro can only control a local device.

Example:
SetData(bits[0], “PLC 1”, LB, 123, 100)
“PLC 1” is connected with the remote HMI so it cannot work.

12. Sample Macro Code

for statement

“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
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

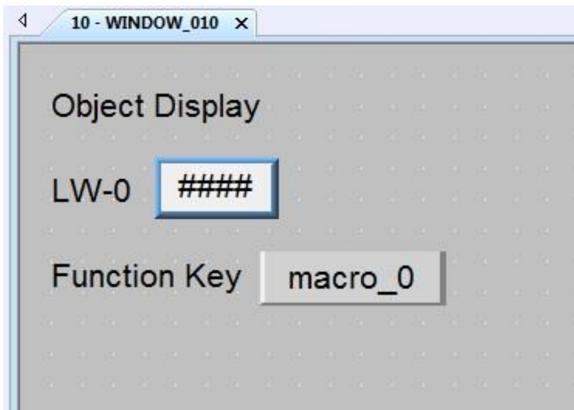
```
macro_command main()

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
```

13. Macro TRACE Function

The TRACE function can be used with EasyDiagnoser to show the current content of the variables. The following example illustrates how the TRACE function can be used in a macro.

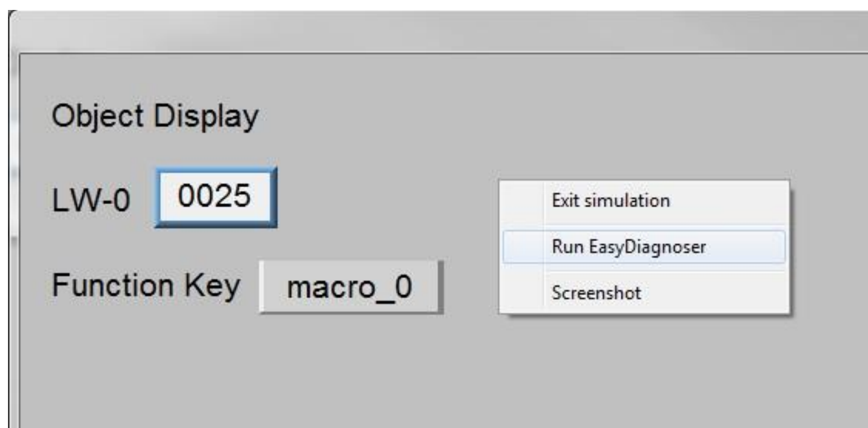
1. Create a new macro in the project.
   Example:
   ```
   macro_command main()

   short a
   GetData(a, "Local HMI", LW, 0, 1)
   a = a + 1
   SetData(a, "Local HMI", LW, 0, 1)
   TRACE("LW0 = %d", a)    // %d indicates that the value in LW-0 is in decimal format

   end macro_command
   ```

2. Add a Numeric object and Function Key object in Window 10 of the project. Configure the Function Key object to execute the macro.



3. Compile the project and start up Off-line Simulation or On-line Simulation.

4. Right-click on the simulator and select "Run EasyDiagnoser" in the popup menu.



5. EasyDiagnoser will start and the Logger window at the bottom of EasyDiagnoser will indicate "Connection established with the target HMI."

6. Click the Function Key (macro_0) in the simulator to execute the macro. The following will be written to the Output window in EasyDiagnoser with each consecutive execution of the macro:

```
Output
[ID 0, Ln 8] LW0 = 1
[ID 0, Ln 8] LW0 = 2
[ID 0, Ln 8] LW0 = 3
[ID 0, Ln 8] LW0 = 4
[ID 0, Ln 8] LW0 = 5
```

Note that the ID number of the macro and the line on which the TRACE function appears are also written to the Output window.

Trace Syntax

Name	TRACE
Syntax	TRACE(format, argument)
Description	Use this function to send the specified string to EasyDiagnoser. Use to 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 the output string. A format specification, which consists of optional (in[ ]) and required fields (in bold), has the following form: %[*flags*] [*width*] [.*precision*] *type* Each field of the format specification is described below: *flags* (optional): – + *width* (optional): A non-negative decimal integer controlling the minimum number of characters printed. *precision* (optional): A non-negative decimal integer that 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 E or e : signed value having 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 – . f : signed value having the form [-]dddd.dddd, where dddd is one or more decimal digits. The length of the output string is limited to 256 characters. The *argument* part is optional.

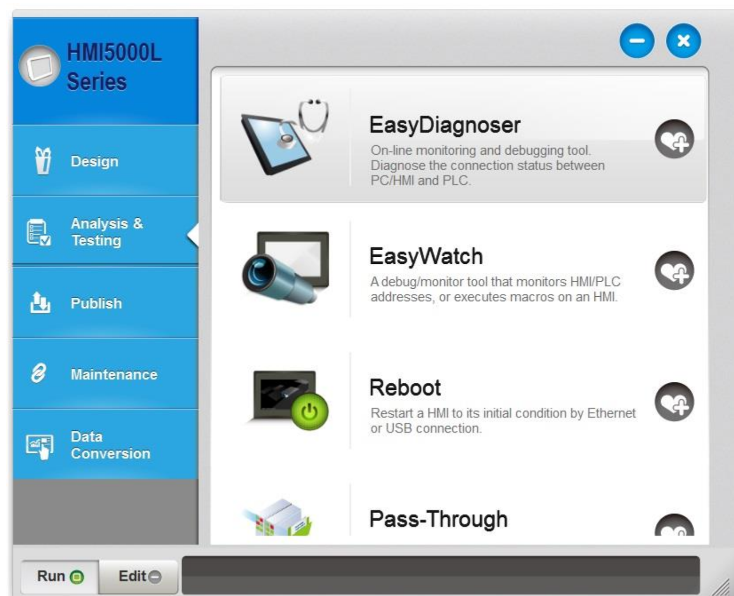
Example	macro_command main()
	
	char c1 = ‘ a’
	short s1 = 32767
	float f1 = 1.234567
	
	TRACE(“The results are”) // output: The results are
	TRACE(“c1 = %d, f1 = %df”, c1, s1, f1)
	// output: c1 = a, s1 = 32767, f1 = 1.234567
	
	end macro_command

Connecting EasyDiagnoser to an HMI

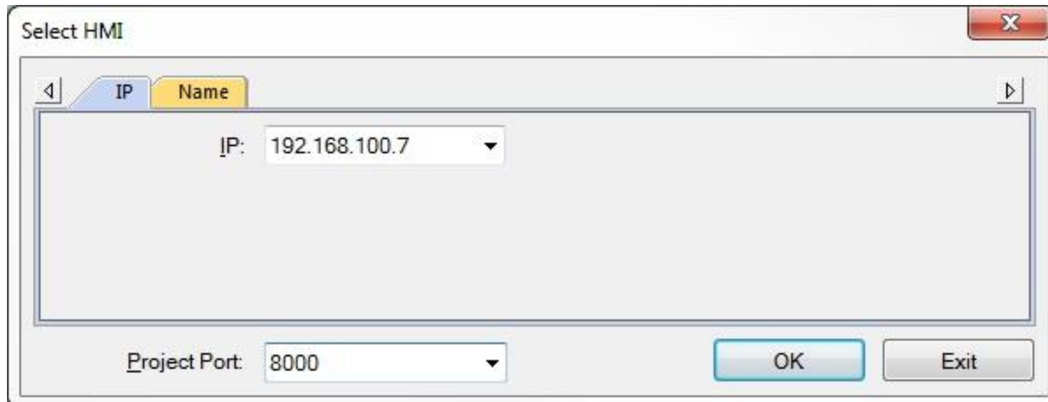
EasyDiagnoser can be launched from Utility Manager and connect to the HMI over Ethernet.

1. Download the project with the TRACE function to the HMI.

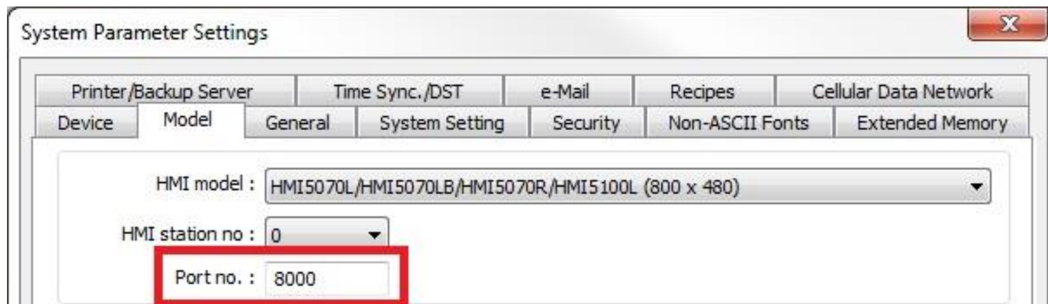
2. Start Utility Manager (Windows Start menu > All Programs > Maple Systems > EZwarePlus > Utility Manager). Click the “Analysis & Testing” tab and select “EasyDiagnoser.”



3. The “Select HMI” window pops up. Enter the IP address of the HMI and the Project Port number (8000 is the default).

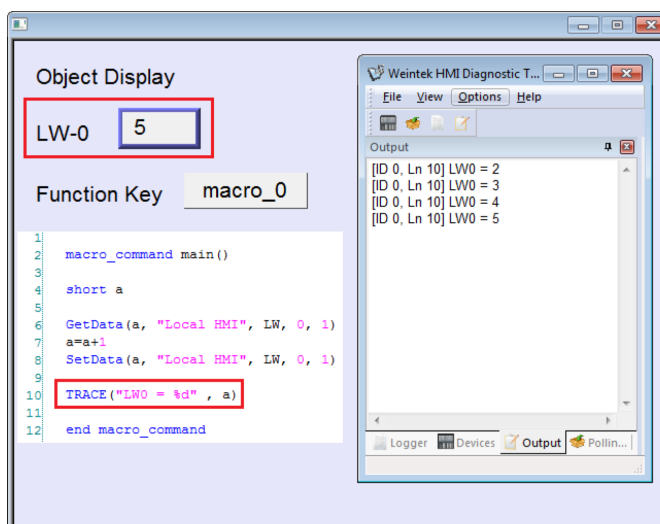


The Project Port number must match the Port no. in the EZwarePlus project’s System Parameters > Model tab.



7. Click “OK” and EasyDiagnoser will launch and establish a connection with the HMI (“Connection established with the target HMI” will appear in the Logger window).

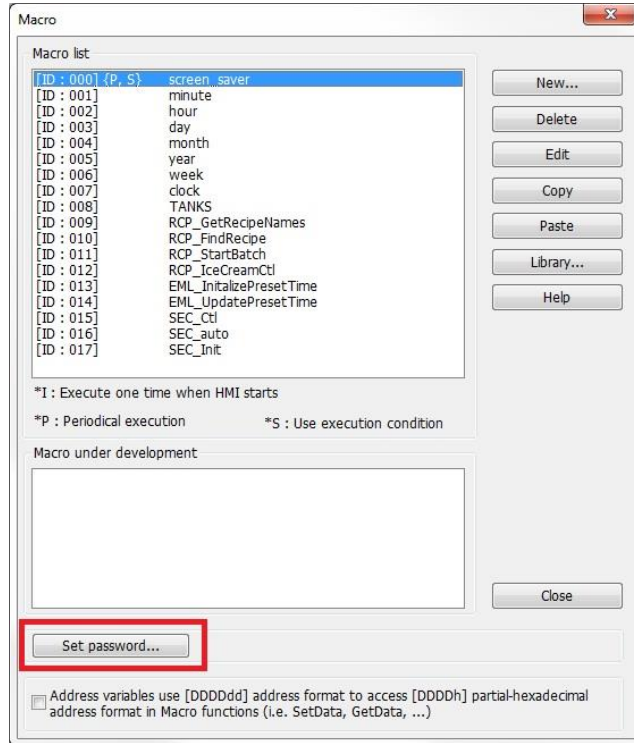
8. Execute the macro in the HMI and the Output window in EasyDiagnoser will display the specified string from the TRACE function.



14. Macro Password Protection

The macros in a project can be password protected to prevent anyone from opening or editing a macro in the Macro list.

1. Click the “Set password” button in the Macro list.



2. In the Password popup window, select the “Password protect” option and enter the password (the default password is 111111). The password can be up to 10 characters long using alphanumeric characters.

3. Once the password is set, the correct password must be entered in order to access the Macro list.
   If the incorrect password is entered three times, you must close EZwarePlus and reopen it in order to try again.

When the macros are password protected, decompiling an EXOB file will not restore the macro contents. The macro contents will be erased.

1010-1046 Rev. 00

Your Industrial Control Solutions Source

www.maplesystems.com

Maple Systems, Inc. | 808 134th St. SW, Suite 120, Everett, WA 98204 | 425.745.3229