Weintek USA, Inc.
www.WeintekUSA.com
(425) 488-1100

User Manual

Macro Reference Guide

Rev. MAR 08, 2021

EasyBuilder Pro v6.05.02 or greater has many built-in functions for data type conversion, receiving and transferring data to a PLC, and mathematical functions.

Table of Contents

Chapter 1. PLC Functions

GetData ()

Description:
Receives data from the HMI memory or an external device.

Syntax:
GetData(*read_data[starting], device_name, address_type, address, data_count*)
or
GetData(*read_data, device_name, address_type, address, 1*)

Argument	Description
read_data[starting]	*read_data* is an array.The data is stored to *read_data[starting]* to *read_data[starting+data_count-1].*
device_name	The PLC name enclosed in the double quotation marks (“) and this name has been defined in the device list of the System Parameters.
address_type	The register type where the data is stored in the PLC.
address	The starting address.
data_count	The amount of data read.

Example:

```
char byData[10]
short wData[6]
int dwData[5]

GetData(byData[0], "Local HMI", LW, 0, 10) //  reads 10 bytes data (= 5 words)
GetData(wData[0], "Local HMI", LW, 0, 6)   //  reads 6 words data
GetData(dwData[0], "Local HMI", LW, 0, 5)   //  reads 5 double-words data (= 10 words)
GetData(wData[0], "Local HMI", "Pressure", 6)  // uses user-defined tag - "Pressure" to indicate device type and address.
```

GetDataEx ()

Description:
Receives data from the HMI memory or an external device. The macro will move on to the next line even if there is no response from the PLC.
Descriptions of *read_data*, *device_name, address_type, address*, and *data_count* are the same as the GetData function.

Syntax:
GetDataEx(*read_data[starting], device_name, address_type, address, data_count*)
or
GetDataEx(*read_data, device_name, address_type, address, 1*)

Argument	**Description**
read_data[starting]	*read_data* is an array.The data is stored to *read_data[starting]* to *read_data[starting+data_count-1].*
device_name	The PLC name enclosed in the double quotation marks (“) and this name has been defined in the device list of the System Parameters.
address_type	The register type where the data is stored in the PLC.
address	The starting address.
data_count	The amount of data read.

Example:

```
char byData[10]
short wData[6]
int dwData[5]

GetDataEx (byData[0], "Local HMI", LW, 0, 10) //  reads 10 bytes data (= 5 words)
GetDataEx (wData[0], "Local HMI", LW, 0, 6)   //  reads 6 words data
GetDataEx (dwData[0], "Local HMI", LW, 0, 5)   //  reads 5 double-words data (= 10 words)
GetDataEx (wData[0], "Local HMI", "Pressure", 6)  // uses user-defined tag - "Pressure" to indicate device type and address.
```

SetData ()

Description:
Sends data to the HMI memory or an external device.

Syntax:
SetData(*send_data[start], device_name, address_type, address, data_count*)
or
SetData(*send_data, device_name, address_type, address, 1*)

Argument	Description
send _data[starting]	*send_data* is an array.The data is defined in *send_data[starting]* to *send _data[starting+data_count-1].*
device_name	The PLC name enclosed in the double quotation marks (“) and this name has been defined in the device list of the System Parameters.
address_type	The register type where the data is stored in the PLC.
address	The starting address.
data_count	The amount of data written.

Example:

```
char byData[10]
short wData[6]

FILL(byData[0], 0, 10)//  populates each member of byData[] with 0, byData[0]=0, byData[1]=0, and so on.
FILL(wData[0], 0, 6)

SetData(byData[0], "Local HMI", LW, 0, 10)//  sends 10 bytes data (= 5 words)
SetData(wData[0], "Local HMI", LW, 0, 6)//  sends 6 words data
SetData(wData[0], "Local HMI", "Pressure", 6 //  use user-defined tag - "Pressure" to indicate device type and address.
```

SetDataEx ()

Description:
Sends data to the HMI memory or an external device. The macro will move on to the next line even if there is no response from the PLC.

Syntax:
SetDataEx(*send_data[start], device_name, address_type, address, data_count*)
or
SetDataEx(*send_data, device_name, address_type, address, 1*)

Argument	Description
send _data[starting]	*send_data* is an array.The data is defined in *send_data[starting]* to *send _data[starting+data_count-1].*
device_name	The PLC name enclosed in the double quotation marks (“) and this name has been defined in the device list of the System Parameters.
address_type	The register type where the data is stored in the PLC.
address	The starting address.
data_count	The amount of data written.

Example:
```
char byData[10]
short wData[6]

FILL(byData[0], 0, 10) //  populates each member of byData[] with 0, byData[0]=0, byData[1]=0, and so on.
FILL(wData[0], 0, 6)

SetDataEx(byData[0], "Local HMI", LW, 0, 10)//  sends 10 bytes data (= 5 words)
SetDataEx(wData[0], "Local HMI", LW, 0, 6)//  sends 6 words data
SetDataEx(wData[0], "Local HMI", "Pressure", 6 //  use user-defined tag - "Pressure" to indicate device type and address.
```

GetError ()

Description:
Gets an error code.

Syntax:
GetError(*err*)

Argument	Description
err	This function saves an error code to this variable.

Example:

```
short err
char byData[10]

GetDataEx(byData[0], "MODBUS RTU", 4x, 1, 10)//  reads 10 bytes = 5 words
                                              //  Must use GetError() to check whether GetDataEx() succeeds or not before
                                              using byData[].

GetError(err)     //  saves an error code to err

if err == 0 then  //  if err is equal to 0, it succeeded in executing GetDataEx() and byData[] has data.

Setdata(byData[0], “Local HMI”, LW, 100, 10) // Display valid data on the HMI.

end if
```

Chapter 2. Free Protocol Functions

GetCTS ()

Description:
Gets CTS state for RS232.

Syntax:
GetCTS(*com_port, result*)

Argument	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.
result	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:

```
char com_port = 3
char result
GetCTS(com_port, result) // gets CTS signal of  COM3
GetCTS(1, result) // get CTS signal of COM 1
```

SetRTS ()

Description:
Raises or lowers the RTS signal of RS-232.

Syntax:
SetRTS(*com_port, source*)

Argument	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.
source	used for raising or lowering the RTS signal.

Example:
char com_port = 1
char value = 1
SetRTS(com_port, value) // raises RTS signal of COM 1
SetRTS(1, 0) // lowers RTS signal of COM 1

Inport ()

Description:
Reads data from a COM port or Ethernet port.

Syntax:
INPORT(*read_data[starting], device_name, read_count, return_value*)

Argument	Description
read_data[starting]	The data is stored to *read_data[starting]* to *read_data[starting+(read_count-1)]*.
device_name	The name of the device defined in the Device list and the device must be a "Free Protocol" –type device.
read_count	The required amount of data read and can be a constant or a variable.
receive_len	The length of data received (unit : byte).

Example:

```
char wResponse[6]
short receive_len

INPORT(wResponse[0], "Free Protocol", 6, receive_len)//  read 6 bytes

if receive_len >= 6 then

SetData(wResponse[0], "Local HMI", LW, 0, 6)//  transfer the data to LW0

end if
```

INPORT2 ()

Description:
Reads data from a COM port or Ethernet port and then pause until the designated time.

Syntax:
INPORT2(*response[starting], device_name, receive_len, wait_time*)

Argument	Description
response[starting]	The data is stored to *response [starting]* to *response [starting+(read_count-1)]*.
device_name	The name of the device defined in the Device list and the device must be a "Free Protocol" –type device
receive_len	The length of the data received and must be a variable. The total length cannot exceed the size of response (unit : byte).
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:

```
char wResponse[6]
short receive_len, wait_time=20

INPORT2(wResponse[0], "Free Protocol", receive_len, wait_time)

if receive_len >= 6 then

SetData(wResponse[0], "Local HMI", LW, 0, 6)

end if
```

INPORT3 ()

Description:
Reads data from a COM port or Ethernet port according to the specified data size.

Syntax:
INPORT3(*response[starting], device_name, read_count, receive_len*)

Argument	Description
response[starting]	The data is stored to *response [starting]* to *response [starting+(read_count-1)]*. 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	The name of the device defined in the Device list and the device must be a "Free Protocol" –type device
read_count	The length of the data read each time.
receive_len	The length of the data received and must be a variable. The total length cannot exceed the size of response. (unit : byte)

Example:

```
char wResponse[6]
short receive_len

INPORT3(wResponse[0], "Free Protocol", 6, receive_len)

if receive_len >= 6 then

    SetData(wResponse[0], "Local HMI", LW, 0, 6)

 end if
```

INPORT4 ()

Description:
Reads data from a COM port or Ethernet port as far as the ending character is reached.

Syntax:
INPORT4(*response[starting], device_name, receive_len, tail_ascii*)

Argument	Description
response[starting]	The data is stored to *response [starting]* to *response [starting+(read_count-1)]*.
device_name	The name of the device defined in the Device list and the device must be a "Free Protocol" –type device
receive_len	The length of the data received and must be a variable. The total length cannot exceed the size of response. (unit : byte)
tail_ascii	Specifies the ending character. Data reading will stop when the ending character is reached.

Example:

```
char tail_ascii = 0x03      // 0x03== ETX
char wResponse[1024]
short receive_len

INPORT4(wResponse[0], "Free Protocol", receive_len, tail_ascii)

INPORT4(wResponse[0], "Free Protocol", receive_len, 0x0d)       // 0x0d == CR

if receive_len >= 6 then

SetData(wResponse[0], "Local HMI", LW, 0, 6)

end if
```

OUTPORT ()

Description:
Sends out the specified data to a PLC or controller via a COM port or Ethernet port.

Syntax:
OUTPORT(*source[starting], device_name, data_count*)

Argument	Description
source[starting]	This function sends out the specified data from *source[starting]* to *source[starting+(data_count-1)]* to the PLC
device_name	The name of a device defined in the device table and the device must be a "Free Protocol" –type device.
data_count	The amount of sent data and can be a constant or a variable. (unit : byte)

Example:

```
char byCommand[32]

FILL(byCommand[0], 0, 32)//  set buffers to a specified value

OUTPORT(byCommand[0], "Free Protocol", 32)//  send 32 bytes
```

PURGE ()

Description:
Clears the input and output buffers associated with the COM port.

Syntax:
PURGE(*com_port*)

Argument	Description
com_port	The COM port number on the HMI, which ranges from 1 to 3. It can be either a variable or a constant.

Example:
short com_port = 3
PURGE(com_port) // purge COM port 3
PURGE(1) // purge COM port 1

Chapter 3. Process Control Functions

ASYNC_TRIG_MACRO ()

Description:
Triggers the execution of a macro asynchronously in a running macro.

Syntax:
ASYNC_TRIG_MACRO (*macro_id*)
or
ASYNC_TRIG_MACRO (*macro_name*)

Argument	Description
macro_id	This function triggers the designated macro via **macro_id** or **macro_name**. macro_id can be a constant or a variable. The current macro will continue executing the following instructions after triggering the designated macro; in other words, the two macros will be active simultaneously.

Example:

```
bool ON = 1, OFF = 0

SetData(ON, "Local HMI", LB, 0, 1)

ASYNC_TRIG_MACRO(5)// call a macro (its ID is 5)
ASYNC_TRIG_MACRO("macro_1") // call a macro (its name is macro_1)

SetData(OFF, "Local HMI", LB, 0, 1)
```

SYNC_TRIG_MACRO ()

Description:
Triggers the execution of a macro synchronously in a running macro. The current macro will pause until the end of execution of this called macro.

Syntax:
SYNC_TRIG_MACRO (*macro_id*)
or
SYNC_TRIG_MACRO (*macro_name*)

Argument	Description
macro_id	This function triggers the designated macro via **macro_id** or **macro_name**. *macro_id* can be a constant or a variable. The current macro will pause until the end of execution of this called macro.

Example:
bool 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)

DELAY ()

Description:
Suspends the execution of the current macro for at least the specified time interval.

Syntax:
DELAY(*time*)

Argument	Description
time	The unit of *time* is milliseconds. *Time* can be a constant or a variable, and *time* can be up to 2147483647. Suspends the execution of the current macro for at least the specified *time*.

Example:
short time =500

DELAY(100)// delay 100 ms
DELAY(time)// delay 500 ms

Chapter 4. Data Operation Functions

FILL ()

Description:
Sets array elements to the specified value.

Syntax:
FILL(*source[starting], preset, count*)

Argument	Description
source[starting]	This function sets elements of an array (*source*) to a specified value (*preset*).
preset	A specified value. It can be a constant or a variable.
count	The amount of elements

Example:

1.
char byCommand[32]
FILL(byCommand[0], 0, 32)// set elements to 0

2.
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

SWAPB ()

Description:
Exchanges the high-byte and low-byte data of a 16-bit (Word).

Syntax:
SWAPB(*source, result*)

Argument	Description
source	This function exchanges the high-byte and low-byte data of a 16-bit *source* and saves it into *result*. *source* can be a constant or a variable.
result	*result* must be a variable.

Example:
short source = 0x1234, result

SWAPB(source, result)// result == 0x3412

SWAPB(0x12345678, result)// result == 0x34127856

SWAPW ()

Description:
Exchanges the high-word and low-word data of a 32-bit (DINT).

Syntax:
SWAPW(*source, result*)

Argument	Description
source	This function exchanges the high-word and low-word data of a 32-bit *source* and saves it into *result*. *source* can be a constant or a variable.
result	*result* must be a variable.

Example:
int source, result

SWAPW(0x12345678, result) // result is 0x56781234
source = 0x12345
SWAPW(source, result) // result is 0x23450001

LOBYTE ()

Description:
Retrieves the low byte of a 16-bit source.

Syntax:
LOBYTE(*source, result*)

Argument	Description
source	This function retrieves the low-byte of a 16-bit *source* and saves it into *result*. *source* can be a constant or a variable.
result	*result* must be a variable.

Example:
short source, result

LOBYTE(0x1234, result) // result is 0x34
source = 0x123
LOBYTE(source, result) // result is 0x23

HIBYTE ()

Description:
Retrieves the high byte of a 16-bit source.

Syntax:
HIBYTE(*source, result*)

Argument	Description
source	This function retrieves the high-byte of a 16-bit *source* and save it into *result*. *source* can be a constant or a variable.
result	*result* must be a variable.

Example:
short source, result

HIBYTE(0x1234, result)// result is 0x12
source = 0x123
HIBYTE(source, result)// result is 0x01

LOWORD ()

Description:
Retrieves the low word of a 32-bit source.

Syntax:
LOWORD(*source, result*)

Argument	Description
source	This function retrieves the low word of a 32-bit *source* and saves it into result. *source* can be a constant or a variable.
result	*result* must be a variable.

Example:
int source, result

LOWORD(0x12345678, result)// result is 0x5678
source = 0x12345
LOWORD(source, result)// result is 0x2345

HIWORD ()

Description:
Retrieves the high word of a 32-bit source.

Syntax:
HIWORD(*source, result*)

Argument	Description
source	This function retrieves the high word of a 32-bit *source* and saves it into result. *source* can be a constant or a variable.
result	*result* must be a variable.

Example:
int source, result

HIWORD(0x12345678, result)// result is 0x1234
source = 0x12345
HIWORD(source, result)// result is 0x0001

INVBIT ()

Description:
Inverts the state of designated bit position of a data source.

Syntax:
INVBIT(*source, result, bit_pos*)

Argument	Description
source	This function inverts the state of the designated bit position (*bit_pos*) of a word (*source*) and saves it into *result*. *source* can be a constant or a variable.
result	*result* must be a variable.
bit_pos	*bit_pos* can be a constant or a variable.

Example:
short bit_pos

INVBIT(4, result, 1)// result = 6

source = 6
bit_pos = 1
INVBIT(source, result, bit_pos)// result = 4

SETBITON ()

Description:
Changes the state of designated bit position of a data source to ON.

Syntax:
SETBITON(*source, result, bit_pos*)

Argument	Description
source	This function changes the state of the designated bit position (*bit_pos*) of a word (*source*) to 1 and saves it into *result*. *source* can be a constant or a variable.
result	*result* must be a variable.
bit_pos	*bit_pos* can be a constant or a variable.

Example:

```
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
```

SETBITOFF ()

Description:
Changes the state of designated bit position of a data source to OFF.

Syntax:
SETBITOFF(*source, result, bit_pos*)

Argument	Description
source	This function changes the state of the designated bit position (*bit_pos*) of a word (*source*) to 0 and saves it into *result*. *source* can be a constant or a variable.
result	*result* must be a variable.
bit_pos	*bit_pos* can be a constant or a variable.

Example:
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

GETBIT ()

Description:
Gets the state of designated bit position of a data source.

Syntax:
GETBIT(*source, result, bit_pos*)

Argument	Description
source	This function gets the state of the designated bit position (*bit_pos*) of a word (*source*) and saves it into *result*. r*esult* value will be 0 or 1. *source* can be a constant or a variable.
result	*result* must be a variable.
bit_pos	*bit_pos* can be a constant or a variable.

Example:
short bit_pos

GETBIT(9, result, 3) // result is 1

source = 4
bit_pos = 2
GETBIT(source, result, bit_pos) // result is 1

Chapter 5. Data Type Conversion Functions

ASCII2DEC ()

Description:
Converts an ASCII string to a decimal value.

Syntax:
ASCII2DEC(*source[starting], result, len*)

Argument	Description
source[starting]	This function transforms a string (*source*) into a decimal value and saves it to a variable (*result*). *source[starting]* represents the first character of the string. *source* can be a constant or a variable.
result	*result* must be a variable.
len	The length of the string. *len* can be a constant or a variable.

Example:
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, a decimal value.

ASCII2FLOAT ()

Description:
Converts an ASCII string to a float value.

Syntax:
ASCII2FLOAT(*source[starting], result, len*)

Argument	Description
source[starting]	This function transforms a string (*source*) into a floating point value and saves it to a variable (*result*). *source[starting]* represents the first character of the string. *source* can be a constant or a variable.
result	*result* must be a variable.
len	The length of the string. *len* can be a constant or a variable.

Example:
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, a floating point value.

ASCII2HEX ()

Description:
Converts an ASCII string to a hexadecimal value.

Syntax:
ASCII2HEX(*source[starting], result, len*)

Argument	Description
source[starting]	This function transforms a string (*source*) into a hexadecimal value and saves it to a variable (*result*). *source[starting]* represents the first character of the string. *source* can be a constant or a variable.
result	*result* must be a variable.
len	The length of the string. *len* can be a constant or a variable.

Example:
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, a hexadecimal value.

BIN2BCD ()

Description:
Converts a binary-type value to a BCD-type value.

Syntax:
BIN2BCD(*source, result*)

Argument	Description
source	This function transforms a binary-type value (*source*) into a BCD-type value and saves it to a variable (*result*). *source* can be a constant or a variable.
result	*result* must be a variable.

Example:
short source, result

BIN2BCD(1234, result)// result is 0x1234

source = 5678
BIN2BCD(source, result)// result is 0x5678

BCD2BIN ()

Description:
Converts a BCD-type value to a binary-type value.

Syntax:
BCD2BIN(*source, result*)

Argument	Description
source	This function transforms a BCD-type value (*source*) into a binary-type value and saves it into a variable (*result*). *source* can be a constant or a variable.
result	*result* must be a variable.

Example:
short source, result

BCD2BIN(0x1234, result)// result is 1234

source = 0x5678
BCD2BIN(source, result)// result is 5678

DEC2ASCII ()

Description:
Converts a decimal value to an ASCII string.

Syntax:
DEC2ASCII(*source, result[starting], len*)

Argument	Description
source	This function transforms a decimal value (*source*) into an ASCII string and saves it to an array (*result*). *source* can be a constant or a variable.
result[starting]	The first character is put into *result[starting]*, the second character is put into *result[starting+1]*, and the last character is put into *result[starting+(len-1)]*. *result* must be a variable.
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. *len* can be a constant or a variable.

Example:

```
short source
char result1[4]
short result2[4]
char result3[6]
source = 5678

DEC2ASCII(source, result1[0], 4)
// result1[0] is '5', result1[1] is '6', result1[2] is '7', result1[3] is '8'
// the length of the string (result1) is 4 bytes( = 1 * 4)

DEC2ASCII(source, result2[0], 4)
// result2[0] is '5', result2[1] is '6', result2[2] is '7', result2[3] is '8'
// the length of the string (result2) is 8 bytes( = 2 * 4)

source=-123
DEC2ASCII(source, result3[0], 6)
// result1[0] is '-', result1[1] is '0', result1[2] is '0', result1[3] is '1'
// result1[4] is '2', result1[5] is '3'
// the length of the string (result1) is 6 bytes( = 1 * 6)
```

FLOAT2ASCII ()

Description:
Converts a floating value to an ASCII string.

Syntax:
FLOAT2ASCII(*source, result[starting], len*)

Argument	Description
source	This function transforms a floating value (*source*) into an ASCII string and saves it to an array (*result*). *source* can be a constant or a variable.
result[starting]	The first character is put into *result[starting]*, the second character is put into *result[starting+1]*, and the last character is put into *result[starting+(len-1)]*. *result* must be a variable.
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. *len* can be a constant or a variable.

Example:
```
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'
```

HEX2ASCII ()

Description:
Converts a hexadecimal value to an ASCII string.

Syntax:
HEX2ASCII(*source, result[starting], len*)

Argument	Description
source	This function transforms a hexadecimal value (*source*) into an ASCII string and saves it to an array (*result*). *source* can be a constant or a variable.
result[starting]	The first character is put into *result[starting]*, the second character is put into *result[starting+1]*, and the last character is put into *result[starting+(len-1)]*. *result* must be a variable.
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. *len* can be a constant or a variable.

Example:
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)

StringDecAsc2Bin ()

Description:
Converts a decimal string to an integer.

Syntax:
success = StringDecAsc2Bin(*source[starting], destination*)
or
success = StringDecAsc2Bin(*"source", destination*)

Argument	Description
source[starting]	This function converts a decimal string to binary data. It converts the decimal string in the *source* parameter into binary data and saves it into a variable (*destination*). The *source* string parameter accepts both a static string (in the form: "source") and a char array (in the form: source[starting]).
destination	*destination* must be a variable.
success	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:

```
char src1[5]="12345"
int result1
bool success1
success1 = StringDecAsc2Bin(src1[0], result1)
// success1=true, result1 is 12345

char src2[5] = "-6789"
short result2
bool success2
success2 = StringDecAsc2Bin(src2[0], result2)
// success2 = true,   result2 is -6789

short result3
bool success3
success3 = StringDecAsc2Bin("32768", result3)
// success3=true, but the result exceeds the data range of result3

char src4[2]="4b"
short result4
bool success4
success4 = StringDecAsc2Bin (src4[0], result4)
// success4=false
```

StringBin2DecAsc ()

Description:
Converts an integer to a decimal string.

Syntax:
success = StringBin2DecAsc (*source, destination[starting]*)

Argument	Description
source	This function converts binary data into a decimal string. It converts the binary data in the *source* parameter into a decimal string and saves it into a variable (*destination*). *source* can be either a constant or a variable.
destination[starting]	*destination* must be a one-dimensional char array, to store the result of the conversion.
success	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*, it returns false. The *success* field is optional.

Example:

```
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, the length of the decimal string after conversion exceeds the size of the destination
```

StringDecAsc2Float ()

Description:
Converts a decimal string to float.

Syntax:
success = StringDecAsc2Float (*source[starting], destination*)
or
success = StringDecAsc2Float (*"source", destination*)

Argument	Description
source[starting]	This function converts a decimal string to floating point values. It converts the decimal string in the *source* parameter into floats and saves it into a variable (*destination)*.
destination	*destination* must be a variable.
success	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:

```
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
```

StringFloat2DecAsc ()

Description:
Converts a float to a decimal string.

Syntax:
success = StringFloat2DecAsc(*source, destination[starting]*)

Argument	Description
source	This function converts a floating point data into a decimal string. It converts the float data in the *source* parameter into a decimal string and saves it into a variable (*destination*). *source* can be either a constant or a variable.
destination[starting]	*destination* must be a one-dimensional char array, to store the result of the conversion.
success	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*, it returns false. The *success* field is optional.

Example:

```
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, the length of the decimal string after conversion exceeds the size of the destination
```

StringHexAsc2Bin ()

Description:
Converts a hexadecimal string to binary data.

Syntax:
success = StringHexAsc2Bin (*source[starting], destination*)
or
success = StringHexAsc2Bin (*"source", destination*)

Argument	Description
source[starting]	This function converts a hexadecimal string to binary data. It converts the hexadecimal string in the *source* parameter into binary data and saves it into a variable (*destination)*.
destination	*destination* must be a variable.
success	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:

```
char src1[5]="0x3c"
int result1
bool success1
success1 = StringHexAsc2Bin(src1[0], result1)
// success1=true, result1 is 3c

short result2
bool success2
success2 = StringDecAsc2Bin("1a2b3c4d", result2)
// success2=true, result2=3c4d.The result exceeds the data range of result2

char src3[2]="4g"
short result3
bool success3
success3 = StringDecAsc2Bin (src3[0], result3)
// success3=false
```

StringBin2HexAsc ()

Description:
Converts binary data to a hexadecimal string.

Syntax:
success = StringBin2HexAsc (*source, destination[starting]*)

Argument	Description
source	This function converts binary data to a hexadecimal string. It converts the binary data in source parameter into a hexadecimal string and saves it into a variable (*destination*). This function cannot convert negative values. *source* can be either a constant or a variable.
destination[starting]	*destination* must be a one-dimensional char array, to store the result of the conversion.
success	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*, it returns false. The *success* field is optional.

Example:
```
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, the length of the decimal string after conversion exceeds the size of the destination
```

DATE2ASCII ()

Description:
Converts today's date to an ASCII string.

Syntax:
DATE2ASCII (*day_offset, date[starting], count, separator*)

Argument	Description
day_offset	Will be added into the ASCII string. *day_offset* can be a constant or variable.
date[starting]	This function block will convert today's date into a string and saves it to *date*. s*tarting* must be a constant.
count	Represents the length of *date*. *count* can be a constant or variable.
separator	Separates year, month, and day. The separator is "/" by default. *Separator* can be either a character or a variable.

Example:
char date_str[10]

DATE2ASCII (0, date_str[0], 10) // today's date is 2020/12/5. data_str is "2020/12/5"

DATE2ASCII (5, date_str[0], 10) // today's date is 2020/12/5. data_str is "2020/12/10"

DATE2ASCII (0, result[0], 10, "_")// today's date is 2020/12/5. data_str is "2020_12_5"

DATE2DEC ()

Description:
Converts today's date to a decimal value.

Syntax:
DATE2ASCII (*day_offset, date*)

Argument	Description
day_offset	Will be added into the decimal value. *day_offset* can be a constant or variable.
date	This function block will convert today's date into a decimal value and saves it to *date*. *date* must be a variable.

Example:
int day_offest=5, date

DATE2DEC (0, date) // today's date is 2020/12/5. date is 20201205

DATE2DEC (day_offest, date) // today's date is 2020/12/5. date is 20201210

Chapter 6. String Operation Functions

String2Unicode ()

Description:
Converts all the characters in the source string to Unicode.

Syntax:
result = String2Unicode(*"source", destination[starting]*)

Argument	Description
source	This function converts all the characters in the source string to Unicode and saves the result into a variable (destination). *source* must be a constant
destination[starting]	*destination* must be a one-dimensional char array, to store the result of the conversion.
result	The length of result string after conversion

Example:
char dest[20]
int result

result = String2Unicode("abcde", dest[0]) // *result* will be set to 10.
result = String2Unicode("abcdefghijklmno", dest[0]) // *result* will be set to 20.

StringCat ()

Description:
Appends source string to destination string.

Syntax:
success = StringCat (*source[starting], destination[starting]*)
or
success = StringCat (*"source", destination[starting]*)

Argument	Description
source[starting]	This function 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[starting]	*destination* must be a one-dimensional char array.
success	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:
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

StringCompare ()

Description:
Performs a case-sensitive comparison of two strings.

Syntax:
result = StringCompare (*str1[starting], str2[starting]*)
or
result = StringCompare (*"string1", str2[starting]*)
or
result = StringCompare (*str1[starting], "string2"*)
or
result = StringCompare (*"string1", "string2"*)

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

Example:
char a1[20] = “abcde”
char b1[20] = “ABCDE”
bool result1
result1 = StringCompare(a1[0], b1[0]) // result1 = false

char a2[20] = “abcde”
char b2[20] = “abcde”
bool result2
result2 = StringCompare(a2[0], b2[0]) // result2 = true

char a3[20] = “abcde”
char b3[20] = “abcdefg”
bool result3
result3 = StringCompare(a3[0], b3[0]) // result3 = false

StringCompareNoCase ()

Description:
Performs a case-insensitive comparison of two strings.

Syntax:
result = StringCompareNoCase (*str1[starting], str2[starting]*)
or
result = StringCompareNoCase (*"string1", str2[starting]*)
or
result = StringCompareNoCase (*str1[starting], "string2"*)
or
result = StringCompareNoCase (*"string1", "string2"*)

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

Example:
```
char a1[20]="abcde"
char b1[20]="ABCDE"
bool result1
result1= StringCompareNoCase(a1[0], b1[0])
// result1=true

char a2[20]="abcde"
char b2[20]="abcde"
bool result2
result2= StringCompareNoCase(a2[0], b2[0])
// result2=true

char a3 [20]="abcde"
char b3[20]="abcdefg"
bool result3
result3= StringCompareNoCase(a3[0], b3[0])
// result3=false
```

StringCopy ()

Description:
Copies one string to the other string.

Syntax:
success = StringCopy (*"source", destination[starting]*)
or
success = StringCopy (*source[starting], destination[starting]*)

Argument	Description
source[starting]	This function copies a static string or a string that is stored in an array to a string (destination). The source string parameter accepts both static string (in the form: "source") and char array (in the form: source[starting]).
destination[starting]	*destination*[starting] must be an one-dimensional char array.
success	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:
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”

StringIncluding ()

Description:
Retrieves a substring of the source string that contains characters in the set string, beginning with the first character in the source string and ending when a character is found in the source string that is not in the target string.

Syntax:
success = StringIncluding (*source[starting], set[starting], destination[starting])*
or
success = StringIncluding (*"source", set[starting], destination[starting])*
or
success = StringIncluding (*source[starting], "set", destination[starting])*
or
success = StringIncluding (*"source", "set", destination[starting])*

Argument	Description
source[starting]	This function 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 parameter accept both static string (in the form: "*source"*) and char array (in the form: *source[starting]*).
set[starting]	The *set* string parameter accept both static string (in the form: "*set"*) and char array (in the form: *set[starting]*).
destination[starting]	*destination*[starting] must be a one-dimensional char array.
success	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:
char src1[20]="cabbageabc"
char set1[20]="abc"
char dest1[20] the length of the retrieved substring exceeds the size of the *destination* buffer
bool success1
success1 = StringIncluding(src1[0], set1[0], dest1[0])
// success1=true, dest1="cabba"

char src2[20]="gecabba"
char dest2[20]
bool success2
success2 = StringIncluding(src2[0], "abc", dest2[0])
// success2=true, dest2=""

char set3[20]="abc"
char dest3[4]
bool success3
success3 = StringIncluding("cabbage", set3[0], dest3[0])
// success3=false, dest3 remains the same because the length of the retrieved substring exceeds the size of the *destination* buffer

StringExcluding ()

Description:
Retrieves a substring of the source string that contains characters that are not in the set string.

Syntax:
success = StringExcluding (*source[starting], set[starting], destination[starting])*
or
success = StringExcluding (*"source", set[starting], destination[starting])*
or
success = StringExcluding (*source[starting], "set", destination[starting])*
or
success = StringExcluding (*"source", "set", destination[starting])*

Argument	Description
source[starting]	This function 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 parameter accept both static string (in the form: "*source"*) and char array (in the form: *source[starting]*).
set[starting]	The *set* string parameter accept both static string (in the form: "*set"*) and char array (in the form: *set[starting]*).
destination[starting]	*destination*[starting] must be a one-dimensional char array.
success	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:
char src1[20]="cabbageabc"
char set1[20]="ge"
char dest1[20]
bool success1
success1 = StringExcluding(src1[0], set1[0], dest1[0])
// success1=true, dest1="cabba"

char src2[20]="cabbage"
char dest2[20]
bool success2
success2 = StringExcluding(src2[0], "abc", dest2[0])
// success2=true, dest2=""

char set3[20]="ge"
char dest3[4]
bool success3
success3 = StringExcluding("cabbage", set3[0], dest3[0])
// success3=false, dest3 remains the same because the length of the retrieved substring exceeds the size of the *destination* buffer

StringFind ()

Description:
Returns the position (zero-based index) of the first character of substring in the source string that matches the target string.

Syntax:
position = StringFind (*source[starting], target[starting])*
or
position = StringFind (*"source", target[starting])*
or
position = StringFind (*source[starting], "target")*
or
position = StringFind (*"source", "target")*

Argument	Description
source[starting]	This function returns the position of the first occurrence of the target string in the source string. The *source* string parameter accept both static string (in the form: "*source"*) and char array (in the form: *source[starting]*).
target[starting]	The *target* string parameter accept both static string (in the form: "*target"*) and char array (in the form: *target [starting]*).
position	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:
```
char src1[20]="abcde"
char target1[20]="cd"
short pos1
pos1= StringFind(src1[0], target1[0])
// pos1=2

char target2[20]="ce"
short pos2
pos2= StringFind("abcde", target2[0])
// pos2=-1, there is no matching substring

char src3[20]="abcde"
short pos3
pos3= StringFind(src3[3], "cd")
// pos3=-1, there is no matching substring
```

StringFindOneOf ()

Description:
Returns the position (zero-based index) of the first character in the source string that is also in the target string.

Syntax:
position = StringFindOneOf (*source[starting], target[starting])*
or
position = StringFindOneOf (*"source", target[starting])*
or
position = StringFindOneOf (*source[starting], "target")*
or
position = StringFindOneOf (*"source", "target")*

Argument	Description
source[starting]	This function returns the position of the first character in the source string that matches any character contained in the target string. The *source* string parameter accept both static string (in the form: "*source"*) and char array (in the form: *source[starting]*).
target[starting]	The *target* string parameter accept both static string (in the form: "*target"*) and char array (in the form: *target [starting]*).
position	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:
```
char src1[20]="abcdeabcde"
char target1[20]="sdf"
short pos1
pos1= StringFindOneOf(src1[0], target1[0])
// pos1=3

char src2[20]="abcdeabcde"
short pos2
pos2= StringFindOneOf(src2[1], "agi")
// pos2=4

char target3 [20]="bus"
short pos3
pos3= StringFindOneOf("abcdeabcde", target3[1])
// pos3=-1, there is no matching substring
```

StringReverseFind ()

Description:
Returns the position (zero-based index) of the last occurrence of target string in the source string.

Syntax:
position = StringReverseFind (*source[starting], target[starting])*
or
position = StringReverseFind (*"source", target[starting])*
or
position = StringReverseFind (*source[starting], "target")*
or
position = StringReverseFind (*"source", "target")*

Argument	Description
source[starting]	This function returns the position of the last occurrence of the target string in the source string. The *source* string parameter accept both static string (in the form: "*source"*) and char array (in the form: *source[starting]*).
target[starting]	The *target* string parameter accept both static string (in the form: "*target"*) and char array (in the form: *target [starting]*).
position	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*

Example:

```
char src1[20]="abcdeabcde"
char target1[20]="cd"
short pos1
pos1= StringReverseFind(src1[0], target1[0])
// pos1=7

char target2[20]="ce"
short pos2
pos2= StringReverseFind("abcdeabcde", target2[0])
// pos2=-1, there is no matching substring


char src3[20]="abcdeabcde"
short pos3
pos3= StringReverseFind(src3[6], "ab")
// pos3=-1, there is no matching substring
```

StringGet ()

Description:
Receives string data from the PLC.

Syntax:
StringGet(*read_data[starting], device_name, address_type, address, data_count*)

Argument	Description
read_data[starting]	This function receives string data from the PLC. The string data is stored into *read_data[starting]* to *read_data[starting+data_count-1]*. This function read characters until the end characters of the string is Null ('\0'). *read_data* must be a one-dimensional char array.
device_name	*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
address_type	*address_type* is the register type where the data is stored in the PLC.
address	*address* is the starting address in the PLC.
data_count	*data_count* is the amount of data read. Reading two ASCII characters is equivalent to reading one 16-bit register.

Example:
char str1[20]

StringGet(str1[0], "Local HMI", LW, 0, 20) // reads up to 10 words (20 ASCII characters) from LW-0~LW-9 to the variables str1[0] to str1[19]

StringGetEx ()

Description:
Receives string data from the PLC and continues executing next command even if there's no response from the PLC.

Syntax:
StringGetEx(*read_data[starting], device_name, address_type, address, data_count*)

Argument	Description
read_data[starting]	This function receives string data from the PLC. The string data is stored into *read_data[starting]* to *read_data[starting+data_count-1]*. This function read characters until the end characters of the string is Null ('\0'). *read_data* must be a one-dimensional char array.
device_name	*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
address_type	*address_type* is the register type where the data is stored in the PLC.
address	*address* is the starting address in the PLC.
data_count	*data_count* is the amount of data read. Reading two ASCII characters is equivalent to reading one 16-bit register.

Example:
```
char str1[20]
short test=0

// macro will continue executing test = 1 even if the MODBUS device is not responding
StringGetEx(str1[0], "MODBUS RTU", 4x, 0, 20)
test = 1

// macro won't continue executing test = 2 until MODBUS device responds
StringGet(str1[0], "MODBUS RTU", 4x, 0, 20)
test = 2
```

StringSet ()

Description:
Sends string data to the PLC.

Syntax:
StringSet(*send_data[starting], device_name, address_type, address, data_count*)

Argument	Description
send_data[starting]	This function sends string data to the PLC. The string data is defined in *send_data[starting]* to *send_data[starting+data_count-1]*. *send_data* must be a one-dimensional char array.
device_name	*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
address_type	*address_type* is the register type where the data is stored in the PLC.
address	*address* is the starting address in the PLC.
data_count	*data_count* is the amount of data written.

Example:
char str1[10] = “abcde”

StringSet(str1[0], “Local HMI”, LW, 0, 10)
// This function transfer each characters of str1 until the end characters of the string is Null (‘\0’).

StringSetEx ()

Description:
Sends string data to the PLC and continues executing next command even if there's no response from the PLC.

Syntax:
StringSetEx(*send_data[starting], device_name, address_type, address, data_count*)

Argument	Description
send_data[starting]	This function sends string data to the PLC. The string data is defined in *send_data[starting]* to *send_data[starting+data_count-1]*. *send_data* must be a one-dimensional char array. The macro will move on to the next line even if there is no response from the PLC.
device_name	*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
address_type	*address_type* is the register type where the data is stored in the PLC.
address	*address* is the starting address in the PLC.
data_count	*data_count* is the amount of data written.

Example:
```
char str1[20]="abcde"
short test=0
// macro will continue executing test = 1 even if the MODBUS device is not responding
StringSetEx(str1[0], "MODBUS RTU", 4x, 0, 20)
test = 1

// macro will not continue executing test = 2 until MODBUS device responds
StringSet(str1[0], "MODBUS RTU", 4x, 0, 20)
test = 2
```

StringInsert ()

Description:
Inserts a string in **a specific location** within the destination string.

Syntax:
success = StringInsert (*pos, insert[starting], destination[starting])*
or
success = StringInsert (*pos, "insert", destination[starting])*
or
success = StringInsert (*pos, insert[starting], length, destination[starting])*
or
success = StringInsert (*pos, "insert", length, destination[starting])*

Argument	Description
pos	This function inserts a string in a specific location within the destination string. The insert location is specified by the pos parameter.
insert[starting]	The *insert* string parameter accepts both static string (in the form: “insert”) and char array (in the form: insert[starting]).
destination[starting]	*destination*[starting] must be a one-dimensional char array.
length	The number of characters to insert can be specified by the *length* parameter
success	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:
```
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"
```

StringLength ()

Description:
Obtains the length of a string.

Syntax:
length = StringLength (*source[starting])*
or
length = StringLength (*"source")*

Argument	Description
source[starting]	This function is used to output the length of a string. The *source* string parameter accepts both static string (in the form: *source*) and char array (in the form: *source[starting]*).
length	The *length* value indicates the length of the source string.

Example:

```
char src1[20]="abcde"
int length1
length1= StringLength(src1[0])
// length1=5

char src2[20]={'a', 'b', 'c', 'd', 'e'}
int length2
length2= StringLength(src2[0])
// length2=5

char src3[20]="abcdefghij"
int length3
length3= StringLength(src3 [2]) // gets the length of the string starting from the “third” character
// length3=8
```

StringMid ()

Description:
Retrieves a substring from the specified position of the source string.

Syntax:
success = StringMid (*source[starting], count, destination[starting])*
or
success = StringMid (*"string", starting, count, destination[starting])*

Argument	Description
source[starting]	The *source* string parameter accepts both static string (in the form: *source*) and char array (in the form: *source[starting]*).
starting	The *starting* parameter specifies the starting position of the *source* string being retrieved.
count	The *count* parameter specifies the length of the substring being retrieved.
destination[starting]	*destination* must be a one-dimensional char array to store the retrieved substring.
success	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:
```
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"
```

StringToUpper ()

Description:
Converts all the characters in the source string to uppercase characters.

Syntax:
success = StringToUpper (*source[starting], destination[starting])*
success = StringToUpper (*"source", destination[starting])*

Argument	Description
source[starting]	This function converts all the characters in the source string to uppercase characters and save the result into a variable (*destination*). The *source* string parameter accepts both static string (in the form: "*source"*) and char array (in the form: *source[starting]*).
destination[starting]	*destination* must be a one-dimensional char array.
success	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:

```
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, the length of the result string after conversion exceeds the size of the destination
```

StringToLower ()

Description:
Converts all the characters in the source string to lowercase characters.

Syntax:
success = StringToLower (*source[starting], destination[starting])*
success = StringToLower (*"source", destination[starting])*

Argument	Description
source[starting]	This function converts all the characters in the source string to lowercase characters and save the result into a variable (*destination*). The *source* string parameter accepts both static string (in the form: "*source"*) and char array (in the form: *source[starting]*).
destination[starting]	*destination* must be a one-dimensional char array.
success	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:
```
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, the length of the result string after conversion exceeds the size of the destination
```

StringToReverse ()

Description:
Reverses the characters in the source string.

Syntax:
success = StringToReverse (*source[starting], destination[starting])*
success = StringToReverse (*"source", destination[starting])*

Argument	Description
source[starting]	This function reverses the characters in the *source* string and stores it in a variable (*destination)*. The *source* string parameter accepts both static string (in the form: "*source"*) and char array (in the form: *source[starting]*).
destination[starting]	*destination* must be a one-dimensional char array.
success	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:
```
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, the length of the result string after conversion exceeds the size of the destination
```

StringTrimLeft ()

Description:
Trims the prefix characters from the source string.

Syntax:
success = StringTrimLeft (*source[starting], set[starting], destination[starting])*
or
success = StringTrimLeft (*"source", set[starting], destination[starting])*
or
success = StringTrimLeft (*source[starting], "set", destination[starting])*
or
success = StringTrimLeft (*"source", "set", destination[starting])*

Argument	Description
source[starting]	This function trims the specified characters in the *set string* from the left 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[starting]*).
set[starting]	The *set* string and *set* string parameters accept both static string (in the form: "*set"*) and char array (in the form: *set [starting]*).
destination[starting])	*destination* must be a one-dimensional char array.
success	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:
```
char src1[20]= "# *a*#bc"
char set1[20]="# *"
char dest1[20]
bool success1
success1 = StringTrimLeft (src1[0], set1[0], dest1[0])
// success1=true, dest1="a*#bc"

char set2[20]={'#', ' ', '*'}
char dest2[4]
bool success2
success2 = StringTrimLeft ("# *a*#bc", set2[0], dest2[0])
// success2=false, the length of the result string after conversion exceeds the size of the destination

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

StringTrimRight ()

Description:
Trims the suffix characters from the source string.

Syntax:
success = StringTrimRight (*source[starting], set[starting], destination[starting])*
or
success = StringTrimRight (*"source", set[starting], destination[starting])*
or
success = StringTrimRight (*source[starting], "set", destination[starting])*
or
success = StringTrimRight (*"source", "set", destination[starting])*

Argument	Description
source[starting]	This function trims the specified characters in the *set string* from the right 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[starting]*).
set[starting]	The *set* string and *set* string parameters accept both static string (in the form: "*set"*) and char array (in the form: *set [starting]*).
destination[starting])	*destination* must be a one-dimensional char array.
success	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:
```
char src1[20]= "# *a*#bc# * "
char set1[20]="# *"
char dest1[20]
bool success1
success1 = StringTrimRight(src1[0], set1[0], dest1[0])
// success1=true, dest1="# *a*#bc"

char set2[20]={'#', ' ', '*'}
char dest2[20]
bool success2
success2 = StringTrimRight("# *a*#bc", set2[0], dest2[0])
// success2=true, dest2="# *a*#bc"

char src3[20]="ab**c *#"
char dest3[4]
bool success3
success3 = StringTrimRight(src3[0], "# *", dest3[0])
// success3=false, the length of the result string after conversion exceeds the size of the destination
```

StringMD5 ()

Description:
Generates 32 characters using MD5 message-digest algorithm.

Syntax:
result = StringMD5(*source[starting], destination[starting])*
or
result = StringMD5(*"source", destination[starting])*

Argument	Description
source[starting]	This function generates a MD5 message-digest string. The *source* string parameters accept both static string (in the form: "*source"*) and char array (in the form: *source[starting]*).
destination[starting]	*destination* must be a one-dimensional char array.
result	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:
```
char source[32] = "password", dest[32]
bool result

result = StringMD5(source[0], dest[0])

result = StringMD5("password", dest[0])
// "result" will be set to 32, which is the length of MD5 string.
// dest[] = 5f4dcc3b5aa765d61d8327deb882cf99
```

Utf82Unicode ()

Description:
Converts a UTF8 string into a Unicode string.

Syntax:
result = Utf82Unicode(*source[starting], destination[starting])*
or
result = Utf82Unicode(*"source", destination[starting])*

Argument	**Description**
source[starting]	The *source* string parameters accept both static string (in the form: "*source"*) and char array (in the form: *source[starting]*).
destination[starting]	*destination* must be a one-dimensional char array.
result	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:

```
char unicode_str[20]
char utf8_str[20]
bool result

String2Unicode("ABC", unicode_str[0])
result = Unicode2Utf8(unicode_str[0], utf8_str[0])
// result will be set to true. utf8_str[] will be "ABC" encoded in UTF8

char dst[20]
bool result2

result2 = Utf82Unicode(utf8_str[0], dst[0])
// result2 will be set to true. dst[] will be "ABC" encoded in Unicode.
```

Unicode2Utf8 ()

Description:
Converts a Unicode string into a UTF8 string.

Syntax:
result = Unicode2Utf8 (*source[starting], destination[starting])*
or
result = Unicode2Utf8 (*"source", destination[starting])*

Argument	**Description**
source[starting]	The *source* string parameters accept both static string (in the form: "*source"*) and char array (in the form: *source[starting]*).
destination[starting]	*destination* must be a one-dimensional char array.
result	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:
```
char unicode_str[20]
char utf8_str[20]
bool result

String2Unicode("ABC", unicode_str[0])
result = Unicode2Utf8(unicode_str[0], utf8_str[0])
// result will be set to true. utf8_str[] will be "ABC" encoded in UTF8
```

UnicodeCat ()

Description:
Appends source string to destination string.

Syntax:
success = UnicodeCat (*source[starting], destination[starting]*)
or
success = UnicodeCat (*"source", destination[starting]*)

Argument	Description
source[starting]	This function 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[starting]	*destination* must be a one-dimensional char array.
success	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:
```
char strSrc[12]="αθβγθδ"
char strDest[28]="ζηθλ1234"
bool result

result = UnicodeCat(strSrc[0], strDest[0])
// "result" will be set to true. "strDest" will be set to "ζηθλ1234αθβγθδ"
```

UnicodeCompare ()

Description:
Performs a case-sensitive comparison of two strings.

Syntax:
result = UnicodeCompare (*str1[starting], str2[starting]*)
or
result = UnicodeCompare (*"string1", str2[starting]*)
or
result = UnicodeCompare (*str1[starting], "string2"*)
or
result = UnicodeCompare (*"string1", "string2"*)

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

Example:
char str1[10]=" θαβθγ"
char str2[8]="αβγδ"
bool result

result = UnicodeCompare(str1[0], str2[0]) // "result" will be set to false.
result = UnicodeCompare(str1[0], "θαβθγ") // "result" will be set to true.

UnicodeCopy ()

Description:
Copies one string to the other string.

Syntax:
success = UnicodeCopy (*"source", destination[starting]*)
or
success = UnicodeCopy (*source[starting], destination[starting]*)

Argument	Description
source[starting]	This function copies a static string or a string that is stored in an array to a string (destination). The source string parameter accepts both static string (in the form: "source") and char array (in the form: source[starting]).
destination[starting]	*destination*[starting] must be a one-dimensional char array.
success	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:
```
char strSrc[14]="αβθγδθε"
char strDest[14]
bool result

result = UnicodeCopy("αβθγδθε", strDest[0])
// "result" will be set to true, strDest = αβθγδθε"
```

UnicodeExcluding ()

Description:
Retrieves a substring of the source string that contains characters that are not in the set string.

Syntax:
success = UnicodeExcluding (*source[starting], set[starting], destination[starting])*
or
success = UnicodeExcluding (*"source", set[starting], destination[starting])*
or
success = UnicodeExcluding (*source[starting], "set", destination[starting])*
or
success = UnicodeExcluding (*"source", "set", destination[starting])*

Argument	Description
source[starting]	This function 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 parameter accept both static string (in the form: "*source"*) and char array (in the form: *source[starting]*).
set[starting]	The *set* string parameter accept both static string (in the form: "*set"*) and char array (in the form: *set[starting]*).
destination[starting]	*destination*[starting] must be a one-dimensional char array.
success	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:
```
char source[14]="γδξκθλθ, dest[8]
char set[4]="λθ"
bool result

result = UnicodeExcluding(source[0], set[0], dest[0])
// "result" will be set to true and "dest" will be set to "γδξκ".
```

UnicodeLength ()

Description:
Obtains the length of a string.

Syntax:
length = UnicodeLength (*source[starting])*
or
length = UnicodeLength (*"source")*

Argument	Description
source[starting]	This function is used to output the length of a string. The *source* string parameter accepts both static string (in the form: *source*) and char array (in the form: *source[starting]*).
length	The *length* value indicates the length of the source string.

Example:
char strSrc[6]="ÅÈÑ"
short length

length = UnicodeLength(strSrc[0]) // " length " is equal to 3

length = UnicodeLength("ÅÈÑ") // " length " is equal to 3

Chapter 7. Mathematic Functions

SQRT ()

Description:
Calculates the square root of source.

Syntax:
SQRT(*source, result*)

Argument	Description
source	This function calculates the square root of *source* and saves the result into *result*. *source* can be a constant or a variable, but *source* must be a non-negative value.
result	*result* must be a variable.

Example:

```
float source, result

SQRT(16, result) // result is 4.0

source = 9.0
SQRT(source, result)// result is 3.0
```

CUBERT ()

Description:
Calculates the cube root of source.

Syntax:
CUBERT(*source, result*)

Argument	Description
source	This function calculates the cube root of *source* and saves the result into *result*. *source* can be a constant or a variable, but *source* must be a non-negative value.
result	*result* must be a variable.

Example:
float source, result

CUBERT (27, result) // result is 3.0

source = 27.0
CUBERT(source, result)// result is 3.0

POW ()

Description:
Calculates the power of source.

Syntax:
POW(*source1, source2, result*)

Argument	Description
source1	This function calculates *source1* to the power of *source2*. *source1* can be a constant or a variable, but *source* must be a non-negative value.
source2	*source2* can be a constant or a variable, but *source* must be a non-negative value.
result	*result* must be a variable.

Example:
float y, result

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

SIN ()

Description:
Calculates the sine of source.

Syntax:
SIN(*source, result*)

Argument	Description
source	This function calculates the sine of *source* (in degrees) and saves the result into *result*. *source* can be a constant or a variable.
result	*result* must be a variable.

Example:
float source, result

SIN(90, result) // result is 1.0

source = 30
SIN(source, result) // result is 0.5

COS ()

Description:
Calculates the cosine of source.

Syntax:
COS(*source, result*)

Argument	Description
source	This function calculates the cosine of *source* (in degrees) and saves the result into *result*. *source* can be a constant or a variable.
result	*result* must be a variable.

Example:
float source, result

COS(90, result) // result is 0

source = 60
COS(source, result) // result is 0.5

TAN ()

Description:
Calculates the tangent of source.

Syntax:
TAN(*source, result*)

Argument	Description
source	This function calculates the tangent of *source* (in degrees) and saves the result into *result*. *source* can be a constant or a variable.
result	*result* must be a variable.

Example:
float source, result

TAN(45, result) // result is 1.0

source = 60
TAN(source, result) // result is 1.732

COT ()

Description:
Calculates the cotangent of source.

Syntax:
COT(*source, result*)

Argument	Description
source	This function calculates the cotangent of *source* (in degrees) and saves the result into *result*. *source* can be a constant or a variable.
result	*result* must be a variable.

Example:
float source, result

COT(45, result) // result is 1.0

source = 60
COT(source, result) // result is 0.5774

SEC ()

Description:
Calculates the secant of source

Syntax:
SEC(*source, result*)

Argument	Description
source	This function calculates the secant of *source* (in degrees) and saves the result into *result*. *source* can be a constant or a variable.
result	*result* must be a variable.

Example:
float source, result

SEC(45, result) // result is 1.414

source = 60
SEC(source, result) // result is 2.0

CSC ()

Description:
Calculates the cosecant of source.

Syntax:
CSC(*source, result*)

Argument	Description
source	This function calculates the cosecant of *source* (in degrees) and saves the result into *result*. *source* can be a constant or a variable.
result	*result* must be a variable.

Example:
float source, result

CSC(45, result) // result is 1.414

source = 30
CSC(source, result) // result is 2.0

ASIN ()

Description:
Calculates the arc sine of source.

Syntax:
ASIN(*source, result*)

Argument	Description
source	This function calculates the arc sine of *source* (in degrees) and saves the result into *result*. *source* can be a constant or a variable.
result	*result* must be a variable.

Example:
float source, result

ASIN(0.8660, result) // result is 60

source = 0.5
ASIN(source, result) // result is 30

ACOS ()

Description:
Calculates the arc cosine of source.

Syntax:
ACOS(*source, result*)

Argument	Description
source	This function calculates the arc cosine of *source* (in degrees) and saves the result into *result*. *source* can be a constant or a variable.
result	*result* must be a variable.

Example:
float source, result

ACOS(0.8660, result) // result is 30

source = 0.5
TAN(source, result) //result is 60

ATAN ()

Description:
Calculates the arc tangent of source.

Syntax:
ATAN(*source, result*)

Argument	Description
source	This function calculates the arc tangent of *source* (in degrees) and saves the result into *result*. *source* can be a constant or a variable.
result	*result* must be a variable.

Example:
float source, result

ATAN(1, result) // result is 45

source = 1.732
TAN(source, result) // result is 60

LOG ()

Description:
Calculates the natural logarithm of a number.

Syntax:
LOG(*source, result*)

Argument	Description
source	This function calculates the natural logarithm of *source* and saves the result into *result*. *source* can be a constant or a variable.
result	*result* must be a variable.

Example:
float source = 100, result

LOG(source, result) // result is approximately 4.6052

LOG10 ()

Description:
Calculates the base-10 logarithm of a number.

Syntax:
LOG10(*source, result*)

Argument	Description
source	This function calculates the base-10 logarithm of *source* and saves the result into *result*. *source* can be a constant or a variable.
result	*result* must be a variable.

Example:
float source = 100, result

LOG10(source, result) //result is 2.0

RAND ()

Description:
Calculates a random integer.

Syntax:
RAND(*result*)

Argument	Description
result	This function generates a random integer and saves the random into *result*. *result* must be a variable.

Example:
short result

RAND(result) // result will vary each time the macro is executed

CEIL ()

Description:
Calculates the smallest integral value that is not less than the input value.

Syntax:
result=CEIL(*source*)

Argument	Description
result	This function calculates the smallest integral value that is not less than *source* and saves the result into *result*. *result* must be a variable.

Example:

```
float x = 3.8
int result

result = CEIL(x)   // result = 4
```

FLOOR ()

Description:
Calculates the largest integral value that is not greater than the input value.

Syntax:
result=FLOOR(*source*)

Argument	Description
result	This function calculates the largest integral value that is not greater than *source* and saves the result into *result*. *result* must be a variable.

Example:

```
float x = 3.8
int result

result = FLOOR(x)          // result = 3
```

ROUND ()

Description:
Rounds the input value to the nearest integral value.

Syntax:
result=ROUND(*source*)

Argument	Description
result	This function rounds *source* to the nearest whole number and saves the result into *result*. *result* must be a variable.

Example:

```
float x = 5.55
int result

result = ROUND(x)        // result = 6
```

Chapter 8. Statistic Functions

AVERAGE ()

Description:
Gets the average value from an array.

Syntax:
AVERAGE(*source[starting], result, count*)

Argument	Description
source[starting]	*source* must be a one-dimensional char array.
result	*result* must be a variable.
count	*count* can be a constant or a variable.

Example:
short data[5] = {1, 2, 3, 4, 5}
float result

AVERAGE(data[0], result, 5) // result is equal to 3

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

HARMEAN ()

Description:
Gets the harmonic mean value from an array.

Syntax:
HARMEAN(*source[starting], result, count*)

Argument	Description
source[starting]	*source* must be a one-dimensional char array.
result	*result* must be a variable.
count	*count* can be a constant or a variable.

Example:
short data[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
float result
HARMEAN(data[0], result, 10) // result is equal to 3.414

MAX ()

Description:
Gets the maximum value from an array.

Syntax:
MAX(*source[starting], result, count*)

Argument	Description
source[starting]	*source* must be a one-dimensional char array.
result	*result* must be a variable.
count	*count* can be a constant or a variable.

Example:
shot data[5] = {1, 2, 3, 4, 5}
short result

MAX(data[0], result, 5) // result is equal to 5

MAX(data[1], result, 3) // 2,3, and 4. The max value is equal to 4

MEDIAN ()

Description:
Gets the median value from an array.

Syntax:
MEDIAN(*source[starting], result, count*)

Argument	Description
source[starting]	*source* must be a one-dimensional char array.
result	*result* must be a variable.
count	*count* can be a constant or a variable.

Example:
short data[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
float result

MEDIAN(data[0], result, 10) // result is equal to 5.5

MIN ()

Description:
Gets the minimum value from an array.

Syntax:
MIN(*source[starting], result, count*)

Argument	Description
source[starting]	*source* must be a one-dimensional char array.
result	*result* must be a variable.
count	*count* can be a constant or a variable.

Example:
short data[5] = {1, 2, 3, 4, 5}
short result

MIN(data[0], result, 5) // result is equal to 1

MIN(data[1], result, 3) // 2,3, and 4. The max value is equal to 2

STDEVP ()

Description:
Gets the standard deviation value from an array.

Syntax:
STDEVP(*source[starting], result, count*)

Argument	Description
source[starting]	*source* must be a one-dimensional char array.
result	*result* must be a variable.
count	*count* can be a constant or a variable.

Example:
short data[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
float result

STDEVP(data[0], result, 10) // result is equal to 2.872

STDEVS ()

Description:
Gets the sample standard deviation value from an array.

Syntax:
STDEVS(*source[starting], result, count*)

Argument	Description
source[starting]	*source* must be a one-dimensional char array.
result	*result* must be a variable.
count	*count* can be a constant or a variable.

Example:

```
short data[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
float result

STDEVS(data[0], result, 10) // result is equal to 3.027
```

Chapter 9. Recipe Database Functions (used for Recipe Database feature)

RecipeGetData ()

Description:
Gets recipe Data.

Syntax:
success=RecipeGetData(*destination, recipe_address, record_ID*)

Argument	Description
destination	This function retrieves the specified recipe data from Recipe Database and save it into a variable (*destination).* *destination* must be a variable
recipe_address	*recipe_address* consists of the recipe name and item name: "recipe_name.item_name".
record_ID	*record_ID* specifies the ID number of the record in the recipe being queried.
success	This function returns a Boolean indicating whether the process is successful or not. If successful, it returns true, otherwise it returns false.

Example:

```
short data=0
char str[20]
short recordID
bool result

recordID = 0
success = RecipeGetData(data, "TypeA.item_weight", recordID)
// From recipe "TypeA", get the data of the item "item_weight" in record 0.

recordID = 1
success = RecipeGetData(str[0], "TypeB.item_name", recordID)
// From recipe "TypeB", get the data of the item "item_name" in record 1.
```

RecipeSetData ()

Description:
Writes data to recipe database.

Syntax:
success=RecipeSetData(*source*, *recipe address*, *record_ID*)

Argument	Description
destination	This function writes data to Recipe Database and save it into a variable (*destination).* *destination* must be a variable
recipe_address	*recipe_address* consists of the recipe name and item name: "recipe_name.item_name".
record_ID	*record_ID* specifies the ID number of the record in the recipe being queried.
success	This function returns a Boolean indicating whether the process is successful or not. If successful, it returns true, otherwise it returns false.

Example:

```
short data=99
char str[20]="abc"
short recordID
bool result

recordID = 0
result = RecipeSetData(data, "TypeA.item_weight", recordID)
// sets data to recipe "TypeA", where item name is "item_weight" and the record ID is 0.

recordID = 1
result = RecipeSetData(str[0], "TypeB.item_name", recordID)
// sets data to recipe "TypeB", where item name is "item_name" and the record ID is 1.
```

RecipeQuery ()

Description:
Queries recipe data.

Syntax:
success=RecipeQuery (*SQL_command, destination*)

Argument	Description
SQL_command	This function uses SQL statements to query recipe data. The number of records from the query result will be stored into a variable (*destination)*. 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.
destination	*destination* must be a variable.
success	This function returns a Boolean indicating whether the process is successful or not. If successful, it returns true, otherwise it returns false.

Example:

```
short total_row=0
char sql[100]="SELECT * FROM TypeB"
short var
bool success

success = RecipeQuery("SELECT * FROM TypeA", total_row)
// Queries Recipe "TypeA" and saves the number of records of query result into total_row.

result = RecipeQuery(sql[0], total_row)
// Queries Recipe "TypeB" and saves the number of records of query result into total_row.

success = RecipeQuery("SELECT * FROM Recipe WHERE Item >%(var)", total_row)
// Queries "Recipe", where "Item" is larger than var and saves the number of records of query result into total_row.
```

RecipeQueryGetData ()

Description:
Gets the recipe data in the query result obtained by **RecipeQuery()**.

Syntax:
success=RecipeQueryGetData (*destination, recipe_address, result_row_no*)

Argument	Description
destination	This function retrieves the recipe data. This function must be called after calling RecipeQuery().
recipe_address	Specify the recipe name in the *recipe_address which is the same name* as RecipeQuery(). *recipe_address* can be static string or char array.
result_row_no	*result_row_no* specifies the row number in the query result.
success	This function returns a Boolean indicating whether the process is successful or not. If successful, it returns true, otherwise it returns false.

Example:

```
short data=0
short total_row=0
short row_number=0
bool result_query
bool success

result_query = RecipeQuery("SELECT * FROM TypeA", total_row)
// Queries Recipe "TypeA" and saves the number of records of query result into total_row.

if (result_query) then
        for row_number=0 to total_row-1

        success= RecipeQueryGetData(data, "TypeA.item_weight", row_number)

        next row_number
end if
```

RecipeQueryGetRecordID ()

Description:
Gets the record ID numbers of those records gained by RecipeQuery().

Syntax:
success=RecipeQueryGetRecordID (*destination, result_row_no*)

Argument	Description
destination	This function gets the record ID numbers of those records obtained by RecipeQuery() and writes the obtained record ID to *destination*. This function must be called after calling RecipeQuery().
result_row_no	*result_row_no* specifies the row number in the query result.
success	This function returns a Boolean indicating whether the process is successful or not. If successful, it returns true, otherwise it returns false.

Example:

```
short recordID=0
short total_row=0
short row_number=0
bool result_query
bool result_id

result_query = RecipeQuery("SELECT * FROM TypeA", total_row)
// Queries Recipe "TypeA" and save the number of records of query result into total_row.

if (result_query) then

        for row_number=0 to total_row-1

        success = RecipeQueryGetRecordID(recordID, row_number)

        next row_number

end if
```

Chapter 10. Data/Event Log Functions (available for non-cMT HMI project)

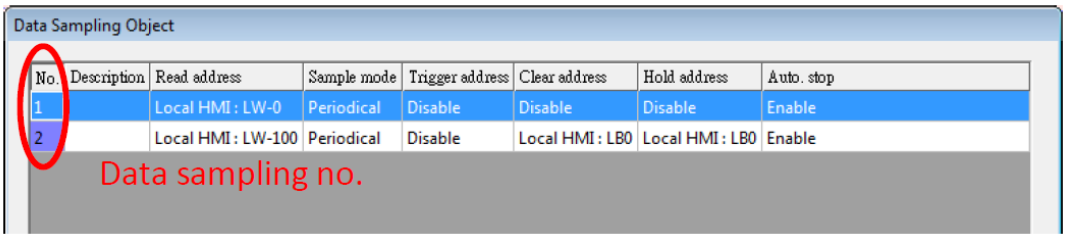
FindDataSamplingDate ()

Description:
Finds the date of the specified data sampling file.

Syntax:
success= FindDataSamplingDate *(data_log_number, index, year, month, day*)

Argument	Description
data_log_number, *index*	This function finds the date of a specified data sampling file using the **data sampling no**. and the **file index**. *data_log_number and index* can be constant or variable.
year, *month*, *day*	The date is stored into *year, month*, and *day,* respectively (in the format: YYYY, MM, and DD.) They must be variables.
success	This function returns a Boolean indicating whether the process is successful or not. If successful, it returns true, otherwise it returns false.

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:
20191210.dtl
20191230.dtl
20200110.dtl
20200111.dtl
The files are indexed as follows:
20191210.dtl -> index is 3
20191230.dtl -> index is 2
20200110.dtl -> index is 1
20200111.dtl -> index is 0



Example:
short data_log_number = 1, index = 2
short year, month, day
bool success

success = FindDataSamplingDate(data_log_number, index, year, month, day)
// if there exists a data sampling file named 20191230.dtl with data sampling number 1 and file index 2, the result after execution: success = 1, year = 2019, month = 12, and day = 30.

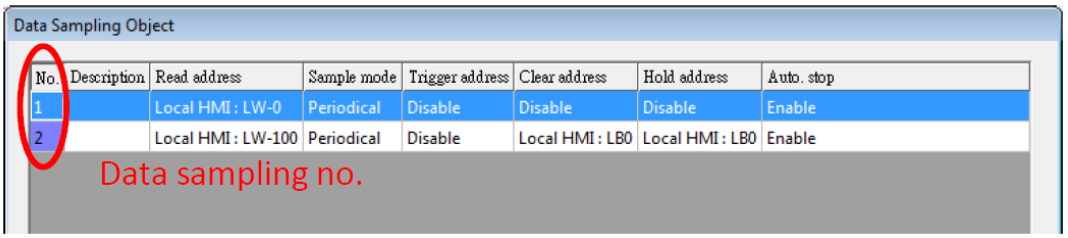
FindDataSamplingIndex ()

Description:
Finds the file index of the specified data sampling file.

Syntax:
success = FindDataSamplingIndex (*data_log_number, year, month, day, index*)

Argument	Description
data_log_number, *year*, *month*, *day*	This function finds the file index of a specified data sampling file using the **data sampling no**. and the **date (in the format of YYYY, MM and DD respectively)**. *data_log_number, year, month* and *day* can be constant or variable.
index	The file index is stored into *index*. *index* must be a variable.
success	This function returns a Boolean indicating whether the process is successful or not. If successful, it returns true, otherwise it returns false.

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:
20191210.dtl
20191230.dtl
20200110.dtl
20200111.dtl
The files are indexed as follows:
20191210.dtl -> index is 3
20191230.dtl -> index is 2
20200110.dtl -> index is 1
20200111.dtl -> index is 0



Example:
short data_log_number = 1, year = 2019, month = 12, day = 10
short index
bool success

success = FindDataSamplingIndex (data_log_number, year, month, day, index)
// if there exists a data sampling file named 20191210.dtl, with data sampling number 1 and file index 2.
The result after execution: success =1 and index =2

FindEventLogDate ()

Description:
Finds the date of the specified event log file.

Syntax:
success= FindEventLogDate(*index, year, month, day*)

Argument	Description
index	This function finds the date of a specified event log file using the file index. *index* can be a constant or a variable.
year, *month*, *day*	The date is stored into *year, month*, and *day* respectively (in the format YYYY, MM, and DD) *year, month, day* must be a variable.
success	This function returns a Boolean indicating whether the process is successful or not. If successful, it returns true, otherwise it returns false.

The event log files are sorted into the file name and 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:
20191210.evt
20191230.evt
20200110.evt
20200111.evt
The files are indexed as follows:
20191210.evt -> index is 3
20191230.evt -> index is 2
20200110.evt -> index is 1
20200111.evt -> index is 0

Example:

```
short index = 1, year, month, day
bool success

success = FindEventLogDate(index, year, month, day)

// if there exists an event log file named 20191230.evt with file index 1, the result after execution: success =1, year =2019, month =12, and day =30.
```

FindEventLogIndex ()

Description:
Finds the file index of the specified event log file.

Syntax:
success= FindEventLogIndex(*year, month, day, index*)

Argument	Description
year, *month*, *day*	This function finds the file index of a specified event log file using the date. *year, month*, and *day* are in the format YYYY, MM, and DD, respectively. *year, month* and *day* can be constant or variable.
index	The file index is stored into *index*. *index* must be a variable.
success	This function returns a Boolean indicating whether the process is successful or not. If successful, it returns true, otherwise it returns false.

The event log files are sorted into the file name and 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:
20191210.evt
20191230.evt
20200110.evt
20200111.evt
The files are indexed as follows:
20191210.evt -> index is 3
20191230.evt -> index is 2
20200110.evt -> index is 1
20200111.evt -> index is 0

Example:
```
short year = 2019, month = 12, day = 10
short index
bool success

success = FindEventLogIndex(year, month, day, index)

// if there exists an event log file named 20191230.evt with data file index 2, the result after execution: success = 1 and index =2.
```

Chapter 11. Checksum Functions

ADDSUM ()

Description:
Adds up the elements of an array to generate a checksum.

Syntax:
ADDSUM(*source[starting], result, data_count*)

Argument	Description
source[starting]	This function adds up the elements of an array (*source*) from *source[starting]* to *source[starting+(data_count-1)]* to generate a checksum and save the checksum into a variable (*result*).
result	*result* must be a variable.
data_count	*data_count* is the number of accumulated elements and can be a constant or a variable.

Example:
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

XORSUM ()

Description:
Uses XOR to calculate the checksum.

Syntax:
XORSUM(*source*[*starting*]*, result, data_count*)

Argument	Description
source[starting]	This function uses XOR to calculate the checksum from *source*[*starting*] to *source*[*starting +data_count - 1*] and save the result into a variable (*result*).
result	*result* must be a variable.
data_count	*data_count* is the amount of the calculated elements of the array and can be a constant or a variable.

Example:
char data[5] = {0x1, 0x2, 0x3, 0x4, 0x5}
short checksum

XORSUM(data[0], checksum, 5)// checksum is 0x1

BCC ()

Description:
Uses an XOR method to calculate the checksum.

Syntax:
BCC(*source[starting], result, data_count*)

Argument	Description
source[starting]	This function uses XOR method to calculate the checksum from *source*[*starting*] to *source*[*starting +data_count - 1*] and save the result into a variable (*result*).
result	*result* must be a variable.
data_count	*data_count* is the amount of the calculated elements of the array and can be a constant or a variable.

Example:
char data[5] = {0x1, 0x2, 0x3, 0x4, 0x5}
short checksum

BCC(data[0], checksum, 5) // checksum is 0x1

CRC ()

Description:
Calculates 16-bit CRC of the variables to generate a checksum.

Syntax:
CRC(*source*[*starting*]*, result, data_count*)

Argument	Description
source[starting]	This function calculates 16-bit CRC of the variables from *source*[*starting*] to source[*starting +data_count - 1*] and save the result into a variable (*result*).
result	*result* must be a variable.
data_count	*data_count* is the amount of the calculated elements of the array and can be a constant or a variable.

Example:
char data[5] = {0x1, 0x2, 0x3, 0x4, 0x5}
short checksum

CRC(data[0], checksum, 5) // checksum is 0xbb2a, 16-bit CRC

CRC8 ()

Description:
Calculates 8-bit CRC of the variables to generate a checksum.

Syntax:
CRC8(*source*[*starting*]*, result, data_count*)

Argument	Description
source[starting]	This function calculates 8-bit CRC of the variables from *source*[*starting*] to source[*starting +data_count - 1*] and save the result into a variable (*result*).
result	*result* must be a variable.
data_count	*data_count* is the amount of the calculated elements of the array and can be a constant or a variable.

Example:
char source[5] = {1, 2, 3, 4, 5}
short CRC8_result

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

Chapter 12. Miscellaneous Functions

Beep ()

Description:
Plays beep sound.

Syntax:
Beep ()

Argument	Description
N/A	This function plays a beep sound with frequency of 800 hertz and duration of 30 milliseconds.

Example:
Beep()

Buzzer ()

Description:
Turns ON / OFF the buzzer.

Syntax:
Buzzer(*On_Off*)

Argument	Description
On_Off	This function turns ON/OFF the buzzer. *On_Off* is a Boolean value and can be a constant or a variable.

Example:
char on = 1, off = 0

Buzzer(on) // turn on the buzzer
DELAY(1000) // delay 1 second

Buzzer(off) // turn off the buzzer
DELAY(500) // delay 500ms

Buzzer(1) // turn on the buzzer
DELAY(1000) // delay 1 second

Buzzer(0) // turn off the buzzer

TRACE ()

Description:
Prints out the current value of variables during run-time of macro for debugging.

Syntax:
TRACE(*format, argument*)

Argument	Description
format	Use this function to send a specified string to the EasyDiagnoser/ cMT Diagnoser. This function 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: **%**[flags] [width] [.precision] **type** The length of the output string is limited to 256 characters. The extra characters will be ignored.
argument	The *argument* part is optional. One format specification converts exactly one argument.

%[flags] [width] [.precision] type
Each field of the format specification is described as below:
flags (optional):
- : Aligns left. When the value has fewer characters than the specified width, it will be padded with spaces on the left.
+ : Precedes the result with a plus or minus sign (+ or -)
width (optional):
A non-negative decimal integer controlling the minimum number of characters printed.
precision (optional):
A non-negative decimal integer which specifies the precision and the number of characters to be printed.

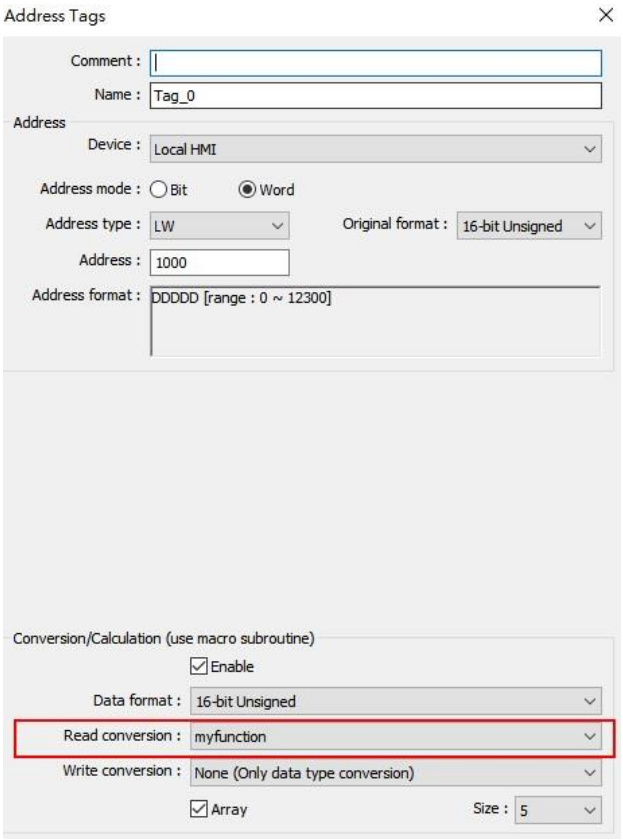
type:
C or c : specifies a single-byte character
d : signed decimal integer
i : signed decimal integer
o : unsigned octal integer
u : unsigned decimal integer
X or x : unsigned hexadecimal integer
lld : signed long integer (64-bit) (cMT Series only)
llu : unsigned long integer (64-bit) (cMT Series only)
f : signed floating-point value
llf : double-precision floating-point value
E or e : Scientific notation in the form "[–]d.dddd **e** [sign]ddd" , where
d is a single decimal digit, dddd is one or more decimal digits, ddd is exactly three decimal digits, and sign is + or –.

Example:

```
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
```

GetCnvTagArrayIndex ()

Description:
When a **user-defined tag** as below is constructed to be an array with the [Read conversion] enabled, the [Read conversion] subroutine can get the corresponding array index to perform unit conversion.



Syntax:
GetCnvTagArrayIndex(*array_index*)

Argument	Description
array_index	The index will be saved into *array_index* (zero-based indexing).

Example:
// Create this subroutine in **Macro Function Library**.
sub unsigned short myfunction(unsigned short param)

short index

GetCnvTagArrayIndex(index)

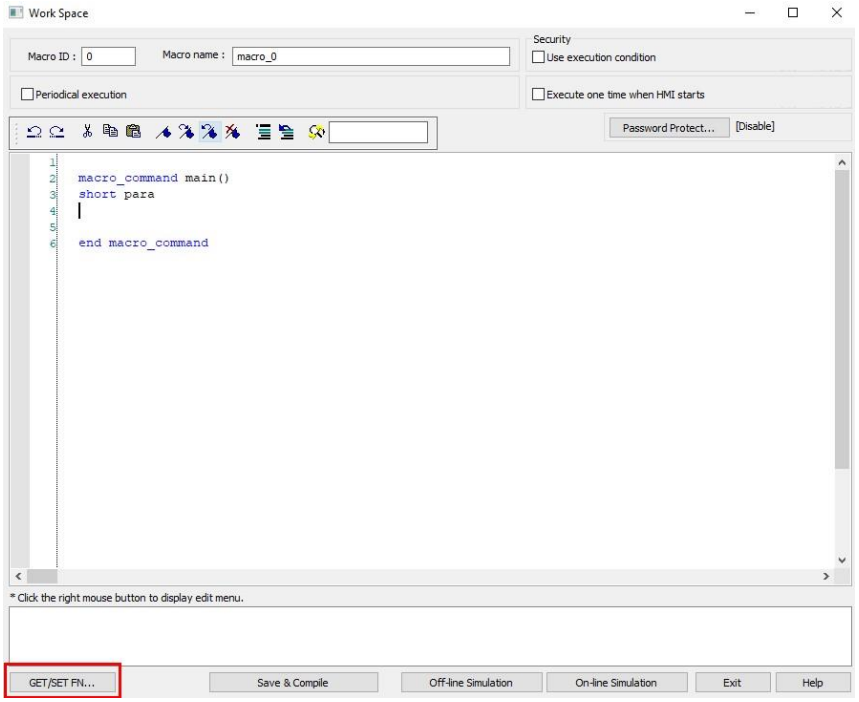
```
    if index==0 then
    param=param*0.5       // conversion for TAG[0]

    else if index==1 then
    param=param*1         // conversion for TAG[1]

    else if index==2 then
    param=param*2         // conversion for TAG[2]
```

```
        else if index==3 then
        param=param*3          // conversion for TAG[3]

        else if index==4 then
        param=param*4          // conversion for TAG[4]

        end if
return param

end sub
```

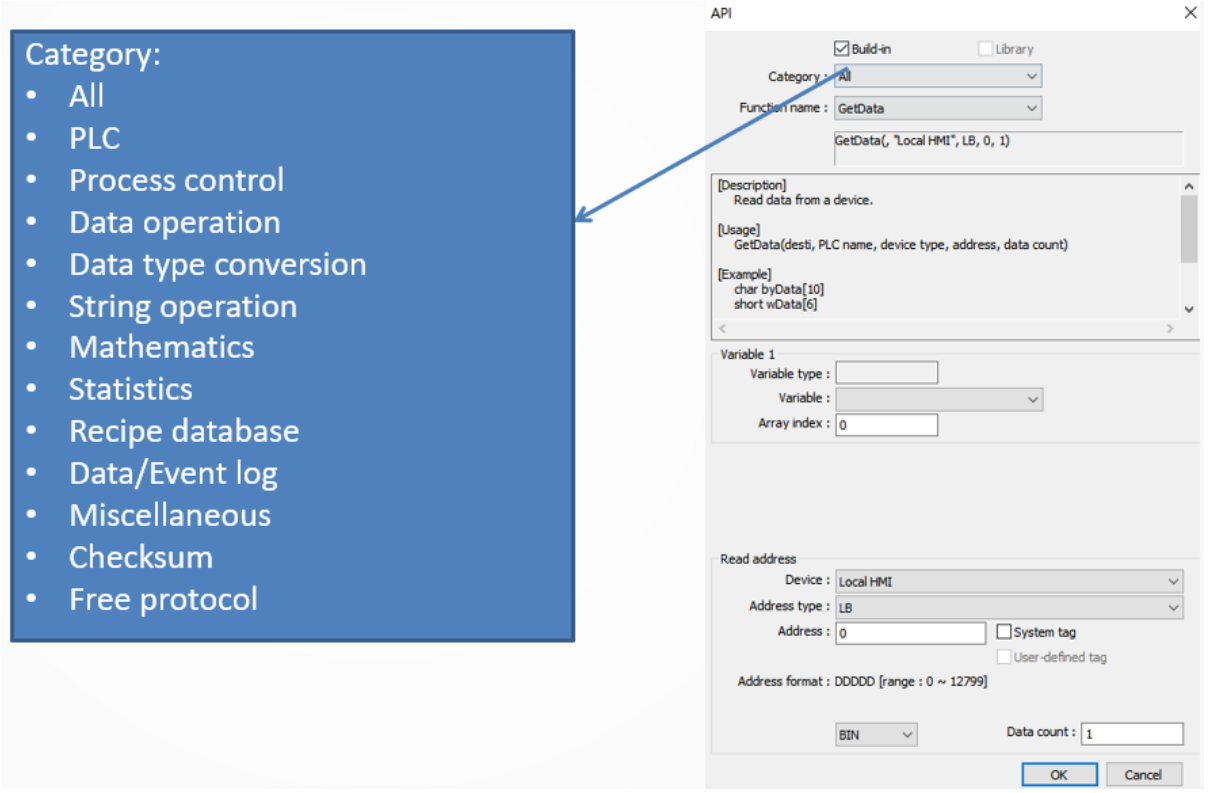
//The result will be:

Appendix A. How to Use the FN Dialog

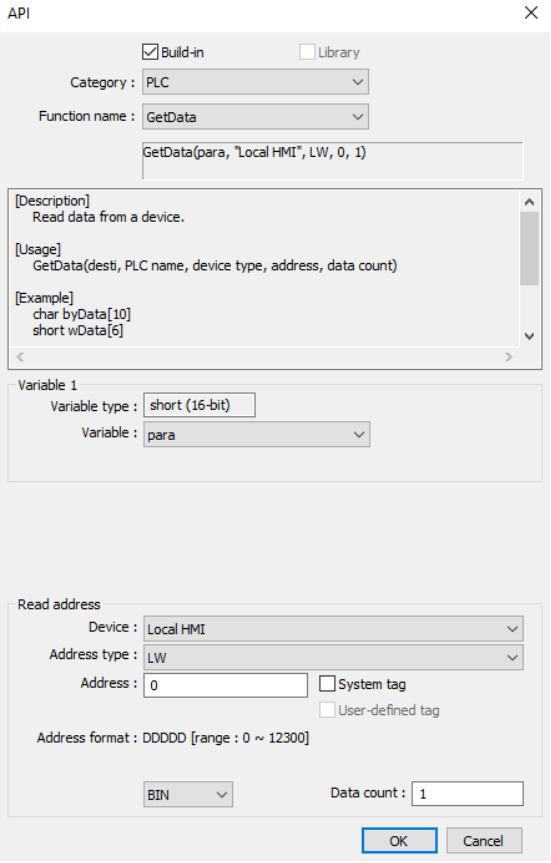
You can declare a function by opening the FN dialog, which provides you with a user interface.



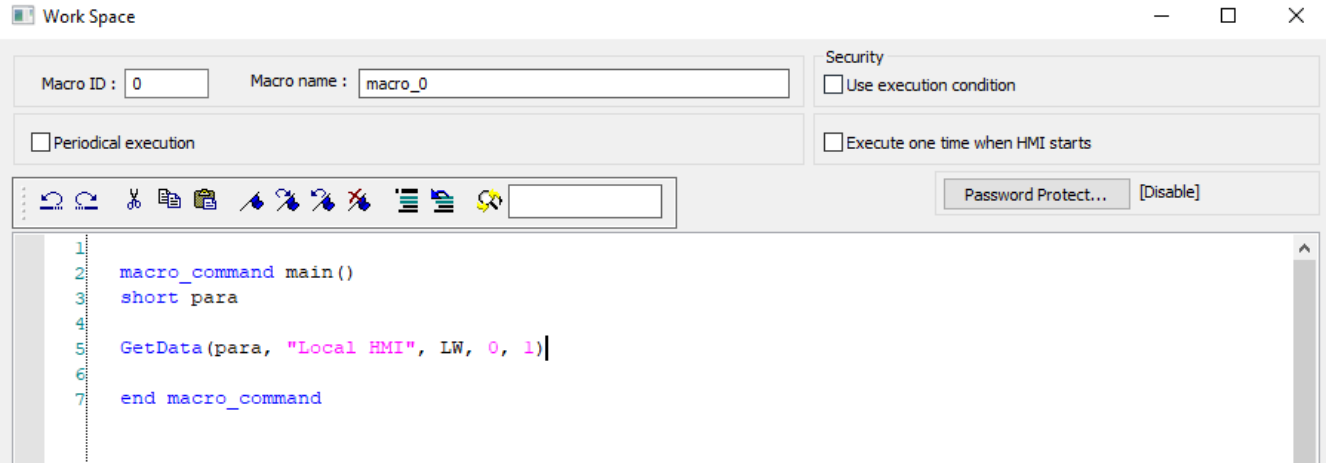
Select a category from the “Category” drop-down list and select a function.



Then fill out the arguments of the function.



Once clicking the OK button above, the function will be written into the macro editor.



Reference Link:
Weintek Labs website: http://www.weintek.com

Founded in 1996, WEINTEK LABS is a global-leading HMI manufacturer and is dedicated to the development, design, and manufacturing of practical HMI solutions. WEINTEK LAB's mission is to provide quality, customizable HMI-solutions that meet the needs of all industrial automation requirements while maintaining customer satisfaction by providing "on-demand" customer service. WEINTEK LABS brought their innovative technology to the United States in 2016, WEINTEK USA, INC., to provide quality and expedient solutions to the North American industrial market.

6219 NE 181s Street STE 120
Kenmore, WA 98028
425-488-1100